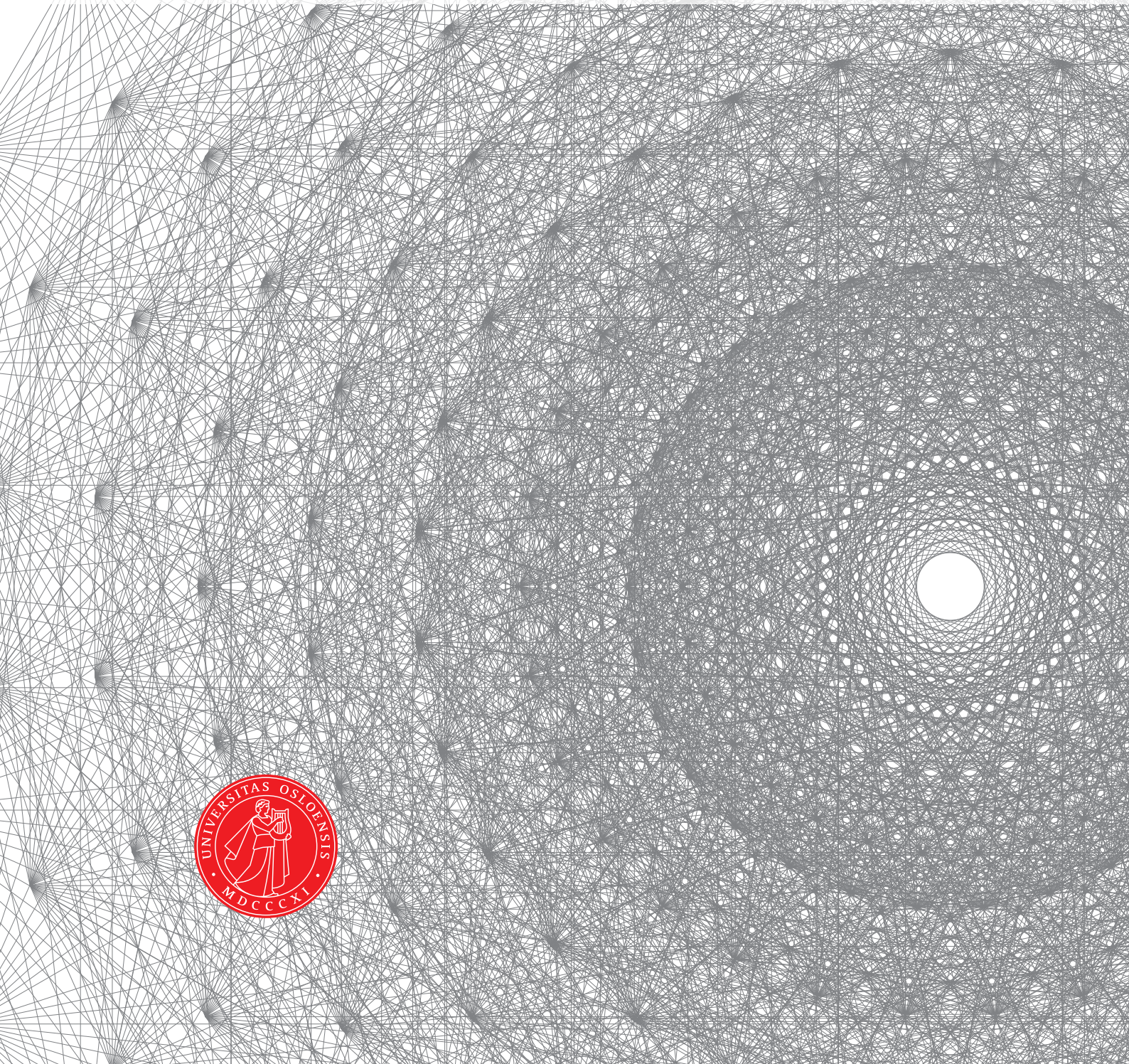
UiO : Department of Mathematics
University of Oslo

Finite-volume methods with dense neural networks

Approximating solutions of two-dimensional scalar conservation laws

Åsmund Danielsen Kvitvang

Master's Thesis, Spring 2021



This master's thesis is submitted under the master's program *Computational Science*, with program option *Applied Mathematics and Risk Analysis*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

Hyperbolic conservation laws are an important part in classical physics to be able to mathematically describe the actions of nature. To obtain approximate solutions of such problems, several numerical methods have been developed, most of which with both advantages and disadvantages in terms of accuracy, efficiency and implementation simplicity.

Inspired by modern computer science we will in this thesis propose numerical methods based on flux approximations obtained by using dense neural networks (DNNs). We will investigate the accuracy and efficiency by performing experiments with Burgers' equation. The main result of this thesis is a proposed numerical method for approximating solutions of two-dimensional nonlinear conservation laws. As there does not exist exact solution formulas of such two-dimensional problems, a possible approach is to use fine-resolution solvers in order to properly approximate the solutions. These solvers are extremely time consuming, and the hope is that the use of pre-trained DNN models will lead to a precise and efficient numerical method. We will also explore the possibility of using a physics-informed loss-function for approximating solutions of one-dimensional conservation laws, and further discuss how this may be applied to the two-dimensional methods.

The DNN based numerical methods tested in this thesis yielded promising results with respect to both accuracy and efficiency. Due to time limitations of this study we have restricted ourselves to only studying Burgers' equation with a narrow sample of parameters. Thus, some uncertainty follows with the results, and thereby uncertainty in the conclusions. However, there are strong indications that the proposed models are valuable, given the right set of parameters.

Acknowledgements

First of all, I would like to thank my supervisor Ulrik Skre Fjordholm for giving me such an interesting topic to investigate over the last year. It has by far been my most educational experience, and you have been both patient and supportive in order for the work to progress.

Secondly, thanks to all my friends at my study hall, B1001. We have had much fun together, and you have all been very helpful when I have needed someone to discuss academically related questions with.

Thanks to all my friends and family who have supported me throughout the years, both at the university and otherwise.

Finally, I would like to thank my partner, Linda T. Haugen. I am eternally grateful that I have you in my life. Thank you for all support you have given me throughout my years as a student.

Contents

CHAPTER 1

Introduction

Hyperbolic conservation laws play an important role in modern physics. At the beginning of the 20th century E. Noether proved a theorem stating that every differentiable symmetry of the action of a physical system has a corresponding conservation law [Noe18]. The perhaps most famous example is given by the first law of thermodynamics, stating that energy can neither be created nor destroyed in a system of constant mass, although it may be converted into different forms [Car60]. In other words, the energy is conserved over time, and may be mathematically described by hyperbolic conservation laws.

The focus in this thesis is set on approximating solutions of nonlinear conservation laws. Similar to other nonlinear equations, gradients tend to 'explode' which occurs as discontinuities in the solutions. This motivates for existence of weak solutions, solving the PDEs in terms of distributions. Such solutions lack uniqueness, leading to the creation of entropy conditions, which ensures physically sensible solutions. The entropy condition is given as the second law of thermodynamics [Car60].

The numerical methods presented in this thesis are developed by using dense neural networks to approximate the flux of scalar conservation laws. Neural networks, or more generally machine learning, has been in the minds of scientists for nearly 80 years – starting with a paper from 1943 by W. McCulloch and W. Pitts [MP43]. This started with a simple thought: is it possible to artificially simulate a brain? This has then developed into its own field of study for solving extremely complex models – calculations that would take a human being maybe decades, or even centuries, to solve by hand. In 2019 M. Raissi et al. introduced physics-informed neural network (PINN), which is a supervised learning framework for solving nonlinear PDEs [RPK19]. These networks are trained to solve supervised learning tasks with respect to given physical laws. Then, in 2020, A. D. Jagtap published an article proposing a conservative PINN method, considering nonlinear conservation laws with promising results. One of the methods proposed in this thesis is rather similar to the latter, as we introduce a physics-informed loss function for a DNN which is trained to approximate the flux of one-dimensional initial-value problems. This is then compared to results produced by a method using a regular MSE-loss function.

The main result of this thesis is the proposal of a DNN based numerical method for approximating solutions of two-dimensional initial-value problems. We will also present some experiments of this method, which we will compare to both the one-dimensional experiments and the two-dimensional Godunov scheme.

Numerical schemes for approximating solutions to nonlinear conservation laws have existed for several generations already, one of which is our basis for network training in one spatial dimension, namely the Godunov scheme [God59]. Two important questions arises when developing a new numerical method for solving any problem.

Question: Is the method accurate enough?

We need to make sure that the approximate solutions reflects the exact solutions of the initial-value problems. Thus, we must measure the accuracy in terms of some chosen error metric that compares the approximations to corresponding reference solutions. For the two-dimensional methods, the accuracy of the approximations will be compared to high resolution approximations obtained from a standard finite-volume method.

Question: Is the method efficient enough?

Efficiency is also an important note. If a developed method is both less efficient and less accurate compared to an already existing method, then it would not be a preferred method in any way. However, if a method proves itself extremely efficient with reduced accuracy, and vice versa, the method could be considered useful, all depending on the problem we work on. Our goal is therefore to study the accuracy of our models, as well as including a short note on the time complexity of the performed calculations.

Outline

Chapter 2 contains the theoretical background needed for this thesis. It is divided into three sections, namely PDE theory, numerical analysis and foundations on machine learning.

Chapter 3 contains derivations of numerical methods, together with experiments and results. It is built up by five sections. The first section gives a common baseline needed for all experiments. In the second section we develop and test a new numerical method for solving one-dimensional initial-value problems, using a dense neural network with mean squared error as loss function. The third section suggests a new physics-informed loss function for solving the exact same problems, with the hopes that this will increase stability of the numerical method by teaching the network some structural properties. The forth section contains the main results of this thesis. Here, we develop a neural network based method for approximating solutions of two-dimensional initial-value problems. We will train the neural models using a high-resolution spatial mesh, with the hope that this numerical method will approximate solutions of two-dimensional initial-value problems more accurately and efficiently than already existing methods. In the fifth and last section we mention potential sources of error in the conducted experiments.

Chapter 4 contains summary and conclusion of the thesis, as well as discussions on potential further work. We will also reflect on what we could have done differently to improve the value of this study.

Appendix A contains the Python code implemented for all the experiments. The resulting implementation is compactly written with package structure, and may be found on https://github.com/aasmunkv/riemannDNNsolver.

Notation

We will now go through the most essential notations and conventions needed throughout this thesis.

Derivatives

We will use a similar notation of derivatives to what L. C. Evans uses [Eva10]. Assume $u : \Omega \to \mathbb{R}$ where $\Omega \subset \mathbb{R}^n$. Then we use the following notations.

(i) The partial derivative of u with respect to x will in most cases be denoted as u_x or $\partial_x u$ (instead of $\frac{\partial u}{\partial x}$). Similarly, the double derivative of u with respect to x_i and x_j is denoted $u_{x_i x_j}$ or $\partial_{x_i x_j} u$, and so on.

(ii) We will denote the gradient of u as ∇u, which is defined as

$$\nabla u \coloneqq (u_{x_1}, u_{x_2}, \ldots, u_{x_n}).$$

(iii) We denote the divergence of u as $\nabla \cdot u$, defined by

$$\nabla \cdot u \coloneqq \sum_{i=1}^{n} \partial_{x_i} u.$$

Machine learning

Given a dense neural network built up by an input layer, L hidden layers and an output layer, the following notation will be used.

Symbol	Description
$\mathbf{X}$	input layer of network
$\hat{\mathbf{Y}}$	output prediction of network
$\mathbf{Y}$	target values of network, for comparison with $\hat{\mathbf{Y}}$
$\mathbb{X}$	dataset created for training, $\mathbb{X} = \begin{pmatrix} \mathbf{X} & \mathbf{Y} \end{pmatrix}$
n_l	number of neurons in layer l
$\mathbf{b}^l$	bias vector of layer l
$\mathbf{a}^l$	vector of neurons in layer l
a_i^l	neuron number i in layer number l
	further we have that $i \in [\![1, n_l]\!]$
$\mathbf{W}^l$	weight matrix between layer $l-1$ and l

Symbol	Description
w_{ij}^l	weight connecting jth neuron in layer $l-1$ to ith neuron in layer l
$\mathbf{W}$	tensor containing all weight matrices i.e. $\mathbf{W} \coloneqq \begin{pmatrix}\mathbf{W}^1 & \mathbf{W}^2 & \cdots & \mathbf{W}^{L+1}\end{pmatrix}$
$\mathbf{z}^l$	non-activated input signal of layer l i.e. $\mathbf{z}^l \coloneqq \mathbf{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l$
z_j^l	element j in $\mathbf{z}^l$
φ_l	activation function between layer $l-1$ and l i.e. $\mathbf{a}^l \coloneqq \varphi_l(\mathbf{z}^l)$
$C(\cdot)$	cost (or loss) function
$\mathcal{H}_l$	full transition between layer $l-1$ and l i.e. $\mathcal{H}_l(\mathbf{a}^{l-1}; \mathbf{W}^l, \mathbf{b}^l) \coloneqq \varphi_l(\mathbf{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l)$
$\mathcal{N}$	fully-connected and dense neural network i.e. $\mathcal{N}(\mathbf{X}) \coloneqq (\mathcal{H}_{L+1} \circ \mathcal{H}_L \circ \cdots \circ \mathcal{H}_1)(\mathbf{X})$
$\lessapprox$	used for description of neural structure, e.g. $4 \lessapprox 16 \lessapprox 64$ means that we have a dense neural network $\mathcal{N}$ with 3 hidden layers with 4, 16 and 64 neurons, respectively

Additional notations

Besides the notation mentioned above, the following table lists up other notations chosen for this thesis.

Symbol	Description
$\mathbb{N}$	natural numbers, not including zero
$\mathbb{Z}$	integers
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$\mathbb{N}_0$	natural numbers, including zero
$\mathbb{R}^n$	n-dimensional Euclidean space, $\mathbb{R} = \mathbb{R}^1$
$\mathbb{R}_+$	positive part of $\mathbb{R}$
Ω	subset of $\mathbb{R}^n$
$\partial\Omega$	boundary of Ω
$\bar{\Omega}$	closure of Ω
Ω_T	cross-product of $\Omega \subset \mathbb{R}^n$ and $[0, T) \subset \mathbb{R}_+$
Γ_T	boundary of Ω_T
$[\![a, b]\!]$	integer interval from a to b, defined as $[\![a, b]\!] \coloneqq [a, b] \cap \mathbb{Z}$
x, y	spatial parameters
t	temporal parameter
$\Delta x, \Delta y, \Delta t$	mesh step size with respect to x, y, t
T	temporal maximum
u	conservative quantity
f, g	flux function
F, G	numerical flux approximation of f, g
F^{God}	Godunov's flux function
$\gamma(t)$	shock wave

Symbol	Description
$s(t)$	shock speed, i.e. $s(t) = \gamma'(t)$
ψ	test function in C_c^1
(η, q)	entropy pair, made up of entropy function η and entropy flux q
$\circ$	composition operator, $(f \circ g)(x) = f(g(x))$
∇_x	gradient operator for spatial dimensions
$\partial_x, \partial_y, \partial_t$	partial derivative with respect to x, y, t
$C^1(\cdot)$	set containing continuously differentiable functions
$C_c^1(\cdot)$	set containing continuously differentiable functions with compact support
$L^\infty(\cdot)$	essentially bounded measurable functions

CHAPTER 2

Background and theory

In this chapter we include all necessary theory needed for this study. Section 2.1 consists of a purely analytical point of view of scalar conservation laws. This includes some examples and motivations for solutions, as well as explicit solution formulas for general one-dimensional scalar conservation laws. In Section 2.2 we will derive the general finite-volume scheme for one-dimensional initial-value problems, followed by an introduction to Godunov's scheme. In Section 2.3 we introduce the theory of dense neural networks, and the algorithms needed to train such models.

2.1 Scalar conservation laws

The importance of a conservation law lies in the simple and elegant way it describes physical phenomena of nature, where the total quantity of some substance is preserved over time. Thus, the quantity is conserved in the sense that it moves around in the spatial dimensions over time, and can not be created, nor destroyed. Let $u = u(x,t)$ denote the measure of a quantity which we want to calculate analytically or estimate numerically. The general *scalar multi-dimensional conservation law* is then given by

$$\partial_t u + \nabla_x \cdot \mathbf{f}(u) = 0, \tag{2.1}$$

where $\mathbf{f}(u) = (f_1(u), f_2(u), \dots, f_n(u))$ denotes the given flux functions and $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ is the spatial parameters. We will start this section by physically interpreting what a scalar conservation law is. Further, we will look into the behavior of characteristics for solutions of such problems, which yields an intuition of how to obtain physically sensible solution formulas. Moreover, we look into the lack of existence of classical solutions, and then move on to presenting weak solutions of both shock waves and rarefaction waves. Lastly, the entropy condition is introduced, which leads to a complete introduction to entropy satisfying solutions for one-dimensional scalar conservation laws.

2.1.1 Physical interpretation

Consider some quantity $u = u(x,t)$, given by $(x,t) \in \Omega \times \mathbb{R}_+$, where $\Omega \subset \mathbb{R}^n$. If we want to calculate the change over time of the total quantity u, that is

$$\frac{d}{dt} \int_\Omega u(x,t)\, \mathrm{d}x,$$

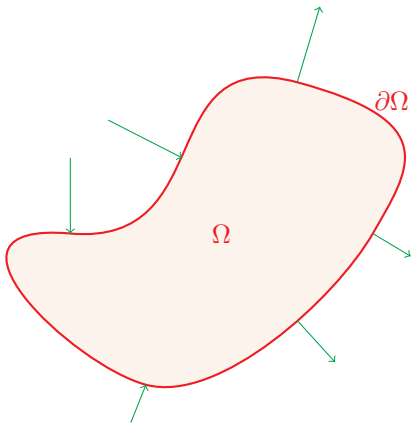


Figure 2.1: A two-dimensional illustration of flux, multiplied with the outward unit normal, on boundary of a domain Ω.

we need to consider two factors, namely the sources/sinks (the amount of quantity produced/destroyed) within the domain and the flux (the amount of quantity going in or out) on the boundary of the domain. This yields

$$\frac{d}{dt}\int_\Omega u(x,t)\,\mathrm{d}x = \int_\Omega s(x,t)\,\mathrm{d}x - \int_{\partial\Omega} \mathbf{f}(u(x,t))\cdot\nu\,\mathrm{d}S(x), \tag{2.2}$$

where $\mathbf{f} = \mathbf{f}(u)$ and $s = s(x,t)$ denotes the flux and source/sink, respectively, while ν is the outward unit normal of the domain and $\mathrm{d}S(x)$ is the spatial domain's surface measure. In Figure 2.1 we have an illustration of the flux, multiplied with the outward unit normal. Using the divergence theorem on the right term of (2.2), we obtain

$$\frac{d}{dt}\int_\Omega u(x,t)\,\mathrm{d}x + \int_\Omega \nabla_x\cdot\mathbf{f}(u)\,\mathrm{d}x = \int_\Omega s(x,t)\,\mathrm{d}x.$$

Stating that the energy is conserved over time within the spatial domain is the same as saying that there is neither sources nor sinks within the domain. Thus, we wind up with the integral form of the conservation law

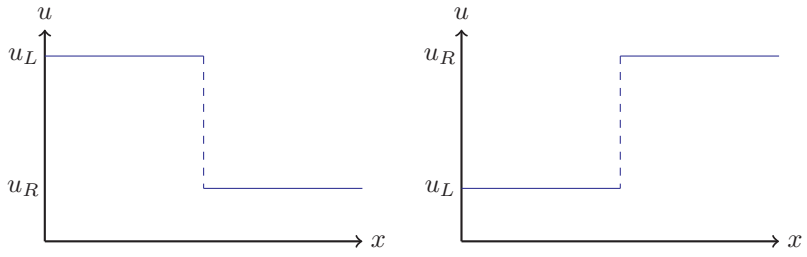
$$\int_\Omega \partial_t u(x,t) + \nabla_x\cdot\mathbf{f}(u)\,\mathrm{d}x = 0.$$

Since this equality holds for all subdomains of Ω we may choose an arbitrary small subdomain to obtain the differential form in (2.1).

2.1.2 Characteristics of Riemann problems with convex flux

Consider a subset $\Omega \subset \mathbb{R}$, and let $f = f(u)$ be a smooth and strictly convex function. Equation (2.1) rewritten as a general one-dimensional *initial-value problem* yields

$$\begin{cases} \partial_t u + \partial_x f(u) = 0, & (x,t)\in\Omega\times\mathbb{R}_+, \\ u(x,0) = u_0(x), & x\in\Omega, \end{cases} \tag{2.3}$$



(a) Shock propagation. (b) Rarefaction propagation.

Figure 2.2: Riemann initial data where (a) results in a shock propagating through space, while (b) results in a rarefaction propagation.

where $f : \mathbb{R} \to \mathbb{R}$ denotes the flux and $u : \Omega \times \mathbb{R}_+ \to \mathbb{R}$ is the unknown to be found. Equation (2.3), together with the initial data

$$u_0(x) = \begin{cases} u_L & \text{if } x < 0, \\ u_R & \text{if } x > 0, \end{cases} \tag{2.4}$$

illustrated in Figure 2.2, is commonly known as a *Riemann problem*. Since the flux is assumed to be a smooth function we may write (2.3) in the quasi-linear form as

$$\partial_t u + f'(u)\partial_x u = 0, \quad (x,t) \in \Omega \times \mathbb{R}_+, \tag{2.5}$$

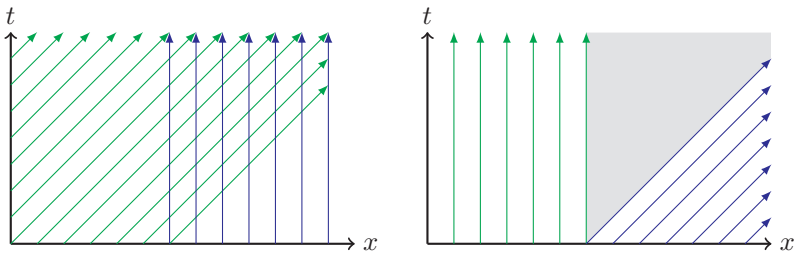
which yields a system where information (quantity) propagates through the spatial domain with speed $f'(u)$. Thus, the velocity field is dependent on the solution $u(x,t)$ itself. By applying the *method of characteristics* onto (2.3) with Burgers' flux function,

$$f(u(x,t)) = \frac{u(x,t)^2}{2}, \tag{2.6}$$

we obtain three different outcomes, depending on whether $u_L > u_R$, $u_L = u_R$ or $u_L < u_R$. The characteristics of the former and latter with $u_L = 1$, $u_R = 0$ and $u_L = 0$, $u_R = 1$ are illustrated in Figures 2.3a and 2.3b, respectively. Thus, we have intersecting characteristics in the case of $u_L > u_R$, while $u_L < u_R$ yields non-existence of solution on a subset of the domain. In the case of equality all characteristics will be parallel to one another, spanning the whole domain. The two illustrated cases generalizes to all cases of $u_L > u_R$ and $u_L < u_R$. This leads to intersection of characteristics, whenever the temporal parameter t satisfies the relations

$$\begin{aligned} u_L < u_R &\iff t < 0, \\ u_L > u_R &\iff t > 0. \end{aligned} \tag{2.7}$$

Summarized, the one-dimensional scalar conservation laws have two types of complications that may arise with respect to the characteristics, namely intersecting characteristics and non-spanning areas, both of which are illustrated in Figure 2.3. The case of intersection is problematic in the sense that every intersecting point in the domain implies a non-uniqueness of the solution $u(x,t)$. On the other hand, the non-spanning, gray area in Figure 2.3b is problematic since this implies that there is a subset $\omega \subset \Omega \times \mathbb{R}_+$ where a solution $u(x,t)$ does not exist.



(a) Shock propagation. (b) Rarefaction propagation.

Figure 2.3: Complications that may arise when working with conservation laws. Figure (a) shows characteristics of a shock wave propagating through space over time, yielding intersecting characteristics. Figure (b) shows characteristics of a rarefaction wave propagating through space over time, yielding an untouched sub-domain.

2.1.3 Weak solutions of Riemann problems

As we have seen, there are Riemann problems where solutions do not exist within a subset of the domain. In 1973, Peter D. Lax provided us with a proof showing that this is also the case for one-dimensional initial-value problems with smooth initial data [Lax73]. His approach is based on showing that the spatial derivative of the solution may 'explode' in finite time, i.e. the solution $u(x,t)$ contains a discontinuity. Many of the PDEs with discontinuous solutions are found in the nature in some sense, indicating that there must be other varieties of solutions. In this section we will introduce these solutions, known as *weak solutions*.

To obtain finite results from integration when working on subsets $\Omega \subset \mathbb{R}^n$ it sometimes makes sense to restrict functions to only attain non-zero values within the subset. This is needed for definition of weak solutions, thus we begin with a definition of *compactly supported functions* [MW99].

Definition 2.1.1 (Compact Support)**.** Consider a subset $\Omega \subset \mathbb{R}^n$. A function $f : \Omega \to \mathbb{R}$ is said to be *supported* in Ω if and only if $f(x) = 0$ for all $x \notin \Omega$. If Ω is closed and bounded it is called a *compact subset*. If this is the case, f is called *compactly supported*. The set of all continuously differentiable functions in $\mathbb{R}^n$ with compact support is denoted $C_c^1(\mathbb{R}^n)$.

Equation (2.3) is the differential form of the conservation law, and this form only holds if the solution u and the flux function f are continuously differentiable. A weak solution is a generalized solution of a PDE where the derivatives of the solution may not exist, formally defined as followed [LeV02].

Definition 2.1.2 (Weak Solution)**.** Let ψ be a continuously differentiable test function in $\mathbb{R} \times \mathbb{R}_+$, with compact support. Further, consider (2.3) as our PDE with initial data $u_0 \in L^\infty(\mathbb{R})$. Then, u is called a weak solution of the PDE if

$$\int_{\mathbb{R}_+} \int_{\mathbb{R}} u\psi_t + f(u)\psi_x \,\mathrm{d}x \,\mathrm{d}t + \int_{\mathbb{R}} u_0(x)\psi(x,0) \,\mathrm{d}x = 0, \tag{2.8}$$

holds for all test functions $\psi \in C_c^1(\mathbb{R} \times \mathbb{R}_+)$.

The weak solutions of a PDE do not need to be differentiable. They do not even need to be continuous, which implies discontinuities occurring as so called *shock waves*. These shock waves cannot be arbitrary in the x-t-plane and must satisfy the conditions given in the following lemma – a proof of the lemma is provided in [GR91].

Lemma 2.1.3 (Rankine–Hugoniot Condition)**.** *Let $\gamma \in C^1(\mathbb{R}_+)$, i.e. a continuously differentiable function in $\mathbb{R}_+$. Further, let $u \in L^\infty(\Omega \times \mathbb{R}_+)$ be of the form*

$$u(x,t) = \begin{cases} u^-(x,t) & \text{if } x < \gamma(t), \\ u^+(x,t) & \text{if } x > \gamma(t), \end{cases} \tag{2.9}$$

where both u^- and u^+ are continuously differentiable functions and γ denotes a shock. Figure 2.4 illustrates such a setup. Then u is a weak solution of (2.3) if and only if the following two properties are satisfied:

- *Both u^- and u^+ solve (2.3) in a classical sense.*
- *The shock speed $s(t) = \gamma'(t)$ satisfies the Rankine Hugoniot condition,*

$$s(t) = \frac{f(u^+(x,t)) - f(u^-(x,t))}{u^+(x,t) - u^-(x,t)}, \tag{2.10}$$

  at $x = \gamma(t)$.

The weak solution formula defined in (2.8) is a generalized form of the differential form of the one-dimensional scalar conservation law. Consequently, all classical solutions are weak solutions.

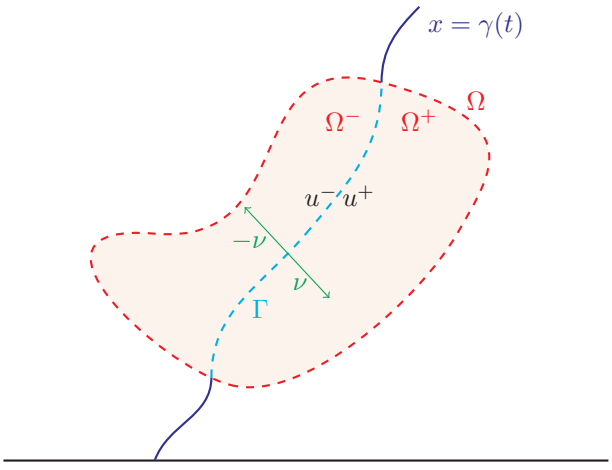


Figure 2.4: Illustration of sets and variables used in Lemma 2.1.3. Red dashed line bounds the open set Ω. Cyan dashed line is a characteristic splitting Ω into two subsets Ω^- and Ω^+. Green vectors are the outward unit normal.

In the rest of this section we will provide solution formulas for the Riemann problem

$$\begin{cases} \partial_t u + \partial_x f(u) = 0, & (x,t) \in \Omega \times \mathbb{R}_+, \\ u(x,0) = \begin{cases} u_L & \text{if } x < 0, \\ u_R & \text{if } x > 0. \end{cases} \end{cases} \tag{2.11}$$

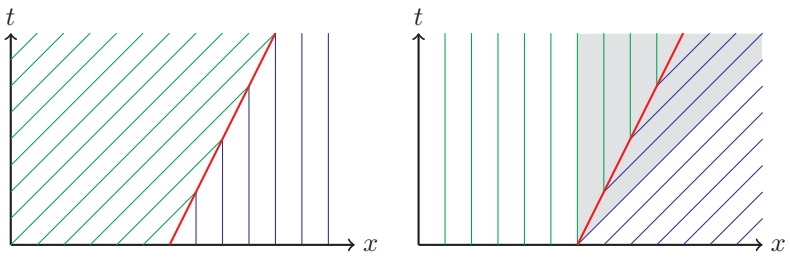
The first solution we will look into is a direct consequence of Lemma 2.1.3.

Corollary 2.1.4. *Let the shock speed $s(t) = \gamma'(t)$ solve the Rankine–Hugoniot condition, (2.10), at the shock $x = \gamma(t)$. Then*

$$u(x,t) = \begin{cases} u_L & \text{if } x < st, \\ u_R & \text{if } x > st, \end{cases} \tag{2.12}$$

is a weak solution of (2.11).

By comparing (2.12) to Lemma 2.1.3 we see that this in fact must be a solution to (2.11). By once again considering Burgers' flux function, (2.6), and applying (2.12), we obtain characteristics illustrated in Figure 2.5. Thus, $u_L > u_R$ yields characteristics emanating from the horizontal axis, whereas $u_L < u_R$ results in information created at the shock.



(a) Shock propagation. (b) Rarefaction propagation.

Figure 2.5: Characteristics of solutions to the scalar conservation law for (a) shock wave and (b) rarefaction wave, using Equation (2.12). Red lines represents discontinuities.

Usually we are looking for a unique and physically sensible solution to a PDE. Since the characteristics represents data flow, a reasonable criterion is to ensure that the information is traveling from the horizontal axis and outward in the domain, i.e. outward from the initial data. From Lemma 2.1.3, we know that every weak solution of (2.3) must satisfy the Rankine–Hugoniot condition, (2.10), so this is obviously an important criterion. However, this is not sufficient to ensure that information travels out from the initial data, see Figure 2.5b. Stating that the data must emanate from the initial bound is equivalent to saying that any shock wave $\gamma(t)$, having a speed of propagation $\gamma'(t) = s(t)$, satisfies

$$f'(u^-) > s(t) > f'(u^+), \tag{2.13}$$

where $f'(u^-)$ and $f'(u^+)$ are the speeds of propagation of any two characteristics on the left and right side of the shock $\gamma(t)$, respectively. This is known as the

Lax entropy condition. Since f is assumed to be strictly convex, we have that $f'(u_L) > f'(u_R)$ implies $u_L > u_R$. Thus, a direct consequence of (2.13) is that the solution proposed in (2.12) is a Lax entropy satisfying weak solution of (2.11) if and only if $u_L > u_R$. The natural continuation is now to construct a unique, Lax entropy satisfying solution in the cases where $u_L < u_R$, which is given by the following lemma [Eva10].

Lemma 2.1.5. *Let $u_L < u_R$ and consider two shock waves propagating with speeds $f'(u_L)$ and $f'(u_R)$. Then*

$$u(x,t) = \begin{cases} u_L & \text{if } x < f'(u_L)t, \\ (f')^{-1}(\frac{x}{t}) & \text{if } f'(u_L)t < x < f'(u_R)t, \\ u_R & \text{if } x > f'(u_R)t, \end{cases} \tag{2.14}$$

is a Lax entropy satisfying weak solution of (2.11).

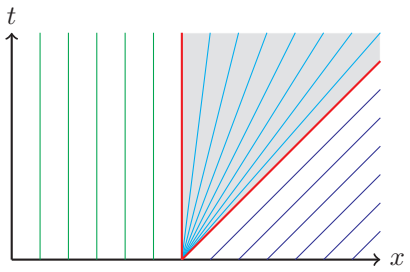
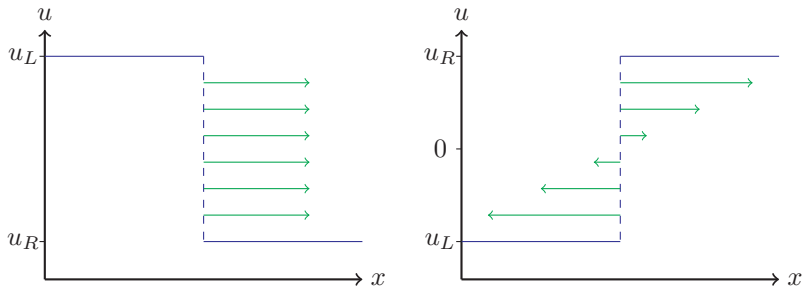


Figure 2.6: Characteristics of the entropy satisfying weak solution of rarefaction wave.

By considering (2.11) with Burgers' flux function, (2.6), and $u_L = 0$, $u_R = 1$, we obtain characteristics illustrated in Figure 2.6 when applying (2.14). Hence, all information travels outwards from the initial state, which seems physically sensible by argumentation above.

We have presented analytical solutions of one-dimensional Riemann problems for both shock waves and rarefaction waves, and observed how the characteristics



(a) Shock propagation. (b) Rarefaction propagation.

Figure 2.7: The velocity field (in green) of a shock wave (a) and rarefaction wave (b).

appear in both cases, see Figures 2.5a and 2.6. However, we have not yet looked into how these solutions propagate through space over time. In Figure 2.7 we have illustrated the velocity fields of two waves, namely a shock wave and a rarefaction wave. The green arrows illustrate the fields of velocity surrounding the discontinuities. In the case of shock waves, where $u_L > u_R$, we see that the wave propagates constantly along the whole discontinuity. On the other hand, the propagation of rarefaction waves, where $u_L < u_R$, depends on where you look, and will result in a smoothing of the discontinuity as time increases.

2.1.4 Entropy condition

So far, we have introduced two conditions, which has made us able to create a foundation for unique and physically sensible solutions of scalar conservation laws, namely Rankine–Hugoniot and Lax entropy. The latter is a local condition, thus, we need to generalize this to a globally rigorous condition. We will derive this for a general spatial dimension, by considering (2.1). By extending this to include a vanishing viscosity term, we obtain

$$\partial_t u^\varepsilon + \nabla_x \cdot \mathbf{f}(u^\varepsilon) = \varepsilon \Delta u^\varepsilon, \tag{2.15}$$

where Δ is the Laplacian operator and $u = \lim_{\varepsilon \to 0} u^\varepsilon$. Further, define the entropy function η and entropy flux $\mathbf{q}$ by

$$\begin{aligned} &\eta : \mathbb{R} \to \mathbb{R} \quad \text{strictly convex,} \\ &\mathbf{q}(u) = \int_0^u \mathbf{f}'(v)\eta'(v)\,\mathrm{d}v \in \mathbb{R}^n, \end{aligned}$$

where $\mathbf{f}'(u) \coloneqq (f_1'(u), f_2'(u), \ldots, f_n'(u))$ – the function pair $(\eta, \mathbf{q})$ is commonly known as an *entropy pair*. Hence, we have $\mathbf{q}'(u) = \mathbf{f}'(u)\eta'(u)$ and thereby the following calculation.

$$\begin{aligned} &\eta'(u^\varepsilon)\partial_t u^\varepsilon + \eta'(u^\varepsilon)\nabla_x \cdot \mathbf{f}(u^\varepsilon) = \eta'(u^\varepsilon)\varepsilon\Delta u^\varepsilon \\ &\Rightarrow \partial_t \eta(u^\varepsilon) + \eta'(u^\varepsilon)\mathbf{f}'(u^\varepsilon)\nabla_x \cdot u^\varepsilon = \varepsilon\Big(\Delta\eta(u^\varepsilon) \underbrace{-\eta''(u^\varepsilon)(\nabla u^\varepsilon)^2}_{\leq 0}\Big) \\ &\Rightarrow \partial_t \eta(u^\varepsilon) + \mathbf{q}'(u^\varepsilon)\nabla_x \cdot u^\varepsilon \leq \varepsilon\Delta\eta(u^\varepsilon) \\ &\Rightarrow \partial_t \eta(u^\varepsilon) + \nabla_x \cdot \mathbf{q}(u^\varepsilon) \leq \varepsilon\Delta\eta(u^\varepsilon) \end{aligned}$$

Hence, by passing $\varepsilon \to 0$ we get that (2.1) must satisfy

$$\partial_t \eta(u) + \nabla_x \cdot \mathbf{q}(u) \leq 0, \tag{2.16}$$

which is known as the *entropy condition* for a scalar multi-dimensional conservation law. For one-dimensional problems, this is reduced to

$$\partial_t \eta(u) + \partial_x q(u) \leq 0,$$

and in sense of distributions, this translates to

$$\int_{\mathbb{R}_+}\int_{\mathbb{R}} \eta(u)\psi_t + q(u)\psi_x \,\mathrm{d}x\,\mathrm{d}t + \int_{\mathbb{R}} \eta(u_0(x))\psi(x,0)\,\mathrm{d}x \geq 0,$$

for all test functions $\psi \in C_c^1(\mathbb{R} \times \mathbb{R}_+)$, satisfying $\psi \geq 0$. Weak solutions satisfying the entropy condition are commonly known as *entropy solutions* – we will state this as a definition.

Definition 2.1.6. A function $u = u(x,t)$ is an entropy solution of (2.1) if and only if

- u is a weak solution of (2.1),
- and u satisfies (2.16) for all entropy pairs $(\eta, \mathbf{q})$.

All entropy pairs $(\eta, \mathbf{q})$ satisfies the given results. However, there is one particular pair which is of greater importance than other pairs, namely the Kružkov entropy pair [Kru70], defined as

$$\eta(u;c) = |u-c|\,, \quad \mathbf{q}(u;c) = \operatorname{sign}(u-c)\big(\mathbf{f}(u) - \mathbf{f}(c)\big), \quad c \in \mathbb{R}.$$

The Kružkov entropy pair combined with (2.16) is called the *Kružkov entropy condition*. This leads to the following lemma – a proof is provided in a book written by H. Holden and N. H. Risebro [HR15].

Lemma 2.1.7. *A function $u = u(x,t)$ is an entropy solution of* (2.1) *if and only if it satisfies the Kružkov entropy condition.*

We have now laid the analytical foundation for the thesis, by introducing explicit solution formulas for solving initial-value problems given in (2.3). We have also looked into some fundamental properties yielding existence and uniqueness of physically sensible solutions. When we develop numerical methods and perform experiments in Chapter 3, these solution formulas will be useful to be able to calculate the accuracy of our methods.

2.2 Finite-volume schemes

During the last decades finite-volume methods have shown themselves to be accurate and useful for solving physical problems, such as conservation laws. In contrast to finite difference methods – which uses pointwise approximations – these volume methods uses averages, or volumes, to approximate the solutions. The comparison scheme we have chosen for this thesis is Godunov's scheme. There are plenty of numerical methods for approximating solutions to nonlinear conservation laws, besides Godunov's scheme, and there are advantages and disadvantages with all of them in terms of efficiency, accuracy and implementation simplicity. Examples of methods we could have considered as our baseline are schemes named *Roe* and *Engquist–Osher*. The Roe scheme is a drastic simplification of Godunov's scheme, which makes it less accurate but more efficient and simple to implement – in fact it does not converge towards the entropy solution. On the other hand, Engquist–Osher scheme is a more accurate and less efficient scheme than Roe's method, and could be considered in place of Godunov's scheme. In an article from 1984, B. V. Leer studied upwind-differencing first-order schemes and their abilities to approximate solutions of Burgers' equation [Lee84]. Godunov's scheme was then compared to Engquist–Osher scheme and Roe scheme, and he came to the conclusion that there is no reason to abandon Godunov's scheme in favor of neither Roe nor Engquist–Osher. There are many studies of Godunov's method, verifying its accuracy. In 1985 R. LeVeque published an article describing a generalization of Godunov's method for solving systems of conservation laws which can be applied for arbitrarily large time steps [LeV85]. He only considered generalizing the

standard Godunov method but the same linearization could be useful elsewhere. Further, in 1988 B. Einfeldt published an article describing a new Godunov-type Riemann-solver, based on Roe scheme, showing that it is sufficient to numerically approximate the largest and smallest signal velocities to obtain an efficient Riemann solver for gas dynamics [Ein88]. The successful application to the shock focusing problem shows the usefulness of the Riemann solver in a higher-order Godunov-type scheme.

In this section we will derive the general form of any one-dimensional finite-volume scheme for approximating scalar conservation laws. In the last part of this section we will also derive Godunov's scheme, which is the basis when we create the datasets for our one-dimensional DNN solvers in Sections 3.2 and 3.3. As for any numerical method, we start with the domain discretization.

2.2.1 Discretization

When we solve a PDE analytically we perform the calculations on the whole domain, i.e. infinitely amount of points within a subset of $\mathbb{R}^n$. However, when we want to obtain a numerical approximation, e.g. with Godunov's scheme, we need to discretize our domain. That is, the domain must be divided into discrete points with some step size in between each point. The spatial and temporal step sizes will be denoted Δx and Δt, respectively. The simplest choice of step size is to use uniformly distributed mesh grids, however, for the temporal domain we will impose a dynamically changing Δt for the sake of stability. We are therefore in need of the notation Δt^n describing the step size of one specific step.

Given a spatial domain $[x_L, x_R] \in \mathbb{R}$ we define our spatial mesh points to be

$$\begin{aligned} x_i &\coloneqq x_L + \left(i + \frac{1}{2}\right)\Delta x, \quad \forall i = [\![0, N_x]\!], \\ \Delta x &\coloneqq \frac{x_R - x_L}{N_x + 1}. \end{aligned}$$

Further, we also define the midpoints between these mesh points, which consequently are

$$x_{i-1/2} \coloneqq x_L + i\Delta x, \quad \forall i = [\![0, N_x + 1]\!].$$

Note that this includes boundary points outside the mesh as well. These midpoints make up $N_x + 1$ subdomains

$$C_i \coloneqq [x_{i-1/2}, x_{i+1/2}],$$

called *computational cells* – also known as *control volumes*. The averages, of which the finite volume methods are based, are calculated with respect to these computational cells. Lastly, the temporal mesh points, with dynamical changes in step size, are defined as

$$t^n \coloneqq t^{n-1} + \Delta t^n, \quad \forall n = 1, 2, \ldots,$$

where the initial temporal point is $t^0 = 0$.

2.2.2 Numerical scheme

We will now derive the numerical schemes for approximations of the scalar conservation laws. The basis of these schemes are the cell averages, defined by the approximation

$$u_i^n \approx \frac{1}{\Delta x}\int_{C_i} u(x,t^n)\,\mathrm{d}x, \quad \forall i \in [\![0, N_x]\!], n = 1, 2 \ldots.$$

For each time step one updates the cell averages of the unknown, starting by using the given initial function $u_0(x)$,

$$u_i^0 \approx \frac{1}{\Delta x}\int_{C_i} u_0(x)\,\mathrm{d}x, \quad \forall i \in [\![0, N_x]\!].$$

Assume now that we have calculated the cell averages u_i^n up to some time t^n, for all i's. To obtain u_i^{n+1}, we integrate (2.3) over domain $C_i \times [t^n, t^{n+1})$, which yields

$$\int_{t^n}^{t^{n+1}}\int_{C_i} u_t\,\mathrm{d}x\,\mathrm{d}t + \int_{t^n}^{t^{n+1}}\int_{C_i} f(u)_x\,\mathrm{d}x\,\mathrm{d}t = 0.$$

Further, by using the fundamental theorem of calculus, we obtain

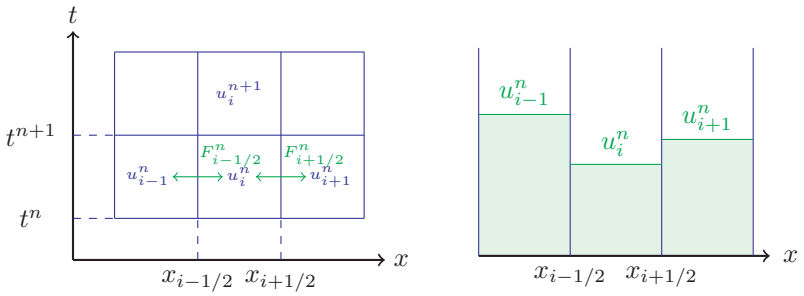
$$\int_{C_i} u(x,t^{n+1}) - u(x,t^n)\,\mathrm{d}x = -\int_{t^n}^{t^{n+1}} f(u(x_{i+1/2},t)) - f(u(x_{i-1/2},t))\,\mathrm{d}t.$$

Furthermore, by defining

$$\bar{F}_{i+1/2}^n \coloneqq \frac{1}{\Delta t^{n+1}}\int_{t^n}^{t^{n+1}} f(u(x_{i+1/2},t))\,\mathrm{d}t, \tag{2.17}$$

and dividing both sides by Δx, we wind up getting

$$u_i^{n+1} = u_i^n - \frac{\Delta t^{n+1}}{\Delta x}\left(\bar{F}_{i+1/2}^n - \bar{F}_{i-1/2}^n\right). \tag{2.18}$$



(a) Mesh grid. (b) Cell average.

Figure 2.8: Mesh grid and cell averages. (a) shows a typical example of a finite volume grid with cell averages and fluxes. (b) illustrates interfaces consisting of Riemann problems.

This formula is not explicit as $\bar{F}^n_{i+1/2}$ requires knowledge of the exact solution. The numerical approximation of this flux component is the useful and clever part in a good finite-volume scheme, one of which is *Godunov's scheme*.

Figure 2.8 may give an intuition on how a finite-volume scheme works. Figure 2.8a shows the cell grid, where each cell contains a cell average, while the green arrows represent the fluxes traveling through each cell interface in spatial direction. Figure 2.8b illustrates the calculated cell averages for three spatial cells, within one temporal step. The interfaces between these averages is then what defines a Riemann problem.

2.2.3 Godunov's scheme

The Russian mathematician S. K. Godunov wrote a paper in 1959 [God59] introducing a brilliant scheme for approximating the numerical flux in (2.18). What we want, is to approximate (2.17) at each cell interface $x_{i+1/2}$, and since the cell averages u^n_i are constant within each cell C_i, see Figure 2.8b, Godunov observed that this makes up a Riemann problem at each of the cell interfaces,

$$\begin{cases} u_t + f(u)_x = 0, \\ u(x, t^n) = \begin{cases} u^n_i & \text{if } x < x_{i+1/2}, \\ u^n_{i+1} & \text{if } x > x_{i+1/2}. \end{cases} \end{cases} \tag{2.19}$$

The solutions of such problems consist of shock waves, rarefactions and compound waves, and can thus be solved at every time level explicitly, in terms of waves arising from each interface (from t^n to t^{n+1}). Further, intersection of waves may occur, which motivates for a condition preventing the waves to interact with the spatial cell boundaries before reaching the temporal boundaries, see Figure 2.8a. As mentioned when discretizing the domain in Section 2.2.1, we will consider dynamically changing temporal step sizes, the reason being these potential collisions. Each of the waves has a finite speed of propagation and this bound is given by $\max_i |f'(u^n_i)|$, i.e. the maximum change of flux within one time step. The temporal step sizes Δt^n will therefore be bounded by

$$\Delta t^n \leq C_{CFL} \frac{\Delta x}{\max_i |f'(u^n_i)|}, \tag{2.20}$$

where $C_{CFL} \in (0, 1)$ is called the *CFL coefficient*. Equation (2.20) is commonly known as the *CFL condition* and is used to ensure stability by ensuring no collision of characteristics – the name has its origin from an article by Courant, Friedrichs and Lewy, republished in 1967 [CFL67]. While deriving the numerical scheme, we will now assume that this CFL condition is satisfied. Moreover, we have that each solution is self-similar, which means that the solution $u(x, t; n, i)$ to the Riemann problem in (2.19) may be written as a function $v(\xi)$ of a single variable $\xi = \frac{x - x_{i+1/2}}{t - t^n}$,

$$u(x, t; n, i) = v\left(\frac{x - x_{i+1/2}}{t - t^n}\right).$$

By the self-similarity property we obtain a constant solution whenever ξ is constant. This is due to the fact that this implies that $v(\xi)$ is constant. For the sake of argument we will assume that $\xi = 0$. This corresponds to the curve

$x = x_{i+1/2}$, for all $t > t^n$. Then, the flux across the cell interface is given by $f(u(x_{i+1/2}, t; n, i)) = f(v(0))$. We now have two cases, namely that v is continuous in $\xi = 0$, or contrary discontinuous. If continuity holds it is easy to see that

$$\lim_{\xi \to 0_+} f(v(\xi)) = \lim_{\xi \to 0_-} f(v(\xi)). \tag{2.21}$$

On the other hand, if discontinuity holds, we have discontinuity along the line $x = x_{i+1/2}$, for all $t > t^n$, thus a stationary shock is positioned at the cell interface. Since this discontinuity must satisfy the Rankine–Hugoniot condition in (2.10), due to the conditions of shock waves, we have

$$f(v(0_+)) - f(v(0_-)) = 0 \cdot (v(0_+) - v(0_-)),$$

and (2.21) is true under discontinuity as well. This leads to the definition of an edge-centered flux,

$$F^n_{i+1/2} \coloneqq f(v(0_+)) = f(v(0_-)), \tag{2.22}$$

and the approximate flux, (2.17), is constant in time and may be calculated by (2.22). Substituting this flux into (2.18) yields our final finite-volume scheme,

$$u_i^{n+1} = u_i^n - \frac{\Delta t^{n+1}}{\Delta x}\Big(F^n_{i+1/2} - F^n_{i-1/2}\Big). \tag{2.23}$$

The formula for the numerical flux in (2.23) can be computed explicitly, and we will state this as a theorem.

Theorem 2.2.1 (Godunov flux)**.** *Let $u = v(\xi)$ be the solution to the scalar Riemann problem in* (2.19) *and consider the numerical scheme in* (2.23) *with the flux in* (2.22)*. Then, the numerical flux $F^n_{i+1/2} = F\big(u_i^n, u_{i+1}^n\big)$ is given by*

$$F^n_{i+1/2} = \begin{cases} \min\limits_{u_i^n \le \theta \le u_{i+1}^n} f(\theta) & \text{if } u_i^n \le u_{i+1}^n, \\ \max\limits_{u_{i+1}^n \le \theta \le u_i^n} f(\theta) & \text{if } u_i^n > u_{i+1}^n. \end{cases} \tag{2.24}$$

This is referred to as the Godunov flux.

To be able to prove Theorem 2.2.1, we first need to define what convex and concave *envelopes* are.

Definition 2.2.2 (Envelope)**.** The *lower convex envelope* of a function f over an interval $[a, b]$ is defined as

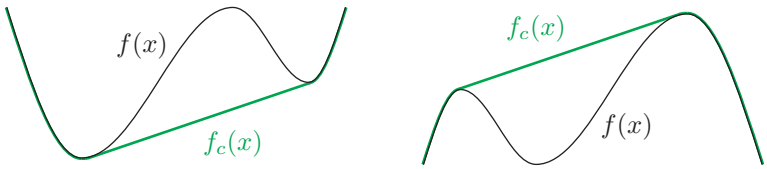
$$f_c(x) \coloneqq \sup\big\{\, g(x) \;\big|\; g(y) \text{ is convex and } g(y) \le f(y),\ \forall y \in [a, b] \,\big\},$$

see Figure 2.9a. Similarly, we define an *upper concave envelope* as the infimum of the set of all concave functions $g : \mathbb{R} \to \mathbb{R}$ where $g(y) \ge f(y)$ for all $y \in [a, b]$, see Figure 2.9b.

The numerical flux given in (2.24) is also valid for non-convex flux functions. As we want to end up with (2.24), and since

$$\min_y f_c(y) = \min_y f(y), \qquad \max_y f_c(y) = \max_y f(y),$$

it is sufficient to consider the convex envelope of f. The following proof is based on a proof presented in 1991, in a book by E. Godlewski and P. A. Raviart [GR91].



(a) Lower convex envelope. (b) Upper concave envelope.

Figure 2.9: Illustrations of a lower convex envelope (a) and an upper concave envelope (b).

The proof is split into two cases where the minimum case and maximum case of (2.24) are proven separately, using convex and concave envelopes, respectively. The book also proves the formula for non-convex flux functions, which is done by subdividing $[u_L, u_R]$ into alternating sub-intervals consisting of rarefactions and shocks.

Proof of Theorem 2.2.1. Let $u(x,t;n,i) = v(\xi)$ solve (2.19). Further, assume that $u_i^n < u_{i+1}^n$ and consider a convex envelope f_c of the flux function f over the interval $[u_i^n, u_{i+1}^n]$. Now we have three possible scenarios for f_c. It is either increasing or decreasing in all of $[u_i^n, u_{i+1}^n]$, or it is vanishing in some point in $[u_i^n, u_{i+1}^n]$. Assume that $f_c' > 0$. Then, the whole solution of the Riemann problem will go to the right in the x-t-plane, and the solution in $x = x_{i+1/2}$ will be the leftmost value of the interval, namely u_i^n. Now, assume that $f_c' < 0$. Then, the solution moves to the left, and the rightmost value of the interval is the solution in $x = x_{i+1/2}$, namely u_{i+1}^n. Thus, these two cases yields

$$v(0) = u(x_{i+1/2}, t; n, i) = \begin{cases} u_i^n & \text{if } f_c' > 0, \\ u_{i+1}^n & \text{if } f_c' < 0, \end{cases} \tag{2.25}$$

which gives us

$$F_{i+1/2}^n = f(v(0)) = \min_{\theta \in [u_i^n, u_{i+1}^n]} f(\theta). \tag{2.26}$$

Assume now that f_c' vanishes in some point $u_*^n \in [u_i^n, u_{i+1}^n]$. Then,

$$f(v(0)) = f(u_*^n) = \min_{\theta \in [u_i^n, u_{i+1}^n]} f_c(\theta) = \min_{\theta \in [u_i^n, u_{i+1}^n]} f(\theta),$$

and (2.26) is true in this case as well. The last equality is a direct consequence of Definition 2.2.2.

It remains to prove (2.24) for $u_i^n > u_{i+1}^n$. In this case we consider a concave envelope f_c of f over $[u_{i+1}^n, u_i^n]$. As above, we now have three cases. If $f_c' > 0$ in the entire interval, the solution will go to the left, and the solution is the rightmost value, namely u_i^n. If, on the other hand, $f_c' < 0$ in the entire interval, the solution of the Riemann problem moves to the right and the solution is u_{i+1}^n. This yields (2.25) once again, however, this corresponds to taking the maximum of the flux function over the domain, which gives us

$$F_{i+1/2}^n = f(v(0)) = \max_{\theta \in [u_{i+1}^n, u_i^n]} f(\theta). \tag{2.27}$$

If we now assume, as above, that f_c' vanishes in some point $u_*^n \in [u_{i+1}^n, u_i^n]$, we obtain

$$f(v(0)) = f(u_*^n) = \max_{\theta \in [u_{i+1}^n, u_i^n]} f_c(\theta) = \max_{\theta \in [u_{i+1}^n, u_i^n]} f(\theta),$$

and the last equality here is also a direct consequence of Definition 2.2.2. This completes the proof. ■

We are now equipped with the numerical baseline for the one-dimensional initial-value problems considered in Chapter 3. The derived Godunov flux is what we will use when creating the dataset for training the DNN model used in our one-dimensional DNN based numerical methods. Further, we will use a two-dimensional Godunov scheme for comparison of the results obtained by our proposed two-dimensional DNN based method.

2.3 Neural networks

Since the dawn of computer technology there has been several theoretical studies regarding neural networks. These are inspired by the biological point of view where humans has been eager to both understand the brain as well as to simulate it. The first crucial step in the direction of developing trainable neural networks came in 1943, when W. McCulloch and W. Pitts wrote a paper modelling a rather simple neural system using electrical circuits [MP43]. Since then, more and more complex neural models have been developed, side by side with the increase of computational power. This has brought us to modern *dense neural networks* (DNN), which is a type of *artificial neural networks* (ANN). In this section we will introduce this topic by first presenting the terminology. Then we will move on to more technical details, by deriving the backpropagation algorithm, and then look at potential problems arising during training in terms of bias-variance trade-off. Further, we will present the choice of activation function for this thesis, and lastly introduce an important theorem within the studies of ANNs, namely the universal approximation theorem.

2.3.1 Terminology

A DNN consists of a set of *neurons* organized into *layers*, yielding a directed graph structure, which serves as a computational system for approximating functions. Figure 2.10 illustrates a general multilayered DNN with two input variables, X_1 and X_2, and one output variable, $\hat{Y}$ – in general, the number of nodes in the input layer and output layer may be arbitrary, but for simplicity we have restricted the figure to two in and one out as this is essential for the first developed numerical method. In between the input layer and output layer there are L hidden layers, where the figure shows the first and last hidden layers with n_1 and n_L neurons, respectively. The yellow nodes illustrate the bias vectors, which work as shifting parameters for our output, for better results. Figure 2.11 illustrates the actions happening in between each pair of layers in the neural network. The output of the previous layer is multiplied with a set of weights and then added, together with a bias. Lastly, the result is then sent through an *activation function* φ_l. The incoming and outgoing values of a layer are often referred to as incoming and outgoing *signals*, respectively. We will now turn the attention towards a more mathematical point of view.

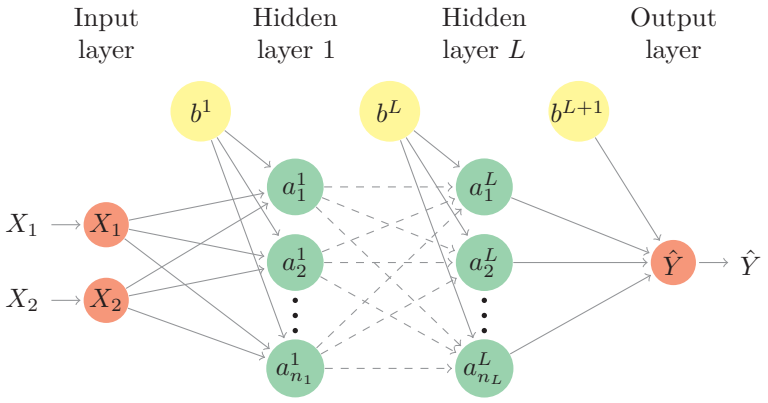


Figure 2.10: Illustration of a general DNN with $n_0 = 2$ input variables, $L \geq 2$ hidden layers and $n_{L+1} = 1$ output. The arrows represent multiplication with weights, summation and activation.

The matrix of weights that connect the neurons in layer $l-1$ with the neurons in layer l will be denoted $\mathbf{W}^l$. If there are n_{l-1} neurons in layer $l-1$ and n_l neurons in layer l this is written out as

$$\mathbf{W}^l = \begin{pmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1n_{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l1}^l & w_{n_l2}^l & \cdots & w_{n_ln_{l-1}}^l \end{pmatrix},$$

where w_{ij}^l is the weight connecting the jth neuron in layer $l-1$ to the ith neuron in layer l. Further, the collection of all weight matrices in the DNN will be denoted $\mathbf{W}$, and is written out as

$$\mathbf{W} = \begin{pmatrix} \mathbf{W}^1 & \mathbf{W}^2 & \cdots & \mathbf{W}^{L+1} \end{pmatrix}.$$

The total signal going out of the neurons in layer l will be denoted $\mathbf{a}^l$ and the total signal coming into layer l is a linear combination of $\mathbf{a}^{l-1}$ and $\mathbf{W}^l$ with added bias, and will be denoted $\mathbf{z}^l$. This is defined as

$$\mathbf{z}^l \coloneqq \mathbf{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l,$$

where $\mathbf{b}^l$ denotes the vector of biases for layer l. The necessity of the bias lies in the fact that it enables the DNN to map zero onto a non-zero output, and vice versa. To ensure nonlinearity of the network so that it may be trained to produce non-trivial outputs we need an *activation function*, denoted $\varphi_l(\cdot)$. Consequently, the outgoing signal in each layer is

$$\mathbf{a}^l \coloneqq \varphi_l(\mathbf{z}^l).$$

Iteration through a given DNN will be called a *forward pass*, with pseudocode shown in Algorithm 1. A forward pass consists of a series of function compositions, with added bias in every iteration. By studying Figure 2.10,

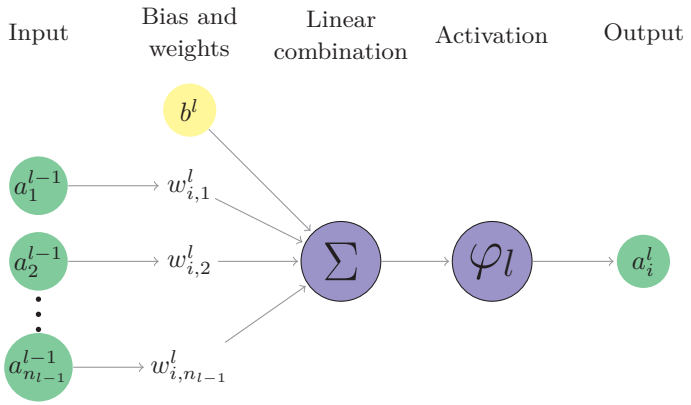


Figure 2.11: Illustration of the actions performed between two arbitrary layers of a DNN. The output of previous layer is multiplied with weights, summed up (including a bias) and lastly sent through an activation function.

Algorithm 1 Feed-forward

Initialize weights and dimensions of network.
$\mathbf{z}^1 = \mathbf{W}^1\mathbf{X} + \mathbf{b}^1$
$\mathbf{a}^1 = \varphi(\mathbf{z}^1)$
for $i = 2, \ldots, L$ **do**
  $\mathbf{z}^i = \mathbf{W}^i\mathbf{a}^{i-1} + \mathbf{b}^i$
  $\mathbf{a}^i = \varphi_i(\mathbf{z}^i)$
$\hat{\mathbf{Y}} = \mathbf{W}^{L+1}\mathbf{a}^L + \mathbf{b}^{L+1}$

together with Figure 2.11, we can therefore explicitly write out the computation, which gives us the following.

$$\begin{aligned}
\hat{\mathbf{Y}} &= \varphi_{L+1}(\mathbf{W}^{L+1}\mathbf{a}^L + \mathbf{b}^{L+1}) \\
&= \varphi_{L+1}(\mathbf{W}^{L+1}\varphi_L(\mathbf{z}^L) + \mathbf{b}^{L+1}) \\
&= \varphi_{L+1}(\mathbf{W}^{L+1}\varphi_L(\mathbf{W}^L\mathbf{a}^{L-1} + \mathbf{b}^L) + \mathbf{b}^{L+1}) \\
&\vdots \\
&= \varphi_{L+1}(\mathbf{W}^{L+1}\varphi_L(\mathbf{W}^L\varphi_{L-1}(\cdots\varphi_1(\mathbf{W}^1\mathbf{X} + \mathbf{b}^1)\cdots) + \mathbf{b}^L) + \mathbf{b}^{L+1})
\end{aligned}$$

By defining

$$\mathcal{H}_l(\mathbf{a}^{l-1}; \mathbf{W}^l, \mathbf{b}^l) := \varphi_l(\mathbf{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l),$$

as the transition between two arbitrary layers $\mathbf{a}^{l-1}$ and $\mathbf{a}^l$ in a DNN, see Figure 2.11, this calculation may be written in terms of composition as $\hat{\mathbf{Y}} = \mathcal{N}(\mathbf{X})$, where

$$\mathcal{N}(\mathbf{X}) := (\mathcal{H}_{L+1} \circ \mathcal{H}_L \circ \cdots \circ \mathcal{H}_1)(\mathbf{X}).$$

Lastly, we end by mentioning that the input variables $\mathbf{X}$ may be referred to as layer number zero and may therefore be denoted $\mathbf{a}^0$. Similarly, the output

layer $\hat{\mathbf{Y}}$ may be denoted $\mathbf{a}^{L+1}$ in a DNN consisting of L hidden layers. The activation function φ_{L+1} between the last hidden layer and the output layer is often chosen to be the identity mapping – this is what we have chosen in the developed numerical methods introduced in Chapter 3.

2.3.2 Backward propagation

When initializing a DNN, the weights are randomly chosen, and therefore not fitted for our problem. Thus, a training session is required, for the DNN to become a useful model. The procedures for such training are called *backpropagation algorithms* – hereby abbreviated BPA. We will assume we are working with a network consisting of N output values. As we will consider a so called supervised learning model, we have a training set consisting of input values to send through the network, as well as target values to compare the output with. The BPA considers the output variable $\hat{\mathbf{Y}} = (\hat{Y}_1, \dots, \hat{Y}_N)$ and compares it to the known target/solution $\mathbf{Y} = (Y_1, \dots, Y_N)$. We then evaluate the error between the target and the output, and propagate backward in the DNN system to adjust the weights and biases to be more accurate – the reason for backpropagation is that we only have information forward in the network. The function used to compute the error is called a *cost function* – also known as a loss function – and the perhaps simplest choice is the *mean squared error*, hereby abbreviated MSE,

$$C(\hat{\mathbf{Y}}, \mathbf{Y}; \mathbf{W}, \mathbf{b}) \coloneqq \frac{1}{N} \sum_{k=1}^{N} \Big(\hat{Y}_k - Y_k \Big)^2, \tag{2.28}$$

which is the one used in our first line of experiments in Section 3.2. The goal is now to find the weights and biases that minimize (2.28), hence we need to find the gradient of C with respect to $\mathbf{W}^l$ and $\mathbf{b}^l$ of each layer l. We will do this component-wise by considering

$$z_j^l \coloneqq \sum_{i=1}^{n_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l, \qquad a_i^l \coloneqq \varphi_l\big(z_i^l\big).$$

Using the chain rule and the fact that $\hat{Y}_k = a_k^{L+1}$, we have

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial a_k^{L+1}} \frac{\partial a_k^{L+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ij}^l},$$

where the first term on the right-hand side may be calculated directly from (2.28), while the two latter terms may be written out as

$$\frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ij}^l} = \varphi_l'\big(z_j^l\big) a_i^{l-1}.$$

This leaves us with the second term which is obtained by the following calculation, where we denote an arbitrary index in layer r as $q_r \in [\![1, n_r]\!]$, with $q_l = j$ and $q_{L+1} = k$.

$$\frac{\partial a_k^{L+1}}{\partial a_j^l} = \frac{\partial a_k^{L+1}}{\partial a_{q_L}^L} \frac{\partial a_{q_L}^L}{\partial a_{q_{L-1}}^{L-1}} \cdots \frac{\partial a_{q_{l+1}}^{l+1}}{\partial a_j^l}$$

Algorithm 2 Backpropagation

Initialize weights and biases of the DNN.
for $i = 1, \dots, M$ **do**
  Run feed-forward Algorithm 1 on input.
  Compute the error term δ^{L+1}.
  for $l = L, \dots, 1$ **do**
    Compute δ^l using (2.29).
    Update weights and biases by computing

$$\mathbf{W}^l_{\text{next}} \leftarrow \mathbf{W}^l_{\text{prev}} - \kappa \delta^l \mathbf{a}^{l-1},$$
$$\mathbf{b}^l_{\text{next}} \leftarrow \mathbf{b}^l_{\text{prev}} - \kappa \delta^l,$$

    where κ is the learning rate.

$$= \left(\frac{\partial a_k^{L+1}}{\partial z_k^{L+1}} \frac{\partial z_k^{L+1}}{\partial a_{q_L}^L}\right)\left(\frac{\partial a_{q_L}^L}{\partial z_{q_L}^L} \frac{\partial z_{q_L}^L}{\partial a_{q_{L-1}}^{L-1}}\right) \cdots \left(\frac{\partial a_{q_{l+1}}^{l+1}}{\partial z_{q_{l+1}}^{l+1}} \frac{\partial z_{q_{l+1}}^{l+1}}{\partial a_j^l}\right)$$
$$= \prod_{r=l+1}^{L+1} \frac{\partial a_{q_r}^r}{\partial z_{q_r}^r} \frac{\partial z_{q_r}^r}{\partial a_{q_{r-1}}^{r-1}}$$
$$= \prod_{r=l+1}^{L+1} \varphi_r'(z_{q_r}^r) w_{q_{r-1},q_r}^r$$

The same calculations holds for the gradient with respect to the bias. Further, by defining

$$\delta_j^l := \frac{\partial C}{\partial a_j^l} \varphi_l'(z_j^l), \tag{2.29}$$

we may write out the change of the cost function with respect to a given weight and bias as

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_j^l a_i^{l-1}, \qquad \frac{\partial C}{\partial b_j^l} = \delta_j^l, \tag{2.30}$$

respectively. The defined expression for δ_j^l is an important measure of error. The first term on the right-hand side of (2.29) is a measure on how fast the cost function changes with respect to jth output signal, while the second term measures how fast the activation function changes with respect to the input signal z_j^l. As we have seen in the derivation of (2.30), we have explicit formulas for computing the gradients with respect to the weights and biases in each layer in the DNN, and are thus equipped for BPA, see Algorithm 2. The learning rate κ mentioned in the algorithm is a measure on how great an adjustment should be for each training epoch. Further, δ^l denotes the vector containing δ_j^l, for all j. Lastly, M denotes the number of training data we have in our dataset.

2.3.3 Bias-variance trade-off

A well known problem within supervised learning is the *bias-variance trade-off*. When developing and testing a new DNN model, we want to capture the best pattern recognition possible, which is done by training on a given

dataset. However, we also want the model to generalize in the best way possible, meaning that the model must be able to produce results well, based on unseen data. A high variance model yields a good fit for the training data, in terms of recognizing regularities, but fails at precision for individual data points. On the other hand, high bias yields a good fit in terms of single data points, but fails at recognizing regularities. It is not possible to obtain the best of both worlds, thus, an optimization problem must be solved. The following derivation is inspired from a book about elements of statistical learning, by T. Hastie et al. [HTF09].

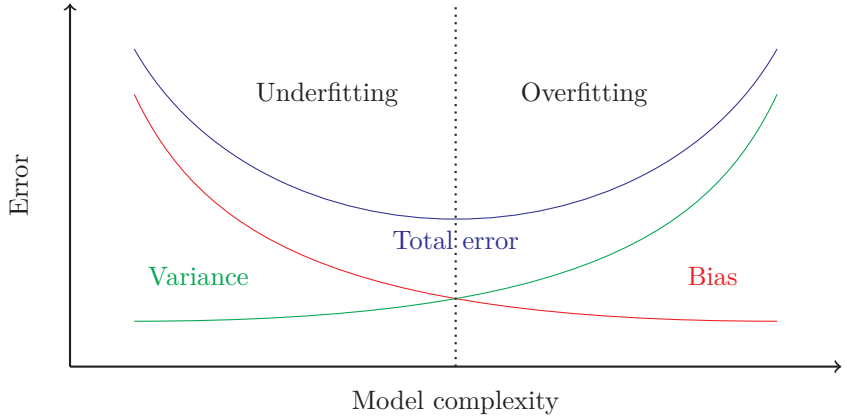


Figure 2.12: Illustration of the bias–variance trade-off. Left-hand side consist of underfitted models, while right-hand side consist of overfitted models. The optimal model complexity is along the dotted line.

Consider a random variable U, and let a function with noise be defined as $F_\varepsilon = F(U) + \varepsilon$, where the noise ε has zero mean and variance σ^2. Further, let $\mathcal{N} = \mathcal{N}(U)$ be the network we want to train with respect to F. The bias of $\mathcal{N}$ is then defined as

$$\text{Bias}(\mathcal{N}) \coloneqq F - E[\mathcal{N}],$$

where $E[\cdot]$ denotes the *expected value*. Furthermore, since F is deterministic, we have that $E[F] = F$. Moreover, the variance of $\mathcal{N}$ may be written in terms of the expected value by the identity

$$\text{Var}(\mathcal{N}) = E\big[(E[\mathcal{N}] - \mathcal{N})^2\big].$$

Lastly, by using the identity $\text{Var}(\varepsilon) = E[\varepsilon^2] - E[\varepsilon]^2$, we have

$$\begin{aligned}\text{Var}(F_\varepsilon) &= E\Big[\big(F_\varepsilon - E[F_\varepsilon]\big)^2\Big] = E\Big[\big(F + \varepsilon - F\big)^2\Big] \\ &= E\big[\varepsilon^2\big] = \text{Var}(\varepsilon) + E[\varepsilon]^2 = \sigma^2.\end{aligned}$$

The MSE-loss function of the network is given by

$$\text{MSE}(\mathcal{N}) = E\big[(F_\varepsilon - \mathcal{N})^2\big],$$

and by using the mentioned identities, we have

$$\text{MSE}(\mathcal{N}) = E\big[(F + \varepsilon - \mathcal{N} + E[\mathcal{N}] - E[\mathcal{N}])^2\big]$$

$$
\begin{aligned}
&= E\big[(F - E[\mathcal{N}])^2\big] + E\big[(E[\mathcal{N}] - \mathcal{N})^2\big] + E\big[\varepsilon^2\big] + 2E[\varepsilon(E[\mathcal{N}] - \mathcal{N})] \\
&\quad + 2E[(F - E[\mathcal{N}])\varepsilon] + 2E[(F - E[\mathcal{N}])(E[\mathcal{N}] - \mathcal{N})] \\
&= \mathrm{Bias}(\mathcal{N})^2 + \mathrm{Var}(\mathcal{N}) + \sigma^2 + 2E[\varepsilon]E[(E[\mathcal{N}] - \mathcal{N})] \\
&\quad + 2(F - E[\mathcal{N}])E[\varepsilon] + 2(F - E[\mathcal{N}])E[(E[\mathcal{N}] - \mathcal{N})],
\end{aligned}
$$

where the three last terms vanishes. Thus, we end up with

$$\mathrm{MSE}(\mathcal{N}) = \mathrm{Bias}(\mathcal{N})^2 + \mathrm{Var}(\mathcal{N}) + \sigma^2,$$

which shows that the MSE may be decomposed in terms of variance and bias – this bias-variance decomposition is illustrated in Figure 2.12. The dotted line illustrates our desired training precision, while right- and left-hand sides yields over- and underfitted models, respectively.

By substituting the function F to be Godunov's flux function F^{God}, and considering two random variables being cell averages u_i^n and u_{i+1}^n, this may be translated to apply for our one-dimensional initial-value problems. Thus, when developing the numerical methods and running experiments, such a bias-variance trade-off must be considered to avoid over- and underfitted models.

2.3.4 Activation functions

The term *activation function* comes from the fact that the functions are used to activate the layers in some sense, i.e. the values are constrained in some desired way. Some properties of activation functions are pretty standard among most experimental data scientists, namely

- nonlinearity, which opens for broader usage of DNNs,
- boundedness, which ensures more stability for gradient-based methods,
- continuous differentiability, which enables for gradient-based optimization,
- monotonicity, which ensures convexity for the error-surface of a single-layered DNN.

The choice of activation function when initializing a DNN is crucial for the sake of performance, but there may be many good and efficient alternatives. The one used in Chapter 3 is chosen due to its simplicity and its well known performance, historically speaking. This might not be the best choice for solving initial-value problems, but since the aim of this thesis is to show that the numerical DNN solvers are usable for solving such problems, we will not focus on optimizing the choice of activation function.

The function used in the DNN solvers presented in this thesis is the *Rectified Linear Unit* function – hereby abbreviated ReLU. This is a piecewise linear, monotone and unbounded function which is differentiable almost everywhere. The function is defined as

$$\varphi(x) = \begin{cases} 0, & 0 < x, \\ x, & x \geq 0, \end{cases}$$

and has the derivative

$$\varphi'(x) = \begin{cases} 0, & 0 < x, \\ 1, & x \geq 0, \end{cases}$$

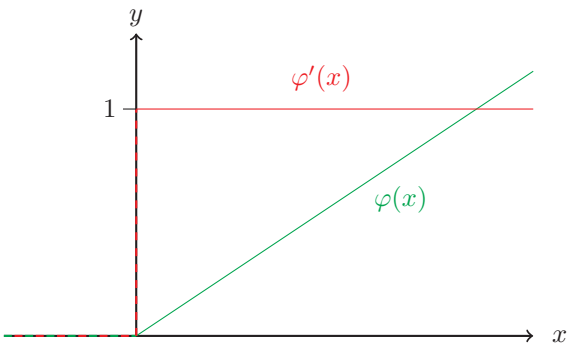


Figure 2.13: ReLU activation function (green), and its derivative (red).

illustrated in Figure 2.13. The lack of continuous differentiability may cause problems. However, its simplicity has shown itself useful when working with complex models, as it does not require a lot of computational power. Another issue with the ReLU function is that it does not span the whole real line. This is a problem since we aim at approximating the flux of a conservation law, and this might hold negative values. But as long as this is known by the developer, a way to avoid this is to not use the ReLU activation in between the last two layers – as mentioned at the end of the introductory terminology we will use the identity mapping in the last transition.

2.3.5 Universal approximation theorem

The research field concerning DNNs is quite young and theories are therefore under constant development. One of the most famous results within the field is the *Universal Approximation Theorem*. The most common version of the result was given by G. Cybenko in 1989 [Cyb89] and K. Hornik in 1991 [Hor91]. Cybenko showed us that by fixing the number of hidden layers in a DNN, one may obtain arbitrarily good results when approximating a continuous function, by only increasing the width of the network, as long as we use a continuous activation function. Contrary to this, Hornik showed that one might as well keep the width of the network fixed, while increasing the depth of the network. Thus, this makes up a theorem stating that we may approximate any continuous function with arbitrary good precision, by using a complex enough DNN model, with a continuous activation function. Quite a few years later, in 2017, L. Zhou extended this result to hold for any Lebesgue measurable function, and not just continuous functions [Zho+17]. However, this only applies for DNNs using the ReLU activation function. We will state this as a theorem.

Theorem 2.3.1. *For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\varepsilon > 0$, there exists a fully-connected ReLU network $\mathcal{N} = \mathcal{N}(x)$ with width $d \leq n + 4$, such that the network satisfies*

$$\int_{\mathbb{R}^n} |f(x) - \mathcal{N}(x)| \, \mathrm{d}x < \varepsilon.$$

Furthermore, as recently as in 2020 more generalized versions extending to non-affine activation functions and arbitrary depth in the network have

been presented [KB20; KL20]. However, since we will use ReLU as activation function in all of our experiments, it is sufficient to consider Theorem 2.3.1.

We started this chapter by looking into the analytical aspect of scalar conservation laws, which gave us explicit solution formulas for one-dimensional initial-value problems. These solutions will work as reference during experiments in Chapter 3, and will tell us how accurate our implemented methods are. Further, we have derived the general form of a one-dimensional finite-volume scheme, which will come in handy when we introduce our new methods in Sections 3.2 and 3.3. The numerical method for approximating the two-dimensional initial-value problems of Section 3.4 will be derived in full detail in that section, but it contains similarities to already derived method. Lastly, we have looked into the fundamental properties of DNN models, as well as how to train such models. The pseudocodes introduced in Algorithms 1 and 2 are the basis of the implementations, and more details may be located in Appendix A.

CHAPTER 3

Numerical methods and experiments

In this chapter we will develop and test DNN based numerical methods, all of which will approximate solutions of initial-value problems – both one- and two-dimensional. The hope is that DNN based two-dimensional methods can give us a significant advantage with respect to efficiency, compared to other numerical methods. During the experiments we will get an indication on usefulness of our methods by looking at both the accuracy and the efficiency. In Section 3.1 we will introduce common preliminaries for all experiments, which includes the accuracy measure used, how to create a good dataset and how to ensure reproducibility of our results. When moving on to Section 3.2 we will present the first numerical method, which is a DNN based method for solving one-dimensional initial-value problems by using an intuitive choice of loss function. The aim of this section is not to outperform the efficiency of alternative numerical schemes, as the DNNs will most likely be significantly more time consuming than e.g. Godunov's scheme. However, the focus is on accuracy to ensure that such methods actually function. In Section 3.3 we will approximate solutions to the same problems as in previous section, but with a physics-informed DNN, i.e. we use a loss function which will teach the network some structural properties of the scalar conservation law. The hope of this section is to obtain more stability and a greater rate of convergence. Section 3.4 contains the main result of this thesis, namely a new numerical DNN based method for approximating solutions of two-dimensional conservation laws. Such problems do not have explicit solution formulas, and existing numerical methods are therefore extremely time consuming as one must use high resolution grids to obtain accurate estimates. Thus, the aim is to obtain significantly more efficiency, with approximately similar results as one would get by using a fine-mesh solver. Lastly, Section 3.5 closes this chapter by mentioning potential sources of error in all experiments.

3.1 Preliminaries

All performed experiments of this thesis have common factors which have been considered throughout the entire process of development. To measure accuracy of the methods we have considered relative errors, which is a metric introduced in the following subsection. Each and every DNN model used in

the numerical schemes require good datasets, thus we will list some important properties of what we consider a great dataset. Lastly, we will look into the degree of reproducibility of results, which includes notes on both the software and hardware.

3.1.1 Error measure

When performing the experiments of the DNN solvers it is important to have a concise and precise measure of accuracy with respect to the reference solutions. Let u_{DNN} denote the DNN based approximation, and let u be the reference solution. A natural and intuitive metric is to calculate the L^p-norm of the difference,

$$e_a = \|u - u_{\text{DNN}}\|_p,$$

known as the *absolute error*, which yields a direct comparison between the approximation and the reference solution. However, comparing absolute errors of two approximations from different initial-value problems, i.e. different initial conditions, does not make sense. We need to scale this error down to a dimensionless number where comparisons may be done, regardless of parameters. By dividing the absolute error by the norm of the reference solution we obtain,

$$e_r = \frac{e_a}{\|u\|_p} = \frac{\|u - u_{\text{DNN}}\|_p}{\|u\|_p},$$

which is known as the *relative error*. This essentially tells us how good of a decimal precision the approximations hold – meaning that $e_r = 10^{-(n+1)}$ may be roughly translated to being precise down to the nth decimal. The choice of L^p-norm varies, often with respect to the specific problem at hand. During the experiments, we will determine the accuracy in terms of the Euclidean distance between the approximate solutions and their corresponding reference solutions. Euclidean distance is often a computationally preferable choice when we are dealing with gradients. Thus, we will consider the Euclidean relative error.

3.1.2 Data generator

Behind every great DNN model, there exists a great dataset. During the experiments, we must generate our own datasets, which contain two pieces of information, namely the input values and the desired output values. When working in the numerical world, there are some limitations we need to consider. We have a finite range of precision, all relying on the software and hardware used for the experiments. We must also make sure that the datasets span as much of the desired space as possible, otherwise the model will be poorly trained due to inconsistencies in the training data. This motivates the following list of things to think about when creating a dataset. Every dataset should be

- accurate: the target values must be calculated as precisely as possible,
- valid: the values must lie within the desired domain,
- consistent: data must be stable, i.e. not discontinuities when changing input slightly,
- complete: data covers all of the domain in best fashion possible,

- unique: the input combinations do not map to more than one combination in the output.

3.1.3 Reproducibility

When doing experiments, it is important that the reader is able to reproduce the results, so that the conclusions given in the thesis may be retested and verified. This would mean that the developer needs to share mainly two bulks of information, namely the seed used under random number generation and the software versions used. For consistency in the experiments, we have used seeding with respect to the two main packages used in the development, namely NumPy and PyTorch. Both of these packages are used with their respective builtin random number generators. The seeding values are in both cases set to be 42, using the following code.

```
1 torch.random.manual_seed(42)
2 torch.manual_seed(42)
3 torch.cuda.manual_seed(42)
4 np.random.seed(42)
```

By using this seed value, we will obtain consistency to some extent both in data generation and the network training process. However, there are some processes within PyTorch which are stochastic. The given seeding is a way of limiting the non-deterministic processes, but will not eliminate these fully. Another source of stochastic processes is the library called cuDNN, which is a GPU-accelerated library. It is possible to set the imported packages to a fully deterministic behavior, however, this is not done due to the lack of efficiency while doing so. The created implementation behind this thesis is already computationally heavy, and we have therefore decided to use the stochastic processes to obtain an efficient code. Table 3.1 shows the versions of software used for the experiments. The reader should also be aware of the differences that may be found in the

Package	NumPy	PyTorch	matplotlib
Version	1.19.2	1.7.0	3.3.2

Table 3.1: Versions of software.

hardware of the computer used for calculations – this could e.g. be two different GPU cards. This could lead to differences in results, even though seeding is applied with respect to the libraries used and all processes are set to be deterministic.

3.2 DNN solver with MSE-loss in $\mathbb{R}$

In this section we will consider a dense neural network (DNN) as simple as they come, with mean squared error (MSE) as our loss function. This network will then work as an approximation of the flux in (2.17), and thereby makes up a complete alternative numerical method for approximating the solution of one-dimensional, initial-value problems,

$$\begin{cases} u_t + f(u)_x = 0, & (x,t) \in \Omega \times \mathbb{R}_+ \\ u(x,0) = u_0(x), & x \in \Omega. \end{cases} \tag{3.1}$$

First, we introduce the methodology, and then move on to the experiments and results. The method will be compared to the Godunov scheme in terms of relative errors, together with a short discussion. We will also study the training process in terms of two factors, namely how the loss and weights changes during each epoch. This will give a strong indication whether or not the DNN models converge toward a stable state.

3.2.1 Numerical method

In this section we will derive a DNN based numerical method for approximating solutions of one-dimensional initial-value problems. Many numerical methods for approximating such problems already exist, one of which is Godunov's method, which is the basis for the training of our DNN models. We will use the discretization introduced in Section 2.2.1, together with the derived general form of any finite-volume scheme, (2.18). The aim of this section is then to replace $\bar{F}$ with a pre-trained DNN, trained on a dataset created by using (2.24), with the hope that this yields accurate approximations of the solution to (3.1).

Let the DNN for our numerical method be denoted $\mathcal{N}$. The creation of training data used for $\mathcal{N}$ is quite straight forward. Godunov's flux function, given in (2.24), is a function dependent on two variables, $F^{\text{God}} : \mathbb{R}^2 \to \mathbb{R}$. Consequently, $\mathcal{N}$ needs 2 input neurons and 1 output neuron, to be able to approximate F^{God}. Further, we let M denote the number of data points to train over, yielding an $M \times 3$-matrix,

$$\mathbb{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & Y_1 \\ \vdots & \vdots & \vdots \\ X_{M,1} & X_{M,2} & Y_M \end{bmatrix},$$

where $X_{i,1}$ and $X_{i,2}$ represents the input values, while Y_i is their respective target values – a network for this data is what is illustrated in Figure 2.10. The pseudocode for the data generator algorithm is given in Algorithm 3. After

Algorithm 3 Data generator

Select $M \times 2$ standard normally distributed numbers $X_{i,1}, X_{i,2}$, $\forall i \in [\![1, M]\!]$.
for $i = 1, \ldots, M$ **do**
    $Y_i = F^{\text{God}}(X_{i,1}, X_{i,2})$

forward passing $X_{i,1}$ and $X_{i,2}$ through the network, using Algorithm 1, we obtain output value $\hat{Y}_i = \mathcal{N}(X_{i,1}, X_{i,2})$ which will be compared to Y_i by using the MSE loss function,

$$C(\hat{\mathbf{Y}}, \mathbf{Y}; \mathbf{W}, \mathbf{b}) \coloneqq \frac{1}{N} \sum_{k=1}^{N} \left(\hat{Y}_{i,k} - Y_{i,k} \right)^2.$$

In this section we simply have $N = 1$, since the output dimension of $\mathcal{N}$ is 1, so we simply write $\hat{Y}_i = \hat{Y}_{i,1}$ and $Y_i = Y_{i,1}$. Thus, the backpropagation algorithm derived in Section 2.3.2, Algorithm 2, will be used by applying this value for N. During the DNN training we will use data batches with a specified batch size, i.e. the M data rows will be divided into equally sized batches. Therefore,

the loss value we will use for backpropagation will be the mean average of the loss from each member of the batch. After each epoch, i.e. when we have iterated through the whole dataset once, we will forward pass unseen data from a validation dataset, which will show how the network performs on unseen data during the training – the network will not adjust based on performance of these results. When the training session is complete we wind up with a fully functional numerical method,

$$u_i^{n+1} = u_i^n - \frac{\Delta t^{n+1}}{\Delta x}\big(\mathcal{N}\big(u_i^n, u_{i+1}^n\big) - \mathcal{N}\big(u_{i-1}^n, u_i^n\big)\big),$$

yielding an approximate solution of (3.1).

3.2.2 Baseline for experiments

We will perform experiments using a standard one-dimensional scalar conservation law, with Burgers' flux function. Thus, the initial-value problem is written out as

$$\begin{cases} \partial_t u(x,t) + \partial_x\Big(\frac{u(x,t)^2}{2}\Big) = 0, & (x,t) \in \Omega \times \mathbb{R}_+, \\ u(x,0) = u_0(x), & x \in \Omega, \end{cases} \tag{3.2}$$

where we will test different choices of initial functions $u_0(x)$. The experiments will be performed using the spatial domain $\Omega = [-1, 1]$. The spatial mesh points are distributed uniformly in the domain, with mesh size $N_x = 50$, and are thence given by

$$x_i \coloneqq -1 + \left(i + \frac{1}{2}\right)\Delta x, \qquad \Delta x \coloneqq \frac{2}{51}.$$

To ensure stability of the method, we induce a CFL-condition, (2.20), with Courant number $C_{CFL} = 1/2$, leading to temporal bounded step sizes defined as

$$\Delta t^n \coloneqq \frac{\Delta x}{2\max_i(f'(u_i^n))} = \frac{\Delta x}{2\max_i(u_i^n)},$$

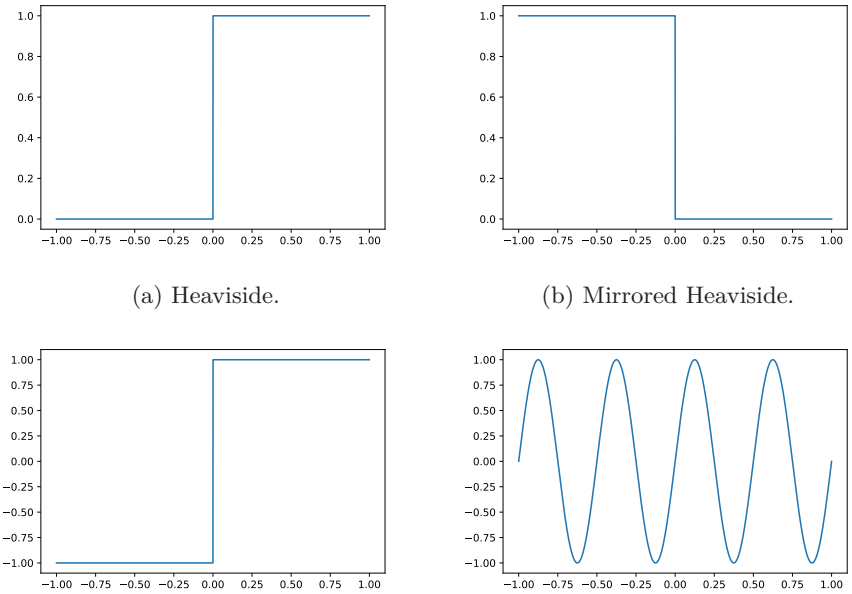
where we have written out the derivative of Burgers' flux function. This will ensure that we avoid collision with spatially neighboring Riemann problems, i.e. the wave reaches the temporal boundary of the cell, before approaching the spatial boundary. Moreover, the maximum time of propagation is set to be $T = 0.5$.

We will perform tests using 4 different initial conditions, which will show propagation over time for both shock waves and rarefaction waves. The first initial function we will test is the well known Heaviside function. This is defined as

$$u_0(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x > 0, \end{cases} \tag{3.3}$$

illustrated in Figure 3.1a. This will hopefully show us how positive valued rarefaction waves typically propagate through space over time. The next initial condition we will test is the additive inverse function of Heaviside, defined as

$$u_0(x) = \begin{cases} 1 & \text{if } x < 0, \\ 0 & \text{if } x > 0, \end{cases} \tag{3.4}$$



(a) Heaviside.

(b) Mirrored Heaviside.

(c) Scaled Heaviside.

(d) Sine wave.

Figure 3.1: The four initial conditions used for one-dimensional experiments. (a) Heaviside step function. (b) Mirrored Heaviside, i.e. the additive inverse of the Heaviside function. (c) Scaled Heaviside. (d) Sine function.

illustrated in Figure 3.1b. This will show us how a single shock wave propagates in space over time. The third function considered is a shifted and scaled Heaviside function, given by

$$u_0(x) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{if } x > 0, \end{cases} \tag{3.5}$$

illustrated in Figure 3.1c. Contrary to the original Heaviside function, this will show us how the networks can handle both negative and positive valued rarefaction waves. Lastly, we will test the DNN based method on continuous initial data, by considering a sine wave given by

$$u_0(x) = \sin(4\pi x), \tag{3.6}$$

illustrated in Figure 3.1d. This will show how the numerical method will be able to handle an alternating combination of shock waves and rarefaction waves.

When performing the experiments one must consider what conditions the boundary of the domain must satisfy. This gives us many choices, some of which are better than others. During experiments with the three initial conditions in Figures 3.1a to 3.1c, we will use a boundary condition known as *Neumann boundary condition* – named after the German mathematician Carl Neumann. This condition specifies the value of derivatives of the solution at the boundary. The derivative at the boundaries $x = -1$ and $x = 1$, at time t^n, is set equal to

the derivative on the boundary for the previous temporal step, t^{n-1}. Thus, the implemented Neumann condition is written out as

$$\partial_x u(x, t^n) = \partial_x u\big(x, t^{n-1}\big), \qquad \forall n = 1, 2, \ldots, \quad x = -1, 1.$$

When considering the fourth initial condition, namely the sine function in Figure 3.1d, we will set a periodic boundary condition to our problem, i.e. changes at the boundary will be reflected on the opposite side of the boundary. This is written out as

$$u(-1, t^n) = u(1, t^n),$$

for all $n \in \mathbb{N}_0$. Apart from the parameters mentioned for the discretization,

Name	**Parameter**
Data distribution	Gauss/Normal
Training data size	$\lvert\mathbb{X}\rvert = 100.000 \times 3$
Validation data size	$\lvert\mathbb{X}_V\rvert = 10.000 \times 3$
Batch size	100
Epochs	20
Loss function	$C =$ `torch.nn.MSELoss`
Activation function	$\varphi =$ `torch.relu`
Optimizer	`torch.optim.Adam`
Learning rate	$\kappa = 10^{-3}$
Hidden layer number	$L \in [\![1, 4]\!]$ *
Nodes per layer	$n_l \in [\![4, 1024]\!]$ *

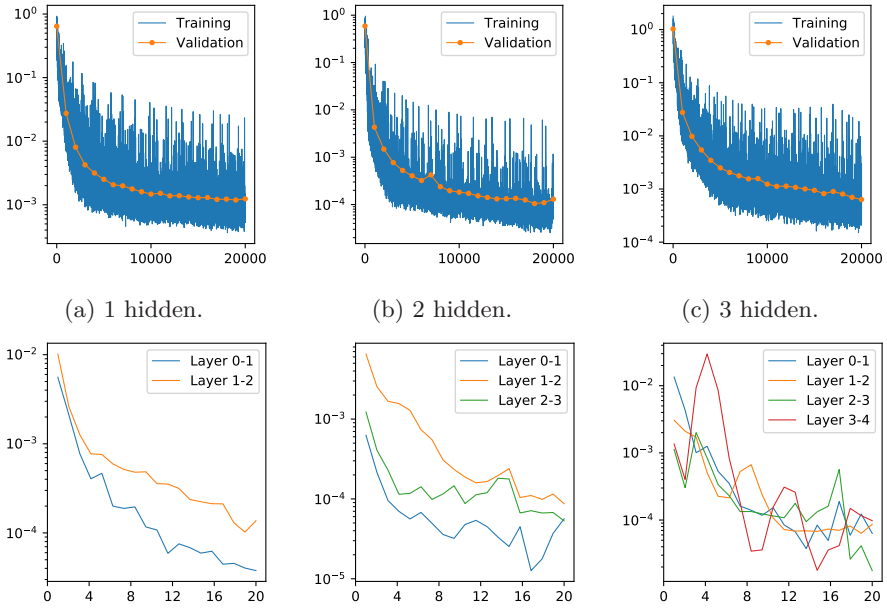
Table 3.2: DNN parameters used for experiments. *These parameters vary.

there are a few other parameters chosen during the implementation of the numerical method, for specific machine learning purposes. These parameters are listed in Table 3.2. The data distribution, being normally distributed, refers to the input parameters created by the dataset generator. The Adam optimizer is an optimization algorithm presented by D. Kingma in an article in 2015 [KB15]. It is a computationally efficient extension to the stochastic gradient descent algorithm. Batch size of 100 means that the network is fed 100 rows of data, before it backpropagates and corrects the weights and biases, thus correcting with respect to the whole batch. The number of epochs being 20 means that we will iterate all batches through the network 20 times during the training process. The two last rows of Table 3.2 marked with a $*$ is the only parameters varied throughout the experiments.

3.2.3 Experiments

In the following experiments we will start by studying how increasing the number of hidden layers in the network affects the solution. This will be done in two steps, first with the total number of nodes being constantly low, and then we perform the same batch of experiments on the same number of layers but with higher number of nodes in each layer – this is to verify our results. Then, we will study how changing the number of nodes affects the results, by

keeping the number of layers constant in the same matter. In theory, the results should improve as long as we keep increasing the complexity of our model, i.e. increasing the number of hidden layers and nodes. However, by increasing either the number of nodes or layers to much, this will result in a very inefficient numerical method, as well as yielding an increase of the run time complexity of our training process – this would also require greater amount of training data and/or epochs. Hence, our goal is to test and determine how good of a model we could make, with as little time complexity as possible. Thus, we are faced with an optimization problem, which consists of a correctness- and runtime-trade-off.



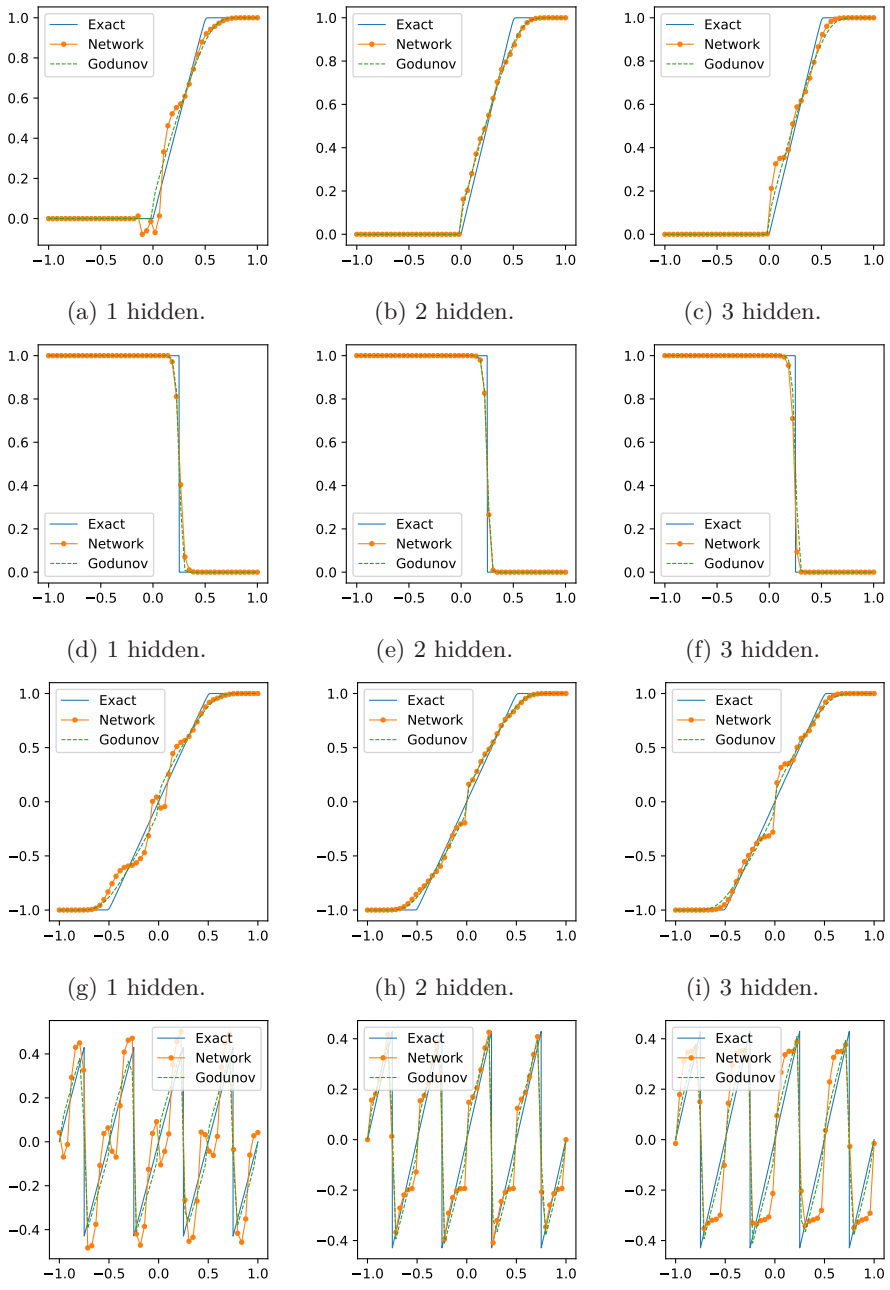
(a) 1 hidden. (b) 2 hidden. (c) 3 hidden.

(d) 1 hidden. (e) 2 hidden. (f) 3 hidden.

Figure 3.2: The training loss (upper blue), validation loss (upper yellow) and change of weights (lower), where the total number of nodes are 24, while the number of hidden layers vary from 1 to 3. The horizontal axes are number of iterations in the loss plots, and number of epochs in weight plots.

Network	**Heaviside**	**Mirrored**	**Scaled**	**Sine**
24	8.56×10^{-2}	2.36×10^{-2}	9.15×10^{-2}	5.24×10^{-1}
$12 \bowtie 12$	1.43×10^{-2}	5.82×10^{-3}	2.36×10^{-2}	1.76×10^{-1}
$8 \bowtie 8 \bowtie 8$	4.39×10^{-2}	4.34×10^{-2}	5.13×10^{-2}	3.22×10^{-1}
$6 \bowtie 6 \bowtie 6 \bowtie 6$	8.65×10^{-2}	5.72×10^{-2}	8.69×10^{-2}	4.35×10^{-1}

Table 3.3: Relative Euclidean error of 16 experiments with varying number of hidden layers and initial functions. The total number of nodes in network is constantly 24. Green cells show the best results for each initial function.



(a) 1 hidden. (b) 2 hidden. (c) 3 hidden.

(d) 1 hidden. (e) 2 hidden. (f) 3 hidden.

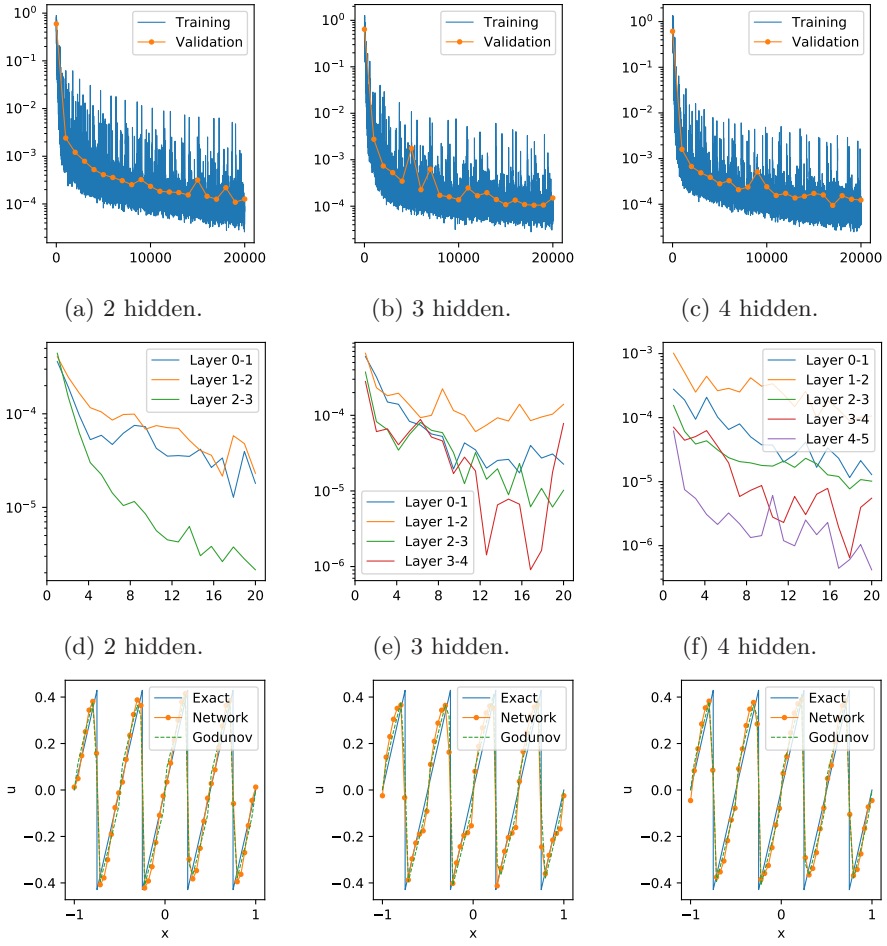
(g) 1 hidden. (h) 2 hidden. (i) 3 hidden.

(j) 1 hidden. (k) 2 hidden. (l) 3 hidden.

Figure 3.3: Result of 1D experiments with 24 nodes, $T = 0.5$, $N_x = 50$, $\Omega = [-1, 1]$. Boundary condition: Neumann (row 1-3) and periodic (row 4). Row 1: Heaviside, row 2: Heaviside mirrored at $x = 0$, row 3: Heaviside scaled and shifted, row 4: sine function. Horizontal axes: spatial dimension, vertical axes: solution values u. Blue: exact solution, green: Godunov's approximation, yellow: DNN based approximation.

We start by creating a DNN with the possibility of arbitrary number of hidden layers and number of nodes, as well as two input parameters and one output parameter. Then, we perform controlled, seeded tests with few layers and nodes, and study the impact of increasing the number of hidden layers, while holding the number of nodes constant. Further, we set the number of nodes to 24. For simplicity, we will not perform experiments with network depth greater than 4, so this allows us to perform four tests with layer sizes 24, 12, 8 and 6 for networks with 1, 2, 3 and 4 hidden layers, respectively. The idea in this first batch of testing is to find out whether or not there exist some advantages by keeping the depth restricted. When running these four simulations for 20 epochs, we obtain the losses illustrated in Figures 3.2a to 3.2c – not including plots for 4 hidden layers. Observe that these losses have flattened out after 20 epochs, which indicates that more training hardly would make any significant difference. At this point, one should be aware that the plots in Figure 3.2 are made with logarithmic scales. Their respective change of weights between each epoch is illustrated in Figures 3.2d to 3.2f, which shows the distribution of change among the weights in the DNN, during the training process. The numerical approximations to the scalar conservation law after time $T = 0.5$ are illustrated in Figure 3.3, which show results for the different number of hidden layers in the columns, from 1 to 3. The rows show different choices of initial conditions, introduced in previous section. As we may observe, there is not necessarily an advantage when introducing a new hidden layer, given that we keep the total number of nodes constantly low (in this case equal to 24 for division simplicity). In the case with one hidden layer we see that the results are somewhat unstable when dealing with rarefaction waves, see Figures 3.3a and 3.3g. On the other hand, we may observe, from Figure 3.3d, that the results are nearly identical to the Godunov flux for shock waves. When alternating between rarefaction waves and shock waves we make the same observation, see Figure 3.3j. From the 12 figures given in Figure 3.3, we see that the absolute best choice of number of hidden layers, when using 24 nodes, is 2, assuming that the nodes are divided equally among the hidden layers. To emphasize this conclusion we can also study the relative errors of the DNN solver with respect to Godunov's scheme, see Table 3.3. The green cells verify that the smallest errors are found in the network consisting of 2 hidden layers. Since our first results are based on quite few nodes, this can not be generalized to examples of sufficiently larger amount of nodes, divided equally among the layers. A reasonable hypothesis, for why 2 hidden layers yield better results than 3 and 4 hidden layers, is that this has to do with the number of nodes per layer being too small in the two latter cases. This leaves us two choices if we want to improve our model, namely either increasing the number of nodes per layer, or keep using the number of nodes as in the case of 2 hidden layers, namely 12, and increasing the number of hidden layers.

The next experiments are performed using a higher number of nodes per layer. The ratios of nodes per layer are kept 1 as above, but the total number of nodes is now set to be 48, i.e. double of the amount from earlier. The expected results would then be that models with 3 and 4 hidden layers have increasingly better outputs, depending on our choice of the total number of nodes. Contrary to the weight changes in previous experiments, Figures 3.4d to 3.4f yields an interesting observation. In these cases, we see that the further outward in the network you go the less of a change is made during training. It seems that



(a) 2 hidden. (b) 3 hidden. (c) 4 hidden.

(d) 2 hidden. (e) 3 hidden. (f) 4 hidden.

(g) 2 hidden. (h) 3 hidden. (i) 4 hidden.

Figure 3.4: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results (lower), in their respective rows. Columns show different number of hidden layers. All DNN models have a total of 48 nodes distributed uniformly.

Network	Heaviside	Mirrored	Scaled	Sine
48	4.32×10^{-2}	3.46×10^{-2}	3.01×10^{-2}	2.03×10^{-1}
$24 \bowtie 24$	2.79×10^{-2}	2.69×10^{-2}	2.32×10^{-2}	1.52×10^{-1}
$16 \bowtie 16 \bowtie 16$	3.29×10^{-2}	3.79×10^{-3}	2.65×10^{-2}	1.95×10^{-1}
$12 \bowtie 12 \bowtie 12 \bowtie 12$	2.52×10^{-2}	2.06×10^{-2}	2.10×10^{-2}	1.02×10^{-1}

Table 3.4: Relative Euclidean error of 16 experiments with varying number of hidden layers and initial functions. The total number of nodes in network is constant 48. Green cells show the best results for each initial function.

the greatest changes during the 20 epochs are done to weights between layer 0 and 1, and then between 1 and 2, etc. And this seems to apply for all four test cases. We will in this case only show the results for the sine waves, as this will tell us the networks behavior for both rarefaction waves and shock waves, see Figures 3.4g to 3.4i. It is hard to tell which is the better model, so as above we will include the table of relative error to easier point out the right one. Glimpsing at Table 3.4, we observe that the relative errors of the results are tremendously better for the cases of 3 and 4 hidden layers. This indicates that our hypothesis may be correct – more results, with different number of nodes and seed values, are needed to be certain.

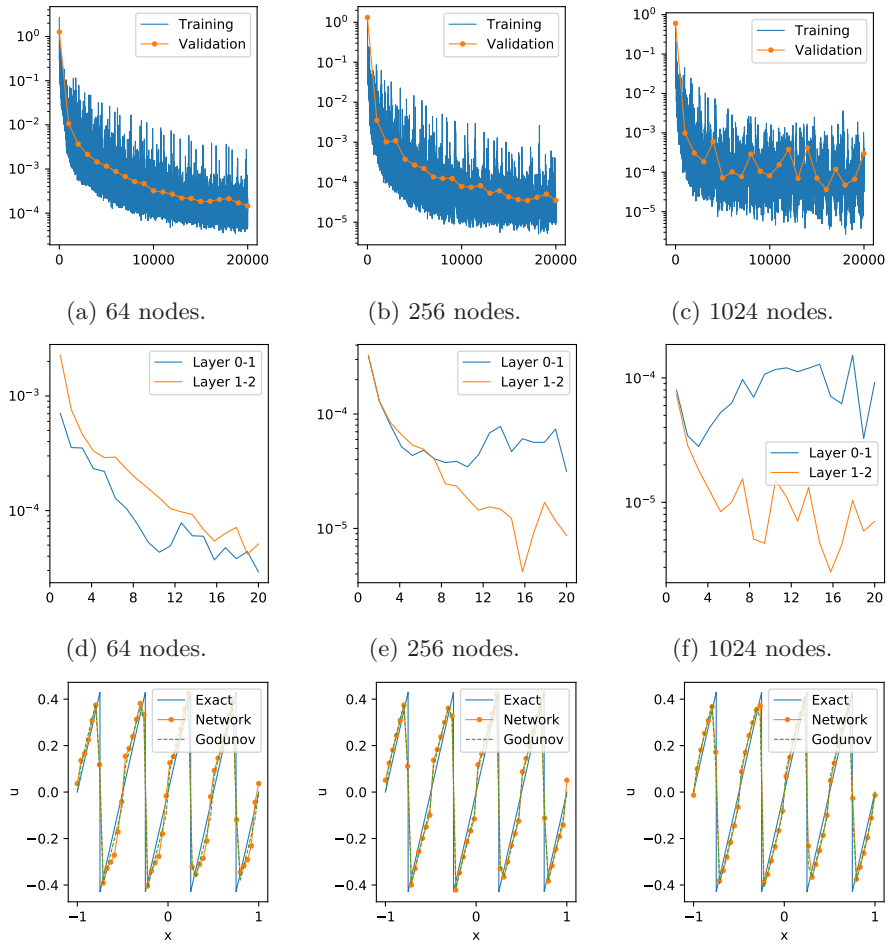
The next batch of experiments is for studying the advantage – or disadvantage – of increasing the number of nodes in each layer, while keeping the number of layers constant. We start by using 1 hidden layer, while varying the number of nodes as 4^i for $i \in [\![1, 5]\!]$, see Table 3.5. Once again, the green cells highlight the best trained models. We see that by increasing the complexity we obtain more satisfactory models. However, this is only true up to some bound of complexity. By increasing the number of nodes too much, the size of the network requires more training – solved by either increasing the data size, changing the batch size or simply increasing the number of epochs to train over. When looking at Figure 3.5, we see that all losses are flattening a great deal already, so increasing the number of epochs to train over would hardly help the models in this case. However, the non-flatness in change of weights in the model of 1024 nodes indicates that more training would most likely increase the correctness of results to some extent, see Figure 3.5f. On the other hand, more training could actually yield worse results, since the training data set might be scarce – worse in the sense that we could wind up overfitting the model. To test the

Network	Heaviside	Mirrored	Scaled	Sine
4	3.08×10^{-1}	2.57×10^{-1}	2.55×10^{-1}	1.13×10^{0}
16	9.55×10^{-2}	4.04×10^{-2}	8.87×10^{-2}	5.46×10^{-1}
64	2.75×10^{-2}	1.44×10^{-2}	2.28×10^{-2}	1.23×10^{-1}
256	7.74×10^{-3}	1.54×10^{-3}	6.12×10^{-3}	5.78×10^{-2}
1024	7.19×10^{-3}	1.13×10^{-2}	8.38×10^{-3}	9.03×10^{-2}

Table 3.5: Relative Euclidean error of 20 experiments with varying number of nodes and initial functions. The number of hidden layers in network is constantly 1. Green cells show the best results for each initial function.

Network	Heaviside	Mirrored	Scaled	Sine
256	6.56×10^{-3}	4.41×10^{-3}	7.61×10^{-3}	7.12×10^{-2}
1024	3.13×10^{-3}	4.03×10^{-3}	5.61×10^{-3}	5.03×10^{-2}

Table 3.6: Relative Euclidean error of the last two rows of experiments in Table 3.5, but with 20 epochs further training, i.e. a total of 40 epochs. The number of hidden layers in network is constantly 1. Green cells show the best results for each initial function. The red cells show cases where the error is worse than in the case of less training.



(a) 64 nodes. (b) 256 nodes. (c) 1024 nodes.

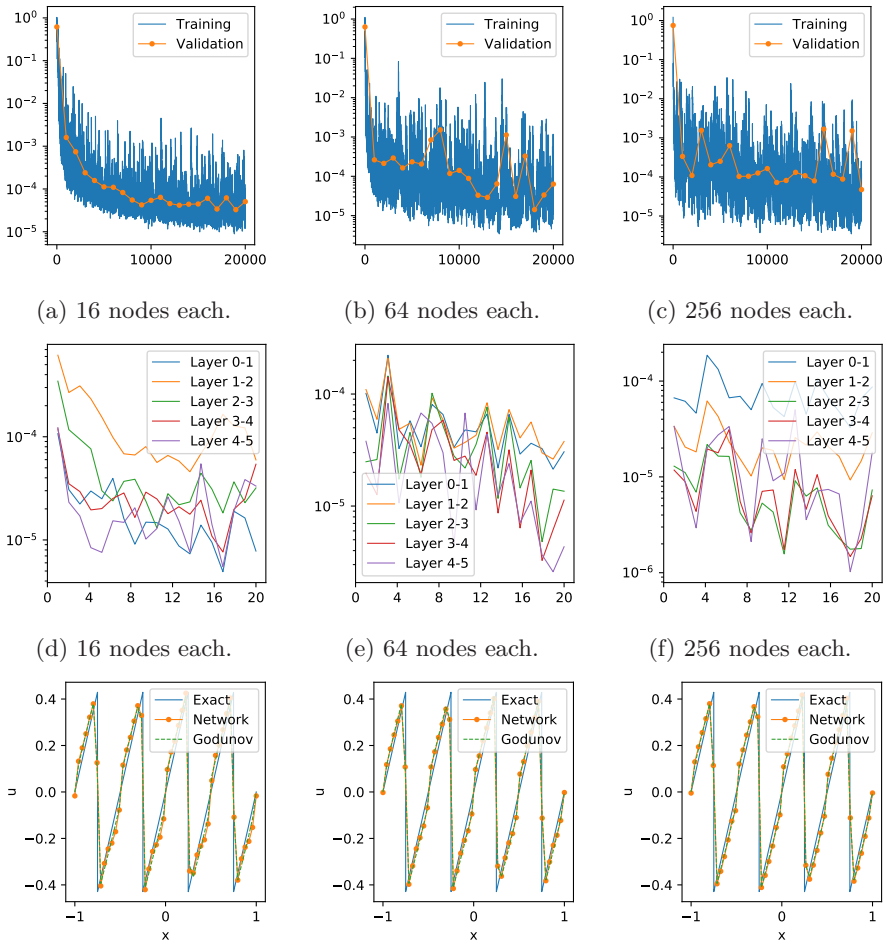
(d) 64 nodes. (e) 256 nodes. (f) 1024 nodes.

(g) 64 nodes. (h) 256 nodes. (i) 1024 nodes.

Figure 3.5: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initial (lower). Varying node number with 1 hidden layer.

hypothesis of whether or not the last mentioned model performs better with more training, we double the number of epochs, leading to training with 40 epochs. For comparison we also train the model of 256 nodes 20 extra epochs. The results are shown in Table 3.6, where the results are as expected, i.e. the most complex model outperforms the other model in every experiment. Another observation made is that the less complex model is actually doing worse than in the previous case of 20 epochs. This indicates that some weights are overfitted to some batches of data.

To generalize our conclusions for multilayered DNNs we need to perform yet another batch of experiments, where we increase the number of hidden layers, and then test with varying number of nodes. We will set the number of layers to 4 and then perform the same experiments as above, i.e. with the number of



(a) 16 nodes each. (b) 64 nodes each. (c) 256 nodes each.

(d) 16 nodes each. (e) 64 nodes each. (f) 256 nodes each.

(g) 16 nodes each. (h) 64 nodes each. (i) 256 nodes each.

Figure 3.6: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initial (lower). Varying node number with 4 hidden layers.

Network	Heaviside	Mirrored	Scaled	Sine
$4 \lessapprox \cdots \lessapprox 4$	8.36×10^{-2}	5.24×10^{-2}	8.42×10^{-2}	6.36×10^{-1}
$16 \lessapprox \cdots \lessapprox 16$	1.53×10^{-2}	3.64×10^{-3}	1.34×10^{-2}	6.78×10^{-2}
$64 \lessapprox \cdots \lessapprox 64$	7.69×10^{-3}	1.79×10^{-3}	1.16×10^{-2}	3.74×10^{-2}
$256 \lessapprox \cdots \lessapprox 256$	8.56×10^{-3}	7.66×10^{-3}	5.25×10^{-3}	2.85×10^{-2}

Table 3.7: Relative Euclidean error of 20 experiments with varying number of nodes and initial functions. The number of hidden layers in network is constantly 4. Green cells show the best results for each initial function.

nodes in each layer being 4^i, $\forall i \in [\![1, 4]\!]$. The training loss and change of weights are illustrated in Figure 3.6, and the relative errors are given in Table 3.7. As we observe from the relative errors we once again have a model which is too complex to be fully trained during the given 20 epochs. This hypothesis is supported by the glimpse of Figure 3.6f, showing the change of weights. As in the previous batch of experiments, we also here see that some of the weight layers have not yet flattened out.

As we have seen from all above experiments, a DNN model on the form described in the methodology works well when looking at the accuracy of the models. However, in the introduction to this thesis we raised two questions in need of consideration when implementing and testing a new numerical method, namely accuracy and efficiency. Therefore, to be able to conclude, we must consider the time complexity of the algorithms used. Let us assume that we are given a pre-trained model, and so, we must compare Godunov's scheme up against our own model. The Godunov flux, given in (2.24), is calculated numerically by essentially performing three steps, namely

(i) check whether $u_i^n < u_{i+1}^n$ or $u_i^n > u_{i+1}^n$,

(ii) create a uniformly distributed list of desired mesh size, lying between u_i^n and u_{i+1}^n,

(iii) and calculate the flux of the list, and extract minimum or maximum value.

Thus, the complexity of Godunov's scheme depends on the optimization algorithm used to find the minimum/maximum values. It is therefore not possible to generalize in a way where we say that the efficiency of any Godunov implementation out-performs our model. However, based on all experiments we have performed, it is required a degree of network complexity which will be out-performed by most Godunov algorithms, to sufficiently approximate the solutions.

This concludes the experiment part of the DNN models for the scalar one-dimensional conservation laws with the intuitive choice of MSE-loss function. The results given in this section are the baseline for comparison, together with the Godunov scheme, when performing the later experiments using a different loss function, and then also when we move on to the numerical analysis of scalar conservation laws of two spatial dimensions.

3.3 DNN solver with extended L^1-loss in $\mathbb{R}$

In this section we will propose an alternative training methodology for the DNN models. The finite-volume method for approximating solutions to the initial-value problems are the same as in Section 3.2.1. The training method involves using a new type of loss function, namely one that will force the network to learn the structural properties of a finite-volume scheme, and not just the numerical flux. The hope is that this will lead to a model with significantly more stability than for the models used in previous section, i.e. we hope for higher rate of convergence. First, we introduce the methodology by describing the creation of datasets and how to use these to train the DNN models. Then, we move on to the experiments and results, testing the accuracy and efficiency of the method. We will perform similar simulations as in Section 3.2.3 – to be

able to make viable comparisons and conclusions. Lastly, we end by drawing lines to convolutional neural networks, since the new method is reminiscent to such neural systems.

3.3.1 Numerical method

We will now derive a DNN based numerical method which will teach a DNN model some structural properties of a scalar conservation law. The hope will then be to obtain a stable method for solving one-dimensional initial-value problems. We use the discretization in Section 2.2.1, together with the derived general form of any finite-volume scheme for one-dimensional conservation laws, see Section 2.2.2. The goal will then be to replace $\bar{F}$ with a DNN which is pre-trained with a loss function that teaches the network the structure of (2.18). The DNN model and its usage within a finite-volume scheme is similar to the described method used in Section 3.2.1. The difference of the two methods lies in the choice of loss function, and consequently the structure of the training data and time complexity of the training session. The dataset is created with the following step-wise procedure, using Godunov's flux approximation, (2.24).

(i) Fix integers M, N and K, and let the spatial step size be $\Delta x = \frac{1}{N}$.

(ii) Uniformly distribute $x_j \in [\Delta x, 1]$ for all $j \in [\![1, N]\!]$ and normally distribute $a_k \in (0, \frac{1}{k})$ for all $k \in [\![1, K]\!]$. Thus, $\mathbf{x} \in \mathbb{R}^N$ and $\mathbf{a} \in \mathbb{R}^K$.

(iii) Calculate input data with the Fourier sine series,

$$v_{i,j}^0 = \sum_{k=1}^{K} a_k \sin(k\pi x_j),$$

for all $i \in [\![1, M]\!]$ and $j \in [\![1, N]\!]$.

(iv) Calculate temporal step sizes

$$\Delta t^i = \frac{\Delta x}{2 \max_j \big(|f'(v_{i,j}^0|\big)},$$

for all $i \in [\![1, M]\!]$.

(v) Calculate target data to be

$$u_{i,j}^1 = v_{i,j}^0 - \frac{\Delta t^i}{\Delta x}\big(F^{\mathrm{God}}\big(v_{i,j}^0, v_{i,j+1}^0\big) - F^{\mathrm{God}}\big(v_{i,j-1}^0, v_{i,j}^0\big)\big),$$

for all $i \in [\![1, M]\!]$ and $j \in [\![1, N]\!]$, where the boundary points with $j = 1, N$ are calculated by treating $v_{i,j}^0$ periodically. The function F^{God} denotes Godunov's flux function in (2.24).

By using this procedure, we obtain a dataset $\mathbb{X}$ as an $M \times 2 \times N$ tensor,

$$\mathbb{X} = \begin{bmatrix} \big[v_{1,1}^0, \ldots, v_{1,N}^0\big] & \big[u_{1,1}^1, \ldots, u_{1,N}^1\big] \\ \vdots & \vdots \\ \big[v_{M,1}^0, \ldots, v_{M,N}^0\big] & \big[u_{M,1}^1, \ldots, u_{M,N}^1\big] \end{bmatrix},$$

where we will forward pass the neighbouring data points $v^0_{i,j}$ and $v^0_{i,j+1}$, for all j, pair-wise through a DNN denoted $\mathcal{N} : \mathbb{R}^2 \to \mathbb{R}$ – thus, we need to pair-wise use the feed-forward algorithm, Algorithm 1, N times in order to finally compute the loss. We emphasize that $v^0_{i,1}$ and $v^0_{i,N}$ are considered neighbouring in the sense of periodicity. The outputs of the network $\mathcal{N}$ will be used as approximations of the flux. Thence, they will be used for calculation of an approximate solution of a scalar conservation law,

$$v^1_{i,j} \coloneqq v^0_{i,j} - \frac{\Delta t^i}{\Delta x}\big(\mathcal{N}\big(v^0_{i,j}, v^0_{i,j+1}; \mathbf{W}, \mathbf{b}\big) - \mathcal{N}\big(v^0_{i,j-1}, v^0_{i,j}; \mathbf{W}, \mathbf{b}\big)\big), \tag{3.7}$$

where $v^1_{i,j}$ will be compared to $u^1_{i,j}$ during the training of $\mathcal{N}$ through the L^1-loss function. The L^1-loss function is defined as

$$C\big(\mathbf{v}^1_i, \mathbf{u}^1_i; \mathbf{W}, \mathbf{b}\big) = \sum_{j=1}^{N} \big|u^1_{i,j} - v^1_{i,j}\big|,$$

where $\mathbf{W}$ and $\mathbf{b}$ denote the weight and bias tensors of $\mathcal{N}$, respectively. This is then followed by a backpropagation algorithm similar to the one given in Algorithm 2. Due to the differences of the loss functions we will perform full derivation of method here as well.

For backpropagation purposes we need to derive the derivative of the loss function with respect to the weights and biases. The derivative with respect to a given weight w^l_{pq} is given by

$$\begin{aligned}\frac{\partial C}{\partial w^l_{pq}} &= \sum_{j=1}^{N} \operatorname{sgn}\big(u^1_{i,j} - v^1_{i,j}\big)\left(-\frac{\partial v^1_{i,j}}{\partial w^l_{pq}}\right) \\ &= \sum_{j=1}^{N} \operatorname{sgn}\big(u^1_{i,j} - v^1_{i,j}\big)\frac{\Delta t^i}{\Delta x}\bigg(\frac{\partial}{\partial w^l_{pq}}\mathcal{N}\big(v^0_{i,j}, v^0_{i,j+1}; \mathbf{W}, \mathbf{b}\big) \\ &\qquad\qquad - \frac{\partial}{\partial w^l_{pq}}\mathcal{N}\big(v^0_{i,j-1}, v^0_{i,j}; \mathbf{W}, \mathbf{b}\big)\bigg),\end{aligned}$$

where we have substituted in the derivative of (3.7). The derivative with respect to the bias is written out in a similar fashion. The only unknown above is $\partial_{w^l_{pq}}\mathcal{N}$, so we will derive this explicitly. As in the introduction to neural networks, Section 2.3.1, we define the inactivated and activated signals going into layer l of $\mathcal{N}$ as

$$\begin{aligned}\mathbf{z}^l &\coloneqq \mathbf{W}^l a^{l-1} + \mathbf{b}^l, \\ \mathbf{a}^l &\coloneqq \varphi_l(\mathbf{z}^l),\end{aligned}$$

where φ_l denotes the activation function of layer l. Component-wise, $\mathbf{z}^l$ and $\mathbf{a}^l$ is written out as

$$\begin{aligned}z^l_q &:= \sum_{p=1}^{n_{l-1}} w^l_{pq} a^{l-1}_p + b^l_q, \\ a^l_q &:= \varphi_l\big(z^l_q\big),\end{aligned}$$

where n_{l-1} denotes the number of nodes in layer $l-1$, a_p^{l-1} is output signal of previous layer (and input in current) and b_q^l is the bias. Assuming that we have L hidden layers in the network, we have that $\mathcal{N} = \mathbf{a}^{L+1}$, and further we let $a_j^{L+1} = \mathcal{N}(v_{i,j-1}^0, v_{i,j}^0)$. Moreover, let $\mathbf{r} = (r_l, r_{l+1}, \ldots, r_{L+1})$ denote a set of indices specifying a neuron path through the network from layer l to $L+1$, where $r_l = q$, $r_{L+1} = j$ and the rest are arbitrarily chosen. Then we have the following calculation for the derivative of the output a_j^{L+1} with respect to a given weight w_{pq}^l.

$$\begin{aligned}
\frac{\partial a_j^{L+1}}{\partial w_{pq}^l} &= \frac{\partial a_j^{L+1}}{\partial a_{r_L}^L}\frac{\partial a_{r_L}^L}{\partial a_{r_{L-1}}^{L-1}}\cdots\frac{\partial a_{r_{l+1}}^{l+1}}{\partial a_q^l}\frac{\partial a_q^l}{\partial w_{pq}^l} \\
&= \left(\frac{\partial a_j^{L+1}}{\partial z_j^{L+1}}\frac{\partial z_j^{L+1}}{\partial a_{r_L}^L}\right)\left(\frac{\partial a_{r_L}^L}{\partial z_{r_L}^L}\frac{\partial z_{r_L}^L}{\partial a_{r_{L-1}}^{L-1}}\right)\cdots\left(\frac{\partial a_{r_{l+1}}^{l+1}}{\partial z_{r_{l+1}}^{l+1}}\frac{\partial z_{r_{l+1}}^{l+1}}{\partial a_q^l}\right)\left(\frac{\partial a_q^l}{\partial z_q^l}\frac{\partial z_q^l}{\partial w_{pq}^l}\right) \\
&= \left(\prod_{m=l}^{L}\frac{\partial a_{r_m}^m}{\partial z_{r_m}^m}\right)\left(\prod_{m=l+1}^{L+1}\frac{\partial z_{r_m}^m}{\partial a_{r_{m-1}}^{m-1}}\right)\frac{\partial a_{r_{L+1}}^{L+1}}{\partial z_{r_{L+1}}^{L+1}}\frac{\partial z_q^l}{\partial w_{pq}^l} \\
&= \left(\prod_{m=l+1}^{L+1}\varphi_{m-1}'\left(z_{q_{m-1}}^{m-1}\right)w_{r_{m-1}r_m}^m\right)\underbrace{\varphi_{L+1}'\left(z_{q_{L+1}}^{L+1}\right)}_{=1}a_p^{l-1} \\
&= \left(\prod_{m=l+1}^{L+1}\varphi_{m-1}'\left(z_{q_{m-1}}^{m-1}\right)w_{r_{m-1}r_m}^m\right)a_p^{l-1}
\end{aligned}$$

Since the last transition in the network consists of an identity activation, we have used that $\varphi_{L+1}' \equiv 1$. Hence, the derivative of the output with respect to a specific weight is dependent on three parts, namely the value a_p^{l-1} traveling along the weight, all weights coming after it and the change of all activations after it. This is the reason why we are backpropagating through the system, since we at each step only have information forward. Similarly, the derivative of a_j^{L+1} with respect to the bias b_q^l is written out as

$$\frac{\partial a_j^{L+1}}{\partial b_q^l} = \prod_{m=l+1}^{L+1}\varphi_{m-1}'\left(z_{q_{m-1}}^{m-1}\right)w_{r_{m-1}r_m}^m.$$

By substituting these results in for the derivative of the loss function above, we obtain

$$\begin{aligned}
\frac{\partial C}{\partial w_{pq}^l} &= \sum_{j=1}^{N}\operatorname{sgn}\left(u_{i,j}^1 - v_{i,j}^1\right)\frac{\Delta t^i}{\Delta x}\left(\frac{\partial a_{j+1}^{L+1}}{\partial w_{pq}^l} - \frac{\partial a_j^{L+1}}{\partial w_{pq}^l}\right), \\
\frac{\partial C}{\partial b_q^l} &= \sum_{j=1}^{N}\operatorname{sgn}\left(u_{i,j}^1 - v_{i,j}^1\right)\frac{\Delta t^i}{\Delta x}\left(\frac{\partial a_{j+1}^{L+1}}{\partial b_q^l} - \frac{\partial a_j^{L+1}}{\partial b_q^l}\right),
\end{aligned}$$

giving us explicit formulas for the derivative of a DNN with respect to the weights and biases. Hence, given a set of activation functions $\{\varphi_l\}_{l=1}^{L+1}$ and a neural network $\mathcal{N}$, we may therefore use this when dealing with the backpropagation during training of the network.

The reason for the superscript indices of input data v and target data u is that this numerical method is designed for the possibility of multiple iterations. That is, the idea is to calulate a new set of data with target values defined by

$$u_{i,j}^2 \coloneqq u_{i,j}^1 - \frac{\Delta t^i}{\Delta x}\big(F^{\text{God}}\big(u_{i,j}^1, u_{i,j+1}^1\big) - F^{\text{God}}\big(u_{i,j-1}^1, u_{i,j}^1\big)\big),$$

which is then compared to a new set of predictions, defined as

$$v_{i,j}^2 \coloneqq v_{i,j}^1 - \frac{\Delta t^i}{\Delta x}\big(\mathcal{N}\big(v_{i,j}^1, v_{i,j+1}^1; \mathbf{W}, \mathbf{b}\big) - \mathcal{N}\big(v_{i,j-1}^1, v_{i,j}^1; \mathbf{W}, \mathbf{b}\big)\big),$$

using the L^1-loss function, $C\big(\mathbf{v}_i^2, \mathbf{u}_i^2; \mathbf{W}, \mathbf{b}\big)$. Likewise, one could iteratively define $u_{i,j}^n$ and $v_{i,j}^n$, for all $n \in \mathbb{N}$. The hope of this iterative approach is to increase stability, and possibly also the convergence of the approximate solution, along with the increase of n. However, due to time limitations and computational resources we have restricted the experiments to $n = 1$.

3.3.2 Baseline for experiments

The baseline for experiments in this section is similar to the one in Section 3.2.2. We are also here interested in performing tests with Burgers' initial-value problem

$$\begin{cases} \partial_t u(x,t) + \partial_x\Big(\frac{u(x,t)^2}{2}\Big) = 0, & (x,t) \in \Omega \times \mathbb{R}_+, \\ u(x,0) = u_0(x), & x \in \Omega, \end{cases} \tag{3.8}$$

with the initial conditions illustrated in Figure 3.1. The discretization parameters, which are identical to previous section, are listed in the upper part

Name	**Parameter**
Spatial domain	$\Omega = [-1, 1]$
Spatial mesh size	$N_x = 50$
Temporal maximum	$T = 0.5$
Courant coefficient	$C_{CFL} = 0.5$
Boundary conditions	Neumann, Periodic
Data distribution	Gauss/Normal
Training data size	$\lvert\mathbb{X}\rvert = 100.000 \times 2 \times 50$
Validation data size	$\lvert\mathbb{X}_V\rvert = 10.000 \times 2 \times 50$
Fourier range	$K = 50$
Batch size	100
Epochs	20
Loss function	$C =$ `torch.nn.L1Loss`
Activation function	$\varphi =$ `torch.relu`
Optimizer	`torch.optim.Adam`
Learning rate	$\kappa = 10^{-3}$
Hidden layer number	$L \in [\![1, 4]\!]$ *
Nodes per layer	$n_l \in [\![4, 1024]\!]$ *

Table 3.8: Parameters used for experiments. *These parameters vary.

of Table 3.8. Further, the DNN specific parameters are listed in the lower part, where the only difference from earlier lies in the choice of loss function, thence also the data structure.

3.3.3 Experiments

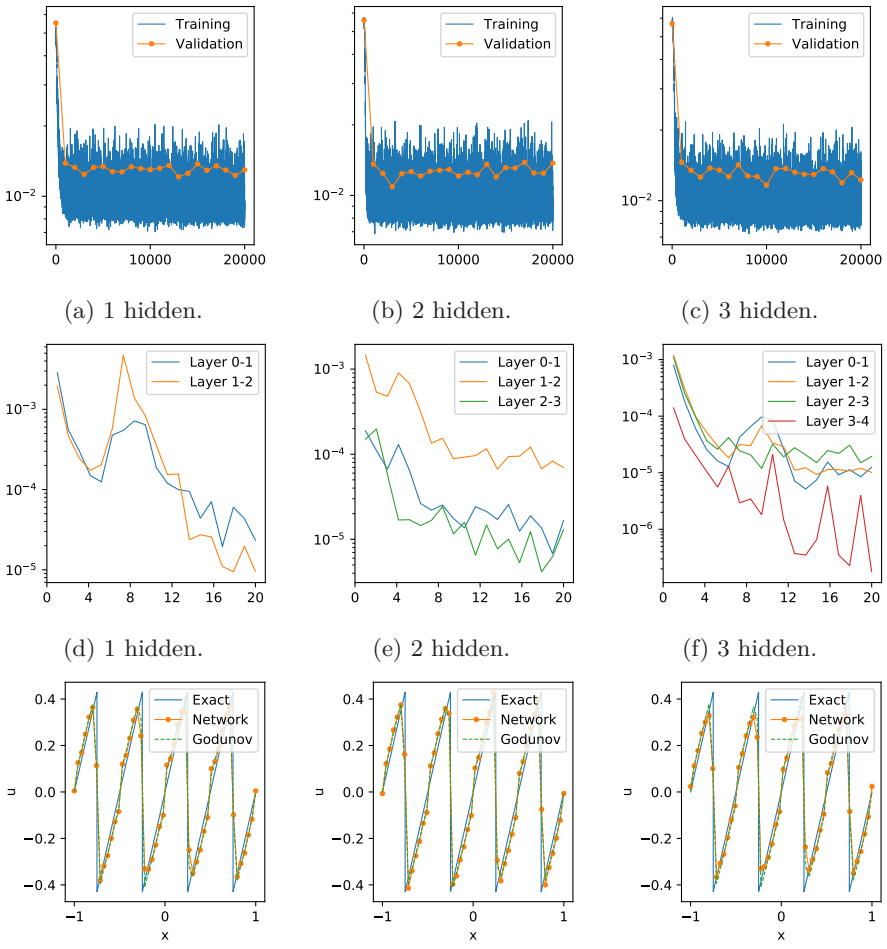
As in the previous section we start by studying how increasing the number of hidden layers affects the solution, and then move on to increasing the number of nodes while keeping the number of hidden layers constant. We will then observe a pattern of behavior by comparing the resulting outputs of relative errors to Tables 3.3 to 3.5.

Similar to Section 3.2.3, the first line of experiments is with varying number of hidden layers, while the total number of nodes is constantly equal to 24. The training and validation loss, as well as the change of weights, is illustrated in Figure 3.7. Contrary to Section 3.2.3, the focus of these figures now lies on the change of weight, while the losses are less informative. The reason why this is the case is that we now work with a loss which is dependent on a total of 50 different results, all at once. What we therefore see in Figures 3.7a to 3.7c is an average loss of multiple outputs, and thus very flattened curves. On the other hand, if we instead glimpse at Figures 3.7d to 3.7f we see that there are adjustments of network happening at every epoch – and more so in the more complex model of 3 hidden layers. The relative errors of these experiments are shown in Table 3.9, with green color highlighting the most accurate models with respect to every initial function. When comparing these values to Table 3.3, we have two main observations, namely that the best choice of the number of layers is the same and that our new model is significantly more stable. The stability is most perceptible with the sine function as initial condition, since the alternating shocks and rarefactions naturally creates a bigger potential for error. This is therefore a strong indication that our new physics-informed L^1-loss is a better choice for models, instead of the more instinctive choice of MSE-loss.

The next batch of experiments to compare previous results with is with varying number of hidden layers and the total number of nodes being constantly equal to 48. As earlier, this is to confirm our hypothesis, being that the models become better with greater depth – as long as the number of nodes is sufficiently large and the training sufficiently good. The losses and changes of weights are illustrated in Figure 3.8, which shows similar conduct as in Figure 3.7. The relative errors of these tests are shown in Table 3.10, of which shows similar fashion as Table 3.4 – it comes to a point where we need to consider

Network	Heaviside	Mirrored	Scaled	Sine
24	3.41×10^{-2}	5.26×10^{-2}	2.58×10^{-2}	9.20×10^{-2}
$12 \bowtie 12$	8.71×10^{-3}	1.59×10^{-3}	1.57×10^{-2}	5.38×10^{-2}
$8 \bowtie 8 \bowtie 8$	3.72×10^{-2}	4.66×10^{-2}	2.91×10^{-2}	1.20×10^{-1}
$6 \bowtie 6 \bowtie 6 \bowtie 6$	1.85×10^{-2}	1.62×10^{-2}	1.95×10^{-2}	7.04×10^{-2}

Table 3.9: Relative Euclidean error of 16 experiments with varying number of layers and 24 nodes. Loss function used is extended L^1-loss. Green cells show the best performing models for each initial condition.



(a) 1 hidden. (b) 2 hidden. (c) 3 hidden.

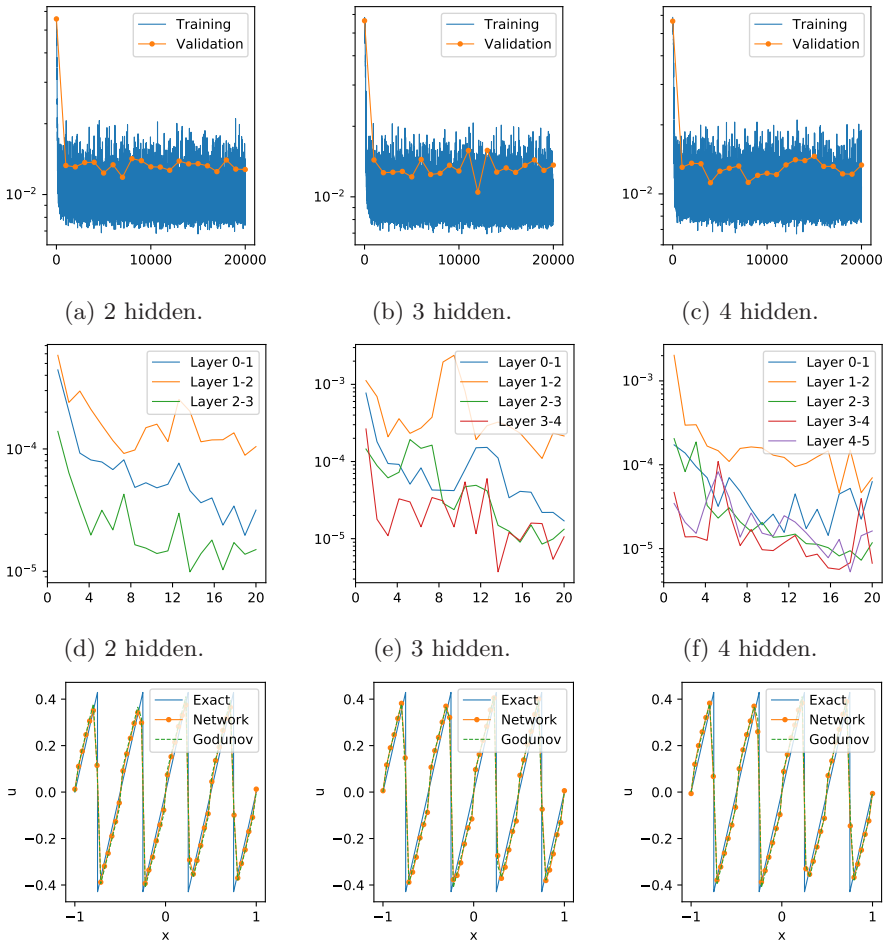
(d) 1 hidden. (e) 2 hidden. (f) 3 hidden.

(g) 1 hidden. (h) 2 hidden. (i) 3 hidden.

Figure 3.7: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initial (lower). Varying number of layers with total of 24 nodes. Loss function used is extended L^1-loss.

Network	Heaviside	Mirrored	Scaled	Sine
48	1.69×10^{-2}	2.35×10^{-2}	1.39×10^{-2}	5.48×10^{-2}
$24 \bowtie 24$	2.83×10^{-2}	4.71×10^{-2}	2.28×10^{-2}	5.92×10^{-2}
$16 \bowtie 16 \bowtie 16$	8.26×10^{-3}	1.18×10^{-2}	1.32×10^{-2}	5.30×10^{-2}
$12 \bowtie 12 \bowtie 12 \bowtie 12$	8.58×10^{-3}	1.68×10^{-3}	9.01×10^{-3}	5.44×10^{-2}

Table 3.10: Relative Euclidean error of 16 experiments with varying number of layers and 48 nodes. Loss function used is extended L^1-loss. Green cells show the best performing models for each initial condition.



(a) 2 hidden. (b) 3 hidden. (c) 4 hidden.

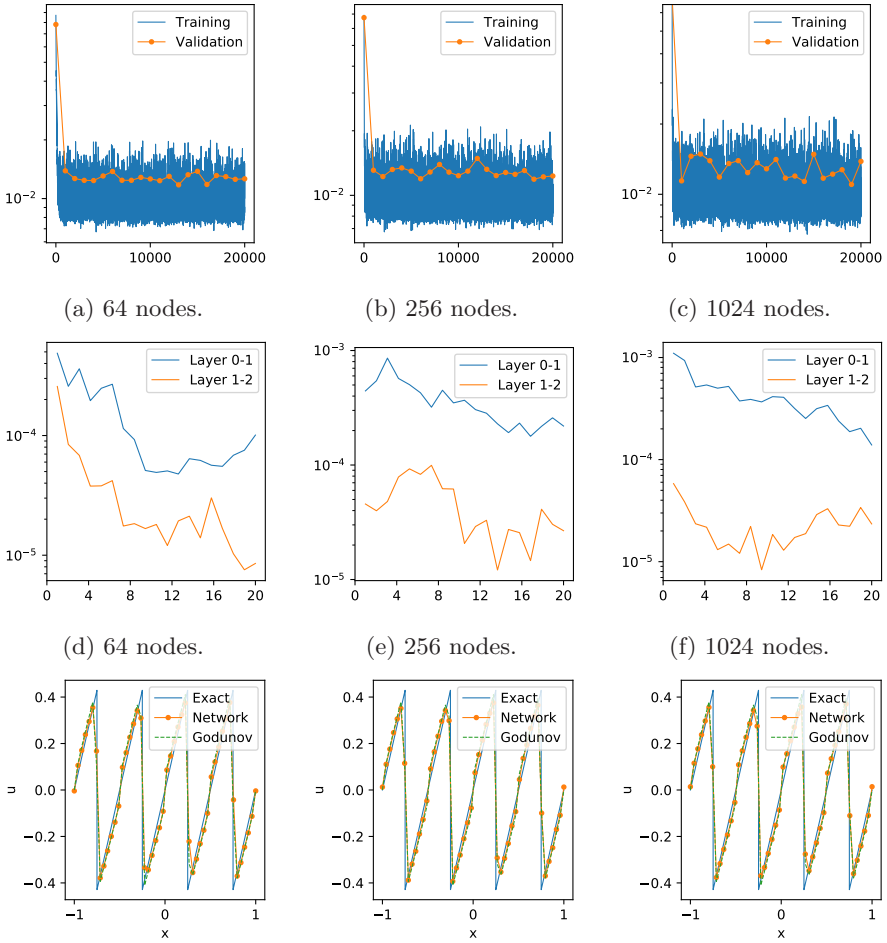
(d) 2 hidden. (e) 3 hidden. (f) 4 hidden.

(g) 2 hidden. (h) 3 hidden. (i) 4 hidden.

Figure 3.8: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initials (lower). Varying number of hidden layers with total of 48 nodes. Loss function used is extended L^1-loss.

not increasing our number of hidden layers any further, as this will result in more instability of the model. As in the first batch of experiments, it shows clear improvements in the error, as well as more stability – especially in the trigonometric case. This is yet another strong indication that we are dealing with a better and more stable choice of loss function, which handles rarefaction waves and shock waves equally good.

Once again, the next batch of experiments consists of comparing our new model to the previous, with respect to increasing the number of nodes, while keeping the total number of layers constant. As before, we therefore perform experiments with 1 hidden layer and 4^i nodes, for $i \in [\![1, 5]\!]$ – the relative errors of all i's are included, but it is not included loss and weight plots for $i = 1, 2$. The losses and changes of weights are illustrated in Figure 3.9. It is quite



(a) 64 nodes. (b) 256 nodes. (c) 1024 nodes.

(d) 64 nodes. (e) 256 nodes. (f) 1024 nodes.

(g) 64 nodes. (h) 256 nodes. (i) 1024 nodes.

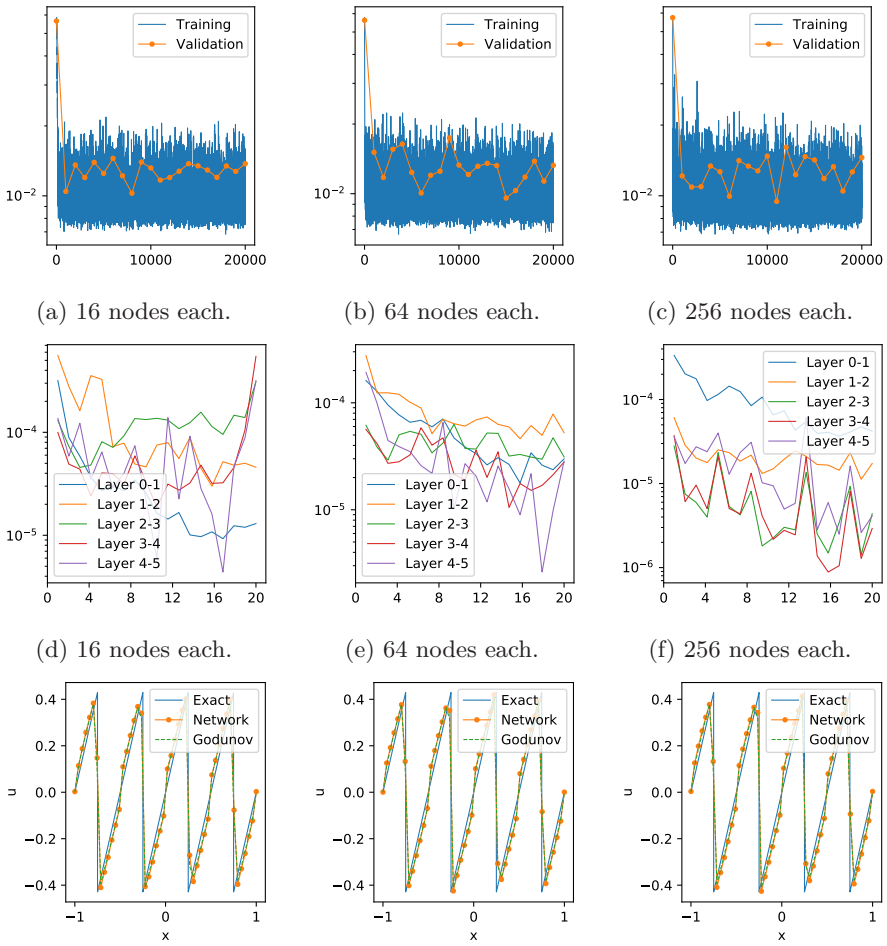
Figure 3.9: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initials (lower). Varying number of nodes with 1 hidden layer. Loss function used is extended L^1-loss.

Network	Heaviside	Mirrored	Scaled	Sine
4	1.70×10^{-1}	3.62×10^{-2}	1.57×10^{-1}	9.55×10^{-1}
16	3.33×10^{-2}	3.30×10^{-2}	2.48×10^{-2}	9.98×10^{-2}
64	3.08×10^{-2}	5.84×10^{-2}	2.41×10^{-2}	9.72×10^{-2}
256	1.92×10^{-2}	2.54×10^{-2}	1.65×10^{-2}	1.06×10^{-1}
1024	4.30×10^{-2}	7.82×10^{-2}	3.83×10^{-2}	7.07×10^{-2}

Table 3.11: Relative Euclidean error of 20 experiments with varying number of nodes and 1 hidden layer. Loss function used is extended L^1-loss. Green cells show the best results for each initial function.

interesting to compare the contrasts of the weight changes with the contrasts of their respective losses. When the two weight-change curves are divergent from one another, we see that the oscillations of the validation losses are more frequent. On the other hand, Figure 3.9d consists of rather similar changes of both sets of weights, and glimpsing at Figure 3.9a we see that this is reflected in the oscillations of the validation loss. The relative errors of these experiments are found in Table 3.11, which supports our already stated conclusion, i.e. the extended L^1-loss outperforms the MSE-loss in both stability and accuracy.

Lastly, we now increase the number of layers to be 4 while varying the number of nodes in each layer. The resulting loss and change of weights are shown in Figure 3.10, while the relative error is shown in Table 3.12. The pattern of these relative errors are fairly similar to earlier observations from



(a) 16 nodes each. (b) 64 nodes each. (c) 256 nodes each.

(d) 16 nodes each. (e) 64 nodes each. (f) 256 nodes each.

(g) 16 nodes each. (h) 64 nodes each. (i) 256 nodes each.

Figure 3.10: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initials (lower). Varying number of nodes with 4 hidden layer. Loss function used is extended L^1-loss.

Network	Heaviside	Mirrored	Scaled	Sine
$4 \gtrless \cdots \gtrless 4$	3.50×10^{-2}	3.33×10^{-2}	3.12×10^{-2}	1.47×10^{-1}
$16 \gtrless \cdots \gtrless 16$	1.72×10^{-2}	2.38×10^{-2}	1.96×10^{-2}	4.99×10^{-2}
$64 \gtrless \cdots \gtrless 64$	1.30×10^{-2}	2.07×10^{-2}	1.18×10^{-2}	3.63×10^{-2}
$256 \gtrless \cdots \gtrless 256$	5.75×10^{-3}	8.58×10^{-4}	1.58×10^{-2}	4.27×10^{-2}

Table 3.12: Relative Euclidean error of 16 experiments with varying number of nodes and 4 hidden layers. Loss function used is extended L^1-loss. Green cells show the best results for each initial function.

previously performed experiments in Section 3.2.3.

As observed above, the new models, trained with the extended L^1-loss, yield very promising results. They have shown themselves significantly more stable across the different initial conditions – this has been most visible when comparing the sine results, as these were the most unstable of the first experiments with regular MSE-loss. These new models require a great deal of computational power during the training, compared to the MSE models. However, when given a pre-trained model, this is obviously not a problem. Summarized, the physics-informed L^1-loss outperforms the MSE-loss in terms of accuracy, with the same degree of efficiency when given pre-trained models. However, the arguments about Godunov's scheme with respect to efficiency – in terms of time complexity – in the conclusion of Section 3.2.3 still applies.

3.3.4 The extended L^1-loss from a convolution point of view

Convolutional Neural Networks – abbreviated CNN – are a special type of ANNs which process data which has a grid-like topology. CNNs are most commonly used within image analysis where one perform convolution on an image using a filter/kernel. However, the setup of the numerical method with extended L^1-loss is rather familiar when comparing it to how a CNN model functions. The weights of any DNN model coincides with the filters of a CNN model, and due to how the training data is structured for our newly developed physics-informed L^1-loss, our input data essentially works in a way that one-dimensional images work in a regular CNN setup. Since the size of our input structure is 2, the algorithm used is therefore equivalent to using a filter of size 2 in a convolutional setting. Due to the equivalence between our DNN models and a CNN, we should be able to increase the efficiency of the implementation sufficiently by setting the DNNs up as convolutional networks. Compared to CNNs where it is common to use iteration of filters through e.g. images, our DNN works as a set of filters, while the dataset works as a set of one-dimensional images. Thus, by looking at the method from a convolutional perspective it may be possible to increase efficiency to a degree where it can compete with Godunov's method.

3.4 DNN solver with MSE-loss in $\mathbb{R}^2$

In this section we will present the main result of the thesis, being the development of a new numerical method. The aim of the method is to approximate solutions

of a two-dimensional initial-value problem, written out as

$$\begin{cases} u_t + f(u)_x + g(u)_y = 0, & (x, y, t) \in \Omega \times \mathbb{R}_+, \\ u(x, y, 0) = u_0(x, y), & (x, y) \in \Omega. \end{cases} \tag{3.9}$$

Exact solution formulas for such problems do not exist, but we may create reference solutions by considering a spatial grid with high resolution. The finer the resolution is, the better the approximation will be. Fine-mesh solvers based on high resolution are extremely time consuming, and our goal is therefore to design a numerical method which may be pre-trained using high resolution, yielding an efficient method only dependent on a coarse spatial mesh.

We will start by deriving the general form of a two-dimensional finite-volume scheme, followed by the derivation of a fine-mesh algorithm. When this is in place, we turn our attention toward the DNN based numerical method and describe how to create datasets and implement such a method. Finally, we will test the method by first comparing results to Section 3.2.3 by setting the vertical axis constant, and then test the method on genuine two-dimensional initial-value problems.

3.4.1 Two-dimensional finite-volume method

We will now derive the general finite-volume scheme for approximating solutions of Equation (3.9). As for any introduction to a numerical method, we will start by defining our discrete temporal domain and spatial domain. Consider the domain to be $[x_L, x_R] \times [y_L, y_R] \times \mathbb{R}_+$ and define step sizes of the spatial dimensions to be

$$\Delta x = \frac{x_R - x_L}{N_x + 1}, \quad \Delta y = \frac{y_R - y_L}{N_y + 1}.$$

Further, as for the one-dimensional method introduced in Section 2.2.2, we define the discrete points to be

$$x_i = x_L + \left(i + \frac{1}{2}\right)\Delta x,$$
$$y_j = y_L + \left(j + \frac{1}{2}\right)\Delta y,$$

for all $i \in [\![0, N_x]\!]$ and $j \in [\![0, N_y]\!]$. Thence, we define the midpoints as

$$x_{i-1/2} = x_L + i\Delta x,$$
$$y_{j-1/2} = y_L + j\Delta y,$$

for all $i \in [\![0, N_x + 1]\!]$ and $j \in [\![0, N_y + 1]\!]$. Furthermore, similar to the one-dimensional methodology, we define the points in time as

$$t^n := t^{n-1} + \Delta t^n, \quad \forall n = 1, 2, \ldots,$$

where Δt^n denotes a dynamically changing temporal step size, while $t^0 = 0$. Moreover, the midpoints are the basis for the computational cells, defined as

$$C_{ij}^n = [x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}] \times [t^n, t^{n+1}],$$
$$C_{ij}^* = [x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}],$$

$$C^n_{i*} = [x_{i-1/2}, x_{i+1/2}] \times [t^n, t^{n+1}],$$
$$C^n_{*j} = [y_{j-1/2}, y_{j+1/2}] \times [t^n, t^{n+1}].$$

Be aware that we in this case use $*$ as notation for dimension exclusion, and not to indicate that we consider all values within the dimension. The integral form of (3.9) is written out as

$$\begin{aligned}
0 &= \iiint_{C^n_{ij}} u_t + f(u)_x + g(u)_y \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}t, \\
&= \iint_{C^*_{ij}} u(x,y,t^{n+1}) - u(x,y,t^n)\,\mathrm{d}x\,\mathrm{d}y \\
&\quad + \iint_{C^n_{*j}} f(u(x_{i+1/2},y,t)) - f(u(x_{i-1/2},y,t))\,\mathrm{d}y\,\mathrm{d}t \\
&\quad + \iint_{C^n_{i*}} g(u(x,y_{j+1/2},t)) - g(u(x,y_{j-1/2},t))\,\mathrm{d}x\,\mathrm{d}t.
\end{aligned}$$

Following this, we define

$$\begin{aligned}
\bar{F}^n_{i+1/2,j} &\coloneqq \frac{1}{\Delta y \Delta t^{n+1}} \iint_{C^n_{*j}} f(u(x_{i+1/2},y,t))\,\mathrm{d}y\,\mathrm{d}t, \\
\bar{G}^n_{i,j+1/2} &\coloneqq \frac{1}{\Delta x \Delta t^{n+1}} \iint_{C^n_{i*}} g(u(x,y_{j+1/2},t))\,\mathrm{d}x\,\mathrm{d}t, \\
u^{n+1}_{ij} &\coloneqq \frac{1}{\Delta x \Delta y} \iint_{C^*_{ij}} u(x,y,t^{n+1})\,\mathrm{d}x\,\mathrm{d}y,
\end{aligned}$$

and consequently obtain

$$\begin{aligned}
0 = \Delta x \Delta y \big(u^{n+1}_{ij} - u^n_{ij}\big) &+ \Delta y \Delta t^{n+1} \Big(\bar{F}^n_{i+1/2,j} - \bar{F}^n_{i-1/2,j}\Big) \\
&+ \Delta x \Delta t^{n+1} \Big(\bar{G}^n_{i,j+1/2} - \bar{G}^n_{i,j-1/2}\Big),
\end{aligned}$$

which leads to

$$\begin{aligned}
u^{n+1}_{ij} = u^n_{ij} &- \frac{\Delta t^{n+1}}{\Delta x}\Big(\bar{F}^n_{i+1/2,j} - \bar{F}^n_{i-1/2,j}\Big) \\
&- \frac{\Delta t^{n+1}}{\Delta y}\Big(\bar{G}^n_{i,j+1/2} - \bar{G}^n_{i,j-1/2}\Big).
\end{aligned} \tag{3.10}$$

This equation is the general form of any two-dimensional finite-volume method for approximating solutions to two-dimensional scalar conservation laws. It is a statement of conservation with respect to one cell average given by the difference in fluxes across the interfaces of the cells.

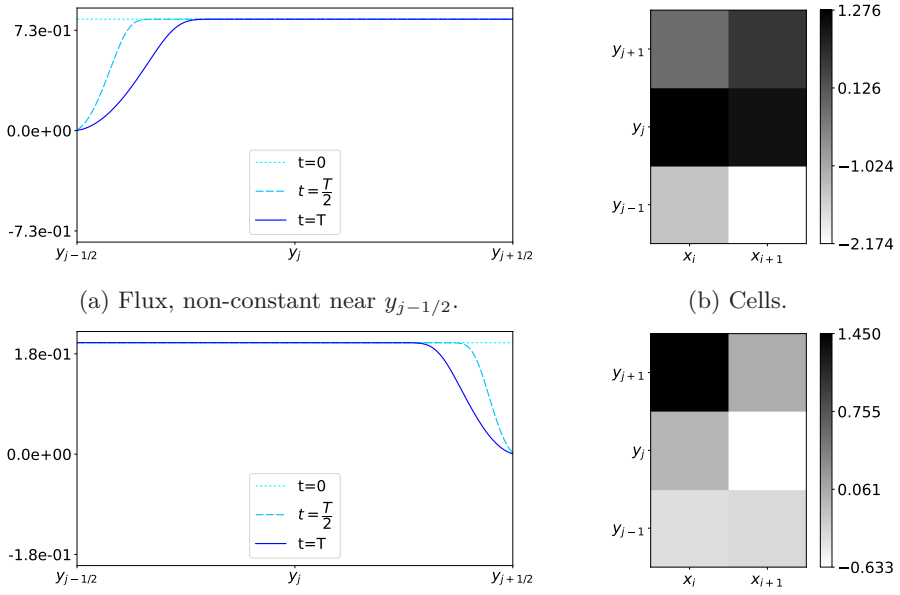
3.4.2 DNN based finite-volume method

Both $\bar{F}$ and $\bar{G}$ in (3.10) need a priori knowledge of the exact solution. Consequently, it does not exist exact solution formulas for $\bar{F}$ and $\bar{G}$, and we will now derive the fine-mesh algorithm for approximating these fluxes,

which forms the basis for our new DNN based method. The problem we are faced with may be formulated as a Riemann problem,

$$u_t + \nabla \cdot \begin{pmatrix} f(u) \\ g(u) \end{pmatrix} = 0, \quad (x, y, t) \in C_{ij}^n,$$
$$u(x, y, t^n) = \begin{cases} u_{i,j}^n & x < x_{i+1/2}, y < y_{j+1/2}, \\ u_{i+1,j}^n & x > x_{i+1/2}, y < y_{j+1/2}, \\ u_{i,j+1}^n & x < x_{i+1/2}, y > y_{j+1/2}, \\ u_{i+1,j+1}^n & x > x_{i+1/2}, y > y_{j+1/2}, \end{cases} \tag{3.11}$$

where we aim at approximating the solution $u = u(x, y, t)$ in a given computational cell $C_{i,j}^n$ – i.e we want to approximate the flux going through the horizontal point $x = x_{i+1/2}$ and vertical point $y = y_{j+1/2}$. The idea of the fine-mesh algorithm is to break every cell of our coarse discretization down to a high-resolution mesh, and then use a one-dimensional numerical scheme in both spatial directions. Thereafter, we will sum all flux values of interest, i.e. all flux values propagating through the cell interface of the coarse mesh. We will explain the implementation of both a data generator and a fine mesh algorithm. However, a question arises when implementing a two-dimensional fine-mesh solver like this: why are we in need of finer grids to properly approximate the solution of a two-dimensional Riemann problem? The answer to this lies in the behavior of the flux across a cell interface. As we may see from Figure 3.11, the



(a) Flux, non-constant near $y_{j-1/2}$.

(b) Cells.

(c) Flux, non-constant near $y_{j+1/2}$.

(d) Cells.

Figure 3.11: An illustration of flux movement in the horizontal direction, at cell interface $x_{i+1/2}$ along $[y_{j-1/2}, y_{j+1/2}]$. Similar behavior with non-constant flux near four-cell intersections applies for flux in the vertical direction. (a) and (c) show the flux magnitude, while (b) and (d) show the respective magnitudes of cell averages.

flux becomes non-constant close to a four-cell intersection – meaning that if we calculate the flux based on the cell averages on each side of an interface, we will loose a significant portion of information, leading to significant errors in the approximations. The reason for this non-constant behavior is the effect caused by the upper and lower cell averages in Figures 3.11b and 3.11d. This motivates the need of an algorithm which considers the flux across the cell interfaces in a piecewise matter.

To properly approximate solutions of (3.11), we need to consider a total of 8 cell averages, constituting two Riemann problems. That is, 6 cells neighbouring the interface of interest in the horizontal direction, and 6 cells neighbouring the interface of interest in the vertical direction, with 4 common cells overlapping. The relevant cells are illustrated in Figure 3.12a, with red and yellow arrows representing the horizontal and vertical fluxes we want to approximate, respectively. The blue lines show the cells to be considered for approximations of the horizontal flux. Similarly, the green cells will be considered for the vertical flux approximation. As already discussed, we want to approximate these fluxes piecewise, by dividing each cell into a finer mesh-grid, illustrated for horizontal flux in Figure 3.12b – the red arrows show the piecewise parts of the flux across the cell interface.

The rest of this section will be used to fully derive the numerical method in details. We start by introducing a fine-mesh solver, which is an algorithm for approximating the numerical fluxes $\bar{F}$ and $\bar{G}$, given in (3.10). Then, we describe how this algorithm will be used with respect to DNNs, and furthermore, how to create the training data for our DNN models. Lastly, we will look back at how to obtain the final results, being the numerical approximate solutions of (3.9).

We will now derive a numerical method for solving two-dimensional Riemann problems on the form given in (3.11). We start by considering the discretization derived in Section 3.4.1 for a general finite-volume scheme. Thus, we have a uniformly spaced mesh grid of the two-dimensional spatial domain, together with a dynamically changing temporal domain, i.e. varying step sizes Δt^n for

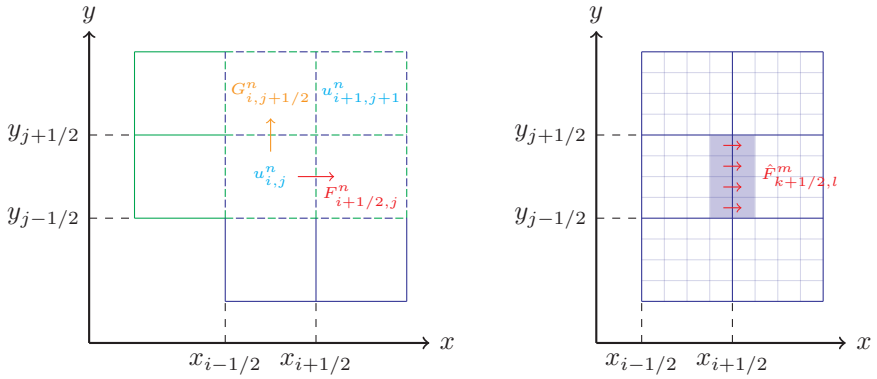


Figure 3.12: Mesh grid and flux in two spatial dimensions. (a) illustrates the relevant cells and fluxes in both directions. (b) illustrates the fine mesh for flux calculation in the horizontal direction.

all temporal points $t^0, t^1, \dots$ up to some chosen maximum time T. Further, we ensure that the temporal step sizes are bounded by a two-dimensional CFL-condition,

$$\Delta t^n \leq C_{CFL} \frac{\min\{\Delta x, \Delta y\}}{\max_{i,j}\left(\left\|\mathbf{f}'\left(u_{i,j}^n\right)\right\|_2\right)}, \tag{3.12}$$

where $C_{CFL} \in (0,1)$ and $\mathbf{f}'(u) = (f'(u), g'(u))$. Moreover, we define domains

$$\begin{aligned} D_{\mathrm{F}} &\coloneqq [x_{i-1/2}, x_{i+3/2}] \times [y_{j-3/2}, y_{j+3/2}] \times [t^n, t^{n+1}], \\ D_{\mathrm{G}} &\coloneqq [x_{i-3/2}, x_{i+3/2}] \times [y_{j-1/2}, y_{j+3/2}] \times [t^n, t^{n+1}], \end{aligned}$$

which we will use for approximating fluxes f and g of (3.11), respectively – these domains are the areas illustrated in Figure 3.12a. The next step is now discretization of D_{F} and D_{G} in a similar matter as in previous section. Let the resulting variable notations be as earlier but with a 'hat' notation, i.e. $\Delta\hat{x}$ and $\Delta\hat{y}$ for spatial step sizes, $\Delta\hat{t}^m$ for dynamically changing temporal step sizes, etc. Further, we let $\hat{u}_{k,l}^0$ equal the corresponding cell average from the coarse domain, e.g. $\hat{u}_{k,l}^0 = u_{i,j}^n$ if $(\hat{x}_k, \hat{y}_l) \in C_{i,j}^*$. We also enforce a CFL-condition on the fine mesh-grid by choosing temporal step sizes satisfying

$$\Delta\hat{t}^m \leq C_{CFL} \frac{\min\{\Delta\hat{x}, \Delta\hat{y}\}}{\max_{k,l}\left(\left\|\mathbf{f}'\left(\hat{u}_{k,l}^m\right)\right\|_2\right)}.$$

Furthermore, let the maximum temporal step be chosen uniformly from interval

$$\hat{T} \in \left(0, C_{CFL} \frac{\min\{\Delta\hat{x}, \Delta\hat{y}\}}{\max_{k,l}\left(\left\|\mathbf{f}'\left(\hat{u}_{k,l}^0\right)\right\|_2\right)}\right). \tag{3.13}$$

There are mainly two reasons for choosing $\hat{T}$ arbitrarily in this interval, first one being that we ensure stability by bounding $\hat{T}$ with respect to the CFL-condition. The second reason is that we are going to train a DNN model to approximate the flux, so by choosing $\hat{T}$ randomly we may train the network to generalize over the temporal parameter. Next, we will calculate the numerical fluxes in both spatial directions, separately, given by Godunov's formula,

$$\hat{F}_{k+1/2,l}^m = \begin{cases} \min\limits_{\hat{u}_{k,l}^m \leq \theta \leq \hat{u}_{k+1,l}^m} f(\theta) & \text{if } \hat{u}_{k,l}^m \leq \hat{u}_{k+1,l}^m \\ \max\limits_{\hat{u}_{k+1,l}^m \leq \theta \leq \hat{u}_{k,l}^m} f(\theta) & \text{if } \hat{u}_{k,l}^m > \hat{u}_{k+1,l}^m \end{cases} \quad \forall k \in [\![0, \hat{N}_x + 1]\!], l \in [\![0, \hat{N}_y]\!],$$

$$\hat{G}_{k,l+1/2}^m = \begin{cases} \min\limits_{\hat{u}_{k,l}^m \leq \theta \leq \hat{u}_{k,l+1}^m} g(\theta) & \text{if } \hat{u}_{k,l}^m \leq \hat{u}_{k,l+1}^m \\ \max\limits_{\hat{u}_{k,l+1}^m \leq \theta \leq \hat{u}_{k,l}^m} g(\theta) & \text{if } \hat{u}_{k,l}^m > \hat{u}_{k,l+1}^m \end{cases} \quad \forall k \in [\![0, \hat{N}_x]\!], l \in [\![0, \hat{N}_y + 1]\!].$$

Lastly, it remains to sum up all flux values of interest, illustrated in Figure 3.12b. This yields good approximations of the fluxes illustrated in Figure 3.12a. The areas of interest with respect to domains D_{F} and D_{G}, are the sub-domains

$$\begin{aligned} d_{\mathrm{F}} &\coloneqq \{x_{i+1/2}\} \times C_{*j}^n \subset D_{\mathrm{F}}, \\ d_{\mathrm{G}} &\coloneqq \{y_{j+1/2}\} \times C_{i*}^n \subset D_{\mathrm{G}}. \end{aligned}$$

Let $\mathbb{I}_{\mathrm{F}}$ and $\mathbb{I}_{\mathrm{G}}$ denote the sets of all indices (k, l, m) such that

$$(\hat{x}_{k+1/2}, \hat{y}_l, \hat{t}^m) \in d_{\mathrm{F}},$$
$$(\hat{x}_k, \hat{y}_{l+1/2}, \hat{t}^m) \in d_{\mathrm{G}},$$

respectively. This yields the final flux approximations

$$F^n_{i+1/2,j} = \frac{1}{|\mathbb{I}_{\mathrm{F}}|} \sum_{(k,l,m)\in\mathbb{I}_{\mathrm{F}}} \hat{F}^m_{k+1/2,l}, \tag{3.14}$$

$$G^n_{i,j+1/2} = \frac{1}{|\mathbb{I}_{\mathrm{G}}|} \sum_{(k,l,m)\in\mathbb{I}_{\mathrm{G}}} \hat{G}^m_{k,l+1/2}. \tag{3.15}$$

By substituting these approximations into (3.10), we obtain

$$u^{n+1}_{ij} = u^n_{ij} - \frac{\Delta t^{n+1}}{\Delta x}\Big(F^n_{i+1/2,j} - F^n_{i-1/2,j}\Big) - \frac{\Delta t^{n+1}}{\Delta y}\Big(G^n_{i,j+1/2} - G^n_{i,j-1/2}\Big),$$

which is a valid and explicit numerical method for approximating (3.11). This fine-mesh solver algorithm for the horizontal direction is given as pseudocode in Algorithm 4 – the vertically directed flux may be obtain in similar manner. This method is what we use to create high-resolution reference solutions.

Algorithm 4 Fine-mesh solver in x-direction

Initialize ordinary grid setup, fluxes and initial condition.
Pass inputs

- u_0 – six neighbouring cells,
- $\hat{N}_x, \hat{N}_y$ – fine-mesh size,
- $\hat{T}$ – Δt^n in coarse mesh.

Initialize fine mesh as $2\hat{N}_x \times 3\hat{N}_y$ matrix.
while $t < \hat{T}$ **do**
  Update $\Delta\hat{t}^m$ wrt. CFL-condition.
  Godunov's one-dimensional scheme in both spatial directions.
  Update $t \leftarrow t + \Delta\hat{t}^m$.
Extract and sum up the final flux within sub-domain C^n_{*j} at interface $x_{i+1/2}$.

The derived numerical fluxes in (3.14) and (3.15) are good numerical approximations, given that we use a fine 'enough' mesh size. However, this algorithm is not computationally effective. Thus, we aim at creating training data for a DNN, using Algorithm 4, and the hope is then to obtain an efficient and precise numerical method for solving general two-dimensional initial-value problems using a pre-trained DNN model. As seen from the derivation of the fine-mesh algorithm, we are in need of 6 neighbouring cell averages in order to sufficiently approximate a flux travelling across a specified cell interface, defined by the Riemann problem in (3.11). Due to the multiple choices we have of $\hat{T}$ in (3.13), the network should also be dependent on a seventh temporal parameter. Thus, we are in need of two DNNs, both with 7 input parameters and 1 output parameter, to approximate $\bar{F}$ and $\bar{G}$. Thence, each row in the datasets should consist of 8 data points, namely the 7 mentioned input data

points, together with an eighth target parameter, which will tell the networks how well they performed during the training process. Let $\mathcal{N}_f$ and $\mathcal{N}_g$ denote the DNNs used for the horizontal and vertical dimensions, respectively. Their respective datasets are written out as

$$\mathbb{X}_f = \begin{bmatrix} X^f_{1,1} & \dots & X^f_{1,7} & Y^f_1 \\ \vdots & \vdots & \vdots & \vdots \\ X^f_{M_f,1} & \dots & X^f_{M_f,7} & Y^f_{M_f} \end{bmatrix}, \qquad \mathbb{X}_g = \begin{bmatrix} X^g_{1,1} & \dots & X^g_{1,7} & Y^g_1 \\ \vdots & \vdots & \vdots & \vdots \\ X^g_{M_g,1} & \dots & X^g_{M_g,7} & Y^g_{M_g} \end{bmatrix},$$

where M_f and M_g denotes the number of data points in each set. The pseudocode for the data generator in horizontal dimension is given in Algorithm 5. The training process will be performed using the feed-forward

Algorithm 5 Data generator in horizontal dimension

Initialize flux function f.
Select $M_f \times 6$ random numbers, standard normally distributed,

$$X^f_{i,1}, \dots, X^f_{i,6}, \quad \forall i \in [\![1, M_f]\!].$$

Hard code Δx and Δy to desired values.
for $i = 1, \dots, M_f$ **do**
  Set $\hat{T}$ uniformly from interval $\left(0, C_{CFL}\frac{\min\{\Delta x, \Delta y\}}{\max_j(\|f'(X^f_{i,j})\|_2)}\right)$, yielding $X^f_{i,7}$.
  Perform Algorithm 4 with scaled temporal value $\frac{\hat{T}}{\min\{\Delta x, \Delta y\}}$.
  Extract flux of interest and calculate mean value, yielding Y^f_i.

algorithm, Algorithm 1, with backpropagation, Algorithm 2, using the MSE-loss function, see Sections 2.3.1 and 2.3.2. By defining

$$\mathbf{U}^n_{i+1/2,j} = \left(u^n_{i,j-1}, u^n_{i+1,j-1}, u^n_{i,j}, u^n_{i+1,j}, u^n_{i,j+1}, u^n_{i+1,j+1}, \frac{\Delta t^{n+1}}{\min\{\Delta x, \Delta y\}}\right),$$
$$\mathbf{U}^n_{i,j+1/2} = \left(u^n_{i-1,j}, u^n_{i-1,j+1}, u^n_{i,j}, u^n_{i,j+1}, u^n_{i+1,j}, u^n_{i+1,j+1}, \frac{\Delta t^{n+1}}{\min\{\Delta x, \Delta y\}}\right),$$

and substituting pre-trained $\mathcal{N}_f$ and $\mathcal{N}_g$ into (3.10), we obtain a formula for the approximate solution of (3.9),

$$u^{n+1}_{i,j} = u^n_{i,j} - \frac{\Delta t^{n+1}}{\Delta x}\left(\mathcal{N}_f\left(\mathbf{U}^n_{i+1/2,j}\right) - \mathcal{N}_f\left(\mathbf{U}^n_{i-1/2,j}\right)\right) - \frac{\Delta t^{n+1}}{\Delta y}\left(\mathcal{N}_g\left(\mathbf{U}^n_{i,j+1/2}\right) - \mathcal{N}_g\left(\mathbf{U}^n_{i,j-1/2}\right)\right),$$

which completes the derivation of our numerical method.

3.4.3 Baseline for experiments

In the following section we aim at testing the DNN based numerical method for approximating solutions of Burgers' two-dimensional initial-value problems,

written out as

$$\begin{cases} \partial_t u + \partial_x\left(\frac{u^2}{2}\right) + \partial_y\left(\frac{u^2}{2}\right) = 0, & (x,y,t) \in \Omega \times \mathbb{R}_+, \\ u(x,y,0) = u_0(x,y), & (x,y) \in \Omega. \end{cases} \tag{3.16}$$

The spatial domain is similar to the one in the one-dimensional problems, being $\Omega = [-1,1] \times [-1,1]$. We use spatial mesh sizes of $N_x = 50$ and $N_y = 50$, giving us mesh points

$$\begin{aligned} x_i &= -1 + \left(i + \frac{1}{2}\right)\Delta x, \qquad \Delta x = \frac{2}{51}, \\ y_j &= -1 + \left(j + \frac{1}{2}\right)\Delta y, \qquad \Delta y = \frac{2}{51}. \end{aligned}$$

We will also here induce a CFL-condition with Courant number $C_{CFL} = 1/2$, by defining

$$\Delta t^n := \frac{\min\{\Delta x, \Delta y\}}{2 \max_{i,j}\left(\left\|\mathbf{f}'\left(u_{i,j}^n\right)\right\|_2\right)} = \frac{\min\{\Delta x, \Delta y\}}{2 \max_{i,j}\left(\sqrt{2}\left|u_{i,j}^n\right|\right)},$$

for all n, where we have used that $\mathbf{f}'(u) = (f'(u), g'(u))$ with $f = g$. Moreover, the maximum time of propagation is set to $T = 0.5$. When performing the fine-mesh algorithm, Algorithm 4, we use spatial mesh sizes $\hat{N}_x = 50$ and $\hat{N}_y = 50$.

The first experiments will be performed using initial conditions similar to what we presented in Section 3.2.2, see Figure 3.1. The first initial function is the Heaviside function in the horizontal direction, with y set to be constant,

$$u_0(x,y) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{3.17}$$

The next function is the additive inverse of Heaviside, given as

$$u_0(x,y) = \begin{cases} 1 & \text{if } x < 0, \\ 0 & \text{if } x > 0. \end{cases} \tag{3.18}$$

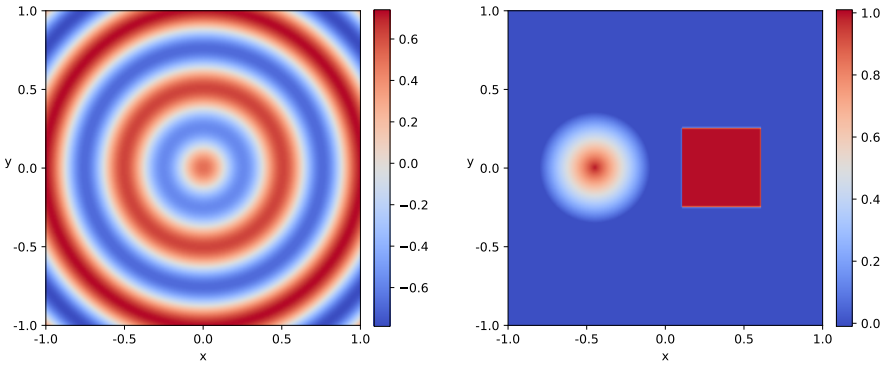
Then we will perform tests using the scaled and shifted version of Heaviside, defined by

$$u_0(x,y) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{3.19}$$

Lastly, we will test with the continuous sine function given by

$$u_0(x,y) = \sin(4\pi x). \tag{3.20}$$

When performing tests on these initial conditions we will compare the results with what we got from the experiments in Section 3.2.3, to ensure that our numerical method is implemented correctly. When we have confirmed that the two-dimensional numerical method is implemented correctly, we will perform further testing on genuine two-dimensional input data, i.e. the y-axis will no



(a) Sigmoid–cosine function.

(b) LeVeque function

Figure 3.13: Authentic two-dimensional initial conditions used for experiments.

longer be kept constant. The first authentic two-dimensional initial function used is a multiplication of a sigmoid function and a cosine function, and is written out as

$$r(x,y) \coloneqq \sqrt{x^2+y^2},$$
$$u_0(x,y) = \frac{e^r}{1+e^r}\cos(4\pi r),$$

illustrated in Figure 3.13a. We will refer to this as the *Sigmoid–cosine function*. The last initial condition used is inspired by R. J. LeVeque [LeV02], and is built up of two regions where one contains a cone structure while the other consists of a two-dimensional step function. The formula for LeVeque's initial condition is

$$r(x,y) \coloneqq \sqrt{(x+0.45)^2+y^2},$$
$$u_0(x,y) = \begin{cases} 1 & \text{if } (x,y) \in (0.1,0.6)\times(-0.25,0.25), \\ 1-\frac{r}{0.35} & \text{if } r < 0.35, \\ 0 & \text{otherwise,} \end{cases}$$

illustrated in Figure 3.13b. This will be referred to as the *LeVeque function*. The advantage of the LeVeque function is that we are able to observe how different objects in the spatial domain may interact with one another.

The boundary conditions of the four former initial conditions are as in Section 3.2.2, i.e. Neumann for the three first and periodic for the latter. A similar mindset is used when choosing the boundary conditions for Sigmoid–cosine function and LeVeque function. The Sigmoid–cosine function is of trigonometric nature, thus we have chosen to use a periodic boundary condition,

$$u(-1,y,t^n) = u(1,y,t^n), \qquad \forall y \in (-1,1),$$
$$u(x,-1,t^n) = u(x,1,t^n), \qquad \forall x \in (-1,1),$$

for all temporal points t^n. On the other hand, the LeVeque function is tested with a Neumann boundary condition, defined as

$$\partial_x u(x, y, t^n) = \partial_x u\big(x, y, t^{n-1}\big), \qquad x \in \{-1, 1\}, \forall y \in (-1, 1),$$
$$\partial_y u(x, y, t^n) = \partial_y u\big(x, y, t^{n-1}\big), \qquad \forall x \in (-1, 1),\ y \in \{-1, 1\},$$

for all $n \geq 1$.

Just like in Section 3.2.2, we have some choices with respect to the machine learning implementations. These are given in Table 3.13, being identical to the parameters from the one-dimensional solver with the exception of the dataset dimensions.

Name	**Parameter**
Data distribution	Gauss/Normal
Training data size	$\lvert\mathbb{X}\rvert = 100.000 \times 8$
Validation data size	$\lvert\mathbb{X}_V\rvert = 10.000 \times 8$
Batch size	100
Epochs	20
Loss function	$C =$ `torch.nn.MSELoss`
Activation function	$\varphi =$ `torch.relu`
Optimizer	`torch.optim.Adam`
Learning rate	$\kappa = 10^{-3}$
Hidden layer number	$L \in [\![1, 4]\!]$ *
Nodes per layer	$n_l \in [\![4, 1024]\!]$ *

Table 3.13: DNN parameters for two-dimensional experiments. *These vary.

3.4.4 Experiments

The first batches of experiments we will perform uses one-dimensional initial conditions so that we are able to verify that the solver is implemented correctly in terms of the one-dimensional solver – this will be done by observing the patterns of relative errors and comparing these to the first one-dimensional experiments. When we are somewhat sure that the implementation works well for one-dimensional problems, we will move on to tests with genuine two-dimensional problems. Since the network takes 7 inputs – compared to 2 inputs for the one-dimensional solvers – we could expect greater error during the training. This is due to the fact that the network needs to learn and recognize sufficiently greater number of permutations of the input values. To reduce this difference of performance one could

- increase the size of training data,
- increase the number of epochs,
- increase, decrease or use dynamically changed batch size,

or a combination of these. We will not do any of these things when testing that the implementation is correct, the reason being that we need to keep most parameters constant to make sure that comparison with the one-dimensional

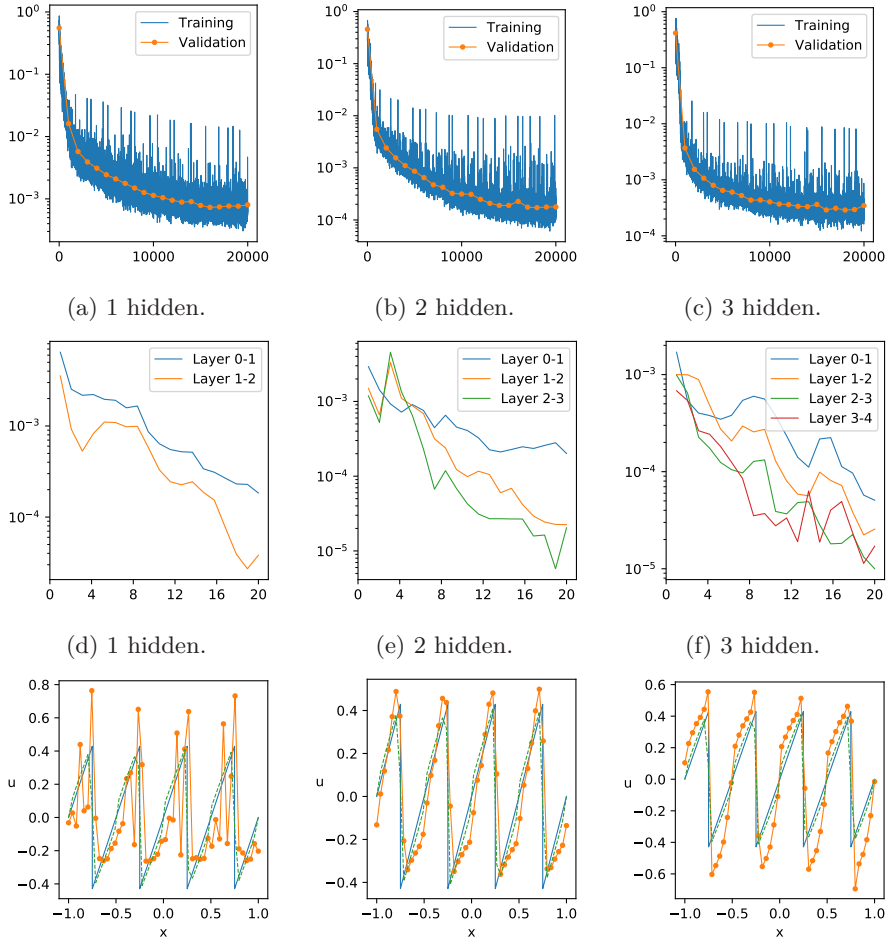
solver is not biased. As seen from Section 2.3.5, the universal approximation theorem states that we need to increase width or depth of the network to be able to reproduce the results from the one-dimensional experiments, see Theorem 2.3.1. Therefore, we could expect greater relative error in all experiments performed in this section, given that we compare network results produced with networks of same width and depth. Compared to the experiments in Sections 3.2.3 and 3.3.3, the implementations of these two-dimensional solvers yield some additional sources of error. The reason for this lies in the implementation differences, where we now must consider a more complex structure of code – we now rely on estimates produced by a fine-mesh solver. The fine-mesh solver contains some parameters of choice of which must be chosen carefully to be able to yield sufficiently small errors in the result. One of the most important parameters in this case is the choice of size for the fine mesh. If this mesh size is chosen poorly we will end up with unsatisfactory target values, and consequently a deficient DNN model.

We start by performing tests with the total number of nodes being 24, and the number of hidden layers varying from 1 to 4. The change of weights and losses during training, as well as results, are illustrated in Figure 3.14 – not including figures for 4 layers. By comparing these figures to Figure 3.2, we see similar tendencies in the training process. However, the results of Figures 3.14g to 3.14i indicates that the network lacks precision relative to the one-dimensional models. This observation is clearer when looking at Table 3.14. By comparing these relative errors to those of Table 3.3, we can see that the network is best with 2 hidden layers here as well. Nevertheless, this is just an indication that the implementation may work correctly. In this case we must actually look back at Figure 3.14 to verify that the network does what we intended it to do, within reasonable range of error. As already observed, we have reasonable size of the relative errors, and the sine waves seem to adjust well along with the Godunov scheme, so all in all a fairly promising result. Despite this, more experiments are needed to support the hypothesis, and we will therefore proceed as in the one-dimensional experiments by adding more neurons to our DNN models, i.e. doubling the width of the network.

As before, we now perform the same batch of experiments, but with the total number of nodes being 48. This yields the training process given in Figure 3.15, which shows a more stable result than for the models with 24 nodes. We can also see that the loss has flattened a great deal – especially for the less complex models – and the changes of weights has shrunk to some degree. However, the change of weights has not yet flattened out in the experiment

Network	**Heaviside**	**Mirrored**	**Scaled**	**Sine**
24	1.35×10^{-1}	1.15×10^{-1}	1.28×10^{-1}	1.19×10^{0}
$12 \lessapprox 12$	7.30×10^{-2}	8.82×10^{-2}	6.89×10^{-2}	5.14×10^{-1}
$8 \lessapprox 8 \lessapprox 8$	6.14×10^{-2}	1.13×10^{-1}	1.08×10^{-1}	6.73×10^{-1}
$6 \lessapprox 6 \lessapprox 6 \lessapprox 6$	1.48×10^{-1}	2.22×10^{-1}	1.46×10^{-1}	8.51×10^{-1}

Table 3.14: Relative Euclidean error of 16 two-dimensional experiments with varying number of hidden layers and 24 nodes. Green cells show the best results for each initial function.



(a) 1 hidden. (b) 2 hidden. (c) 3 hidden.

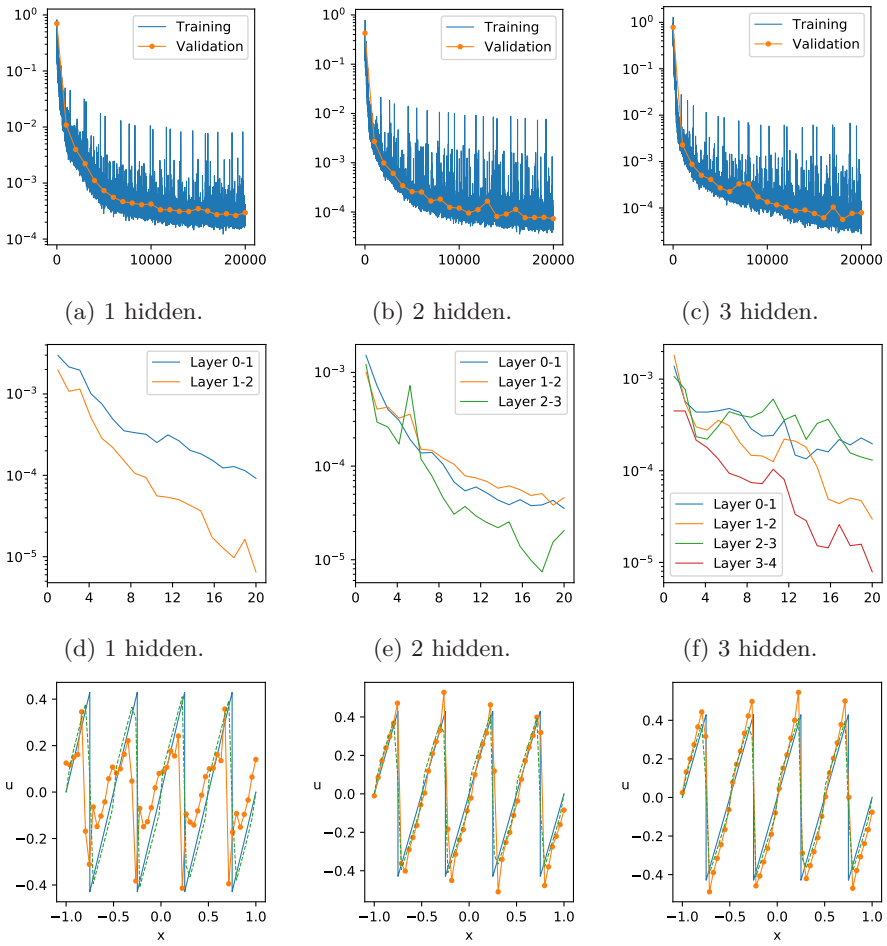
(d) 1 hidden. (e) 2 hidden. (f) 3 hidden.

(g) 1 hidden. (h) 2 hidden. (i) 3 hidden.

Figure 3.14: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initial (lower). Two-dimensional experiments with varying number of hidden layers and a total of 24 nodes.

Network	Heaviside	Mirrored	Scaled	Sine
48	5.39×10^{-2}	5.27×10^{-2}	8.72×10^{-2}	1.02×10^{0}
$24 \bowtie 24$	8.26×10^{-2}	8.06×10^{-2}	8.16×10^{-2}	4.69×10^{-1}
$16 \bowtie 16 \bowtie 16$	2.46×10^{-2}	1.77×10^{-2}	3.61×10^{-2}	2.51×10^{-1}
$12 \bowtie 12 \bowtie 12 \bowtie 12$	4.13×10^{-2}	5.55×10^{-2}	7.87×10^{-2}	6.55×10^{-1}

Table 3.15: Relative Euclidean error of 16 two-dimensional experiments with varying number of hidden layers and 48 nodes. Green cells show the best results for each initial function.



(a) 1 hidden. (b) 2 hidden. (c) 3 hidden.

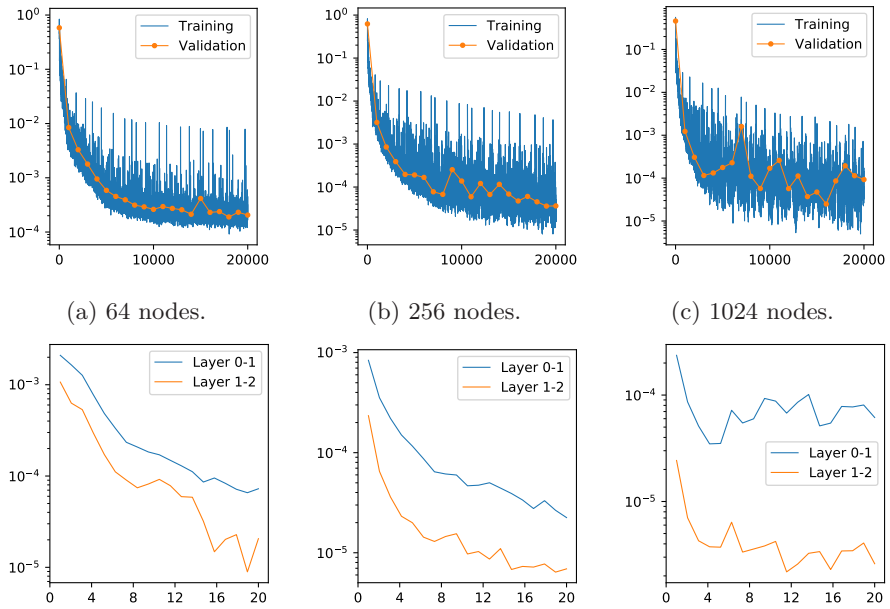
(d) 1 hidden. (e) 2 hidden. (f) 3 hidden.

(g) 1 hidden. (h) 2 hidden. (i) 3 hidden.

Figure 3.15: The training loss (upper blue), validation loss (upper yellow), change of weights (middle) and results with sine initial (lower). Two-dimensional experiments with varying number of hidden layers and a total of 48 nodes.

with 3 hidden layers, which indicates that more training is probably necessary. The corresponding relative errors are given in Table 3.15, which shows that the model with best performance is the one with 3 hidden layers, of which contains 16 nodes each. By comparing these values to Table 3.4, we see that our new models performs poorer when holding 4 hidden layers, which is expected as we have the exact same number of training data with a more complex input structure. Thus, as mentioned in the introduction to this section, we would probably need more complex models, to be able to reproduce results from Section 3.2.3 using the two-dimensional solver.

As a last confirmation of performance, we will experiment with different number of nodes in the DNNs, while keeping the number of layers constant. By setting the number of hidden layers to 1 and varying the number of nodes to



(a) 64 nodes. (b) 256 nodes. (c) 1024 nodes.

(d) 64 nodes. (e) 256 nodes. (f) 1024 nodes.

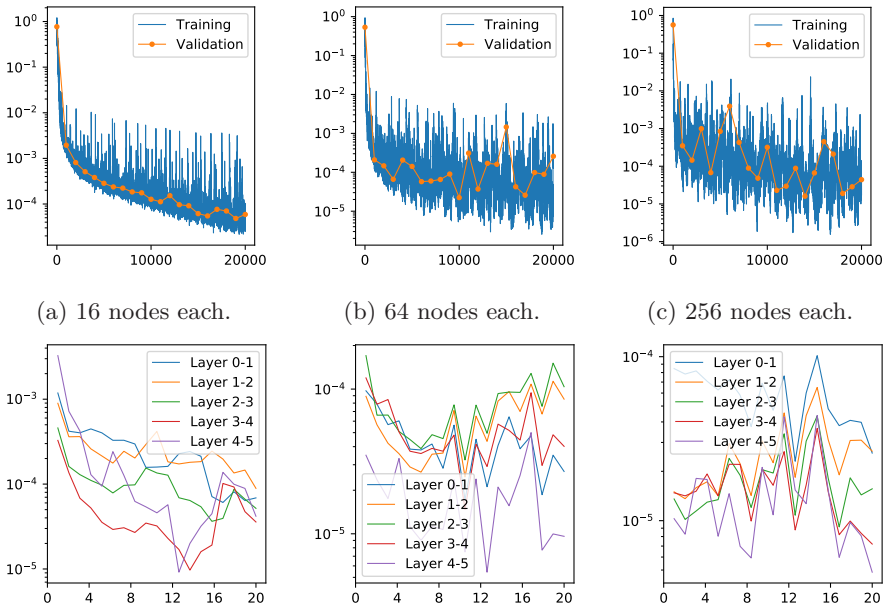
Figure 3.16: The training loss (upper blue), validation loss (upper yellow) and change of weights (lower). Two-dimensional experiments with varying number of nodes and 1 hidden layer.

Network	Heaviside	Mirrored	Scaled	Sine
4	4.17×10^{-1}	3.98×10^{-1}	3.63×10^{-1}	1.20×10^{0}
16	2.09×10^{-1}	1.31×10^{-1}	1.90×10^{-1}	1.89×10^{0}
64	1.11×10^{-2}	1.19×10^{-1}	8.45×10^{-2}	6.78×10^{-1}
256	4.26×10^{-2}	3.92×10^{-2}	4.57×10^{-2}	8.93×10^{-1}
1024	5.68×10^{-2}	8.91×10^{-2}	5.09×10^{-2}	1.58×10^{-1}

Table 3.16: Relative Euclidean error of 20 two-dimensional experiments with varying number of nodes and 1 hidden layer. Green cells show the best results for each initial function.

be 4^i for all $i \in [\![1, 5]\!]$, we obtain the relative errors listed in Table 3.16. The relative errors shows somewhat the same pattern as earlier, however, yet again with poorer results – due to lack of network complexity. The corresponding training loss and weight changes are shown in Figure 3.16, and by comparing these to Figure 3.5, we see similar tendencies in the movement and flatness of the curves.

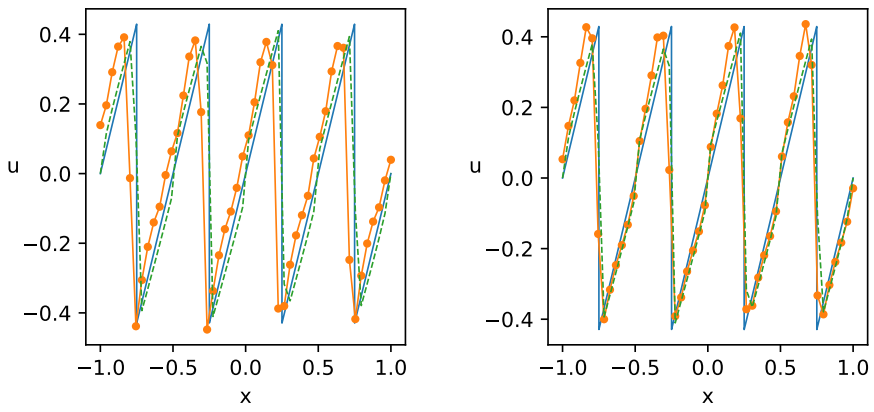
By increasing the number of layers to 4, while keeping the same number of nodes, we obtain results given in Table 3.17, which confirms poor performance compared to what we achieved in Sections 3.2.3 and 3.3.3. We see that 64 nodes in each of the four layers is the best practice in the latter table. The training progress of these experiments is given in Figure 3.17, showing a clear similarity



(a) 16 nodes each. (b) 64 nodes each. (c) 256 nodes each.

(d) 16 nodes each. (e) 64 nodes each. (f) 256 nodes each.

Figure 3.17: The training loss (upper blue), validation loss (upper yellow) and change of weights (lower). Two-dimensional experiments with varying number of nodes and 4 hidden layers.



(a) 1 hidden layer, 256 nodes. (b) 4 hidden layer, 64 nodes.

Figure 3.18: Results of two-dimensional experiments with sine initial condition for the best DNN models of 1 (left) and 4 (right) layers, namely 256 and 64 nodes, respectively.

Network	Heaviside	Mirrored	Scaled	Sine
$4 \bowtie \cdots \bowtie 4$	1.84×10^{-1}	2.52×10^{-1}	2.00×10^{-1}	1.00×10^{0}
$16 \bowtie \cdots \bowtie 16$	7.00×10^{-2}	4.64×10^{-2}	5.78×10^{-2}	5.75×10^{-1}
$64 \bowtie \cdots \bowtie 64$	5.24×10^{-2}	8.74×10^{-2}	3.14×10^{-2}	3.26×10^{-1}
$256 \bowtie \cdots \bowtie 256$	1.15×10^{-1}	1.75×10^{-1}	9.10×10^{-2}	3.27×10^{-1}

Table 3.17: Relative Euclidean error of 16 two-dimensional experiments with varying number of nodes and 4 hidden layers. Green cells show the best results for each initial function.

to the behavior in Figure 3.6. To make sure that the models actually does what we intended them to do we may inspect the behavior of the sine curves over time, illustrated in Figure 3.18. These figures shows that both models produce decent results, both similar to what the Godunov scheme produces.

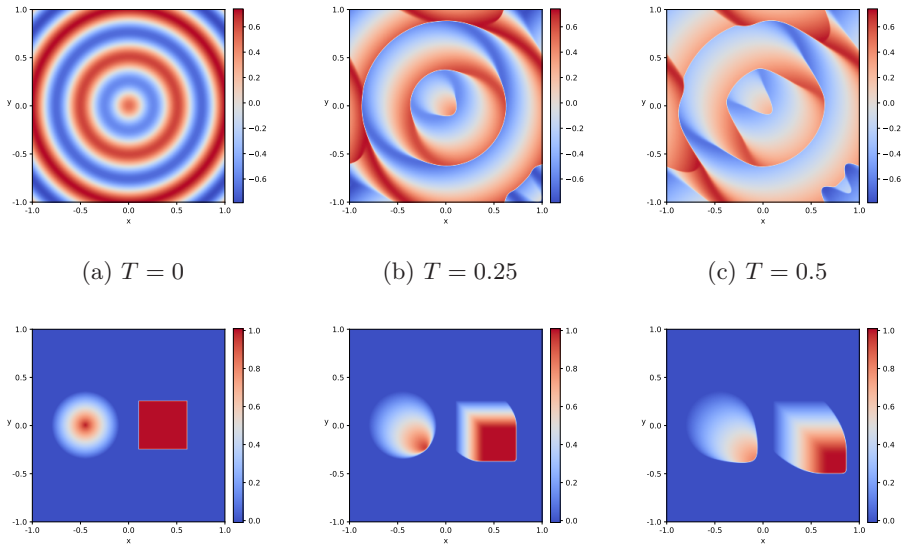
As predicted in the introduction to this section, the accuracy of the method is poorer when we use the same DNN structure as in Section 3.2.3. However, the results indicate that the numerical method functions as intended, with sufficient accuracy. Thus, we move on to studying initial-value problems where the vertical axis is non-constant.

3.4.5 Genuinely 2D experiments

We have now executed all comparison experiments, and they all make up a strong indication that the implementation works as intended. As illustrated in Section 3.4.3, we will now move on to initial conditions yielding authentic two-dimensional initial-value problems, see Figure 3.13. This will indicate how good the models approximate solutions of actual two-dimensional problems, as we have only looked at change in one spatial dimension so far. We will conduct experiments using the exact same DNN models as in Section 3.4.4, however, now we pick the four best models, one from each of the four batches of experiments. By looking back at the relative errors from each of the tables, this yields hidden layer structures $12 \bowtie 12$, $16 \bowtie 16 \bowtie 16$, 256 and $64 \bowtie 64 \bowtie 64 \bowtie 64$. We will also include two different temporal steps, namely $T = 0.25$ and $T = 0.5$, to be able to observe the propagation through time to some extent. Lastly, we will also perform tests with mesh sizes $N_x = 100$ and $N_y = 100$ to see whether or not the method seems to converge toward the reference solutions. All illustrated results will be compared to the reference solutions given in Figure 3.19, which are approximate solutions calculated on spatial mesh with size 1000×1000.

We start by running experiments with the Sigmoid–cosine function as our initial data, see Figure 3.19a. This yields the relative errors shown in the two leftmost columns of Table 3.18. The relative errors are in these cases calculated with respect to the reference solutions, by up-scaling the mesh of the approximate solutions from 50×50 to 1000×1000. The respective results are given in Figure 3.20, showing the temporal state $T = 0.25$ in the leftmost column, whereas the rightmost column shows the state $T = 0.5$. As reflected in the relative errors, we see that the two most precise DNN solvers are the ones with network structures $16 \bowtie 16 \bowtie 16$ and $64 \bowtie 64 \bowtie 64 \bowtie 64$. This makes sense, since earlier experiments have showed that the most complex models are

the best performed ones, given that they are trained properly. The last row of Table 3.18 contains the relative error of the Godunov scheme with respect to the reference solution. As we can see, the Godunov scheme yields more accurate approximation compared to our DNN models when considering the Sigmoid–cosine initial function. The reason for this may be a poorly trained DNN model or poor precision in the training data. It is most likely a combination of



(a) $T = 0$ (b) $T = 0.25$ (c) $T = 0.5$

(d) $T = 0$ (e) $T = 0.25$ (f) $T = 0.5$

Figure 3.19: Reference solutions with mesh size 1000×1000, for two-dimensional initial-value problems with Sigmoid–cosine (upper) and LeVeque (lower) initials.

Network structure	**Sigmoid–cosine** $T = 0.25$	$T = 0.5$	**LeVeque** $T = 0.25$	$T = 0.5$
$12 \bowtie 12$	4.41×10^{-1}	6.32×10^{-1}	2.96×10^{-1}	4.02×10^{-1}
$16 \bowtie 16 \bowtie 16$	4.32×10^{-1}	5.07×10^{-1}	2.32×10^{-1}	3.04×10^{-1}
256	4.92×10^{-1}	6.32×10^{-1}	2.54×10^{-1}	3.70×10^{-1}
$64 \bowtie 64 \bowtie 64 \bowtie 64$	4.27×10^{-1}	4.90×10^{-1}	3.02×10^{-1}	4.40×10^{-1}
Godunov	4.20×10^{-1}	4.76×10^{-1}	2.41×10^{-1}	3.12×10^{-1}

Table 3.18: Relative Euclidean error of two-dimensional experiments with genuine two-dimensional initial conditions. Hidden layers vary from 1 to 4, with varying number of nodes. The DNN models used are the best performed ones from their respective experiments of two-dimensional problems (with 1D initial conditions) above. The errors are calculated by up-scaling the DNN approximations from 50×50 to 1000×1000 mesh, to match their respective reference solutions. The last row shows the performance of Godunov's two-dimensional scheme, for comparison. Green cells show the best results for each temporal maximum and initial function.

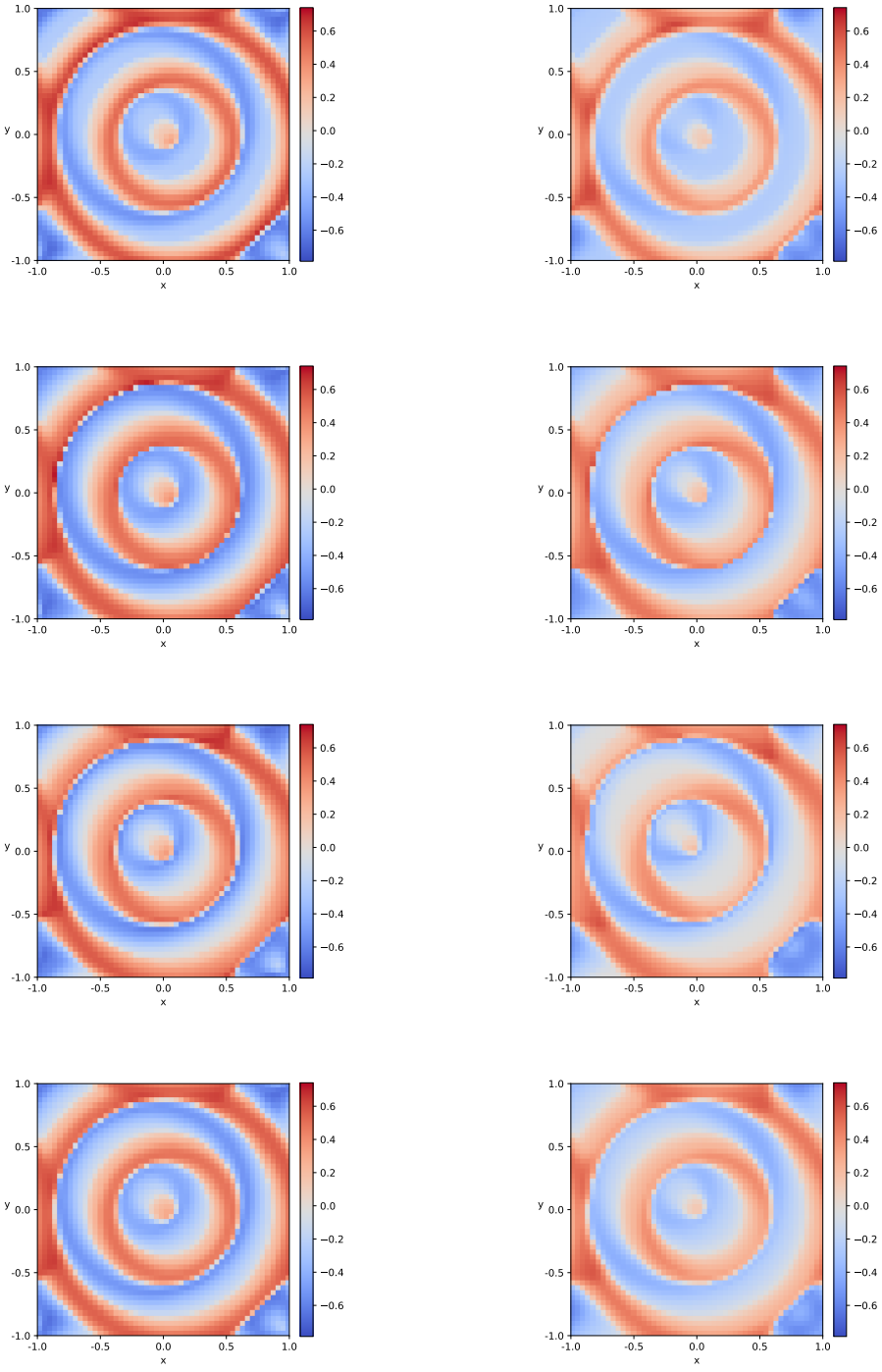


Figure 3.20: Approximate solutions of Burgers' equation with Sigmoid–cosine function. Columns: $T = 0.25, 0.5$. Rows: DNN structures $12 \rtimes 12$, $16 \rtimes 16 \rtimes 16$, 256 and $64 \rtimes 64 \rtimes 64 \rtimes 64$, respectively. Spatial mesh size 50×50.

these, together with the lack of optimization with respect to other parameters, e.g. batch size, number of epochs etc. The run-time of these experiments are listed in the upper leftmost column of Table 3.19, showing that the usage of such a DNN based method is much less time-consuming than performing the fine-mesh algorithm.

The next batch of results are the ones using the LeVeque initial function, see Figure 3.19d. By looking at the two rightmost columns of Table 3.18, it seems that the best performing DNN model is the one with network structure $16 \times 16 \times 16$. By comparing the relative errors of this model to the ones of Godunov's method, we see that our model outperforms the accuracy of Godunov's method. Figure 3.21 illustrates the wave propagations of these experiments, and by comparing these figures with the respective reference solution we see that second row seems to fit the most – this row is the results corresponding to the 3-layered network with 16 nodes each. The run-time of these experiments are given in the upper rightmost column of Table 3.19, emphasizing once more that the temporal aspect of the fine-mesh solver is by far outperformed by such DNN based methods.

Lastly, we perform experiments using the same DNN models, but with increased mesh resolution. By setting $N_x = 100$ and $N_y = 100$, and running the exact same experiments for Sigmoid–cosine function and LeVeque function, we obtain the relative errors listed in Table 3.20. We see that the relative errors of each and every experiment has shrunk, compared to Table 3.18. Further, we will show results for the best performed models, which for both initial conditions are the DNN with structure $16 \times 16 \times 16$, illustrated in Figure 3.22. Since we now use a finer mesh, it is easier to observe the structures of the approximate solutions, when comparing these figures to the reference solutions in Figure 3.19. We see obvious similarities between these approximations and the reference solutions, which indicates that the numerical method performs well. The best

Network structure	**Sigmoid–cosine**	**LeVeque**
12×12	0.65	0.93
$16 \times 16 \times 16$	0.80	1.27
256	0.55	0.80
$64 \times 64 \times 64 \times 64$	0.92	1.35
Fine-mesh solver	30.15	28.13
12×12	6.77	7.82
$16 \times 16 \times 16$	7.50	10.85
256	5.48	6.67
$64 \times 64 \times 64 \times 64$	8.27	11.80
Fine-mesh solver	> 30.15	> 28.13

Table 3.19: Run-time of DNN solver and fine-mesh solver, measured in minutes. Time for DNN solver is based on a pre-trained network. Time for fine-mesh solver is based on implementation used to train network. All methods in upper half have been run using mesh size 50×50, whereas the lower half are methods with mesh size 100×100. The fine-mesh solver is not ran with the latter mesh size as this is extremely time consuming.

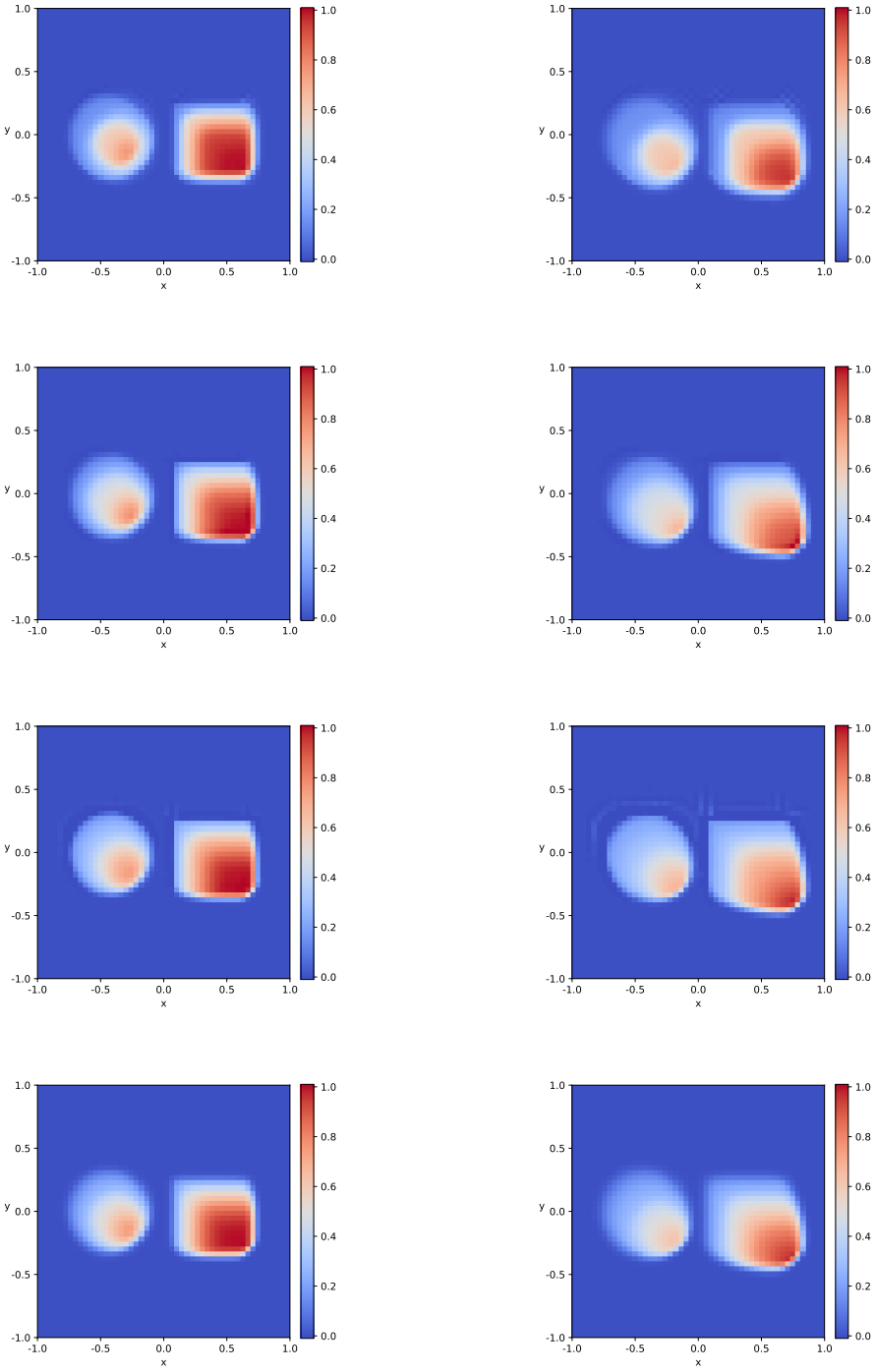


Figure 3.21: Approximate solutions of Burgers' equation with LeVeque function. Columns: $T = 0.25, 0.5$. Rows: DNN structures $12 \rtimes 12$, $16 \rtimes 16 \rtimes 16$, 256 and $64 \rtimes 64 \rtimes 64 \rtimes 64$, respectively. Spatial mesh size 50×50.

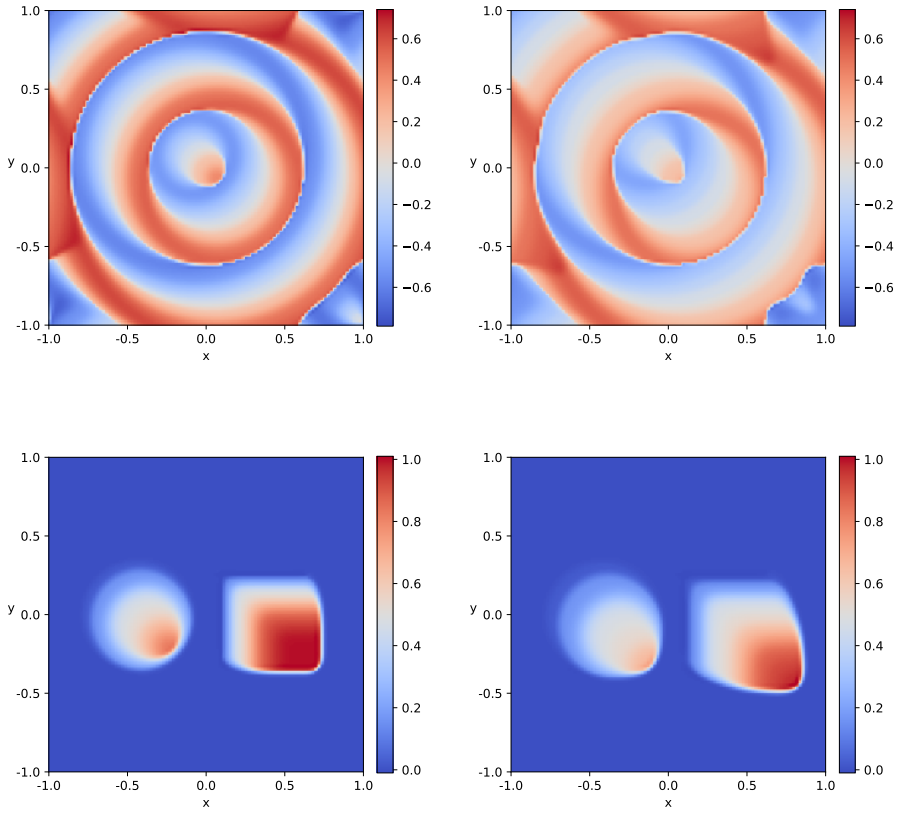


Figure 3.22: DNN based approximate solution of initial-value problems with Sigmoid-cosine (upper) and LeVeque (lower) initial functions, with temporal maximum $T = 0.25$ (left) and $T = 0.5$ (right). The mesh size is 100×100.

Network structure	**Sigmoid–cosine** $T = 0.25$	$T = 0.5$	**LeVeque** $T = 0.25$	$T = 0.5$
$12 \lessapprox 12$	3.39×10^{-1}	5.95×10^{-1}	2.60×10^{-1}	3.70×10^{-1}
$16 \lessapprox 16 \lessapprox 16$	2.98×10^{-1}	3.93×10^{-1}	1.68×10^{-1}	2.18×10^{-1}
256	4.46×10^{-1}	6.11×10^{-1}	2.06×10^{-1}	3.21×10^{-1}
$64 \lessapprox 64 \lessapprox 64 \lessapprox 64$	3.25×10^{-1}	4.04×10^{-1}	2.47×10^{-1}	3.96×10^{-1}
Godunov	2.75×10^{-1}	3.38×10^{-1}	1.72×10^{-1}	2.23×10^{-1}

Table 3.20: Relative Euclidean error of two-dimensional experiments with genuine two-dimensional initial conditions, using mesh size 100×100. Hidden layers vary from 1 to 4, with varying number of nodes. The DNN models used are the best performed ones from their respective experiments of two-dimensional problems (with 1D initial condition) above. The errors are calculated by up-scaling the DNN approximations from 100×100 to 1000×1000 mesh, to match their respective reference solution. The green cells show the best results for each temporal maximum and initial function.

performing methods are illustrated in green in Table 3.20, showing the same pattern as previous experiments. This is as expected since we are using the exact same pre-trained DNN models. The lower part of Table 3.19 shows the run-time of these experiments. Since the fine-mesh solver is extremely time-consuming, these experiments are not ran with this method. However, we know that the time would be sufficiently greater than when we ran the method using lower mesh resolution, which is a degree of efficiency outperformed by the DNN based methods.

Summarized, we have looked at two-dimensional conservative initial-value problems, using Burgers' flux in both spatial directions, with Sigmoid–cosine and LeVeque initial conditions. The DNN based method has been compared to Godunov's method for two different mesh sizes. With the LeVeque initials we have seen that the DNN based method have the potential to outperform Godunov's method with respect to accuracy. Further, by comparing all error-estimates of Godunov's two-dimensional method in this thesis with the ones given in an article by L. Gosse [Gos14], we see similar tendencies, indicating correct implementations of the comparison scheme. Moreover, the run-time of the method has been compared to an accurate fine-mesh algorithm, which has shown that a DNN based method are way more efficient. Thus, we may conclude that the proposed two-dimensional method has considerable potential for further studies, given that we pre-train a DNN model sufficiently, using sensible parameters.

3.5 Sources of error

All the methods we have implemented and tested are dependent on a high number of parameters. The number of distinct experiments we are able to do increases exponentially with respect to the parameters, and it is therefore a problematic number of tests needed to be able to conclude precisely. Thus, the performed experiments are a selection chosen to get an indication whether or not the DNN based numerical methods work reasonably well. A bad choice of this selection can effect the results, and consequently the conclusions.

A critical source of error in the results above is that it has been chosen one specific seed during the experiments. If we are unfortunate, this has resulted in a bad sample of data, or unstable training. The fix for this problem is to run the experiments several times with different seeds, i.e. performing a bootstrap method. The reason why this has not been done is that the run time is proportional to the number of seeds chosen, and it would take an awful lot of time to perform all the experiments with the desired precision.

A typical source of error is rounding errors, which is a consequence of the finite precision in the computer. All implementations are written carefully, with this in mind. However, it may be hard to detect such errors, so this may still have impacted the results.

CHAPTER 4

Summary and further work

In this thesis we have explored the possibility of using DNN models as a tool for obtaining approximate solutions of conservative initial-value problems. We introduced a rigorous foundation on the theoretical aspects, and then moved on to proposing three numerical schemes, of which the two-dimensional method is considered the main result. All methods have been implemented and tested to a degree where conclusions may be stated.

4.1 Discussion and conclusion

The first method, introduced in Section 3.2, was the DNN based numerical method for solving one-dimensional conservative initial-value problems, using DNNs with MSE-loss. The accuracy of the performed experiments was acceptable. However, the method yielded poor efficiency, which was expected and hypothesized on beforehand.

Secondly, we developed a numerical method based on a physics-informed neural network, see Section 3.3. The only difference between this method and the previous was the choice of loss function, which in this case was a physics-informed L^1-loss dependent on the structure of the finite-volume scheme. This method resulted in an obvious improvement of stability, compared to when we tested with regular MSE-loss. An additional benefit of this method is that the time-complexity of the method is equal to the previous one, since the complexity is only added during the training process of the DNN. Thus, this method is preferable to the above, given pre-trained DNN models.

Lastly, we proposed a new finite-volume method for approximating solutions to two-dimensional conservative initial-value problems, see Section 3.4. The DNN models used was in this case the standard MSE-loss models, similar to the ones used for the first method. These results also showed themselves somewhat satisfactory, but did not yield convincingly good accuracy compared to a two-dimensional Godunov solver. However, as we did not focus on optimizing the parameters of the method, the accuracy ended satisfactory close to the comparison scheme. When performing tests on the LeVeque initial-value problem we actually obtained better results, which was a relief since the Godunov scheme performed on a coarse mesh in two spatial dimensions is considered a bad-performed two-dimensional scheme. Further, the run-time table showed that the DNN based method was by far more efficient than the fine-mesh algorithm itself. Thus, by training a DNN model to a sufficient degree, with

optimized choice of parameters, we can expect the method to be both efficient and accurate.

Summarized, it might be advantageous to consider DNN based numerical schemes to approximate solutions of two-dimensional conservative initial-value problems. However, it should be executed a parameter grid-search in order to pre-train such a model to a satisfactory degree. Moreover, as strongly indicated by the experiments in Section 3.3.3, the second and third method should be considered together, creating a physics-informed neural network for approximating solutions of two-dimensional conservation laws. The biggest problem with such a method is the time complexity of the implementation of a training data generator, as well as the training process itself. On the other hand, we mentioned the idea of convolutional neural networks in Section 3.3.4, which could ease the computational complexity to a significant degree. Another area of machine learning which could help increase the efficiency of a numerical scheme is *unsupervised learning*. However, by using such models we would most likely loose accuracy of the results.

4.2 Improvements

Since the physics-informed L^1-loss outperforms the regular MSE-loss, a natural improvement of this study would be to include tests of the two-dimensional solvers, using models with this loss function. However, this is excluded as this would demand a great deal of resources, both temporal and with respect to computational power.

Another strong improvement of this study would be to apply the tested models on actual physical problems, such as the shallow water equations. This is not considered simply due to time limitations, as this would require a whole new setup of implementation since this includes a system of conservation laws. This raises criticism to me as developer, as I could have facilitated this in advance, had I ever thought so far ahead.

To increase efficiency of all methods, there are some possibilities which are not yet mentioned. One option is to consider a lower level programming language. Python is actually written in C, and all Python code must therefore be compiled down to C before the logic is further translated to machine code. Consequently, all Python implementations demands sufficiently more time than C implementations. Another possibility is to consider gradient boosting libraries. An example of this is *XGBoost*, which is an optimized distributed gradient boosting library designed to be highly efficient. Moreover, we could also consider using variable spatial mesh sizes $\Delta\hat{x}$ and $\Delta\hat{y}$. As observed in the motivation to Section 3.4.2, we have non-constant flux nearby a four cell intersection. Thus, if we chose the fine-mesh cell sizes to be e.g. normally distributed – i.e. close to a four cell intersection $\Delta\hat{x}$ and $\Delta\hat{y}$ decreases – we could obtain similar accuracy with increased efficiency.

4.3 Further work

The above mentioned improvements contains a natural continuation for studies on these problems. The first thing that should be considered is probably tests with the extended L^1-loss function on two-dimensional problems. In

Section 3.3.4 we mentioned how such problems may be considered in the eyes of an image analyst as a convolutional problem, so this other way to look at it could be a strong foundation for improvement of the time complexity, and therefore also make it possible to ease computations if this loss is tested on two-dimensional problems.

When experiments with L^1-loss has been conducted with satisfactory results, it is essential to properly implement and test the method for a system of conservation laws, e.g. by testing with the shallow water equations. We do not have explicit formulas for Godunov's flux when dealing with such systems, thus, the DNN based method has a better chance at outperforming alternative methods and will therefore be potentially more valuable.

After all necessary tests are adequately executed, it should be completed a full stability and convergence analysis of the DNN based method. In 1980 M. G. Crandall and A. Majda published an article proving convergence and stability of monotone difference approximations for scalar conservation laws [CM80], e.g. Godunov's scheme. To prove similar results for the DNN based methods, we are dependent on restricting the DNN with respect to both the network structure and training process. That is, we could begin such an analysis by assuming that the DNNs have one specific activation and loss function, and moreover, restrict the depth and width of the network to be bounded by some reasonably small integer.

APPENDIX A

Python implementations

In this appendix we will include short versions of the implemented code. The fully completed code may be found on GitHub in repository `riemannDNNsolver` under username `aasmunkv`.

GitHub link: https://github.com/aasmunkv/riemannDNNsolver

This appendix contains logic for Python implementation of DNN based methods, Godunov's method and data generators. In cases where methods of classes are similar to already mentioned code, we only include the methods which are fairly distinct. The code given here explains the logic thoroughly but will not be functioning since logic with respect to type-conversions is removed, e.g. conversion between NumPy-arrays and PyTorch-tensors. This is done for the sake of readability.

A.1 DNN based one-dimensional scheme

The package contains a sub-package called `dnn1d`, which contains two code files, namely the network and the DNN based finite-volume scheme for one-dimensional problems. In this section we include code snippets from both of these files.

The backpropagation for physics-informed L^1-loss are similar to the backpropagation with MSE-loss, see Listing A.3. However, with `nn.L1Loss` on lines 12, 26 and 33, as well as with an additional loss calculations given by the function `loss_var`, see Listing A.4.

```
1  def __init__(self, dimensions, activation=F.relu, final_activation=False):
2      self.dimensions = dimensions
3      self.size = len(dimensions)
4      self.activation = activation
5      self.final_activation = final_activation
6
7      self.layer_inp = nn.Linear(dimensions[0], dimensions[1])
8      self.layer_hid = nn.ModuleList([
9          nn.Linear(dimensions[i], dimensions[i+1])
10             for i in range(1, self.size - 2)
11     ])
12     self.layer_out = nn.Linear(dimensions[-2], dimensions[-1])
```

Listing A.1: Initialization of the network.

```
def feedForward(self, inp):
    inp = inp
    out = self.layer_inp(inp)
    out = self.activation(out)
    for layer in self.layer_hid:
        out = self.activation(layer(out))
    out = self.layer_out(out)
    return out if not self.final_activation else self.activation(out)
```

Listing A.2: Feed forward in network.

```
def backward(self, data_train, data_val, epochs, batchsize, destination,
     name):
    # Initialize inp_train,out_train,inp_val,out_val from data_train,
     data_val

    # Make a 'data-feeder'
    sampler = torch.utils.data.DataLoader(
        range(self.N),
        batch_size=self.batchsize,
        shuffle=True
    )

    # Initialize lists: loss_train, loss_val, weights
    val_loss = nn.MSELoss()(
        self.network.forward(inp_val), out_val
    )
    self.loss_val.append(val_loss)
    best_loss, cur_loss = np.inf, np.inf

    for epoch in range(self.epochs):
        loss_tmp = []
        for i, batch in enumerate(sampler):
            self.opt.zero_grad()
            loss_out = self.network.forward(
                inp_train[batch]
            )
            loss_tar = out_train[batch]
            cur_loss = nn.MSELoss()(loss_out, loss_tar)
            cur_loss.backward(retain_graph=True)
            self.opt.step()
            self.loss_train.append(cur_loss.data)
            if cur_loss < best_loss:
                best_loss = cur_loss
                torch.save(self.network, '[destination]/[filename]')
        val_loss = nn.MSELoss()(
            self.network.forward(inp_val), out_val
        )
        self.loss_val.append(val_loss)
        self.weights.append(
            [self.network.layer_inp.weight]
            + [l.weight for l in self.network.layer_hid]
            + [self.network.layer_out.weight]
        )
```

Listing A.3: Backpropagation with MSE-loss.

```
def loss_var(inp, out):
    inp = inp
    out = out
    out = out.squeeze(-1)
    loss = torch.zeros_like(inp, requires_grad=False)
```

```
6
7       dx = 1/self.N
8       dt = dx/(torch.max(torch.abs(self.dfdu(inp))))
9       C = dt/dx
10
11      loss[:,:-1] = inp[:,:-1] - C*(out[:,1:] - out[:,:-1])
12      loss[:,-1] = inp[:,-1] - C*(out[:,0] - out[:,-1])
13      return loss
```

Listing A.4: Backpropagation with physics-informed L^1-loss.

```
1  class DNNscheme:
2      def __init__(self, f, dfdu, u0, bnd_cond, xmin, xmax, Nx, network, T
       =1.0, C=0.5):
3          self.f = lambda U: f(U)
4          self.dfdU = lambda U: dfdu(U)
5          self.u0 = lambda x: u0(x)
6          self.bnd_cond = bnd_cond
7          self.xmin, self.xmax, self.Nx = xmin, xmax, Nx
8          self.x = torch.linspace(xmin, xmax, Nx)
9          self.dx = (xmax - xmin)/(Nx-1)
10         self.T = T
11         self.C = C
12         self.dt, self.Nt = None, None
13         self.net = network
14         self.u = [self.u0(self.x)]
15
16     def set_dt(self):
17         a = torch.max(torch.abs(self.dfdU(self.u[-1])))
18         dt = self.C*self.dx/a
19         self.Nt = np.ceil(self.T/dt)
20         _, self.dt = np.linspace(0, self.T, self.Nt, retstep=True)
21
22     def dnn_flux(self, u_l, u_r):
23         U = torch.stack((u_l, u_r), dim=1)
24         pred = []
25         for u in U:
26             pred.append(self.net.forward(u))
27         return torch.tensor(pred)
28
29     def scheme(self):
30         C = self.dt/self.dx
31         u = self.u[-1]
32         u_next = torch.zeros(u.shape)
33         u_next[1:-1] = u[1:-1] - C*(self.dnn_flux(u[1:-1], u[2:]) - self.
       dnn_flux(u[:-2], u[1:-1]))
34         if (self.bnd_cond=='periodic'):
35             u_next[0]  = u[0]  - C*(self.dnn_flux(u[0] , u[1]) - self.
       dnn_flux(u[-2] , u[0]))
36             u_next[-1] = u[-1] - C*(self.dnn_flux(u[-1], u[1]) - self.
       dnn_flux(u[-2] , u[-1]))
37         elif (self.bnd_cond=='dirichlet'):
38             u_next[0]  = u[0]
39             u_next[-1] = u[-1]
40         elif (self.bnd_cond=='neumann'):
41             u_next[0]  = u_next[1]  + (u[0]  - u[1])
42             u_next[-1] = u_next[-2] + (u[-1] - u[-2])
43         elif (self.bnd_cond=='robin'):
44             a, b = 1, 1
45             c = (a*u[1])/self.dx + (b - a/self.dx)*u[0]
46             u_next[0] = (c - (a*u_next[1])/self.dx)/(b - a/self.dx)
47             d, e = 1, 1
48             f = (d*u[-1])/self.dx + (e - d/self.dx)*u[-2]
```

```
            g = 1 - (e*self.dx)/2
            u_next[-1] = u[-1] - g*u[-2] + g*u_next[-2]
        else:
            u_next[0]  = u[0]
            u_next[-1] = u[-1]
        self.u.append(u_next)

    @property
    def solve(self):
        t = 0
        while t<self.T:
            self.set_dt()
            t += self.dt
            if t>=self.T:
                self.dt -= (t-self.T)
                t = self.T
            self.scheme()
```

Listing A.5: DNN based finite-volume scheme.

A.2 DNN based two-dimensional scheme

The package contains a sub-package called `dnn2d`, which contains two code files, namely the network and the DNN based finite-volume scheme for two-dimensional problems. In this section we include code snippets from both of these files.

The implementations for solving two-dimensional problems are fairly similar to the implementations of the one-dimensional scheme. However, there are some crucial differences, and we will list the methods varying the most relative to above implementations. This includes how to calculate the flux, Listing A.6, and the scheme logic itself, Listing A.7.

```
def get_flux(u):
    F = torch.zeros((u.size(0)-2, u.size(1)-1))
    for i in range(F.size(0)):
        for j in range(F.size(1)):
            F_inp = torch.zeros(7)
            F_inp[:-1] = u[i:i+3,j:j+2].reshape(6)
            F_inp[-1] = self.dt/np.min((self.dx,self.dy))
            F[i,j] = self.net.forward(F_inp)
    return F

def dnn_flux(self):
    u_pad = torch.cat(  (
        self.u[-1][:,-2].reshape(self.u[-1].size(0),1),
        self.u[-1],
        self.u[-1][:,1].reshape(self.u[-1].size(0),1)
    ), dim=1    )
    u_F = torch.cat(    (
        u_pad[-2,:].reshape(1, u_pad.size(1)),
        u_pad,
        u_pad[1,:].reshape(1, u_pad.size(1))
    ), dim=0    )
    u_G = u_F.rot90()
    F, G = get_flux(u_F), get_flux(u_G).rot90(3)
    return (F,G)
```

Listing A.6: DNN based flux algorithm.

```
def scheme(self):
    u = self.u[-1]
    C_x, C_y = self.dt/self.dx, self.dt/self.dy
    god_flux_f, god_flux_g = self.dnn_flux()

    u_next = torch.zeros(u.shape)
    u_next[1:-1,1:-1] = u[1:-1,1:-1] \
        - C_x*( god_flux_f[1:-1,2:-1] - god_flux_f[1:-1,1:-2]
        ).reshape([self.u[-1].size(0)-2,self.u[-1].size(1)-2]) \
        - C_y*( god_flux_g[2:-1,1:-1] - god_flux_g[1:-2,1:-1]
        ).reshape([self.u[-1].size(0)-2,self.u[-1].size(1)-2])

    if self.bnd_cond=='periodic':
        u_next[1:-1, 0] = u[1:-1,0] \
            - C_x*( god_flux_f[1:-1,1] - god_flux_f[1:-1,0] ) \
            - C_y*( god_flux_g[2:-1,0] - god_flux_g[1:-2,0] )
        u_next[1:-1,-1] = u[1:-1,-1] \
            - C_x*( god_flux_f[1:-1,-1] - god_flux_f[1:-1,-2] ) \
            - C_y*( god_flux_g[2:-1,-1] - god_flux_g[1:-2,-1] )
        u_next[0, 1:-1] = u[0,1:-1] \
            - C_x*( god_flux_f[0,2:-1] - god_flux_f[0,1:-2] ) \
            - C_y*( god_flux_g[1,1:-1] - god_flux_g[0,1:-1] )
        u_next[-1,1:-1] = u[-1,1:-1] \
            - C_x*( god_flux_f[-1,2:-1] - god_flux_f[-1,1:-2] ) \
            - C_y*( god_flux_g[-1,1:-1] - god_flux_g[-2,1:-1] )
        u_next[0,0] = (u_next[0,1] + u_next[1,0])/2
        u_next[0,-1] = (u_next[0,-2] + u_next[1,-1])/2
        u_next[-1,0] = (u_next[-1,1] + u_next[-2,0])/2
        u_next[-1,-1] = (u_next[-1,-2] + u_next[-2,-1])/2

    elif self.bnd_cond=='neumann':
        u_next[:, 0] = u_next[:, 1] + C_x*(u[:, 0] - u[:, 1])
        u_next[:,-1] = u_next[:,-2] + C_x*(u[:,-1] - u[:,-2])
        u_next[0, :] = u_next[1, :] + C_y*(u[0, :] - u[1, :])
        u_next[-1,:] = u_next[-2,:] + C_y*(u[-1,:] - u[-2,:])

    self.u.append(u_next)
```

Listing A.7: DNN based finite-volume scheme.

A.3 Godunov's one-dimensional scheme

The package contains a sub-package called `godunov`, which contains two code files, namely Godunov's scheme for both one- and two-dimensional problems. In this section we include code snippets from the one-dimensional file.

The Godunov scheme is implemented in a similar matter as to what is done for the DNN based one-dimensional scheme above. However, the flux calculations are done by logic in Listing A.8, and not by using a DNN.

```
def godunov_flux(self, U_L, U_R):
    U_L, U_R = torch.flatten(U_L), torch.flatten(U_R)
    arr = torch.linspace(0,1,resolution)
    arr_mesh, U_L_mesh = torch.meshgrid(arr, U_L)
    _, U_R_mesh = torch.meshgrid(arr, U_R)
    diff = (U_R_mesh-U_L_mesh)
    flip = (diff < 0)*(-1)
    arr_mesh_flip = torch.abs(arr_mesh + flip)
    U_min = torch.min(U_L, U_R).reshape(U_L.size(),1)
    arr_scaled = torch.add(arr_mesh_flip*torch.abs(diff), U_min)
    arr_f = self.f(arr_scaled)
```

```
    flux_min = torch.min(arr_f, axis=0)[0]
    flux_max = torch.max(arr_f, axis=0)[0]
    cond = torch.stack(( (U_L < U_R), (U_R <= U_L) ))
    cond_shape = cond.shape[1:]
    cond = cond.reshape(cond.size()[0], flux_min.size()[-1])
    flux = (flux_min*cond[0] + flux_max*cond[1])
    return flux.reshape(cond_shape)
```

Listing A.8: Godunov's scheme.

A.4 Godunov's two-dimensional scheme

In this section we include code snippets from the two-dimensional file from sub-package called `godunov`. The two-dimensional Godunov scheme is similar to the regular, but with flux calculation in both spatial dimensions, separately. The implementation of this is located in Listing A.9.

```
def get_flux(u):
    F = torch.zeros((u.size(0), u.size(1)-1))
    for i in range(0,F.size(0),10):
        F[i:i+10,:] = self.godunov_flux(
            u[i:i+10,:-1],u[i:i+10,1:]
        ).reshape(F[i:i+10,:].size())
    return F

def god_flux(self):
    u_pad = torch.cat(  (
        self.u[-1][:,-2].reshape(self.u[-1].size(0),1),
        self.u[-1],
        self.u[-1][:,1].reshape(self.u[-1].size(0),1)
    ), dim=1    )
    u_F = torch.cat(    (
        u_pad[-2,:].reshape(1, u_pad.size(1)),
        u_pad,
        u_pad[1,:].reshape(1, u_pad.size(1))
    ), dim=0    )
    u_G = u_F.rot90()
    F, G = get_flux(u_F[1:-1]), get_flux(u_G[1:-1]).rot90(3)
    return (F,G)
```

Listing A.9: Godunov based flux algorithm.

A.5 One-dimensional data generator

This section contains the implementations for both one-dimensional data generators. Listing A.10 shows the code used to generate data for one-dimensional experiments with MSE-loss, whereas Listing A.11 contains data generator for the experiments with physics-informed L^1-loss.

```
class Dataset:
    def __init__(self, N, f, loc=0.0, scale=1.0):
        self.god_flux_mesh_size = 10000
        self.N = N
        self.f = f
        self.l = loc
        self.s = scale
        self.data = torch.zeros((self.N, 3))

```

```
    @property
    def create(self):
        self.data[:,0] = torch.randn((self.N)) * self.s + self.l
        self.data[:,1] = torch.randn((self.N)) * self.s + self.l

        for i in range(0,self.N, 1000):
            U_L = self.data[i:i+1000, 0]
            U_R = self.data[i:i+1000, 1]
            arr = torch.linspace(0, 1, resolution)
            arr_mesh, U_L_mesh = torch.meshgrid(arr, U_L)
            _, U_R_mesh = torch.meshgrid(arr, U_R)
            diff = (U_R_mesh-U_L_mesh)
            flip = (diff < 0)*(-1)
            arr_mesh_flip = torch.abs(arr_mesh + flip)
            U_min = torch.min(U_L, U_R).reshape(U_L.size(),1)
            arr_scaled = torch.add(
                arr_mesh_flip*torch.abs(diff), U_min
            )
            arr_f = self.f(arr_scaled)
            flux_min = torch.min(arr_f, axis=0)[0]
            flux_max = torch.max(arr_f, axis=0)[0]
            cond = torch.stack(( (U_L < U_R), (U_R <= U_L) ))
            cond_shape = cond.shape[1:]
            cond = cond.reshape(
                cond.size()[0], flux_min.size()[-1]
            )
            self.data[i:i+1000,2] = (
                flux_min*cond[0] + flux_max*cond[1]
            ).reshape(cond_shape)

    @property
    def get_data(self):
        return self.data

    def save(self, destination, filename):
        torch.save(self.data, '[destination]/[filename]')

    def load(self, destination, filename):
        self.data = torch.load('[destination]/[filename]')
```

Listing A.10: Data generator for one-dimensional experiments.

```
class Dataset_L1Loss:
    def __init__(self, M, N, K, f, dfdu):
        self.M = M
        self.N = N
        self.K = K
        self.f = f
        self.dfdu = dfdu
        self.data = torch.zeros((self.M, 2, self.N))

    @property
    def create(self):
        dx = 1/self.N
        x = torch.transpose(
            torch.linspace(dx, 1, self.N).expand(1, self.N), 0, 1
        )
        for i in range(self.M):
            coeffs = torch.tensor([
                np.random.normal(0,1/k) for k in range(1,self.K+1)
            ]).reshape(self.K,1)
            k_inds = torch.linspace(1, self.K, self.K)
            v_inp = torch.sin(x*k_inds*np.pi) @ coeffs
```

```
            dt = dx/(2*torch.max(torch.abs(self.dfdu(v_inp))))
            v_inp_neg, v_inp_pos = v_inp.roll(1), v_inp.roll(-1)
            F_pos = self.godunovFlux(v_inp, v_inp_pos)
            F_neg = self.godunovFlux(v_inp_neg, v_inp)
            v_out = v_inp - (dt/dx)*(F_pos - F_neg)
            self.data[i,0,:] = v_inp[:]
            self.data[i,1,:] = v_out[:]

    def godunov_flux(self, U_L, U_R):
        U_L, U_R = torch.flatten(U_L), torch.flatten(U_R)
        arr = torch.linspace(0,1,resolution)
        arr_mesh, U_L_mesh = torch.meshgrid(arr, U_L)
        _, U_R_mesh = torch.meshgrid(arr, U_R)
        diff = (U_R_mesh-U_L_mesh)
        flip = (diff < 0)*(-1)
        arr_mesh_flip = torch.abs(arr_mesh + flip)
        U_min = torch.min(U_L, U_R).reshape(U_L.size(),1)
        arr_scaled = torch.add(
            arr_mesh_flip*torch.abs(diff), U_min
        )
        arr_f = self.f(arr_scaled)
        flux_min = torch.min(arr_f, axis=0)[0]
        flux_max = torch.max(arr_f, axis=0)[0]
        cond = torch.stack(( (U_L < U_R), (U_R <= U_L) ))
        cond_shape = cond.shape[1:]
        cond = cond.reshape(cond.size()[0], flux_min.size()[-1])
        flux = (flux_min*cond[0] + flux_max*cond[1])
        return flux.reshape(cond_shape)

    def save(self, destination, filename):
        torch.save(self.data, '[destination]/[filename]')

    def load(self, destination, filename):
        self.data = torch.load('[destination]/[filename]')
```

Listing A.11: Data generator for one-dimensional experiments with physics-informed L^1-loss.

A.6 Two-dimensional data generator

This section contains two classes, namely the fine-mesh algorithm, Listing A.13, and the data-generator class which uses this algorithm, Listing A.12.

```
class Dataset_2D:
    def __init__(self, M, N, f, dfdu, g, dgdu):
        self.M = M
        self.f, self.dfdu = f, dfdu
        self.g, self.dgdu = g, dgdu
        self.N = N
        self.T = None
        self.data = torch.zeros(self.M, 8)
        self.data[:,:-2] = torch.randn(self.M,6)
        self.flux = []
        self.dx = 1.0
        self.dy = 2/3

    def set_T(self, u0):
        num = np.min((self.dx,self.dy))
        denom = 2*torch.max(np.sqrt(self.dfdu(u0)**2 + self.dgdu(u0)**2))
        self.T = float(np.random.rand(1)*float(num/denom))

```

```
    def set_F(self, flux, ind):
        flux_of_interest = flux[:,self.N:2*self.N,flux.size(2)//2]
        self.flux.append(flux_of_interest)
        self.data[ind, -1] = torch.mean(flux_of_interest)


    def create(self, cuda_num = 0):
        for i, u0 in enumerate(self.data[:,:-2]):
            self.set_T(u0=u0)
            self.data[i,-2] = self.T
            solver = FineMeshSolver(
                u0 = u0,
                N = self.N,
                f = self.f,
                dfdu = self.dfdu,
                g = self.g,
                dgdu = self.dgdu,
                T = self.T/np.min((self.dx,self.dy)),
                cuda_num=cuda_num
            )
            solver.solve
            flux = torch.stack(solver.god_flux[0])
            self.set_F(flux=flux, ind=i)

    def save(self, destination, filename):
        torch.save(self.data, '[destination]/[filename]')

    def load(self, destination, filename):
        self.data = torch.load('[destination]/[filename]')
```

Listing A.12: Data generator for two-dimensional experiments.

```
class FineMeshSolver:
    def __init__(self, u0, N, f, dfdu, g, dgdu, T, cuda_num = 0):
        self.u0 = u0
        self.N = N
        self.f, self.dfdu = f, dfdu
        self.g, self.dgdu = g, dgdu
        self.T = T
        self.C = 0.5
        self.god_flux_mesh_size = 100
        self.x_size, self.y_size = 2*self.N, 3*self.N
        self.x, self.dx = np.linspace(
            -1, 1, self.x_size, retstep=True
        )
        self.y, self.dy = np.linspace(
            -1, 1, self.y_size, retstep=True
        )
        self.dt = None
        self.god_flux = [[],[]]
        init_mesh = torch.zeros((self.y_size, self.x_size))
        init_mesh[:self.N,:self.N] = self.u0[0]
        init_mesh[:self.N,self.N:] = self.u0[1]
        init_mesh[self.N:2*self.N,:self.N] = self.u0[2]
        init_mesh[self.N:2*self.N,self.N:] = self.u0[3]
        init_mesh[2*self.N:,:self.N] = self.u0[4]
        init_mesh[2*self.N:,self.N:] = self.u0[5]
        self.u = [init_mesh]

    def set_dt(self):
        num = self.C*np.min((self.dx,self.dy))
        denom = torch.max( torch.sqrt(
            self.dfdu(self.u[-1])**2 + self.dgdu(self.u[-1])**2
```

```
        ) )
        dt = num/denom
        Nt = int(np.ceil(self.T/dt))
        _, self.dt = np.linspace(
            0, self.T, Nt, retstep=True
        )

    @property
    def solve(self):
        t = 0
        while t < self.T:
            self.set_dt()
            t += self.dt
            if t >= self.T:
                self.dt -= (t-self.T)
                t = self.T
            self.set_godunov_flux()
            self.godunov()

    def godunov(self):
        god_flux_f = self.god_flux[0][-1]
        god_flux_g = self.god_flux[1][-1]
        Cx, Cy = self.dt/self.dx, self.dt/self.dy

        u_next = torch.zeros(self.u[-1].size())
        u_next[:,:] = (self.u[-1][:,:]) \
            - Cx*( god_flux_f[:,1:] - god_flux_f[:,:-1]
            ).reshape((self.u[-1].size(0),self.u[-1].size(1))) \
            - Cy*( god_flux_g[1:,:] - god_flux_g[:-1,:]
            ).reshape((self.u[-1].size(0),self.u[-1].size(1)))
        self.u.append(u_next)

    def set_godunov_flux(self):
        u = torch.empty(self.u[-1].size(0)+2, self.u[-1].size(1)+2)
        u[1:-1, 1:-1] = self.u[-1]
        u[1:-1, 0] = u[1:-1, 1]
        u[1:-1,-1] = u[1:-1,-2]
        u[0, :] = u[1, :]
        u[-1,:] = u[-2,:]

        F = torch.empty(self.u[-1].size(0), self.u[-1].size(1)+1)
        F[:,:] = self.godunov_flux(
            u[1:-1, :-1], u[1:-1, 1:], func=self.f
        ).reshape((self.u[-1].size(0), self.u[-1].size(1)+1))
        self.god_flux[0].append(F)

        G = torch.empty(self.u[-1].size(0)+1, self.u[-1].size(1))
        G[:,:] = self.godunov_flux(
            u[:-1, 1:-1],u[1:, 1:-1], func=self.g
        ).reshape((self.u[-1].size(0)+1, self.u[-1].size(1)))
        self.god_flux[1].append(G)

    def godunov_flux(self, U_L, U_R, func):
        U_L, U_R = torch.flatten(U_L).type(torch.float64), torch.flatten(
 U_R).type(torch.float64)
        arr = torch.linspace(0,1,self.god_flux_mesh_size)
        arr_mesh, U_L_mesh = torch.meshgrid(arr, U_L)
        _, U_R_mesh = torch.meshgrid(arr, U_R)
        diff = (U_R_mesh-U_L_mesh)
        flip = (diff < 0)*(-1)
        arr_mesh_flip = torch.abs(arr_mesh + flip)
        U_min = torch.min(U_L, U_R).reshape(U_L.size(),1)
```

```
        arr_scaled = torch.add(
            arr_mesh_flip*torch.abs(diff), U_min
        )
        arr_f = func(arr_scaled)
        flux_min = torch.min(arr_f, axis=0)[0]
        flux_max = torch.max(arr_f, axis=0)[0]
        cond = torch.stack(( (U_L < U_R), (U_R <= U_L) ))
        cond_shape = cond.shape[1:]
        cond = cond.reshape(cond.size()[0], flux_min.size()[-1])
        flux = (flux_min*cond[0] + flux_max*cond[1]).reshape(cond_shape)
        return flux
```

Listing A.13: Fine-mesh solver for two-dimensional experiments.

A.7 Additional

The following code snippets are additional lines of code essential for the package. This includes imported packages, GPU-specifier and initial functions used.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import os
```

Listing A.14: Imports.

```
self.device = "cuda" if torch.cuda.is_available() else "cpu"
# then use obj.to(self.device) to cast object to GPU/CPU memory
```

Listing A.15: Initialize computation device (GPU/CPU).

```
class InitialFunc:
    def __init__(self, func_name):
        self.func_name = func_name
        self.func_menu = ['heavi', 'heavi_rev', 'heavi_scaled', 'sine', '
    sigmoid_cosine', 'leveque']
        self.func = None
        self.set_func()

    def set_func(self):
        if self.func_name in self.func_menu:
            eval("self."+self.func_name+"()")

    def sigmoid_cosine(self):
        def splash(mesh):
            r = np.sqrt(mesh[:,:,0]*mesh[:,:,0] + mesh[:,:,1]*mesh[:,:,1])
            return (np.exp(r)/(1 + np.exp(r)))*np.cos(4*np.pi*r)
        self.func = lambda mesh: splash(mesh)

    def leveque(self):
        def cs(mesh):
            x = np.logical_and(np.greater(mesh[:,:,1], 0.1),   np.less(mesh
    [:,:,1], 0.6))
            y = np.logical_and(np.greater(mesh[:,:,0], -0.25), np.less(mesh
    [:,:,0], 0.25))
            sq = np.round(1.0*np.logical_and(x, y))
            r = np.sqrt((mesh[:,:,1] + 0.45)**2 + mesh[:,:,0]**2)
            circ = (1-r/0.35)*np.less(r, 0.35)
```

```
25              return (sq + circ)
26          self.func = lambda mesh: cs(mesh)
27
28      def heavi(self):
29          def heaviside(mesh):
30              low_mesh = np.logical_and(np.greater_equal(mesh,-1.0), np.less(
        mesh,0.0))
31              hig_mesh = np.logical_and(np.greater_equal(mesh,0.0), np.
        less_equal(mesh,1.0))
32
33              low_x, low_y = low_mesh[:,:,0], low_mesh[:,:,1]
34              hig_x, hig_y = hig_mesh[:,:,0], hig_mesh[:,:,1]
35
36              func = (0.0)*np.logical_and(low_x, low_y) \
37                  + (0.0)*np.logical_and(low_x, hig_y) \
38                  + 1.0*np.logical_and(hig_x, low_y) \
39                  + 1.0*np.logical_and(hig_x, hig_y)
40              return func
41          self.func = lambda mesh: heaviside(mesh)
42
43      def heavi_rev(self):
44          def heaviside_rev(mesh):
45              low_mesh = np.logical_and(np.greater_equal(mesh,-1.0), np.less(
        mesh,0.0))
46              hig_mesh = np.logical_and(np.greater_equal(mesh,0.0), np.
        less_equal(mesh,1.0))
47              low_x, low_y = low_mesh[:,:,0], low_mesh[:,:,1]
48              hig_x, hig_y = hig_mesh[:,:,0], hig_mesh[:,:,1]
49              func = (1.0)*np.logical_and(low_x, low_y) \
50                  + (1.0)*np.logical_and(low_x, hig_y) \
51                  + 0.0*np.logical_and(hig_x, low_y) \
52                  + 0.0*np.logical_and(hig_x, hig_y)
53              return func
54          self.func = lambda mesh: heaviside_rev(mesh)
55
56      def heavi_scaled(self):
57          def heaviside_scaled(mesh):
58              low_mesh = np.logical_and(np.greater_equal(mesh,-1.0), np.less(
        mesh,0.0))
59              hig_mesh = np.logical_and(np.greater_equal(mesh,0.0), np.
        less_equal(mesh,1.0))
60              low_x, low_y = low_mesh[:,:,0], low_mesh[:,:,1]
61              hig_x, hig_y = hig_mesh[:,:,0], hig_mesh[:,:,1]
62              func = (-1.0)*np.logical_and(low_x, low_y) \
63                  + (-1.0)*np.logical_and(low_x, hig_y) \
64                  + 1.0*np.logical_and(hig_x, low_y) \
65                  + 1.0*np.logical_and(hig_x, hig_y)
66              return func
67          self.func = lambda mesh: heaviside_scaled(mesh)
68
69      def sine(self):
70          self.func = lambda mesh: np.sin(4*np.pi*mesh[:,:,0])
71
72      def get_func(self):
73          return self.func
74
75      def get_name(self):
76          return self.func_name
```

Listing A.16: All initial functions used throughout this thesis.

Bibliography

[Car60] Carnot, S. *Reflections on the motive power of fire*. And other papers on the second law of thermodynamics by É. Clapeyron and R. Clausius. Edited with an introduction by E. Mendoza. Dover Publications, Inc., New York, 1960, pp. xxii+152.

[CFL67] Courant, R., Friedrichs, K., and Lewy, H. "On the partial difference equations of mathematical physics." In: *IBM J. Res. Develop.* vol. 11 (1967), pp. 215–234.

[CM80] Crandall, M. G. and Majda, A. "Monotone difference approximations for scalar conservation laws." In: *Math. Comp.* vol. 34, no. 149 (1980), pp. 1–21.

[Cyb89] Cybenko, G. "Approximation by superpositions of a sigmoidal function." In: *Math. Control Signals Systems* vol. 2, no. 4 (1989), pp. 303–314.

[Ein88] Einfeldt, B. "On Godunov-type methods for gas dynamics." In: *SIAM J. Numer. Anal.* vol. 25, no. 2 (1988), pp. 294–318.

[Eva10] Evans, L. C. *Partial differential equations*. Second. Vol. 19. Graduate Studies in Mathematics. American Mathematical Society, Providence, RI, 2010, pp. xxii+749.

[God59] Godunov, S. K. "A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics." In: *Mat. Sb. (N.S.)* vol. 47 (89) (1959), pp. 271–306.

[Gos14] Gosse, L. "A two-dimensional version of the Godunov scheme for scalar balance laws." In: *SIAM J. Numer. Anal.* vol. 52, no. 2 (2014), pp. 626–652.

[GR91] Godlewski, E. and Raviart, P.-A. *Hyperbolic systems of conservation laws*. Vol. 3/4. Mathématiques & Applications (Paris) [Mathematics and Applications]. Ellipses, Paris, 1991, p. 252.

[Hor91] Hornik, K. "Approximation capabilities of multilayer feedforward networks." In: *Neural Networks* vol. 4, no. 2 (1991), pp. 251–257.

[HR15] Holden, H. and Risebro, N. H. *Front tracking for hyperbolic conservation laws*. Second. Vol. 152. Applied Mathematical Sciences. Springer, Heidelberg, 2015, pp. xiv+515.

[HTF09] Hastie, T., Tibshirani, R., and Friedman, J. *The elements of statistical learning*. Second. Springer Series in Statistics. Data mining, inference, and prediction. Springer, New York, 2009, pp. xxii+745.

[KB15] Kingma, D. P. and Ba, J. "Adam: A Method for Stochastic Optimization." In: Published as a conference paper at the 3rd International Conference for Learning Representations (ICLR), San Diego. 2015. arXiv: 1412.6980.

[KB20] Kratsios, A. and Bilokopytov, E. "Non-Euclidean Universal Approximation." In: Published as a conference paper at the 34th Conference on Neural Information Processing Systems (NeurIPS). 2020. arXiv: 2006.02341.

[KL20] Kidger, P. and Lyons, T. J. "Universal Approximation with Deep Narrow Networks." In: Published as a conference paper at the 33rd Conference on Learning Theory (COLT). 2020. arXiv: 1905.08539.

[Kru70] Kružkov, S. N. "First order quasilinear equations with several independent variables." In: *Mat. Sb. (N.S.)* vol. 81 (123) (1970), pp. 228–255.

[Lax73] Lax, P. D. *Hyperbolic systems of conservation laws and the mathematical theory of shock waves*. Conference Board of the Mathematical Sciences Regional Conference Series in Applied Mathematics, No. 11. Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1973, pp. v+48.

[Lee84] Leer, B. van. "On the Relation Between the Upwind-Differencing Schemes of Godunov, Engquist–Osher and Roe." In: *SIAM Journal on Scientific and Statistical Computing* vol. 5 (1984), pp. 1–20.

[LeV02] LeVeque, R. J. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002, pp. xx+558.

[LeV85] LeVeque, R. J. "A large time step generalization of Godunov's method for systems of conservation laws." In: *SIAM J. Numer. Anal.* vol. 22, no. 6 (1985), pp. 1051–1073.

[MP43] McCulloch, W. S. and Pitts, W. "A logical calculus of the ideas immanent in nervous activity." In: *Bull. Math. Biophys.* vol. 5 (1943), pp. 115–133.

[MW99] McDonald, J. N. and Weiss, N. A. *A course in real analysis*. Biographies by Carol A. Weiss. Academic Press, Inc., San Diego, CA, 1999, pp. xx+745.

[Noe18] Noether, E. "Invariante Variationsprobleme." In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* vol. 1918 (1918), pp. 235–257.

[RPK19] Raissi, M., Perdikaris, P., and Karniadakis, G. E. "Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." In: *J. Comput. Phys.* vol. 378 (2019), pp. 686–707.

[Zho+17] Zhou, L. et al. “The Expressive Power of Neural Networks: A View from the Width.” In: Published as a conference paper at the 31st Conference on Neural Information Processing Systems (NeurIPS). 2017. arXiv: 1709.02540.