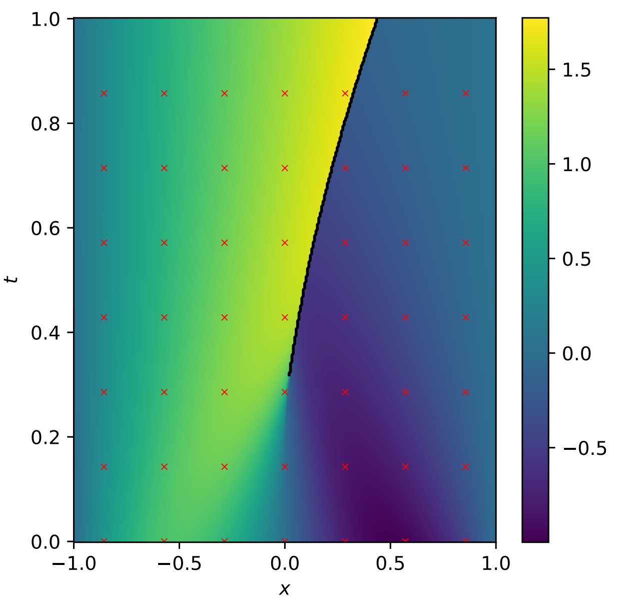
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Bendik Skundberg Waade

Numerics-informed neural networks and inverse problems with hyperbolic balance laws

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjetil Olsen Lye
Co-supervisor: Espen Robstad Jakobsen
June 2023



Bendik Skundberg Waade

Numerics-informed neural networks and inverse problems with hyperbolic balance laws

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjetil Olsen Lye
Co-supervisor: Espen Robstad Jakobsen
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Preface

This thesis marks the end of my Master of Science (M.Sc.) in Applied Physics and Mathematics with specialization in Industrial Mathematics. The degree was carried out at the Department of Mathematical Sciences (IMF) at the Norwegian University of Science and Technology (NTNU) in Trondheim. The thesis was written under the supervision of Kjetil Olsen Lye at Sintef.

I would like to thank Kjetil for introducing me to a vast and exciting new field, and for his invaluable input and guidance. I also want to thank my friends for tirelessly indulging me in discussions about code bugs and implementation details, and for teaching me the art of table tennis and the value of lunch breaks. Finally, thank you to my parents and my girlfriend for offering support throughout my degree, especially through the years of the pandemic.

Abstract

We introduce *numerics-informed neural networks* (NINN), a framework designed to learn source terms in hyperbolic partial differential equations. NINNs combine classic numerical schemes with neural networks to accurately solve inverse problems related to these equations. In this work, we present the general framework, along with a convergence result that relates the source approximation error to the training loss. To implement training, we develop two specialized algorithms tailored for different usecases, and evaluate their performance on a hyperbolic system. The *sweeping* algorithm, in particular, enables the use of NINNs in scenarios where complete solution measurements are unavailable. We extensively test the proposed method on a diverse set of problems, comparing its performance under varying data quantities. In combination with this thesis, a Python library for performing differentiable simulation of hyperbolic partial differential equations was developed. Our work showcases the potential of NINNs as a powerful tool for solving inverse problems with hyperbolic partial differential equations, paving the way for future advancements in this field.

Vi introduserer *numerikkbevisste nevrale nettverk* (NINN), et rammeverk utviklet for å lære kildeledd i hyperbolske partielle differensialligninger. NINN kombinerer klassiske numeriske metoder med nevrale nettverk for å løse inverse problemer knyttet til disse ligningene. I dette arbeidet presenterer vi det generelle rammeverket sammen med et konvergensresultat som relaterer feilen i kildeapproksimasjonen til treningsfeilen. For å håndtere treningen utvikler vi to spesialiserte algoritmer tilpasset ulike bruksområder, og evaluerer ytelsen på et hyperbolsk system. Spesielt muliggjør *sweeping*-algoritmen bruk av NINN i scenarier der komplette målinger av løsningen ikke er tilgjengelige. Vi tester den foreslåtte metoden grundig på en rekke ulike problemer og sammenligner ytelsen under varierende mengder data. I tillegg er det utviklet et Python-bibliotek for differensierbar simulering av hyperbolske partielle differensialligninger i kombinasjon med denne avhandlingen. Vårt arbeid viser potensialet til NINN som et kraftig verktøy for å løse inverse problemer med hyperbolske partielle differensialligninger og legger grunnlaget for fremtidige fremskritt på dette feltet.

Contents

Chapter 1

Introduction

Many physical models useful in engineering are based on the concept of a *balance law*. Basically, these laws model the change in some interesting quantity by considering its movement around the domain due to forces acting on it, in combination with potential sources of creation or destruction of the quantity. Such models give rise to *partial differential equations* (PDE), a class of equations that in many cases are hard or even impossible to solve exactly. An important subfield of applied mathematics is therefore to construct efficient numerical methods for approximating solutions to PDEs, and during the past 300 years mathematicians have built a comprehensive toolkit of methods for dealing with PDEs numerically. These methods generally solve the *forward problem* related to PDEs: given a PDE, find or approximate a representation of the solution.

In certain cases, we are not interested in finding a solution to a given PDE. Instead we seek to use information about the solution, such as measurements, to infer unknown properties of the PDE itself. Examples include finding the mass distribution of the earth given measurements of the gravitational force on the surface, and inferring the shape of an object given its resonant frequencies. The latter was famously covered by Marc Kac in his 1966 lecture named "Can one hear the shape of a drum?" [21]. These types of problems are called *inverse*, and their analysis is complicated and computationally demanding [19, 4].

Recently, Raissi et. al. [34] published a revolutionary framework for handling both forward and inverse problems related to PDEs based on *neural networks*, which are powerful machine learning models capable of representing general functions in high dimensional spaces. Their method, called *physics-informed neural networks* (PINN), essentially sidestepped the established methods on the domain completely by providing a way of iteratively coercing a neural network to approximately solve a PDE globally. PINNs have also shown some promise for inverse problems [20, 34, 3], as unknown scalar parameters of the PDE can be included as learnable parameters in the network.

However, PINNs generally require the solution to be smooth. This is a problem when dealing with balance laws where advection is dominant, as these result in hyperbolic PDEs, a class of equations that are known to produce non-smooth solutions. There are ways of adapting PINNs to these cases; notably cPINNs [20], wPINNs [6], and perturbing the PDE slightly to avoid a hard discontinuity. Unfortunately, these adaptations can be difficult to work with in practice.

We propose a way of mediating some of these issues. Our method, named *numerics-informed neural networks* (NINN), use established and powerful numerical methods within the framework of PINNs in order to solve general inverse problems related to hyperbolic PDEs arising from balance laws. We start by presenting some general theory regarding balance laws, hyperbolic equations, and neural networks, before thoroughly testing the performance of NINNs on a selection of test problems with increasing complexity. We will show that NINNs are capable of accurately solving inverse problems in cases where the PDE solution develops discontinuities, where the unknown target function is non-smooth, and even when the solution measurements are incomplete. Finally we discuss general insights developed throughout this work before suggesting interesting ways of further developing this type of method.

Chapter 2

Qualitative behaviour of balance laws

2.1 Motivation

Let Ω be a domain, and $u : \mathbb{R}_+ \times \Omega \to \mathbb{R}$ be some quantity distributed across Ω. We can think of u as the concentration of a solute in a container, the heat distribution of some body, or density of cars along a road. A simple model for the distribution of u can be constructed by considering two factors contributing to change in u, its *flux* and its *sources*. More precisely, looking at some subset $\omega \subseteq \Omega$ we relate the time rate of change of the total amount of u in ω to the amount of u exiting ω plus the amount of u created inside ω. Mathematically, we may describe this relation by the equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\omega u \,\mathrm{d}\omega = -\int_{\partial\omega} f \cdot n \,\mathrm{d}s + \int_\omega q \,\mathrm{d}\omega \,, \tag{2.1}$$

where $f = f(u)$ is the flux of u, n is the unit normal pointing out of ω, and q is the source function. We call (2.1) a *balance law in integral form.* Note that negative values for q signify the destruction of u, and negative source functions are sometimes called *sinks.* If $q \equiv 0$, we say that (2.1) is *homogeneous*, and we call it a *conservation law.* Assuming u is smooth and bounded almost everywhere in Ω, we can use the divergence theorem and the general Leibniz rule to write (2.1) as

$$\begin{aligned}\int_\omega u_t \,\mathrm{d}\omega &= -\int_\omega \nabla \cdot f \,\mathrm{d}\omega + \int_\omega q \,\mathrm{d}\omega \\ \int_\omega u_t + \nabla \cdot f \,\mathrm{d}\omega - q \,\mathrm{d}\omega &= 0.\end{aligned}$$

This must hold for any t and any measurable $\omega \subseteq \Omega$, which implies that the expression

$$u_t + \nabla \cdot f = q \tag{2.2}$$

holds almost everywhere on $\mathbb{R}_+ \times \Omega$. Equation (2.2) is called a *balance law in differential form*, and most of the equations we will study here is of this form. Mathematically, (2.2) is a hyperbolic partial differential equation. We often extend this definition to cases where $u : \mathbb{R}_+ \times \Omega \to \mathbb{R}^m$, in which case (2.2) is called hyperbolic if the Jacobian of f is diagonalizable over the real numbers. This is always true in the scalar case, when $m = 1$. Later we will see that solutions to hyperbolic PDEs can develop discontinuities known as *shocks*, breaking the assumption of smoothness of u. This will eventually lead us to relax our definition of a solution of (2.2), but it is important to know that in these cases, the actual description of the system is given by the integral form (2.1).

2.2 Characteristics of hyperbolic equations

We will from now on limit ourselves to the case of a single spatial dimension, although many of the methods described here also apply to more general cases. Initially, we also assume u is smooth. This assumption will be challenged in Section 2.3, but it allows us to define a useful method for solving (2.2) in simple cases. As the solution is smooth, the chain rule gives $(f(u))_x = f_u(u)u_x$, allowing us to rewrite (2.2) as

$$u_t + f_u u_x = q. \tag{2.3}$$

Let $x(t)$ be some path in the xt plane. We examine the value of u along this path. By the chain rule we have

$$\frac{\mathrm{d}}{\mathrm{d}t}u(t,x(t)) = u_t(t,x(t)) + \dot{x}(t)u_x(t,x(t)).$$

Notice that if $\dot{x}(t) = f_u(u(t,x(t)))$, we can use (2.3) to infer that

$$\frac{\mathrm{d}}{\mathrm{d}t}u = q \tag{2.4}$$

along $x(t)$. Paths $x(t)$ such that $\dot{x}(t) = f_u(u(t,x(t))$ are called *characteristic curves*, or just *characteristics*, of (2.3).

Now, if we impose an initial condition on the solution to (2.3) we get the initial value problem (IVP)

$$\begin{cases} u_t + f_u u_x = q \\ u(0,x) = u_0(x) \end{cases} \tag{2.5}$$

for known functions f, q, u_0. Let $x(t)$ be a characteristic starting in $x_0 \in \mathbb{R}$, that is $x(t)$ solves the IVP

$$\begin{cases} \dot{x}(t) = f_u \\ x(0) = x_0. \end{cases}$$

This is often called the *characteristic equation* corresponding to (2.5). Using (2.4) we get that

$$u|_{x(t)} = u_0(x_0) + \int_0^t q|_{x(s)}\,\mathrm{d}s\,, \tag{2.6}$$

where

$$u|_{x(t)} \equiv u(t,x(t)).$$

Thus, the value of u is known along the characteristic. In fact, we can obtain the solution in any point $(\tilde{t},\tilde{x})$ as long as there is some characteristic $x(t)$ of (2.5) such that $x(\tilde{t}) = \tilde{x}$. This solution approach is called the *method of characteristics*. Informally, (2.6) describes some initial value $u_0(x_0)$ being carried along the characteristic, picking up or losing magnitude along the way according to sources and sinks described by q.

The simplest example of a balance equation of the form (2.5) is the case of linear transport, where $f(u) = au$ for some constant $a \in \mathbb{R}$. This gives the IVP

$$\begin{cases} u_t + au_x = q \\ u(0,x) = u_0(x), \end{cases} \tag{2.7}$$

with corresponding characteristic equation

$$\begin{cases} \dot{x}(t) = a \\ x(0) = x_0. \end{cases} \tag{2.8}$$

The characteristics take the form

$$x(t) = x_0 + at,$$

and are plotted in Figure 2.1 for different x_0 and a.

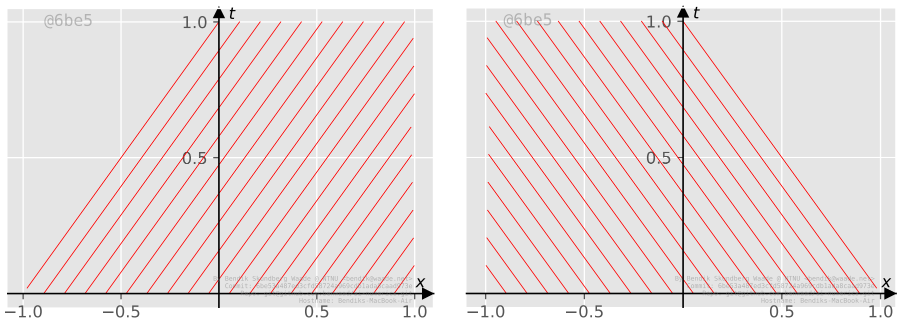

(a) $a = -1$ **(b)** $a = 1$

Figure 2.1: The characteristics of (2.7) for different values of a.

Along these lines, the solution is given by (2.6). Notice that every point in the xt halfplane is hit by exactly one characteristic. In fact, the characteristic hitting a given point $(\tilde{t}, \tilde{x})$ starts at

$$\tilde{x}_0 = \tilde{x} - a\tilde{t},$$

and is thus given by the equation

$$x(t) = \tilde{x} + a(t - \tilde{t}).$$

Substituting into (2.6) yields

$$u(\tilde{t}, \tilde{x}) = u_0(\tilde{x} - a\tilde{t}) + \int_0^{\tilde{t}} q(s, \tilde{x} + a(s - \tilde{t}))\,\mathrm{d}s\,. \tag{2.9}$$

That is, the value $u_0(\tilde{x}_0)$ is carried along the x axis with speed a, changed only by the compounded contributions of q along the way.

2.3 Weak solutions

Notice that the form of the characteristics in the case of linear transport were particularly nice. The fact that f_u was independent on the solution u meant we could easily find a closed form expression for the characteristic lines. In addition, every point in the xt halfplane were hit by exactly one of these lines. This is not true for general fluxes f, which can be illustrated by considering the flux function $f(u) = \frac{1}{2}u^2$, yielding the inviscid Burgers' equation

$$\begin{cases} u_t + (\frac{1}{2}u^2)_x = q \\ u(0, x) = u_0(x). \end{cases} \tag{2.10}$$

Now the characteristic equation reads

$$\begin{cases} \dot{x}(t) = u|_{x(t)} \\ x(0) = x_0, \end{cases}$$

that is, the shape of the characteristics depend on the solution value they carry. In order to find an expression for the characteristics, we first note that since

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(u|_{x(t)}\right) = q|_{x(t)},$$

along a characteristic $x(t)$, we have that

$$u|_{x(t)} = u_0(x_0) + \int_0^t q|_{x(s)}\,\mathrm{d}s\,,$$

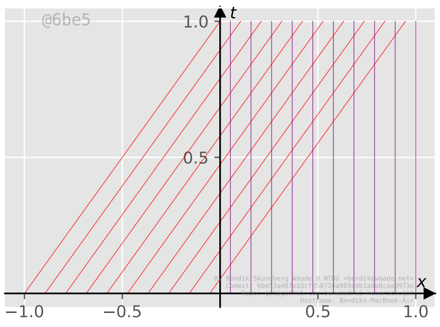


Figure 2.2: Characteristics of Burgers' equation with initial condition u_0^s. Characteristics starting in positive and negative x_0 are colored differently to signify that they are carrying different solution values.

just like in (2.6). We can substitute this into the characteristic equation to get

$$\begin{cases} \dot{x}(t) = u_0(x_0) + \int_0^t q|_{x(s)} \,\mathrm{d}s \\ x(0) = x_0. \end{cases}$$

Thus, the characteristics satisfy the implicit equation

$$x(t) = x_0 + u_0(x_0)t + \int_0^t \int_0^\tau q|_{x(s)} \,\mathrm{d}s \,\mathrm{d}\tau \,. \tag{2.11}$$

For simplicity, assume now that $q(t, x) = 0$, meaning the solution is constant along the characteristic lines. It is perfectly possible to consider a more general source function, but it contributes an unnecessary complication to the following example. The characteristics become

$$x(t) = x_0 + u_0(x_0)t, \tag{2.12}$$

which are straight lines with slope equal to the initial condition value at their x intercept. Choosing, for instance, the initial condition

$$u_0^s(x) = \begin{cases} 1 & x \leq 0 \\ 0 & x > 0 \end{cases}, \tag{2.13}$$

the characteristics will meet in the right half plane (see Figure 2.2). This means the points in the right half plane no longer have a unique characteristic, and therefore a unique solution value, associated with it. We call this phenomenon a *shock*, and we will later see that this causes a moving discontinuity, or *shockwave*, in the solution.

Hyperbolic equations with piecewise constant initial conditions containing a single discontinuity (such as (2.13)) are often called Riemann problems, and they are an important special case of the general equation (2.2). Finite volume methods for instance, an important class of numerical techniques for more general hyperbolic equations, are built by considering a superposition of Riemann problems on a grid on the domain. More on this in Section 3.

When beginning the discussion on characteristics in Section 2.2, we assumed the solution was smooth. However, even in the case of linear transport it is clear that this assumption breaks down when we use discontinuous initial data. We can see this from (2.9), which in the homogeneous case simply translates the initial data according to the wave speed a. One can imagine that by restricting ourselves to smooth initial data, we avoid the problem of discontinuities in the solution entirely. This is true for the linear flux discussed in Section 2.2. Unfortunately however, in the case of nonlinear flux, the emergence of shocks is not restricted to cases with discontinuous initial data. In fact, even when u_0 is infinitely smooth, as long as there is some point x_s for which $u_0'(x_s) < 0$, the corresponding solution to Burgers' equation will develop a discontinuity in finite time. We do not prove this fact here, but note that it prohibits a global classical solution to Burgers' equation for practically all interesting initial data.

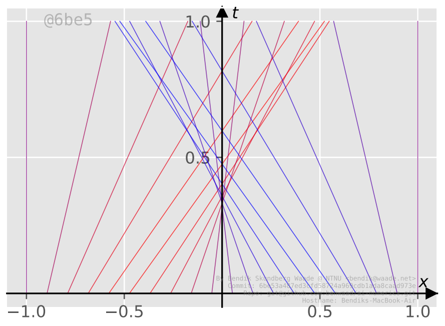


Figure 2.3: The characteristics of Burgers' equation with initial data given by (2.14) and no source.

As an example, consider the homogeneous Burgers' equation with initial data

$$u_0(x) = -\sin(\pi x). \tag{2.14}$$

The characteristics are plotted in Figure 2.3. Notice how the characteristics meet at $x = 0$ after some amount of time. The solution (we will shortly define precisely what we mean with the word "solution" in this case) at various time points can be seen in Figure 2.4. Notice how the two extrema move towards each other and collide to produce the discontinuity at the same point in time and space for which the characteristics collide.

In order to handle initial data that induce shocks, we need to introduce the concept of a *weak solution*. Let $\Omega = \mathbb{R}_+ \times \mathbb{R}$ and imagine there exists a smooth solution to the general 1D hyperbolic equation

$$\begin{cases} u_t + f(u)_x = q & \text{on } \Omega, \\ u(0,x) = u_0(x) \end{cases}. \tag{2.15}$$

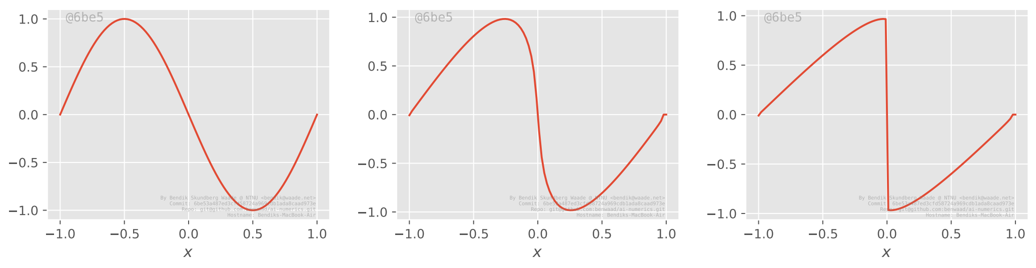
Define the space $C_c^1(\Omega)$ of continuously differentiable functions on compact support in Ω, and let $\varphi \in C_c^1(\Omega)$. In particular, this means that there exists a $K \in \mathbb{R}$ such that for all $t, x > K$, $\varphi(t,x) = 0$. We call such a function a *test function*. Multiplying (2.15) by φ and integrating over Ω, we get

$$\begin{aligned} \int_\Omega u_t\varphi + f(u)_x\varphi &= \int_\Omega q\varphi \\ \int_{\mathbb{R}}\int_{\mathbb{R}_+} u_t\varphi + \int_{\mathbb{R}_+}\int_{\mathbb{R}} f(u)_x\varphi &= \int_\Omega q\varphi \\ \int_{\mathbb{R}}\left[[u\varphi]_0^\infty - \int_{\mathbb{R}_+} u\varphi_t\right] + \int_{\mathbb{R}_+}\left[[f(u)\varphi]_{-\infty}^\infty - \int_{\mathbb{R}} f(u)\varphi_x\right] &= \int_\Omega q\varphi \\ \int_\Omega u\varphi_t + f(u)\varphi_x &= -\int_\Omega q\varphi - \int_{\mathbb{R}} u_0\varphi_0. \end{aligned} \tag{2.16}$$

Here $\varphi_0(x) = \varphi(0,x)$. We call $u \in L^\infty(\Omega)$ a *weak solution* to (2.15) if relation (2.16) is satisfied for all $\varphi \in C_c^1(\Omega)$. If u satisfies (2.15) pointwise, we call it a *strong* or *classical solution*. It is clear that any strong solution to (2.15) is also a weak solution, and it can be proven that if $u \in C^1(\Omega)$ is a weak solution, then it is also a strong solution. The difference is that weak solutions are not required to be differentiable or even continuous, they only need to be in $L^\infty(\Omega)$.

2.4 Rankine-Hugoniot and entropy conditions

The concept of weak solutions allow us to talk about discontinuous functions as solutions to (2.15). However, we still need to find them. In particular we need to be able to describe the shockwave, that is the discontinuity in the solution that arises from the collision of characteristics. Recall that



(a) $t = 0$ **(b)** $t = 0.25$ **(c)** $t = 0.5$

Figure 2.4: The solution to the homogeneous Burgers' equation with sinusoidal initial data for three different time points.

when u is discontinuous, the physical interpretation is described by the integral form (2.1). The problem with weak solutions to the differential form (2.2) is that they are in general not unique. The physics they model, however, are deterministic. Thus we need to find a set of constraints that reduce the number of weak solutions considered "physical". The first, known as the *Rankine-Hugoniot* condition, can be derived by considering conservation over a domain surrounding the shockwave. The following discussion is based on example 1.3 in [15].

Assume for a moment that $q(t, x) = 0$, and that a weak solution u to (2.15) has an isolated shockwave moving along the curve $x = \gamma(t)$. Isolated means there exist a neighborhood around γ in which u satisfies (2.15) classically on each side of γ. Let D be such a neighborhood, meaning γ intersects it and separates it into two regions D^+ and D^- where u is a strong solution, see Figure 2.5. Importantly, the trace of u along γ is different in D^+ and D^-. Let φ be a test function whose support is contained in D. Then

$$\begin{aligned} 0 &= \int_D u\varphi_t + f(u)\varphi_x \\ &= \int_{D^+} [u\varphi_t + f(u)\varphi_x] + \int_{D^-} [u\varphi_t + f(u)\varphi_x] \, . \end{aligned} \tag{2.17}$$

Using the fact that u is a strong solution in D^+, we get

$$\begin{aligned} \int_{D^+} u\varphi_t + f(u)\varphi_x &= \int_{D^+} u\varphi_t + f(u)\varphi_x + (u_t + (f(u))_x)\varphi \\ &= \int_{D^+} (u\varphi)_t + (f(u)\varphi)_x \\ &= \int_{D^+} [\partial_t, \partial_x]^T \cdot [u\varphi, f(u)\varphi]^T \\ &= \int_{\partial D^+} \varphi[u, f(u)]^T \cdot n^+, \end{aligned} \tag{2.18}$$

where we used the divergence theorem to obtain the last relation. Here n^+ is the unit normal pointing out of ∂D^+. A similar computation can be done for ∂D^-, yielding

$$\int_{D^-} u\varphi_t + f(u)\varphi_x = \int_{\partial D^-} \varphi[u, f(u)]^T \cdot n^-.$$

Since φ has support contained in D, it is zero everywhere on ∂D^+ except along γ. Define $\gamma_\varphi = \gamma \cap \operatorname{supp}(\varphi)$. Then,

$$\int_{\gamma_\varphi} \varphi[u^+, f(u^+)]^T \cdot n^+ = \int_{\gamma_\varphi} \varphi[u^-, f(u^-)]^T \cdot n^+, \tag{2.19}$$

where u^+ and u^- denote the trace of u along γ when restricted to respectively D^+ and D^-. Notice that we use n^+ on both sides of (2.19). This is because $n^+ = -n^-$ along γ_φ, which combined with (2.17) yields (2.19). Since $n^+ = k[-\gamma'(t), 1]$ for some normalizing constant k,

$$\int_I (-u^+\gamma' + f(u^+))\varphi = \int_I (-u^-\gamma' + f(u^-))\varphi, \tag{2.20}$$

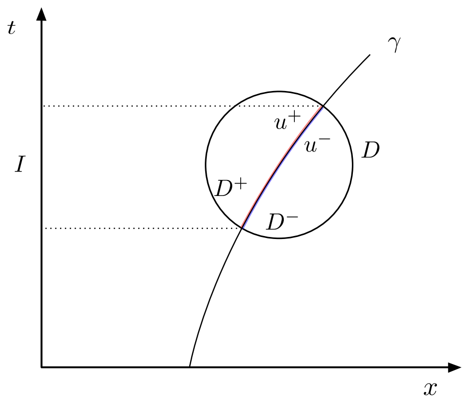


Figure 2.5: Diagram showing the setup for the Rankine-Hugoniot condition. We have strong solutions u^+ and u^- in respectively D^+ and D^-, separated by the shockwave γ.

where $I = \{t \in \mathbb{R}^+ : \gamma(t) \in \gamma_\varphi\}$ is drawn in Figure 2.5. Since φ and D (and thus I) are arbitrary, the integrands must be equal. Rearranging, we find

$$\frac{\mathrm{d}\gamma}{\mathrm{d}t} = \frac{f(u^+) - f(u^-)}{u^+ - u^-}. \tag{2.21}$$

We call (2.21) the Rankine-Hugoniot condition.

The Rankine-Hugoniot condition gives a condition that the shockwave must satisfy in order for it to be a physically acceptable weak solution. Revisiting the homogeneous Burgers' equation with initial data u_0^s defined in (2.13), we apply the condition to find that

$$\begin{aligned}\frac{\mathrm{d}\gamma}{\mathrm{d}t} &= \frac{\frac{1^2}{2} - \frac{0^2}{2}}{1-0}\\ &= \frac{1}{2}.\end{aligned}$$

Noticing that $\gamma(0) = 0$, we can solve this to find the shockwave path

$$x = \gamma(t) = \frac{1}{2}t. \tag{2.22}$$

The characteristics carrying different values are now separated by the shockwave, and every point in the xt halfplane outside of the shockwave have a unique solution value associated to it. Specifically, the weak solution in this case is given by

$$u^s(t,x) = \begin{cases} 1 & x < \frac{1}{2}t \\ 0 & x > \frac{1}{2}t \end{cases}. \tag{2.23}$$

The characteristics of this solution, along with the shockwave, are plotted in Figure 2.6.

In summary, the collision of characteristic lines produce shocks in the solution, and these shocks propagate according to the Rankine-Hugoniot condition (2.21). However, we do not yet have a way of handling areas of the domain containing no characteristics. Define

$$u_0^r(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}, \tag{2.24}$$

and notice that using this as the initial condition in the homogeneous Burgers' equation will produce an area in the domain that no characteristics ever reach. This case is illustrated in Figure 2.7, which corresponds to two wavefronts moving away from each other.

Recall that an issue with dealing with weak solutions is that they are generally not unique. For instance, the function

$$\tilde{u}(t,x) = \begin{cases} 0 & x \leq \frac{1}{2}t \\ 1 & x > \frac{1}{2}t \end{cases} \tag{2.25}$$

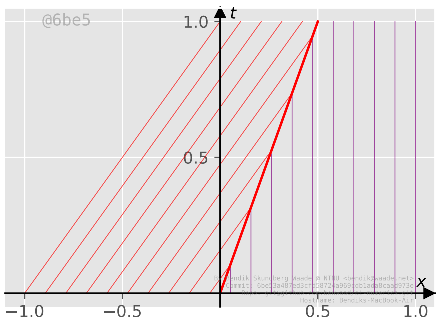


Figure 2.6: Characteristics of Burgers' equation with initial condition u_0^s, now with a shockwave satisfying the Rankine-Hugoniot condition separating them.

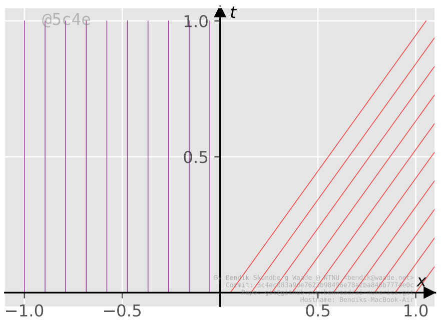


Figure 2.7: Characteristics of Burgers' equation equation with initial condition u_0^r.

is a weak solution to the homogeneous Burgers' equation with initial condition u_0^r. However, this solution leads to characteristic lines emanating out of the shock, see Figure 2.8(a). This is problematic as it means the solution is not stable to perturbations. Specifically, by adding a small viscosity term to (2.10), yielding

$$u_t + (\frac{1}{2}u^2)_x = \epsilon u_{xx}, \tag{2.26}$$

we change the solution dramatically. We would like the solution of the inviscid Burgers' equation to be the limit of (2.26) as $\epsilon \to 0$, so this instability with respect to the addition of viscosity is undesirable in a physical solution.

In order to avoid this instability, we can require that no characteristics move out of the shock. For a Riemann problem with shock position γ, left value u_l, and right value u_r, this means requiring

$$f'(u_l) > \frac{\mathrm{d}\gamma}{\mathrm{d}t} > f'(u_r). \tag{2.27}$$

Equation (2.27) is called the *Lax entropy condition.* We will discuss this name at the end of this section, but notice for now that $\tilde{u}$ breaks this condition.

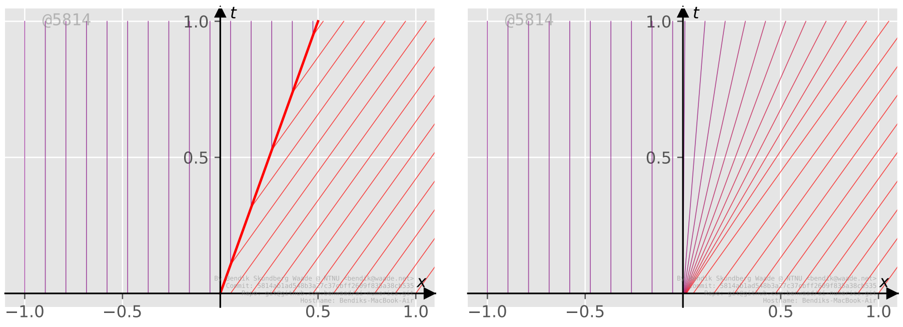
We can construct a solution that satisfies (2.27) by noticing that solutions to the general homogeneous scalar hyperbolic equation are scale invariant. That is, let u is a solution to

$$u_t + f(u)_x = 0, \tag{2.28}$$

and define $u^\lambda(t,x) = u(\lambda t, \lambda x)$. Then

$$\begin{aligned}\Big[u_t^\lambda + f(u^\lambda)_x\Big]_{(t,x)} &= \Big[\lambda u_t + \lambda f(u)_x\Big]_{(\lambda t, \lambda x)} \\ &= 0,\end{aligned}$$

so u^λ is also a solution to (2.28) (with initial condition $u^\lambda(0,x) = u(0,\lambda x)$). It is therefore reasonable to search for solutions $u(t,x) = w(x/t) = w(\xi)$. Assume for a moment that $f \in C^2$ and



(a) Unstable ($\tilde{u}$).

(b) Stable (u^r).

Figure 2.8: Characteristic lines for different weak solutions to Burgers' equation.

f' is monotone, and insert $w(\xi)$ into (2.28). We get

$$-\frac{x}{t^2}w_\xi + \frac{1}{t}f'(w)w_\xi = 0. \tag{2.29}$$

Since f' is monotone, we can invert it to obtain

$$w = (f')^{-1}(\xi). \tag{2.30}$$

As an example, take Burgers' equation with (2.24) as initial condition. The characteristics are $x = x_0$ for $x_0 < 0$, and $x = x_0 + t$ for $x_0 > 0$. Here $f'(u) = u$, so (2.30) gives us the solution $w(t,x) = \frac{x}{t}$. Using this solution to connect the two wavefronts, we obtain

$$u^r(t,x) = \begin{cases} 0 & x < 0 \\ \frac{x}{t} & 0 \leq x < t \\ 1 & t \leq x \end{cases}. \tag{2.31}$$

This solution is called the *rarefaction wave* solution. Note that u^r is continuous, and it satisfies (2.27) since it lacks a propagating shock. In fact, this solution is the limit as $\epsilon \to 0$ of the solution of (2.26). A discussion on this solution and its properties can be found in [40]. The characteristics for the rarefaction wave can be seen in Figure 2.8(b).

Recall that during our motivation for the rarefaction wave solution, we claimed it was reasonable to look for solutions $u(t,x) = w(\xi)$ with $\xi = x/t$. Notice that both the shockwave (2.23) and the rarefaction wave (2.31) can be formulated as functions only of the variable ξ. This will be important later in Section 3.

When f' is not monotone, we must replace it by a monotone function which in some sense is *close* to f'. We will not do that here, but refer the reader theorem 2.2 in [15], where Holden and Risebro shows that given some regularity conditions, the rarefaction solution satisfies (2.27) when replacing f by its so called *convex envelope*.

In reality, both the Rankine-Hugoniot condition (2.21) and the Lax entropy condition (2.27) are special cases of the more general *Kružkov entropy condition*. Consider the hyperbolic PDE

$$u_t + f(u)_x = 0. \tag{2.32}$$

The general entropy condition can be derived by forcing solutions of (2.32) to be the limit of solutions to the viscous equation

$$u_t^\epsilon + f(u^\epsilon)_x = \epsilon u_{xx}^\epsilon \tag{2.33}$$

as $\epsilon \to 0$. Let H, G be differentiable functions, with H convex, satisfying

$$H(u)_t + G(u)_x = 0 \tag{2.34}$$

for smooth solutions u of (2.32). We call such a pair of functions an *entropy pair*.

Assume now that u is a weak solution to (2.32). We say that u satisfies the Kružkov entropy condition if

$$H(u)_t + G(u)_x \leq 0 \tag{2.35}$$

is satisfied weakly, that is,

$$\iint \Big(H(u)\varphi_t + G(u)\varphi_x \Big) \geq 0 \tag{2.36}$$

for all test functions φ, where the inequality is flipped due to a negative sign arising from integration by parts. In this case, we call u an *entropy solution.*

Recall that the reason we sought conditions such as Rankine-Hugoniot and the Lax entropy condition was that weak solutions to (2.32) were in general not unique, and often represented unphysical solutions. The Kružkov entropy condition solves this in the sense that for a piecewise twice differentiable flux f and L^1 initial data u_0 of bounded variation, there exists a unique entropy solution to (2.32) (theorem 2.14 of [15]).

In practice, it is often easier to apply Rankine-Hugoniot together with Lax entropy, which is valid for convex fluxes, than to apply Kružkov directly. For nonconvex fluxes, one can switch Lax entropy for the *Oleĭnik entropy condition*

$$\frac{f(k) - f(u_r)}{k - u_r} < \frac{\mathrm{d}\gamma}{\mathrm{d}t} < \frac{f(k) - f(u_l)}{k - u_l},$$

which is equivalent to Kružkov in scalar problems with isolated discontinuities ([15], p. 59). It is important to note that although the existence and uniqueness result for entropy solutions of (2.32) is satisfactory in the scalar case, the situation becomes considerably more difficult in multidimensional systems.

Chapter 3

Numerical schemes

One of the biggest challenges with building numerical schemes for hyperbolic equations is the existence of shocks. The finite difference approximation to the derivative

$$\frac{u(t, x + \Delta x) - u(t, x)}{\Delta x}$$

blows up in the presence of a shock as $\Delta x \to 0$, indicating that we should not expect finite difference methods to provide good approximations near discontinuities. We instead typically employ *finite volume methods* (FVM), a general class of schemes designed to handle discontinuities. An advantage of FVMs is that they are *conservative*, meaning in particular that they produce solutions with the correct shock position.

The purpose of this chapter is not to provide a thorough explanation of modern FVMs, but rather to introduce a few simple methods that can approximate hyperbolic equations to a reasonable accuracy. Nevertheless, we will later see that the methods presented here are powerful enough to allow numerics-informed neural networks, the main topic of this thesis, to solve inverse problems with hyperbolic equations to a very high accuracy. We will first explore the rationale behind FVMs and present two first order schemes, the Godunov and Lax-Friedrich schemes. Then we present the second order Lax-Wendroff scheme along with a more general and modern Central Upwind scheme due to Kuragnov et. al. [24].

Finite volume methods essentially reduce the solution of a hyperbolic equation with general initial condition to the solution of a set of Riemann problems. Recall that a Riemann problem is a hyperbolic equation

$$u_t + f(u)_x = 0 \tag{3.1}$$

equipped with the initial condition

$$u_0(x) = \begin{cases} u_l & x \leq 0 \\ u_r & x > 0 \end{cases}. \tag{3.2}$$

In this chapter we will analyse the homogeneous case, as the addition of a source function is quite simple numerically. We will explain this further in Section 4.4. In Sections 2.3 and 2.4 we analyzed properties of the solution of Riemann problems, focusing especially on Burgers' equation. In particular we explored how solution values are conserved along characteristic lines, and how collision and separation of characteristics produce shocks and rarefaction waves in the solution.

The discussion in this chapter is based on [30]. In the finite volume framework, we imagine our spatial dimension is partitioned into a finite number of *volume elements*, or *cells*, C_j. We are interested in approximating the cell averages

$$U_j^n = \frac{1}{h} \int_{C_j} u(t_n, x) \tag{3.3}$$

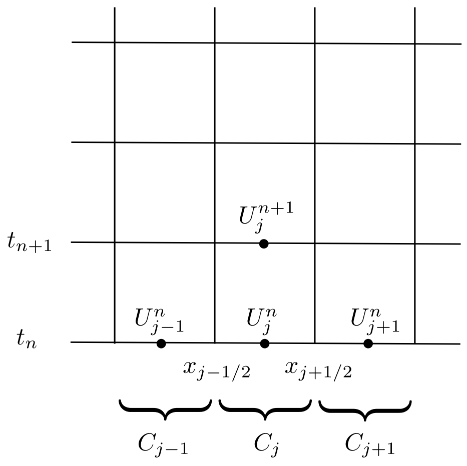


Figure 3.1: Finite volume grid in one spacial dimension.

for time points t_n in some grid, where h is the volume of C_j. To this end we would like to build an update rule for finding $\{U_j^{n+1}\}_j$ given the cell averages $\{U_j^n\}_j$ at the previous time point.

Assume the spatial domain is the interval $[a, b]$. Let $x_j = a + (j + 1/2)\Delta x$, $j = 0, \ldots, N$, denote the cell midpoints, where $\Delta x = \frac{b-a}{N+1}$. The corresponding set of volume elements is

$$C_j = [x_{j-1/2}, x_{j+1/2}).$$

The time discretization is defined by $t_n = n\Delta t$ for some time interval Δt. The grid is plotted in Figure 3.1.

In order to find our time stepping rule we integrate (3.1) over $C_j \times [t_n, t_{n+1}]$, yielding

$$\int_{C_j} \int_{t_n}^{t_{n+1}} u_t + \int_{C_j} \int_{t_n}^{t_{n+1}} f(u)_x = 0 \tag{3.4}$$

$$\int_{C_j} u(t^{n+1}, x) - \int_{C_j} u(t^n, x) = -\int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) + \int_{t_n}^{t_{n+1}} f(u(t, x_{j-1/2})). \tag{3.5}$$

Dividing both sides by Δx and applying the approximation (3.3) we obtain

$$U_j^{n+1} = U_j^n - \frac{\Delta t}{\Delta x}(F_{j+1/2}^n - F_{j-1/2}^n), \tag{3.6}$$

where

$$F_{j+1/2}^n = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})). \tag{3.7}$$

Equation (3.6) can be interpreted as expressing that the change in cell average from one time point to the next is equal to the net flux into the cell during the time interval. Different finite volume methods are constructed by applying different approximations to $F_{j+1/2}^n$, and these approximations are often called *numerical fluxes.*

Sections 3.1 to 3.3 present methods designed for scalar equations, while Section 3.4 considers a method built specifically for hyperbolic systems. The scalar methods can also be extended to handle systems. For more information on how this extension can be done, [30, 27, 40, 15] provide comprehensive discussions. In particular, we will need a system scheme for the shallow water system in Chapter 8.

3.1 Godunov

One of the simplest numerical fluxes is that of Godunov [10]. Godunov's scheme is based on the observation that cell averages are constant in each cell. Cells can have different averages, so there may be discontinuities at cell interfaces $x_{j+1/2}$. Thus, at each timestep t_n, we have a Riemann problem

$$\begin{cases} u_t + f(u)_x = 0 \\ u(t^n, x) = \begin{cases} U_j^n, & x \le x_{j+1/2} \\ U_{j+1}^n, & x > x_{j+1/2} \end{cases} \end{cases} \tag{3.8}$$

for every cell interface.

Recall from Section 2.4 that the solution to the Riemann problem of Burgers' equation was a function of the single variable $\xi = x/t$ both for $u_l > u_r$ and for $u_l < u_r$. This is not special for Burgers' equation, but happens for every homogenous hyperbolic equation as long as each component of the flux function is twice continuously differentiable. This statement is a corollary of Lax' theorem, which is presented with proof as theorem 5.17 of [15]. Therefore we know that the solution to the Riemann problem across the interface $x_{j+1/2}$ can be written as a function $\bar{U}_j(\bar{\xi})$ with $\bar{\xi} = \frac{x - x_{j+1/2}}{t - t_n}$. In particular, this means that the solution is constant at the cell interface $\bar{\xi} = 0$.

Now, $\bar{U}_j$ is either continuous or discontinuous at the interface. Let $\bar{U}_j(0^+)$ denote the limit of $\bar{U}_j$ when $\bar{\xi} \to 0$ from above, and let $\bar{U}_j(0^-)$ denote the limit when $\bar{\xi} \to 0$ from below. If $\bar{U}_j$ is continuous, then $f(\bar{U}_j(0^-)) = f(\bar{U}_j(0^+))$ trivially. If it is discontinuous we have a stationary shock at $\bar{\xi} = 0$. Rankine-Hugoniot gives

$$f(\bar{U}_j(0^+)) - f(\bar{U}_j(0^-) = 0 \cdot (\bar{U}_j(0^+) - \bar{U}_j(0^-)) = 0,$$

so the fluxes at either side of the interface are still equal. Thus, at the interface, $\bar{U}_j$ is constant and the flux is continuous. The numerical flux reduces to

$$F_{j+1/2}^n = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(U(t, x_{j+1/2})) = f(\bar{U}_j(0)).$$

Godunov provides an explicit formula for the numerical flux by solving the Riemann problem exactly. However, this formula involves an optimization procedure for each cell interface at each timestep, and is time consuming for complicated flux functions in high refinement. However, for flux functions with a minimum at U^* and no other extremal points, the formula can be written

$$F_{j+1/2}^n = \max \Big\{ f\big(\max\{U_j^n, U^*\}\big), f\big(\min\{U_{j+1}^n, U^*\}\big) \Big\}. \tag{3.9}$$

This formula can be used for instance with Burgers' equation, where $U^* = 0$.

In order for the previous analysis to work, it is important that the solutions to the Riemann problems at different cell interfaces do not interact. In other words, we must not allow enough time to pass between steps that wavefronts from adjacent Riemann problems can collide. The maximum wavespeed in our problem setup will be

$$\max_j |f'(U_j^n)|,$$

so if we require

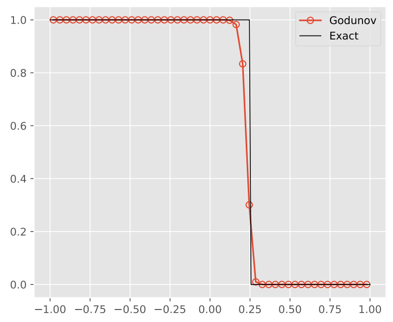
$$\max_j |f'(U_j^n)| \frac{\Delta t}{\Delta x} < 1 \tag{3.10}$$

we can guarantee that the Riemann solutions will not interact before the next timestep. We call (3.10) the *CFL condition.*

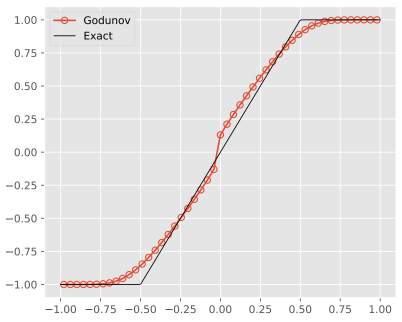
To illustrate Godunovs performance, we apply it to Burgers' equation for two different Riemann initial conditions. The first case uses $u_l = 1$, $u_r = 0$ to illustrate shock solutions, and the second uses $u_l = -1$, $u_r = 1$ to illustrate rarefaction waves. The results are plotted in Figure 3.2.

3.2 Lax-Friedrich

Godunov's method works well with Burgers' equation because we have an explicit expression for the solution of the Riemann problem (3.8) in this case. However, such an expression may not be



(a) $u_l = 1$, $u_r = 0$



(b) $u_l = -1$, $u_r = 1$

Figure 3.2: Godunov method on Burgers' equation with Riemann conditions, $M = 50$.

available for more general hyperbolic systems. In addition, if the flux function does not possess a single optimum we have to compute an optimization step at each cell interface at each timepoint, which can be very costly when the mesh is highly refined.

In order to mitigate some of these issues, we try to approximate the solution to the Riemann problem instead of calculating it explicitly. We model the solution as given by two waves traveling to the left and right of the interface with speeds $s^l_{j+1/2}$ and $s^r_{j+1/2}$. The approximate solution becomes

$$u(t,x) = \begin{cases} U^n_j, & x \le s^l_{j+1/2}t \\ U^*_{j+1/2}, & s^l_{j+1/2}t < x \le s^r_{j+1/2}t \\ U^n_{j+1}, & x < s^r_{j+1/2}t \end{cases}, \tag{3.11}$$

where $U^*_{j+1/2}$ can be found by applying the Rankine-Hugoniot conditions. This gives

$$\begin{aligned} f(U^n_{j+1}) - F^n_{j+1/2} &= s^r_{j+1/2}(U^n_{j+1} - U^*_{j+1/2}) \\ F^n_{j+1/2} - f(U^n_j) &= s^l_{j+1/2}(U^*_{j+1/2} - U^n_j), \end{aligned}$$

where $F^n_{j+1/2}$ is the intermediate flux value. If we consider the wave speeds known, we can solve this system of equations to get

$$F^n_{j+1/2} = \frac{s^r_{j+1/2} f(U^n_j) - s^l_{j+1/2} f(U^n_{j+1}) + s^r_{j+1/2} s^l_{j+1/2}(U^n_{j+1} - U^n_j)}{s^r_{j+1/2} - s^l_{j+1/2}}. \tag{3.12}$$

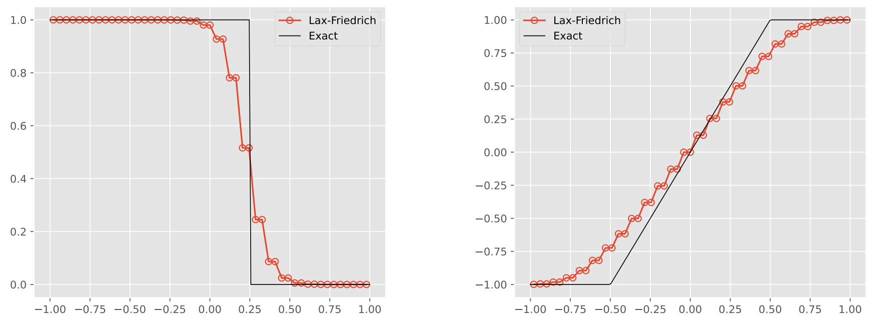
Different schemes can be built by choosing different values for $s^l_{j+1/2}$ and $s^r_{j+1/2}$. The Lax-Friedrich scheme chooses these values as the maximum allowed speeds still ensuring that neighboring Riemann problems do not interact in a single timestep. That is,

$$s^l_{j+1/2} = -\frac{\Delta t}{\Delta x}, \qquad s^r_{j+1/2} = \frac{\Delta t}{\Delta x}$$

Using these values in (3.12) yields the Lax-Friedrich flux

$$F^n_{j+1/2} = \frac{f(U^n_{j+1}) + f(U^n_j)}{2} - \frac{\Delta x}{2\Delta t}(U^n_{j+1} - U^n_j). \tag{3.13}$$

We apply the Lax-Friedrich method to the same test problems as Godunov. The results are plotted in Figure 3.3. Notice that Lax-Friedrich is more diffusive than Godunov. This reduction in performance is balanced by the fact that we do not need to exactly solve the Riemann problems in each step, making it less computationally demanding for more complicated fluxes.



(a) $u_l = 1,\ u_r = 0$ **(b)** $u_l = -1,\ u_r = 1$

Figure 3.3: Lax-Friedrich method on Burgers' equation with Riemann conditions, $M = 50$.

3.3 Lax-Wendroff

In order to understand why Lax-Friedrich is so diffusive, notice that its numerical flux satisfies

$$\begin{aligned}\frac{\Delta t}{\Delta x}\left[F^n_{j+1/2} - F^n_{j-1/2}\right] &= \frac{\Delta t}{\Delta x}\left[\frac{f(U^n_{j+1}) - f(U^n_{j-1})}{2} - \frac{\Delta x}{2\Delta t}(U^n_{j+1} + U^n_{j-1})\right] \\ &= \frac{\Delta t}{\Delta x}\left[\bar{F}^n_{j+1/2} - \bar{F}^n_{j-1/2}\right] - \frac{U^n_{j+1} - 2U^n_j + U^n_{j-1}}{2} \qquad (3.14)\end{aligned}$$

where

$$\bar{F}^n_{j+1/2} = \frac{f(U^n_{j+1}) + f(U^n_j)}{2}.$$

Thus the Lax-Friedrich scheme is equivalent to adding an artificial diffusion term $(\Delta x^2/\Delta t)u_{xx}$, and then approximating the averaged flux integral (3.7) using forward Euler

$$\frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) \approx \bar{F}^n_{j+1/2}, \qquad (3.15)$$

meaning the flux at the interface is approximated as the average of the left and right fluxes at $t = t_n$. As $\Delta t/\Delta x$ is determined by the CFL condition this diffusion term will disappear in the limit $\Delta x \to 0$, but it is significant for lower refinements.

Following the approach of [13], the Lax-Wendroff scheme can be derived by approximating the flux integrals (3.7) using the midpoint rule

$$\frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) \approx f^{n+1/2}_{j+1/2}.$$

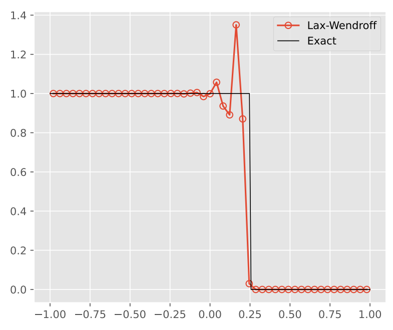
In order to approximate the midpoint flux value $f^{n+1/2}_{j+1/2}$, we use the Lax-Friedrich method on a grid with $\widetilde{\Delta x} = \frac{1}{2}\Delta x$ over a timestep $\widetilde{\Delta t} = \frac{1}{2}\Delta t$. Then

$$\begin{aligned}U^{n+1/2}_{j+1/2} &= U^n_{j+1/2} - \frac{\widetilde{\Delta t}}{2\widetilde{\Delta x}}\left[\bar{F}^n_{j+1} - \bar{F}^n_j\right] + \frac{1}{2}\left[U^n_{j+1} - 2U_{j+1/2} + U^n_j\right] \\ U^{n+1/2}_{j+1/2} &= \frac{1}{2}(U^n_{j+1} + U^n_j) - \frac{\Delta t}{2\Delta x}\left[\bar{F}^n_{j+1} - \bar{F}^n_j\right], \qquad (3.16)\end{aligned}$$

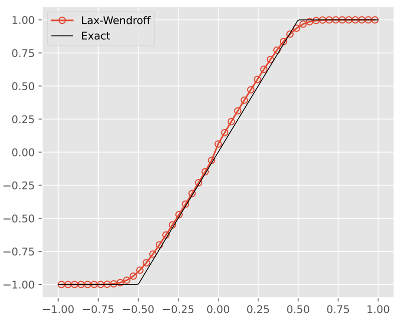
and the numerical flux becomes

$$F^n_{j+1/2} = f\left(U^{n+1/2}_{j+1/2}\right). \qquad (3.17)$$

The Lax-Wendroff scheme is formally second order [27], but it is known to produce oscillations near shocks. This can be seen in Figure 3.4, where we have applied Lax-Wendroff to Burgers' equation



(a) $u_l = 1$, $u_r = 0$



(b) $u_l = -1$, $u_r = 1$

Figure 3.4: Lax-Wendroff method on Burgers' equation with Riemann conditions, $M = 50$.

with the same Riemann initial conditions as in Sections 3.1 and 3.2. There are ways of mediating this, for instance with the use of *flux limiters*, but we will reserve the use of Lax-Wendroff to linear equations with smooth initial conditions where we can achieve second order convergence. For information about the oscillation issue and flux limiters the reader is referred to chapter 15 and 16 in [27].

3.4 Central Upwind

As previously mentioned, the mathematical theory underlying systems of hyperbolic equations, as well as corresponding finite volume extensions, are not within the scope of this work. However, a system solver is necessary in order to address the shallow-water equations discussed in Chapter 8. Consequently, we present a numerical scheme known as the central upwind (CUW) method, developed by Kurganov et al. [24], without going into too much detail.

The CUW method offers several advantages. It is computationally efficient, straightforward to implement, and avoids the substantial numerical diffusion associated with methods like Lax-Friedrich. Notably, its successful application to the shallow-water equations can be found in prior studies such as [13].

The general form of a 1D hyperbolic system is

$$Q_t + f(Q)_x = q, \tag{3.18}$$

with solution $Q : \mathbb{R}_+ \times \mathbb{R} \to \mathbb{R}^m$, flux $f : \mathbb{R}^m \to \mathbb{R}^m$, and source $q : \mathbb{R}_+ \times \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. Let J_f denote the Jacobian of f, and let $\lambda_1(Q)$, $\lambda_m(Q)$ denote respectively the smallest and largest eigenvalue of J_f. Suppose we are looking at a cell interface where the values of the solution Q at the left and right of the interface are respectively Q_L and Q_R. Define

$$\begin{aligned} a^+ &= \max_{Q \in \{Q_L, Q_R\}} \Big(\lambda_m(Q), 0\Big) \\ a^- &= \min_{Q \in \{Q_L, Q_R\}} \Big(\lambda_1(Q), 0\Big). \end{aligned}$$

The numerical flux of the CUW method is then given by

$$F^{\text{CUW}} = \frac{a^+ f(Q_L) - a^- f(Q_R)}{a^+ - a^-} + \frac{a^+ a^-}{a^+ - a^-}\Big(Q_R - Q_L\Big). \tag{3.19}$$

As we will see in Chapter 8, the shallow-water flux has a closed form expression for the eigenvalues of its Jacobian. This means the constants a^+, a^- can be computed very fast, making the CUW scheme inexpensive.

Chapter 4

Neural networks

In essence, neural networks are highly parameterized, smooth almost everywhere mappings from $\mathbb{R}^n$ to $\mathbb{R}^m$. They consist of a certain number of affine maps called *layers*, that are composed with nonlinear functions called *activations*. Each layer consist of a number of parameters that need to be determined in order to optimize performance in some given task. The determination of these parameters is called *training*. The history of neural networks traces back to the late 1950s with Rosenblatt's groundbreaking work on the *Perceptron* [35]. However, it wasn't until the introduction of the *backpropagation* algorithm in 1986 [37] for efficient parameter optimization that neural networks gained significant academic interest. This advancement enabled the training of large networks with remarkable efficiency.

Since then, neural networks have emerged as the cornerstone of some of the most powerful computational models ever devised. Notably, *convolutional* networks have revolutionized image processing [12], while *recurrent* networks have excelled in analyzing time series data [38]. More recently, *transformer* models built on the attention mechanism [41] have made significant strides in video and text processing [28].

In this work we will use neural networks as approximators to source terms in hyperbolic PDEs, which will require some customizations to the standard training procedure usually applied to networks. However, we start with an explanation of the structure of the most basic neural networks used in practice today, the so called *multilayer perceptron* models (MLP), and their usual training procedure.

4.1 Multilayer perceptrons

4.1.1 Basic structure

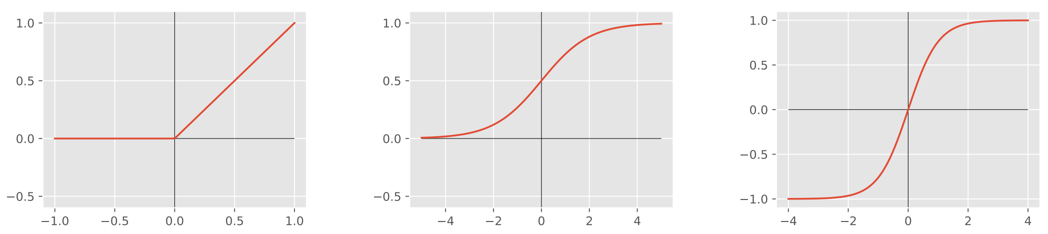
An MLP is a function $\Phi_\theta^H : \mathbb{R}^n \to \mathbb{R}^m$ that can be written as a composition

$$\Phi_\theta^H(x) = A_L \circ \varphi \circ A_{L-1} \circ \varphi \circ \cdots \circ A_1(x) \tag{4.1}$$

of affine maps $A_\ell : \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_{\ell+1}}$, $\ell = 1, \ldots, L$, and nonlinear activation functions φ acting componentwise on their input. That is, the activation $\varphi : \mathbb{R} \to \mathbb{R}$ is extended to $\mathbb{R}^d$ using the convention $\varphi([x_1, \ldots, x_d]^{\mathrm{T}}) = [\varphi(x_1), \ldots, \varphi(x_d)]^{\mathrm{T}}$. The affine maps can be defined by

$$A_\ell = W_\ell x + \beta_\ell, \tag{4.2}$$

where $W_\ell \in \mathbb{R}^{d_{\ell+1} \times d_\ell}$ is called the *weight* and $\beta_\ell \in \mathbb{R}^{d_{\ell+1}}$ is called the *bias*. Here $d_{L+1} = m$ and $d_1 = n$. The affine maps A_ℓ are called *layers*, with *input dimension* d_ℓ and *output dimension* or *node count* $d_{\ell+1}$. Notice that the output dimension of layer ℓ is the input dimension of layer $\ell + 1$. The specific elements of each layer's weight and bias are the *trainable* parameters of the network, in other words the parameters that are optimized during the training procedure. We denote by



(a) ReLU **(b)** Sigmoid **(c)** Tanh

Figure 4.1: Different types of activation functions.

$\theta = \{W_\ell, \beta_\ell\}_\ell$ the set of all trainable parameters. We call structure defining parameters such as L and $\{d_\ell\}_\ell$ *hyperparameters*, and they need to be chosen explicitly by the user. We denote by $H = \{L, \{d_\ell\}_\ell, \varphi\}$ the set of hyperparameters of the network.

Note that the specific choice of activation function φ is also a hyperparameter. The most common choice today is the *rectified linear unit* (ReLU) function [39], defined as

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

Recall that this function will be applied componentwise to its $\mathbb{R}^d$ input. Other choices are available, such as for instance the *sigmoid*

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.3}$$

or the hyperbolic tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{4.4}$$

Common for these these functions is that they can be evaluated and differentiated quickly on a computer, and that they are nonlinear. Their shapes are plotted in Figure 4.1.

Neural networks are useful in problems that can be cast as function approximation tasks. For instance, image classification tasks can be cast as approximating the function f mapping images in some image space to the probability distribution representing their correspondence to a set of labels. Usually we train networks by applying them to datasets where we know the correct output of f, measure the error made by the network, and apply some gradient based nonlinear optimization routine to determine the trainable parameters θ. Such datasets are called *training sets*, and we measure the network error using a function known as the *loss* or *objective*. For instance, if we are trying to approximate a function $f : \mathbb{R}^2 \to \mathbb{R}$, the training set $\mathcal{S} \subset \mathbb{R}^2 \times \mathbb{R}$ will consist of a discrete set of points $\{x_i\}_{i\in\mathcal{I}} \subset \mathbb{R}^2$ and their corresponding images under f, $\{y_i = f(x_i)\}_{i\in\mathcal{I}} \subset \mathbb{R}$, where $\mathcal{I}$ is some index set. Then $\mathcal{S} = \{(x_i, y_i)\}_{i\in\mathcal{I}}$. The loss function is usually based on the mean squared error,

$$\frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} (\Phi_\theta^H(x) - y)^2. \tag{4.5}$$

4.1.2 Universal approximation

MLPs demonstrate exceptional performance in practical function approximation tasks. This is not a mere coincidence, as MLPs possess the interesting property of being ”universal approximators”. Under certain conditions, this theoretical property ensures their ability to approximate arbitrary measurable functions with arbitrary precision.

Intuitively, the universal approximation property guarantees that within a compact domain, any measurable function can be accurately approximated by an MLP. This powerful notion was formalized by a series of articles by among others Hornik et. al. in 1980s-90s, where they established several variants of the universal approximation theorem (UAT) for MLPs [9, 17, 18, 16, 33].

The establishment of the universal approximation property for MLPs, a significant theoretical advancement, solidifies their role as versatile function approximators, providing a theoretical backing to their wide-ranging practical success.

Universal approximation is a property related to the density of the class $\Sigma^\varphi(\mathbb{R}^n, \mathbb{R}^m)$ of neural networks from $\mathbb{R}^n$ to $\mathbb{R}^m$ with activation function φ. Thus the reason for the many variants of the UAT: they are based on different notions of density, different network architectures, and in particular different classes of activation functions. In 1993, Leshno et. al. gives a general result providing necessary and sufficient conditions on φ for $\Sigma^\varphi(\mathbb{R}^n, \mathbb{R}^m)$ to have universal approximation in L^p on compact subsets of $\mathbb{R}^n$ [26]. We state a modified version here.

Theorem 1 (Universal approximation theorem I)**.** *Let μ denote the Lebesgue measure, $\Sigma^\varphi(\mathbb{R}^n, \mathbb{R}^m)$ denote the class of neural networks from $\mathbb{R}^n$ to $\mathbb{R}^m$ with activation φ and $C \subset \mathbb{R}^n$ be compact. Then $\Sigma^\varphi(\mathbb{R}^n, \mathbb{R}^m)$ is dense in $L^p_\mu(C, \mathbb{R}^m)$ if and only if φ is not a polynomial.*

Theorem 1 is both powerful and general, and as stated by Hornik et. al. when discussing their own version of the result, "it implies that any lack of success in applications must arise from inadequate training, insufficient number of hidden units or the lack of a deterministic relationship between input and target" ([17], p. 363). However, it has its limitations. Specifically, in Chapter 8 we will use a neural network Φ^H_θ to approximate a function B while *simultaneously* using $\partial_x \Phi^H_\theta$ to approximate B_x. Theorem 1 does not guarantee arbitrary approximation to both a function and its derivatives. Luckily, this case was covered in a paper by Hornik et. al. in 1990 [18], guaranteeing universal approximation to a function and its derivatives up to some order. It even applies to weak derivatives, making it very useful in the context of hyperbolic equations. A modified version is presented here.

Theorem 2 (Universal approximation theorem II)**.** *Let $U \subset \mathbb{R}^n$ be open and bounded, and define the class $W^m_p(U, \mu)$ of functions having (weak) derivatives up to order m belonging to $L^p_\mu(U)$. Suppose there exists a point $x \in U$ such that any ray originating in x has a unique intersection with ∂U. Then $\Sigma^\varphi(\mathbb{R}^n, \mathbb{R}^m)$ is dense in $W^m_p(U, \mu)$ whenever φ is given by (4.3) or (4.4).*

The version of Theorem 2 stated in [18] is more general than this, in particular allowing any *l-finite* activation function, but the version stated here will suffice for our purposes. The property that U contains a point x such that any ray originating in x has a unique intersection with the boundary ∂U, known as being *starshaped with respect to a point*, is always satisfied for the rectangular domains $[0, T] \times [x_l, x_r]$ considered in our experiments.

4.1.3 Training

Assume now we want to train a network Φ^H_θ to approximate a function $f : \mathbb{R}^n \to \mathbb{R}^m$, and that we have access to a training set $\mathcal{S} \subset \mathbb{R}^n \times \mathbb{R}^m$. Define the loss

$$\mathcal{L}^H_\theta(x, y) = \left\| \Phi^H_\theta(x) - y \right\|^2. \tag{4.6}$$

The goal is to find the trainable parameters θ^* that optimizes the performance of the network on the training set under this loss function,

$$\theta^* = \arg\min_\theta \sum_{(x,y)\in\mathcal{S}} \mathcal{L}^H_\theta(x, y). \tag{4.7}$$

Notice that since Φ^H_θ is a composition of smooth a.e. functions it is differentiable a.e., in particular with respect to its weights and biases. Since $\mathcal{L}^H_\theta$ is differentiable with respect to Φ^H_θ, we can use standard *gradient descent* to optimize θ. That is, for every point $(x, y) \in \mathcal{S}$ we evaluate $\mathcal{L}^H_\theta(x, y)$ and change the weights of the network according to the step

$$\theta \mapsto \theta - \eta \nabla_\theta \sum_{(x,y)\in\mathcal{S}} \mathcal{L}^H_\theta(x, y), \tag{4.8}$$

where $\eta > 0$ is known as the *learning rate.* It is important to note that the problem of nonconvex optimization is NP-hard in the worst case [31], and that we cannot guarantee convergence in the general problem. Even though it can be shown that this method will converge under certain regularity assumptions [25, 1], in practice we can only rely on empirical evidence for its convergence. In the case of the vanilla gradient descent algorithm in (4.8), empirical evidence suggest it to be very slow in practice [36].

Today, it is common to employ some variant of *batched stochastic gradient descent* (SGD). These methods are based on randomly drawing some subset $\hat{\mathcal{S}} \subset \mathcal{S}$ of the training set and evaluating

$$\sum_{(x,y)\in\bar{\mathcal{S}}} \nabla_\theta \mathcal{L}_\theta^H(x,y), \tag{4.9}$$

using this in place of the full-dataset gradient in the parameter update of (4.8). In the next iteration, a different subset is drawn. This way we can perform multiple parameter updates for every pass through the entire training set, or *epoch*, while still providing enough data each update to optimize against the more global statistical properties of the dataset. In other words, assuming $\bar{\mathcal{S}}$ is large enough that its distribution approximates that of $\mathcal{S}$ (with properties such as the mean approximately equal), each parameter update using (4.9) will be a step in the approximate direction that minimizes the expected loss on the entire dataset

$$E_{p(x,y)}\left[\mathcal{L}_\theta^H(x,y)\right],$$

over θ. One of the most common variants of SGD in industry today is the ADAM optimizer [22]. its name is derived from *adaptive moment estimation*, and it uses estimates of the first and second moments of the gradient to adapt the learning rate between iterations. We will use ADAM in the numerical experiments presented in the following chapters.

To practically evaluate the derivatives, notice that as the network is a composition of smooth functions, its gradient will be the product of the derivatives of the components. This fact is used in the backpropagation algorithm [37] to evaluate the gradient of the network with respect to its trainable parameters given a point $x_i, i \in \mathcal{I}$. In essence the algorithm computes $\mathcal{L}_\theta^H(x_i, y_i)$ while storing the latent outputs of all layers in the network, then evaluates each layer's gradient function on the previous layer's output before multiplying them all together, finally multiplying with the gradient of $\mathcal{L}_\theta^H$ with respect to the network outputs. The reader is referred to [37] for details.

Backpropagation has allowed researchers to develop programming interfaces for training general neural network architectures, some notable examples being Google's TensorFlow [29] and PyTorch [32] of the PyTorch Foundation. Both libraries are open source, and widely used in industry. Technically, TensorFlow and PyTorch provide an API to perform general *automatic differentiation*, of which backpropagation is an example. Details on different techniques for automatic differentiation, along with practical examples, can be found in section 6.5 of [11].

In this work, we will mainly use PyTorch v.2.0.0 with Python v.3.10.10 as programming language. There are several reasons for this. Python was chosen for its fast prototyping speed, as well as its large ecosystem of packages for numerical simulation and visualization. There are, for example, preexisting packages for simulating hyperbolic equations, most of them based on fast C++ backends available through for instance the NumPy interface [14]. This motivates the choice of PyTorch as the automatic differentiation backend, as the PyTorch Tensor constructs share most of their API with NumPy NDArrays. Thus, a lot of preexisting simulation code can be applied practically without modification to PyTorch Tensors, which support automatic differentiation. PyTorch also has a C++ API, providing access to the vast amount of preexisting C++ simulation code as well. Nevertheless, a custom numerical simulation library in Python was developed in conjunction with this thesis, specifically designed to natively support PyTorch Tensors. The code is uploaded to the `ninn` GitHub repo (https://github.com/benwaad/ninn) [2].

4.2 Usage in PDEs

Assume we have a PDE

$$u_t + \mathcal{D}(u; \lambda) = 0, \ x \in \Omega, \ t \in [0, T] \tag{4.10}$$

for some function u and differential operator $\mathcal{D}(u; \lambda)$ acting on u, parameterized by λ. There are several ways neural networks can be useful in the analysis of (4.10). Notably, several variants of *physics-informed neural networks* (PINN) have emerged in literature after their introduction in 2017 by Raissi et. al. [5, 34]. PINNs are neural networks that are trained to satisfy (4.10), along with initial and boundary conditions, by encorporating a residual term in the loss function and minimizing at randomly chosen collocation points in $[0, T] \times \Omega$. Specifically, we define the residual

$$\mathcal{F}(t, x) = \partial_t \Phi_\theta^H(t, x) + \mathcal{D}(\Phi_\theta^H; \lambda)(t, x),$$

where the derivatives can be computed using automatic differentiation, and incorporate the term

$$\text{MSE}_\theta^H(\mathcal{F}) = \frac{1}{N} \sum_{i=1}^{N} |\mathcal{F}(t_i, x_i)|^2, \tag{4.11}$$

where $\{(t_i, x_i)\}_{i=1}^N \subset [0, T] \times \Omega$ is the set of collocation points, into the loss function together with terms penalizing deviation from the initial and boundary conditions. Hence the name physics-informed: the network incorporates the physics of the problem, described by (4.10), into its training procedure.

PINNs can be used to obtain a neural network representation of a solution of (4.10) in an unsupervised and mesh free fashion, and with the inclusion of known values of u at the collocation points we can simultaneously learn the parameters λ in simple cases [34]. Recall that the problem of learning the model parameters λ was called inverse problems, as opposed to the forward problem of learning the solution u. This technique can be adapted to hyperbolic operators $\mathcal{D}$ arising from conservation laws by separating the computational domain into finite volume cells and representing the solution in each cell by a separate neural network, enforcing continuity of flux at the cell interfaces with an additional MSE term in the loss function. Jagtap et. al. introduce this method, known as *cPINN*, in [20] and show that it performs better than regular PINNs on conservation laws.

PINNs allow the inclusion of general system constraints as terms in the loss function, making them a very flexible class of methods for general PDE problems. For instance, [3] shows that they can be employed with success in complex scenarios in fluid dynamics for both forward and inverse problems. As they are generally mesh free, PINNs are also well suited to tackle problems in high dimensional domains where traditional computational methods break down due to the *curse of dimensionality* [23].

However, most PINN experiments with inverse problems have focused on learning a small, discrete set of unknown parameters λ. For instance, [34] experiments with learning $\lambda = (\lambda_1, \lambda_2)$ in the viscid Burger's equation

$$u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0,$$

and [20] considers a phase field with two unknown parameters in a biomedical flow system. Their performance on inverse problems involving for instance general solution dependent source functions have not yet been properly investigated.

In addition, PINNs represent the unknown solution u by a neural network also in the inverse problem when we have access to measurements $\{U_i\}_i$, relying on forcing the network output at the collocation points $\{(t_i, x_i)\}_i$ to equal the measured values. Although this allows for general positioning of the collocation points, the introduction of a network representation of u introduces an error to the system that can be removed by relying on well tested numerical algorithms for PDEs. In particular, methods such as the ones described in Chapter 3 encodes the physics of the problem in discrete time stepping schemes based on a proper mathematical foundation, eliminating the need for a network representation of u in inverse problems. Thus, we can use the methods from Chapter 3 in combination of a network representation of all unknown parts of $\mathcal{D}(\cdot; \lambda)$ itself in order to solve inverse problems without the need for an explicit model of the solution. This will be the topic of the following section.

4.3 Numerics-informed neural networks

Assume a PDE is given by (4.10) for $\Omega \subset \mathbb{R}^n$, $u : [0,T] \times \Omega \to \mathbb{R}^m$, where

$$\mathcal{D}(u; q) = \nabla \cdot f(u) - q$$

is a hyperbolic operator containing an unknown source term $q : [0,T] \times \Omega \times \mathbb{R}^m \to \mathbb{R}^m$. Define a finite volume mesh on $[0,T] \times \Omega$ with centroids $\{x_i\}_{i=0}^M$ and timesteps $\{t_n\}_{n=0}^N$, with spatial and temporal grid spacing Δx and Δt respectively. Assume also that we have some numerical scheme G acting on stencils s_i^n of the grid, such that

$$U_i^{n+1} = G(s_i^n) + \Delta t q(t_n, x_i, U_i^n) \tag{4.12}$$

For example, in the Godunov method from Section 3.1, $n = m = 1$, $s_i^n = \{U_{i-1}^n, U_i^n, U_{i+1}^n\}$, and

$$G(s_i^n) = U_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2}^n - F_{i-1/2}^n)$$

where $F_{i\pm 1/2}^n$ is the Godunov numerical flux given by (3.9). In the specific case of hyperbolic operators, the solution can develop discontinuities that make it difficult to provide an error analysis of (4.12), which for smooth solutions or simpler differential operators is usually simple. However, in the scalar case there does exist convergence results even for very general problems. For instance, Fjordholm and Lye proves a convergence theorem for monotone finite volume schemes even when the initial data is of unbounded total variation(theorem 1 of [8]). The theory of convergence for numerical schemes is wasted in the standard PINN framework, as we attempt to directly minimize the PDE residual using nonlinear optimization techniques without robust convergence results. There are modifications that can be done to standard PINNs that allows for a proper error analysis, such as the wPINN algorithm [6], but the added complexity make them more diffult both to implement and to train.

Assume we have access to a set of measurements U_i^n, $i = 0, \dots, M$, $n = 0, \dots, N$. Define the index sets

$$\mathcal{X} = \{0, \dots, M\}, \quad \mathcal{T} = \{0, \dots, N-1\}, \quad \mathcal{J} = \mathcal{T} \times \mathcal{X}. \tag{4.13}$$

Our training set is

$$\mathcal{S} = \{(t_n, x_i, s_i^n, U_i^{n+1})\}_{(n,i)\in\mathcal{J}}$$

Recall that s_i^n consists of stencils of the measurements $\{U_i^n\}_{\mathcal{J}}$, meaning that $\mathcal{S}$ consists entirely of known values. Values that depend on the network parameters will be marked with a caret.

It is common to refer to an element $(t_n, x_i, s_i^n, U_i^{n+1}) \in \mathcal{S}$ as a *row* of the training set, as we in practice typically arrange the elements of $\mathcal{S}$ in a matrix where the indices i, n change along the column dimension. A *column* is then all the measurements of a single variable given in the dataset.

The idea is to represent the source function q with a neural network, and use the numerical scheme (4.12) to design a loss function. Let $\|\cdot\|$ denote the Euclidian 2-norm, and define a network $\hat{q}_\theta^H : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ with the loss function

$$\mathcal{L}_\theta^H(t_n, x_i, s_i^n, U_i^{n+1}) = \left\|U_i^{n+1} - G(s_i^n) - \Delta t \hat{q}_\theta^H(t_n, x_i, U_i^n)\right\|^2, \tag{4.14}$$

where we have assumed that every stencil s_i^n contains its central point U_i^n. As (4.14) is smooth as a function of the network parameters θ, we can proceed to train the network with standard SGD optimizers such as ADAM. Notice that this loss function is *scheme agnostic*, meaning it allows us to freely choose a numerical scheme on a case to case basis. This means we can easily adapt it to any PDE for which we can define a numerical scheme. It is also simple to extend it to different time stepping algorithms than the simple Euler method employed in (4.12) and the general conservative scheme (3.6). We will call $\hat{q}_\theta^H$, together with (4.14), a *numerics-informed neural network* (NINN). NINNs can be adapted to perform a forward problem by adding a network representation of u, but unlike PINNs they are specifically designed to perform inverse problems when the parameter space is an infinite-dimensional function space.

Note that NINNs are designed to train in an *offline* setting. This means that we first collect and organise the dataset, then train the network to approximate the source. The opposite is called *online* training, where we continuously receive new training set rows and use this incoming stream of observations to gradually train the network. While it is possible to construct an online training procedure for NINNs, this aspect has not been emphasized in this work.

4.4 A convergence result

We want the NINN loss function (4.14) to be related to L^2 approximation error of the network's source prediction. In fact, Eidnes and Lye proved that for systems of *ordinary differential equations* (ODE), the NINN loss as defined in (4.14) is proportional to the L^2 source approximation error with proportionality constant Δt, when using the simple forward Euler method as the numerical scheme (theorem 1 of [7]). We will return to ODEs and the forward Euler method in Chapter 5, but first we provide a similar result for the general NINN loss function.

Recall from Chapter 3 that finite volume schemes, at least in their simplest forms, represent the solution to a PDE by its average inside a collection of computational cells $\{C_i\}_i$ on a timegrid $\{t_n\}_n$. Recall that

$$U_i^n = \frac{1}{\Delta x}\int_{C_i} u(t_n, x),$$

that is the value of the i^{th} cell average at timepoint t_n. Consider the general hyperbolic scalar equation

$$u_t + f(u)_x = q, \tag{4.15}$$

with the source function $q = q(t, x, u)$. Following the same procedure as in eqs. (3.4) and (3.5), we arrive at the expression

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left[F_{i+1/2}^n - F_{i-1/2}^n\right] + \frac{1}{\Delta x}\int_{C_i}\int_{t_n}^{t_{n+1}} q, \tag{4.16}$$

where

$$F_{i+1/2}^n = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(u(t, x_{i+1/2})).$$

Assume we employ a numerical method G acting on stencils $\{s_i^n\}_i^n$, and that this scheme has a truncation error $\tau_G(\Delta t)$ associated with it. That is,

$$U_i^{n+1} = G(s_i^n) + \tau_G(\Delta t) + \frac{1}{\Delta x}\int_{C_i}\int_{t_n}^{t_{n+1}} q.$$

Notice that $\tau_G(\Delta t)$ is independent on Δx, as $\Delta x = c\Delta t$ by the CFL condition (3.10). Thus, the NINN loss function (4.14) can be written

$$\mathcal{L}_\theta^H(t_n, x_i, s_i^n, U_i^{n+1}) = \left\|\tau_G(\Delta t) + \left[\frac{1}{\Delta x}\int_{C_i}\int_{t_n}^{t_{n+1}} q\right] - \Delta t\hat{q}_\theta^H(t_n, x_i, U_i^n)\right\|^2. \tag{4.17}$$

Assume that the scheme has order p, that is $\tau_G(\Delta t) = \mathcal{O}(\Delta t^{p+1})$. We use the fact that $\hat{q}_\theta^H = q + (\hat{q}_\theta^H - q)$ to write

$$\mathcal{L}_\theta^H(t_n, x_i, s_i^n, U_i^{n+1}) = \left\|\left[\frac{1}{\Delta x}\int_{C_i}\int_{t_n}^{t_{n+1}} q\right] - \Delta t q(z_i^n) - \Delta t\left(\hat{q}_\theta^H(z_i^n) - q(z_i^n)\right) + \mathcal{O}(\Delta t^{p+1})\right\|^2, \tag{4.18}$$

where we have simplified by writing $z_i^n = (t_n, x_i, U_i^n)$. A simple Taylor expansion gives that

$$\int_{t_n}^{t_{n+1}} q\left(t, x_i, u(t, x_i)\right) = \Delta t q(z_i^n) + \mathcal{O}(\Delta t^2),$$

meaning

$$\frac{1}{\Delta x}\int_{C_i}\int_{t_n}^{t_{n+1}} q = \int_{t_n}^{t_{n+1}} q\left(t, x_i, u(t, x_i)\right) + \mathcal{O}(\Delta x^2\Delta t) \tag{4.19}$$

$$= \Delta t q(z_i^n) + \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2\Delta t). \tag{4.20}$$

Thus,

$$\mathcal{L}_\theta^H(t_n, x_i, s_i^n, U_i^{n+1}) = \left\|\Delta t\left(\hat{q}_\theta^H(z_i^n) - q(z_i^n)\right) + \mathcal{O}(\Delta t^{\min(1,p)+1}) + \mathcal{O}(\Delta x^2 \Delta t)\right\|^2 \tag{4.21}$$

$$= \frac{1}{\Delta t^2}\left\|\left(\hat{q}_\theta^H(z_i^n) - q(z_i^n)\right) + \mathcal{O}(\Delta t^{\min(1,p)}) + \mathcal{O}(\Delta x^2)\right\|^2 \tag{4.22}$$

Ignoring second order terms and higher, we can write

$$\left\|\hat{q}_\theta^H(z_i^n) - q(z_i^n)\right\|^2 = \frac{\mathcal{L}_\theta^H(t_n, x_i, s_i^n, U_i^{n+1})}{\Delta t^2}, \tag{4.23}$$

meaning the loss function (4.14) provides a scaled estimate of the L^2 error of the network approximation to q in the point $(t_n, x_i, u(t_n, x_i))$. The mean square error using $\mathcal{L}_\theta^H$ as kernel is then a scaled estimate of the global L^2 error of the network approximation.

Notice that the proportionality constant is now Δt^2, whereas the result in [7] simply had Δt. This is because the result in (4.23) is stated using squared norms. The reason for this is that the mean square error, which is applied batch-wise when training, is an approximation to the squared L^2 error, not the L^2 error itself. From a training perspective, applying the square root to the computed loss is an unnecessary operation, as it does not change the optimum. Thus we chose to work with squared norms.

Indeed, the analysis done in this section can be extended practically without modifications to apply for the general p-norm. In eiter case, if we take the p^{th} root of the loss function to obtain the actual p-norm, we end up with the familiar Δt proportionality arrived at by Eidnes and Lye, meaning eq. (4.23) extends theorem 1 in [7] to work with arbitrary schemes for PDEs.

In fact, (4.21) preserves the truncation errors made when the scheme is not exactly satisfied by the measurements, which extends the analysis in [7]. For instance, it tells us that not much is gained in terms of convergence from increasing the order of the scheme G. The reason for this is that the error in the integral approximation in eqs. (4.19) and (4.20) will dominate the truncation error of the scheme. If we used a numerical quadrature with higher order, we would gain more by increasing the accuracy of the scheme. We have not explored this in this thesis, but it represents a possible avenue of further research.

Section 6.3 of [7] provides a discussion as to why the result (4.23) is interesting and necessary. During training, we are generally trying to find $\hat{q}_\theta^H$ by minimizing the L^p norm of the difference between the measurements $\{U_i^n\}_i^n$ and the *solution predictions* $\{\hat{U}_i^n\}_i^n$, that is the cell averages generated with the numerical scheme using the neural network. Denote by u_θ the inclusion $\ell^1(\mathcal{J}) \hookrightarrow L^1(R_+ \times \mathbb{R})$ of $\{\hat{U}_i^n\}_i^n$ defined by $\hat{U}_i^n \mapsto \sum_{i,n} \hat{U}_i^n \mathbb{1}_{[t_n, t_{n+1}) \times C_i}$, where $\mathbb{1}$ is the indicator function. For $i \in \mathcal{X}$ let $u_i(t) = u(t, x_i)$ and s_i be a stencil centered in x_i, for instance $s_i = \{u_{i-1}, u_i, u_{i+1}\}$. The training process leads to a sequence $\{u_{\theta_k}\}$, where k denotes the training iteration, that converges to u in L^p. In the semidiscretized form, we can view (4.15) as an ODE

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = G(s_i) + q(t, x, u_i), \tag{4.24}$$

where G discretizes the spatial derivatives. We ignore truncation errors for the sake of argument, but they are easily handled using the triangle inequality. Then the function $u_{i,\theta}$ is defined by

$$\frac{\mathrm{d}u_{i,\theta}}{\mathrm{d}t} = G(s_i) + \hat{q}_\theta^H(t, x, u_i). \tag{4.25}$$

Notice that the scheme G takes a stencil of the true semidiscretized solution u_i, as this leads to timestepping from a stencil of measurements in the fully discretized case. Now, notice that

$$\hat{q}_\theta^H - q = \frac{\mathrm{d}u_{i,\theta}}{\mathrm{d}t} - \frac{\mathrm{d}u_i}{\mathrm{d}t},$$

so the approximation error is the same as the difference of the time derivatives of the true and predicted solution. Even though our training process provides $u_\theta \to u$ in L^p, we cannot automatically assume that this leads to $\frac{\mathrm{d}u_{i,\theta}}{\mathrm{d}t} \to \frac{\mathrm{d}u_i}{\mathrm{d}t}$ in L^p, since L^p convergence between two functions does not guarantee L^p convergence between their derivatives. Thus, in the limit $\Delta t \to 0$ when

$$\frac{u_i(t+\Delta t) - u_i(t)}{\Delta t} \to \frac{\mathrm{d}u_i}{\mathrm{d}t},$$

we cannot expect $\hat{q}_\theta^H \to q$. This is reflected in (4.23), as for smaller Δt we need to train to a lower training error to achieve the same source approximation accuracy.

4.5 Differentiable simulators

Note that (4.14) use the neural network to generate a prediction of the solution at the next timestep. It is vital that this solution prediction is differentiable, as we would otherwise not be able to train the network. In implementation, this requires procedures for simulating a PDE in a fully differentiable manner. We call such procedures *differentiable simulators*, and we present two different approaches.

The numerical schemes presented in Chapter 3 use previous timesteps to iteratively fill out the solution space. This allows us to construct a differentiable simulator in two different ways, namely *pointwise* and *sweeping* training. In short, the pointwise simulator works well in cases where the dataset consists of measurements of all quantities of interest at each gridpoint, while the sweeping simulator allows us to handle cases where only some components of the system are practically measurable, leading to so-called *incomplete* measurements.

Consider the hyperbolic PDE

$$Q_t + f(Q)_x = B, \quad x \in \Omega \subset \mathbb{R}, \quad 0 \le t \le T$$

where $Q : \mathbb{R}_+ \times \mathbb{R} \to \mathbb{R}^m$ and $B : \mathbb{R}_+ \times \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ is the source function. This setup mimics the shallow-water equations, where cases of incomplete measurements can arise in practice. We will discuss the shallow-water equations in Chapter 8. Let the index sets $\mathcal{X}, \mathcal{T}, \mathcal{J}$ be defined as in (4.13). Define a mesh $\{(t_n, x_i)\}_{(n,i)\in\mathcal{J}}$ on $[0,T] \times \Omega$ and imagine we have the dataset $\mathcal{S} = \{(t_n, x_i, s_i^n, Q_i^{n+1})\}_{\mathcal{J}}$ as described in Section 4.3 on this mesh. Recall that $\mathcal{S}$ consists entirely of known values. We wish to approximate B using a NINN $\hat{B}$ with a numerical scheme G.

Let $Q_i^n[k]$, $k \in \mathcal{K}$, denote the kth component of the solution Q at the point (t_n, x_i), where $\mathcal{K} = \{1, \dots, m\}$ is an index set. First, we assume that we have complete measurements of Q, meaning each measurement of Q consists of a measurement of $Q[k]$ for all $k \in \mathcal{K}$. We define the pointwise differentiable simulator by shuffling the dataset and picking out random elements of $\mathcal{S}$, before applying the loss $\mathcal{L}_\theta^H$ defined in eq. (4.14). This corresponds to performing a single network evaluation for each training point in $\mathcal{S}$, which in a batched setting with batch size d means we are performing d network evaluations for every weight update. The procedure is illustrated in Figure 4.2. In experiments, we will see that this constant relationship between network evaluations and weight updates makes the pointwise procedure converge in a fast and stable manner.

However, it is not always possible to design the training algorithm this way. Specifically, in cases of incomplete datasets, we do not have enough information in the stencils $\{s_i^n\}_{\mathcal{J}}$ to generate a step with the scheme G. We are forced to change our training procedure to account for this. Assume now that we have access to complete measurements of Q_i^0 for all i, that is, the initial condition is completely described. However, for $n > 0$ we only have access to a subset $Q_i^n[k]$, $k \in \tilde{\mathcal{K}} \subset \mathcal{K}$, of the components of Q.

If we want to train against a measurement $Q_i^{n+1}[k]$, $k \in \tilde{\mathcal{K}}$, we need to fill the missing data in all preceding timepoints with approximations provided by our NINN in a differentiable manner. That is, starting with our initial conditions on the entire spatial grid, we need to perform $n+1$ timesteps for all x_i on the grid in order to obtain a solution prediction that we can match against our data at t_{n+1}, and we need to perform this in a way that allows us to compute the network gradients at loss evaluation. This way, we generate the stencil itself using the network. In fact, as we need to generate a solution prediction in all gridpoints before t_{n+1}, we can treat the targets

$\{Q_i^{n+1}\}_{i\in\mathcal{X}}$ as a batch and evaluate loss on all of them to determine the weight update. We call this the sweeping differentiable simulator, see Figure 4.3.

Notice that as we only have access to measurements of components $k \in \tilde{\mathcal{K}}$, we can only evaluate loss for these components. The modified loss function becomes

$$\mathcal{L}_\theta^H(t_n, x_i, s_i^n, Q_i^{n+1}) = \sum_{k\in\tilde{\mathcal{K}}} \left| Q_i^{n+1}[k] - G(\hat{s}_i^n)[k] - \Delta t \hat{B}_\theta^H(t_n, x_i, \hat{Q}_i^n[k]) \right|^2, \tag{4.26}$$

where the stencil s_i^n and the previous solution value Q_i^n has been marked with a caret to emphasize that they depend upon previous network predictions, and therefore on the network parameters. Clearly the dependence of the loss function on the network parameters is a lot more complicated in sweeping compared to pointwise training, but as long as all operations performed on the network realizations are differentiable, a software package implementing automatic differentiation, such as PyTorch, will have no problem tracking these dependencies and computing the gradients when required.

It is important to notice that for large n, we need to perform many timesteps in order to perform a single parameter update. For longer time horizons T, this leads to much longer training times. However, all experiments conducted in this work ran in under a minute on a 2020 MacBook Air with an M1 chip and 8GB of RAM. Thus training times will largely be considered negligible in the following chapters, even though they can come to matter in an operational setting.

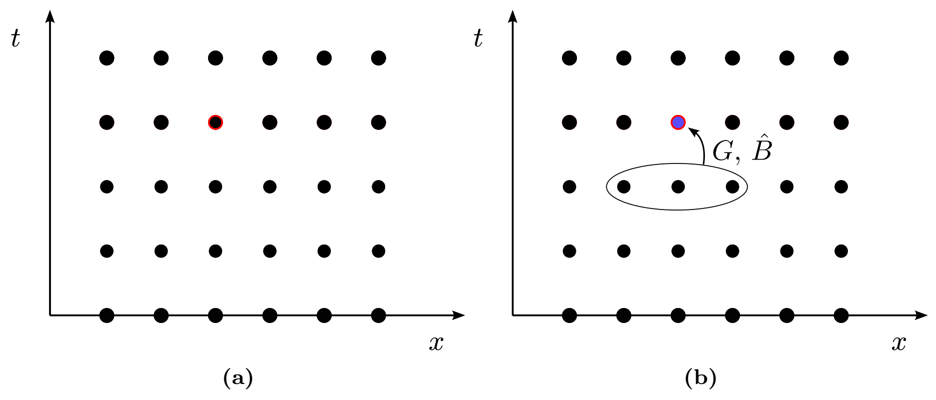


Figure 4.2: Pointwise training for a complete dataset. The filled in dots mark gridpoints where we have access to complete measurements of Q, and the red ring marks the point we wish to train against in this iteration. The right figure shows that we can use the stencil at the previous timepoint to obtain an approximation of the target point, denoted by blue filling, using the scheme G and the network $\hat{B}$.

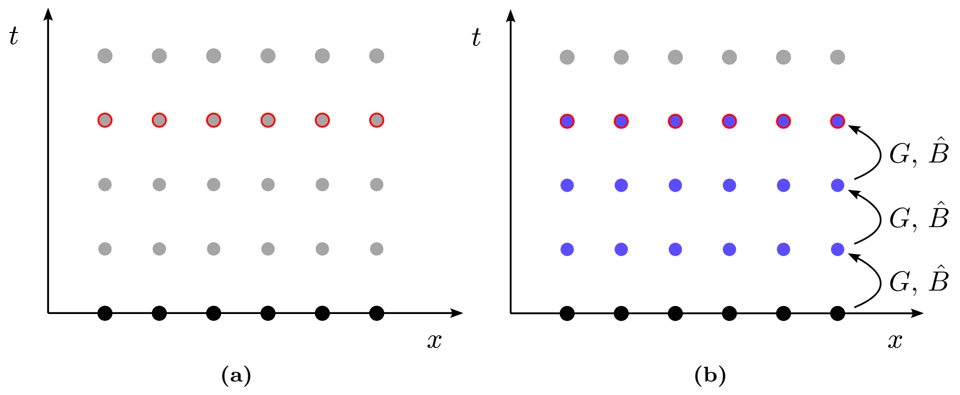


Figure 4.3: Sweeping training for an incomplete dataset. The black filled dots mark gridpoints where we have access to complete measurements of Q, and the gray dots mark points where we only have access to $Q[k]$ for k in some subset $\tilde{\mathcal{K}}$. The red ring marks the points we wish to train against in this iteration. The right figure shows that we must timestep the entire initial condition to obtain approximations to Q, denoted by blue filling, for all timepoints before our target.

Chapter 5

Learning source terms in scalar ODEs

As a motivating example, we show that a neural network can be used to approximate the source term in an ordinary differential equation (ODE) given measurements of the path of the solution. ODEs are special cases of PDEs where the unknown solution u does not depend on spatial derivatives of the solution. If we only apply a spatial discretization to the balance law (2.2), we obtain a semidiscretized system of ODEs, motivating an analysis of NINNs performance on scalar ODEs first. Consider the ODE

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t} = F(t,u) \\ u(t_0) = u_0 \end{cases} \tag{5.1}$$

Assume we want to approximate the path of u numerically on some time grid consisting of equidistant points $t_0, t_1, \dots, t_N$. We can generate such an approximation using the forward Euler method

$$U_{n+1} = U_n + \Delta t F(t_n, U_n), \tag{5.2}$$

where Δt is the distance between time points. Recall that producing such a simulation solves the forward problem for this ODE. In other words, given an ODE, we produce predictions U_n for the solution.

Assume now that F is not known, but we are instead given measurements $U_0, U_1, \dots, U_N$ of the solution u at time points $t_0, \dots, t_N$. We would like to find an approximation $\hat{F}$ to the function F. This is an inverse problem for this ODE, and it is a problem of regression. Recall from Section 4.1.2 that as neural networks are universal approximators, they can, under certain constraints, be used to approximate any given measurable function on a compact domain. Recall also that we are not guaranteed to be able to find such an approximating network. We have to rely on heuristics, and hope that the procedures outlined in Chapter 4 lead to an accurate approximation.

As in Section 4.3 we do not have direct access to F, so we cannot simply measure the error

$$\sum_n \Big(F(t_n, U_n) - \hat{F}(t_n, U_n)\Big)^2,$$

which is normally used for regression tasks. Instead, we use a numerical scheme such as (5.2) with $\hat{F}$ acting in place of the real source F to generate $\hat{U}_{n+1} \approx u(t_{n+1}) = U_{n+1}$ given a data point U_n. Using the forward Euler method, we would first generate

$$\hat{U}_{n+1} = U_n + \Delta t \hat{F}(t_n, U_n), \tag{5.3}$$

for every n, and then use

$$\sum_n (U_{n+1} - \hat{U}_{n+1})^2 \tag{5.4}$$

to evaluate the performance of the network. Since all operations used in (5.3) and (5.4) are differentiable, we can safely use automatic differentiation to train the network $\hat{F}$.

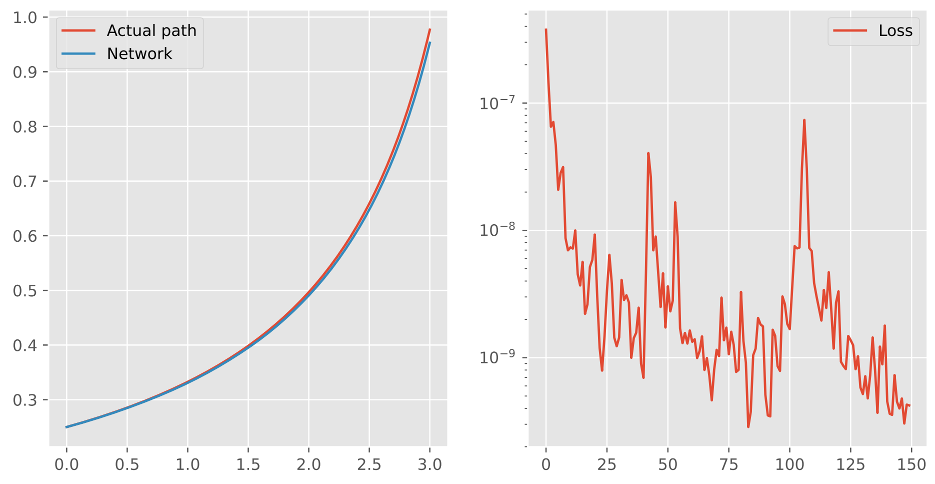


Figure 5.1: The left plot shows the paths generated using the Euler method with the exact F as well as the network $\hat{F}$. The right plot shows the loss function of the network as a function of the training iterations.

5.1 Experiments

5.1.1 Quadratic source

As a test problem, we used the forward Euler method to generate observations of the function u solving (5.1) with

$$F(t, u) = u^2 \tag{5.5}$$

and $u_0 = 0.25$. The time grid was an equidistant partition of the interval $[0, 3]$ with $\Delta t = 0.01$. This resulted in a dataset $U_0, \dots, U_N$ with $N = 298$. We used a neural network $\hat{F}$ consisting of 3 hidden layers and an output layer to approximate F. The hidden layers each had 30 units and used the ReLU function as activation. The weights were initialized according to the He uniform scheme. The output layer had a single unit with no activation. We used the Adam optimizer with a learning rate of 0.005 and a batch size of 10.

After training for 150 epochs, we used $\hat{F}$ in combination with the Euler method to generate a simulation of the solution of (5.1). The L^2 error was $1.92 \cdot 10^{-4}$. The results are presented in Figure 5.1.

The initial experiment demonstrates that given enough data, a NINN is able to accurately predict a source function using a relatively small neural network. Although the experiment represents a straightforward scenario, it shows that the NINN training procedure is at least feasible.

5.1.2 Scaled datasets

When training neural networks it can often be beneficial to appropriately scale the training set. Since we are training not against direct measurements of F, but rather mapping the outputs of F to our solution space using the numerical scheme, scaling is somewhat more complicated here than for usual regression tasks. It is still possible however, and to demonstrate this we scaled the measurements in our test problem down to the range $[0, 1]$ prior to training. That is, prior to training we applied the transformation

$$\tilde{U}_n = \frac{U_n - U_{min}}{U_{max} - U_{min}} =: T(U_n), \tag{5.6}$$

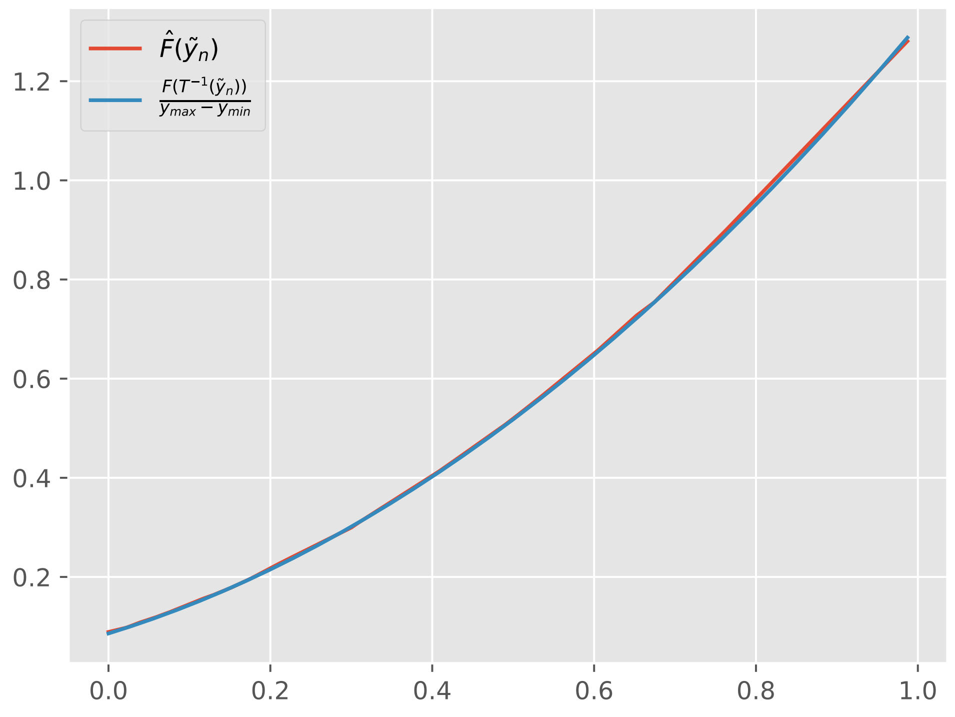


Figure 5.2: The figure shows that the network satisfies (5.8) after training. The L^2 error was $2.43 \cdot 10^{-5}$.

to our training set, where $U_{min} = \min_n U_n$ and $U_{max} = \max_n U_n$. Notice that T is invertible, so we can scale the data back when we use the network to generate the actual path. Importantly, this means the network is not actually approximating F. To see this, note that the network is trained to satisfy

$$\tilde{U}_{n+1} \approx \tilde{U}_n + \Delta t \hat{F}(\tilde{U}_n). \tag{5.7}$$

We used the forward Euler method to generate the training data, so

$$\begin{aligned} U_{n+1} &= U_n + \Delta t F(U_n) \\ \Rightarrow \quad \tilde{U}_{n+1} &= T(U_n + \Delta t F(U_n)) \\ &= \frac{U_n + \Delta t F(U_n) - U_{min}}{U_{max} - U_{min}} \\ &= \tilde{U}_n + \Delta t \frac{F(U_n)}{U_{max} - U_{min}}. \end{aligned}$$

Thus, we are really training the network to satisfy

$$\hat{F}(\tilde{U}_n) \approx \frac{F(T^{-1}(\tilde{U}_n))}{U_{max} - U_{min}}. \tag{5.8}$$

These functions are plotted in Figure 5.2, showing that the relation holds. These relations become more difficult to compute when we switch to more complex numerical schemes in the following chapters, and for the test cases explored in this thesis we can achieve high quality approximations without scaling. Thus, in the following chapters, we will not apply scaling to our traninig set.

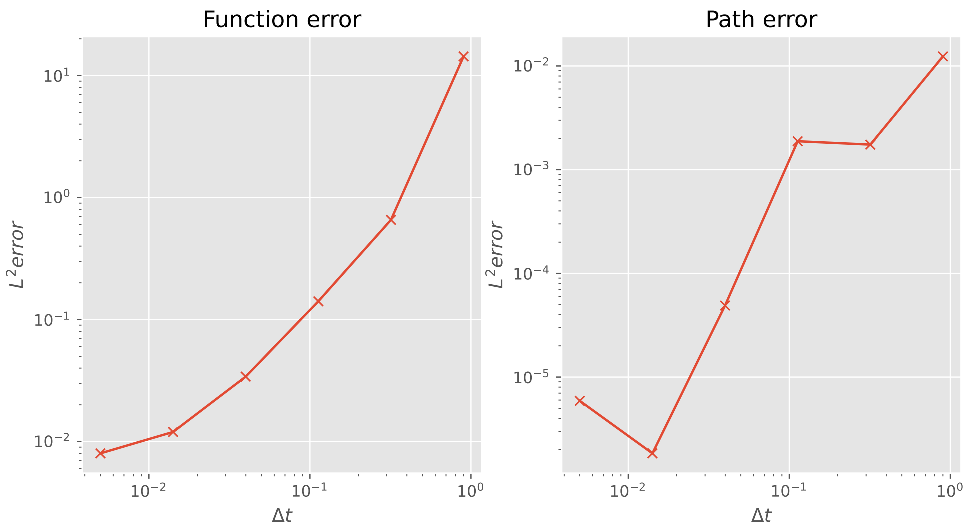


Figure 5.3: The results of the refinement experiment. The left plot shows the L^2 error between the network and (5.8), and the right shows the error in the path produced using forward Euler with $\hat{F}$ instead of (5.5).

5.1.3 Time refinement

The time refinement also plays a role in how well the network performs, as it decides the size of the training set. In general, we expect better performance when more data is seen during the training phase, that is when Δt is small. To quantify this, we measured the L^2 error of the network prediction as well as the path it generated through the forward Euler process while varying Δt. The network was compared to the function in (5.8), and the path it generated was compared against the path generated by (5.5). We used the same network structure as before, and picked the best performing weights seen after 140 epochs. The network weights were reset after each Δt change. The results can be seen in Figure 5.3.

Unsurprisingly, the network performs better when it has access to more training data. It is however worth noting that the path error does not decrease monotonously with Δt. This is a phenomenon we will see in the following chapters as well. Nevertheless, the finest discretizations yield errors close to 10^{-6}, which is very low. The network is implemented using single precision (32 bit) floating point numbers, meaning the machine precision is roughly 10^{-8}.

Chapter 6

Learning source terms in the linear transport equation

In this section we will consider the case of linear flux, giving rise to the linear transport equation

$$\begin{cases} u_t + au_x = q \\ u(0,x) = u_0(x) \end{cases} \tag{6.1}$$

with periodic boundary conditions. Recall from the discussion in Section 2.2 that the characteristics of (6.1) are straight lines defined by the wave speed a, and thus any smooth initial data will result in smooth solutions. Recall also that the solution was simply the initial data u_0 translated according to the wave speed a added to the compounded contributions of q along the characteristics.

Let $\Omega = [-1, 1]$, and define a mesh on $[0, T] \times \Omega$ by the gridpoints $x_i = -1 + i\Delta x$, $i = 0, \dots, M - 1$, and $t_n = n\Delta t$, $n = 0, \dots, N - 1$. We are interested in learning q from a dataset $\{U_i^n\}_i^n$ containing observations of the solution at each gridpoint. Instead of the forward Euler scheme used in Section 5, we will now use the Lax-Wendroff scheme introduced in Section 3.

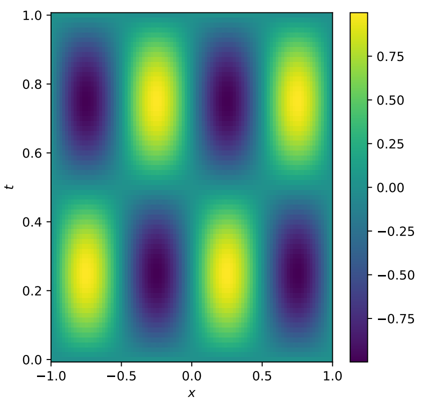
In the following we separate between the terms *source prediction* and *solution prediction*. A source prediction is the output of our neural network for some given input, interpreted as an approximation to the hidden source function q. A solution prediction is an approximation to the solution u of a hyperbolic equation obtained using a numerical technique on observations of the solution and the source prediction. We will denote solution predictions by $\hat{U}$.

Recall from Section 5 that the loss function matched U_{n+1} against a solution prediction generated by the forward Euler method using the current solution value U_n and the source prediction $\hat{F}(t_n, U_n)$. Now we will use the Lax-Wendroff scheme to generate the solution prediction, which depends on three local stencil points $U_{i-1}^n, U_i^n, U_{i+1}^n$ as well as a source prediction $\hat{q}(t_n, x_i, U_i^n)$. As (6.1) is linear, the solution will not develop shocks for smooth initial conditions and source functions. Thus we can avoid the issue of oscillations in the solution generated by Lax-Wendroff, still recieving the benefits of its high order.

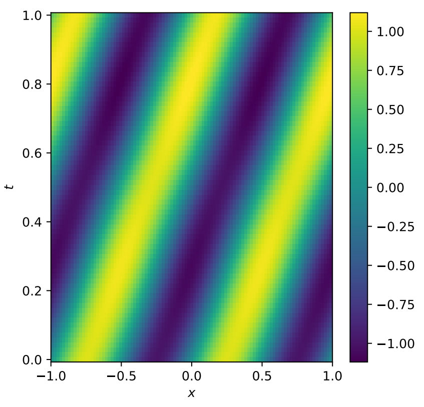
As measurements $\{U_i^n\}_i^n$ we will use values generated by the scheme using the true source function. A training step consists of evaluating the network in a point (t_n, x_i, U_i^n) to obtain a source prediction $\hat{q}_i^n$, and then matching U_i^{n+1} against

$$\hat{U}_i^{n+1} = \mathrm{LW}(U_{i-1}^n, U_i^n, U_{i+1}^n) + \Delta t \hat{q}_i^n,$$

where $\mathrm{LW}(U_{i-1}^n, U_i^n, U_{i+1}^n)$ denotes a Lax-Wendroff step from the stencil $\{U_{i-1}^n, U_i^n, U_{i+1}^n\}$. Thus, our training set consists of rows containing the coordinates (t_n, x_i) as well as the entire stencil $\{U_{i-1}^n, U_i^n, U_{i+1}^n, U_i^{n+1}\}$. In practice, each training step will be performed in a vectorized fashion over a batch of such rows, and the loss for each row in the batch will be accumulated before backpropagating.



(a) Source function q



(b) Corresponding solution u

Figure 6.1: Setup for the learning problem in Section 6.1.1.

6.1 Experiments

6.1.1 Sinusoidal source

Let $T = 1$, $a = 1$,

$$q(t, x, u) = \sin(2\pi t)\sin(2\pi x), \tag{6.2}$$

and

$$u_0(x) = \sin(2\pi x). \tag{6.3}$$

See Figure 6.1 for a visualization. Notice that these functions all take values on $[-1, 1]$. We will not scale the values before training as we did in the previous section. We will use a spatial grid with $M = 100$ cells. In order to satisfy the CFL condition, we set a threshold γ and calculate

$$N = \left\lceil \frac{aTM}{|\Omega|\gamma} \right\rceil, \tag{6.4}$$

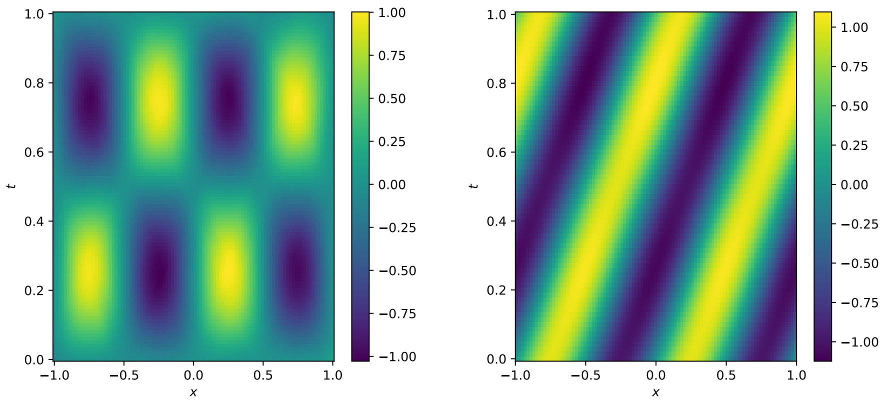
ensuring $a\Delta t/\Delta x < \gamma$. With $\gamma = 0.7$ we obtain $N = 72$.

We used a neural network with a similar configuration as in Section 5, that is 3 hidden layers each containing 30 units. Here the hidden layers were activated by a leaky ReLU function with a slope of 0.01, and the weights were initialized according to the He uniform scheme, see table 6.1. We used the Adam optimizer with a learning rate of 0.005, and the batch size was 32. We ran the training loop 5 independent times, each time resetting the model weights, and trained for 8 epochs each run.

Figure 6.2 shows predictions of the model averaged over the 5 independent runs. Figure 6.3 shows the training loss history. Figure 6.4 shows plots of the squared error in the source prediction and solution prediction. The L^2 error of the source prediction and solution prediction were $2.03 \cdot 10^{-3}$ and $2.62 \cdot 10^{-4}$ respectively.

The performance of the algorithm was evaluated for several values of M. Specifically, we computed both the source error and solution error of the averaged predictions over 5 independent training runs for each $M = 2^k$, $k = 3 \ldots, 9$. The results are plotted in Figure 6.5.

Figures 6.1 and 6.2 indicate that we are able to learn sinusoidal source functions well, especially when the frequency is not too high. However, in this case we provided the network with a large amount of training data. With $M = 100$, $N = 72$, the network had access to $M(N-1) = 7100$ data points during training. Looking at Figure 6.5 we see that even when coarsening the grid down to $M = 8$, giving the network only 40 training points in the domain, we still perform quite well.



(a) Predicted source $\hat{q}$ **(b)** Corresponding solution prediction $\hat{U}$

Figure 6.2: Network predictions for the problem in Section 6.1.1. Plotted is the averaged results from 5 independently trained models.

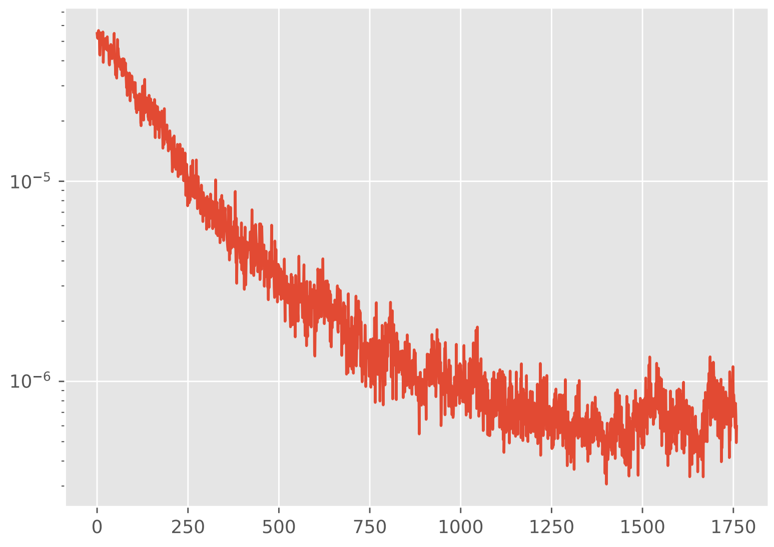
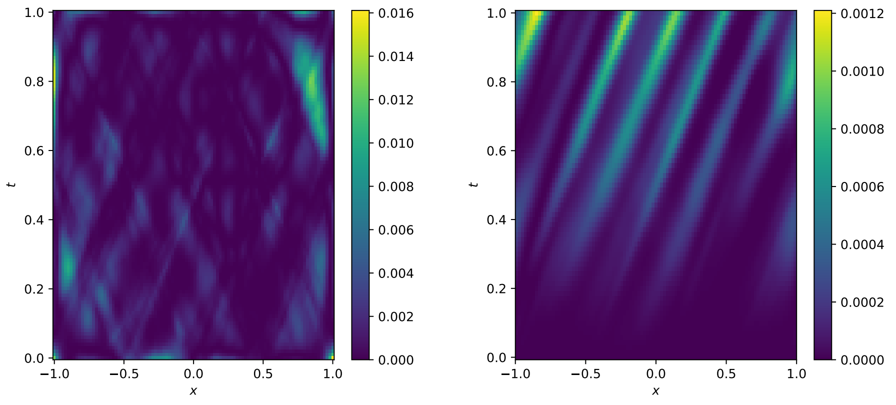


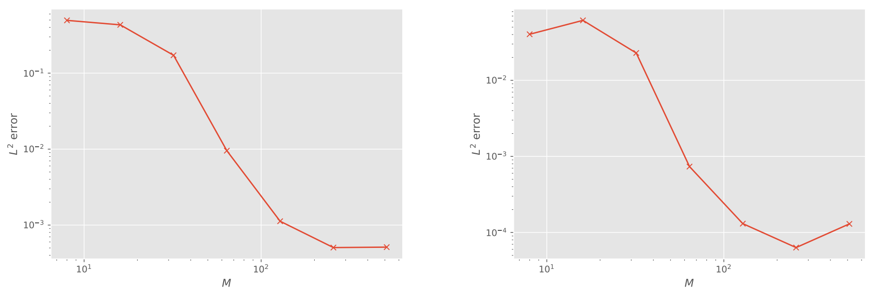
Figure 6.3: Averaged training loss history for Section 6.1.1.



(a) Error in source function

(b) Error in solution

Figure 6.4: Plots of the squared error in the network prediction of the source function, and the error in the path generated from this prediction for Section 6.1.1.



(a) Source

(b) Solution

Figure 6.5: The L^2 error for increasing values of M in Section 6.1.1, still forcing a CFL threshold of 0.7.

Layer	Input dimension	Output dimension	Activation
1	2	30	Leaky ReLU
2	30	30	Leaky ReLU
3	30	30	Leaky ReLU
4	30	1	Linear

Table 6.1: Neural network structure used in Section 6.1.1. The slope of the leaky ReLU was 0.01, and the optimizer was Adam with learning rate 0.005.

Looking at Figure 6.4, there does not seem to be a pattern in the source error. Interestingly, however, it seems like errors made during the solution prediction are propagated along characteristic lines until $t = T$. This makes sense as the solution values themselves propagate along these lines, and because of the time stepping procedure used in the numerical scheme, errors are compounded as they pass through time. However, some characteristics propagate a much higher error than others. Notice also that as the boundary conditions are periodic, errors are propagated through the right side of the domain only to reappear on the left.

Chapter 7

Learning source terms in Burgers' equation

We now move on to Burgers' equation

$$\begin{cases} u_t + \left(\frac{1}{2}u^2\right)_x = q \\ u(0,x) = u_0(x). \end{cases} \tag{7.1}$$

As discussed in Section 2.3, the solution will now develop shocks and/or rarefaction waves for any nontrivial initial condition.

Let again $\Omega = [-1,1]$, and define a mesh on $[0,T] \times \Omega$ by the gridpoints $x_i = -1 + i\Delta x$, $i = 0,\dots,M-1$, and $t_n = n\Delta t$, $n = 0,\dots,N-1$. As the Lax-Wendroff scheme is known to produce oscillations close to shocks, we now switch to the Godunov scheme. As Godunov uses the same stencil as Lax-Wendroff, the training set will have the exact same structure as in Chapter 6, it will just be generated in a different way. This means that we can use the same training procedure, allowing us to reuse much of our training code.

7.1 Experiments

7.1.1 Sinusoidal source

Let $T = 1$,

$$q(t,x,u) = \sin(2\pi t)\sin(2\pi x), \tag{7.2}$$

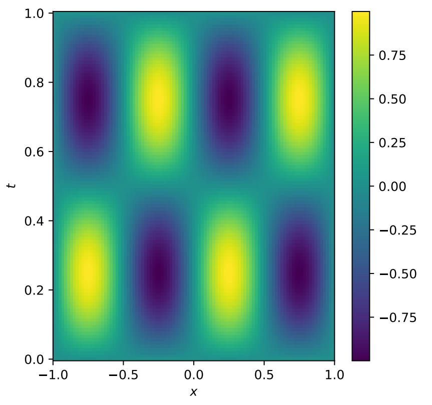
and

$$u_0(x) = -\sin(\pi x). \tag{7.3}$$

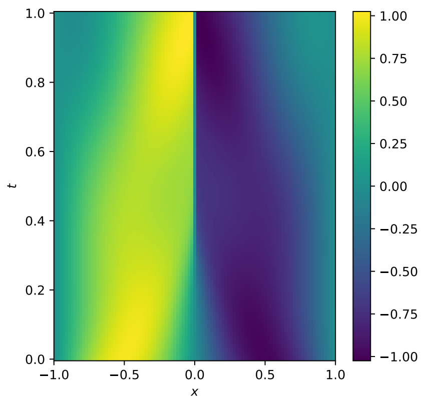
See Figure 7.1 for a visualization.

We will again use a spatial grid with $M = 100$ cells and a CFL threshold of $\gamma = 0.7$. As the wave speed is now unknown, we assume a bound $u \leq 1.5$ on $[0,T] \times \Omega$ and use (6.4) to obtain $N = 107$, satisfying the CFL condition as long as u does not breach the assumed bound. Looking at Figure 7.1, this is not an unreasonable assumption. We keep Δt constant to simplify implementation.

We used an ensemble model consisting of 5 independently trained networks, each with the structure described in Table 6.1, where the model prediction is the average prediction of each network. Each network was trained for 8 epochs, and was optimized using the Adam algorithm with learning rate 0.005 and batch size 32. Figure 7.2 shows predictions of the model. Figure 7.3 shows the averaged training loss history for each network. Figure 7.4 shows plots of the squared error in the source prediction and solution prediction. The relative L^2 error of the source and solution predictions were $1.89 \cdot 10^{-3}$ and $1.62 \cdot 10^{-3}$.

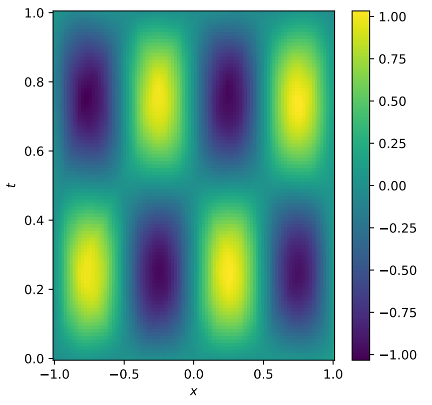


(a) Source function q

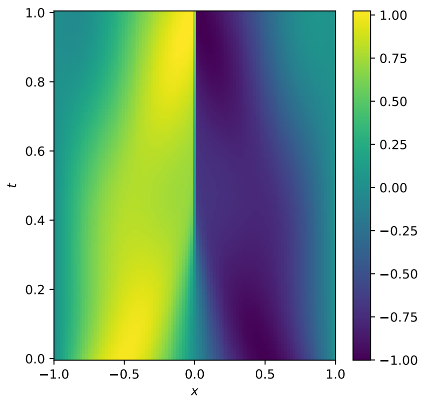


(b) Corresponding solution u

Figure 7.1: Setup for the learning problem in Section 7.1.1.



(a) Predicted source $\hat{q}$



(b) Corresponding solution prediction $\hat{u}$

Figure 7.2: Network predictions for the learning problem in Section 7.1.1.

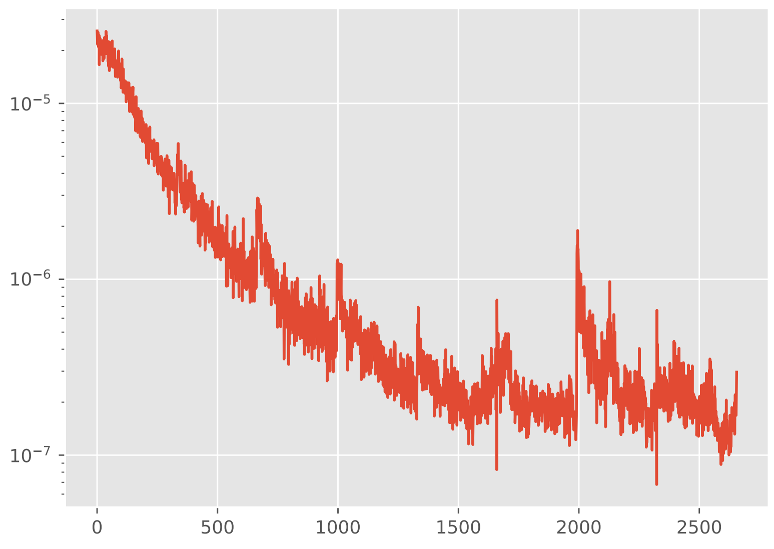
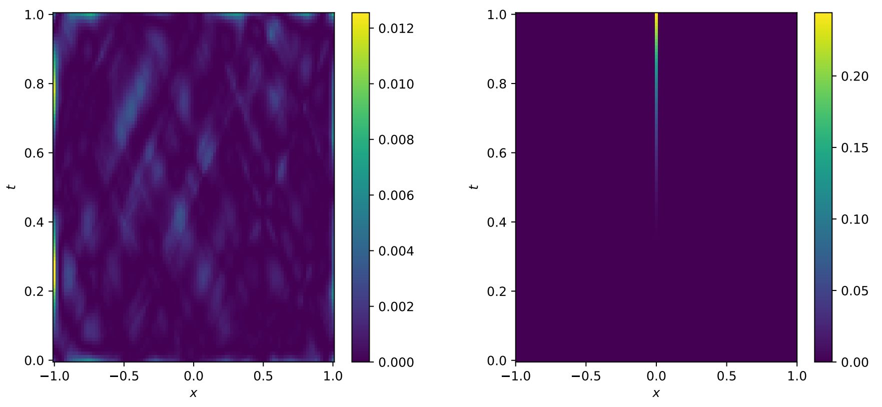
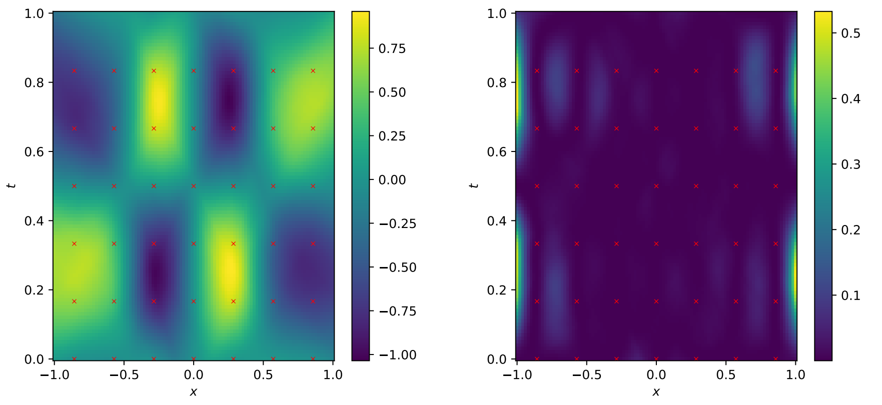


Figure 7.3: Averaged training loss history for Section 7.1.1.



(a) Error in source function **(b)** Error in solution

Figure 7.4: Plots of the squared error in the model prediction of the source function, and the error in the path generated from this prediction for Section 7.1.1.



(a) Source prediction **(b)** Error

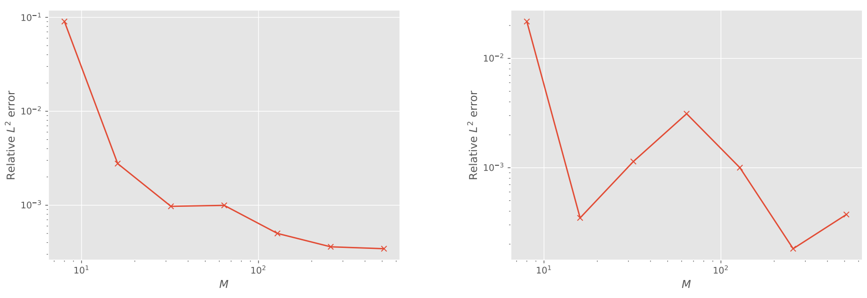
Figure 7.5: The source predictions of the neural network when $M = 8$, with corresponding L^2 error. The red marks shows the location of the centroids of the finite volume cells in the training set.

We repeat the experiment with a rougher grid, setting $M = 8$. As the dataset now consists of much fewer points, we train for 200 epochs to reach convergence. The number of epochs was increased to account for the fact that a smaller training set will lead to fewer weight updates per epoch when the batch size is kept constant. In this case, the network achieved a relative L^2 error of $8.60 \cdot 10^{-2}$ in the source prediction. The results, which can be seen in Figure 7.5, shows that the method produce reasonable predictions even in cases with a very limitied number of observations.

Notice that there is an area at the top of the plots in Figure 7.5 that completely lacks training points. That is because in order to use a point (t, x) in the training set, we need access to the solution values $u(t + \Delta t, x)$. As we only have access to observations of the solution in $[0, T] \times \Omega$, we cannot use training points with $t = T$.

Figure 7.6 shows the L^2 source and solution errors of the model for different values of M, allowing the algorithm to reach convergence. We see that the performance gain in source prediction from increasing M starts to disappear after about $M = 100$. This is to be expected, as the convergence rate of the source is proportional to the loss (see Section 4.4), and with a network implemented with single precision floating point parameters there is a limit to how low we can push the loss value.

Unlike in Section 6, we now have a discontinuity that emerges after some time as a result of two



(a) Source **(b)** Solution

Figure 7.6: The relative L^2 error for increasing values of M in Section 7.1.1, still enforcing a CFL threshold of 0.7.

wavefronts colliding. However, we are still able to learn the source quite well. Figure 7.4 shows that the solution error is dominated by the error at the shock, but it seems the predicted source is not affected by this at all. The reason we are able to learn the source function even in the shock region, is that the shock is not captured in the backpropagation when producing the gradient of a networks prediction with respect to its parameters. To reiterate the argument from Section 4.4, letting $G(U_i^n)$ denote a step with our numerical scheme centered at U_i^n, we can write

$$U_i^{n+1} = G(U_i^n) + \Delta t q(t_n, x_i), \tag{7.4}$$

$$\hat{U}_i^{n+1} = G(U_i^n) + \Delta t \hat{q}(t_n, x_i). \tag{7.5}$$

Then the loss function becomes

$$(U_i^{n+1} - \hat{U}_i^{n+1})^2 = \Delta t^2 (q(t_n, x_i) - \hat{q}(t_n, x_i))^2, \tag{7.6}$$

which is independent on U. Thus, when we calculate the gradient of (7.6) with respect to the network parameters in order to apply our optimization routine, we never need to use the derivative with respect to U in the chain rule expansion, meaning the training algorithm never notices the discontinuity. One might then imagine that this only works when q is a function of t and x only, and that the backpropagation becomes more difficult when q also depends on the solution. This is in fact not a problem. Imagine adding a dependence on U_i^n to q and $\hat{q}$ in (7.6). As U_i^n is independent on the network parameters θ, when we differentiate (7.6) with respect to θ during training, all differentials of U_i^n vanish. Thus the training process is unaffected by the additional dependence on U_i^n, even when the solution develops discontinuities. This will be illustrated in the following section.

7.1.2 Non-smooth solution dependent source

We will still operate with $T = 1$ and

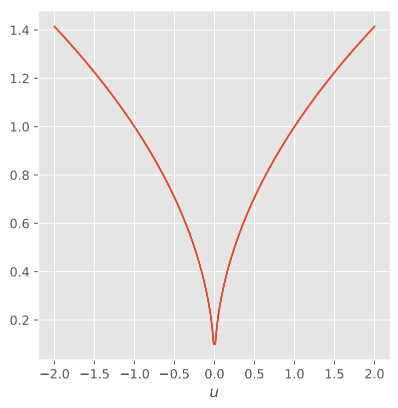
$$u_0(x) = -\sin(\pi x). \tag{7.7}$$

However, we now let

$$q(t, x, u) = \sqrt{|u|}. \tag{7.8}$$

This case is illustrated in Figure 7.7. We will use the ensemble model described in the previous section, but we change the first input dimension of each network from 2 to 3.

There are now in particular two potential problems for the model. First of all, we are now dealing with a source that is not differentiable in points $(t, x, 0)$. Secondly, we have a problem related to extrapolation. In the previous cases we only needed the values (t, x) from the training set to evaluate the network, and as we chose the grid in advance, we were free to pick these training points freely as long as we made sure to generate observations of u at these points. We are still able to pick (t, x) freely, but we must now use the generated observations of u that corresponds


(a) Source function q

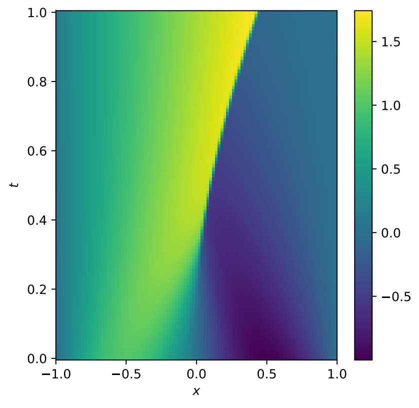

(b) Corresponding solution u

Figure 7.7: Setup for the learning problem in Section 7.1.2. The source takes values from $\mathbb{R}^3$, but it only varies with the last dimension.

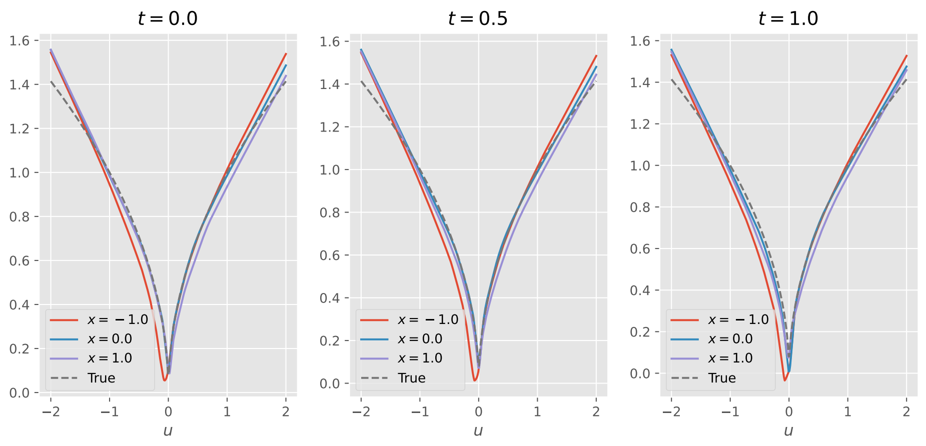


Figure 7.8: Source predictions for a range of different values of t and x in Section 7.1.2.

to this point in our training set. This means we can make sure that during training, the networks see training points with a varied selection of values for t and x, but we do not necessarily have any control of the selection of u values in the training set. For instance, we can see from Figure 7.7(b) that in Section 7.1.2 the networks will only ever have access to u values roughly inside $[-1, 1.5]$, an interval that is skewed as a result of the nonnegativity of the source. Another difficulty is that a fixed point $(\tilde{t}, \tilde{x})$ is only ever paired with a single value $\tilde{u}$ in the training set, meaning that the networks must first learn independence from t and x in order to use information on u from other spatiotemporal points. Thus, the model will have to extrapolate between the missing values.

Figure 7.8 shows the model predictions for u varying in $[-2, 2]$ at several different fixed points (t, x). Clearly, even when the model only has access to very limited information on u, it is able to learn the source function well. We see that the prediction error is slightly higher for $u < -1$, which is reasonable given that we had no training points in this regime. However, even in areas where we had few to none training points, the model provides a decent approximation to the source. We notice in particular that it has learned independence from t and x, and that the position of the non-differentiable point in the source function is only off by a small amount only for x values of -1.

Figure 7.9 shows the solution generated by the model. We notice in particular that the model generates a solution with a correctly positioned shock.

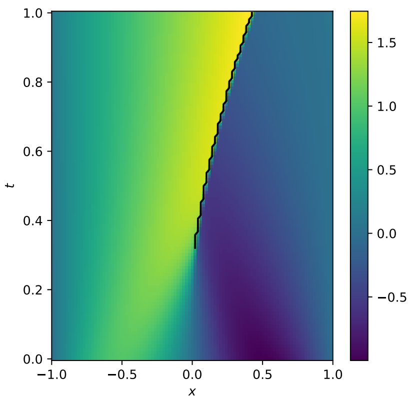


Figure 7.9: The solution generated using the trained model from Section 7.1.2. The black line shows the shock position in the reference solution.

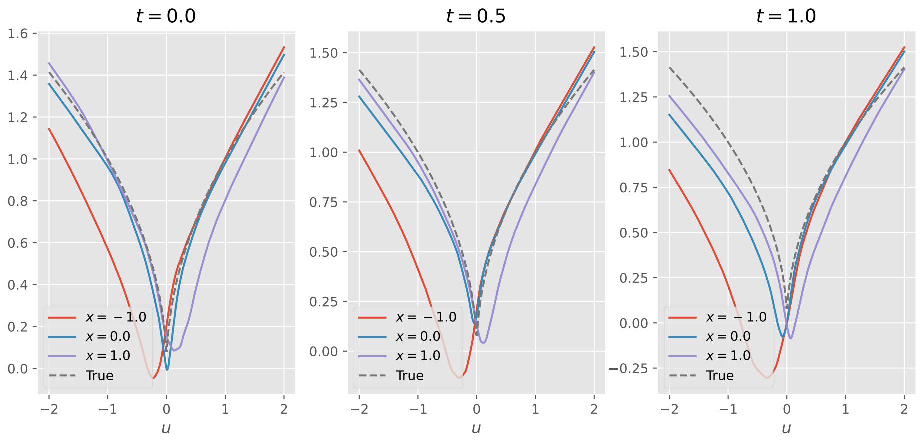


Figure 7.10: Model predictions for Section 7.1.2 when $M = 8$.

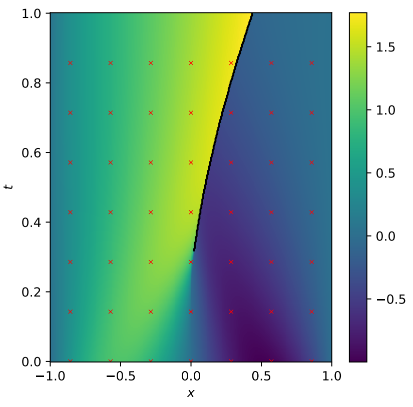


Figure 7.11: The solution generated using the trained model from Section 7.1.2 when $M = 8$. The black line shows the shock position in the reference solution.

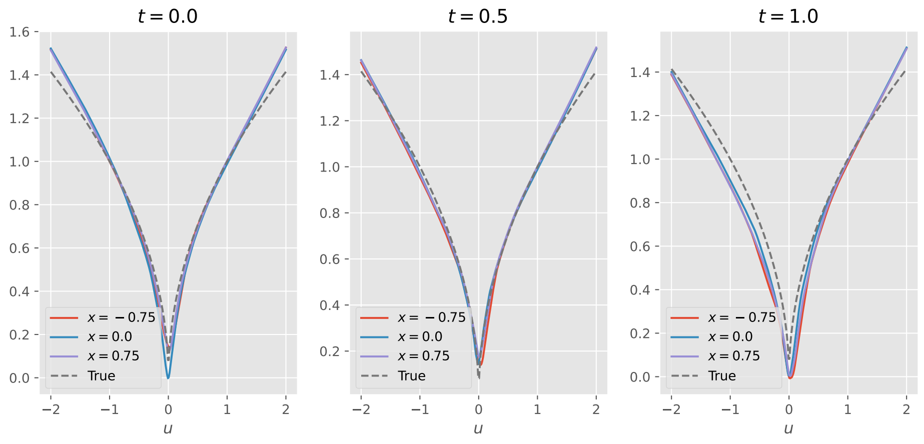


Figure 7.12: Source predictions for Section 7.1.2 with $M = 8$ and enforcement of periodicity.

Like in Section 7.1.1, we repeat this experiment for a rough grid. Setting $M = 8$ we train for 200 epochs and visualize the results in Figure 7.10. The predictions are now considerably worse than in the high resolution case, but the network is still able to predict the general shape of the source function. Notice how the predictions for $x = 0$ and $x = 1$ are better than for $x = -1$ at all timepoints. For times $t = 0$ and $t = 0.5$ it is slightly easier for the network to predict the source when $x = 0$ than when $x = 1$, but this difference disappears when $t = 1$. This is connected to the positions of the training points, shown in Figure 7.11.

It is very important for the network to correctly predict the source function at $x = 0$, as this is close to the position of the shock. Differences in the shock position between the true and predicted solution corresponds to a big difference in the true and predicted solution value, which is penalized heavily by the loss function. As the shock moves towards $x = 1$, it is reasonable to think that the network also needs to be accurate at the right side of the domain, and therefore prioritizes accuracy at $x > 0$ as well. By the same argument the network is somewhat less incentivised to correctly learn the source at $x = -1$, as the solution values predicted here will not propagate to the shock in time.

This is problematic as Ω is considered periodic, and thus the network should predict the same function at $x = -1$ and $x = 1$. The true source is of course periodic, but it seems this property is not prioritized by the network. Periodicity can be enforced by adding an additional penalty in the loss function. Adding such a penalty produce the results shown in Figure 7.12. We now achieve much better approximations, and all we did was enforcing a property we had a priori

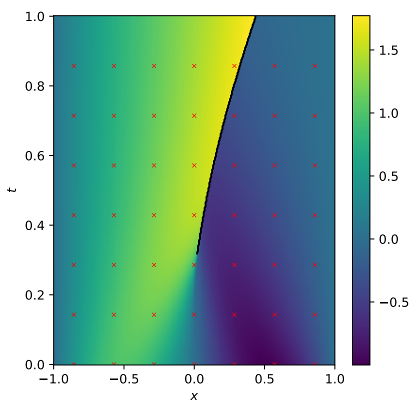


Figure 7.13: The solution generated using the trained model from Section 7.1.2 with $M = 8$ and enforcement of periodicity. The black line shows the shock position in the reference solution.

knowledge about. This is invaluable in practice, as we can incorporate expert knowledge of the source function by either pretraining or forcing certain properties of the network during training. Figure 7.13 shows that the network still generates a solution with a correct shock position.

Chapter 8

Learning bathymetry through the shallow water equations

As a final test setting, we consider the shallow water equations (SWE) in 1D. This is a hyperbolic system modelling the flow of fluid with a free surface when the characteristic height of the surface is small compared to the characteristic length scale in the spatial dimension. Let $B = B(x)$ describe the bottom topography, which for instance in oceanography corresponds to the height of the seafloor relative to some baseline. In this context we call B the *bathymetry*. Let $h = h(t, x)$ be the height of the free surface relative to B and $u = u(t, x)$ the depth-averaged velocity of the fluid in the x direction at time t and position x. See Figure 8.1 for a visualization.

Using the conservation of mass and momentum we can derive the shallow water equations

$$\begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ -ghB_x \end{bmatrix}, \tag{8.1}$$

where $g = 9.81\,\mathrm{m/s^2}$ is the gravitational acceleration on the earth's surface. See for instance [27] for details on the derivation of (8.1). The flux function is

$$F(h, u) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}, \tag{8.2}$$

and its Jacobian has eigenvalues $\lambda_{1,2} = hu \pm \sqrt{gh}$. We will use the notation

$$Q_i^n = \begin{bmatrix} h_i^n \\ h_i^n u_i^n \end{bmatrix}$$

for the vector of observations of the conserved quantities at position x_i and time t_n, and $Q_i^n[k]$, $k = 1, 2$, will denote the k^{th} component of this vector.

The goal is to learn the bathymetry through observations of h and u. An added complication with the SWE compared to the previous examples is that the source term depends on B_x and not B directly. As neural networks are differentiable with respect to their input, we can train a network $\hat{B}$ by forcing $\hat{B}_x$ to generate solution predictions through some numerical scheme that matches our observations of h and u. The difference from the previous examples is that we will not be able to learn the reference level of the bathymetry. As we are only training $\hat{B}_x$ to approximate B_x, our network predictions will at best satisfy $\hat{B} = B + C$ for some $C \in \mathbb{R}$. We will therefore assume there is one x for which we have access to an accurate measurement of $B(x)$. Initially we will also assume we have access to accurate measurements of both h and u on our entire spatiotemporal grid. We will use the CUW method from Section 3.4 for all examples involving the SWE.

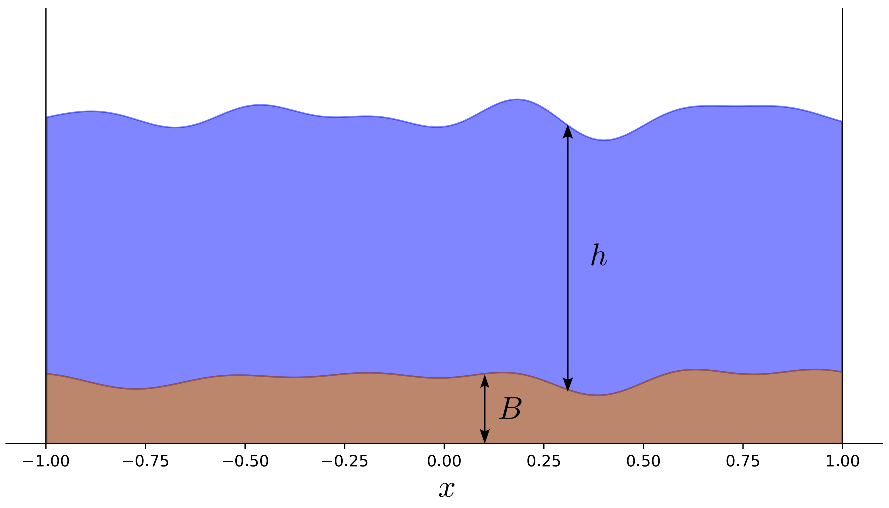


Figure 8.1: Diagram for the shallow water equations.

8.1 Experiments

8.1.1 Predicting bathymetry gradients directly

Consider the domain $[0,T]\times\Omega$ for $T=1$ and $\Omega=[-1,1]$. We will equip equation (8.1) with zero Neumann conditions, that is

$$h_x(t,-1)=h_x(t,1)=0 \tag{8.3}$$
$$u_x(t,-1)=u_x(t,1)=0. \tag{8.4}$$

Let $\mu=1/4$, $\sigma=1/10$, and define the bathymetry as

$$B(x)=\exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right)+\frac{1}{2}\exp\left(-\frac{(x-\mu+1/4)^2}{\sigma^2}\right) \tag{8.5}$$

and the initial conditions as

$$h_0(x)=\exp\left(-\frac{(x+1/2)^2}{\sigma^2}\right)+\frac{3}{2}-B(x) \tag{8.6}$$
$$u_0(x)=0. \tag{8.7}$$

Our grid will consist of $M=100$ spatial cells and $N=526$ timepoints.

In order to check if our method is still feasible in this case, we first try to let the network represent $B_x(x)$ in the training process. That is, an unbatched training step will consist of producing a approximation $\hat{B}_x(x_i)$ to $B_x(x_i)$ using our network, and then matching $Q_i^n[2]$ against

$$\mathrm{CUW}(Q_{i-1}^n,Q_i^n,Q_{i+1}^n)[2]-\Delta t g\hat{B}_x(x_i)Q_i^n[1],$$

where CUW denotes the stepping function of the central upwind scheme.

We independently train 5 networks with structure according to table 6.1, except for an input dimension of 1 instead of 2, and pick the one with the lowest training loss after 4 epochs. We used the Adam optimizer with learning rate 0.0005 and batch size 32.

Figure 8.2 shows the free surface of the solution generated by the central upwind scheme when using the true and predicted bathymetry derivatives. The surfaces are very similar to each other,

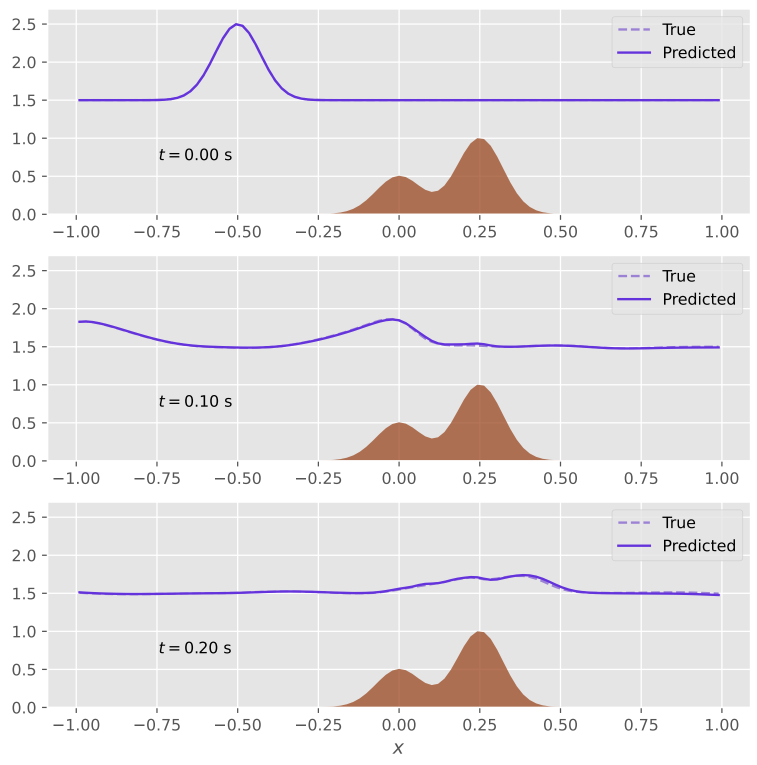


Figure 8.2: Solution predictions at different timepoints for Section 8.1.1 when using the network as an approximator to the derivative B_x of the bathymetry. The brown region is the true bathymetry.

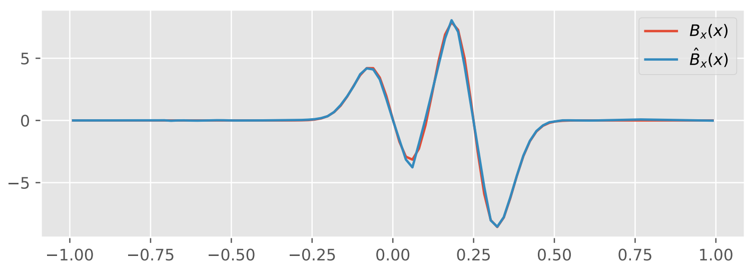


Figure 8.3: Network predictions for Section 8.1.1 when using the network as an approximator to B_x.

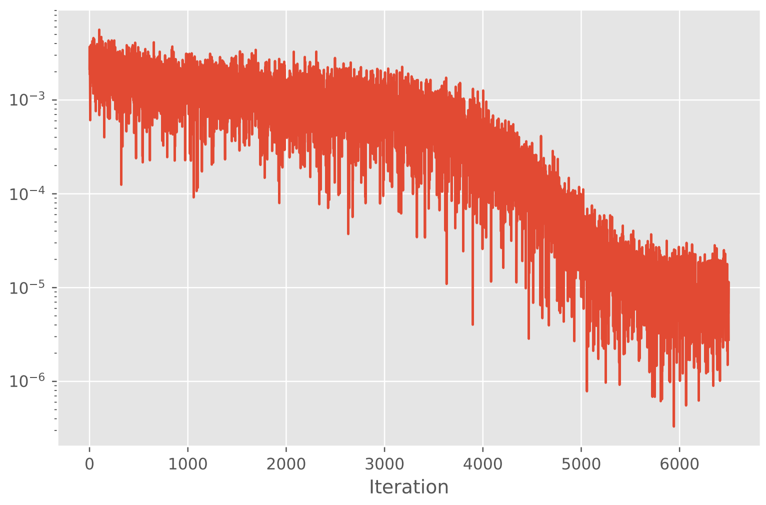


Figure 8.4: Network training history for Section 8.1.1 when using the network as an approximator to B_x.

indicating that the network was able to provide an approximation that performs similarly to B_x when applied to a numerical scheme. In figure 8.3 we see that the network was in fact able to approximate B_x with high accuracy. Figure 8.4 shows the training history of the best performing network after 4 epochs, where we can see that the performance didn't start to improve until the end of the third epoch.

8.1.2 Predicting the bathymetry itself

We now try to let the network approximate B, and modify the training procedure to compute $\hat{B}_x$ using automatic differentiation. That is, we compute the network output $\hat{B}(x_i)$ and apply automatic differentiation to obtain $\hat{B}_x(x_i)$ before proceeding as before with the numerical scheme. We also use the assumption that we have access to one accurate measurement of the bathymetry to remove the constant error in the prediction $\hat{B}$ described earlier.

Figure 8.5 shows the solution generated by the true and predicted bathymetry. Notice that letting the network predict B directly lowered the accuracy of the generated solution. Figure 8.6 shows that the network is able to correctly predict the general shape of the bathymetry, but it struggles in areas where the derivative change rapidly. The training history in Figure 8.8 shows that we are not able to push the loss value much lower than 10^{-4}, roughly an order of magnitude higher after the same number of training steps when using the network as an approximator to $\hat{B}_x$. We can get a clue as to why this is happening by comparing B_x to $\hat{B}_x$. Notice from Figure 8.6 that the predictions of the network resemble a piecewise linear function, meaning the derivatives will look piecewise constant. Figure 8.7, which shows the derivatives of the network compared to the true values, confirms this.

One possible reason why this might be happening is that we are using a leaky ReLU activation function in our network, which is piecewise linear resulting in piecewise constant derivatives. This explains why the network was able to perform well when used as an approximator to B_x, since we at no point were forced to differentiate the network with respect to its input. Recall from the discussion in Section 4.1.2 that Theorem 1, which allows the use of leaky ReLU, does not cover the case of simultaneous function and derivative approximations. In other words, it does not guarantee that arbitrary good approximations $\Phi^H_{\theta*} \approx B$ and $\partial_x \Phi^H_{\theta*} \approx B_x$ can be achieved by a network *at the same time*. Theorem 2 provides the desired guarantee, but it only allows for sigmoid or hyperbolic tangent activation functions. In light of this, we repeat this experiment with a network with the smooth hyperbolic tangent as activation in order to check whether the results improve. The new

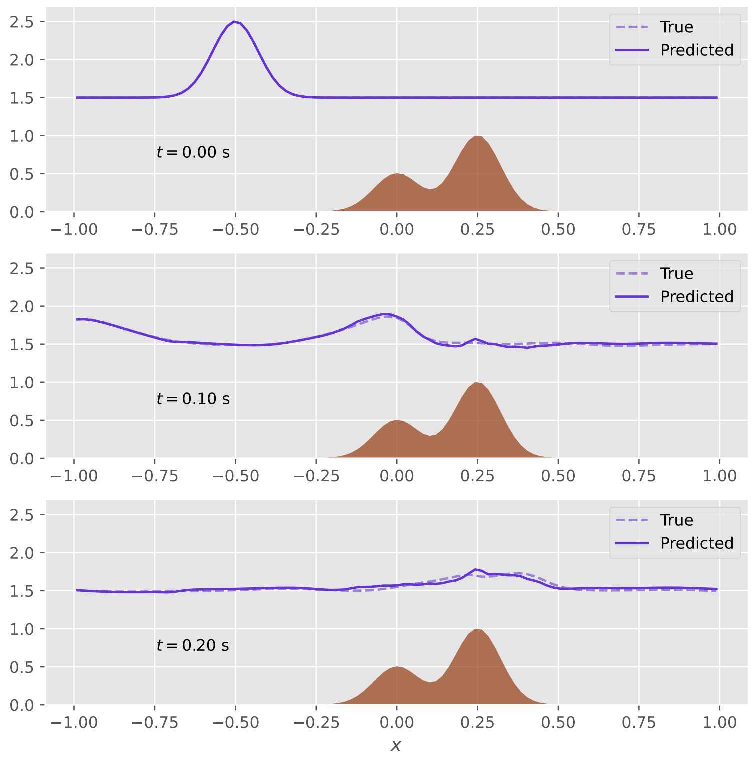


Figure 8.5: Solution predictions at different timepoints for Section 8.1.2 when using the network as an approximator to B.

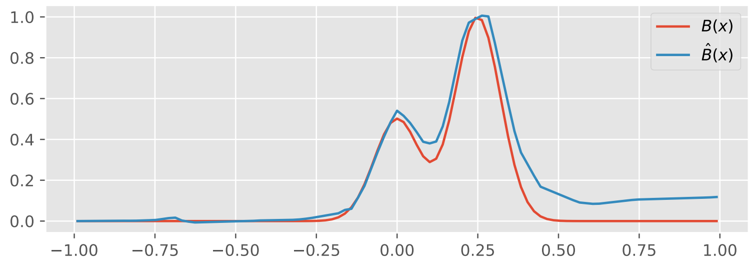


Figure 8.6: Network predictions for Section 8.1.2 when using the network as an approximator to B.

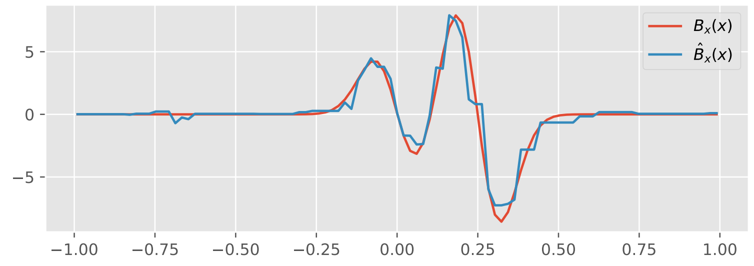


Figure 8.7: Network derivatives for Section 8.1.2 when using the network as an approximator to B.

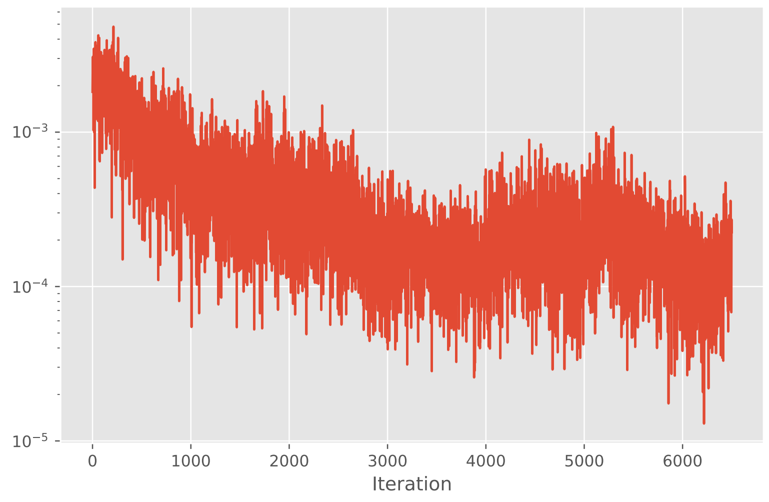


Figure 8.8: Network training history for Section 8.1.2 when using the network as an approximator to B.

Layer	Input dimension	Output dimension	Activation
1	1	30	Tanh
2	30	30	Tanh
3	30	30	Tanh
4	30	1	Linear

Table 8.1: Structure of the hyperbolic tangent activated network used in Section 8.1.2.

structure is outlined in table 8.1.

We see a clear improvement when switching to hyperbolic tangent activation. Figure 8.9 shows that the solution prediction is much better than before, and the training history in Figure 8.12 shows that the performance on the training set increased by roughly three orders of magnitude. Figures 8.10 and 8.11, plotting respectively the network predictions and their derivatives compared to the true values, show that we are now able to compute a completely smooth approximation to the bathymetry, meaning we can achieve a much higher accuracy on the derivatives.

The fact that we are able to improve the results this much by simply changing activation, illustrates a generally very important part of using neural networks as approximators: hyperparameter tuning. The focus of this thesis have been on achieving reasonable predictions through an indirect training approach, and not much weight has been put on tuning the hyperparameters of the networks in order to squeeze out the last bit of performance available. In this case, we made small changes to the structure based on a visual inspection of the predictions, and it turned out to work. However, repeating this experiment with for instance sigmoid activation gives approximations that are smooth, but their accuracy is in fact lower than with leaky ReLU. In addition, ReLU activation is generally considered to perform best for the majority of learning problems with neural networks [39], and is the most used activation function in practice. In [39], Sharma et. al. recognize that the choice of activation function is an important hyperparameter in neural networks, and suggest starting with ReLU and switch to others if the results are not satisfactory, just as we have done here. In an industrial setting, it would be beneficial to perform some kind of proper parameter search in order to achieve the best possible results for the specific usecase, but this is not the focus of this thesis. In addition, most experiments conducted throughout this work has been done with very similar networks, differing only in activation and input dimension, indicating that NINNs are able to perform well without a careful and time consuming hyperparameter search.

The case in Section 8.1.2 was chosen in order to show the potential of neural networks as approximators to unknown parts of systems of hyperbolic equations. The case itself, however, is somewhat

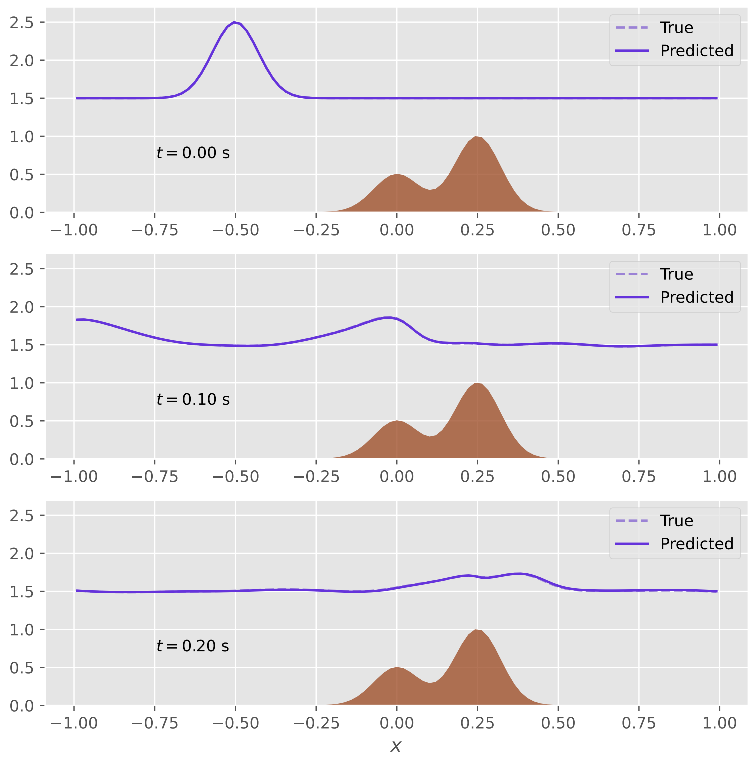


Figure 8.9: Solution predictions at different timepoints for Section 8.1.2 when using the hyperbolic tangent network as an approximator to B.

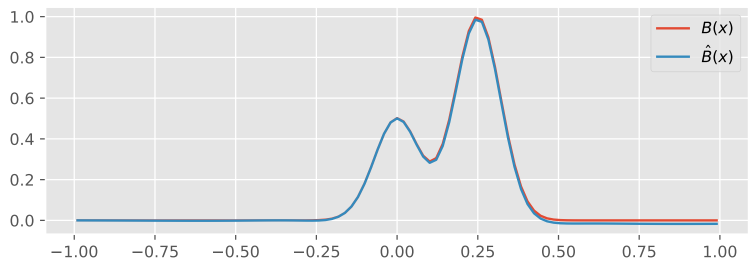


Figure 8.10: Predictions for Section 8.1.2 when using the tanh network as an approximator to B.

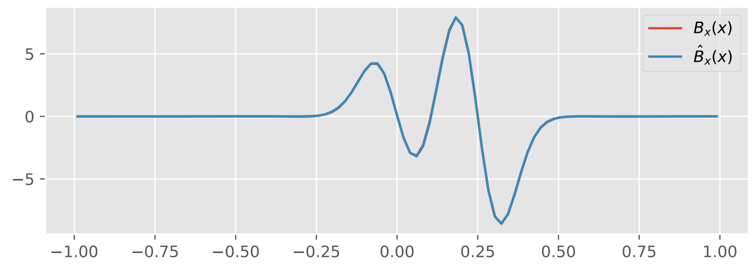


Figure 8.11: Network derivatives for Section 8.1.2 when using the tanh network as an approximator to B.

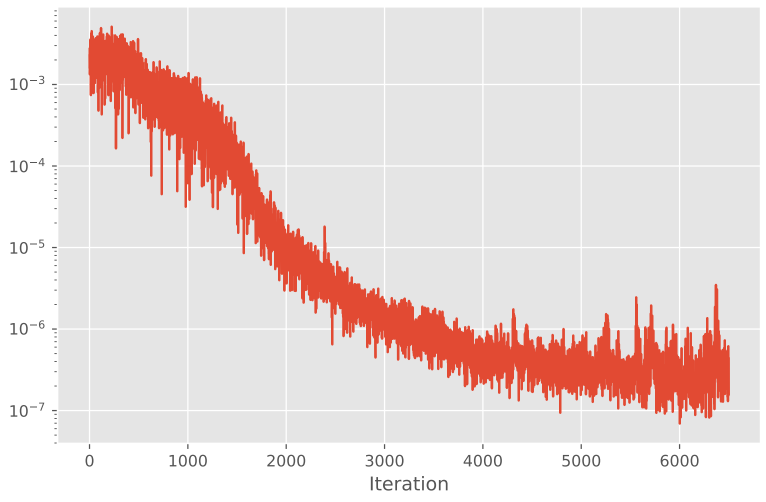


Figure 8.12: Network training history for Section 8.1.2 when using the tanh network as an approximator to B.

unrealistic. We are assuming we have access to accurate measurements of both h and u on a refined grid across our entire domain. In reality, it would be more reasonable to for instance only have access to measurements of u, or some sparse measurements of h spread across the domain in addition to measurements of u on a more refined grid. In order to use our technique, we will however need access to at least some complete measurements of the vector Q at $t = 0$. This motivates the following case.

8.1.3 Incomplete measurements

Consider the domain $[0, T] \times \Omega$ with $\Omega = [-1, 1]$. We again equip (8.1) with the boundary conditions (8.3) and (8.4), and use the bathymetry (8.5). The initial conditions are (8.6) and (8.7). The difference to Section 8.1.2 lies in the measurements we have available. We will now assume we have access to measurements of the entire vector Q only at $t = 0$, and for the following timepoints we only have access to u.

This is a good candidate for the sweeping training procedure described in Section 4.5, as we now only have access to a subset $\tilde{\mathcal{K}} = \{2\}$ of the components of Q at each gridpoint. Notice that for later timesteps we need to perform multiple prediction steps before we can evaluate the loss and update the network weights. This means that training for later timesteps will be more time consuming than for early timesteps. As the bathymetry is time independent, we can account for this by performing fewer iterations at later timesteps. This can be interpreted as treating the first timesteps as a sort of pretraining of the network, as we are performing several weight updates on the early, inexpensive timesteps in order to learn the general shape of the bathymetry before fine tuning on the more computationally demanding timesteps. This way we do not need to iterate a bad network all the way to the last timesteps just for a single weight update. For this reason we do not shuffle the dataset, but rather train on each timestep sequentially from $n = 1$ to $n = N$.

Figures 8.13 to 8.16 show results generated by the sweeping training procedure applied to the system in Section 8.1.1, changing the time horizon to $T = 0.2$ with $N = 105$ for performance reasons. We trained a single network with structure according to table 8.1 for three epochs. In each epoch, the training procedure involved a different number of iterations for different subsets of timesteps. Specifically, we began by training the first 20% of timesteps for 15 iterations. Subsequently, we performed 5 iterations on the remaining timesteps until reaching the last 40% of the dataset, where each timestep was trained for a single iteration.

Figures 8.13 to 8.15 show that the network is able to perform very well even when it lacks ac-

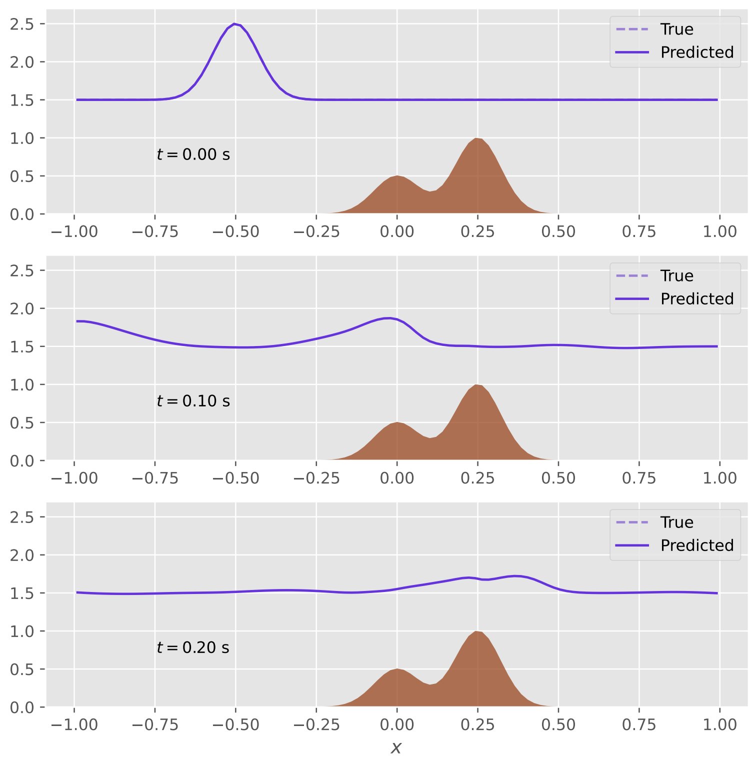


Figure 8.13: Solution predictions at different timepoints for Section 8.1.3 when using the sweeping training procedure.

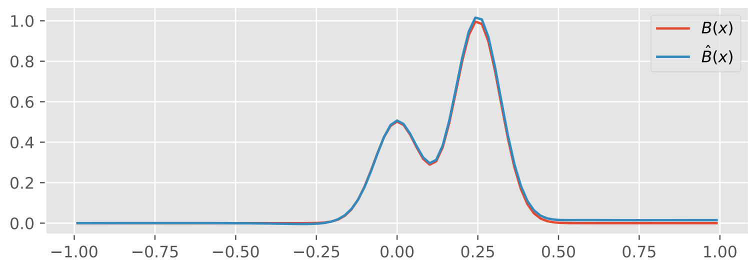


Figure 8.14: Predictions for Section 8.1.3 when using the sweeping training procedure.

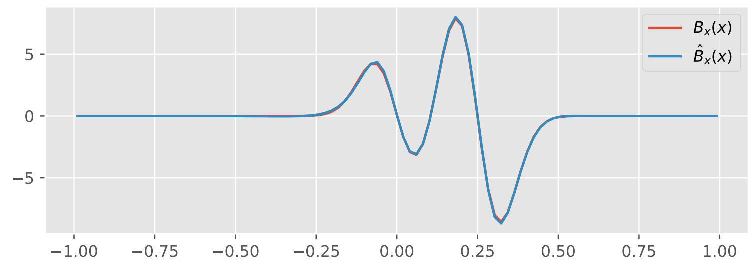


Figure 8.15: Network derivatives for Section 8.1.3 when using the sweeping training procedure.

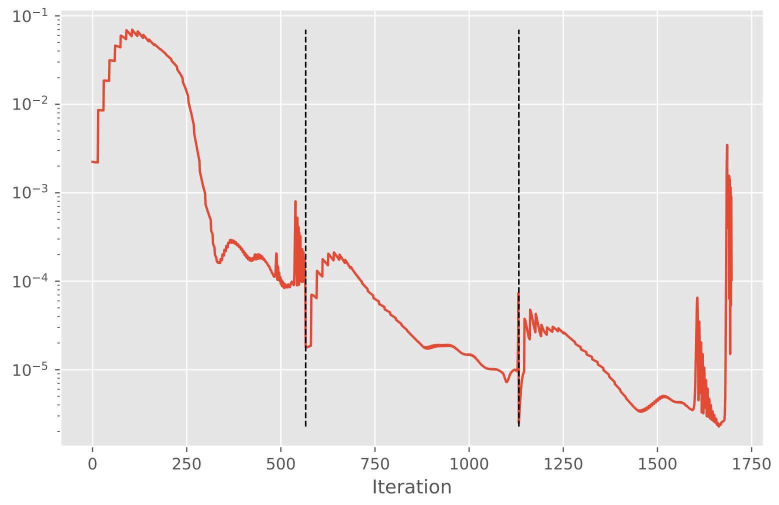


Figure 8.16: Network training history for Section 8.1.3 when using the sweeping training procedure. The stipled black lines mark the end of each epoch.

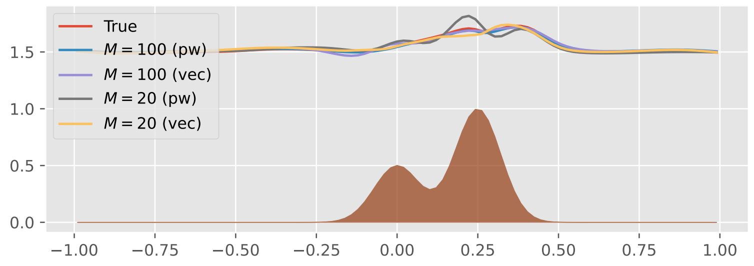


Figure 8.17: Comparison of the solution predictions at $t = 0.2$ in Section 8.1.3 for different M under both the pointwise (pw) and sweeping (vec) training procedure. The abbreviation ”vec” comes from an implementation detail, but it refers to the sweeping algorithm.

cess to the complete dataset. Notice that even though the network never had access to explicit measurements of h for $t > 0$, the solution prediction still has the correct relative surface height. The network misses by a small margin in Figure 8.14 for $x = 0.25$ and $x > 0.5$, but the network derivatives provide a very good approximation of B_x. The training history in Figure 8.16 shows that most of the error is eliminated in the first timesteps of each epoch.

To demonstrate the versatility of NINNs in predicting bathymetry, we conducted a set of experiments to assess the network's performance under varying training set qualities. In particular, we evaluated the performance of networks trained using high ($M = 100$) and low ($M = 20$) data amounts, considering both complete and incomplete Q measurements for comparison. A realistic scenario entails limits on the number of measurement instruments positioned along the spatial dimension due to cost, but we can generally assume that it is inexpensive to keep each instrument's measurement frequency high. Thus we kept $N = 105$ constant throughout the experiment. Figure 8.17 show the solution predictions of each network. Figures 8.18 and 8.19 shows the predictions for B and B_x respectively. The ground truth line is hidden behind the $M = 100$ (pw/vec) predictions for both B and B_x.

In Figures 8.20 and 8.21 we have plotted the relative L^2 errors for the B, B_x, and Q predictions for each M and training procedure in the experiment. When $M = 100$, the pointwise procedure clearly outperforms the sweeping procedure in the source predictions, even though they admit similarly accurate solution predictions. This is remarkable, as the sweeping procedure essentially starts with only half the information available in the pointwise case, and must recreate this information based

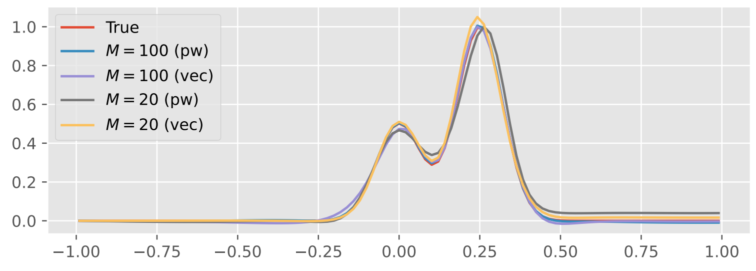


Figure 8.18: Comparison of bathymetry predictions in Section 8.1.3 under different M and training procedures.

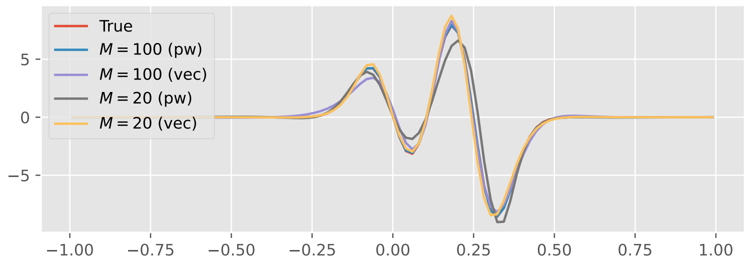


Figure 8.19: Comparison of bathymetry derivatives in Section 8.1.3 under different M and training procedures.

on its own source predictions. Even though we are able to perform this recreation accurately, the sweeping procedure still suffers from a lower source prediction accuracy. In the low data regime ($M = 20$), we see that the pointwise procedure suffers considerably as a result of the decreased data amount, whereas the sweeping procedure is able to perform at roughly the same level as for $M = 100$. Even though the sweeping source prediction is considerably worse for $M = 20$ than for $M = 100$, the source predictions have approximately the same accuracy. Meanwhile, the pointwise solution predictions become about 10 times worse, while the B and B_x predictions become about 100 times worse. It is also worth noting that in all cases, the pointwise procedure was roughly 10 times faster than the sweeping procedure.

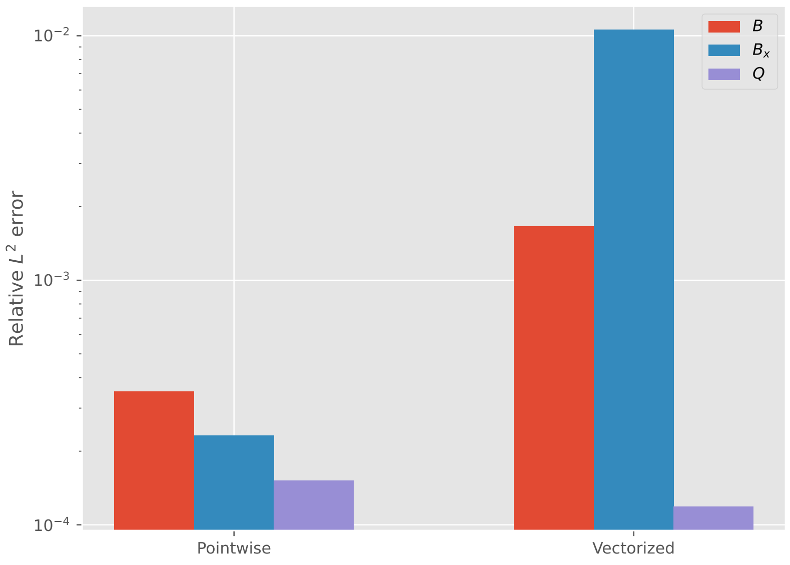


Figure 8.20: Comparison of relative prediction errors in Section 8.1.3 for $M = 100$ sorted by training procedure. The label ”Vectorized” comes from an implementation detail, but it refers to the sweeping algorithm.

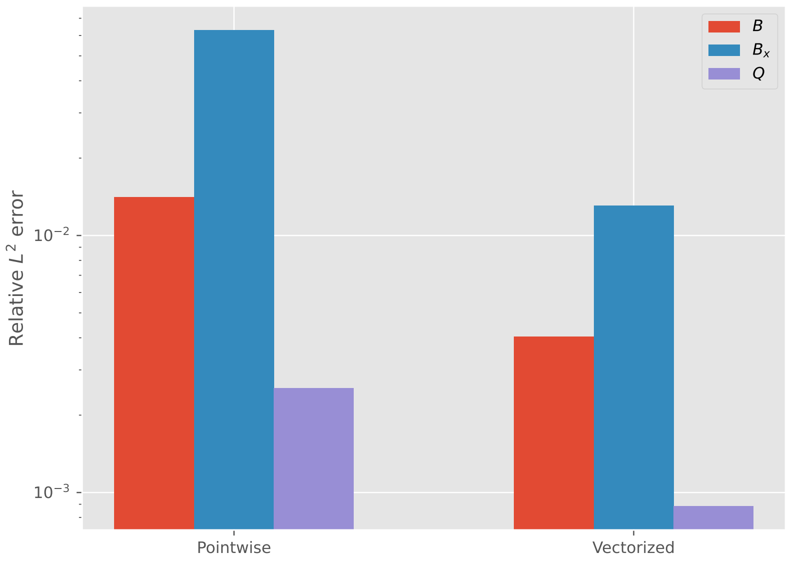


Figure 8.21: Comparison of relative prediction errors in Section 8.1.3 for $M = 20$ sorted by training procedure.

Chapter 9

Discussion

9.1 Summary

Experiments with ODEs, linear and nonlinear scalar hyperbolic equations, as well as the shallow-water hyperbolic system conducted throughout the course of this work suggest that numerics-informed neural networks are able to handle general inverse problems in a wide array of cases. NINNs are shown to provide very accurate approximations when they have access to a large amount of data, and they they are flexible in that their loss function can be adapted in low data regimes to enforce a priori qualitative properties such as periodicity, providing accurate predictions even here. As opposed to PINNs, the adoptation of numerical schemes allow NINNs to simultaneously provide a source and solution prediction to the PDE without an explicit neural network representation of the solution. In the case of hyperbolic systems, this means the training procedure of NINNs can be adapted to handle datasets where only partial information of the solution is available. Experiments with the shallow-water equations indicate that this *sweeping* training procedure adaptation is performant, but considerably slower than its *pointwise* counterpart available in the case of complete datasets.

The incorporation of numerical methods designed to deal with discontinuous solutions into the training procedure make NINNs highly robust with respect to hyperbolic effects such as shock formations. In fact, as the specific choice of numerical method can be decoupled nicely from the training code, NINNs can employ domain specific schemes to handle any type of PDE for which we have efficient numerical methods. In particular, this means that NINNs can easily be integrated with existing simulation codebases that support automatic differentiation. However, this makes NINNs dependent on a mesh over the domain. Meshes can be difficult to generate and computationally expensive in higher dimensional spaces, especially in usecases involving complex geometries. Fortunately, most conservation laws of interest in engineering and physics are formulated in relatively low dimensional spaces, and there exist sophisticated software packages for mesh generation in nontrivial geometries. NINNs are not designed to solve the problems of conventional mesh based methods in higher dimensions, but rather to use these methods to provide accurate solutions to inverse problems in common usecases.

Experiments with Burgers' equation show that NINNs are able to accurately predict non-smooth source functions with a solution dependence. Even though the performance drops in the low data regime, we showed that accurate results could again be obtained by including a priori knowledge about the periodicity of the source into the loss function. Even when the source function depends only indirectly on the function we are interested in learning, specifically on the gradient of the bathymetry in the shallow-water case, the smoothness of neural networks allow us to approximate the bathymetry directly by incorporating network gradients in the loss function.

Even though ReLU activation performs well when only using network outputs directly in the loss function, NINNs perform considerably better with hyperbolic tangent activation when it is necessary to include network gradients in the loss. Otherwise, little to no hyperparameter tuning is needed to obtain good performance. All experiments were conducted with essentially the same network, differing only in activation and input dimension, indicating that the method is generally robust with respect to network architecture. Even so, in cases where optimal performance is vital, the loss function can be used to tune hyperparameters as it gives a relatively direct measure of the performance of the network when we account for the (generally known) truncation error of the numerical method.

9.2 Future work

Even though NINNs show promise when applied to hyperbolic equations, it is important to properly investigate their performance both on more complicated multidimensional hyperbolic systems, and on more general differential operators. In Chapter 8, for instance, more realistic bathymetries could be generated using Perlin noise in order to get a more accurate picture of their performance in real use cases. As all datasets used in this work was generated using the same numerical schemes employed in the NINNs, it is also interesting to see how they perform when applied to either real, noisy data, or to data generated using different schemes.

A proper comparison of NINNs and PINNs is also needed. Specifically, a comparison between the performance of NINNs and the specialized wPINNs and cPINNs. As wPINNs and cPINNs are designed using the same principles as PINNs, but specialized to handle hyperbolic PDEs, they represent interesting challengers to NINNs for inverse problems with such equations.

This work provides a general exploration of the capabilities and limitations of NINNs, and it would be interesting to see how they perform in an operational setting. A possible direction for future work is to provide a reformulation of the training procedure to work in an online setting. This adaptation is essential to handle scenarios involving real time observations. By enabling the network to provide precise and continuously updated approximations of shifting source functions, it would significantly improve the practical utility of NINNs in dynamic environments. Another interesting avenue is to generalize the loss function (4.14) to use more accurate quadrature rules in order to assess the impact of scheme's truncation error on the convergence and performance of NINNs, as discussed in Section 4.4.

Finally, the success of the sweeping training procedure indicates that a NINN could be trained using only auxiliary data. That is, a NINN could be trained without actual observations of the solution as long as we have data on some property of the system obtainable from the solution in a differentiable manner. For instance, it is possible to locate the wavefront of a moving shockwave using only differentiable operations on the solution. Thus, it is theoretically possible to train a NINN using only these location measurements, as we can use the full procedure to generate a solution prediction, and then obtain a prediction of the measured property. An obstacle with this approach is the strong non-uniqueness of solutions possessing these properties, especially in high dimensional systems. To mediate this, it could for instance be paired with an approximate Bayesian analysis of the network parameters using a Monte Carlo approach to obtain a maximum likelihood estimate of the network parameters, and by extension the source function itself. Exploring how this affects the performance of NINNs in practice is an exciting possibility for future research.

Chapter 10

Conclusion

Numerics-informed neural networks are able to perform inverse tasks in hyperbolic PDEs with high precision, even when supplied with limited data quantities. The incorporation of numerical schemes into the network training procedure makes the method robust with respect to shock formation in the solution, and it can even handle non-smooth, solution dependent source functions. It also performs well on hyperbolic systems where the dataset is incomplete. In cases where the gradient of the network with respect to the input is needed in the training process, hyperbolic tangent activation was found to outperform ReLU. Otherwise, the method requires remarkably little hyperparameter tuning in order to produce high quality results.

The fact that the network and training can be decoupled from the numerical simulation in the code means that NINNs can easily be integrated into existing simulator code. The Python/PyTorch stack allows NINNs to interact nicely with preexisting code based on the NumPy interface. This flexibility can allow researchers to use deep learning in underdetermined hyperbolic systems with few implementational difficulties.

Bibliography

[1] Arora, S et al. 'A convergence analysis of gradient descent for deep linear neural networks'. In: *arXiv preprint arXiv:1810.02281* (2018).

[2] benwaad. *benwaad/ninn: Thesis release*. Version v1.0. June 2023. DOI: 10.5281/zenodo.8066384. URL: https://doi.org/10.5281/zenodo.8066384.

[3] Cai, S et al. 'Physics-informed neural networks (PINNs) for fluid mechanics: A review'. In: *Acta Mechanica Sinica* 37.12 (2021), pp. 1727–1738.

[4] Colton, DL, Ewing, RE, Rundell, W et al. *Inverse problems in partial differential equations*. Vol. 42. Siam, 1990.

[5] Cuomo, S et al. 'Scientific machine learning through physics–informed neural networks: where we are and what's next'. In: *Journal of Scientific Computing* 92.3 (2022), p. 88.

[6] De Ryck, T, Mishra, S and Molinaro, R. 'wPINNs: Weak Physics informed neural networks for approximating entropy solutions of hyperbolic conservation laws'. In: *arXiv preprint arXiv:2207.08483* (2022).

[7] Eidnes, S and Lye, KO. 'Pseudo-Hamiltonian neural networks for learning partial differential equations'. In: *arXiv preprint arXiv:2304.14374* (2023).

[8] Fjordholm, US and Lye, KO. 'Convergence rates of monotone schemes for conservation laws for data with unbounded total variation'. In: *Journal of Scientific Computing* 91.2 (2022), p. 32.

[9] Funahashi, KI. 'On the approximate realization of continuous mappings by neural networks'. In: *Neural networks* 2.3 (1989), pp. 183–192.

[10] Godunov, SK. 'A difference method for the numerical computation of discontinuous solutions of hydrodynamic equations'. In: *Math. Sbornik* (1959). URL: https://cir.nii.ac.jp/crid/1573668924626753536.

[11] Goodfellow, I, Bengio, Y and Courville, A. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] Gu, J et al. 'Recent advances in convolutional neural networks'. In: *Pattern recognition* 77 (2018), pp. 354–377.

[13] Hagen, TR et al. 'How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine'. In: *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF* (2007), pp. 211–264.

[14] Harris, CR et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[15] Holden, H and Risebro, NH. *Front tracking for hyperbolic conservation laws*. 2nd ed. Springer, 2015.

[16] Hornik, K. 'Approximation capabilities of multilayer feedforward networks'. In: *Neural networks* 4.2 (1991), pp. 251–257.

[17] Hornik, K, Stinchcombe, M and White, H. 'Multilayer feedforward networks are universal approximators'. In: *Neural networks* 2.5 (1989), pp. 359–366.

[18] Hornik, K, Stinchcombe, M and White, H. 'Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks'. In: *Neural networks* 3.5 (1990), pp. 551–560.

[19] Isakov, V. *Inverse problems for partial differential equations.* Vol. 127. Springer, 2006.

[20] Jagtap, AD, Kharazmi, E and Karniadakis, GE. 'Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems'. In: *Computer Methods in Applied Mechanics and Engineering* 365 (2020), p. 113028.

[21] Kac, M. 'Can one hear the shape of a drum?' In: *The american mathematical monthly* 73.4P2 (1966), pp. 1–23.

[22] Kingma, DP and Ba, J. 'Adam: A method for stochastic optimization'. In: *arXiv preprint arXiv:1412.6980* (2014).

[23] Köppen, M. 'The curse of dimensionality'. In: *5th online world conference on soft computing in industrial applications (WSC5).* Vol. 1. 2000, pp. 4–8.

[24] Kurganov, A, Noelle, S and Petrova, G. 'Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations'. In: *SIAM Journal on Scientific Computing* 23.3 (2001), pp. 707–740.

[25] Lee, JD et al. 'Gradient descent only converges to minimizers'. In: *Conference on learning theory.* PMLR. 2016, pp. 1246–1257.

[26] Leshno, M et al. 'Multilayer feedforward networks with a nonpolynomial activation function can approximate any function'. In: *Neural networks* 6.6 (1993), pp. 861–867.

[27] LeVeque, RJ. *Numerical methods for conservation laws.* 2nd ed. Springer, 1992.

[28] Li, Y et al. 'Contextual transformer networks for visual recognition'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[29] Martín, A et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[30] Mishra, S, Fjordholm, US and Abgrall, R. *Numerical methods for conservation laws and related equations.* https://www.uio.no/studier/emner/matnat/math/MAT-IN9240/h17/pensumliste/numcl_notes.pdf. Online, accessed 12 May 2023.

[31] Murty, KG and Kabadi, SN. *Some NP-complete problems in quadratic and nonlinear programming.* Tech. rep. 1985.

[32] Paszke, A et al. 'Automatic differentiation in PyTorch'. In: (2017).

[33] Pinkus, A. 'Approximation theory of the MLP model in neural networks'. In: *Acta Numerica* 8 (1999), pp. 143–195. DOI: 10.1017/S0962492900002919.

[34] Raissi, M, Perdikaris, P and Karniadakis, GE. 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations'. In: *Journal of Computational physics* 378 (2019), pp. 686–707.

[35] Rosenblatt, F. 'The perceptron: a probabilistic model for information storage and organization in the brain.' In: *Psychological review* 65.6 (1958), p. 386.

[36] Ruder, S. 'An overview of gradient descent optimization algorithms'. In: *arXiv preprint arXiv:1609.04747* (2016).

[37] Rumelhart, DE et al. 'Backpropagation: The basic theory'. In: *Backpropagation: Theory, architectures and applications* (1995), pp. 1–34.

[38] Salehinejad, H et al. 'Recent advances in recurrent neural networks'. In: *arXiv preprint arXiv:1801.01078* (2017).

[39] Sharma, S, Sharma, S and Athaiya, A. 'Activation functions in neural networks'. In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[40] Toro, EF. *Numerical Methods for Fluid Dynamics: A Practical Introduction.* 3rd ed. Springer, 1999.

[41] Vaswani, A et al. 'Attention is all you need'. In: *Advances in neural information processing systems* 30 (2017).