Diffusion-free neural network-based algorithm for hyperbolic equations

Xiao Yang

A thesis submitted to the Faculty of Graduate and Post-doctoral Affairs
in partial fulfilment of the requirements for the degree of

Master of Mathematics

Carleton University
Ottawa, Ontario

@ 2022
Abstract

In this thesis, we propose a new neural network based algorithm for solving one-dimensional hyperbolic equations. We study the algorithm performance on a Riemann problem and compared it to a traditional Godunov solver to illustrate its accuracy.
Acknowledgements

This thesis serves as a requirement for Master of Science degree in Mathematics from Carleton University. The thesis idea is jointly proposed and supervised by professor Arian Novruz from University of Ottawa, Department of Mathematics and Emmanuel Lorin from Carleton University, Department of Mathematics.
Glossary

AD: Automatic Differentiation

Adam: Derived from Adaptive Moment Estimation

AdaGrad: Adaptive Gradient Algorithm

AdaDelta: An improved version of AdaGrad

ANN: Artificial Neuron Network

BFGS: Broyden-Fletcher-Goldfarb-Shanno algorithm

BVP: Boundary Value Problems

CD: Coordinate Descent

CL: Conservation Law

Collocation Points: A number of points chosen inside the constructed geometric domain

IBVP: Initial Boundary Value Problem

L-BFGS: Limited Broyden-Fletcher-Goldfarb-Shanno algorithm

SGD: Stochastic Gradient Descent

PINN: Physics Informed Neural Network
Contents

Abstract ii
Acknowledgements iii
Glossary iv

1 Introduction 1

2 1D Hyperbolic Equations of Conservation Laws 5
  2.1 Hyperbolic First-Order PDEs 5
     2.1.1 Conservation Law 6
     2.1.2 The Method of Characteristics 7
  2.2 Principles for Solving PDEs 9
  2.3 Weak Solutions to Hyperbolic PDEs 12

3 Physics Informed Neural Network (PINN) 17
  3.1 Neural Networks 18
  3.2 PINN Theory For Forward Problems 27
  3.3 Optimization Algorithms 32
## CONTENTS

3.3.1 First Order Methods ........................................ 33
3.3.2 Second order methods: Quasi-Newton Methods ............ 45
3.3.3 Derivative-Free Optimization ................................ 49
3.4 PINN Examples of Forward Problems .......................... 51
  3.4.1 ODE: Population Growth Example .......................... 51
  3.4.2 PDE: Schrödinger’s Equation ............................. 54
  3.4.3 PDE: 1D Burgers’ Equation ............................... 57

4 A Non-Diffusive PINN-based Algorithm .......................... 62
  4.1 Background .................................................. 62
  4.2 Non-Diffusive Algorithm ..................................... 63
  4.3 Riemann Example: Capturing Shock Waves .................... 67
    4.3.1 Analytic Solution ....................................... 68
    4.3.2 Traditional Numerical Solution: Godunov Solver ....... 69
    4.3.3 Numerical Experiments .................................. 71
  4.4 Discussion .................................................. 74

5 Numerical Computation ........................................... 76
  5.1 Existing Libraries: Computational Aspects .................... 76
  5.2 Practical Implementation for Hyperbolic Equations .......... 78

Bibliography ...................................................... 80

A Python code repository ........................................ 86

B PINN library documentation .................................... 87
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>1 input + 1 neuron</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>4 inputs + 1 neuron</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>A 1D four-layer network example</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>A four-layer network</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>A five-layer network</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>ODE PINN-solution ($R = 1$)</td>
<td>53</td>
</tr>
<tr>
<td>3.7</td>
<td>Schrödinger’s equation PINN</td>
<td>56</td>
</tr>
<tr>
<td>3.8</td>
<td>Solution to problem with sin initial condition (left); Plot of loss value to number of iterations with Adam optimizer (right)</td>
<td>58</td>
</tr>
<tr>
<td>3.9</td>
<td>Solution to problem with sin initial condition at final time (left); Plot of loss value to number of iterations with L-BFGS optimizer (right)</td>
<td>58</td>
</tr>
<tr>
<td>3.10</td>
<td>Attempt 1: Solution to problem with initial condition $g(x)$ (left); Plot of loss value to number of iterations with Adam optimizer (right)</td>
<td>60</td>
</tr>
</tbody>
</table>
3.11 Attempt 2: Solution to problem with $g(x)$ initial condition (left); Plot of loss value to number of iterations with L-BFGS optimizer (right). ................................................................. 60

4.1 (left) Initial Condition $u_0(x)$; (right) Characteristic Lines . . . . . 68

4.2 Experiment 1. (top left) Solution of reference. (top right) Direct PINN solution. (bottom) Neural network solution. ................. 72

4.3 Experiment 1. Loss function. ............................................. 73

4.4 Experiment 2. Loss function. ............................................. 74

4.5 Experiment 2. Space-time solution. ................................. 75

4.6 Experiment 2. Godunov at CFL=0.99 and neural network solution at time $T = 0.5$. .......................... 75
Chapter 1

Introduction

Partial differential equations play a crucial role in many fields of engineering and physics. For example, the Navier Stokes equations model the turbulence of air and water flow, and it can be applied for the design of aircraft as well as weather prediction. The heat equation models how heat diffuses through a given region, and is used in geophysics for instance [7]. Despite the wide applications of PDEs, solving PDEs is usually not an easy task. As we know, most PDEs do not have analytical solutions, and numerical algorithms have been developed to solve them.

The finite difference method (FDM) and finite element method (FEM) are among the most frequently used methods to solve PDEs. We summarize these two major classical methods FEM [32], [39] and FDM [38] for comparison purpose. Finite difference method consists of:

• Choose a set of points in the domain, called mesh grid, where we will approximate the solution.

• Approximate the partial derivatives in the PDE using finite differences of
appropriate order at all mesh grid points.

- Substitute the finite differences into the PDE to form a list of algebraic equations.

- Solve the algebraic equation system, including the boundary conditions.

On the other hand, the finite element method divides complicated domain into finite elements and approximates the variational form of the PDE.

- Construct geometry and divide the domain into finite elements.

- Obtain finite-dimensional version of variational form and establish equation at the vertices of the mesh.

- Solve the linear system equation for the whole domain.

These traditional methods are deeply understood, but they have limitations such as high computational complexity when tackling high-dimensional problems. It was not until the 2010s [16], the CRUNCH group at Brown University proposed a new technique based on neural networks, which can avoid large matrix computations and perform well on problems with high spatial dimensions.

The theory of neural networks date back to 1989, when Kurt Hornik, Maxwell Stinchcombe, and Halbert White proved [12] that the multi-layer feed forward networks are capable of approximating any Borel measurable function to any desired degree of accuracy. In another word, neural network is a universal estimator to any continuous functions.

Nowadays, neural network techniques have been flourishing in computer vision and natural language processing since deep learning packages are available in
early 2010s. As a universal estimator, neural networks’ application does not only lie in computer science and statistical learning, but also in mathematics. Physics informed neural networks (PINN) emerged as an alternative to traditional approximation methods to solve differential equations. In fact, its performance on PDEs with smooth solution is promising [5]. However, this method remains unsound in certain circumstances, [28] [5] and can not fully replace existing classical methods to solve a the given problem on its own.

Typically, a PDE is solved using a neural network by minimizing the so-called loss function, which measures norm residue of the PDE in a number of unstructured points, called learning nodes/points. The neural network which approximates the solution is defined by a number of parameters which minimize the loss function. Although there are many results [15], [24], demonstrating the power of this approach, there are still a number of difficulties [19] associated to this approach in comparison to the classical methods.

There are many reasons why PINN under-performs [16], [19]. One of the major reasons is that regular PINNs directly plugs into the neural network to solve the optimization problem. This solution is not the same as the weak solution given by the variational form of the PDE. Hence, the regular PINN solution is not correct. If a PDE does not have smooth solutions, then the optimization problem associated to the residue of the PDE is strongly non-linear in terms of neural network parameters. Hence, the solution of a global minimizer is compromised. Also, PINN does not perform well in stiff problems [28].

There is an insufficient amount of non-regular PINN based algorithms that are specifically designed for discontinuous problems. As a natural attempt to improve
PINN’s performance in discontinuous problems, this thesis is a contribution to the solution of first-order hyperbolic PDEs. We develop a new neural network-based method for solving weak solutions to these equations. Our method is based on an original view of the interaction of the characteristics meeting on a discontinuity curve and the Rankine-Hugoniot condition.

The structure of this thesis is as follows. First, we present hyperbolic PDE’s solution theory, then discuss the classical Physics informed neural network method and summarize commonly used optimization algorithms, with common equations solved via the standard PINN as examples. To check the precision of our method, we then study a Riemann problem, which has analytic solution. We use newly proposed PINN-based algorithm to capture weak solutions then compare the result to that of a traditional Godunov solver, which is based on finite volume/difference method. Finally, we discuss the numerical computation of PINN in practical aspect, summarizing current computational tools and comment on specific implementation issues for the proposed algorithm.
Chapter 2

1D Hyperbolic Equations of Conservation Laws

Recall that the standard Physics informed neural network algorithms performed well when solving well-posed initial value problems that has classical smooth solutions [25]. However, when the solution is discontinuous, such as a simple step function, regular PINNs do not work properly anymore. We introduce hyperbolic equation of conservation laws to understand this situation when the method of characteristics fails.

2.1 Hyperbolic First-Order PDEs

Let us begin with hyperbolic PDEs. We frequently come across three-types of second-order PDEs, which are classified as elliptic, hyperbolic, and parabolic. The symbol of the operator allows to classify all kinds of PDEs.

In particular, hyperbolic PDEs describe wave propagation phenomena, and the
smoothness of their solution depend on the smoothness of the initial and boundary conditions. For instance, if there is a jump at the start or at the boundaries, then the jump will propagate as a discontinuity in the solution. Specifically, if there is a disturbance in the initial condition, not every point in the domain reacts to the disturbance at the same time. We are particularly interested in equations with non-simultaneous propagation, because the time-stepping brings challenge to the performance of numerical methods.

2.1.1 Conservation Law

Let us now focus on hyperbolic equations of conservation laws [27]. Consider a particular type of conservation law (CL):

\[
    u_t = -q(u)_x, \quad u(x, 0) = g(x). \tag{2.1.1}
\]

with \( x \in \mathbb{R}, t > 0 \) and \( g : \mathbb{R} \to \mathbb{R} \) is a sufficiently smooth given function. \( q \in C(D) \) for some open set \( D \subseteq \mathbb{R}^2 \). From a modelling point of view, function \( u \) represents the density or the concentration of quantity \( Q \), with \( q(u) \) be its flux function. The proposed equation provides a link between the density and flux, thus expressing a conservation law.

To elaborate, the amount of flux \( Q \) is given by \( \int_{x_1}^{x_2} u(x, t) dx \). The conservation law states the rate of change of \( Q \) in \([x_1, x_2]\) is determined by the net flux through the end points of the interval, See [27]. The flux is \( q(u) \), and the conservation law translates to

\[
    \int_{x_1}^{x_2} u_t(x, t) dx = q(u(x_1, t)) - q(u(x_2, t)),
\]
CHAPTER 2. 1D HYPERBOLIC EQUATIONS OF CONSERVATION LAWS

\[ \int_{x_1}^{x_2} [u_t(x,t) + q(u(x,t))]_x dx = 0. \]

If the flux \( q(u) = vu \), \( v \) is a constant, then the model equation becomes a transport equation. If \( q(u) = u^2/2 \), (2.1.1) is called inviscid Burgers' equation, which is a first-order quasi-linear hyperbolic equation, see [27].

2.1.2 The Method of Characteristics

Let us denote \((x_0,0)\) an arbitrary point on the \((x,t)\) plane, with \( x \in \mathbb{R}, t \in \mathbb{R}^+ \) and \( q \) is smooth. In Equation (2.1.1), we take derivative and compare with the original equation, we have

\[
\frac{d}{dt} u(x(t), t) = u_x(x(t), t)x'(t) + u_t(x(t), t) = 0,
\]

\[ u_t(x(t), t) + q'(g(x_0))u_x(x(t), t) = 0. \]

Hence,

\[ x'(t) = q'(g(x_0)), \]

\[ x(t) = q'(g(x_0))t + x_0, x(0) = x_0. \]

We have obtained a family of straight lines on which the solution to equation (2.1.1) is constant. We define the characteristic lines as follow, and present a linear example.

**Definition 2.1 (Characteristics)** The straight lines \( x : t \mapsto q'(g(x_0))t + x_0 \) are called characteristic lines, where \( x_0 \in \mathbb{R} \) is an arbitrary constant and \( t \in \mathbb{R}^+ \).

**Example 2.1 (Linear Transport Equation)** Consider a linear transport equa-
CHAPTER 2. 1D HYPERBOLIC EQUATIONS OF CONSERVATION LAWS

...tion that models the pollution in a channel [27], a simpler form of conservation law when \( q(u) = vu \). Let \( u \) be the concentration of pollutant, \( v > 0 \) is a constant representing the stream speed, with \( x > 0, t > 0, g \in C^1(\mathbb{R}) \). The pure transport equation is given by

\[
    u_t + vu_x = 0, \quad u(x,0) = g(x). \tag{2.1.2}
\]

Setting \( v = (v,1)^T \), the transport equation can be written as \( u_t + vu_x = \nabla u \cdot v \). \( \nabla u \) is orthogonal to level curves of \( u \), and level curves of \( u \) are parallel to \( v \). This fact inspired the characteristic method to determine the value of \( u \) at any given point. Indeed, rewrite \( u \) as \( w(t) = u(x(t),t) = u(vt + x_0, t) \), as \( w'(t) = vu_x(x(t),t) + u_t(x(t),t) = vu_x(vt + x_0, t) + u_t(vt + x_0, t) = 0 \), we can conclude \( w(t) = u(x(t),t) \) is a constant along the group of characteristic lines \( x(t) = vt + x_0 \) for an arbitrary \( x_0 \).

In order to find the value of \( u \) at any given point \( (x^*, t^*) \) in the domain, we can simply find its value at the initial point \( (x_0, 0) \), or any point that \( u \)'s value is given as an initial condition, as the value of \( u(x^*, t^*) \) is the same as \( u(x_0, 0) \) if \( (x_0, 0) \) and \( (x^*, t^*) \) are connected by a characteristic line i.e. \( u(x^*, t^*) = u(x_0, 0) = g(x_0) = g(x^* - vt^*) \). We conclude that the solution to the initial value problem is given by \( u(x, t) = g(x - vt) \), representing a travelling wave.

Example 2.2 (1D Burgers’ Equation) Consider the case when \( q(u) = u^2/2 \), the conservation law initial value problem becomes

\[
    u_t + uu_x = 0, \quad u(x,0) = g(x). \tag{2.1.3}
\]
The method of characteristics gives solution \( u(x,t) = g(x - ut) \) when i) the characteristics do not intersect, and ii) there is no region without any characteristics. For instance, if we have a discontinuous \( g \), such as the step function, for some \( (x^*,t^*) \), it could belong to more than one characteristic curves.

### 2.2 Principles for Solving PDEs

Generally speaking, PDEs need to satisfy boundary conditions (BC). Suppose the PDE is defined on the domain \( \Omega \), standard BCs are

- **Dirichlet BC**: the solution value is specified on boundary, i.e. given a function \( f: \partial \Omega \to \mathbb{R} \), we require \( u(x) = f(x) \), \( x \in \partial \Omega \), the boundary of the domain \( \Omega \).

- **Neumann BC**: normal derivative of solution is specified on boundary, i.e. given a function \( f: \partial \Omega \to \mathbb{R} \), we require \( (\partial u/\partial n)(x) = f(x) \), \( x \in \partial \Omega \), where, \( \partial u/\partial n = \nabla u(x) \cdot n(x) \), and \( n(x) \) is the unit outward normal at \( x \in \partial \Omega \).

- **Robin BC**: a linear combination of the solution and its normal derivative is specified on boundary.

Further, we say a given problem for a PDE is well-posed, see [7] if

- The problem has a solution (existence),

- The solution is unique (uniqueness),

- The solution depends continuously on initial and boundary conditions given in the problem (stability).
We address the well-posed principle by providing case by case hyperbolic equation examples.

**Example 2.3** 1D wave equation is an example of hyperbolic equation that satisfies Definition 2.1. Let $u$ be a two dimensional function in domain $\Omega = (-\infty, \infty) \times (0, \infty)$, where $u$ models a travelling wave. We define the problem as follow,

$$u_{tt} - c^2 u_{xx} = 0, \quad u(x, 0) = g(x), \quad u_t(x, 0) = h(x).$$  \hspace{1cm} (2.2.1)

For this wave equation, if $g$ is not two times continuously differentiable, then no classic solution to the problem exists in the classical sense, and the problem is ill-posed. If $g \in C^2$ and $h \in C^1$, then the problem is well-posed and its solution is given by d’Alembert’s formula [7],

$$u(x,t) = F(x+ct) + G(x-ct), \quad (x,t) \in \Omega,$$

where $F$ and $G$ are $C^2$-functions, which satisfy

$$F(x) + G(x) = g(x),$$

$$cF'(x) - cG'(x) = h(x).$$

Integrating $cF'(x) - cG'(x) = h(x)$, we obtain $cF(x) - cG(x) = \int_0^x h(\xi)d\xi + C_1$. Solving for $F$ and $G$ leads to

$$F(x) = \frac{1}{2} g(x) + \frac{1}{2c} \int_0^x h(\xi)d\xi + C_1,$$

$$G(x) = \frac{1}{2} g(x) - \frac{1}{2c} \int_0^x h(\xi)d\xi - C_1,$$
CHAPTER 2. 1D HYPERBOLIC EQUATIONS OF CONSERVATION LAWS

where \( C_1 \) is an arbitrary real constant. Hence, the solution is given by

\[
u(x,t) = \frac{1}{2}(g(x-ct) + g(x+ct)) + \frac{1}{2c} \int_{x-ct}^{x+ct} h(\xi) d\xi, \quad (x,t) \in \Omega.
\]

Example 2.4 Consider an inviscid Burger’s equation on \( \Omega = (-\infty, \infty) \times (0, \infty) \), with \( \alpha, \beta \in \mathbb{R} \).

\[
u_t + \nu u_x = 0, \quad \nu(x,0) = g(x) = \alpha x + \beta.
\]

The corresponding characteristics are defined as solution to

\[
\frac{du}{dt} = u_x \frac{dx}{dt} + u_t, \quad \frac{dx}{dt} = u(x,t) = u(\xi,0) = g(\xi), \quad x(t) = g(\xi)t + \xi,
\]

where \( \xi \) is an arbitrary constant. Since solution is constant along the characteristics, \( \nu \) is given by \( \nu(x,t) = u(\xi,0) = g(\xi) = g(x-g(\xi)t) = \alpha(x-ut) + \beta \). Solve for \( \nu \) explicitly, we have \( \nu(x,t) = (\alpha x + \beta)/(\alpha t + 1) \).

If \( \alpha = 1, \beta = 0 \), we have \( \nu(x,t) = x/(t+1) \). As the characteristics cover the entire \((x,t)\) plane, and we have unique solution. The problem is well-posed in this case. In contrast, if \( \alpha = -1, \beta = 0 \), then

\[
u(x,0) = g(x) = -x, \quad \nu(x,t) = -(x-ut) = \frac{x}{t-1}, \quad x(t) = -\xi t + \xi.
\]

We notice that when \( t = 1 \), the characteristics intersect, and the solution is multi-valued. The well-posed principle does not apply anymore. Similarly, if \( g(x) = \sin x \), there are characteristics on \((x,t)\) plane that also intersect. If the solution is not unique or not differentiable, the PDE is ill-posed in the classical sense.
Both cases show the necessity to extend the solutions into the class of non-smooth functions, which motivates the introduction of weak solution.

2.3 Weak Solutions to Hyperbolic PDEs

We noticed that smooth initial data of hyperbolic conservation laws can blow up, leading to discontinuities in finite time. In fact, there are many types of physical waves that are discontinuous in nature, and can not be predicted by linear wave equations. When method of characteristics fails, such as in Example 2.5, a weak solution, or integral solution, to a hyperbolic equation of conservation law is defined to extend the notion of solution.

Definition 2.2 (Weak Solution) Let $\mathbb{R}^{2+}$ denote the upper-half plane, a weak solution to (2.1.1) is a function $u : (x, t) \in \mathbb{R}^{2+} \mapsto \mathbb{R}$ such that [26]

$$
\int_{-\infty}^{\infty} \int_{0}^{\infty} \left[ u(x, t) \phi_t(x, t) + q(u(x, t)) \phi_x(x, t) \right] dx dt + \int_{-\infty}^{\infty} g(x) \phi(x, 0) dx = 0
$$

for any $\phi \in C^1_0(\mathbb{R}^{2+})$, where $C^1_0(\mathbb{R}^{2+}) = \{ \phi \in C^1_0(\mathbb{R}^{2+}): \text{support of } \phi \subset B_r((0, 0)) \cap \mathbb{R}^{2+} \}$, and where $r \in \mathbb{R}^+$ is the radius of open ball $B_r(0, 0)$ centered at $(0, 0)$.

The idea is to carry the derivatives onto the test function $\phi$ via an integration by parts [27]. We have thus obtained an integral equation, which is valid for every test function $\phi$, without any classical derivatives on $u$. The solution $u$ can be discontinuous. Such a solution is considered non-classical, and is called a weak solution. Now, after we defined a weak solution, we can find a necessary condition for a discontinuous, piece-wise smooth weak solution.
CHAPTER 2. 1D HYPERBOLIC EQUATIONS OF CONSERVATION LAWS

Theorem 2.1 (Rankine-Hugoniot) [26] Let $N$ be an open neighborhood in open upper-half plane, and suppose parametrization of the curve $C : (\alpha, \beta) \ni t \to x(t)$ divides the open neighborhood $N$ into two parts: $N_l$ and $N_r$. Let $u$ be a weak solution of $u_t + q(u)_x = 0$ such that

- $u$ is a classical solution of $u_t + q(u)_x = 0$ in both $N_l, N_r$.
- $u$ undergoes a jump discontinuity $[[u]]$ at curve $C$.
- The jump $[[u]]$ is continuous along $C$.

For any point $p = (x, t) \in C$, let $s = x'(t)$ be slope of $C$ at $p$. Then the following relations hold along $C$:

$$s[[u]] = [[q(u)]] = q(u_r(p)) - q(u_l(p)), \quad [[u(p)]] = u_r(p) - u_l(p).$$

The proof of Rankine-Hugoniot’s jump condition is given by [7]. This necessary condition combined with method of characteristics allow us to construct weak solutions known as shock waves.

Example 2.5 Consider the following initial value problem,

$$u_t + \left(\frac{u^2}{2}\right)_x = 0, \quad u(x, 0) = g(x),$$

$$g(x) = \begin{cases} 
0, & x < 0, \\
1, & x > 0.
\end{cases}$$

The characteristics are straight lines: $x : t \mapsto g(x_0)t + x_0, \; x_0 \in \mathbb{R}, \; t \in \mathbb{R}^+$, and
the solution is defined as

$$u(x, t) = \begin{cases} 
0, & x < 0, \\
1, & x > t.
\end{cases}$$

The region \(\{(x, t) : 0 < x < t\}\) is not covered by characteristics. The rarefaction solution is defined as

$$u(x, t) = \begin{cases} 
0, & x \leq 0, \\
x/t, & 0 < x < t, \\
1, & x \geq t.
\end{cases}$$

There also exists a shock wave solution obtained thanks to the Rankine Hugoniot condition. Since \(u_l = 0, u_r = 1, f(u) = u^2/2, f(u_l) = 0, f(u_r) = 1/2, s'(t) = 1/2, s(0) = 0\), a shock starts at 0, a straight line of discontinuities. \(x(t) = t/2, t > 0\). Therefore, the shock wave solution is defined as

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

From the abundance of characteristic curves, there are a huge number of solutions, but based on the physics associated, we often only expect a unique solution. A selection criteria to pick the most physically reasonable solution among all weak solutions is hence needed. The most widely used criteria is called an entropy condition. Recall that in Problem (2.1.1), \(G(x, t, u)\) gives the classical solution
for small times, where

\[ G(x, t, u) = u - g(x - q'(u)t) = 0. \]

If we assume \( g'(x) > 0, q''(u) > C > 0 \), where \( C \) is a constant, then by the implicit function theorem, we get

\[ u_x = \frac{-G_x}{G_u} = \frac{g'}{1 + tg'q''}. \]

where \( g \) is evaluated at \( x - q'(u)t \), and \( u_x \leq E/t \), a constant \( E = 1/C \), \( t \in \mathbb{R}^+ \).

**Definition 2.3 (Entropy Condition)** If there exists \( E \geq 0 \) such that for every \( x, z \in \mathbb{R}, t > 0 \), the solution satisfies \( u(x+z,t) - u(x,t) \leq \frac{E}{t}z \), then we say weak solution \( u \) satisfies the entropy condition [7].

**Definition 2.4 (Lax Entropy Condition)** If the slope of the discontinuity curve satisfies \( q'(u_l) > x'(t) > q'(u_r) \), then we say weak solution \( u \) satisfies the Lax entropy condition [7].

**Theorem 2.2** If \( q \in C^2(\mathbb{R}) \), \( q \) is strictly convex or concave and \( g \) is bounded, then there exists a unique solution to Problem (2.1.1) that satisfies the entropy condition [27].

In Problem (2.1.1), if the initial condition is of the form

\[ g(x) = \begin{cases} 
  u_+, & x > 0, \\
  u_-, & x < 0, 
\end{cases} \]
then we call problem (2.1.1) a Riemann problem, where $u_+$ and $u_-$ are unequal constants. The solution to Riemann problem can be summarized as follows.

**Theorem 2.3 (Riemann Problem)** Let $q \in C^2(\mathbb{R})$ be strictly convex, and $q'' > 0$, then

1. If $u_+ < u_-$, the unique entropy solution is a shock wave, defined as

   $$u(x,t) = \begin{cases} 
   u_+ , & \frac{x}{t} > s'(t), \\
   u_- , & \frac{x}{t} < s'(t),
   \end{cases}$$

   where $s$ denotes the shock velocity, such that

   $$s'(t) = \frac{q(u_+) - q(u_-)}{u_+ - u_-}.$$

2. If $u_+ > u_-$, the unique entropy solution is a rarefaction wave, defined as

   $$u(x,t) = \begin{cases} 
   u_- , & \frac{x}{t} < s'(t), \\
   (q')^{-1}(\frac{x}{t}), & q'(u_-) < \frac{x}{t} < q'(u_+), \\
   u_+ , & \frac{x}{t} > s'(t),
   \end{cases}$$

Riemann problems serve as a good numeric examples to perform experiments with PINN, See [27].
Chapter 3

Physics Informed Neural Network (PINN)

Many numerical methods have been developed for solving differential equations. For instance, finite difference methods produce a solution in the form of an array that contains the value of the solution at selected mesh grid points. Spectral methods use basis-functions to represent the solution in analytic form and transform the original problem to a system of algebraic equations. Existing numerical methods often rely on the solution to systems of algebraic equations, which result from the discretization of the domain, i.e. the usage of mesh.

However, solving a large matrix system can encounter many computational issues especially when the matrix is not sparse or the dimension is too high [14]. These facts motivate to seek for alternatives that can avoid disadvantages induced by matrix computations, such as PINN method in late 2010s [5].

The idea of using artificial neural networks to solve ordinary differential equa-
tions is around as old as 1998, when Lagaris [20] proposed a neural network based method using trial solution. This method is straightforward in theory, and has the desired advantage of avoiding mesh usage and matrix computation. The resulted solution is a feed-forward neural network, which is differentiable and in analytic closed form. We start to illustrate the mathematical idea behind the PINN method and demonstrate a few practical examples in this chapter. First, let us begin with an introduction to neural networks, for the general audience.

3.1 Neural Networks

We formally denote by \( x^{(i)} \) the input vector, also known as an example. If \( x^{(i)} \) has \( d \) dimensions, we denote by \( x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}) \), \( d \in \mathbb{N}^+ \). Index \( i \) represents \( i \) th example in the data set which contains \( n \) points, \( n \in \mathbb{N}^+ \). In machine learning, we use the examples (training points) to perform different tasks. See remark 3.5.

In a two-dimensional non-time-dependent PDE, we denote the \( i \) th training point by \( \{ (x_1^{(i)}, x_2^{(i)}) \} \), \( i \in \mathbb{N} \) in the \( x,y \) plane. If the PDE is time-dependent or has higher dimensions, we simply treat the time variable as another feature, same as spatial variable \( x \) and \( y \). In the three-dimensional case, on the \( x,y,z \) plane, the \( i \) th training point is denoted by \( (x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) \) and is \( (x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, x_4^{(i)}) \) for an evolution equation. We start with the simplest case when \( x \in \mathbb{R} \).

**Definition 3.1 (Neuron)** Let \( x \) be a vector in \( \mathbb{R}^d \), \( W \) be a vector in \( \mathbb{R}^{(1 \times d)} \), \( b \) be a vector in \( \mathbb{R} \), and \( \sigma : \mathbb{R} \rightarrow \mathbb{R} \) be a non-linear function. A neuron is a function defined as \( \gamma : x \mapsto \sigma(Wx + b) \).
Example 3.1 (Single Neuron) Suppose the input variable $x \in \mathbb{R}$, and parameter $W \in \mathbb{R}$, $b \in \mathbb{R}$. The left circle in Figure 3.1 represents $x$ as the input variable, while the right circle is a single neuron $\sigma(Wx + b)$, where $\sigma: \mathbb{R} \to \mathbb{R}$ is the non-linear sigmoid function. Suppose now $x \in \mathbb{R}^d$, $d = 4$, and $W \in \mathbb{R}^{1 \times d}$, $b \in \mathbb{R}$, the

![Diagram of a single neuron with 1 input and 1 output]

Figure 3.1: 1 input + 1 neuron.

The neuron on the right part of Figure 3.2 is defined as $\sigma(Wx + b)$, where $\sigma: \mathbb{R} \to \mathbb{R}$ is the nonlinear sigmoid function.

![Diagram of a neuron with 4 inputs and 1 output]

Figure 3.2: 4 inputs + 1 neuron.
Remark 3.1 (First layer (input layer)) We can also call the circles representing input variables neurons. If the input variable \( x = (x_1, x_2, \ldots, x_d) \) has dimension \( d \), we say \( d = n_1 \), the number of neurons at the first layer (input layer). Generally, there are \( n_j \) neurons at each layer, \( j = 1, 2, \ldots, L \), \( L \) is a constant representing the total number of layers.

Definition 3.2 (Neural Network) A neural network is a repeated composition of linear and nonlinear functions, where the nonlinear function is called activation function. Let \( x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d \), \( d = n_1 \). Suppose there are \( L \) layers (compositions) in the neural network, where \( L \in \mathbb{N}^+ \) is called number of layers. Each layer has its corresponding parameters, denoted by \( W[j] = (W_1^{[j]}, W_2^{[j]}, \ldots, W_{n_j}^{[j]}) \), and \( b[j] = (b_1^{[j]}, b_2^{[j]}, \ldots, b_{n_j}^{[j]}) \). We define the composition at each layer as

\[
\gamma_j : x \mapsto \sigma(W^{[j]}x + b^{[j]}), \quad j = 2, \ldots, L,
\]

\[
\gamma_1 : x \mapsto x, \quad j = 1.
\]

Hence, a \( L \)-layer, \( d \) dimensional neural network is defined as (3.1.1), where the number of sigmoid composition is a predefined constant (hyper-parameter) \( L \in \mathbb{N}^+ \) called number of layers. Specifically, at each layer, the parameter matrices \( W[j] \), \( b[j] \), \( j = 1, 2, \ldots, L \) have different values and dimensions.

\[
N(x; W, b) = \gamma_L \circ \gamma_{L-1} \circ \cdots \circ \gamma_3 \circ \gamma_2 \circ \gamma_1 \tag{3.1.1}
\]

Remark 3.2 (Layer Output Notation) In general, at the layer level, if an input \( x \in \mathbb{R}^{n_1} \) with a total number of layers \( L \), layers 1 and \( L \) are called the input and output layers, respectively. The action at each layer \( j \) of the network can be
summarized as follows.

\[
a^{[1]} = x = (x_1, x_2, \cdots, x_d), \quad d = n_1,
\]

\[
a^{[j]} = \sigma(W^{[j]}a^{[j-1]} + b^{[j]}) \in \mathbb{R}^{n_j}, \quad j = 2, \cdots, L.
\]

Suppose at layer \( j \), there are \( n_j \) neurons, where \( n_j \) is a predetermined integer. Overall, the neural network is a map from \( \mathbb{R}^{n_1} \) to \( \mathbb{R}^{n_L} \).

For example, suppose \( x \in \mathbb{R}^{n_1} \), we denote \( z_2 = W^{[2]}x + b^{[2]}, \quad a_2 = \sigma(W^{[2]}x + b^{[2]}) \) at the second layer \( j = 2 \). At \( j \)th layer, \( z_j = W^{[j]}x + b^{[j]}, \quad a_j = \sigma(W^{[j]}x + b^{[j]}) \in \mathbb{R}^{n_j}, \quad \sigma : \mathbb{R}^{n_{j-1}} \rightarrow \mathbb{R}^{n_j} \) is defined in the component-wise manner, so that \( \{\sigma(z)\}_j = \sigma(z_j) \). The vector of outputs from the next layer has the form: \( \sigma(Wa + b) \), where \( a \in \mathbb{R}^{n_{j-1}} \) is the output from the previous layer, \( b \in \mathbb{R}^{n_j} \).

**Remark 3.3 (Weights and Bias Notation)** \( W^{[j]} \) is a matrix of dimension \( n_j \times (n_j - 1) \), and \( b^{[j]} \) is a column vector of dimension \( n_j \times 1 \). We denote \( W^{[j]} \in \mathbb{R}^{n_j \times n_{j-1}} \) the matrix of weights at layer \( j \). Similarly, \( b^{[j]} \in \mathbb{R}^{n_j \times 1} \) is the vector of biases for layer \( j \). If we discuss network at a general level, simply denote the parameter matrices as \( W \) and \( b \), without layer labels.

**Examples of Activation Functions**

We introduced neural networks using the notation of the sigmoid activation function, See Example 3.2. There are also other alternative activation functions that are more frequently used in practice, such as the hyperbolic tangent function \( \tanh(x), \quad x \in \mathbb{R} \).
Example 3.2 (Sigmoid Function) Let $x \in \mathbb{R}$, the sigmoid function is defined as a smooth version of step function, with the following derivative property.

\[
\sigma(x) = \frac{1}{1 + e^{-x}}, \\
\sigma'(x) = \sigma(x)(1 - \sigma(x)).
\]

The sigmoid function is only one choice the so-called activation function that adds nonlinearity to a linear term, providing smoothing effects.

Example 3.3 (ReLU Function) ReLU (rectified linear unit) function is defined as

\[
ReLU(x) = \max(0, x) = x^+, \quad x \in \mathbb{R}.
\]

At each layer, every neuron outputs a single real number, which is passed to every neuron in the next layer. At the next layer, each neuron forms its own weighted combination of these values, adds its own bias, and applies the sigmoid or other nonlinear activation functions.

In summary, the dimension of parameter $W^{[j]}$, $b^{[j]}$ corresponds to the number of neurons $n_j$, at $j$ th layer, which are the two parameter vectors for optimization task later. The total number of compositions corresponds to the number of network layers $L$, which is predefined and will not change after we determine the network structure. The network architecture (function structure) is completely up to user’s own design, for simplicity, we use a four-layer network, with sigmoid activation function to illustrate the basic idea.
Examples of Neural Networks

Example 3.4 (1D 4 layer network) For $x \in \mathbb{R}$, we consider a one dimensional four-layer neural network, with only 4 neurons. The representation of the network are 4 circles (neurons) connected by a straight line, See Figure 3.3.

![Diagram of a 1D 4-layer network](image)

Figure 3.3: A 1D four-layer network example.

At Layer 2, the parameters are $W^{[2]} \in \mathbb{R}^{1 \times 1}$ and $b^{[2]} \in \mathbb{R}^{1 \times 1}$. At layer 4, $W^{[4]} \in \mathbb{R}^{1 \times 1}$ and $b^{[4]} \in \mathbb{R}^{1 \times 1}$. Hence, the network is a four-times composition.

\[
\begin{align*}
N : (x; W, b) & \rightarrow N(x; W, b), \\
\gamma_j : (x; W^{[j]}, b^{[j]}) & \rightarrow \gamma_j(x; W^{[j]}, b^{[j]}) = \sigma(x; W^{[j]}, b^{[j]}), \\
N(x; W, b) & = \gamma_4 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1(x) \\
& = \sigma(W^{[4]}(\sigma(W^{[3]}(\sigma(W^{[2]}x + b^{[2]})) + b^{[3]})) + b^{[4]}) \in \mathbb{R}
\end{align*}
\]

Now, let us return to the general case. Assume that the input $x \in \mathbb{R}^{n_1}$ and the steepness and location of the transition in the sigmoid function can be changed by scaling and shifting the argument by weighting ($W$) and biasing ($b$) the input.
By changing the dimension of $\mathbf{W}$ and $\mathbf{b}$, we can alter the inner structure of a network.

**Example 3.5 (2D 4 layer network)** For $\mathbf{x} \in \mathbb{R}^2$, consider a two-dimensional four-layer network with $\mathbb{R}^2$ output. In this four layer network, suppose the input $\mathbf{x} \in \mathbb{R}^2$, the weights and biases for the second layer may be represented by a matrix $\mathbf{W}^{[2]} \in \mathbb{R}^{2 \times 2}$ and a vector $\mathbf{b}^{[2]} \in \mathbb{R}^{2 \times 1}$, respectively. See Figure 3.4.

![Diagram of a four-layer network](image)

Figure 3.4: A four-layer network

The output from Layer 2 then has the form $\mathbf{a}^{[2]} = \sigma(\mathbf{W}^{[2]}\mathbf{x} + \mathbf{b}^{[2]}) \in \mathbb{R}^2$. Similarly for Layer 3, $\mathbf{a}^{[3]} = \sigma(\mathbf{W}^{[3]}(\sigma(\mathbf{W}^{[2]}\mathbf{x} + \mathbf{b}^{[2]})) + \mathbf{b}^{[3]}) \in \mathbb{R}^3$ with $\mathbf{W}^{[3]} \in \mathbb{R}^{3 \times 2}$, $\mathbf{b}^{[3]} \in \mathbb{R}^{3 \times 1}$. While Layer 4 is the output layer, with $\mathbf{W}^{[4]} \in \mathbb{R}^{2 \times 3}$, $\mathbf{b}^{[4]} \in \mathbb{R}^2$. Thus the overall output is $\mathbf{x} \in \mathbb{R}^2 \rightarrow N(\mathbf{x}; \mathbf{W}, \mathbf{b}) \in \mathbb{R}^2$, $\mathbf{y}^{(i)} = N(\mathbf{x}^{(i)})$ for each sample.
i. The mathematical definition for this network is,

\[ N : (x; W, b) \mapsto N(x; W, b), \]

\[ \gamma_j : (x; W^{[j]}, b^{[j]}) \mapsto \gamma_j(x; W^{[j]}, b^{[j]}) = \sigma(x; W^{[j]}, b^{[j]}), \]

\[ N(x; W, b) = \gamma_4 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1(x) \]

\[ = \sigma(W^{[4]}(\sigma(W^{[3]}(\sigma(W^{[2]}x + b^{[2]})) + b^{[3]})) + b^{[4]})) \in \mathbb{R}^2. \]

**Example 3.6 (4D 5 layer network)** Consider the following four dimensional five-layer neural network \( (L = 5) \), \( n_1 = 4, n_2 = 3, n_3 = 4, n_4 = 5 \) and \( n_5 = 2 \), so the dimensions of parameters are \( W^{[2]} \in \mathbb{R}^{3 \times 4}, W^{[3]} \in \mathbb{R}^{4 \times 3}, W^{[4]} \in \mathbb{R}^{5 \times 4}, W^{[5]} \in \mathbb{R}^{2 \times 5} \). In addition, we have \( b^{[2]} \in \mathbb{R}^{3}, b^{[3]} \in \mathbb{R}^{4}, b^{[4]} \in \mathbb{R}^{5} \) and \( b^{[5]} \in \mathbb{R}^{2} \).

See Figure 3.5.

![Figure 3.5: A five-layer network.](image)
The mathematical definition of this five-layer network is hence

\[ N : (x; W, b) \mapsto N(x; W, b), \]
\[ \gamma_j : (x; W^{[j]}, b^{[j]}) \mapsto \gamma_j(x; W^{[j]}, b^{[j]}) = \sigma(x; W^{[j]}, b^{[j]}) , \]
\[ N(x; W, b) = \gamma_5 \circ \gamma_4 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1(x) \]
\[ = \sigma(W^{[5]}(\sigma(W^{[4]}(\sigma(W^{[3]}(\sigma(W^{[2]}x + b^{[2]})) + b^{[3]})) + b^{[4]})) + b^{[5]})) \]
\[ \in \mathbb{R}^2. \]

**Remark 3.4 (Total Number of Parameters)** In a \( L \) layer neural network, the parameter matrices are \( W^{[j]} \) and \( b^{[j]} \) at layer \( j \), \( j = 1, 2, 3 \ldots L \). As \( W^{[j]} \in \mathbb{R}^{n_j \times n_{j-1}} \) and \( b^{[j]} \in \mathbb{R}^{n_j \times 1} \), there are \( n_j \times n_{j-1} \) weights and \( n_j \times 1 \) bias at layer \( j \).

Hence, the total number of parameter is

\[ N_{TP} = \sum_{j=2}^{L} (n_j \times n_{j-1}) + n_j \times 1 \]
\[ = \sum_{j=2}^{L} n_j \times (n_{j-1} + 1). \]

A neural network is a nonlinear composite function, with given parameter dimensions. The idea of neural network approximation is to find the best parameter matrices \( W^{[j]}, b^{[j]} \) for all \( j \) by optimization, typically the parameter before optimization is initialized randomly.

**Remark 3.5 (Types of Learning Tasks)** In practice, we use a network to produce a vector output \( a^{[n_L]} \in \mathbb{R}^{n_L} \) to approximate or predict the target variable vector \( y^{(i)} \), or the vector \( y \) in a form \( \hat{y} = N(x; W, b) \), where \( n_L \) represents number of neurons at the \( L \) th layer (output layer). If we have prior information
about target variable $y$, then the task is called supervised-learning, which consists of classification, regression, function approximation etc. If the target variable $y$ is continuous, the problem is frequently approximation or regression. If it is discrete, then the problem is a classification problem, with $\mathbf{a}^{[L]} \in \mathbb{R}^{n_L}$ being integers $0, 1, 2 \cdots K$, $K > 2$, $K \in \mathbb{N}^+$ to classify data into $K$ categories.

We can approximate different functions via a neural network, including the solution of a PDE.

## 3.2 PINN Theory For Forward Problems

PINN uses a neural network to approximate a solution to a differential equation which satisfies the given initial and boundary conditions. The method itself does not require large data sets (points in domain) to train [5]. We use neural networks $N(x; W, b)$ as defined above to approximate the space-time solution. The loss function is constructed with the PDE and its initial, and boundary conditions. This problem formulation process that includes the PDE, initial, and boundary conditions given by physics into the loss function is the so-called incorporation of physical laws, hence the method is called physics informed neural network (PINN).

We begin with an example from [20] to illustrate the trial solution idea, which motivated the standard PINN algorithm. Let us start with an ODE.

**Example 3.7 (ODE Trial Solution)** Consider the following ODE, with initial condition $u(0) = 1$, and $x \in [0, 1]$.

$$\frac{du}{dx} + u(x + \frac{1 + 3x^2}{1 + x + x^3}) = x^3 + 2x^2 + x^2 \frac{1 + 3x^2}{1 + x + x^3} \tag{3.2.1}$$
We search for a trial solution \( \phi(x) \) of form (3.2.2),

\[
\phi(x) = 1 + xN(x; W, b). \tag{3.2.2}
\]

The trial solution \( \phi \) is hence selected to automatically satisfies the initial condition when we plug it back to ODE (3.2.1). The loss function \( L \) is the residual to optimize the parameters \( W, b \) in the network \( N(x; W, b) \).

\[
L(W, b) = \left\| \frac{d\phi}{dx} + \phi(x) + \frac{1 + 3x^2}{1 + x + x^3} - (x^3 + 2x^2 + x^2 + \frac{1 + 3x^2}{1 + x + x^3}) \right\|_2
\]

Next, we continue with a PDE example, where the trial solution method is applied to a more delicate example [20].

**Example 3.8 (Poisson Equation with Dirichlet BC)** Consider a two dimensional Poisson equation, with Dirichlet boundary condition on unit square \( (x, y) \in [0,1]^2 \), \( f(x, y) \) is a given 2D function, and \( f_0, f_1, g_0, g_1 \) are given sufficiently smooth functions.

\[
\begin{cases}
\nabla^2 u(x, y) = f(x, y),
\quad \\
u(0, y) = f_0(y), \quad u(1, y) = f_1(y), \\
u(x, 0) = g_0(x), \quad u(x, 1) = g_1(x).
\end{cases} \tag{3.2.3}
\]

We denote the trial solution as \( \phi \), and the loss function by \( L \). Function \( A \) is
chosen to satisfy the above Dirichlet boundary conditions.

\[
\phi(x, y; W, b) = A(x, y) + x(1 - x)y(1 - y)N(x, y; W, b),
\]

\[
A(x, y) = (1 - x)f_0(y) + xf_1(y) + (1 - y)(g_0(x) - [(1 - x)g_0(0) + xg_0(1)])
+ y(g_1(x) - [(1 - x)g_1(0) + xg_1(1)]).
\]

Since the trial solution is selected such that it can satisfy the boundary condition automatically, we obtain the loss function \( L \) as follow,

\[
L(W, b) = \|\nabla^2 \phi(x, y; W, b) - f(x, y)\|_2
\]

(3.2.4)

\( N(x, y; W, b) \) is the approximated solution, where the parameters \( W, b \) are randomly initialized.

We have shown how to incorporate the Dirichlet boundary conditions to the loss function’s construction, and we present another example to demonstrate how other types of boundary conditions can be incorporated as well.

**Example 3.9 (Poisson Equation with Mixed BC)** Consider the same set up and domain from the last example. If we have a mixed boundary condition instead, we need to change our choice of trial solution.

\[
\begin{cases}
\nabla^2 u(x, y) = f(x, y), \\
u(0, y) = f_0(y), & u(1, y) = f_1(y), \\
u(x, 0) = g_0(x), & \frac{\partial u(x, 1)}{\partial y} = g_1(x).
\end{cases}
\]

(3.2.5)
Let the trial solution be \( \psi(x) \) defined as follows,

\[
\psi(x; W, b) = B(x, y) + x(1-x)y[N(x, y; W, b) - N(x, 1; W, b) - \frac{\partial N(x, 1; W, b)}{\partial y}],
\]

\[
B(x, y) = (1-x)f_0(y) + xf_1(y) + g_0(x) - [(1-x)g_0(0) + xg_0(1)]
\]

\[
+ y(g_1(x) - [(1-x)g_1(0) + xg_1(1)]).
\]

Hence, the loss function is obtained similarly.

\[
L(W, b) = \|\nabla^2 \psi(x, y; W, b) - f(x, y)\|_2
\] (3.2.6)

The main idea of Lagaris’ approach is then to select a trial solution to incorporate the boundary condition. However, such selection of trial solution can not be applied to every single equation, as each boundary condition has different trial solutions and they are not unique. If we have more complicated boundary conditions, construction of such a trial solution may not be possible at all.

Lagaris’ idea is a stepping stone to the more general PINN method, that we simply separate the loss function to different sub-parts to train, but the trial solution can still be used (if we can find one) to greatly reduce the training.

In order to introduce the general PINN method, we consider a parameterized nonlinear PDE example.

Let \( x \in \Omega \subseteq \mathbb{R}^n, t \in [0, T], \mathcal{L} \) is a nonlinear operator and \( f \) is a given function: \( f : (x, t) \in \mathbb{R}^n \times \mathbb{R}^+ \mapsto \mathbb{R} \). Consider \( u_t + \mathcal{L}[u] = f(x, t) \), we select different operators to demonstrate the PINN method.

**Example 3.10 (Non-linear PDE)** For example, the one dimensional Burgers’ equation has the corresponding operator \( \mathcal{L}[u] = (u^2/2)_x \). We proceed by approx-
imating $u$ by a neural network $N(x, t; W, b)$. This network can be derived by applying the chain rule for differentiating composited functions using automatic differentiation [10], and has the same parameters as the network representing $u$, albeit with different activation functions due to the action of the differential operator $L$. The parameters can be learned by minimizing the mean squared error loss, denoted by $MSE$ [25].

**Definition 3.3 (Mean Squared Error)** Suppose $Y$ is the vector of observed values of the variable being predicted, and $\hat{Y}$ is the predicted values obtained from $n$ data points. Then the mean squared error of the predictor $\hat{Y}$ is computed as

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

We denote $(x^i_b, t^i_b)$ points on boundary of space-time domain and $(x^i_d, t^i_d)$ (colocation) points for the PDE residuals. Let $N_b$ denote the number of training (collocation) points on the boundary of the domain $\Omega$, including points $(x^i_b, 0)$ for initial conditions and $N_d$ denotes the number of points selected on the interior of the domain $\Omega$.

$$MSE = MES_b + MSE_d,$$

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} |u(x^i_b, t^i_b) - u^i|^2,$$

$$MSE_d = \frac{1}{N_d} \sum_{i=1}^{N_d} |u_t(x^i_d, t^i_d) + u(x^i_d, t^i_d)u_x(x^i_d, t^i_d) - f(x^i_d, t^i_d)|^2.$$

The loss $MSE_b$ corresponds to the initial and boundary data, while $MSE_d$ enforces the structure imposed by the differential equation at a finite set of collocation points.
Essentially, the process of PINN algorithm lies on customizing the loss function, based on the combination of PDE, initial and boundary conditions. The selection of training (collocation) points on $\Omega$ leads to justifies the terminology of data-driven approach. Once we obtain the loss function, the remaining task is to apply an optimizer to find the parameters $\mathbf{W}, \mathbf{b}$ of the network $N(x, t; \mathbf{W}, \mathbf{b})$ that minimizes the lost function $MSE$. In the next section, we introduce several existing optimization algorithms which are used in current neural network applications.

### 3.3 Optimization Algorithms

In current machine learning applications, researchers frequently treat the optimization algorithms as black boxes to use, although there are many existing gradient descent algorithms [1], [6], [17], [37], most of which are stochastic gradient descent variants. Researchers can not tell which modified algorithm is clearly superior than others, as different algorithms’ performance vary depending on the problem context. There are frequent situations when commonly used optimization algorithms fail. We are very interested in the comparison of existing methods to the 1D conservation law.

Currently, optimizer design and its trajectory’s connection to PDE is a very active research direction. We introduce existing algorithms that are proven successful. Optimization algorithms are often classified according to the orders of derivatives used in the gradient update step [20].
CHAPTER 3. PHYSICS INFORMED NEURAL NETWORK (PINN)

3.3.1 First Order Methods

Vanilla Gradient Descent (GD)

The vanilla gradient descend [13], also known as batch gradient descent, is the earliest method proposed to compute optimized parameters within a network. The idea is to update the parameters only in the opposite direction of the gradient of the objective function (loss function).

Let $T$ denote the total iteration, also known as epoch numbers and suppose there are $N$ data points in a given data set. Let $f : \mathbb{R}^D \rightarrow \mathbb{R}$ be a general, differentiable loss function, where $D$ is the dimension of input data. $\theta = [\theta_1, \theta_2, \ldots, \theta_j \ldots \theta_D] \in \mathbb{R}^D$, $j$ is an integer index from 1 to dimension $D$, number of parameters, $t$ represents the iteration index, a single time you update the entire parameter set.

Our goal is to minimize $f(\theta_t)$. If the multi-variable loss function $f(\theta_t)$ is differentiable in a neighbourhood of point $\theta_t$, then the value of $f(\theta_t)$ decreases fastest from the initial value $\theta_t$ in the direction of its negative gradient: $-\nabla f(\theta_t)$. Naturally, if

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t), \quad (3.3.1)$$

for a real number $\alpha \in \mathbb{R}^+$ as step size (learning rate), then $f(\theta_t) > f(\theta_{t+1})$. The latter term in (3.3.1) is subtracted from point $\theta_t$ in order to move to the opposite direction of derivative to reach the global minimum. We start from initial value of $\theta_0$, and apply the update rule, there is a corresponding sequence of points: $\theta_0, \theta_1, \theta_2 \ldots \theta_t$, such that $f(\theta_0) > f(\theta_1) > f(\theta_2) > \ldots f(\theta_t)$ and the sequence $(\theta_t)$
may converge to the global minimum. In fact, if the objective function $f$ is convex, all local minimum are also global minimum, hence in this case, $(\theta_t)$ is guaranteed to converge to a certain global minima $\theta$.

However, for non-convex functions, there is no guarantee that the algorithms converge, furthermore, this naive gradient descent performs worse in high-dimensional problems, who frequently oscillates at local minimum [30]. We denote $g_t$ as an intermediate value of gradient of $f$, the algorithm is given as follow:

**Algorithm 1 Vanilla Gradient Descent**

1: procedure GD($\alpha, \theta_t, f(\theta_t)$)
2: Initialization hyper parameters
3: $\alpha \leftarrow \alpha_0$ \hfill $\triangleright$ Initialization Value
4: $\theta_t \leftarrow \theta_0$ \hfill $\triangleright$ Initialization Value
5: $T \leftarrow$ epoch number \hfill $\triangleright$ Initialization Value
6: for $t = 1 : T$ do \hfill $\triangleright$ For each epoch
7: $g_t \leftarrow \nabla_{\theta_t} f(\theta_t)$ \hfill $\triangleright$ Compute Gradient of the objective function
8: $\theta_t = \theta_{t-1} - \alpha \cdot g_t$ \hfill $\triangleright$ Update in the negative gradient direction
9: return $\theta_T$

Note that the loop at each time step updates the gradient $\nabla_{\theta_j} f(\theta_t)$ for $j \in \{1 \ldots D\}$. We used vectored computation to encapsulate parameter representation. In practice, $\alpha_0$ is often taken equal to 0.01, and randomly initialize $\theta_0$ using a uniform distribution. In fact, the initialization value of parameter can affect training speed, consequences and refined initialization schemes are proposed. For example, the Xavier scheme initializes the weights such that the variances of the activation are the same across every layer. The constant variance helps prevent the gradient from exploding or vanishing [8].

The vanilla GD calculates every parameter’s derivative at each time step $t$. The presented algorithm used vector notation, ignoring the inner loop of parameter
$\theta_t$. Hence, for a single epoch, the GD method has computational complexity of $O(ND)$ [30]. However, the computational cost is too high when handling high dimensional problems (large $D$). Also, manual tune of the step size $\alpha$ is also needed for vanilla GD.

**Stochastic Gradient Descent (SGD)**

One problem that occurs in the vanilla gradient descent is that due to the large number $N$ of training samples, it is computationally expensive to calculate loss introduced by all samples. Therefore, researchers came up with the idea that we only calculate gradient of parameters for one random sample to perform update, through one epoch. This largely reduces computational cost. The resulting stochastic gradient is proven to be an unbiased estimate $\hat{\theta}_{sgd}$ of real parameter $\theta_t$. The resulted algorithm complexity is $O(D)$, independent of the sample size $N$, because of random selection.

The update rule is much faster than vanilla gradient descent when the sample size is large, but update direction oscillates due to random selection. When we slowly decreases the learning rate, parameters updated by SGD show the same convergence behaviour as those by vanilla gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively [30]. SGD is given in algorithm 2, adopting the same notations from GD.
Algorithm 2 Stochastic Gradient Descent

1: procedure SGD($\alpha, \theta_t, f(\theta_t)$)
2: initialization hyper parameters
3: $\alpha \leftarrow \alpha_0$ \hspace{1cm} \text{$\triangleright$ Initialization Value}
4: $\theta_t \leftarrow \theta_0$ \hspace{1cm} \text{$\triangleright$ Initialization Value}
5: $T \leftarrow$ epoch number \hspace{1cm} \text{$\triangleright$ Initialization Value}
6: for $t = 1 : T$ do
7: Randomly pick sample $j$ from $N$ samples
8: $g_t \leftarrow \nabla_{\theta_t} f(\theta_t)$ \hspace{1cm} \text{$\triangleright$ Compute gradient w.r.t $\theta$}
9: $\theta_t = \theta_{t-1} - \alpha \cdot g_t$ \hspace{1cm} \text{$\triangleright$ Update entire parameter vector}
10: $t \leftarrow t + 1$
11: return $\theta_T$

Mini-batch SGD

We only pick one sample from the training set in SGD for each update step, which leads to strong oscillations in the gradient values. As a result, it takes more iterations for the parameter sequence $(\theta_t), t = 1 \cdots T$ to reach the global minimum $\theta^*$. Even we may not reach it at all. Hence, the mini-batch method is proposed to reduce instability induced by random selection.

Mini-batch SGD is a compromise between GD and SGD [30]. The Mini-batch SGD (MSGD) method uses $b$ samples instead of one, at each iteration, making convergence more stable, reducing variance. Batch gradient descent may cause objective function fall into local minimum, where SGD escapes from local minimum by random jumps. Although we select more samples in the mini-batch variant, the oscillations of gradient value always exists, despite mitigated when compared to the usual SGD. As for learning rate (step size), a too small learning rate results in slow convergence rate, while a large rate will hinder convergence potentially, making objective function fluctuate around minimum.
Consider the $N$ training samples form an array, indexed from 1 to $N$, we divide the set into $M$ subsets, and the resulting subsets are called mini-batches of size $b$, i.e. $N = M \cdot b$. We then perform SGD on each batch.

**Algorithm 3** Mini-batch Stochastic Gradient Descent

1: procedure MSGD($\alpha, \theta_t, f(\theta_t), b$)
2: initialization hyper parameters
3: $\alpha \leftarrow \alpha_0$ \hspace{1cm} $\triangleright$ Initialization Value
4: $\theta_t \leftarrow \theta_0$ \hspace{1cm} $\triangleright$ Initialization Value
5: $T \leftarrow$ epoch number \hspace{1cm} $\triangleright$ Initialization Value
6: Divide the training set into $M$ batches of size $b_0$
7: $b \leftarrow b_0$ \hspace{1cm} $\triangleright$ Initialize batch size
8: for $t = 1 : T$ do
9: $\Sigma g_i \leftarrow 0$
10: Randomly pick a mini-batch from $M$ mini-batches
11: for $i = 1 : b_0$ do \hspace{1cm} $\triangleright$ For each point in the selected batch
12: $g_i \leftarrow \nabla \theta f(\theta_i)$ \hspace{1cm} $\triangleright$ Compute Gradient at sample $i$
13: $\Sigma g_i \leftarrow g_i + \Sigma g_{i-1}$ \hspace{1cm} $\triangleright$ Add the gradient of each point in batch
14: $i \leftarrow i + 1$
15: $g_t = \Sigma g_i / b_0$ \hspace{1cm} $\triangleright$ Compute the average gradient of $b$ points
16: $\theta_t = \theta_{t-1} - \alpha \cdot g_t$ \hspace{1cm} $\triangleright$ Update Parameter
17: $t \leftarrow t + 1$
18: return $\theta_T$

Similar to the learning rate $\alpha$, the mini-batch size $b$ is also a predetermined constant (hyper parameter) and the selection of $b$ can also affect training speed [23]. Empirically, we often set the batch size from 30 to 500, [23], depending on the scale $N$ of the given data set.

Although the batch technique was proposed to refine SGD, there are still remaining challenges. For example, hyper parameter tuning process is mostly empirical, thus lacking rigorous justification. Specifically, selecting a learning rate $\alpha$ which is too small leads to slow convergence, while a learning rate that is too large can cause the loss function to fluctuate around the minimum or even
to diverge. If our data are sparse, for instance, \( \theta_2, \theta_4, \theta_{10} \) are extremely small compare to other \( \theta_j \)s contained in the parameter vector \( \theta = [\theta_1, \ldots, \theta_j, \ldots, \theta_D] \), we might not want to update all of the \( \theta_j \)s in \( \theta \) to the same extent, but perform a smaller update for rarely occurring features.

**Remark 3.6 (Epochs and Iterations)** An epoch refers to the whole data set has undergone forward propagation and back propagation once. Running multiple epochs means feeding the data set to train multiple times, avoiding under-fitting. The total number of batches needed to complete one epoch is called (the number of) iterations. For instance, if we have 1000 data points, and divided them into 10 batches, each of size 100, then the number of batches needed to complete one epoch is 10. We ran 10 iterations in a single epoch.

\[
\text{epochs} = \text{batch size} \times \text{iterations}
\]

**SGD Momentum Method**

The batch SGD does not compute the exact derivative of objective function at each data point. Instead, it estimates the derivative on a small batch of samples, so that the gradient does not always go in the optimal direction. As a result, there is need to preserve certain degree of the influence from previous update direction to the next iteration, leading to the introduction of the momentum method.

The term momentum is an analogy to momentum in physics [1]: the weight vector \( \theta \) is treated as a particle traveling through a parameter space, incurring acceleration from the gradient of the objective function, analogy to the force. The core idea is to keep information of last update direction. Variable \( v \) represents
speed, given by direction and rate of parameter’s movement in the parameter space $\Theta$. We denote this speed as follow,

$$ v = \alpha \left( \frac{\partial f(\theta_t)}{\partial \theta} \right) $$

where $\theta_t \in \Theta, t = 1, 2 \ldots T$. Momentum method helps accelerate the SGD [1] in the relevant direction and dampens oscillations. It does this by adding another constant (hyper parameter) $\mu$, where $\mu$ is the momentum coefficient between $[0, 1]$, often known as momentum factor. It is usually set to $\mu_0 = 0.9$ to imitate the fraction in Physics [30]. $v_t$ is the previous update of the update vector of the past time step to the current update vector, an intermediate value. The classic momentum SGD is summarized in the following algorithm.

**Algorithm 4 SGD (Classical Momentum)**

1: procedure SGD Momentum($\alpha, \mu, \theta_t, f(\theta_t)$)  
2:   initialization hyper parameters  
3:   $\alpha \leftarrow \alpha_0$ $\quad \triangleright$ Initialization Value  
4:   $\mu \leftarrow \mu_0$ $\quad \triangleright$ Initialization Value  
5:   $\theta_t \leftarrow \theta_0$ $\quad \triangleright$ Initialization Value  
6:   $T \leftarrow$ epoch number $\quad \triangleright$ Initialization Value  
7:   $v_0 \leftarrow \alpha \left( \frac{\partial f(\theta_0)}{\partial \theta_0} \right)$ $\quad \triangleright$ Initialize speed  
8:   for $t = 1 : T$ do  
9:       $g_t \leftarrow \nabla_{\theta} f(\theta_t)$ $\quad \triangleright$ Calculate Gradient  
10:      $v_{t+1} = \mu v_t - \alpha g_t$. $\quad \triangleright$ Accumulate Gradient  
11:      $\theta_{t+1} \leftarrow \theta_t + v_{t+1}$ $\quad \triangleright$ Apply Update  
12:      $t \leftarrow t + 1$  
13:   return $\theta_T$
Nesterov Accelerated Gradient Descent

The SGD momentum method with a proper momentum factor $\mu$ reduces the oscillation of parameter values when the learning rate is large. The selection of the proper size of the momentum factor is an open problem, which is similar to other hyper parameter tuning [31]. If the momentum factor is small, it is hard to obtain the effect of improving convergence speed. If the momentum factor is large, the current point may jump out of the optimal value point.

Nesterov Accelerated Gradient descent (NAG) makes further improvement over the traditional momentum method [29]. It differs from classical momentum in the sense that the gradient is evaluated after the current speed is applied, i.e. adding the speed term to parameter then calculate gradient. Additionally, it is possible to introduce a decay factor $d$, $d \in [0, 1]$ to decrease the initial learning rate $\alpha_0$ at each step $t$: i.e. $\alpha_t \leftarrow \alpha_0/(1 + dt)$. This improvement eliminates the need to tune $\alpha$, and further inspired the design of Adagrad.

Adagrad: Adaptive Learning Rate

In SGD, we can only adjust the learning rate after training. Adagrad eliminates such need to tune the hyper-parameter manually, but we still need to set an initial $\alpha$. See NAG. It adapts the learning rate to the parameters, performing smaller updates for small $\theta_j, j = 1, 2, \ldots, D$ and larger updates for larger $\theta_j$. Hence, Adagrad is well-suited for dealing with sparse data[6]. In its update rule, Adagrad modifies the general learning rate $\alpha$ at each time step $t$ for every parameter $\theta_t$ based on the past gradients that have been computed for $\theta_t$. We denote the accumulation of gradient $G_t$, with a small positive real number $\epsilon$ added to avoid singularity
Algorithm 5 Nesterov Accelerated Gradient descent

1: procedure NAG($\alpha, \mu, \theta_t, f(\theta_t)$) 
2: initialization hyper parameters 
3: $\alpha \leftarrow \alpha_0$  
4: $\mu \leftarrow \mu_0$  
5: $\theta_t \leftarrow \theta_0$  
6: $T \leftarrow$ epoch number 
7: $v_1 \leftarrow \alpha(\frac{\partial f(\theta_0)}{\partial \theta_0})$  
8: for $t = 1 : T$ do 
9: $\tilde{\theta}_t \leftarrow \theta_t + \mu v_{t-1}$  
10: $v_{t+1} = \mu v_t - \alpha(\frac{f(\tilde{\theta}_t)}{\tilde{\theta}_t})$.  
11: $\theta_{t+1} \leftarrow \theta_t + v_{t+1}$  
12: $t \leftarrow t + 1$  
13: return $\theta_T$

issues. The comprehensive experiments [6] showed that adaptive methods clearly outperform compared to their non-adaptive counterparts. However, Adagrad’s

Algorithm 6 Adagrad Optimizer

1: procedure ADAGRAD($\alpha, \theta_t, f(\theta_t)$) 
2: initialization hyper parameters 
3: $\alpha \leftarrow \alpha_0$  
4: $\theta_t \leftarrow \theta_0$  
5: $T \leftarrow$ epoch number 
6: for $t = 1 : T$ do 
7: $g_t \leftarrow \nabla_\theta f(\theta_t)$  
8: $G_t \leftarrow \sqrt{\sum_{i=1}^{t} g_i^2} + \epsilon$.  
9: $\alpha_{t+1} \leftarrow \alpha_t / G_t$  
10: $\theta_{t+1} \leftarrow \theta_t - \alpha_{t+1} g_t$  
11: $t \leftarrow t + 1$  
12: return $\theta_T$

main weakness is its accumulation of the squared gradients in the denominator [30]: since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually
become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. A manual initial learning rate is still needed for AdaGrad, and it is not suitable for nonconvex problems.

**AdaDelta: Further Improvement on Accumulated Gradient**

In 2012, Matthew proposed [37] AdaDelta, a first order method that does not require manual tuning of the learning rate. Recall that when the loss function approaches minimum, it can oscillate back and forth. The goal is to slow down the parameter update. Researchers attempted by adding momentum term, and decreased learning rate by introducing a decay factor or shrink $\alpha_0$ by accumulated gradient, but the zero denominator of $\alpha_t, t \to \infty$ needs to be solved. AdaDelta aims to improve Adagrad. It seeks to reduce Adagrad’s monotonically decreasing learning rate. Recall Adagrad’s update rule,

$$
\theta_{t+1} = \theta_t - \frac{\alpha_0 \cdot g_t}{\sqrt{\sum_{i=1}^t g_i^2 + \epsilon}}.
$$

It still has hand-tuned global learning rate, but each step has dynamic learning rate, which grows with the inverse of gradient. However, the accumulation of gradient in the denominator continues to decrease the learning rate until proper training stops, and thus the selection of initial global learning rate has an impact. Instead of accumulating all past squared gradients, AdaDelta restricts the window of accumulated past gradients to a fixed size $w$ [37]. Thus, the accumulated gradient will not tend to infinity and becomes local. AdaDelta’s gradient accu-
mulation is a moving average. We denote the moving average of a function \( g \) by \( \mathbb{E}(g)_t \), \( \Delta \theta = \theta_{t+1} - \theta_t \) and the root square mean \( RMS(g)_t = \sqrt{\mathbb{E}(g)_t + \epsilon} \) at each iteration \( t \). The experiment [37] on hand-written digit classification and speech data showed AdaDelta outperformed momentum, SGD, and Adagrad method. In implementation, we often take \( \rho_0 = 0.9 \) [37].

### RMSprop

RMSprop is an unpublished adaptive learning rate optimizer proposed by Geoff Hinton. Its motivation is similar to that of AdaDelta, which focuses on tackling the quickly decaying learning rate in Adagrad. RMSprop tackles this by keeping a moving average of the squared gradient and adjusting the weight updates by this magnitude.

### Adam: A comprehensive Improvement in 2015

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially
Algorithm 8 RMSprop

1: procedure RMSprop($\alpha, \rho, \theta, f(\theta)$)  
2:     initialization hyper parameters  
3:     $\alpha \leftarrow \alpha_0 \quad \triangleright \text{Initialization Value}$  
4:     $\rho \leftarrow \rho_0 \quad \triangleright \text{Initialization Value}$  
5:     $\theta_t \leftarrow \theta_0 \quad \triangleright \text{Initialization Value}$  
6:     $T \leftarrow \text{epoch number} \quad \triangleright \text{Initialization Value}$  
7:     for $t = 1 : T$ do  
8:         $g_t \leftarrow \nabla_{\theta} f(\theta_t) \quad \triangleright \text{Calculate gradient}$  
9:         $E(g^2)_t \leftarrow \rho E(g^2)_{t-1} + (1 - \rho) (g)_t^2 \quad \triangleright \text{Accumulate Gradient}$  
10:        $\theta_{t+1} = \theta_t - \frac{\alpha g_t}{\sqrt{E(g^2)_t + \epsilon}} \quad \triangleright \text{Apply Update}$  
11:        $t \leftarrow t + 1$  
12:     return $\theta_T$

decaying average of past squared gradients like AdaDelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to the momentum method. Adam seeks to combine RMSprop and Adagrad to compute individual adaptive learning rates. Formally, Adam is also a first order gradient based optimization algorithm [17]. The algorithm updates exponential moving averages of the gradient ($mt$) and the squared gradient ($vt$) where the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages [17].

In practice, the following values are selected: $\alpha_0 = 0.001, \beta_{1,0} = 0.9, \beta_{2,0} = 0.999, \epsilon = 10^{-8}$. The convergence analysis of Adam does not apply to problems considered in this work. Empirically, Adam often outperforms compared to other methods. The experiment [17] used a neural network model with two fully connected hidden layers with 1000 hidden units each and ReLU activation with mini-batch size of 128. Adam shows better convergence than other methods. In fact, it is the most widely used method today.
Algorithm 9 Adam Optimizer

1: procedure Adam($\theta_t, f(\theta_t), \beta_1, \beta_2, \alpha$) 
2:     initialization hyper parameters
3:     $\alpha \leftarrow \alpha_0$ \hspace{1cm} \triangleright \text{Initialization Value}
4:     $\beta_1 \leftarrow \beta_{1,0}$ \hspace{1cm} \triangleright \text{Initialization Value}
5:     $\beta_2 \leftarrow \beta_{2,0}$ \hspace{1cm} \triangleright \text{Initialization Value}
6:     $T \leftarrow$ epoch number \hspace{1cm} \triangleright \text{Initialization Value}
7:     initialize moment vectors:
8:     $m_0 \leftarrow 0$
9:     $v_0 \leftarrow 0$
10:    while $\theta_t$ not converge do
11:        $t \leftarrow t + 1$ \hspace{1cm} \triangleright \text{Time Step}
12:        $g_t \leftarrow \nabla_{\theta_t} f(\theta_{t-1})$ \hspace{1cm} \triangleright \text{Gradient w.r.t objective function}
13:        $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ \hspace{1cm} \triangleright \text{biased first moment estimation}
14:        $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) (g^2)_t$ \hspace{1cm} \triangleright \text{biased second moment estimation}
15:        $\hat{m}_t \leftarrow m_t / (1 - \beta_1)$ \hspace{1cm} \triangleright \text{bias-corrected first moment}
16:        $\hat{v}_t \leftarrow v_t / (1 - \beta_2)$ \hspace{1cm} \triangleright \text{bias-corrected second moment}
17:        $\theta_{t} \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ \hspace{1cm} \triangleright \text{Update parameter}

3.3.2 Second order methods: Quasi-Newton Methods

First order methods have their own limitations. For example, even an optimizer suitable for high dimensional large scale program, like Adam, can fail to converge in certain cases. Also for faster convergence rate, second order derivative is included, which naturally leads to a more precise minimization of the loss function. We further introduce common second-order methods to compare its performance to first-order methods on the 1-d conservation law.

The basic idea of Newton’s method is to combine the usage of first order derivative i.e. gradient, and second order derivative i.e. Hessian matrix to approximate the loss function $f(\theta)$ by a second order Taylor polynomial. Later, we solve for $\theta_t$ of this quadratic function for optimal parameters. The process continues until the desired parameter converges. Quasi-Newton method’s update rule is developed
from Newton’s formula. We then extend this to higher dimension, and define $d_t$ as Newton’s direction, $\alpha_t$ the learning rate to produce the update rule for $\theta_t$. Assume $f$ has continuous second derivatives, the Newton algorithm is given by

- Scalar case: $\theta_{t+1} = \theta_t - \frac{f' (\theta_t)}{f'' (\theta_t)}$
- Vector case: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - (\nabla^2 f (\theta_t))^{-1} \cdot \nabla f (\theta_t)$,

and the Quasi-Newton algorithm is given by

$$d_t = (\nabla^2 f (\theta_t))^{-1} \cdot \nabla f (\theta_t), \quad \theta_{t+1} = \theta_t + \alpha_t \cdot d_t$$

Theoretically, the addition of second order derivative provides a much faster convergence rate. However, in practice, the storage and calculation of the inverse Hessian matrix term $(\nabla^2 f (\theta_t))^{-1}$ is quite expensive. As a result, the idea of using an approximate inverse Hessian matrix is proposed. There are different ways to approximate the Hessian matrix. This set of the problem can be summarized as the Quasi-Newton Condition, which is a prerequisite to the BFGS algorithm introduced later.

**Quasi-Newton Condition**

Assume the loss function $f(\theta_t)$ can be approximated by a quadratic function. Expand $f(\theta_t)$ at $\theta = \theta_{t+1}$ and take gradient on both sides,

$$f(\theta) = f(\theta_{t+1}) + \nabla f(\theta_{t+1})^T (\theta - \theta_{t+1}) + \frac{1}{2} (\theta - \theta_{t+1})^T \nabla^2 f(\theta_{t+1}) (\theta - \theta_{t+1})$$

$$\nabla f(\theta) = \nabla f(\theta_{t+1}) + \nabla^2 f(\theta_{t+1}) (\theta - \theta_{t+1})$$
Let $B_{t+1}$ be the approximated Hessian matrix $\nabla^2 f(\theta_{t+1})$ that satisfies the Quasi-Newton condition:

$$s_t = \theta_{t+1} - \theta_t,$$

$$u_t = \nabla f(\theta_{t+1}) - \nabla f(\theta),$$

$$u_t = B_{t+1} \cdot s_t.$$

We use $H_t = (B_t)^{-1}$ to represent the approximated inverse Hessian matrix, the update rule of general quasi-Newton method becomes

$$g_t = \nabla f(\theta_t),$$

$$d_t = -(B_t)^{-1} \cdot g_t,$$

$$\theta_{t+1} = \theta_t + \alpha_t \cdot d_t.$$

**BFGS**

The first way of updating approximated Hessian is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, with corresponding update rule of $H_t$ given by:

$$B_{t+1} = B_t - \frac{B_t s_t s_t^T B_t}{s_t^T B_t s_t} + \frac{u_t u_t^T}{u_t^T s_t},$$

$$H_{t+1} = (I - \frac{s_t u_t^T}{s_t^T u_t}) \cdot H_t \cdot (I - \frac{u_t s_t^T}{s_t^T u_t}) + \frac{u_t s_t^T}{s_t^T u_t}.$$

The listed quasi-Newton algorithm still cannot solve large scale optimization problem as the approximation of Hessian generates a sequence of matrices [30], request-
ing a substantial amount of memory to store these matrices, especially for high dimensional problems.

L-BFGS

L-BFGS is an improved version of BFGS, enabling itself to handle high dimensional problems. L-BFGS does not fully compute the $D \times D$ Hessian approximate and only keeps a few $n$ dimensional vectors. The main idea is to store the vector sequence in calculating approximate $H_{t+1}$ instead of the complete $H_t$. The Hessian approximate $H_{t+1}$ hence obtained from a sequence pair $(s_t, u_t)_{t=p+1}^{t}$ such that,

$$H_{t+1} = (I - \frac{s_t u_t^T}{u_t^T s_t}) \cdot H_t \cdot (I - \frac{u_t s_t^T}{u_t^T s_t}) + \frac{s_t s_t^T}{u_t^T s_t} = V_t^T H_t V_t + \rho s_t s_t^T$$

$$\rho = \frac{1}{s_t^T u_t}, V_t = I - \rho u_t s_t^T$$

Algorithm 10 Two-Loop Recursion for $H_t g_t$

1: procedure TWO-LOOP($\nabla f(\theta_t), u_t, s_t$)
2: \hspace{1em} Set $g_t = \nabla f(\theta_t)$
3: \hspace{1em} Set $H^0_t = \frac{s_t^T u_t}{\|u_t\|^2} I$
4: \hspace{1em} for $l = t - 1$ to $l = t - p$ do
5: \hspace{2em} $\alpha_l = \rho s_t^T g_{t+1}$
6: \hspace{2em} $g_l = g_{t+1} - \alpha_l u_t$
7: \hspace{2em} $r_{t-p-1} = H^0_t g_t - \rho$
8: \hspace{1em} for $l = t - p$ to $l = t - 1$ do
9: \hspace{2em} $\beta_l = \rho u_t^T r_{l-1}$
10: \hspace{2em} $\rho_l = \rho_{l-1} + s_t (\alpha_l - \beta_l)$
11: $H_{t+1} g_{t+1} = \rho$ \hspace{1em} ▷ Output $H_{t+1} g_{t+1}$

In general, Quasi-Newton methods have convergence rate faster than linear rate. However, Quasi-Newton method needs a large storage space; therefore it is...
Algorithm 11 L-BFGS

1: procedure L-BFGS(θ₀ ∈ ℝⁿ, ε < 0)
2:   Set g₀ = ∇f(θ₀)
3:   Set t = 0, u₀ = 1, s₀ = 1
4:   while ∥gₜ∥ < ε do
5:     Choose H₀ₜ = sₜᵀuₜ ∥uₜ∥⁻¹
6:     Set gₜ = ∇f(θₜ)
7:     Set dₜ = −Hₜgₜ from two-loop recursion
8:     θₜ₊₁ = θₜ + αₜdₜ
9:     if k > p then
10:        Discard (sₜ−p, yₜ−p) from storage
11:     Compute and Save sₜ = θₜ₊₁ − θₜ, uₜ = gₜ₊₁ − gₜ
12:     t ← t + 1

not suitable for handling the optimization of large-scale problems [30].

Despite there are many existing attempts to compare optimization algorithms on various data sets from epidemiology and computer vision, such as in [33], and [11] we are still interested in how the most commonly used first-order method Adam and second-order method L-BFGS perform under the PINN setting, see section 3.4.3 for the experiments. On the other hand, if the derivatives are not able to be obtained or are not defined, we may attempt derivative-free optimization methods, which does not have much attempts in the literature so far.

3.3.3 Derivative-Free Optimization

In practical problems, there are situations that the objective function does not exist or is not feasible to calculate. The way of finding the optimal point only by function values, without gradient information is called derivative-free optimization. The first way is to use heuristic algorithms that worked well, including clas-
sical simulated annealing arithmetic, genetic algorithms, ant colony algorithms, and particle swarm optimization [36]. They all yield global approximate solution but have weak theoretical support. We refer to [18], [21] and [4] for information on this group of algorithms, which are worth attempted to solve a discontinuous problem when derivatives are not obtainable.

For instance, the coordinate descent method is a derivative-free method. It is suitable for non-differentiable objective function. The main idea is to select bases $e_1, e_2, e_3 \ldots e_D$, as the search direction and minimize the objective function $f(\theta)$ in each corresponding direction.

Suppose $\theta_t$ is already obtained, the $j$th dimension of $\theta_{t+1}$ is given by computing [36] $\theta_{t+1} = \arg \min_{\theta_j} f(\theta_{t+1}^1 \ldots \theta_{j} \ldots \theta_{D}^t)$. The convergence is guaranteed by $f(\theta_{t+1}) < f(\theta_t) < \cdots < f(\theta_0)$, similarly to vanilla gradient descent [30]. $D$ represents the number of dimension of input (features); From $e_1$ to $e_D$, there are $D$ directions corresponding to each feature variable, and the update order can be arbitrary.

The keynote difference is that coordinate descent method has its optimization direction fixed from the beginning. Hence, it does not need to compute gradient at each step. In contrast, the usual gradient descent schemes decides its update direction by computing current position gradient at each time step $t$. Apparently, the calculation in coordinate descent method is far simpler, especially for complicated objective functions. However, the selection of coordinate system must be cautious.

The cyclic scheme is the most straightforward. Each $\theta_{i_k}$ is selected in a cyclic manner. We can also change the selection method to random, picking $i_k$ by
Algorithm 12 Coordinate Descent [35]: cyclic indexing

\begin{algorithm}
\begin{algorithmic}[1]
  \Procedure{CD-cyclic}{$\theta_0, f(\theta_0)$}
  \For{$k = 1, 2, \ldots$ until convergence}
  \State Indexing: select coordinate index $i_k$
  \State $i_{k+1} = (k \mod D) + 1$
  \State Update: update $\theta_{i_k}^k$ via $f, \theta_k$ holding other coordinates same
  \State $\theta_{i_k}^k = \arg \min_{\theta_{i_k}} f(\theta_{i_k}; x_{x \neq i_k})$
  \EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

$P(i_k = j) = p_j$, where $p_j$ is pre-assigned probability of certain distribution. For example, $p_j = 1/D$ if we sample the index uniformly. The derivative-free methods are in general slower than those who uses derivatives. We refer to [35] for more details. Overall, derivative-free optimization methods are a great future work direction to overcome situations when derivative-based optimizers do not perform well in regular PINNs.

3.4 PINN Examples of Forward Problems

After a theoretical illustration of PINN principles, we presented several optimization algorithms which can be used in the training. It is possible to test different optimizers on a specific problem, given a fixed network size. In general, existing optimizers perform well on continuous problems. We then provide more practical examples to demonstrate PINN’s performances.

3.4.1 ODE: Population Growth Example

Let us start with a naive population ODE model.

Example 3.11 (Population Growth) Let $f$ be a unknown function that repre-
sents the population growth rate over time $t$, with $t > 0$ with $f(0) = 1$. Let $R \in \mathbb{R}$ be a constant that represents the maximum population growth rate. Consider the ODE

$$
\frac{df}{dt} = Rt(1 - t),
$$

(3.4.1)

We use $N(t; \mathbf{W}, \mathbf{b})$ to approximate the solution to (3.4.1). First, we randomly select $M$ collocation points $t_j, j = 1, 2 \ldots M$ from a uniformly spaced grid in the time domain, i.e. uniformly distributed points from $(0, 1)$. Our loss function consists of two parts,

$$
MSE = MSE_{ode} + MSE_{bc},
$$

$$
MSE_{ode} = \frac{1}{M} \sum_{j=1}^{M} \left( \frac{dN(t_j; \mathbf{W}, \mathbf{b})}{dt} - Rt_j(1 - t_j) \right)^2,
$$

$$
MSE_{bc} = (N(t_0 = 0; \mathbf{W}, \mathbf{b}) - 1)^2.
$$

The time variable is treated as a new dimensional variable, hence avoiding time-stepping and loops in the traditional finite difference and finite element methods. Next, we use automatic differentiation to calculate the derivative $N$ with respect to $t$ and $\mathbf{W}, \mathbf{b}$. We then use the obtained derivative to update the parameters $\mathbf{W}, \mathbf{b}$ by a selected optimization algorithm. The optimization algorithm updates the parameters of interest at each layer $j, j = 1, 2, \cdots, L$. The following is an example of vanilla gradient descent update to the parameters:

$$
\mathbf{W}^{[j]} = \mathbf{W}^{[j-1]} - \alpha \nabla_{\mathbf{w}} MSE,
$$
\[ \mathbf{b}^{[j]} = \mathbf{b}^{[j-1]} - \alpha \nabla_b \text{MSE}. \]

After updating parameters from all layers using GD for a given number of times (epochs), we have the final parameter \( \mathbf{W}^*, \mathbf{b}^* \) that minimizes the loss function \( \text{MSE} \). Therefore the resulted network \( t \mapsto N(t; \mathbf{W}^*, \mathbf{b}^*) \) gives the approximated neural network solution.

In our implementation, we used a two-layer network with hyperbolic tangent activation function. We selected 20 collocation points from \( (0, 1) \). (in practice, more than 10000 collocation points for better precision) and ran 20000 iterations using Adam [17] optimizer, with \( \alpha = 0.01 \), \( \beta_1 = 0.9 \), \( \beta_2 = 0.999 \). Figure 3.6 provides a comparison of NN-solution with analytic solution, and a plot of the loss function value advanced through the number of epochs. As we can see from the plot, the NN-solution closely matches the analytic solution, with a loss value less than \( 10^{-4} \) after 20000 epochs.

![Figure 3.6: ODE PINN-solution (\( R = 1 \)).](image-url)
3.4.2 PDE: Schrödinger’s Equation

The one-dimensional nonlinear Schrödinger equation is an equation which in particular is used to study optics and fluid dynamics, including nonlinear wave propagation in optical fibers, See [25]. We are here mainly interested in how PINN handle complex PDE solutions.

Example 3.12 (1D Schrödinger) Let \( x \in [-5, 5], \ t \in [0, \frac{\pi}{2}] \), \( h \) denotes the actual solution and \( N(x,t; W, b) \) be the neural network.

\[
\begin{align*}
    i h_t + 0.5h_x x + |h|^2 h &= 0, \\
    h(0, x) &= 2 \text{sech}(x), \\
    h(t, -5) &= h(t, 5), \\
    h_x(t, -5) &= h_x(t, 5).
\end{align*}
\]

\( h : (x,t) \in [-5, 5] \times [0, \frac{\pi}{2}] \mapsto \mathbb{C} \). We use \( h \) to derive the loss function, and then replace \( h \) with the network \( N(x,t; W, b) \) in computation. Denote \( u \) the real part of \( h \), and \( v \) imaginary part as \( h = [u, v] \). We define the residual as follows,

\[
f = ih_t + 0.5h_x x + |h|^2 h
\]
We define the total loss function $MSE$ in three different parts [25].

$$MSE = MSE_{ic} + MSE_{bc} + MSE_{pde},$$

$$MSE_{ic} = \frac{1}{N_0} \sum_{i=1}^{N_0} |h(x^i_0, 0) - t^i_0|^2,$$

$$MSE_{bc} = \frac{1}{N_b} \sum_{i=1}^{N_b} (|h^i(-5, t^i_b) - h^i(5, t^i_b)|^2 + |h^i_x(-5, t^i_b) - h^i_x(5, t^i_b)|^2),$$

$$MSE_{pde} = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t^i_f, x^i_f)|^2,$$

where $\{t^i_b, i = 1, 2, \ldots, N_b\}$ represents the collocation points at the boundary, and $\{t^i_0, i = 1, \ldots, N_0\}$ represents $N_0$ points selected from the initial domain i.e. when $t = 0$, the domain is $\{x : x \in [-5, 5]\} \times \{t = 0\}$. $MSE_{ic}$ corresponds to the loss on the initial conditions, $MSE_{bc}$ denotes the loss function corresponding to the periodic boundary conditions, and $MSE_{pde}$ is the residual of the Schrödinger equation. The equation under consideration is one-dimensional. One of the advantages of PINN is that we can treat time variable $t$ as another dimensional variable. We use a four-layer fully connected network with depth 100, and select $N_f = 20000$ collocation points $t^i_f$ from interior and $N_b = 50$ points $t^i_b$ for boundary of domain, where $i$ is the index of selected points. The solution via Adam optimizer for 20000 epochs is obtained by our test. The result [25] consists of real and imaginary part. In Figure 3.7 the horizontal axis represents time while the vertical axis represents space.
Figure 3.7: Schrödinger’s equation PINN.
3.4.3 PDE: 1D Burgers’ Equation

Suppose we have a pipe, with fluid flowing in it. Burgers’ equation models the speeds of fluid at each location of the pipe as time progresses. We consider the following Burgers’ equation [22].

Example 3.13 (1D viscous Burger) Let \( x \in [-1, 1] \), \( t > 0 \) represent the spatial and temporal location, respectively. \((x, t) \in \Omega = [-1, 1] \times [0, 1]\), we consider

\[
\begin{align*}
  u_t + \left( \frac{u^2}{2} \right)_x &= \mu u_{xx}, \\
  u(-1, t) &= u(1, t) = 0, \\
  u(x, 0) &= -\sin(\pi x),
\end{align*}
\]

where \( \mu = 0.01 \) is a constant that models the viscosity of the fluid. We use a neural network \( N(\cdot, \cdot; W, b) \) to approximate the solution, where \( W, b \) denote the parameters to optimize. We define the PDE residual \( f = N_t + NN_x - \mu N_{xx} \), and the loss function \( MSE \) as the sum of loss from interior, boundary of the domain and the initial condition.

\[
MSE = MSE_{ic} + MSE_{bc} + MSE_{pde},
\]

\[
MSE_{ic} = \frac{1}{N_0} \sum_{i=1}^{N_0} |N(x^i_0, 0) + \sin(\pi x^i_0)|^2,
\]

\[
MSE_{bc} = \frac{1}{N_b} \sum_{i=1}^{N_b} (|N(-1, t^i_b) - 0|^2 + |N(1, t^i_b) - 0|^2),
\]

\[
MSE_{pde} = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(x^i_f, t^i_f)|^2.
\]

The set \( \{(x^i_b, t^i_b), i = 1, \ldots, N_b \} \) represents collocation points in the time in-
interval to enforce the boundary conditions and \( \{(x^i_f, t^i_f), i = 1, \ldots, N_f\} \) represent points inside the space-time domain. The network consists of 3 hidden layers. After 15000 epochs i.e. total number of iterations through a training set, with Adam and L-BFGS, respectively. The results are reported based on [22], See Figure 3.8 and 3.9.

Figure 3.8: Solution to problem with sin initial condition (left); Plot of loss value to number of iterations with Adam optimizer (right).

Figure 3.9: Solution to problem with sin initial condition at final time (left); Plot of loss value to number of iterations with L-BFGS optimizer (right).
In Figure 3.8 (left), we report the solution in the \((x_1, x_2)\) plane. Notice that our initial condition is a smooth sine function, optimization performs well for convex functions. However, if the initial condition is a discontinuous function, such as a step function:

\[
g(x) = \begin{cases} 
0, & x < 0, \\
1, & x > 0.
\end{cases}
\]

The implementation of a step function can be achieved by the lambda function in Python. The regular PINN algorithm presented in this chapter gives poor performance on its test loss curve. The first attempt to handle this issue is to try more powerful optimizers, such as SGD variants and higher order methods. We used a fully connected neural network of depth 4 (3 hidden layers) and width 20. We randomly selected 2540 collocation points inside the space-time domain, 80 points on boundary, 160 points for initial condition via the Xavier initialization technique, also known as Glorot normal initializer. The first attempt used Adam optimizer for 15000 iterations with learning rate 0.001, See Figure 3.10. The second attempt used L-BFGS, See Figure 3.11.

The results represent the solution given by regular PINN, a direct solver. Model via Adam optimizer took approximately 108 seconds in CPU time while L-BFGS only took 65 seconds. Despite much faster convergence with L-BFGS optimizer, the solution is different from the analytic one. Regular PINN is a direct solver that directly plug in network as approximation. However, in this problem, the step functions’ derivatives are not defined, therefore direct plug-in can not give the weak solution we desired. Theoretically, we need to use a test function to perform integration by parts for the weak solution. Hence, we need an algorithm
Figure 3.10: Attempt 1: Solution to problem with initial condition $g(x)$ (left); Plot of loss value to number of iterations with Adam optimizer (right).

Figure 3.11: Attempt 2: Solution to problem with $g(x)$ initial condition (left); Plot of loss value to number of iterations with L-BFGS optimizer (right).
of fundamentally different scheme that can provide an accurate approximation of the (weak) solution.
Chapter 4

A Non-Diffusive PINN-based Algorithm

4.1 Background

As early as in 1998, before the development of deep learning packages, such as Tensorflow, Pytorch in 2015 which enable large scale machine learning [2] [10], Lagaris [20] proposed to use neural networks to approximate the solution to differential equations. On the other hand, the key Nesterov optimization algorithm [1] in neural networks was proposed as early as in 1983. The PINN method itself is a collocation method. The usage of neural network is innovating. Since 2016, a thorough regular PINN method was proposed by Raissi [25]. Although the method performs well on problems with classical solutions, the regular PINN performs badly on discontinuous problems and the training process is stuck at bad local minima. In particular, when neural networks are used to approximate
solutions of hyperbolic conservation laws, they do not usually give satisfactory performances. We thus propose a new PINN-based algorithm that can accurately capture shock waves.

### 4.2 Non-Diffusive Algorithm

Consider the following Initial Value Problem, with \( f \in C^2 \) convex and \( u_0 \in L^1_{\text{loc}}(\mathbb{R}) \cap \text{BV}(\mathbb{R}), \Omega \subseteq \mathbb{R} \).

\[
\begin{align*}
u_t + f(u)_x &= 0, \quad \text{on } Q := \Omega \times [0, T], \\
u(\cdot, 0) &= u_0, \quad \text{on } \Omega,
\end{align*}
\]

along with appropriate boundary conditions depending on the sign of \( f'(u) \) at the boundary. For practical computations, the boundary conditions are directly implemented within the neural network.

The corresponding solution contains a shock wave initially located at say \( x = 0 \). The line of discontinuity is denoted by \( \gamma : t \to \gamma(t), \) with \( \gamma(0) = 0 \) with Lax entropy condition. Rankine-Hugoniot’s jump condition along the discontinuity for \( t \in [0, T] \), reads

\[
\gamma'(t)[u^+(\gamma(t), t) - u^-(\gamma(t), t)] = f(u^+(\gamma(t), t)) - f(u^-(\gamma(t), t)),
\]

and the Lax shock condition reads

\[
f'(u^-(\gamma(t), t)) > \gamma(t) > f'(u^+(\gamma(t), t)).
\]
Hence the line of discontinuity separates \( Q = \Omega \times [0,T] \), \( \Omega = [a,b] \) in two time-dependent subdomains denoted \( Q_T^- \) and \( Q_T^+ \) defined as follows

\[
Q_T^+ = \{ (x,s), s \in [0,T], \ x \in \Omega : x > \gamma(s) \} \\
= \left\{ (x,s), s \in [0,T], \ x \in \Omega_s^+: = \left\{ \frac{b-y}{b}(\gamma(s) - b) + b : y \in (0,b) \right\} \right\},
\]

\[
Q_T^- = \{ (x,s), s \in [0,T], \ x \in \Omega : x < \gamma(s) \} \\
= \left\{ (x,s), s \in [0,T], \ x \in \Omega_s^- : = \left\{ \frac{a-y}{a}(\gamma(s) - a) + a : y \in (a,0) \right\} \right\}.
\]

We can easily rewrite (4.2.1)-(4.2.2) in the form of a system of 2 equations

\[
u_t^+ + f(u^+) = 0, \quad \text{on } Q_T^+,
\]

\[
u^+(-,0) = u_0|_{\Omega}^+, \quad \text{on } \Omega_0^-,
\]

which are coupled through the Rankine-Hugoniot condition, for \( t \in [0,T] \)

\[
f(u^+(\gamma(t),t)) - f(u^-(\gamma(t),t)) = \gamma'(t)[u^+(\gamma(t),t) - u^-(\gamma(t),t)].
\]

The proposed approach consists in approximating the solutions in each subdomain with neural networks. We denote by \( N^\pm(x,t) \) the neural networks approximating \( u^\pm \) (with parameters \( \theta_u^\pm \)), and \( N_\gamma(t) \) the neural network approximating \( \gamma \) (with parameters \( \theta_\gamma \)). The global solution \( N \) is then constructed as \( N(x,t) = N^\pm(x,t) \) with \( (x,t) \in Q_T^\pm \). The Rankine-Hugoniot conditions are expressed in terms of \( N^\pm(x,t) \) and \( N_\gamma(t) \) as follows

\[
\partial_t N_\gamma(t)[N^+(N_\gamma(t),t) - N^-(N_\gamma(t),t)] = f(N^+(N_\gamma(t),t)) - f(N^-(N_\gamma(t),t)).
\]

Practically, the subdomain \( Q_T^\pm \) will be evaluated thanks to \( N_\gamma(t) \) in place of \( \gamma(t) \).
Denoting \( \theta = (\theta_{N^+}, \theta_{N^-}, \theta_{N_0}) \), the optimized parameters are obtained by solving the problem

\[
\text{find } \theta^* \in \Theta \text{ such that } L(\theta^*) = \min \{L(\theta), \theta \in \Theta_{\text{ad}}\},
\]

(4.2.3)

where

\[
L(\theta) = \lambda \left( \|\partial_t N^- (\cdot, \cdot) + \partial_x f(N^- (\cdot, \cdot))\|_{L^2(Q^-_0)}^2 \right) + \|\partial_t N^+ (\cdot, \cdot) + \partial_x f(N^+ (\cdot, \cdot))\|_{L^2(Q^+_0)}^2 \\
+ \mu \|\partial_t N_0(t)[N^+(N_0(t),t) - N^-(N_0(t),t)] - [f(N^+(N_0(t),t)) - f(N^-(N_0(t),t))]\|_{L^2(0,T)}^2 \\
+ \kappa \left( \|N^-(\cdot,0) - u_0(\cdot)\|^2_{L^2(Q^-_0)} + \|N^+(\cdot,0) - u_0(\cdot)\|^2_{L^2(Q^+_0)} \right),
\]

for some positive parameters \( \lambda, \mu \) and \( \kappa \), and \( \Theta_{\text{ad}} \) is the set of admissible weights.

Computationally, recall our objective is to find a group of functions: \( \gamma(t), u^-(x,t), u^+(x,t) \) that satisfies 4.2.1 and 4.2.2. We define the boundary points as points on the left, right and on the initial discontinuity curve as: \( Q^+_0 = (a,0) \times \{0\}, Q^-_0 = (0,b) \times \{0\}, \gamma_0 = (a,b) \times \{0\} \).

We use three neural networks to approximate \( u^+, u^-, \gamma \) respectively. Let \( x = (x,t) \), the networks are defined as \( N^+ = N^+(x; W^+, b^+), N^- = N^-(x; W^-, b^-) \), and \( N_0 = N_0(t; W^\gamma, b^\gamma) \). \( W^+, W^-, W^\gamma \) are the respective weights to the neural networks. \( N_\gamma : (0,T) \mapsto \mathbb{R} \) approximates the discontinuity curve \( \gamma \). Further denote the parameters in vector form as

\[
\theta_{N^+} = [W^+, b^+], \quad \theta_{N^-} = [W^-, b^-], \quad \theta_{N_0} = [W^\gamma, b^\gamma].
\]

We define two transformations to describe the moving effect of the initial
points.

\[ T^+ : Q_0^+ \mapsto Q^+, \quad T^- : Q_0^- \mapsto Q^- \]

We define the collocation points in different parts of domain as follow,

1. \( \{ z_i^l \} \): interior collocation points at the left of discontinuity curve, in \( Q^+ \).

2. \( \{ z_i^r \} \): interior collocation points at the right of discontinuity curve, in \( Q^- \).

3. \( \{ z_i^\gamma \} \): \( n_i^\gamma \) interior collocation points on the discontinuity curve, on \( \gamma_0 \).

4. \( \{ z_i^{\Gamma_l} \} \): \( n_i^{\Gamma_l} \) boundary collocation points at the left of discontinuity curve, in \( Q_0^+ \).

5. \( \{ z_i^{\Gamma_r} \} \): \( n_i^{\Gamma_r} \) boundary collocation points at the right of discontinuity curve, in \( Q_0^- \),

where \( n_i^{\Gamma_l} \) and \( n_i^{\Gamma_r} \) are integers.

Using the collocation points (training points) and networks, we can construct the total loss function \( L \) under the discrete \( l_2 \) norm approximated by Monte-Carlo. The loss function consists of four parts, and they are loss from the left
right domain, jump condition and initial condition respectively.

\[ L_l(x; W^+, b^+) = \frac{1}{n_{l1}} \sum_i \| \partial_t N^+(\cdot; T^+(z_l^i)) + f(N^+(\cdot; z_l^i)) \times \partial_t N^+(\cdot; z_l^i) \|_2^2, \]

\[ L_r(x; W^-, b^-) = \frac{1}{n_{r1}} \sum_i \| \partial_t N^-(\cdot; T^-(z_r^i)) + f(N^-(\cdot; z_r^i)) \times \partial_t N^-(\cdot; z_r^i) \|_2^2, \]

\[ L_\gamma(x; W^\gamma, b^\gamma) = \frac{1}{n_\gamma} \sum_i \left\| \partial_t N_\gamma(\cdot; z_\gamma) - \frac{f(N^+(\cdot; N_\gamma(z_\gamma^i))) - f(N^-(\cdot; N_\gamma(z_\gamma^i)))}{N^+(\cdot; z_\gamma^i) - N^-(\cdot; N_\gamma(z_\gamma^i))} \right\|^2_2, \]

\[ L_{ic} = \frac{1}{n_{l1}} \sum_i \| N^+(\cdot; z_{\Gamma_l}) - g(z_{\Gamma_l}) \|_2^2 + \frac{1}{n_{r1}} \sum_i \| N^-(\cdot; z_{\Gamma_r}) - g(z_{\Gamma_r}) \|_2^2, \]

\[ L = L_l + L_r + L_\gamma + L_{ic}. \]

The remaining part is to select an optimizer and minimize the total loss such that the parameters \( \theta \) for each of the three proposed neural networks can be determined.

### 4.3 Riemann Example: Capturing Shock Waves

Consider a Riemann problem 4.2.1 and 4.2.2. In this experiment we consider \( \Omega \times [0, T] = (-4, 3) \times [0, 3/4] \), with a concave flux \( f(u) = 4u(2 - u) \). Note that hyperbolic equation of conservation law with convex or concave flux functions have solutions with simple structure, such as classical shock waves and rarefactions waves. The initial data is given by \( u_0 \), and the corresponding characteristics are
defined by $x$, See Figure 4.1.

\[
\begin{align*}
u_0(x) &= \begin{cases} 
1, & x < -2, \\
\frac{1}{2}, & -2 < x < 0, \\
\frac{3}{2}, & 0 < x.
\end{cases} \\
x(t) &= \begin{cases} 
x_0, & x_0 \leq 1, \\
x_0 + 4t, & 1 < x_0 \leq 3, \\
x_0 - 4t, & x_0 > 3.
\end{cases}
\end{align*}
\]

Figure 4.1: (left) Initial Condition $u_0(x)$; (right) Characteristic Lines

### 4.3.1 Analytic Solution

The exact solution can be computed analytically. In the time interval $[0, 1/2]$ it is constituted by a rarefaction and a shock wave with constant velocity. Then, in the time interval $[1/2, 3/4]$ the initial shock wave interacts with the rarefaction wave to produce a new shock with non-constant velocity. More specifically the solution is given by

\[
\begin{align*}
u(x,t) &= \begin{cases} 
1, & x < -2, \\
1 + \frac{x + 2}{8t}, & -2 < x \leq -2 + 4t, \\
\frac{1}{2}, & -2 + 4t < x < 0, \\
\frac{3}{2}, & 0 < x.
\end{cases} \\
u(x,t) &= \begin{cases} 
1, & x < -2, \\
1 + \frac{\gamma(t) - 2}{8t}, & -2 < x \leq \gamma(t), \\
\frac{3}{2}, & \gamma(t) < x.
\end{cases}
\end{align*}
\]
where $\gamma$ is the line of discontinuity and it solves

$$f\left(\frac{3}{2}\right) - f\left(\frac{2 + \gamma(t)}{8t} + 1\right) = \gamma'(t)\left(\frac{1}{2} - \frac{\gamma(t) + 2}{8t}\right),$$

for $t \in [1/2, 1]$ and $\gamma(1/2) = 0$.

### 4.3.2 Traditional Numerical Solution: Godunov Solver

We present a solution given by a Godunov scheme [9], which is based on an exact Riemann solver. Consider a uniform discretization of the domain $[x_l, x_r]$, using the following nodes,

$$x_j = x_l + (j + \frac{1}{2})\Delta x, \quad j = 0, 1, \ldots N, \quad \Delta x = \frac{x_r - x_l}{N+1}$$

We define the mid-points as follows,

$$x_{j-\frac{1}{2}} = x_j - \frac{\Delta x}{2} = x_l + j\Delta x, \quad j = 0, 1 \ldots N$$

The finite volume method uses computational cells $C_j = (x_{j-\frac{1}{2}}, x_{j+\frac{1}{2}})$, instead of mesh points $x_{j-\frac{1}{2}}$ as in finite difference scheme. Each time step discretization is $t_n = n\Delta t, \ n \geq 0$. Finite volume methods approximate the cell average at each time $t_n$:

$$u^n_j = \frac{1}{\Delta x} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x, t_n) \, dx$$

In order to obtain the cell averages $u(x, t_{n+1})$ at the next time $t_{n+1}$, we proceed
as follows. We integrate the equation over domain $(x_{j-\frac{1}{2}}, x_{j+\frac{1}{2}}) \times [t_n, t_{n+1}]$, i.e.

$$\int_{t_n}^{t_{n+1}} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u_t + f(u)_x \, dx \, dt = 0.$$ 

Using the fundamental theorem of calculus, this can be rewritten as

$$\int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x, t_{n+1}) - u(x, t_n) \, dx = \int_{t_n}^{t_{n+1}} f(u(x_{j-\frac{1}{2}}, t)) - f(u(x_{j+\frac{1}{2}}, t)) \, dt$$

Setting $F^{n}_{j+\frac{1}{2}}$ the flux as follow,

$$F^{n}_{j+\frac{1}{2}} = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(u(x_{j+\frac{1}{2}}, t)) \, dt,$$

and divide by $\Delta x$, we obtain

$$u^{n+1}_{j} = u^{n}_{j} - \frac{\Delta t}{\Delta x} (F^{n}_{j+\frac{1}{2}} - F^{n}_{j-\frac{1}{2}}),$$

which is a statement of conservation. The Godunov method approximates the flux $F^{n}_{j+\frac{1}{2}}$ at each $x_{j+\frac{1}{2}}$. In each cell $C_j$ a Riemann problem $u_t + f(u)_x = 0$ is indeed defined by

$$u(x, t_n) = \begin{cases} 
  u^n_j, & x < x_{j+\frac{1}{2}}, \\
  u^n_{j+1}, & x > x_{j+\frac{1}{2}}.
\end{cases}$$

We denote $F, u$ as the exact flux and solution, $G, v$ as their corresponding ap-
proximations. The Godunov approximation $G_{j+\frac{1}{2}}$ to the flux $F_{j+\frac{1}{2}}$ is thus

$$F_{j+\frac{1}{2}} \approx G_{j+\frac{1}{2}} = \begin{cases}
\min_{u_j^n \leq \theta \leq u_{j+1}^n} f(\theta), & f'(u_j^n) \leq f'(u_{j+1}^n) \\
\max_{u_{j+1}^n \leq \theta \leq u_j^n} f(\theta), & f'(u_{j+1}^n) \leq f'(u_j^n).
\end{cases}$$

Hence the Godunov scheme reads

$$v_{j+1}^n = v_j^n - \frac{\Delta t}{\Delta x} (G_{j+\frac{1}{2}} - G_{j-\frac{1}{2}}).$$

Next, we propose some experiments based on Riemann problem, using the regular PINN method, our method, and the Godunov scheme.

### 4.3.3 Numerical Experiments

**Experiment 1** We introduce 2 subdomains $\Omega_0^1 = (-4, 0)$ and $\Omega_0^2 = (0, 3)$. Notice that for NN-algorithm, we use instead a regularized discontinuity for the non-entropic discontinuity located at $x = -1$. We introduce 3 neural networks, two space-time dependent and one time dependent, with 2 hidden layers and 20 neurons each. In both experiments, the selection of network parameters are empirical and we use the same network size for simplicity and consistency. We initialized the weights and bias using Xavier technique and selected a large epoch number 20000 to ensure the training quality. We report the (exact) solution of reference Figure 4.2 (top left), the direct PINN solution Figure 4.2 (middle), the neural network solution Figure 4.2 (top right), and the loss function as function of the epoch number, See Figure 4.3.

At the beginning of the training, the loss decreases fast. Notice that after
around 20000 epochs, the loss value stopped decreasing because the minimum is approached. Thus, the loss curve looks like a jump at 20000 epochs. The small fluctuation occurs due to the stochastic behaviour of the optimizer. In practice, SGD variants are used for training, which contain the batch-training technique, i.e. at each step, the gradients are calculated using different samples. Thus, there can be naturally small fluctuations of the loss value. If gradients are calculated using the entire data set, like in GD, then this phenomena can be avoided.

Let us mention that using the same neural data, a direct PINN algorithm provides a very inaccurate solution, unlike our algorithm.
CHAPTER 4. A NON-DIFFUSIVE PINN-BASED ALGORITHM

Experiment 2 We consider now \( \Omega \times [0,T] = (-1,2) \times [0,0.5], \) \( f(u) = u^2/2, \) and

\[
    u_0(x) = \begin{cases} 
        1, & -1 < x < 0, \\
        9/10, & 0 < x < 1, \\
        0, & 1 < x < 2. 
    \end{cases}
\]

We compute the solution with three initial subdomains: \( \Omega^1_0 = (-1,0), \) \( \Omega^2_0 = (0,1), \) \( \Omega^3_0 = (1,2). \) Note that the initial condition is constituted by two entropic shock waves. On Figure 4.5 we report the loss function as function of epoch and the space-time neural network solution at \( T = 0.5. \) We notice that unlike the Godunov scheme which naturally produces some numerical diffusion on the slowest shock waves (even at CFL = 0.99), the neural network solution is diffusion-free. This is an interesting property which is more generally not shared with standard hyperbolic equation solvers.
4.4 Discussion

PINN methods show very good results at least for problems with smooth solutions. As deep learning packages grow rapidly, the neural network method is a trending contribution that can help data-driven forecasting of physical processes, model predictive control, multi-physics modelling and simulations. However, PINN also has disadvantages. The regular PINN is a direct solver, which does not comply with variational forms. Furthermore, there is no guarantee to converge to the global minimum.
Figure 4.5: Experiment 2. Space-time solution.

Figure 4.6: Experiment 2. Godunov at CFL=0.99 and neural network solution at time $T = 0.5$. 
Chapter 5

Numerical Computation

5.1 Existing Libraries: Computational Aspects

Physics-informed neural networks is a new class of universal function approximators which is capable of encoding underlying physical laws that govern a given data-set, and can be described by partial differential equations [25]. There are several packages developed specifically for PINN [16]. Because of the uniqueness of each loss function and domain of PDEs, researchers prefer hard coding certain problems so the existing packages are sometimes rigid to customize. Different packages have different ways to handle boundary conditions, either as hard constraint or soft constraint. In the soft constraint case, if boundary conditions are not embedded in the neural network but are included in the loss, multiple losses’ optimization leads to increased number of parameters to train. In contrast, if the boundary conditions are embedded in the network, there are less parameters to train, as we showed in Lagaris’ trial solution examples in Chapter 3.
Among the existing libraries, DeepXDE from Karniadakis’s CRUNCH group at Brown University is the most popular one. This library includes residual-based adaptive refinement (RAR), a strategy for optimizing the distribution of residual points during the training stage which is comparable to FEM refinement approaches. The package has five applications in its first version in 2019, including Poisson equation across a L-shaped domain, 2D Burgers’ equation, first-order Volterra integral equation, inverse Problem for the Lorenz System, and diffusion–reaction systems [5].

There are alternative libraries, such as Peng’s IDLRnet [24], and Viana’s PML-PINN [34] developed recently. But they have less customization features compared to DeepXDE. For instance [5], the Neurodiff library enforces hard constraints to incorporate boundary conditions. SciANN is an implementation of PINN as a high-level Keras wrapper, suitable for replicating existing problems. PyDENs is a solver for heat and wave equations, with hard constraints as well. Moreover, after the first release in 2019, the development appears to have stopped in 2020.

DeepXDE is too rigid to implement an algorithm different from the regular PINNs, so we used Jax [3] to implement our proposed algorithm. Although DeepXDE’s customization level can be improved, which relies on the corresponding machine learning packages’ update, overall, I still recommend DeepXDE as the priority package to use. Not only because it has stable development and maintenance, but also for the soft constraints and different available backends including Tensorflow, Pytorch and Jax.
Table 5.1: PINN packages [16]

<table>
<thead>
<tr>
<th>Software</th>
<th>Usage</th>
<th>Language</th>
<th>Backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deepxde</td>
<td>Solver</td>
<td>Python</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>SimNet</td>
<td>Solver</td>
<td>Python</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>PyDEns</td>
<td>Solver</td>
<td>Python</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>NeuroDiffEq</td>
<td>Solver</td>
<td>Python</td>
<td>Pytorch</td>
</tr>
<tr>
<td>NeuralPDE</td>
<td>Solver</td>
<td>Julia</td>
<td>Julia</td>
</tr>
<tr>
<td>SciANN</td>
<td>Wrapper</td>
<td>Python</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>ADCME</td>
<td>Wrapper</td>
<td>Julia</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>GPyTorch</td>
<td>Wrapper</td>
<td>Python</td>
<td>Pytorch</td>
</tr>
<tr>
<td>Neural Tangents</td>
<td>Wrapper</td>
<td>Python</td>
<td>Jax</td>
</tr>
</tbody>
</table>

5.2 Practical Implementation for Hyperbolic Equations

When implementing PINN, we need to address each of the PINN modules. From neural network architecture options, activation function type, optimizer choice to boundary conditions’ space constrains. Because our algorithm requires domain decomposition and transforming collocation points, the implementation is different from regular PINN implementation via an existing package such as DeepXDE. We selected Jax [3], a machine learning framework developed by Google and constructed our network from scratch with Adam as selected optimizer.

Practically, L-BFGS performs better, with less iterations, than Adam when solving PDEs with a smooth solution [30], as L-BFGS uses a second derivative of loss function. However, L-BFGS tends to stuck at a bad local minimum when it comes to stiff problems. For standard PINN, convergence has not been rigorously proven, therefore, there is obviously no unique solution, which are obtained from
non-convex optimization.
Bibliography


BIBLIOGRAPHY


Appendix A

Python code repository

GitHub: https://github.com/Barnett888/PINN_1d_conservation_law

Jax documentation: https://jax.readthedocs.io/en/latest/
Appendix B

PINN library documentation

Deepxde: https://deepxde.readthedocs.io/en/latest/

NeuroDiffEq: https://neurodiffeq.readthedocs.io/en/latest/intro.html