AN ANALYSIS OF PHYSICS-INFORMED NEURAL NETWORKS

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

2021

Edward Small
Department of Mathematics

Contents

Word count 13637

List of Tables

List of Figures

Abstract

Whilst the partial differential equations that govern the dynamics of our world have been studied in great depth for centuries, solving them for complex, high-dimensional conditions and domains still presents an incredibly large mathematical and computational challenge. Analytical methods can be cumbersome to utilise, and numerical methods can lead to errors and inaccuracies. On top of this, sometimes we lack the information or knowledge to pose the problem well enough to apply these kinds of methods.

Here, we present a new approach to approximating the solution to physical systems - physics-informed neural networks. The concept of artificial neural networks is introduced, the objective function is defined, and optimisation strategies are discussed. The partial differential equation is then included as a constraint in the loss function for the optimisation problem, giving the network access to knowledge of the dynamics of the physical system it is modelling.

Some intuitive examples are displayed, and more complex applications are considered to showcase the power of physics informed neural networks, such as in seismic imaging. Solution error is analysed, and suggestions are made to improve convergence and/or solution precision. Problems and limitations are also touched upon in the conclusions, as well as some thoughts as to where physics informed neural networks are most useful, and where they could go next.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual Property Statement

A List of Notation and Abbreviations

0.1 Notation

$\alpha, \beta, \gamma, ...$	Lowercase Greek letters used for scalars $\mathbb{R}$
a, b, x, y, z	Used for vectors $\mathbb{R}^n$
x_i	A generic element inside the vector x
x_k	State of vector x in iteration k of an algorithm
$\hat{y}$	An estimate of the true output y
A, B, C, W, M	Used for matrices $\mathbb{R}^{n \times m}$ or sets
$A_{i,j}$	A submatrix inside the matrix A
f, g, ϕ, σ	Functions mapping one set to another
f_t, ϕ_{yy}, u_x	Functions that have been differeniated w.r.t. the subscript
f_n, ϕ_n	The nth function in a series or set
F, G	ANN approximation the lowercase functions f, g
L	Number of layers in a network
l	A specific layer in a network
N	The number of elements in a set or series
$n^{[l]}$	A specific neuron in a specific layer
$\mathcal{L}$	Loss function
$\|\|x\|\|_p$	vector norm $\left(\sum_i \|x_i\|^p\right)^{\frac{1}{p}}$
$\mathcal{J}$	The Jacobian
$\mathcal{H}$	The Hessian
$\mathcal{N}$	A differential operator
Λ	A physics-informed neural network
θ	Values that parameterise a ANN, such as weights and biases

0.2 Abbreviations

AD	Automatic Differentiation
ADAM	Adaptive Moment Estimation
ANN	Artificial Neural Network
AutoGrad	Adaptive Stochastic Gradient Descent
BC	Boundary Condition
CPU	Central Processing Unit
FD	Finite Difference
FEA	Finite Element Analysis
FWI	Full Waveform Inversion
GPU	Graphics Processing Unit
IBM	International Business Machine
IC	Initial Condition
KdV	Korteweg–De Vries
ML	Machine Learning
MRI	Magnetic Resonance Imaging
MSE	Mean Squared Error
NLP	Natural Language Processing
L-BFGS	Limited memory Broyden–Fletcher–Goldfarb–Shanno
LU	Lower Upper Factorisation
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
PINN	Physics-informed Neural Network
RMSProp	Root Mean Square Propogation
SAS	Statistical Analysis Software
SGD	Stochastic Gradient Descent
s.t.	Such that
TV	Total Variation
WRI	Wavefield Reconstruction Inversion
w.r.t	With respect to

Acknowledgements

I would like to express my immense gratitude to my supervisor Dr Oliver Dorn for the vast amount of experience and wisdom he supplied whilst writing this thesis. He was a fantastic source of inspiration for much of my time at the University of Manchester, and his guidance was crucial for the research I have conducted. My enjoyment of his lecture course in numerical optimisation and inverse problems was what originally motivated me to study this topic.

Of course, I also have to think about those who were immediately around me. Whether it was my mother Annette, who would bring me the occasional coffee when I was deep in thought, my father Graham, who always had time to listen to me, or my friend Matt, who was a good reminder that taking a break can be a good thing, I will be forever grateful. I especially appreciated them nodding along as I excitedly explained what I was researching, despite none of them really understanding what I was saying.

And finally, I have to thank one particularly special person who couldn't be around me, but still offered me constant, endless support - my partner Jess. Australia has never felt so far away as it has for the last 2 years, but here is hoping we will see each other soon.

Chapter 1

Introduction

Machine Learning is a relatively young and fast developing branch of mathematics. Whilst the first mention of machine learning wasn't until 1952 by IBM computer scientist Arthur Samuel (who was developing complex algorithms for computers to effectively play checkers [1]), the foundations of machine learning are scattered throughout modern history, with its inception standing on the shoulders of many well-known mathematicians, such as Laplace, Markov [10], Bayes [6], and Turing [9], to name a few. It poses the elegant question *'what if my computer could learn?'* That is to say, how could one create an algorithm that was dynamic in its problem solving approach - an algorithm that could learn from its mistakes, and iteratively adapt and evolve to become more accurate?

Modern mathematicians have spent decades refining these learning processes and, coupled with the age of *Big Data*[1], these algorithms have been put to incredible use. From facial recognition software [11], to self-driving cars [12] and targeted advertisement [13], machine learning algorithms have consumed vast amounts of data to create discrete solutions to problems that no human ever could.

One recent advancement in a sub-branch of machine learning, artificial neural networks (ANNs), is to use the known the physics of a system as a constraint for optimising the learning process, ultimately finding approximate solutions to partial differential equations (PDEs). Whilst this process is somewhat complex, it can not only lead to more accurate results, but also achieve these results at a higher computational speed. These special types of network constraints are known as *physics-informed neural networks* (PINNs).

[1]SAS, one of the largest data management and analytics companies in the world, defines *Big Data* as datasets that are so large, fast (large amounts of new data generated per time step) or complex (highly variable with high veracity) that analysing the data cannot be done through traditional statistical methods. This kind of data poses a unique challenge in terms of processing and storage.

1.1 What is an Artificial Neural Network?

PINNs are a subtype of ANNs, and so before learning about PINNs it is important to have a basic understanding of ANNs - how they process data, produce results, and how they learn.

1.1.1 The Network

As the name suggests, ANNs are heavily inspired by the basic operations and understanding of a neuron - the nerve cell that is the building block of the nervous system. A single artificial neuron is known as a *perceptron*[2]. A perceptron takes a set of inputs $x \in \mathbb{R}^n$ and gives a single output $y \in [-1, 1]$, where y is a measure of 'how active' the perceptron is. A perceptron achieves this by applying 3 fundamental steps to the inputs:

1. Sum the product of each input $x_i \in x$ with a corresponding weight $w_i \in w$. Each weight w_i can be interpreted as a measure of how sensitive a perceptron is to each individual input x_i. As an example, if $w_i = 0$ then the input x_i has no influence on the activation of the perceptron.

2. Add a bias β to this sum. This bias can be interpreted as a measure of how active a neuron would be if $x_i = 0$ for $i = 1, ..., n$.

3. Put this value through an activation function $\sigma : \mathbb{R} \to [-1, 1]$, such as tanh.

The activation of a perceptron a can therefore be represented as

$$a = \sigma\left(\sum_{i=1}^{n}(x_i w_i) + \beta\right) \tag{1.1}$$

which visually looks like

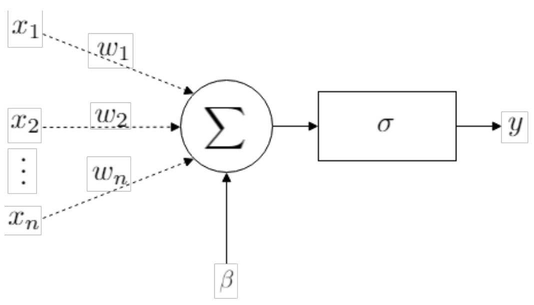

Figure 1.1: Visual representation of a single perceptron taking in n inputs.

Intuitively then, an ANN F (which approximates f) is just a collection of these perceptrons feeding into each other [14], the activation of previous neurons influencing the activation of others such that $F : x \to \hat{y}$. Typically, an ANN is organised into $l = 1, 2, ..., L$ layers, where each layer has $n^{[l]}$ number of neurons. Any layer between the input layer $n^{[1]}$ and the output layer $n^{[L]}$ is usually referred to as a *hidden layer* because its state is not accessible by a user [15]. The user only has access to the input layer and output layer.

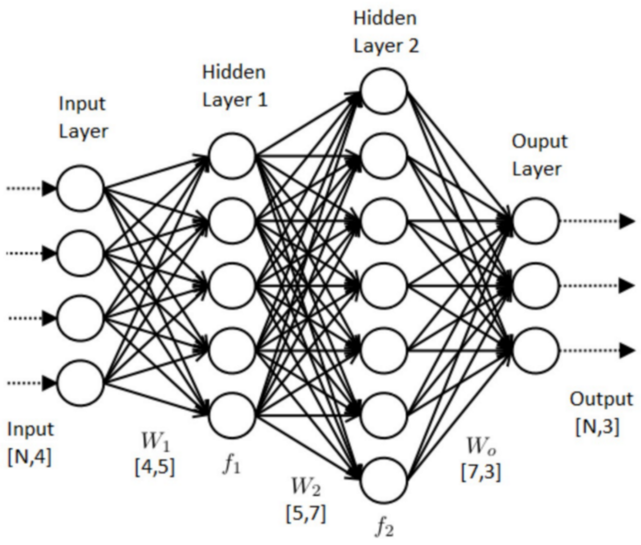

Figure 1.2: An ANN with $x \in \mathbb{R}^4$ inputs, 2 hidden layers, and an output layer $\hat{y} \in \mathbb{R}^3$

For $l = 2, ..., L - 1$ the activation $a^{[l]}$ is therefore

$$\begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{n^{[l]}}^{[l]} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n^{[l-1]}} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n^{[l-1]}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]},1} & w_{n^{[l]},2} & \cdots & w_{n^{[l]},n^{[l-1]}} \end{bmatrix} \begin{bmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_{n^{[l-1]}}^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix} \right) \tag{1.2}$$

which can be condensed into

$$a^{[l]} = \sigma(W^{[l]} a^{[l-1]} + b^{[l]}) \tag{1.3}$$

where $a^{[l]}, b^{[l]} \in \mathbb{R}^{n^{[l]}}$ and $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$. This is the key to how ANNs propagate information through the system to find a mapping for $f : x \to y$.

Neurons can be connected in many different ways to create special types of networks, such as *convolutional networks*[2] or *recurrent networks*[3]. Figure 1.2 is an example of a *fully connected* ANN, as every node in layer $l - 1$ connects to every node in l. For simplicity, we will mostly consider fully connected, feedforward ANNs, unless otherwise specified.

[2]A convolutional network has a sense of locality or grouping in the input data in such a way that not every neuron in a layer connects to every neuron in the next.

[3]A recurrent network retains some memory of previous inputs and outputs in such a way that previous entries can influence future results.

An important result from ANNs is that a network F with a single hidden layer can approximate any continuous function f within a region I_n (which is n-dimensional) to an arbitrary precision $\epsilon > 0$ such that

$$|F(x) - f(x)| < \epsilon \qquad \forall \quad x \in I_n \tag{1.4}$$

provided that no limit is placed on the values of W and b (the weights and biases of the ANN F). The precision, therefore, is dependent on the amount of neurons. This is known as the *universal approximation theorem*[8].

1.2 What is a Partial Differential Equation?

A partial differential equation (PDE) is a general equation that describes the relationship between rates of changes between variables in a multi-variable function [16]. A famous example is the 1-dimensional heat equation, which describes the heat flux on a rod with 1 dimension, x, through time, t, such that

$$u_t = ku_{xx} \tag{1.5}$$

where $u_t = \frac{\partial u}{\partial t}$ is the change in temperature through time, $u_{xx} = \frac{\partial^2 u}{\partial x^2}$ is the spatial change in the temperature through space, and k is a convective term that describes how easily energy can flow in/out of the rod [17]. The equation relates the change in temperature over time at a specific point in space and time to the change in temperature through space at a specific point in space and time.

PDEs are used generally to describe all kinds of physcial systems, such as acoustics, diffusion, electromagnatism, fluid dynamics, and even quantum mechanics [18], so finding solutions to these equations (given certain initial conditions (ICs)[4] and boundary conditions (BCs)[5]) is an incredibly important mathematical endeavour. Unfortunately, finding solutions to PDEs of real physical phenomena can be incredibly challenging for a multitude of reasons.

There are two main types of solutions: *analytical solutions* and *numerical solutions*. Analytical solutions can be thought of as closed-form solutions that match both the PDE, and the ICs and BCs of a problem, sometimes uniquely (depending on how the problem is defined). Solving a PDE analytically can be an incredibly time-intensive task, involving classifying the PDE and then utilising intricate methods, such as periodic extensions, Fourier series [19], and scattering [20]. Even then, there is no guarantee that the solution will be easy to use, and sometimes the general solution is

[4]If we know the initial conditions then we know the general shape of the function for all variables at time $t = 0$

[5]If we know the boundary conditions, we know what the behaviour of the function should be for all time t at the extreme parts of the domain of interest.

in the form of a complex integral, or an infinite series

$$\phi(x,t) = \sum_{n=1}^{\infty} \alpha_n \phi_n(x,t) \tag{1.6}$$

where each ϕ_n is a solution to the PDE. Thus, even if $\alpha_n \to 0$ as $n \to \infty$ rapidly, we must accept that we can only estimate the true solution to an arbitrary precision.

Finding analytical solutions in real-world modelling is incredibly challenging for other reasons too. These challenges include not knowing the exact ICs and BCs of a problem, or having a data-set that is too sparse, or a domain that is too complex to accurately define in a usable way. The problem deepens considerably when one considers the fact that some widely used PDEs, such as the famous Navier-Stokes (NS) equations describing the motion of fluid, are actually not well understood. In fact, we do not even know if smooth solutions always exist for 3 dimensional fluid flow, and turbulence and singularities still pose a huge problem [21]. Since the NS equations govern the dynamics of physical phenomena like atmospheric dynamics, and therefore weather predictions, it should be clear why study of these equations is paramount.

Numerical methods can help with this, such as finite difference (FD) and finite element analysis (FEA). FD involves estimating the value of derivatives to estimate the value of the function at the next point in space and time, whereas FEA uses variational methods to produce an approximate solution to the boundary conditions across individual, easier to solve elements, and then stitching the solution together using interpolation.

These techniques can have their own issues. For example, the errors in FD methods are propagated into the next iteration, and so using them for large domains for large time is somewhat challenging [22]. FEA can have continuity issues when crossing over elements, called flux jumps, that lead to energy loss [23]. Accurate solutions for either method in complex domains often relies on lot of computing power, and if we want to know the value of a function at a particular point in space and time, we often need to know the value of the surrounding points in space and time - something that is not necessary for an analytical solution.

Chapter 2

Training a Network to Approximate a Function

The goal of an ANN may be to replicate the behaviour of an unknown function $f : \mathbb{R}^n \to \mathbb{R}^m$. In machine learning there are three main types of learning processes

1. Supervised learning.
2. Unsupervised learning[1].
3. Reinforcement learning[2].

Supervised learning is used when a labelled data set is available [25]. That is to say, whilst the true function f may be unknown, we do have access to a set of inputs $X \in \mathbb{R}^{n\times p}$ and matched outputs $Y \in \mathbb{R}^{m\times p}$ where, if $x \in X$ is a column in X and $y \in Y$ is a corresponding column in Y, then

$$f(x) = y \tag{2.1}$$

If an ANN approximating f is called F, then we can say that the estimated output of F for the data set $x \in X$ is called $\hat{y} \in \hat{Y}$, such that

$$F(x) = \hat{y} \tag{2.2}$$

The goal of an ANN, therefore, is to minimise the *loss function* [59] $\mathcal{L}$ over all inputs and outputs

$$\min_{\theta} \mathcal{L}(\theta) = \min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\text{loss}}(y, F(x, \theta)) \tag{2.3}$$

[1]Unsupervised learning is used when the inputs for a function are known, but the corresponding outputs are not. The goal, therefore, is usually to find classes, groups or structures in the input data. This type of learning is often used in *dimension reduction*

[2]Reinforcement learning is used when a network learns through a penalty/reward system. Given a certain goal the network should optimise itself to produce the largest reward with the smallest penalty

where θ are the trainable parameters that dictate the behaviour of F, such as the weights and biases. There are many different ways to measure the loss, (hinge loss, contrast loss, entropy, etc), but the simplest to understand is minimising the square of the euclidean norm of the outputs [26], so

$$\begin{aligned} \mathcal{L}_{\text{loss}}(y, F(x,\theta)) &= \sqrt{(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + ... + (y_n - \hat{y}_n)^2}^2 \\ &= ||y - \hat{y}||_2^2 \end{aligned} \tag{2.4}$$

meaning that a minimum is found when there is no change that can be made to the parameters θ that reduces the magnitude of the difference between the function output and the ANN output for the same input.

Clearly, then, finding a minimum set of parameters θ is somewhat complex. If an ANN has L layers, each with the largest layer having m nodes into n nodes with $n \geq m$, then each pass through the network is approximately $\mathcal{O}(L(nm + n))$. We know $L \leq n$, and so, if $m = n$, a forward pass scales as $\mathcal{O}(n^2)$, as the largest cost is the matrix-vector multiplication (which is completed for every connection). As an example, for the simple ANN described in figure 1.2, going from hidden layer 1 to hidden layer 2 requires 42 calculations in of itself, so it is easy to see how this cost can increase dramatically as the size of the ANN increases.

Instinctively, if the the scenario occurs where $L = n$ then we must only have one neuron in each layer, and thus we have n amounts of scalar multiplications and additions, still leading to $\mathcal{O}(n^2)$.

Training the network is where most of the expense appears. In the worst case scenario, we may require n iterations using n data points to shift $n^2 + n$ weights and biases, giving a time complexity of $\mathcal{O}(n^4)$. This is why it is important to use intelligent strategies when designing and training an ANN.

2.1 Optimisation Algorithms

Today, there exist many methods that one could use to minimise equation 2.3. For linear problems, conjugate gradient methods can find the minimum in n steps, where n is the dimension of the input data [27]. Different methods require different knowledge of the function, such has *Jacobians*[3] and/or *Hessians*[4], but depending on what is known about the function, different strategies can be employed to find a minimum numerically. Some methods even use estimates of the Hessian and/or inverses of matrices, which can speed up computation time [28].

[3]If $x \in \mathbb{R}^n$ are inputs and $f : x \to y \in \mathbb{R}^m$, then the Jacobian $\mathcal{J} \in \mathbb{R}^{m \times n}$ of f is a matrix that holds the partial derivatives of each output of f_i with respect to each input x_j, so $\mathcal{J}_{ij} = \frac{\partial f_i}{\partial x_j}$

[4]If $x \in \mathbb{R}^n$ are inputs and $f : x \to y \in \mathbb{R}$, then the Hessian $\mathcal{H} \in \mathbb{R}^{n \times n}$ of f is a matrix that holds the 2nd partial derivatives and mixed derivatives of f with respect to each pair inputs $\{x_i, x_j\}_{i,j=1}^n$, so $\mathcal{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$

Non-linear problems, such as minimising the loss function for an ANN, are a little more complicated, but numerical methods do exist for these problems (such as Gauss-Newton) [29]. Some methods include *stochastic gradient descent* (SGD), the *limited-memory Broyden–Fletcher–Goldfarb–Shanno* (L-BFGS) algorithm (which is a second order, quasi-newton method, as it uses an estimate of the inverse of the second order derivatives (Hessian) to choose an intelligent search direction), and the ADAM optimiser (which combines two minimisation methods to accelerate learning).

2.1.1 Stochastic Gradient Descent

Normal gradient descent operates by initialising a set of random weights and biases θ_0 and finding the gradient of the loss function across all data points with respect to all parameters for each iteration. We then update θ by moving in this direction by a magnitude equal to either a pre-selected value, called the training parameter γ, or a more specific value that requires calculation, called step-size α [30].

$$\theta_{k+1} = \theta_k + \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x_i; \theta_k) \tag{2.5}$$

However, if the data-set n is very large then each step is computationally taxing. To save on this cost, we can divide the data into subsets and minimise randomly over each of these smaller subsets instead, called stochastic gradient descent (SGD) [31]. This increases iteration speed significantly. There are two main types of SGD

1. Full SGD, where each iteration minimises the objective function over an individual data point $x_i \in x$
$$\theta_{k+1} = \theta_k + \gamma \nabla_\theta \mathcal{L}(x_i; \theta_k) \tag{2.6}$$

2. Mini-batching, where $n_s \ll n$ number of data points are minimised over per iteration, so
$$\theta_{k+1} = \theta_k + \gamma \frac{1}{n_s} \sum_{i=1}^{n_s} \nabla_\theta \mathcal{L}(x_i; \theta_k) \tag{2.7}$$

Mini-batching can be done randomly so that each subset of x is randomly generated per iteration, or the data is pooled into separate subsets before training.

2.1.2 ADAM Optimisation

Adaptive Moment Estimation (ADAM) algorithm combines *adaptive gradident descent* (AdaGrad) with *Root Mean Square Propogation* (RMSProp) [32]. The main driving force behind Adam is that descent direction takes into account the current momentum [33], and the step size is carefully generated to create a *trust region*, outside of which we cannot be sure what the shape of the objective function is.

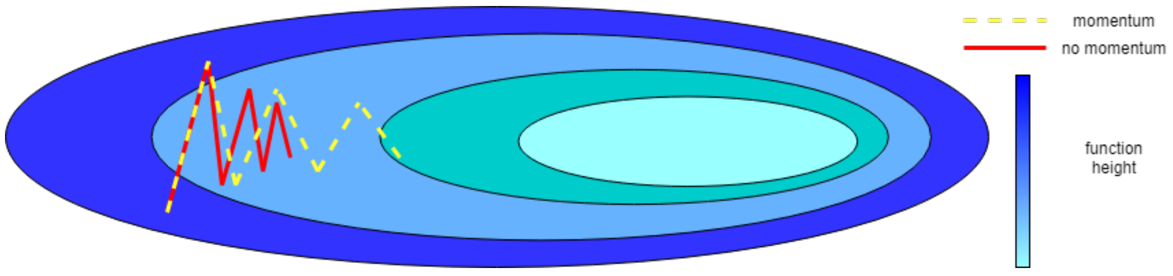

Figure 2.1: Visual representation of how momentum effects descent direction

ADAM calculates momentum by using moving averages of the gradient and the squared gradient, with hyper parameters β_1 and β_2 controlling the exponential decay.

Algorithm 1 ADAM optimisation

```
Require: $\alpha$                                        ▷ Step size
Require: $\epsilon$                                      ▷ Threshold for convergence
Require: $\beta_1, \beta_2 \in [0, 1)$                   ▷ Exponential decay rates for moment estimates
Require: $\mathcal{L}$                                   ▷ Objective function
Require: $\theta_0$                                      ▷ Initial parameters
  $m_0 \leftarrow 0$                                     ▷ 1st moment vector of 0s
  $v_0 \leftarrow 0$                                     ▷ 2nd moment vector of 0s
  $i \leftarrow 0$                                       ▷ Set up iteration counter
  $\delta \leftarrow 10^{-8}$                            ▷ Prevents division by 0
  while $\mathcal{L}(\theta_i) > \epsilon$ do
    $i \leftarrow i + 1$                                 ▷ Update iteration counter
    $\mathcal{J}_i \leftarrow \nabla_{\theta_{i-1}}(\theta_{i-1})$   ▷ Find gradients
    $m_i \leftarrow \beta_1 m_{i-1} + (1 - \beta_2)\mathcal{J}_i$   ▷ Update biased 1st order estimate
    $v_i \leftarrow \beta_2 v_{i-1} + (1 - \beta_1)\mathcal{J}_i \odot \mathcal{J}_i$   ▷ Update biased 2nd moment estimate
    $\hat{m}_i \leftarrow \frac{m_i}{1-\beta_1^i}$       ▷ Bias-corrected 1st order estimate
    $\hat{v}_i \leftarrow \frac{v_i}{1-\beta_2^i}$       ▷ Bias-corrected 2nd order estimate
    $\theta_i \leftarrow \theta_{i-1} - \alpha\frac{\hat{m}_i}{\sqrt{\hat{v}_i}+\delta}$   ▷ Update parameters
  end while
  return $\theta_i$
```

Usual values for the hyper parameters are $\beta_1 = 0.9$, and $\beta_2 = 0.999$, with $a \odot b$ representing the element wise multiplication between vectors a and b. The step size is bounded by a region around the current point, outside of which we lack information to know gradients accurately.

2.1.3 L-BFGS Optimisation

For large-scale problems with many parameters, it is important to keep in mind that we cannot operate on unlimited memory. The larger the number of parameters, the larger the Jacobian $\mathcal{J}$ and Hessians $\mathcal{H}$ grow. L-BFGS limits memory usage by only considering the past m updates when estimating the Hessian for the kth step $\mathcal{H}_k$ [34].

The BFGS algorithm is as follows

Algorithm 2 BFGS

```
Require: $\theta_0$                                   ▷ Initial guess for optimal parameters
Require: $\epsilon$                                   ▷ Tolerance for minimisation
Require: $\mathcal{H}_0$                              ▷ First estimate of Hessian (usually $I$)
Require: $\mathcal{L}$                                ▷ The objective function to minimise
  $k \leftarrow 0$
  while $\mathcal{L}(\theta_k) > \epsilon$ do
    $p_k \leftarrow -\mathcal{H}_k^{-1}\nabla\mathcal{L}(\theta_k)$                ▷ Find search direction
    $\alpha_k \leftarrow \min_\alpha \mathcal{L}(\theta_k + \alpha p_k)$          ▷ Perform line search (exact or inexact)
    $\theta_{k+1} \leftarrow \theta_k + \alpha_k p_k$                    ▷ Update parameters
    $y_k = \nabla\mathcal{L}(\theta_{k+1}) - \nabla\mathcal{L}(\theta_k)$      ▷ Find difference between gradients
    $\mathcal{H}_{k+1} \leftarrow \mathcal{H}_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{\mathcal{H}_k s_k s_k^T \mathcal{H}_k^T}{s_k^T \mathcal{H}_k s_k}$   ▷ Update Hessian estimate
  end while
  return $\theta_k$
```

and so

$$\theta_{k+1} = \theta_k + \mathcal{H}_k^{-1}\frac{1}{n}\sum_{k=i}^{n}\nabla_\theta\mathcal{L}(x_i;\theta) \tag{2.8}$$

If $\mathcal{H}_0 = I$, the identity matrix, then the first iteration is equivalent of steepest descent. The inverse of the estimated Hessian can be found directly by considering the *Sherman-Morrison* formula [38], so

$$\mathcal{H}_{k+1}^{-1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right)\mathcal{H}_k^{-1}\left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k} \tag{2.9}$$

Inexact line search is used to find an α that *sufficiently reduces* $\mathcal{L}(\theta_k + \alpha p_k)$, rather than finding the exact minimum, as this is computationally cheaper. Fundamentally, the learning parameter that is used in gradient descent methods is here replaced with an estimated inverse Hessian, which gives a much more intelligent descent direction. This is because we have approximate knowledge of the gradients and the change in the gradients. If $\mathcal{H}_k^{-1}$ is the exact inverse Hessian, and the algorithmic cost is quadratic, this algorithm can reach the minimum in a single step.

2.1.4 Inexact Line Search

A step size that 'sufficiently reduces' the objective function is usually defined in accordance with the *Armijo rule* [36] and the *Wolfe condition* [37].

The Armijo rule states that, given a search direction for the kth iteration p_k has been found, the following inequality must hold

$$\mathcal{L}(\theta_k + \alpha_k p_k) \leq \mathcal{L}(\theta_k) + c_1\alpha_k p_k^T\nabla\mathcal{L}(\theta_k) \tag{2.10}$$

Where $0 < c_1 < 1$ (usually small, eg $c_1 \approx 10^{-4}$). That is to say that, we expect the step length to update θ such that it is below a reduced tangent from the point θ_k in the direction of p_k using back tracking line search. It strikes a balance between

1. Not taking steps so large that $\mathcal{L}(\theta_{k+1}) \geq \mathcal{L}(\theta_k)$.

2. Not taking steps so small that convergence is too long.

Since the algorithm is a backtracking linesearch, it will usually take the maximum value of α_k that satisfies the condition.

The Wolfe condition on curvature states that the following inequality must hold

$$|p_k^T \nabla \mathcal{L}(\theta_k + \alpha_k p_k)| \leq c_2 |p_k^T \nabla \mathcal{L}(\theta)| \tag{2.11}$$

Where $0 < c_1 < c_2 < 1$ (for quasi-newton methods like BFGS, $c_2 \approx 0.9$). Fundamentally, we expect that if the step we take is minimising the objective function, then the gradient at this point should be less than the previous point (ie, if we are approaching a minimum, the function should be getting 'flatter').

2.1.5 Validation

Once an ANN has been trained, it can be tested on a separate labelled data-set to see how well it performs on data it hasn't seen before. This data-set is called a *validation data-set* [39]. The validation phase is incredibly important for a variety of reasons

1. The training data-set may not capture every aspect of the problem the ANN is trying to solve. Having a smaller data-set to test the network against can assist a designer in finding gaps in the training data.

2. It can uncover over fitting, a common issue in ML. The ANN may perform incredibly well on the training data-set, but perform poorly in validation. Methods, such as data augmentation[5] and complexity reduction[6], can then be employed to prevent overfitting.

3. Though not an issue for PINNs, the validation phase also needs to ensure fairness across protected variables.

[5]Data augmentation is when a designer artificially adds data to a data set by perturbing original data in a measurable way, such as mirroring an image.

[6]Complexity reduction is simply when a designer simplifies the model that is being used, such as removing nodes and layers. In simplest terms, the more nodes and layers a network has, the more degrees of freedom it has. So, by removing some of these degrees of freedom, overfitting can be avoided.

Chapter 3

Neural Networks for Partial Differential Equations

Classic ANNs and the usual methods of solving or approximating PDEs can suffer from the same complications - a poorly defined problem or a lack of data. For analytical solutions, we need to have closed-form equations for the initial and boundary conditions in the domain of interest. For accurate FEA, we often need a reasonably fine set of data points (usually called a mesh). This mesh needs to be finer still if the PDEs that are being solved require a very smooth solution, as elements need to agree across element boundaries to the kth derivative.

A normal ANN would have some of the same drawbacks. To approximate a complex physical system (to a high tolerance) from data alone may require a dense data set, which is not always available. A classic problem is finding solutions to the 2-dimensional acoustic wave equation, defined as

$$\rho \nabla \cdot \left(\frac{1}{\rho} \nabla u \right) - \frac{1}{v^2} \frac{\partial^2 u}{\partial t^2} = -\rho \frac{\partial^2 f}{\partial t^2} \tag{3.1}$$

where

- $\rho(x, y)$ is a density function that describes how dense the material is at the point (x, y).
- $u(x, y, t)$ is the wave field, the pressure response due to the acoustic wave
- $v(x, y) = \sqrt{\frac{k(x,y)}{\rho(x,y)}}$ is the velocity, with k being the adiabatic compression modulus[1]
- $f(x, y, t)$ is a source term, where force is injected (possibly creating an acoustic wave, such as a small explosion).

[1]Adiabatic compression modulus describes the ability for a material to be squashed. It describes a materials volume decrease as pressure increases.

This equation has applications in earth modelling, and measuring seismic waves (explored in chapter 5). For no source term, and constant density, the equation can be reduced to

$$\nabla^2 u - \frac{1}{v^2}\frac{\partial^2 u}{\partial t^2} = 0 \tag{3.2}$$

where $\nabla^2 u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + ... + \frac{\partial^2 u}{\partial x_n^2}$ is the laplacian operator, with each x_i being a spatial variable. This the simplest form of the wave equation, known as canonical form. Unlike numerical methods, using physics-informed neural networks (PINNs) do not rely in discretising the domain in time and space (a process which often introduces errors). Because of this, PINNs have the capability to learn behaviours and outputs outside of the time-space domain, and can learn these behaviours much more quickly.

3.1 The Loss Function for PINNs

Fundamentally, whilst the PDEs themselves do not give us direct information on what the solution should look like, it does give us an indication on how the solution should behave. Using equation 3.2 as an example, the PDE tells us that the solution should be twice differentiable both spatially and temporally, and that the sum of the average pressure in the spatial directions should be equal to the weighted difference in the temporal directions (which is weighted by the velocity squared) at each point.

If an ANN was given an incredibly dense, well populated, noiseless data-set, it would converge to this behaviour, following from the universal approximation theorem (assuming the size of the network and computational power was no object). However, this level of data cannot always be made available. It is almost impossible, for example, to densely measure the pressure of an acoustic wave over 2km in width and depth. Even if this data set was available, how accurately and precisely could we measure this data? Can we account for the noise, and how will this effect the solution? PINNs work around this lack of data by creating special loss functions that allow the network to converge to a solution by minimising the loss between known outputs and estimated outputs whilst also honouring the expected behaviour of the function described by the PDE.

Take an ANN Λ that estimates the behaviour of a wave field of a function u that is described by equation 3.1. If a standard loss function is described in equation 2.3, which we will now call the data loss, and a differential operator that describes acoustic waves $\mathcal{N}$ is defined as

$$\mathcal{N}(\Lambda) = \rho\nabla \cdot \left(\frac{1}{\rho}\nabla\Lambda\right) - \frac{1}{v^2}\frac{\partial^2 \Lambda}{\partial t^2} + \rho\frac{\partial^2 f}{\partial t^2} \tag{3.3}$$

Clearly, if Λ is an exact solution to the acoustic wave equation, then $\mathcal{N}(\Lambda) = 0$. We can exploit this by adding it as a constraint to the loss function. That is to say, not only do we expect the solution Λ to minimise the difference between known outputs $u(x_i, y_i, t_i)$ and estimated outputs $\Lambda(x_i, y_i, t_i; \theta)$, but we also expect the solution to minimise the value of $\mathcal{N}(\Lambda)$. The physics-informed loss therefore looks like [60]

$$\mathcal{L}(\theta) = \frac{1-\lambda}{N_d} \sum_{i=1}^{N_d} ||u(x_i, y_i, t_i) - \Lambda(x_i, y_i, t_i)||_2^2 + \frac{\lambda}{N_s} \sum_{j=1}^{N_s} ||\mathcal{N}(\Lambda(x_j, y_j, t_j))||_2^2 \tag{3.4}$$

We call the first term the data loss, and the second term the physics loss, where

- $0 \leq \lambda \leq 1$ is a chosen weight parameter. The larger the value the larger the physics contribution is
- N_d is the size of the data-set
- Each $u(x_i, y_i, t_i)$ is a known output of the unknown wave field function u from the data set of size N_d
- Each $\Lambda(x_i, y_i, t_i)$ is the output of the PINN at points which have known values
- N_s is the number of sampling points over the entire domain
- Each $\mathcal{N}(\Lambda(x_j, y_j, t_j))$ is an output of 3.3 for the current parameters θ

3.2 Differentiating a Neural Network

Clearly then, PINNs rely heavily on accurate derivatives. As shown in chapter 1, Jacobians and Hessians are constructed from gradients (or estimated gradients) of the objective function for a sub-set of data in order to minimise it, and these matrices can be incredibly large. These matrices are then used to adjust the network parameters. If the number of adjustable parameters θ for a PINN Λ is n, then $\mathcal{J} \in \mathbb{R}^n$ and $\mathcal{H} \in \mathbb{R}^{n \times n}$, and n can be arbitrarily large. Not only this, but the physics-informed loss from equation 3.4 also requires differentiation with respect to the input variables, which could also be of a high dimension. Therefore, it is obvious that an efficient, accurate way of differentiating an ANN is devised.

When computing the derivatives, there are four main options available [61]

- *Manual Differentiation*: Taking the known function (in this case, the neural network Λ) and differentiating it by hand as many times as is necessary.
- *Numerical Differentiation*: Using finite difference approximations.
- *Symbolic Differentiation*: Using computational expression manipulation libraries, such as SymPy.
- *Automatic Differentiation*: Applying the chain rule in sequence to the individual operators in Λ to evaluate the derivative.

3.2.1 Problems with Differential Techniques

When operating on PINNs, which can have high dimensional inputs and have many parameters, a lot of these techniques present their own set of issues.

Manual Differentiation

Whilst, ultimately, this would be the most accurate way to calculate the derivatives, it is an incredibly cumbersome task. Not only are large ANNs complex expressions, but the number of equations one would have to derive could be astronomical (potentially in the millions). For Λ this would mean differentiating with respect to each input to the kth order to construct the physics loss, with respect to each parameter in θ to construct the Jacobian, and with respect to each possible pairing of parameters in θ to construct the Hessian. Of course, humans are also prone to calculation errors.

Not only is the size of the task an issue, but it is also not a dynamic approach at all. Each derivative would have to be coded by hand, and if the design of the network changed in anyway, an engineer would have to restart the entire process from scratch.

Numerical Differentiation

Numerical differentiation is a widely used technique, which involves approximating derivatives by taking an estimate from the Taylor series expansion of a function. Take a function f. We can estimate the value of the derivative at the point x to an arbitrary precision by taking the Taylor expansion [62]

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{f''(x)\Delta x^2}{2!} + ... + \frac{f^{(n)}(x)\Delta x^n}{n!} + \mathcal{O}(\Delta x^n) \tag{3.5}$$

If we take $n = 1$, the approximate solution of the first derivative can be found, so

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{3.6}$$

Provided that the step size Δx is small enough, the Δx term and beyond should be sufficiently small that they can be ignored. However, we also need to ensure that $|\Delta x| > \epsilon$, where ϵ is machine epsilon[2] , to ensure that rounding errors are avoided [**?**].

As an example, take the 2 dimensional heat equation on the $[0,1] \times [0,1]$ square

[2]Machine epsilon is the upper bound on the relative error for floating point (computational arithmetic) numbers. For single precision $\epsilon = 5 \times 10^{-6}$, for double precision (which is the standard on most modern computers) $\epsilon = 5 \times 10^{-15}$, and for quad precision (used on specialist machines) $\epsilon = 5 \times 10^{-33}$

domain with the following conditions

$$\begin{cases} \phi_t = \alpha(\phi_{xx} + \phi_{yy}) \quad \text{in } \Omega \\ \phi(0, y, t) = 100 \quad (BC_1) \\ \phi(1, y, t) = 25 \quad (BC_2) \\ \phi(x, 0, t) = 200 \quad (BC_3) \\ \phi(x, 1, t) = 0 \quad (BC_4) \\ \phi(x, y, 0) = 50 \quad \text{in } \Omega \quad (IC) \end{cases} \tag{3.7}$$

with $\alpha = 1.28 \times 10^{-4}$, which is the thermal diffusivity of the material. The Taylor series can be used to get a finite difference approximation, so

$$\begin{aligned} \phi_t &\approx \frac{\phi_{ij}^{(k+1)} - \phi_{ij}^{(k)}}{\Delta t} \\ \phi_{xx} &\approx \frac{\phi_{i-1j}^{(k)} - 2\phi_{ij}^{(k)} + \phi_{i+1j}^{(k)}}{\Delta x^2} \\ \phi_{yy} &\approx \frac{\phi_{ij-1}^{(k)} - 2\phi_{ij}^{(k)} + \phi_{ij+1}^{(k)}}{\Delta y^2} \end{aligned} \tag{3.8}$$

Where k is the time step, i and j are the spatial position x and y respectively, Δt is the time step, and Δx and Δy are the spatial steps. If we take $h^2 = \Delta x^2 = \Delta y^2$, then we can make a substitution into the PDE and rearrange to get

$$\phi_{ij}^{(k+1)} = \left(1 - \frac{4\Delta t\alpha}{h^2}\right)\phi_{ij}^{(k)} + \Delta t\alpha\left(\frac{\phi_{ij-1}^{(k)} + \phi_{i-1j}^{(k)} + \phi_{ij+1}^{(k)} + \phi_{i+1j}^{(k)}}{h^2}\right) \tag{3.9}$$

That is to say, if we know the entire state of the plate at some time $t = k$, then we can use this information to make an estimate of the heat distribution of the plate at time $t = k + 1$.

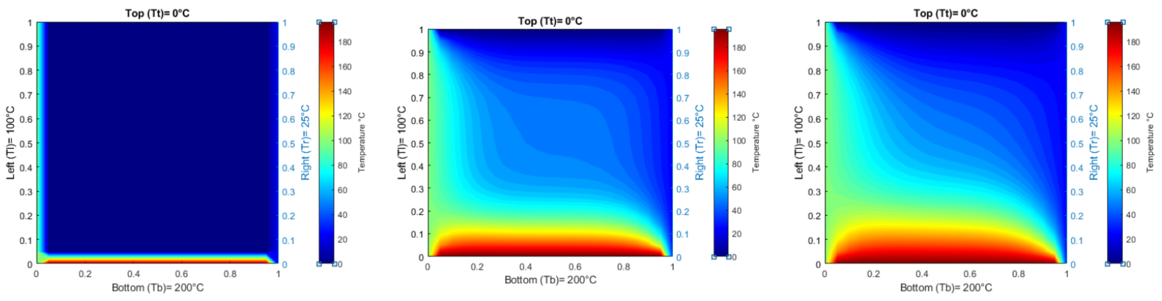

Figure 3.1: Finite difference approximation of the heat equation for 3.7, completing the example. From left to right: $t = 0$, $t = 10$, $t = 20$. Code in appendix A.5.1

Clearly, the finite difference method has its merits. It is simple to use and implement, can be quick to calculate for small time and domains, and the error is easy to quantify.

It is also reasonably dynamic - changing the structure of the PINN would not alter the code or strategy used to approximate gradients.

It does, also, have its draw backs. One such drawback is that we can only work forward from the ICs [?]. If we wanted to know the solution for large t, we would need to (accurately) compute all values of $\phi(x, y, t)$ up to this point. Step size in both time and space must also be considered. Too large, and the solution will not be adequately accurate. Too small, and the solution is prone to rounding errors due to machine precision. This problem is exacerbated by much more complex, higher order PDEs, such as fourth order PDEs (sometimes called biharmonic equations)[65] which then require that $h^4 > \epsilon$. This sets a real limit on the step size that can be used, and can therefore introduce inaccuracies. Since each iteration's accuracy is also reliant on the previous iteration's accuracy, the errors can easily propagate forward. Fourth order PDEs are not a rare occurrence. A particularly famous one, which appears in structural mechanics and engineering, is the plate bending problem

$$\nabla^4 w = \frac{p}{D} \tag{3.10}$$

where p is the load distribution on the plate, and D is the Young's modulus[3]. There are PDEs that have an even higher order (that have physical applications), but they are considerably more rare. One 5th order PDE is the Kaup–Kupershmidt equation [66], which has similar applications to the more famous Korteweg–De Vries (KdV) equation used in modelling shallow water waves [67].

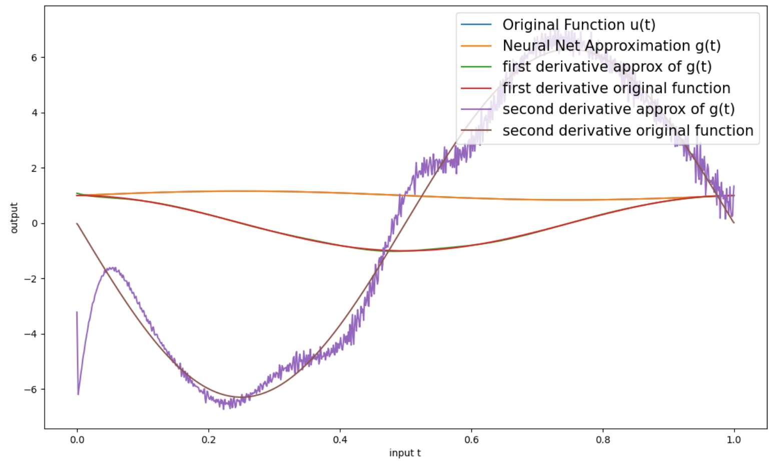

Figure 3.2: A ANN approximation estimating the solution to a simple ODE. The approximate solution is incredibly accurate, as is the 1st derivative. Despite this, estimating the 2nd derivative using finite difference methods shows how much smoothness we can lose when compared to the actual 2nd derivative. Code in appendix A.4.1

[3]Young's modulus, named after Thomas Young, is a ratio between the tensile stress strength of the material over the strain strength of the matieral.

Since machine learning relies on accurate gradients for (sometimes) millions of parameters to optimise the ANN, introducing rounding errors and truncation errors in the training phase can have a huge impact on the ANN's ability to converge to an adequate solution, especially since gradients are already only estimates of the space when using stochastic methods. Badly chosen step sizes can also lead to numerical instabilities. On top of this, when considering a PINN, the loss function includes the PDE as part of the optimisation, and so higher order PDEs would suffer from the same issues as the finite difference method.

Symbolic Differentiation

Symbolic differentiation takes the expression described by a function and applies the differential rules (product rule, chain rule, etc) to return a new function f', which is the differential of the function f. It is, in truth, very similar to manual differentiation, except that a computer handles the calculation to return a new expression. This is possible because, when it comes to differentiation, there are really only 8 rules to follow - the challenge, therefore, is applying them correctly.

Symbolic differentiation confronts many of the issues that plague the first two methods. The computer handles the heavy lifting of calculating the expression, which solves the main problems behind applying manual differentiation. Also, the solution that is found is exact, and so we avoid the numerical errors that estimating from the Taylor series expansion bring.

However, it does also have problems when applied to computational functions, especially when applied to ANNs. Firstly, it cannot take into account conditional computational logic (such as if, while, and for). Secondly, the ANN would need to be expressed in a closed form, which, whilst technically possible, adds another challenge. Thirdly, symbolic differentiation is subject to extreme expression swell [61], (an example of which is in appendix A.1.1). Some rules for derivatives, such as the product rule, naturally lead to an increase in terms in the equation, and so calculating the derivatives can be incredibly cumbersome.

3.2.2 Automatic Differentiation

Automatic differentiation (AD) solves many of the challenges presented above, as it can calculate the derivative at a point to machine precision. It does this by utilising two aspects of compuatational mathematics - *dual numbers* and *computational graph representation* [68]. Fundamentally, all functions are composed of simple operations that can be differentiated, and intermediate variables that store information in the function at different points. AD uses this fact, and exploits the chain rule to combine derivatives of smaller sub functions to find the numerical value of the derivative directly, instead of attempting to calculate a closed form expression, or estimate it using surrounding points.

Dual Numbers

Dual numbers are a special way of representing numbers in floating point arithmetic [69], which can be leveraged to calculate derivatives of functions at the same time as calculating the primary output. Take z to be a floating point number and ϵ to be infinitesimally small. We can then say that $z = a + b\epsilon$, which can be represented in a matrix form as

$$a = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \quad b\epsilon = \begin{bmatrix} 0 & b \\ 0 & 0 \end{bmatrix} \tag{3.11}$$

and so

$$\epsilon = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \implies \epsilon^2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{3.12}$$

Now, take a generic polynomial P of degree N, which can be represented in the concise form

$$P(x) = a_0 + \sum_{n=1}^{N} a_n x^n \tag{3.13}$$

It follows that

$$\begin{aligned} P(x+\epsilon) &= a_0 + \sum_{n=1}^{N} a_n (x+\epsilon)^n \\ &= a_0 + a_1(x+\epsilon) + a_2(x+\epsilon)^2 + ... + a_N(x+\epsilon)^N \\ &= a_0 + a_1(x+\epsilon) + a_2(x^2 + 2\epsilon x + \epsilon^2) \\ &+ a_3(x^3 + 3\epsilon x^2 + 3\epsilon^2 x + \epsilon^3) + ... \end{aligned} \tag{3.14}$$

However, since $\epsilon^2 = 0$, all terms ϵ^k with $k \geq 2$ can be removed, and so

$$\begin{aligned} P(x+\epsilon) &= a_0 + a_1(x+\epsilon) + a_2(x^2 + 2\epsilon x) \\ &+ a_3(x^3 + 3\epsilon x^2) + ... + a_N(x^N + N\epsilon x^{N-1}) \\ &= a_0 + \sum_{n=1}^{N} a_n x^n + \epsilon \sum_{n=1}^{N} a_n n x^{(n-1)} \\ &= P(x) + \epsilon \frac{\partial P(x)}{\partial x} \end{aligned} \tag{3.15}$$

So, by evaluating $P(x+\epsilon)$ for a given value x in dual number form, we have calculated the value of P at the point x and the derivative at the point x, all in one pass, to machine precision. In general, if a function f is differentiable at a point x, then

$$f(x+\epsilon) = f(x) + \epsilon \frac{\partial f(x)}{\partial x} \tag{3.16}$$

provided that ϵ^2 is computationally 0.

Computational Graph

A computational graph is simply a way to represent a function, making it easy to understand the flow of data [70]. It is very similar to the representation of ANNs in figure 1.2, except that it is slightly more generic (as it applies to all types of functions, not just ANNs) and more detailed, as it also shows the operations involved between each step.

A short example would be to imagine a function $f : \mathbb{R}^2 \to \mathbb{R}$. If $x \in \mathbb{R}^2$, and

$$f(x) = \left(\frac{x_1}{x_2} + \cos(x_1)\right)\left(\frac{x_1}{x_2} + e^{x_2}\right) \tag{3.17}$$

then its computational graph representation could look like

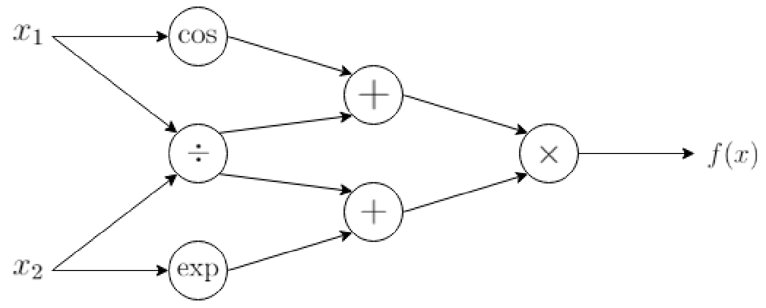

Figure 3.3: Computational graph representation of eq 3.17, showing how 2 variables are manipulated together and operated on to generate any output

This type of representation is useful in three ways. Firstly, it is a reasonably digestible way to see how variables are operated on in a given function. Secondly, it allows the algorithm to exploit repeated operations and variables. As an example, the division x_1/x_2 appears twice in equation 3.17, and this information is utilised in the graph with the operation only being performed once. Thirdly, this representation allows us to systematically apply the chain on intermediate variables in order to efficiently calculate partial derivatives of the function with respect to the inputs in such a way that it can all be done in one pass.

Calculating Derivatives

As mentioned prior, AD does not find an expression for the derivative with respect to any variable, but instead finds the value of the derivative at specific points with respect to any variable. It does this by exploiting the chain rule, which it can do in one of two ways - *forward-mode* AD or *reverse-mode* AD.

Forward AD is more efficient when the function we are differentiating f maps to more outputs then inputs, and vice versa for reverse mode AD. Take the example defined in equation 3.17, but this time we want to keep a record of intermediate variable values.

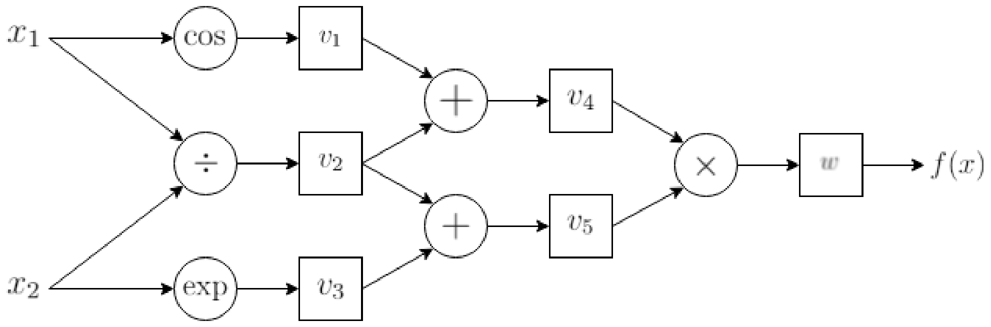

Figure 3.4: Computational graph representation of eq 3.17, this time storing the value

As we calculate the values for each variable, (often called the primals), we can also calculate the gradient for each variable with respect to an input (often called the tangents). Assume, for a moment, the input vector is $x = [1, 2]^T$, and we require the value of f at this point, as well as the gradient with respect to x_1. Defining $\dot{v_i}$ as any variable differentiated with respect to x_1, we can then perform a forward pass, calculating the primal and tangent of each variable by utilising the properties of dual numbers.

Primals		Tangents	
Expression	Value (6 d.p.)	Expression	Value (6 d.p.)
$x_1 = x_1$	1	$\dot{x_1} = 1$	1
$x_2 = x_2$	2	$\dot{x_2} = 0$	0
$v_1 = \cos(x_1)$	0.540302	$\dot{v_1} = \dot{x_1}\sin(x_1)$	-0.841471
$v_2 = \frac{x_1}{x_2}$	0.5	$\dot{v_2} = x_2^{-1}$	0.5
$v_3 = e^{x_2}$	7.389056	$\dot{v_3} = 0$	0
$v_4 = v_1 + v_2$	1.040302	$\dot{v_4} = \dot{v_1} + \dot{v_2}$	-0.341471
$v_5 = v_2 + v_3$	7.889056	$\dot{v_5} = \dot{v_2} + \dot{v_3}$	0.5
$w = v_4 v_5$	8.207001	$\dot{w} = \dot{v_4} v_5 + v_4 \dot{v_5}$	-2.173732

Table 3.1: Represented here are the values of the intermediate variables for the computational graph in figure 3.4. By exploiting the chain rule, one forward pass calculates both the value of the function at a given point and its gradient. The computer doesn't really 'know' the expression for the tangents - these values are calculated as a byproduct of using dual numbers. They are then used with the chain rule to calculate the derivative.

Therefore, $\frac{\partial f}{x_1} = \dot{w} = -2.173732$. We can verify this as true by evaluating the expression of the derivative at the point (which we can do here because f is simple), so

$$\frac{\partial f}{\partial x_1} = \left(\frac{x_1}{x_2} + \mathrm{e}^{x_2}\right)\left(\frac{1}{x_2} - \sin(x_1)\right) + \frac{\cos(x_1) + \frac{x_1}{x_2}}{x_2} \tag{3.18}$$

and so $\frac{\partial f}{\partial x_1}\big|_{x_1=1,x_2=2}$ is -2.173732, as expected (in fact, it is correct to 14 decimal places, as we would expect from machine precision). Using a finite difference with a step size of $\Delta x_1 = 1 \times 10^{-5}$ yields $\frac{\partial f}{\partial x_1}\big|_{x_1=1,x_2=2} \approx -2.1738$, which is only correct to 3 decimal places, and thus we lose a great deal of precision, even for this very simple function.

Therefore, primals and tangents can be calculated in parallel. Using forward AD, an entire column of the Jacobian is generated in a single pass, and less calculation is required because intermediate values and repeated operations can be exploited more effectively. For $m > n$, this is an efficient way to calculate gradients. However, if $n > m$, we can instead use reverse mode AD to construct the Jacobian each row at a time instead. Constructing the Jacobian column wise means each pass finds the value of all outputs with respect to one input, and constructing the Jacobian row wise finds the value of single output with respect to each input. The cost of running forward mode AD is $\mathcal{O}(n)$, whereas reverse mode is $\mathcal{O}(m)$. Other modes of differentiation either introduce errors, and/or are $\mathcal{O}(nm)$ [71].

Many of the functions in physics have a high dimensional input, but a low dimensional output, which means that reverse mode AD is a very good option for calculating derivatives. This is especially apparent in machine learning, where the parameter size is (usually) significantly larger than output size.

3.3 The Universal Approximator

Whilst the main driving force behind PINNs is the way the loss function is defined, there is more to consider when designing the network. Firstly, we should ask ourselves what other information we can directly encode into the PINN to increase the speed of convergence.

Two such pieces of information are the boundary conditions and ICss. Consider, for a moment, the problem of modelling a 2-dimensional vibrating membrane on a 2×3 rectangular domain. Fix at the edges (*Dirichlet* boundary condition[4]), and define the initial shape and velocity of the membrane within the domain such that

$$\begin{cases} u_{tt} = 6^2(u_{xx} + u_{yy}) \quad \text{in } \Omega \\ u = 0 \quad \text{on } \partial\Omega \quad (BC) \\ u = xy(2-x)(3-y) \quad \text{for } t = 0 \quad (IC_1) \\ u_t = 0 \quad \text{for } t = 0 \quad (IC_2) \end{cases} \tag{3.19}$$

[4]Named after *Peter Gustav Lejeune Dirichlet*, the Dirichlet boundary condition is such that the function takes a specific, fixed solution along the boundary of the domain

so the domain is $[0,2] \times [0,3]$. This PDE has a series solution

$$u(x,y,t) = \frac{576}{\pi^2} \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \Bigg(\frac{(1+(-1)^{i+1})(1+(-1)^{j+1})}{i^3 j^3} \sin\Big(\frac{i\pi x}{2}\Big) \sin\Big(\frac{j\pi y}{3}\Big) \cos(\pi\sqrt{9i^2+4j^2}t) \Bigg) \tag{3.20}$$

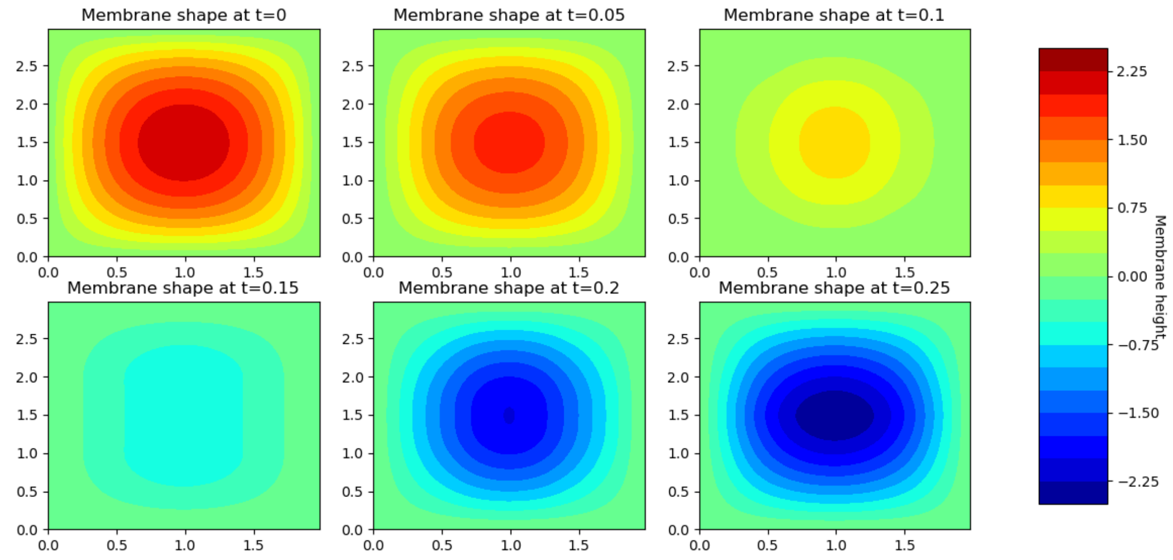

Figure 3.5: Series solution for a simple vibrating membrane with Dirichlet boundary conditions and no dampening, as described in 3.19

Take U to be a universal approximator and $\Lambda(x,y,t)$ to be an untrained PINN. In this case, the differential operator in the loss function is simply

$$\mathcal{N}(\Lambda) = \Lambda_{tt} - 6^2(\Lambda_{xx} + \Lambda_{yy}) \tag{3.21}$$

and so the loss function is defined exactly as it is in equation 3.4, with the new differential operator described above. The first instinct is to simply create the universal approximator $U(x,y,t)$ to solely be equal to the the PINN [72], so

$$U(x,y,t) = \Lambda(x,y,t) \tag{3.22}$$

However, this means that Λ has to learn the ICs and BCs first before applying any time towards learning how to map the rest of the function. Clearly, since this information is already known here, it would be more prudent to give the PINN this information beforehand [73]. To encode the IC, we can instead define

$$\begin{aligned} U(x,y,t) &= t\Lambda(x,y,t) + u(x,y,0) \\ &= t\Lambda(x,y,t) + xy(2-x)(3-y) \end{aligned} \tag{3.23}$$

with the loss function being

$$\mathcal{L}(\theta) = (1-\lambda)\frac{1}{N_d}\sum_{i=1}^{N_d} ||u(x_i, y_i, t_i) - U(x_i, y_i, t_i)||_2^2 + \lambda\frac{1}{N_s}\sum_{j=1}^{N_s} ||\mathcal{N}(U(x_j, y_j, t_j))||_2^2 \quad (3.24)$$

Now, whenever $t = 0$ the PINN automatically satisfies the ICs perfectly, and this behaviour does not need to be approximated at all. We can do exactly the same for the BCs by expanding the universal approximator even further, such that

$$\begin{aligned} U(x, y, t) &= u(\partial\Omega, t)t\Lambda(x, y, t) + u(x, y, 0) \\ &= xy(2-x)(3-y)t\Lambda(x, y, t) + xy(2-x)(3-y) \end{aligned} \quad (3.25)$$

Since $f(x, y) = y(2-x)(3-y) = 0$ for $x = 0$, $x = 2$, $y = 0$, or $y = 3$ (the extreme parts of the domain of interest), this satisfies the BCs. This small amount of work means the the universal approximator perfectly models the BCs for all time $t \geq 0$, and the ICs for all space within the domain before the training phase has even started.

It is important to note that this is not always possible to do. We may simply lack the knowledge of the BCs/ICs, the data may be noisy or sparse, or the boundaries may be open or ill-defined. However, it does highlight that there are methods that exist outside of the training phase to ensure that the PINN converges to the true solution.

Chapter 4

Simple PINN Example

All material in the previous chapters can be applied to build a PINN for almost any problem, provided it is well defined. In cases where the problem is ill-defined, a PINN will still converge to the closest solution it can, and which solution it converges to will often be dependent on the initial parameters, the data it has access to, and the structure of the network.

4.1 General Framework

This section utilises the framework built by M. Raissi, et al. (2017) [50]. The paper established that well defined linear and nonlinear PDE problems can be solved to a high level of accuracy using PINNs. Not only this, but a second follow up paper showed that functions and coefficients from a PDE could be reconstructed if enough data in the domain of interest is provided [51]. These two discoveries lay the foundation of the further work that is discussed in the next chapter.

The code (provided in A.4.2, in which there is more detail on use) utilises the Tensorflow package with the L-BFGS optimisation algorithm in Python to approximate solutions to PDEs. In order to produce meaningful results, the program requires 5 things from the user:

1. The size of the domain in both spatial and temporal dimensions, so $a \leq x \leq b$ and $0 \leq t \leq T$.

2. The ICs of the problem for the spatial domain at $t = 0$. This can either be as a function, or as discrete data points.

3. The prescribed BC. Again, this can be delivered as a function for all $0 \leq t \leq T$, or as discrete points.

4. The network structure (input size, output size, number of hidden layers, and node per layer).

5. The governing equations to create the custom loss function

Some other parameters, such as solution precision, max iterations, data sample size, etc can also be changed, though there is a default option. This is because there is a trade off between time to get a solution, and the accuracy that is provided by the PINN.

4.2 1D Wave Equation

4.2.1 Analytical Solution

To demonstrate a simple and intuitive example, take the 1D wave equation with the conditions

$$\begin{cases} u_{tt} = u_{xx} & \text{for } 0 \leq t \leq 4, \quad 0 \leq x \leq 2 \\ u(x,0) = x(2-x) & \text{IC} \\ u_t(x,0) = 0 & \text{IC} \\ u(0,t) = u(2,t) = 0 & \text{Boundary Condition} \end{cases} \tag{4.1}$$

which has the following analytical solution (proof in appendix A.1.2)

$$u(x,t) = \sum_{n=1}^{\infty} -\frac{8\pi n \sin(\pi n) + 16\cos(\pi n) - 16}{\pi^3 n^3} cos\left(\frac{n\pi t}{2}\right) \sin\left(\frac{n\pi x}{2}\right) \tag{4.2}$$

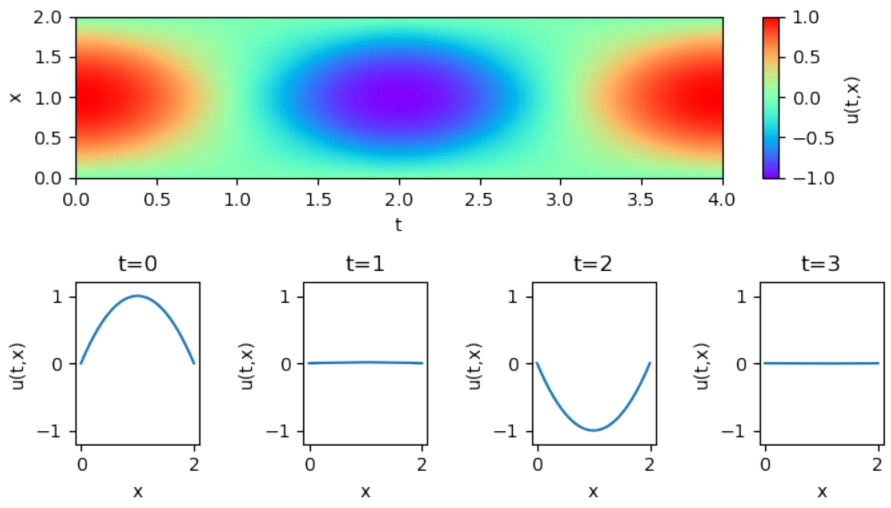

Figure 4.1: Behaviour of the wave described in 4.1. The wave will oscillate like this for all time, since there is no dampening present, and has period 4

4.2.2 Neural Network

In order for a PINN to approximate the solution, no additional information is required at all. In fact, we can approximate a close solution with even less information, as the IC and BC data set might be discrete. In this case, the BC and IC were discretised, and so the network had no knowledge of the behaviour of the IC or BC for all time, only at specific points.

For the 1D wave problem, the PINN $\Lambda(\theta)$ had to minimise

$$\mathcal{L}(\theta) = \sum_{i=1}^{N_i}\Big(u(x_i,0)-\Lambda(x_i,0;\theta)\Big)+\sum_{j=1}^{N_b}\Big(u(x_b,t_j)-\Lambda(x_j,t;\theta)\Big)+\sum_{k=1}^{N_p}\Big(\mathcal{N}(\Lambda(x_k,t_k;\theta))\Big) \tag{4.3}$$

where

- The first sum is a loss term for the IC, where $N_i = 1000$ are equidistant points taken along the IC
- The second sum is a loss term for the boundary, where $N_j = 1000$ are randomly selected points along the boundary, so $t_j \in [0, T]$ and $x_b \in \{0, 2\}$
- The third term is the physics loss, where $N_p = 10000$ are randomly selected points across the entire domain, and $\mathcal{N}(\Lambda) = \Lambda_{tt} - c^2\Lambda xx$ (with c=1 is the wave speed for this problem)

Derivatives are calculated using dual numbers and automatic differentiation by Tensorflow's inbuilt *GradientTape* function, which allows a program to *watch* the network operations with respect to inputs. These functions can be nested in themselves to calculate higher order derivatives.

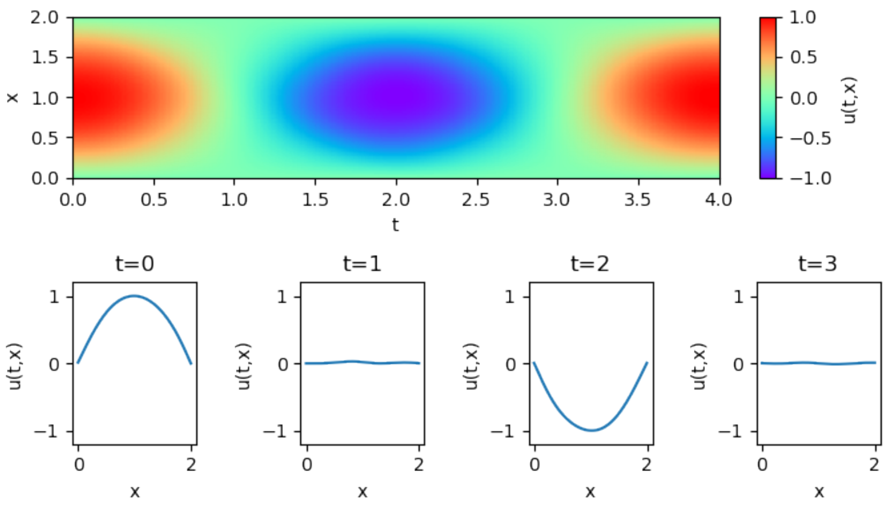

Figure 4.2: Behaviour of the wave as approximated by an PINN. For the above example network structure was $[64, 32, 16, 8]$, so 4 hidden layers with a descending hierarchy.

4.3 Error Analysis

The approximate solution provided by the PINN in figure 4.2 is clearly quite a good fit. The general behaviour is correct, and the solution snapshots at $t = 0, 1, 2, 3$ are also very similar to the analytical solution. Since the analytical solution is available, in this case, it is possible to do a reasonably detailed error analysis.

Error analysis of this very simple problem is a crucial endeavor, as it could allow for better informed and more intelligent design considerations when creating PINNs for more complex wave-related problems. It is important to explore where errors occur, in what magnitude they occur, and why they occur.

4.3.1 Measuring Error

Error between the solution and the approximation can be a challenging thing to quantify. However, there are three possible inspirations for estimating the errors between the analytical solution and the PINN solution.

- The energy error.
- The mean squared error (MSE).
- The relative L_2 norm error.

Often, we are limited to finding an upper bound on the error, and thus can only know the worst case scenario. FEA estimates the error by studying how the solution differs from the approximation in an *energy norm* sense [52] [53].

$$\begin{aligned}\int_\Omega (\nabla u - \nabla u_h) \cdot (\nabla u - \nabla u_h) &= ||u - u_h||_E^2 \\ &= ||u||_E^2 - ||u_h||_E^2\end{aligned} \tag{4.4}$$

where u is the exact solution, u_h is the approximate solution, and $\int_\Omega$ is the domain integral. In lieu of this, FEA uses a reference solution to find how the energy norm changes between solutions as the grid is refined, testing for convergence.

$$||e_{ref}||_E^2 = ||u_{ref}||_E^2 - ||u_h||_E^2 \tag{4.5}$$

where u_{ref} is a fine mesh reference solution, and u_h is a solution on a coarser mesh. This reference error can be bounded by considering the largest possible error, which occurs on the element with the longest edge, or calculated explicitly by running the calculations for a fine mesh. This reference method is worth remembering, as we may not always have access to the true solution u. However, in this case we do know the true solution, and can directly use the true energy norm difference. For this specific problem then, the energy norm error is

$$e_{energy} = \int_0^2 \int_0^4 (\nabla u - \nabla\Lambda) \cdot (\nabla u - \nabla\Lambda) dtdx \tag{4.6}$$

Since we are dealing with two functions that are continuous in the domain, the energy norm error is a good approximation to use, as other measures require us to discretise the domain.

The *MSE* is defined as

$$e_{MSE} = \frac{1}{N_e} \sum_{i=1}^{N_e} \Big(u(x_i, t_i) - \Lambda(x_i, t_i) \Big)^2 \tag{4.7}$$

where N_e is the number of data samples, and pairs (x_i, t_i) are sampled data points. Fundamentally, it is a measure of the average squared error at a finite number of points in the domain. These points can be randomly selected, but here we have taken a defined grid 200×400 of equidistant points in x and t, giving $N_e = 80000$.

The final possible method, used in linear algebra, is the *relative L_2 norm error*. Again, we take a discrete sample of data points from the solution space and the approximation space and store them as vectors x and b respectively (in this case, 80000 once again). We then find the euclidean distance of the difference between the solution and the approximation vectors, relative to the euclidean distance of the solution.

$$e_L = \frac{\sqrt{\sum_{i=1}^{N_e} (x_i - b_i)^2}}{\sqrt{\sum_{i=1}^{N_e} x_i^2}} = \frac{||x - b||_2}{||x||_2} \tag{4.8}$$

4.3.2 Architecture Design

Selecting the number of hidden layers and nodes per layer is an important step when designing any kind of ANN. Surprisingly, there is very little solid theory revolving around how and why certain architectures work better than others. When posed with the question ‘how should I know how many hidden layers to use’ in 2013, Yoshua Bengio, who is the Head of the Montreal Institute for Learning Algorithms, stated *‘Very simple. Just keep adding layers until the test error does not improve anymore.’*

One school of thought is to study the data-set and see how many straight, connected lines are needed to sufficiently partition the data into defined groups. The number of connections between each line is the number of nodes in hidden layer 1, the number of connections between the set of once-connected lines is the number of nodes in layer 2, etc until all lines are connected [54].

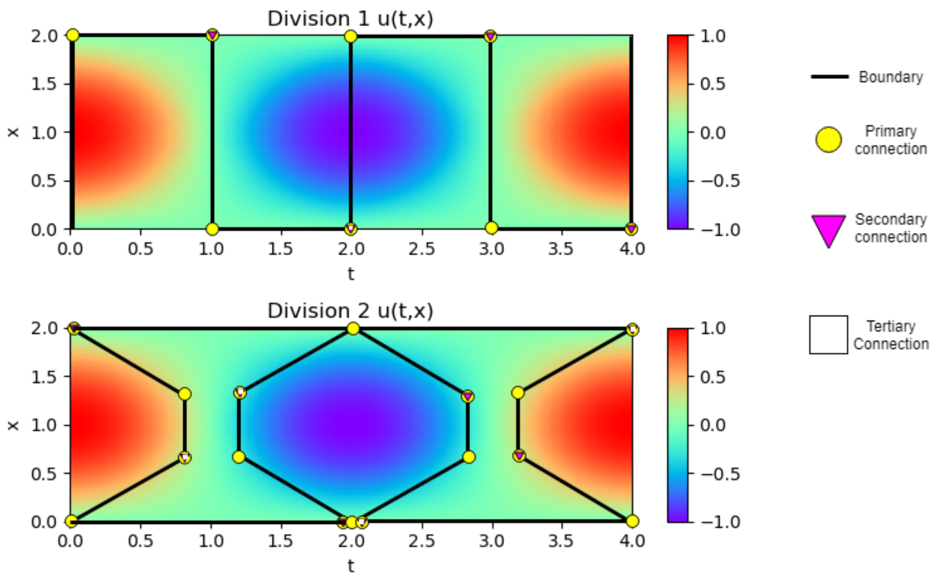

Figure 4.3: Two possible simple partitions of the behaviour of the wave. Division 1 divides the data into sections where derivatives change property, and division 2 categories subsets of data by how close the value is to -1, 0, or 1. Division 1 would imply a 8-4-2 network, whereas division 2 implies a 16-8-4-2 network.

Most theories work backwards. That is to say, we create a large ANN that definitely works, and then, through a process called *pruning* [55], layers can be removed. A good rule of thumb is to analyse the weights and biases after training. Weights that are approximately 0 (or practically 0 relative to others) can often be removed, as this shows that they have very little impact on the output. Another method is to continually add layers until the error no longer gets smaller, as suggested before. At this point, one layer is then removed to make the function behaviour a little less specific to the training data. Both of these techniques can also prevent overfitting.

Here, some different PINN structures were applied, mainly studying how depth impacted the ability for the PINN to converge.

Structure	Parameters	e_{energy}	e_{MSE}	e_L
128	513	8.858×10^{-5}	1.956×10^{-4}	2.7223×10^{-2}
$64 - 64$	4,417	3.622×10^{-6}	5.509×10^{-5}	1.444×10^{-2}
$20 - 20 - 20 - 20$	1,341	3.522×10^{-4}	2.721×10^{-4}	3.204×10^{-2}
$32 - 16 - 16 - 32$	1,473	4.475×10^{-5}	1.711×10^{-4}	2.541×10^{-2}
$64 - 32 - 16 - 8$	2,945	1.1815×10^{-5}	7.648×10^{-5}	1.699×10^{-2}
$8 - 4 - 2$	73	3.10×10^{-3}	5.954×10^{-4}	4.742×10^{-2}
$16 - 8 - 4 - 2$	233	1.11×10^{-3}	5.262×10^{-4}	4.458×10^{-2}

Table 4.1: Table of errors for differing neural network structures across a spectrum of errors. Most of the architectures give a reasonable estimate of the solution, even when the number of trainable parameter is small. However, training a system with 4417 parameters on an Intel quadcore i7 took over an hour (30000 iterations), whereas 73 parameters took less than a minute. This is why finding sufficiently small network is crucial.

Upon observation, there is clearly a balance to be struck between layer depth and number of nodes. Whilst having the largest number of parameters did create the most accurate approximation, it is not always the case that more parameters means more accuracy. Other much simpler architectures were also reasonably accurate, and there is a huge trade off to consider during the training phase. Training the $[64, 64]$ network took over an hour, whereas training the smaller networks, such as the $[8, 4, 2]$ network, took only minutes. However, the main take away is that even simple networks can capture the general behaviour of a vibrating wave.

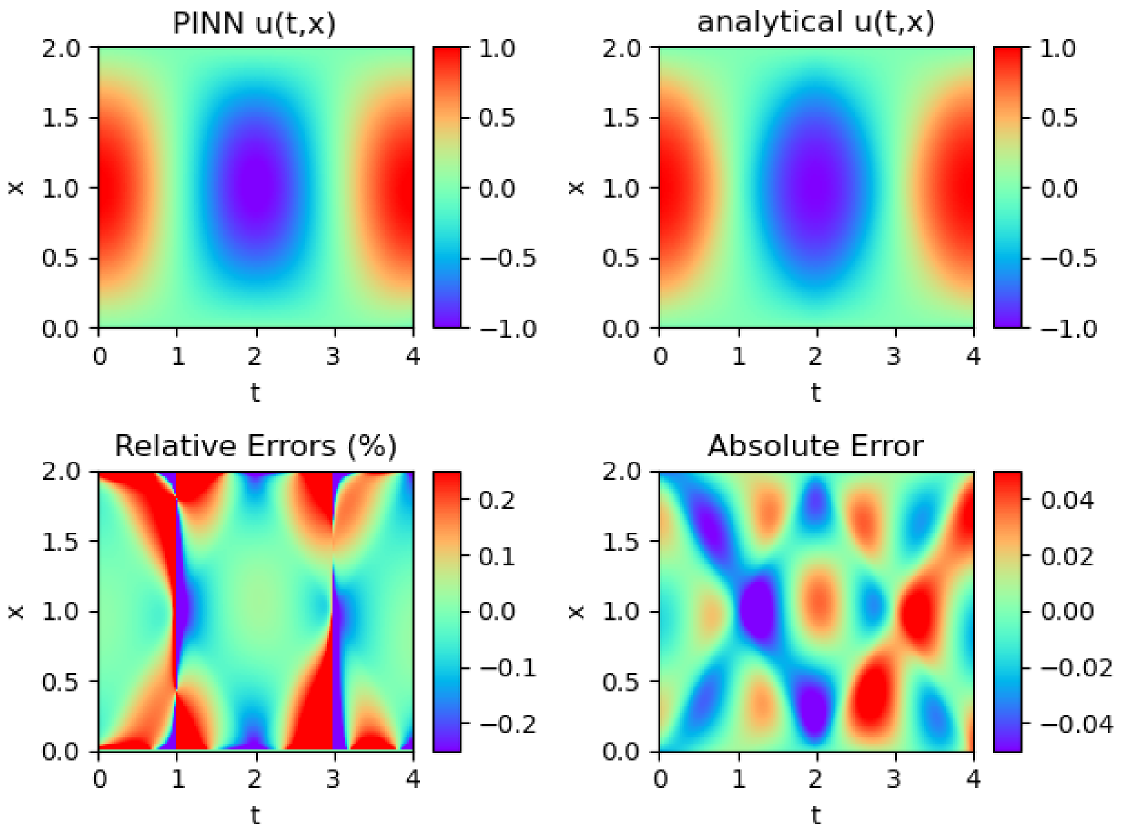

Figure 4.4: Error between true solution and the 8-4-2 network, which was the simplest network tested.

Figure 4.4 shows the final solution for the simplest network with the fewest parameters. Whilst the errors shown here are the largest out of any of the networks, it is clear from the plot that the general behaviour is captured. Due to the incredibly short time to train this network, it could be used as a quick estimate to inform a designer how to build a more intelligent network. The absolute time to convergence could be sped up dramatically by using a graphics processing unit (GPU) instead of a CPU, which was unfortunately not an option here. The L-BFGS was used to find a minimum with $m = 50$.

4.3.3 Solution Convergence

Knowing the total error for the whole approximate solution is only half of the problem. Something that is more interesting is exploring where these errors occur, and why this might be the case. As an example, let us take the estimate from the $[64, 32, 16, 8]$ network.

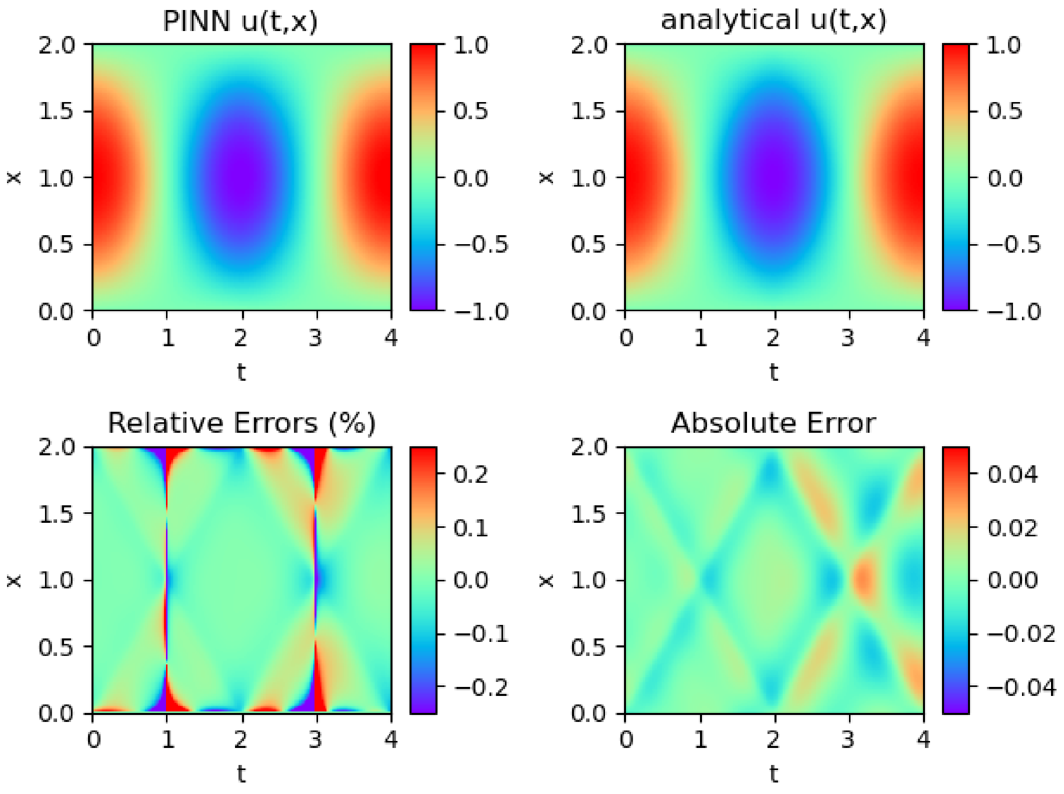

Figure 4.5: Error between true solution and the 64-32-16-8 network

Judging from table 4.1 and figure 4.5, the solutions is a reasonably good fit. The relative errors are occasionally large, but for good reason. The relative error is calculated as

$$e_{rel} = \frac{\Lambda(x,t)}{u(x,t)} - 1 \tag{4.9}$$

Large relative errors will occur when $u(x,t) \approx 0$ by the nature of the equation used to calculate them. Elsewhere, we can also see that when $|u(x,t)| > 0$ the relative errors are very small, except in the crossing pattern that is noticeable in the absolute errors. For this network, the convergence was tracked. This was done by taking a snapshot of the approximation space every 50 iterations, and the comparing it to the true solution via the L_2 error.

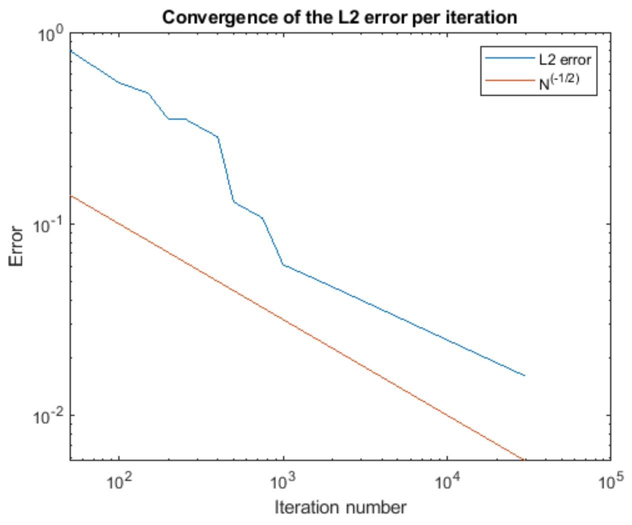

Figure 4.6: Convergence of the L_2 error has minimisation progresses

The solution approximately converges at a rate of $\mathcal{O}(N^{\frac{1}{2}})$, so

$$e_{conv} = \gamma N^{-\frac{1}{2}} \tag{4.10}$$

where N is the number of iterations performed. In this case, $\gamma \approx \frac{5}{2}$. This rate of convergence is also the same for FEA [58], where N would be the number of nodes (also called degrees of freedom) in the mesh. More interesting, however, is how this convergence occurs and where the larger errors are located.

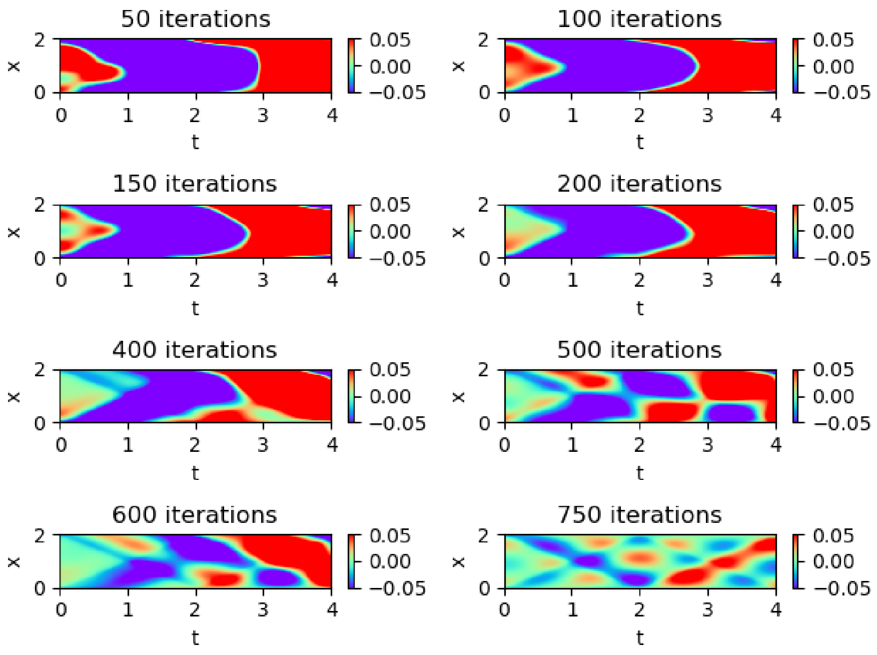

Figure 4.7: Convergence behaviour of $\Lambda(x,t)$ for 64-32-16-8 network. Each figure shows the absolute error against the true solution.

There seem to be two main observations.

1. Errors seem to 'propogate' out from the IC, with areas near the IC having a lower absolute error.

2. The errors have a sort of structure or pattern to them. This structure or pattern is seemingly consistent across differing network architectures (a good example is figure 4.4 vs figure 4.5) - the error is simply more intense on the networks with fewer parameters.

4.4 Improving the Learning Process

The previous section described how the PINN converges to the true solution $u(x, t)$. Knowing this, we can then infer how and why we might improve the performance of the learning process, which could then be applied to much larger scale, more complex problems (which are explored in a later chapter).

4.4.1 Error Propogation

One of the points regarding the error is that, in the approximate solution for the 1D wave problem, errors seem to increase as we move away from the IC, and figure 4.7 confirms this.

For this problem, all of the collected data comes from two places

1. The boundary of the problem, where $u(0, t) = u(2, t) = 0$

2. The IC of the problem, where $u(x, 0) = x(2 - x)$

Thus, we can infer that to have a low error at $(x = x_1, t = t_1)$, the network must first have a good approximation at $(x = x_n, t < t_1)$, where x_n is a point in the neighbour hood around x_1. This conclusion is sensible when we recall that the network learns behaviours by using second order derivatives. In other words, to optimise a single point in space and time it requires good estimates of the surrounding points in space and time, and the best initial estimates will always be near points in the domain that are densely packed with data.

If, instead of taking data from the IC, we take the same amount of data, but at $t = 2$ (where $u = x(x - 2)$), we should expect the approximation to converge from the centre of the domain moving outwards.

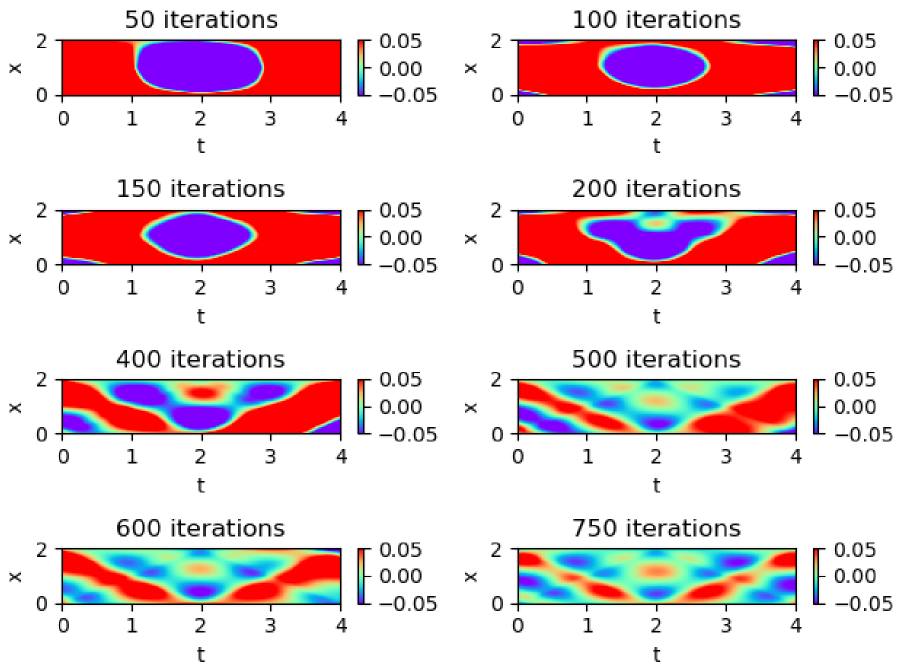

Figure 4.8: Convergence behaviour of $\Lambda(x,t)$ for data taken from $t = 2$, rather than taken directly from the IC.

The prediction appears to be correct. We can clearly see that the optimisation must first find a good solution around the given data before finding the solution closer to $t = 0$ and $t = 4$. On top of this, the pattern of the errors still seems to take the same shape as for the previous approximation (which used data strictly from $t = 0$. This is discussed later.

Most PINN minimisation algorithms, including the algorithm used in this section, take random samples of the domain every iteration to minimise the physics loss term over the entire domain. However, the error propagation highlights that it is somewhat pointless to attempt to learn behaviours for $t = t_0 + t_1$ where $|t_1| \gg 0$ if the behaviour at $t = t_0$ is poorly approximated and $u(x,t)$ is (mostly) known at $t = t_0$. This logic leads to a potential theory in how to converge to a solution in either fewer iterations, or with fewer data points per iteration, thus increasing the computational speed.

Theorem 1 *Let Y be the domain in which a PDE is defined. Let $X^{(0)} = X_1^{(0)} \cup X_2^{(0)} \cup \ldots \cup X_n^{(0)} \subseteq Y$ where $X_k^{(0)}$ is a part of the domain where known data is dense, partitioned in such a way that $X_i^{(0)} \cap X_j^{(0)} = \varnothing$ for $i \neq j$. Then, for the first m iterations or until a suitable tolerance is found, the loss is defined as*

$$\mathcal{L}^{(0)}(\theta) = \frac{(1-\lambda)}{N_d} \sum_{i=1}^{N_d} \Big(\Lambda(x_i, t_i; \theta) - u(x_i, t_i) \Big) + \frac{\lambda}{N^{(0)}} \sum_{j=1}^{N^{(0)}} \Big(\mathcal{N}(\Lambda(x_j^{(0)}, t_j^{(0)}; \theta)) \Big) \quad (4.11)$$

where each (x_i, t_i) is a known data point, $(x_j^{(0)}, t_j^{(0)}) \in X^{(0)}$ is a randomly sampled point *inside densely packed areas, and $N^{(0)}$ is the number of randomly sampled points from $X^{(0)}$. After a set number of iterations m (or convergence), define $X^{(1)} = X_1^{(1)} \cup X_2^{(1)} \cup \dots X_n^{(1)}$ such that each $X_i^{(1)} \supset X_i^{(0)}$ and $X_i^{(1)} \cap X_j^{(1)} = \varnothing$ for $i \neq j$, so $X^{(1)} \supset X^{(0)}$. Define*

$$\mathcal{L}^{(1)}(\theta) = \frac{(1-\lambda)}{N_d} \sum_{i=1}^{N_d} \Big(\Lambda(x_i, t_i; \theta) - u(x_i, t_i) \Big) + \frac{\lambda}{N^{(1)}} \sum_{j=1}^{N^{(1)}} \Big(\mathcal{N}(\Lambda(x_j^{(1)}, t_j^{(1)}; \theta)) \Big) \quad (4.12)$$

where each $(x_j^{(1)}, t_j^{(1)}) \in X^{(1)}$ is a randomly sampled point inside $X^{(1)}$, and run until convergence or m iterations. Continue until $X^{(k-1)} \subset X^{(k)} = X_1^{(k)} \cup X_2^{(k)} \cup \dots \cup X_n^{(k)} = Y$, where $X_i^{(k)} \supset X_i^{(k-1)}$ and $X_i^{(k)} \cap X_j^{(k)} = \varnothing$ for $i \neq j$. Then, use the usual loss function that takes random samples from the whole domain.

$$\mathcal{L}^{(k)}(\theta) = \frac{(1-\lambda)}{N_d} \sum_{i=1}^{N_d} \Big(\Lambda(x_i, t_i; \theta) - u(x_i, t_i) \Big) + \frac{\lambda}{N^{(k)}} \sum_{j=1}^{N^{(k)}} \Big(\mathcal{N}(\Lambda(x_j^{(k)}, t_j^{(k)}; \theta)) \Big) \quad (4.13)$$

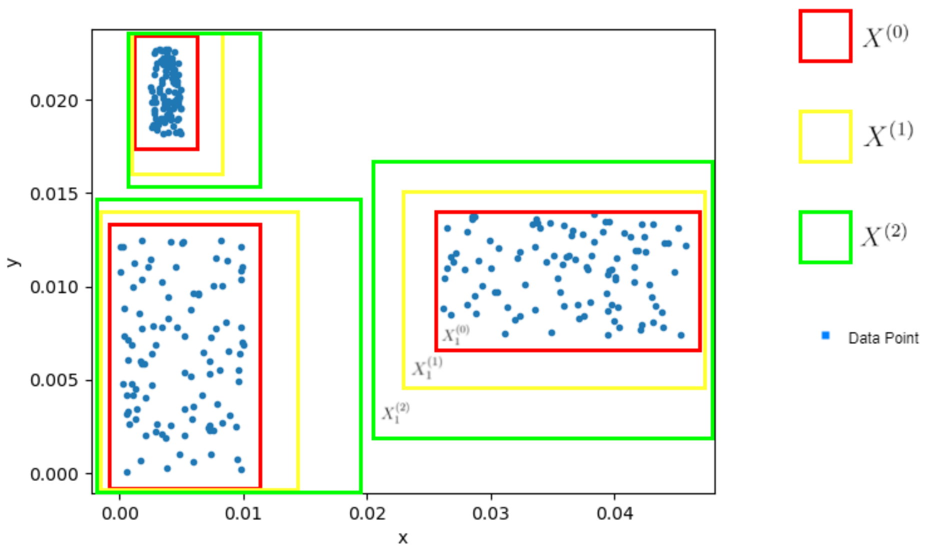

Figure 4.9: One such possible grouping of data for a data-set over a 2D domain. Each iteration of $X^{(i)}$ grows the accessible domain around each group of data. The algorithm would model behaviours inside each partition before expanding to behaviours in other parts of the domain

Theorem 1 suggests that the optimisation process should focus on approximating the function behaviour near data points before expanding the domain and resolving errors in places with little data. This is because areas that lack data rely on the accuracy of surrounding behaviours to model the solution correctly, so minimising errors in dense data areas should be the initial priority.

4.4.2 Error Patterns

The second point highlights that the absolute error is not randomly distributed through the domain, but appears to have structure. This structure looks to be consistent in each approximate solution, and is unaffected by the location of the data or the network structure.

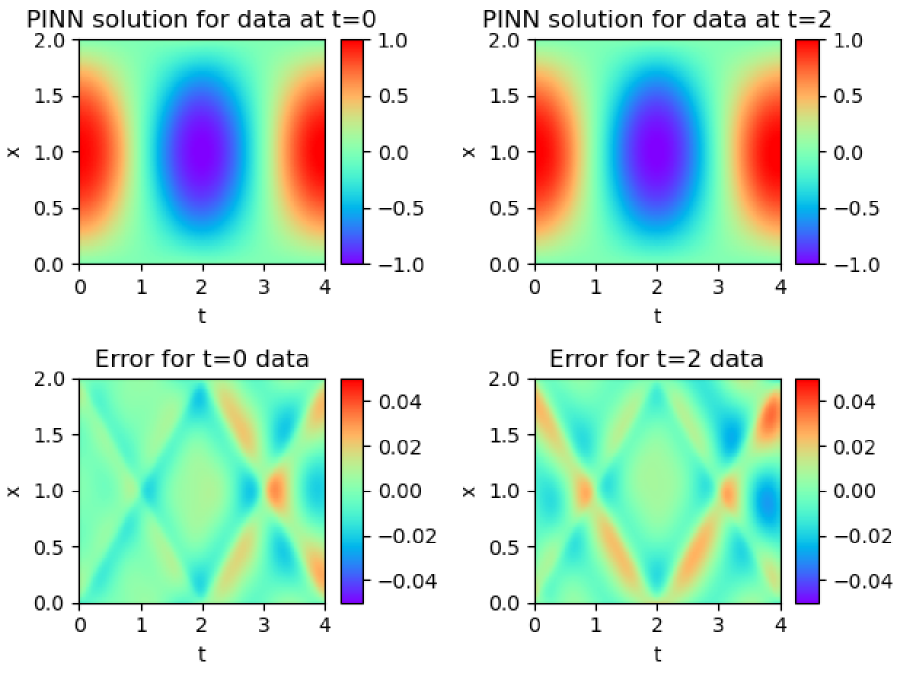

Figure 4.10: The approximate solution and error patterns for data sets from different domain locations. The pattern is independent of the data distribution

Since the physical behaviour of waves is reasonably intuitive to understand, it is easy to see that these errors seem to occur when the wave is at its 'curviest'. The errors seem to swap in a crossing pattern, going from the boundary, to the centre, and back. This is likely because of errors in the derivative.

- Errors appear at the boundary when $|u(x,t)| \to 1$ because this is when $\left|\frac{\partial u}{\partial x}\right|$ is at its largest (the wave exhibits maximum spatial curvature).

- Errors appear in the centre of the wave when $|u(x,t)| \to 0$ because this is when $\left|\frac{\partial u}{\partial t}\right|$ is at its largest (the centre of the string is experiencing large relative speed, or maximum temporal curvature).

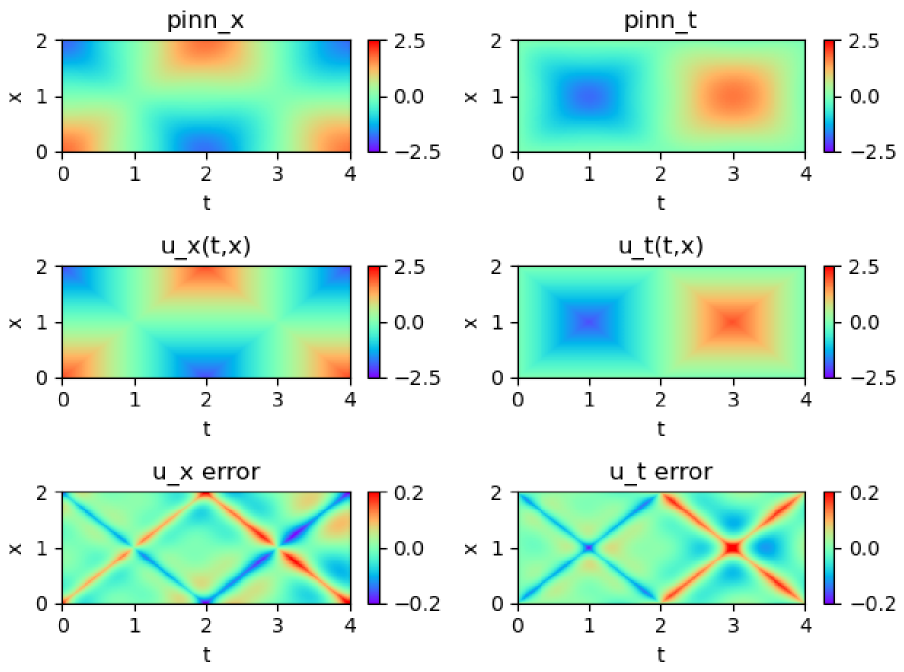

Figure 4.11: Errors between the analytical solution derivatives and the PINN solution derivatives for all data at $t = 0$

Errors in the derivative obviously follow the pattern seen in the approximate solution error. The errors also appear be of a higher relative magnitude, and (as before) it propagates away from the IC, with the best estimate of the overall derivative being where data is most dense.

This discovery is somewhat interesting, as it appears that analysing where rapid change occurs in the PINN could tell us information about where errors may appear without having to know the analytical solution. As a litmus test, the same analysis was performed on a quasi-linear PDE.

Quasi-linear PDEs are a special subclass of PDE in that they can develop *shock formations*[1]. As the solution tends towards a shock point, we often find very sharp changes in behaviour, or even discontinuities.

A famous example of a quasi-linear PDE is the viscous Burger's equation, which is used in modelling fluid mechanics, nonlinear acoustics, and gas dynamics. It is defined as

$$\phi_t + \phi\phi_x = \nu u_x x \tag{4.14}$$

where ν is the viscosity parameter. For small values of ν, the solution can develop shocks, and these can make the system very challenging to solve. Take the following

[1]Shocks form in solutions at t_s for PDEs when the point $u(x, t_s)$ has conflicting information on what value it should take for some x (ie, the solution becomes multi-varied for $t \geq t_s$). This often implies that the assumptions of the governing equations are not valid beyond this point

problem, described and solved in Raissi, M. et al. [50]

$$\begin{cases} \phi_t + \phi\phi_x - \frac{0.01}{\pi}\phi_{xx} = 0 & \text{for } 0 \leq t \leq 1, \quad -1 \leq x \leq 1 \\ \phi(x,0) = -\sin(\pi x) & \text{IC} \\ \phi(-1,t) = \phi(1,t) = 0 & \text{BCs} \end{cases} \tag{4.15}$$

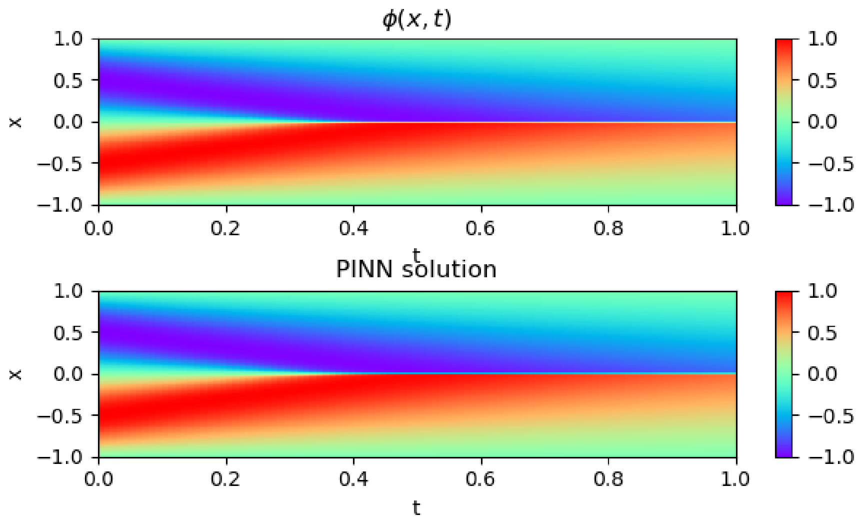

Figure 4.12: True solution to the Burger's equation and the PINN approximate solution

The data for the solution is taken directly from the cited paper. Clearly, the approximate solution looks to be a good fit, and the L_2 error was 4.9×10^{-4}. However, we are interested in where errors appear, if they do at all.

From looking at the form of the solution in figure 4.12, and following on from the logic described in the analysis of error locations for the 1D wave equation, we should expect 2 areas of error.

1. Small errors approximately for $t < 0.4$ in a delta shape (pointing right) due to the derivative with respect to time. The t derivative is largest here, but it isn't enormous.

2. Large errors for approximately $t > 0.4$, $x \approx 0$ due to the large derivative with respect to space. In fact, it almost becomes a discontinuity.

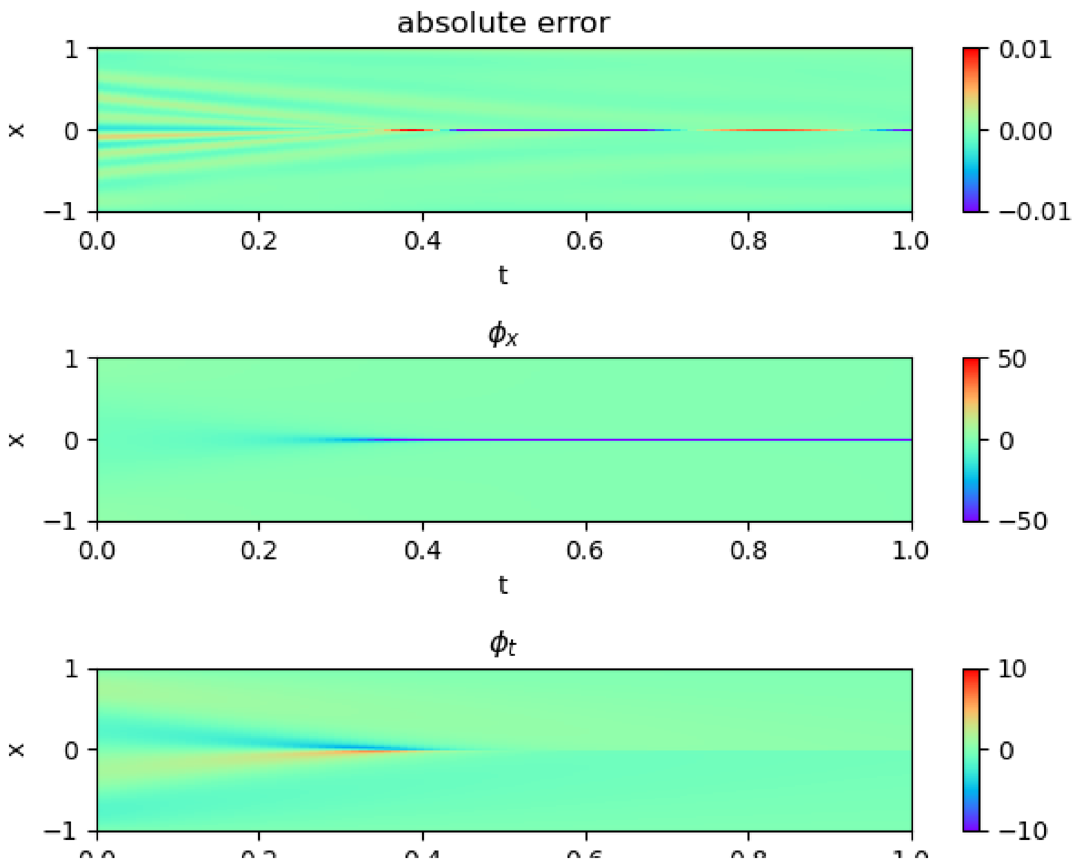

Figure 4.13: Top: The error between the analytical solution and the PINN approximate solution. Middle: the derivative with respect to space. The derivative at $x \approx 0$ dwarfs all others. Bottom: the derivative with respect to time. It is significantly smaller than the extremes in the spatial derivative

Clearly, the prediction was accurate. Despite the small L_2 error documented in the study, the error is almost entirely caused by the incredibly steep derivatives in the x direction. This reasons that we can improve the approximate solution with more intelligent data sampling.

Theorem 2 *Take $\Lambda(x,t)$ to be a physics-informed neural network which closely approximates the general behaviour of some function $u(x,t)$, and take $\epsilon > 0$ to be the acceptable tolerance to say the solution has converged. Take m to be the maximum number of iterations. If, on iteration m, we have*

$$\frac{1-\lambda}{N_d}\sum_{i=1}^{N_d}||u(x_i,y_i,t_i)-\Lambda(x_i,y_i,t_i)||_2^2+\frac{\lambda}{N_s}\sum_{j=1}^{N_s}||\mathcal{N}(\Lambda(x_j,y_j,t_j))||_2^2>\epsilon \tag{4.16}$$

then the network has failed to converge to a solution in finite time. Take $\Lambda(x,t)$ and calculate

$$\mathcal{J}_\Lambda = \nabla\Lambda(x,t) \tag{4.17}$$

such that the Jacobian $\mathcal{J}_\Lambda$ is the vector that holds the derivative of $\Lambda(x,t)$ with respect *to each variable. Since errors are most likely to occur when $\mathcal{J}_\Lambda$ is relatively large, we can use $\mathcal{J}$ as a probability density function for data sampling, instead of sampling uniformly across the domain. That is to say as*

$$\frac{||\nabla\Lambda(x_i,t_i)||_2}{\max||\nabla\Lambda(x,t)||_2} \to 1, \quad P[(x_i,t_i) \in N_s] \to 1 \tag{4.18}$$

where N_s is the physics term sampling data-set.

Fundamentally, if we suspect that, after training, the PINN approximation may have large errors due to derivatives we can introduce another training step (or, perhaps, it could be referred to as a PINN validation step). Due to the fact that errors are likely to be occurring in areas of high relative change (large derivatives), in late training phases we should sample from these areas more densely, and then check if the PINN is minimised over the whole domain.

The suggestion here is to use the derivatives in the domain as a surrogate for a probability density distribution. The higher the derivative is in a certain area of the domain, the more likely we are to sample a point from this location.

Chapter 5

Current Applications and Research Areas

PINNs have already been put to use in academic settings in order to produce neural networks that solve/estimate physical systems. These problems range from small scale problems, like MRI scans [40] and natural language processing[1] [41], to large scale problems, such seismic imaging and fluid dynamics.

PINNs can be utilised to solve the forward problem of estimating the solution to a PDE, or the inverse problem of approximating a function coefficient in a system of equations. Both types of problem will be explored here.

5.1 Geophysical Tomography

Geophysical tomography is the process of applying non-destructive strategies to investigate the properties and structures of subsurface terrain [42]. Many different types of techniques can be used to model sub-surface structures, depending on the size of the domain and the detail a geologist may need.

One widespread application, which can produce high resolution images, is seismic imaging, which uses strong acoustic waves measured at different points to construct an approximation of underground parameters. *Wavefield reconstruction inversion* (WRI) is one such method that is particularly popular [**?**].

5.1.1 Wavefield Reconstruction Inversion

WRI is an alternative approach to seismic imaging, and builds upon techniques established in *forward wavefield inversion* (FWI). FWI is a data-driven, constrained, non-linear optimisation method that uses the known physics of the acoustic wave equation

[1]NLP is the computational technique of taking in continuous data (such as wavelengths, amplitudes, etc) and analysing the data in such a way that a computer can and interpret individual words and speech

to reconstruct features from partial measurements of the wave equation, specifically in frequency-domain, rather than time-domain [?]. The preference of using frequency domain over time domain is that waves of different frequencies behave differently depending on the properties of the medium it is passing through. They can reflect, refract, sheer, and penetrate. The frequency domain wave (Helmholtz) equation is

$$(\nabla^2 + \frac{\omega^2}{c(x)^2})u(x,\omega) = q(\omega)\delta(x - x_s) \tag{5.1}$$

where u is the acoustic seismic wavefield in frequency domain, ω is the angular frequency, x is the spatial location, c is the velocity of the medium, and $q(\omega)\delta(x - x_s)$ is the frequency domain source at $x = x_s$ (so $\int_{-\infty}^{\infty} q(\omega)\delta(x - x_s)dx = q(\omega)$). In short hand, the discrete Helmholtz operator A (also known as the impedance matrix) will be such that

$$A(m,\omega) = L^2 + \omega^2\text{diag}(m) \tag{5.2}$$

where L^2 is the discretised Laplacian operator ∇^2, and m is the discrete squared slowness of the medium $\frac{1}{c(x)^2}$.

The basic premise is that acoustic waves are fired into the surface, and the wavefield response is measured by remote sensors, often on or near the surface of the terrain. Each sensor d_r collects data points from each acoustic source q_s.

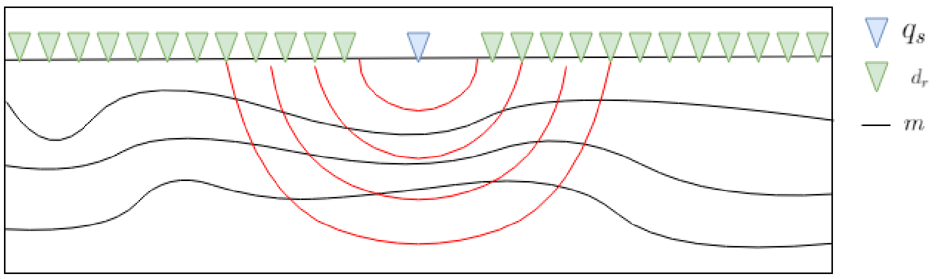

Figure 5.1: A basic representation of imaging using seismic waves. Waves (in red) are propelled through the medium m (in most cases, earth) from a source q_s, and sensors d_r take measurements of the wave response on the surface. The response changes depending on the structure of the medium

Using this data, a constrained optimisation problem can be obtained, such that

$$\min_{m,u} \frac{1}{2} \sum_{\omega} \sum_{s,r} ||\Gamma_{s,r}u_s(\omega) - d_{s,r}||_2^2 W_{s,r} \qquad \text{s.t. } A(m,\omega)u_s(\omega) = q_s(\omega) \tag{5.3}$$

Where $W_{s,r}$ is a weight operator applied to the data residual. The constraint can be eliminated by using inner products and Lagrangian multipliers, called the *adjoint-state* or *reduced-space* formulation such that

$$\mathcal{L}(m, u_s, v_s) = \frac{1}{2} \sum_{s,r} ||\Gamma_{s,r}u_s - d_{s,r}||_2^2 W_{s,r} + \sum_{s} \langle v_s, A(m)u_s - q_s \rangle_x \tag{5.4}$$

where $\langle \cdot, \cdot \rangle_x$ is the inner product on spatial coordinates, and v_s is the Lagrange multiplier. Taking the partial derivative with respect to each minimisation parameter yields

$$\begin{aligned}
\frac{\partial \mathcal{L}(m, u_s, v_s)}{\partial m} &= \sum_s \left(\frac{\partial A(m) u_s}{\partial m} \right)^* v_s \\
\frac{\mathcal{L}(m, u_s, v_s)}{\partial u_s} &= \sum_{s,r} \Gamma^*_{s,r} W^*_{s,r} (\Gamma_{s,r} u_s - d_{s,r}) + \sum_s A^*(m) v_s \\
\frac{\partial \mathcal{L}(m, u_s, v_s)}{\partial v_s} &= \sum_s A(m) u_s - q_s
\end{aligned} \tag{5.5}$$

where $*$ is the conjugate transpose. The last equation implies that if we are at a minimum then $A(m)u_s = q_s$, and therefore the constraint in the original minimisation problem is satisfied. To be concise, solving the wave equation and the adjoint equation each iteration yields the gradient, which can then be used to calculate a local minimum.

However, this type of inversion is plagued by three main problems:

1. Solving the wave equation for all space twice per iteration is computationally taxing, and the constructed Hessian is usually dense.
2. There is an extreme non-linear dependence on the earth model m
3. The problem is nonconvex, and so solutions are very rarely unique. Thus, the optimal solution found is dependent on the starting parameters of the earth model m. In other words, local minima are real complication. This is often called cycle skipping, and occurs when the data generated by the model $\Gamma_{s,r} u_s$ is more than half a cycle away from the recorded data $d_{s,r}$.

WRI tackles this problem by giving the objective function more degrees of freedom. It achieves this by relaxing the physics constraint slightly by some weight λ^2 [44], and is instead added directly into the objective function as a penalty parameter.

$$\min_{m,u_s} \mathcal{L}_\lambda(m, u_s) = \sum_{s,r} \left(||\Gamma_{s,r} u_s - d_{s,r}||_2^2 + \lambda^2 ||A(m) u_s - q_s||_2^2 \right) \tag{5.6}$$

In other words, we expect the optimal earth solution to minimise a balance between the physics and the data, rather than perfectly conforming to the physics. This removes the need to solve the adjoint wave equation, and the Hessian is sparse if λ is small (with small λ being interpreted as not relying on the physics as much) [45]. Recently, WRI has been formulated in time-domain problems, rather than frequency domain, to alleviate some computational cost (as LU factorisation can be more easily leveraged to solve the many linear systems) [46].

5.1.2 Solving the Wave Equation with PINNs

The two cornerstones of wavefield inversion are

1. Constructing a wavefield that satisfies the source data and the receiver data.

2. Using the estimated wavefield to reconstruct the terrain under the surface boundary

Physics-informed machine learning has shown great promise in accurately reconstructing wavefields for complex domains, as shown by the University of Oxford in *Solving the Wave Equation with Physics-informed Deep Learning* (B. Mosely et al.) [47]. The paper applies the methods and strategies highlighted in chapters 2 and 3 to estimate the acoustic response to waves propagated through mediums of varying complexity.

Method

The study used the 2D acoustic wave equation (as described in equation 3.1) to create a loss function

$$\mathcal{L} = \frac{1}{N_u}\sum_{i=1}^{N_u} ||u(t_i, x_i, s_i) - \Lambda(t_i, x_i, s_i)||^2 + \frac{1}{N_\Lambda}\sum_{j=1}^{n_\Lambda} ||\mathcal{N}(\Lambda(t_j, x_j, s_j; \theta); \lambda)||^2 \tag{5.7}$$

where (t_i, x_i) are known initial values, (t_j, x_j) are points sampled inside the whole domain, Λ is the PINN, λ is a weight function for the physics minimisation, θ are the trainable parameters that define Λ, and $\mathcal{N}$ is the differential operator that describes the acoustic wave equation, as defined in equation 3.3. Surprisingly, AD was not used in order to accurately calculate the gradients needed for the physics term in the loss function, but the authors do note that it could (and, really, should) be used.

Network architecture, and how different architectures may change the solution, was not a variable that was explored in this study. Motivated by Raissi et al. (2019), the activation function $\sigma : \mathbb{R} \to [0, \infty]$ used was the SoftPlus activation function

$$\sigma(x) = \ln(1 + e^x) \tag{5.8}$$

and has a derivative equal to the logistic map, $\sigma'(x) = \frac{1}{1+e^{-x}}$. It is often used because it has a reduced likelihood of vanishing gradients. The PINN design was fixed at 10 layers with 1024 hidden channels, and was fully connected. The input space was the spatial coordinates $x \in \mathbb{R}^2$, the temporal coordinate $t \in \mathbb{R}$, and the source term s, with the output being the wavefield $u \in \mathbb{R}$. Each layer was put through the activation function, except the final layer, which was just kept as linear (otherwise the network output would always be $\Lambda(x, t) > 0$).

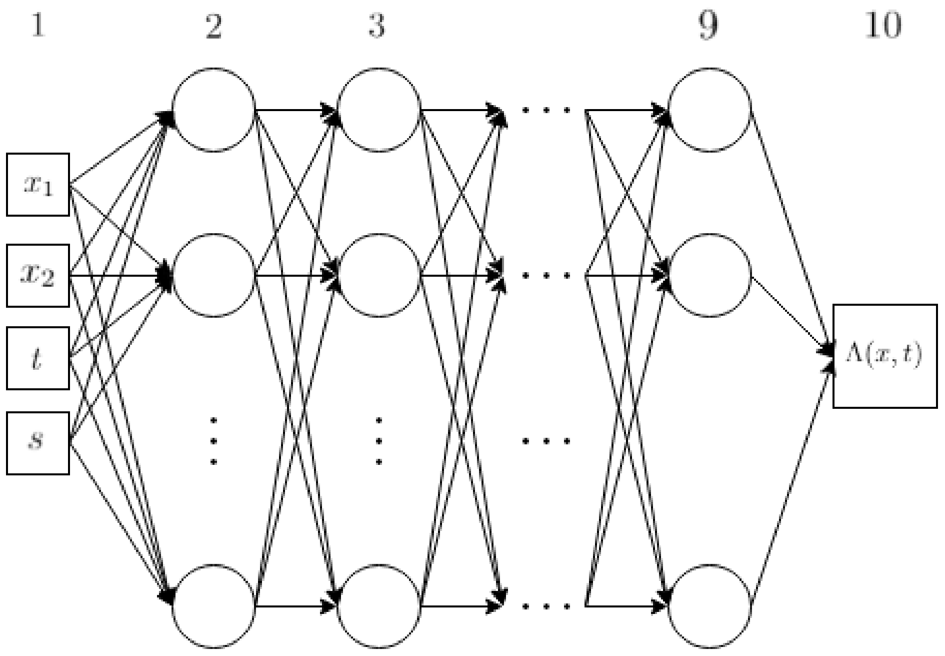

Figure 5.2: Network architecture for solving the wave equation. 10 layers make up the network, with a total of 1024 nodes. The final output layer was linear, rather than using a non-linear activation function.

The study used the ADAM optimisation algorithm to calculate a minimum for the network weights and biases.

Data

There were two types of data needed for this study: *initial wave* data and *medium velcoity* data. Since the study was only concerned with the forward problem of modelling the wavefield, the medium density and velocity was known prior. The study used 3 data sets of varying complexity.

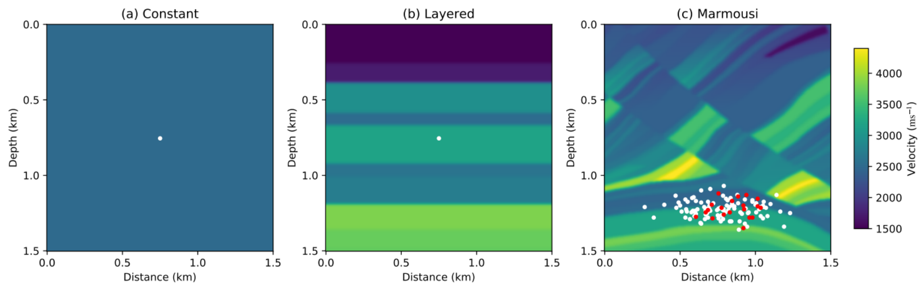

Figure 5.3: Different data sets describing the medium velocity. White points are sources that generate acoustic waves

Data-set 1 was a medium of constant velocity, data-set 2 had medium velocity varying spatially in the vertical direction (so a pseudo-stratified velocity), and data-set 3 is from the famous Marmousi model[2], which has velocities varying in all spatial directions.

[2]The Marmousi model data-set was created in 1988, and has become an industry standard data-set

In order to calculate sufficient ICs for training, finite difference methods were used for the first T_1 training steps to create a discrete wavefield. Therefore, the boundary data is from the set $x_i \in [0, X_{\max}]$ and $t_i \in [0, T_1]$, where $T_1 \ll T_{\max}$. For the constant and stratified data-sets, $T_1 = 0.02$ seconds, which was the first 10 time steps. For the Marmousi data-set, $T_1 = 0.04$ seconds, so the first 20 times steps were used, as were multiple simulations from multiple sources.

The training was split into two distinct phases: the boundary phase and the physics phase. Phase 1 allows the network to prioritise minimising the boundary loss term without the interference of the physics loss. Half way through the iterations (at step 500'000) the physics loss term was introduced, allowing the network to minimise over the rest of the domain.

Result

A full FD simulation was run for each data-set, which represents the 'ground truth', and the error is the difference between the FD model and the PINN model. The aim of the study was to see how well a PINN could model wave propagation through a known 2D velocity domain, and how much better it performed vs a standard ANN.

Standard ANN have tended to capture the outward propagating wave, but usually fail to capture more subtle behaviours such as sheer, reflection, and refraction, and thus cannot generalise the solution far outisde the training domain (so for $t > T_{\max}$). This problem is significantly lessened by introducing the physics loss.

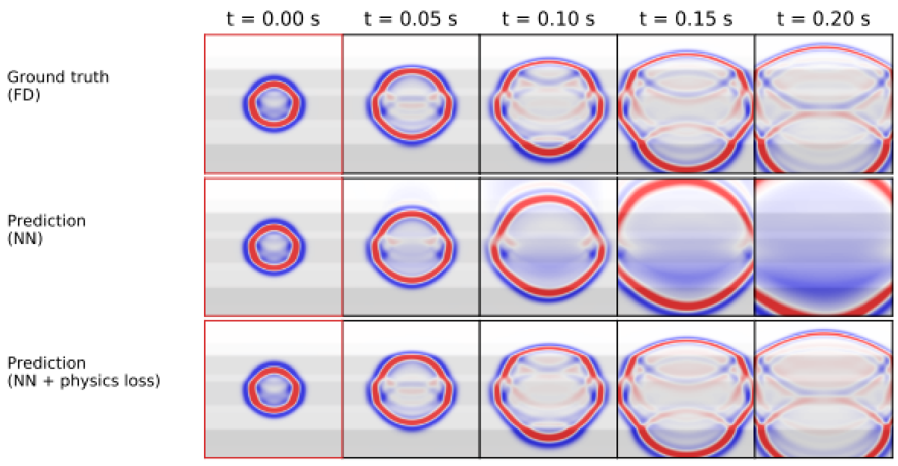

Figure 5.4: Wave behaviour estimate for the stratified data-set using 3 methods: FD, ANN, PINN. The ANN fails to capture many of the reflected wave behaviours at larger t, but does model the leading behaviour

The errors in the ANN approach tend to occur at fault lines (where the velocity changes significantly). This is likely because the ANN is given no information for how the wave

for testing seismic imaging techniques. Originally created by the Institut Francais du P´etrole, the data is synthetic, and is based on the sub-terrain structure found in the North Quenguela trough in the Cuanza basin field should change through time and space as the velocity of the medium changes, and so cannot learn the properties of reflected waves, etc, without sufficient data (and this data is often lacking in real applications).

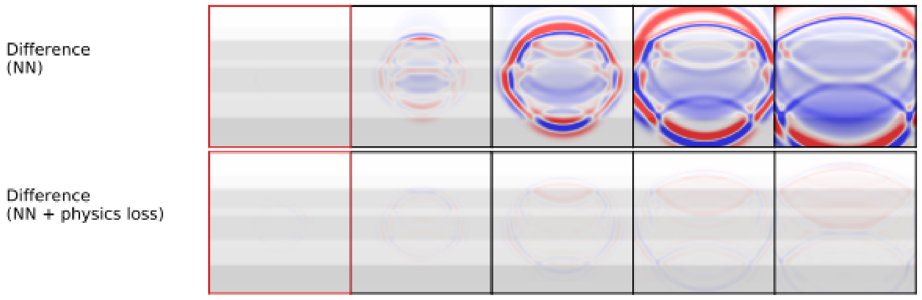

Figure 5.5: The difference between the FD simulation and each of the two networks. The error between the PINN and the FD simulation is almost negligible

The authors highlight that the greatest advantage of using a PINN over a FD simulation is that the solution is much better generalised. As an example, for the 3rd data-set multiple sources were used to create a PINN. If one wanted to estimate the wavefield generated from a new source for the same domain at, say, $t = 0.4$, the FD method would require simulating all time over all space up to the required time using the new source term. Due to the fact that the PINN solution is more generalised, further simulations are not required to find this value, and one can simply entire the time, space, and source information as required. This means that using a trained PINN is in the order of $\times 1000$ faster than using FD.

The PINN approach is also easily scalable, as adding dimensions, changing data-sets, and altering the physics is very simple to do. However, the discontinuities in the medium velocity can still cause errors to appear in the solution, though this is also a problem for FD methods. To avoid this issue, the 3rd data-set was smoothed slightly where two different velocities interface. There are three potential solutions to this problem:

1. Use individual PINNs for each section of the subsurface terrain, which would allow for discontinuities in the medium.

2. Introduce a loss term that allows for wavefield seperation.

3. Use a more intelligent sampling scheme for the data, so areas of high velocity change are sampled more frequently during minimisation, as dicussed in the previous chapter.

The conclusion, however, is that PINNs can learn many of the intricate behaviours that waves exhibit, can create a continuous solution that can accurately estimate values well outside of the training data, and are extremely quick to use once trained.

5.1.3 The Inverse Problem

Since it has been shown that PINNs can find accurate solutions for the wavefield, the more crucial question is whether or not they can be used as a substitute to solve the inverse problem - finding the distribution of medium velocity. Chao Song et al. from King Abdullah University (2021) [48] applied the PINN described in chapter 3 to better inform the WRI method.

Method

The key idea behind this study is that two PINNs were used to reconstruct the velocity field.

1. The first PINN reconstructed the scattered wavefield on a per frequency basis.
2. The second PINN used the reconstructed wavefield from the first PINN to make changes to the velocity field (squared slowness).

Instead of using the time-domain wave equation, the study utilised the Lippmann Schwinger form of the acoustic wave equation [49]

$$\omega^2 m \delta u + \nabla^2 \delta u = i\omega^2 \delta m u_0 \tag{5.9}$$

which finds the scattered wavefield $\delta u = u - u_0$ for differing angular frequencies ω, where u is the wavefield and

$$u_0(x) = iH_0^{(1)}(\omega\sqrt{m_{n0}}|x - x_s|) \tag{5.10}$$

for isotropic, 2D domains, is the background wavefield, with x_s being a point source location. $\delta m = m - m_0$ is the perturbed squared slowness, with m_0 being the slowness of the homogeneous earth model. $H_n^{(k)}$ is a kth kind Hankel function, defined as

$$H_0^{(1)}(z) = \frac{1}{i\pi}\int_0^\infty \frac{e^{(\frac{z}{2})(\frac{t-1}{t})}}{t}dt \tag{5.11}$$

which has real and imaginary parts.

Using this set of equations, the physics-informed loss function (which is referred to as MSE in this particular piece of literature) for the forward problem was defined in the standard way of a balance between the physics and the data

$$\begin{gathered}\mathrm{MSE}_f = \sum_{is}^{N_s}\left(\frac{1}{N_r}\sum_{i=1}^{N_r}||\delta d^i - \delta u(x_r^i)||_2^2 + \right.\\ \left.\frac{\alpha}{N}\sum_{i=1}^{N}||\omega^2 m_1^i \delta u^i + \nabla^2 \delta u^i + \omega^2(m_1^i - m_0^i)u_0^i||_2^2\right)\end{gathered} \tag{5.12}$$

where the first term is the data loss term, and the second term encapsulates the physics of the system (via the Lippmann Schwinger equation). The adjustable parameters in the equation define the behaviour of the scattered wavefield δu. Each is is a source location, $\delta d = d^i - u_0(x = x_r)$ is recorded scattered data, and m_1 is the initial squared slowness model. α is a weight term, which was set to 0.00001 (simply through trial and error testing), and N is the number of points selected for the physics constraint.

The physics informed loss function for the inverse problem was defined as

$$MSE_b = \frac{1}{N_p}\sum_{i=1}^{N_p} ||\omega^2 m^i \delta u^i + \nabla^2 \delta u^i + \omega^2(m^i - m_0^i)u_0^i||_2^2 + \epsilon\sqrt{\left(\frac{\partial m^i}{\partial x}\right)^2 + \left(\frac{\partial m^i}{\partial z}\right)^2} \tag{5.13}$$

which, noticeably, has no data term. The second term, called the total-variation term (TV), was added to stabilise the training process, with $\epsilon = 0.1$. Here, the scattered wave δu is the PINN solution for a fixed frequency. The inverse PINN uses the forward PINN to then make changes to the earth model m. In truth, the algorithm is deceptively simple.

Algorithm 3 PINN-based WRI

```
Require: $d$                                   ▷ The observed data-set
Require: $m_0$                                 ▷ The background squared slowness model
Require: $m_1$                                 ▷ An initial squared slowness model
Require: $u_0$                                 ▷ The background wavefield
Require: $n$                                   ▷ Maximum number of iterations
Require: $[f_{\min} : df : f_{\max}]$          ▷ Selected angular frequencies
Require: $\alpha$                              ▷ Weight for physics term in forward solver
  for $f = [f_{\min} : df : f_{\max}]$ do
      for $i = 1 : n$ do
          Reconstruct the scattered wavefield for fixed frequency
          Reconstruct velocity model using reconstructed scattered wavefield
      end for
  end for
  return Inverted velocity model
```

The velocity model is therefore continually improved by modelling the scattered wavefield of differing frequencies.

Due to the complexity of the task, size of the domain, and dynamic nature of the domain, the networks used in this study were quite large. The forward (scattered wavefield) PINN used a descending hierarchy structure with 8 hidden layers, ordered as $[128, 128, 64, 64, 32, 32, 16, 16, 8, 8]$, whereas the inverse (velocity) PINN used a flat structure with 8 hidden layers, ordered as $[20, 20, 20, 20, 20, 20, 20, 20]$. The input for the scattered wavefield PINN was $\mathbb{R}^3$, containing a 2D location x and a source term

x_s, and the output was $\mathbb{R}^2$ which were the real and imaginary parts of the scattered wavefield. The input for the velocity model PINN was a 2D point, and the output was simply the velocity.

The networks were trained using the 20'000 iterations of the ADAM optimisation algorithm, followed by 150'000 iterations of L-BFGS.

Data

The method was tested on two data-sets, but we will focus on 2D Marmousi model for two reasons

1. The data-set was used in the previous study, so gives a good point of comparison.

2. It was the more complex of the two data-sets, and so is more interesting to analyse.

The data consisted of

1. 10 sources, spaced evenly on the surface of the terrain.

2. 301 receivers, spaced evenly at a depth of 25m, which collect data for the data loss term.

3. 80'000 randomly sampled points inside the domain, which collect points for the physics loss

The initial velocity model was generated in such a way that the velocity increased as z increase (so a stratified model, not too dissimilar to the second data-set used in the previous study).

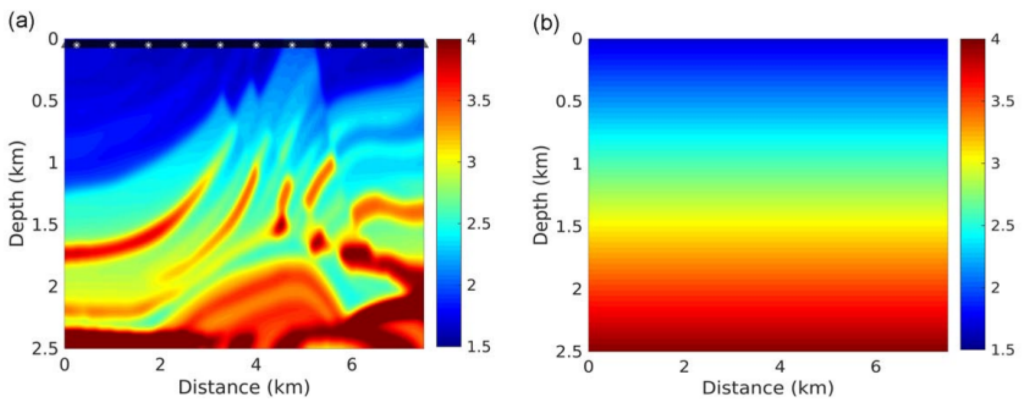

Figure 5.6: Left: True Marmousi velocity model. Right: Initial velocity model for PINN WRI algorithm

Result

The resulting velocity field after performing PINN-based WRI for 3Hz, 4Hz, and 5Hz (3 sets of angular frequencies) produced an earth model that had some characteristics of the true solution, but was far too smooth. This follows from the experimental data that was analysed in the previous chapter. The error here is caused by the scattered wavefield being too smooth. In other words, areas with large changes in derivative had a tendency to have high errors. However, despite the fact that the velocity model from PINN-based WRI wasn't a fantastic approximation of the true solution, it did prove to be incredibly useful.

In section 5.1.1, the problems and shortcomings of FWI and WRI were discussed in detail. One of these shortcomings is that the the minimisation problem is often plagued by local minimums, and that the final solution is dependent on the initial earth model provided. Producing a good inital earth model is incredibly challenging, and is often somewhat impossible. As such, a stratified initial mode (such as in figure 5.6) is often used.

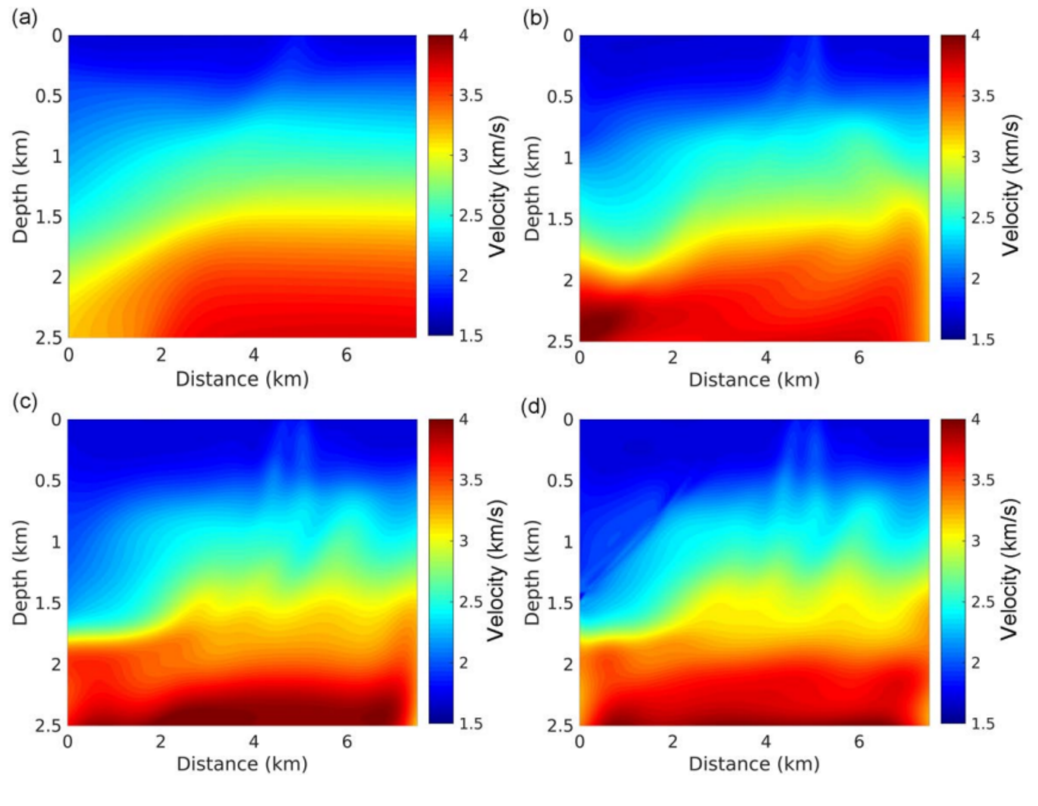

Figure 5.7: Top left: PINN velocity model after 1 iteration using 3Hz. Top right: PINN velocity model after 4 iterations of 3Hz. Bottom left: PINN velocity model after 4 iterations of 4Hz. Bottom right: Pinn velocity model after 3 iterations of 5Hz

The solution provided by the PINN-based WRI, whilst being too smooth to realise high resolution details, does capture some of the general behaviour of the true 2D Marmousi model. It is certainly a better fit than the initial velocity model. The final velocity model created by the PINN-based WRI was then used as the initial earth model for FWI, and its solution was compared to the solution of using the stratified data as the initial model.

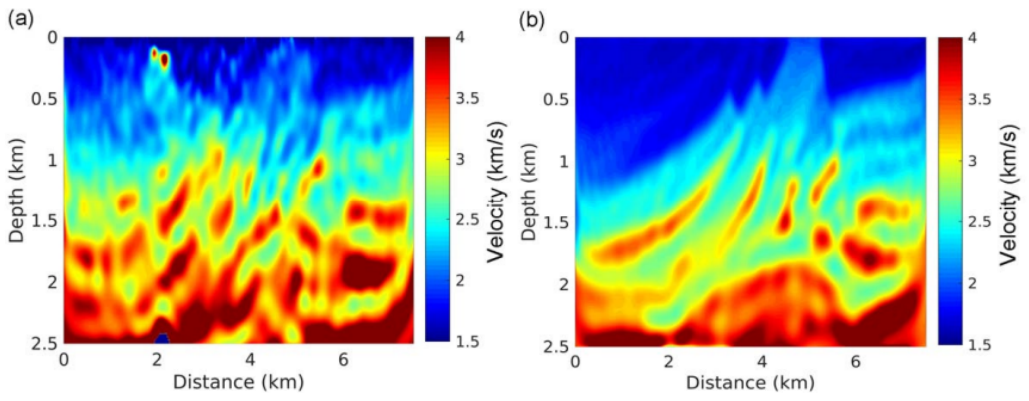

Figure 5.8: Left: Estimated velocity model from stratified initial velocity model using FWI. Right: Estimated velocity model from PINN solution initial velocity model using FWI.

Using the PINN WRI solution as the initial model gave a significantly better final velocity model than using the stratified model. It still appears a little too smooth, but the main features of the solution certainly match the true data.

In comparing the PINN method with other numerical techniques, it was commented that, whilst FD methods work quickly for small scale problems, they are not well suited to large scale, high dimensional problems. This is because they cannot be calculated in parallel like a PINN algorithm can be. Further more, adding more complexity to the properties of the domain, such as having vertical transversely isotropic media, or isotropic elastic media dramatically increases the size of the impedance matrix used for FD methods.

Finally, as was mentioned in the previous study, the flexibility of this method cannot be understated. In order to make WRI and FD methods work for a higher dimensional domain, where the sub-surface structure is more complex (eg, has fluid-saturated porous sections), the algorithms would need huge modifications, whereas the PINN approach simply requires altering the loss function to include differing equations.

The major shortcomings here are that the scattered wavefield solutions are overly smooth, which in turn leads to an inverted velocity model that is also too smooth. One solution is to potentially increase the size of the network, allowing for more degrees of freedom, or use an individual network for each frequency (which could then be done in parallel). However, this also obviously increases computational cost considerably.

Chapter 6

Conclusion

Due to the creation of packages like Tensorflow and Pytorch in python, and flux in Julia, creating artificial neural networks has never been easier. On top of this, memory has become incredibly inexpensive and, because of the parallel nature of many of the operations when using a neural network, large and powerful GPUs can be easily leveraged to achieve incredibly fast computing times. As such, minimising the loss function for high dimensional problems with many parameters is significantly faster than it was just a few years ago.

Thanks to advances in computational differentiation, it has been shown that the loss function for artificial neural networks can be created in such a way as to accurately approximate the solution for physical systems in parts of the domain where no data exists. Further more, these *universal approximators* can be cleverly designed in order to have explicit access to known behaviours, such as ICs and BCs.

The solutions that physics-informed neural networks produce are continuous functions. They therefore have the superior characteristic of being able to produce values for inputs that are outside of the domain or beyond known data, unlike other numerical methods, and they do not require any further calculations in order to refine the solution (such as smaller steps sizes in finite difference methods, or a finer mesh in finite element analysis).

Furthermore, a deep dive into error analysis has shown how, why and where errors may occur in a physics-informed neural network approximation. These errors tend to occur for two reasons:

1. The nearest pocket of dense data is relatively far away.

2. The relative derivatives in this part of the domain are large.

These errors occurred both in the ideal wave system in chapter 4, and the wavefield reconstruction inversion method in chapter 5. As such, we can estimate points of large error in the approximate solution, and potentially employ data sampling techniques to improve the accuracy of these areas using the physics loss term. This pattern of high derivative and high error is consistent across recent studies, both in the forward problem and in the inverse problem, and is the main limitation when using physics-informed neural networks

It is not suggested that physics-informed neural networks replace conventional methods for solving PDEs, but instead act as another tool one could use to solve a problem or even validate a solution. As is shown in chapter 5, PINN solutions certainly have there place in solving real world problems, and these uses still require deeper and broader experimentation to find the limit of their applications. These networks are particularly useful in complex systems as they are incredibly flexible and easy to modify to incorporate new physics, new data, or new dimensions.

Further work certainly needs to be done in error analysis and convergence rate. The groundwork is laid out in this thesis, but still requires more rigorous proof and theory. A better understanding of how a solution is converged to could prevent problems such as local minimums, which already plague many of the optimisation problems we face today. Errors clearly manifest themselves in pre-defined ways, and as such they could be tackled more effectively during the minimisation process, rather than ex post facto.

Bibliography

[1] Samuel, Arthur L. (1959). '*Some Studies in Machine Learning Using the Game of Checkers*'. IBM Journal of Research and Development. 44: 206–226. doi:10.1147/rd.441.0206

[2] Rosenblatt, F. (1958). '*The Perceptron: A Probabilistic Model for Information Storage and Organisation in the Brain*'. Psychological Review, Vol. 65, No. 6: 386–408.

[3] Swetz, Frank J. (2017). '*Mathematical Treasure: Leibniz's Papers on Calculus - Integral Calculus*, Convergence, Mathematical Association of America: 292-300.

[4] Archibald, T. Fraser, C. Grattan-Guiness, I. (2004) '*The History of Differential Equations, 1670–1950*'. European Mathematical Society, Vol. 1, Issue 4: 2729-2794. doi: 10.4171/OWR/2004/51

[5] Babuska, I. Szabo, B. (1991) '*Introduction to the Finite Element Method*'. Wiley: 1-50.

[6] Bayes, T. (1763) '*An Essay Towards Solving a Problem in the Doctrine of Chances*'. The Royal Society Publishing, Vol. 53, doi: https://doi.org/10.1098/rstl.1763.0053

[7] Revuz, D. (1984) '*Markov Chains*'. Elsevier, Vol. 11: 8-39, doi: https://doi.org/10.1016/S0924-6509(08)70188-0

[8] Cybenko, G. (1989) '*Approximation by Superpositions of a Sigmoidal Function*'. Springer, Mathematics of Control,Signals, and Systems Vol. 2: 303-314, doi: https://doi.org/10.1007/BF02551274

[9] Turing, A. M. (1950) '*Computing Machinery and Intelligence*'. Springer, Mind Vol. 59, Issue 236: 433-460, doi: https://doi.org/10.1093/mind/LIX.236.433

[10] L. Rabiner and B. Juang. (1986) '*An introduction to hidden Markov models*'. EEE ASSP Magazine, vol. 3, no. 1: 4-16. 433-460, doi: 10.1109/MASSP.1986.1165342

[11] Boutros1, F. Damer, N. Kirchbuchner, F. Kuijper, A. (2021) '*ElasticFace: Elastic Margin Loss for Deep Face Recognition*'. doi: arXiv:2109.09416

[12] Grigorescu, Sorin, Trasnea, Bogdan, Cocias, Tiberiu, Macesanu, Gige (2020) '*A survey of deep learning techniques for autonomous driving*'. Wiley, Journal of Field Robotics, Issue 37: 362–386 doi: arXiv:http://dx.doi.org/10.1002/rob.21918

[13] Perlich. Claudia. Dalessandro. Stitelman, B. Raeder, O. Provost, T. F. (2013) '*Machine Learning for Targeted Display Advertising: Transfer Learning in Action*'. Springer, Machine learning (Online), Vol. 95: 362–386 doi: 10.1007/s10994-013-5375-2

[14] Baum, E. B. (1988) '*On the Capabilities of Multilayer Perceptrons*'. Elsevier, Journal of Complexity, Vol. 4: 193–214 doi: 10.1016/0885-064X(88)90020-9

[15] Zhu, W. Miao, J. Qing, L. (2013) '*Extreme Learning Machines 2013: Algorithms and Applications*'. Springer, Extreme Support Vector Regression: 25-28

[16] Rauch, Jeffrey (1997) '*Partial Differential Equations*'. Springer, Graduate Texts in Mathematics, Vol. 128: 1–2

[17] Mamun, Abdulla - Al (2018) '*A study on an analytic solution 1D heat equation of a parabolic partial differential equation and implement in computer programming*'. International Journal of Scientific and Engineering Research, Vol. 9, Issue 9: 913-921

[18] Sobolev, S. L. (1964) '*Lecture 1 - Derivation of the Fundamental Equations*'. Partial Differential Equations of Mathematical Physics: 1-21 doi: https://doi.org/10.1016/B978-0-08-010424-9.50007-X

[19] Kirkwood, J, R. (2013) '*Fourier Series*'. Mathematical Physics with Partial Differential Equations: Elsevier: 187-212, doi: https://doi.org/10.1016/B978-0-12-386911-1.00004-5

[20] Miller, P. D. Perry, P. A. (2019) '*Nonlinear Dispersive Partial Differential Equations and Inverse Scattering*'. Mathematical Physics with Partial Differential Equations: Elsevier: doi: 10.1007/978-1-4939-9806-7

[21] Fefferman, C. L. '*Existence and Smoothness of the Navier-Stokes Equation*', problem description available via the Clay Mathematics Institute

[22] Stynes, M. O'Riordan, E. Gracia, J. L. (2018) '*Error Analysis of a Finite Difference Method on Graded Meshes for a Time-Fractional Diffusion Equation*'. Society for Industrial and Applied Mathematics, Vol. 55, No. 2: 1057-1079

[23] Utku, S. Melosh, R. J. (1984) '*Solution errors in finite element analysis*'. Elsevier, Computers and Structures, Vol. 18, Issue 3: 379-393 1057-1079, doi: https://doi.org/10.1016/0045-7949(84)90058-0

[24] Wunderlich, W. Cramer, H. Steinl, G. (1998) '*An adaptive finite element approach in associated and non-associated plasticity considering localization phenomena*'. Elsevier, Studies in Applied Mechanics, Vol. 47: 293-308, doi: https://doi.org/10.1016/S0922-5382(98)80016-0

[25] Berry, M. W. Mohamed, A. Bee, W. Y. (2020) '*Supervised and Unsupervised Learning for Data Science*'. Springer: v-vi

[26] Janocha, K. Czarnecki, W. M. (2017) '*On Loss Functions for Deep Neural Networks in Classification*'. Elsevier, DeepMind,: 1-10, doi: arXiv:1702.05659

[27] Shewchuk, J. R. (1994) '*An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*'. Carnegie Mellon University, School of Computer Science, Edition 1

[28] Sakata, S. Ashida, F. Zako, M. (2010) '*Comparative study on gradient and Hessian estimation using the Kriging method and neural network approximation*'. Elsevier, Mathematical and Computer Modelling, Vol. 51: Issues 3-4: 309-319 doi: https://doi.org/10.1016/S0922-5382(98)80016-0

[29] Milan, M. Militky, J. (2011) '*Nonlinear Regression Models*'. Woodhead Publishing India, Statistical Data Analysis: 667-762 doi: https://doi.org/10.1533/9780857097200.667

[30] Ruder, S. (2017) '*An overview of gradient descent optimization algorithms*'. Insight Centre for Data Analytics, doi: arXiv:1609.04747

[31] Sakata, S. Ashida, F. Zako, M. (2007) '*A Stochastic Approximation Method*'. Annals of Mathematical Statistics, Vol. 22: 400-407 doi: 10.1214/aoms/1177729586

[32] Kingma, D. P. Ba, J. (2017) '*Adam: A Method for Stochastic Optimization*'. doi: arXiv:1412.6980

[33] Qina, N. (1998) '*On the momentum term in gradient descent learning algorithms*'. Elsevier, Neural Networks, Vol. 12, Issue 1: 145-151 doi: https://doi.org/10.1016/S0893-6080(98)00116-6

[34] Xiao, Y. Wei, Z. Wange, Z. (2008) '*A limited memory BFGS-type method for large-scale unconstrained optimization*'. Elsevier, Computers and Mathematics with Applications, Vol. 56, Issue 4: 1001-1009 doi: https://doi.org/10.1016/j.camwa.2008.01.028

[35] Sherman, J, Morrison, W. J. (1950) '*Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix*'. The Annals of Mathematical Statistics, Vol. 21, Issue 1: 124-127 doi: 10.1214/aoms/1177729893

[36] Shi, Z. J. She, J. (2003) '*A gradient-related algorithm with inexact line searches*'. Elsevier, Journal of Computational and Applied Mathematics, Vol. 170: 349–370 ISSN: 0377-0427

[37] Asl, A. Overton, M. L. (2017) '*Analysis of the Gradient Method with an Armijo-Wolfe Line Search on a Class of Nonsmooth Convex Functions*'. Elsevier, Optimization and Control: 5-6 doi: arXiv:1711.08517v2

[38] Sherman, J, Morrison, W. J. (1950) '*Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix*'. The Annals of Mathematical Statistics, Vol. 21, Issue 1: 124-127 doi: 10.1214/aoms/1177729893

[39] Liu F., Yang M. (2005) '*Verification and Validation of Artificial Neural Network Models*'. Springer, Advances in Artificial Intelligence, Lecture Notes in Computer Science, Vol. 3809: 124-127 doi: https://doi.org/10.1007/11589990_137

[40] Herten, R. L. M. Chiribiri, A. Breeuwer, M. (2020) '*Physics-informed neural networks for myocardial perfusion MRI quantification*'. Electrical Engineering and Systems Science, Image and Video Processing doi: arXiv:2011.12844v2

[41] Hughes, T. W. Williamson, I. D. Minkov, M. Fan, S. (2019) '*Wave physics as an analog recurrent neural network*'. Science Advances, Vol. 5, Issue 12 doi: 10.1126/sciadv.aay6946

[42] Moorkamp, M. Lelievre, P. G. Linde, N. Khan, A. (2016) '*Integrated Imaging of the Earth : Theory and Applications*'. Wiley, Geophysical Monograph Series, Vol. 218, Edition 1: 31-40 doi: 10.1126/sciadv.aay6946

[43] Virieux, J. Asnaashari, A. Brossier, B. Metivier, L. Ribodetti, A. Zhou, W. (2014) '*An introduction to full waveform inversion*'. Society of Exploration Geophysicists. doi: https://doi.org/10.1190/1.9781560803027.entry6

[44] Li, ZC., Lin, YZ., Zhang, K. et al. T (2017) '*Time-domain wavefield reconstruction inversion.*'. Appl. Geophys, Vol. 14: 523-528 doi: https://doi-org.manchester.idm.oclc.org/10.1007/s11770-017-0629-6

[45] Symes, W. W. (2020) '*Wavefield reconstruction inversion: an example*'. AIOP Publishing Ltd, Inverse Problems, Vol. 36, No. 10

[46] Amestoy, P. et al (2016) '*Fast 3D frequency-domain full-waveform inversion with a parallel block low-rank multifrontal direct solver: Application to OBC data from the North Sea*'. Geophysics, Vol. 81, No. 6. doi: 10.1190/GEO2016-0052.1

[47] Mosely, B. Markham, A. Nissen-Meyer, T. (2020) '*Solving the wave equation with physics-informed deep learning*'. Geoscience Frontiers, Vol. 11, Issue 6. 1993-2001. doi: https://doi.org/10.1016/j.gsf.2020.07.007

[48] Song, C. Alkhalifah, T. (2021) '*Wavefield reconstruction inversion via physics-informed neural networks*'. (preprint) doi: arXiv:2104.06897

[49] Lipmann, B. A. Schwiner, J. (1950) '*Variational principles for scattering processes*'. American Physical Society, Vol. 79, Issue 4. doi: https://doi.org/10.1103/PhysRev.79.469

[50] Raissi, M. Perdikaris, P. Karniadakis, G. E. (2017) '*Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*'. Journal of Computational Physics, Vol. 378: 686-707: doi: https://doi.org/10.1016/j.jcp.2018.10.045

[51] Raissi, M. Perdikaris, P. Karniadakis, G. E. (2017) '*Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*'. Journal of Computational Physics, Vol. 378: 686-707: doi: https://doi.org/10.1016/j.jcp.2018.10.045

[52] Navarro-Jimenez, J. M. Tur, M. Albedla, J. Rodenas, J. J. (2018) '*Large deformation frictional contact analysis with immersed boundary method*'. Springer, Computational Mechanics, Vol. 62 doi: 10.1007/s00466-017-1533-x

[53] Wunderlich, W. Cramer, H. Steinl, G. (1998) '*An adaptive finite element approach in associated and non-associated plasticity considering localization phenomena*'. Studies in Applied Mechanics, volume 47: 293-308 doi: https://doi.org/10.1103/PhysRev.79.469

[54] Reed, R., Marksll, R. J. (1999) '*Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*'. MIT press

[55] Blalock, D. Ortiz, J. J. G. Frankle, J. Guttag, J. (2020) '*What is the state of pruning?*'. Proceedings of Machine Learning and Systems doi: arXiv:2003.03033

[56] Xenophontos, C. (2003) '*A note on the convergence rate of the finite element method for singularly perturbed problems using the Shishkin mesh*'. Elsevier, Applied Mathematics and Computation, Vol. 142: 545-559 doi: 10.1016/S0096-3003(02)00338-7

[57] Causon, D. M. Mingham, C. G. (2003) '

[58] Xenophontos, C. (2003) 'A note on the convergence rate of the finite element method for singularly perturbed problems using the Shishkin mesh'. Elsevier, Applied Mathematics and Computation, Vol. 142: 545-559 doi: 10.1016/S0096-3003(02)00338-7'. Elsevier, Applied Mathematics and Computation, Vol. 142: 545-559 doi: 10.1016/S0096-3003(02)00338-7

[59] Wang, W. Ma, Y. Zhao, K. Tian, Y. (2020) '*A Comprehensive Survey of Loss Functions in Machine Learning*'. Springer, Annals of Data Sciences, doi: https://doi.org/10.1007/s40745-020-00253-5

[60] Karniadakis, G. E. Kevrekidis, I. G. Lu, L. Perdikaris, P. Wang, S. Yang, L. (2021) '*Physics-informed machine learning*'. Nature Reviews Physics, volume 3: 422-440 doi: https://doi.org/10.1038/s42254-021-00314-5

[61] Baydin, A. G. Pearlmutter, B. A. Radul, A. A. Siskind, J. M. (2018) '*Automatic Differentiation in Machine Learning: a Survey*'. Journal of Machine Learning Research, vol. 18, Issue 1: 1-43

[62] Seidel, P. (2011) '*Numerical differentiation*'. MIT lecture notes, Fall 2011: https://math.mit.edu/classes/18.01/F2011/lecture14.pdf

[63] Higham, N. (2002) '*Accuracy and Stability of Numerical Algorithms*'. Siam, 2nd edition: 37-38 doi: https://doi.org/10.1137/1.9780898718027

[64] Spijker (1971) '*On the structure of error estimates for finite-difference methods*'. Numerical Mathematics, Vol. 18: 73-100 doi: https://doi.org/10.1007/BF01398460

[65] Axelsson, O. (1973) '*Notes on the Numerical Solution of the Biharmonic Equation*'. IMA Journal of Applied Mathematics, Vol. 11, Issue 2,: 213–226 doi: https://doi.org/10.1093/imamat/11.2.213

[66] Mustafa, I. Miah, M. Chowdhury, A. Ali, S. Hadi, R. Mehmet, A. A., Chu, Y. (2020) '*New exact solutions for the Kaup-Kupershmidt equation*'. AIMS Mathematics, Vol. 5, Issue 6,: 6726-6738 doi: 10.3934/math.2020432

[67] Sandstede, B. (2002) '*Stability of Travelling Waves*'. Elsevier, Handbook of Dynamical systems, Vol. 2: 983-1055 doi: https://doi.org/10.1016/S1874-575X(02)80039-X

[68] Kalos-Szirmay, L. (2021) '*Higher Order Automatic Differentiation with Dual Numbers*'. Periodica Polytechnica Electrical Engineering and Computer Science, Vol. 65: 1-10 doi: https://doi.org/10.3311/PPee.16341

[69] Penunuri, F. Carvente, O. Zambrano, M. A. Peon, R. Cruz-Villar, C. A. (2019) ‘*Dual Numbers for Algorithmic Differentiation*’. IIngeniería, vol. 23, no. 3: 71-81

[70] Ali, K. M. Riaz, F. (2011) ‘*Applications of Graph Theory in Computer Science*’. Third International Conference on Computational Intelligence, Communication Systems and Networks: 142-145. doi: 10.1109/CICSyN.2011.40

[71] Margossian, C. C. (2018) ‘*A Review of automatic differentiation and its efficient implementation*’. Wiley, WIREs Data Mining and Knowledge Discovery, Communication Systems and Networks: 142-145. doi: 10.1002/widm.1305

[72] Leephakpreeda (2002) ‘*Novel determination of differential-equation solutions: universal approximation method*’. Elsevier, Journal of Computational and Applied Mathematics, Vol. 146: 443-457

[73] Leephakpreeda (1998) ‘*Artifical Neural Networks for Solving Ordinary and Partial Differential Equations*’. Institute of Electrical and Electronics Engineers, Transactions on Neural Networkss, Vol. 9, No. 5: 987–1000: doi: 10.1109/72.712178

Appendix A

A.1 Proofs and Examples

A.1.1 Expression Swell

ANNs are, fundamentally, activation functions $\sigma : x \to [-1, 1]$ that are continually summed and embedded into each other, sometimes thousands of times. Expression swell is a phenomena in computing where by, as the calculation progresses, the size of the problem grows exponentially.

Take the following, simple function

$$f(x) = \tanh\left(\sin\left(\cos\left(x\right)\right)\right) \tag{A.1}$$

which has 3 operations. Clearly, if it was an ANN, it would be incredibly simple. Taking the derivative with respect to x yields

$$f'(x) = -\sin\left(x\right)\cos\left(\cos\left(x\right)\right)\operatorname{sech}^2\left(\sin\left(\cos\left(x\right)\right)\right) \tag{A.2}$$

which has 9 operations. We again differentiate to get

$$\begin{aligned} f''(x) = &-2\sin^2\left(x\right)\cos^2\left(\cos\left(x\right)\right)\operatorname{sech}^2\left(\sin\left(\cos\left(x\right)\right)\right)\tanh\left(\sin\left(\cos\left(x\right)\right)\right) \\ &- \sin^2\left(x\right)\sin\left(\cos\left(x\right)\right)\operatorname{sech}^2\left(\sin\left(\cos\left(x\right)\right)\right) \\ &- \cos\left(x\right)\cos\left(\cos\left(x\right)\right)\operatorname{sech}^2\left(\sin\left(\cos\left(x\right)\right)\right) \end{aligned} \tag{A.3}$$

which has 27 operations. We again differentiate to get

$$
\begin{aligned}
f'''(x) = &-\operatorname{sech}^2(\sin(\cos(x)))\big(4\sin^3(x)\cos(\cos(x))\sin(\cos(x))\tanh(\sin(\cos(x))) \\
&+4\cos(x)\sin(x)\cos^2(\cos(x))\tanh(\sin(\cos(x))) \\
&-2\sin^3(x)\cos^3(\cos(x))\operatorname{sech}^2(\sin(\cos(x))) \\
&+3\cos(x)\sin(x)\sin(\cos(x)) - \sin^3(x)\cos(\cos(x)) - \sin(x)\cos(\cos(x)) \\
&-2\sin(x)\cos(\cos(x))\operatorname{sech}^2(\sin(\cos(x)))\tanh(\sin(\cos(x))) \\
&\big(2\sin^2(x)\cos^2(\cos(x))\tanh(\sin(\cos(x))) + \sin^2(x)\sin(\cos(x)) \\
&+\cos(x)\cos(\cos(x))
\end{aligned}
\tag{A.4}
$$

which has... a lot more than 3 operations! 27 operations for a computer is not very many, but large ANNs can have thousands of fully connected nodes, and so $f(x)$ could have millions operations alone. This size increases dramatically during symbolic differentiation.

A.1.2 Solution to 1D Wave Equation

1D waves of all kinds with a multitude of initial and BCs have been studied for centuries, and they therefore make for a nice problem to test PINNs because solutions are well understood. For 1D waves with Dirichlet BCs, the solution can be found by D'Almbert's method, Fourier transform, or separation of variables. Here we will show that the solution to the problem described in (insert ref here) is the series solution (insert ref here) using separation of variables.

The 1D wave equation takes the following form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{A.5}$$

We search for a solution that takes the form $u(x,t) = X(x)T(t)$, which yields

$$\frac{1}{X}\frac{d^2 X}{dx^2} = \frac{1}{c^2}\frac{1}{T}\frac{d^2}{dt^2} \tag{A.6}$$

by substituting in the expected form into A.6. Since both sides are functions of a different variable, they must be constant, and so

$$
\begin{aligned}
\frac{1}{X}\frac{d^2 X}{dx^2} &= -k^2 \\
\frac{1}{c^2}\frac{1}{T}\frac{d^2}{dt^2} &= -k^2
\end{aligned}
\tag{A.7}
$$

These can then be solved as coupled ODEs. Solving for $X(x)$ and $T(t)$ gives

$$\begin{aligned} X(x) &= A\cos(kx) + B\sin(kx) \\ T(t) &= C\cos(kct) + D\cos(kct) \end{aligned} \tag{A.8}$$

Applying the BCs finds the constants in the solutions. If

$$u(x,t) = (A\cos(kx) + B\sin(kx))(C\cos(kct) + D\cos(kct)) \tag{A.9}$$

then using $u(0,t) = 0$ gives $A = 0$ and $k = \frac{n\pi}{2}$, where n is a positive integer. This gives a particular solution

$$u_n(x,t) = \big(\alpha_n \cos(k_n ct) + \beta_n \cos(k_n ct)\big) \sin\left(\frac{n\pi x}{2}\right) \tag{A.10}$$

The IC gives $u_t(x,0) = 0$, and so $\beta_n = 0$, giving

$$u_n(x,t) = \alpha_n \cos(k_n ct) \sin\left(\frac{n\pi x}{2}\right) \tag{A.11}$$

The orthogonality of the sine function means that, for a string of length 2, we have

$$\begin{aligned} \alpha_n &= \int_0^2 u(x,0)\sin\left(\frac{n\pi x}{2}\right)dx \\ &= \int_0^2 x(2-x)\sin\left(\frac{n\pi x}{2}\right)dx \\ &= -\frac{8\pi n\sin(\pi n) + 16\cos(\pi n) - 16}{\pi^3 n^3} \end{aligned} \tag{A.12}$$

with $ck_n = \frac{cn\pi}{2} = \frac{n\pi}{2}$ (for $c = 1$), which gives the series solution

$$u(x,t) = \sum_{n=1}^{\infty} -\frac{8\pi n\sin(\pi n) + 16\cos(\pi n) - 16}{\pi^3 n^3} cos\left(\frac{n\pi t}{2}\right)\sin\left(\frac{n\pi x}{2}\right) \tag{A.13}$$

A.2 Code for Errors and Plotting

A.2.1 Wave Error

Calculates and plots the errors between solutions for the wave equation

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.colors import Normalize
```

```
# load data
true = np.load('../data/waveformTruecis1.npy')
pinn_edge = np.load('../data/waveformPINNcis1_64_32_16_8.npy')
pinn = np.load('../data/pinn_16_8_4_2.npy')

# set up grid for plotting
x = np.linspace(0, 2, true.shape[0])
t = np.linspace(0, 4, true.shape[1])

# calculate L" norm error
trueL2 = np.linalg.norm(true, ord=2)
pinnL2 = np.linalg.norm(pinn, ord=2)
L2error = np.linalg.norm((true.flatten() - pinn.flatten()), ord=2) / np.linalg.norm(true.flatten(), ord=
abserror = true - pinn
#show L2 norm error

# allocate memory to hold errors
error = np.zeros(true.shape)
mseerror = np.zeros(true.shape)
for i in range(2, error.shape[0]-2):
    for j in range(0, error.shape[1]):
        if abs(true[i, j]) > 1e-12: # avoid division by 0
            error[i,j] = (pinn[i,j] / true[i,j]) - 1
            mseerror[i,j] = (true[i,j] - pinn[i,j]) ** 2

MSE = mseerror.sum() / (mseerror.shape[0] * mseerror.shape[1])

truegrad = np.gradient((true))
pinngrad = np.gradient(pinn)
diffgrad = np.zeros((2,200,400))
diffgrad[0] = truegrad[0] - pinngrad[0]
diffgrad[1] = truegrad[1] - pinngrad[1]

energyerror = np.sum(np.dot(diffgrad[0], diffgrad[0].T) * np.dot(diffgrad[1], diffgrad[1].T))

print('energy error: ', energyerror)
print("MSE: ", MSE)
print("L2 error: ", L2error)
# display data
fig = plt.figure()
gs = GridSpec(3, 2)
plt.subplot(gs[0, 0])
vmin, vmax = -1, +1
plt.pcolormesh(t, x, pinn_edge, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('PINN solution for data at t=0')
```

```
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[0, 1])
vmin, vmax = -1, +1
plt.pcolormesh(t, x, pinn, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('PINN solution for data at t=2')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[1, 0])
vmin, vmax = -0.05, 0.05
plt.pcolormesh(t, x, true-pinn_edge, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('Error for t=0 data')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[1, 1])
vmin, vmax = -0.05, 0.05
plt.pcolormesh(t, x, abserror, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('Error for t=2 data')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[2, :])
vmin, vmax = -0.05, 0.05
plt.pcolormesh(t, x, abserror + (true-pinn_edge), cmap='rainbow', norm=Normalize(vmin=vmin,
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('Error for t=2 data')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()
plt.savefig('../Graphs/error_test.png')
plt.show()
```

A.2.2 Wave Gradient Error

Calculates and plots the errors between gradients of solutions for the wave equation

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.colors import Normalize

# load data
true = np.load('../data/waveformTruecis1.npy')
pinn = np.load('../data/waveformPINNcis1_64_32_16_8.npy')

# set up grid for plotting
x = np.linspace(0, 2, true.shape[0])
t = np.linspace(0, 4, true.shape[1])

# calc error
truegrad = np.gradient(true, 1/100)
pinngrad = np.gradient(pinn, 1/100)
truegrad_x = truegrad[0]
truegrad_t = truegrad[1]
pinn_x = pinngrad[0]
pinn_t = pinngrad[1]

grad_error_x = (truegrad_x - pinn_x)**2
grad_error_t = (truegrad_t - pinn_t)**2


error = np.sum(grad_error_x+truegrad_t)
print(error)

absrange_x = np.max(np.abs(truegrad[0]- pinngrad[0]))
absrange_t = np.max(np.abs(truegrad[1]- pinngrad[1]))
relrange_x = np.max(np.abs((truegrad[0] / pinngrad[0]) - 1))
relrange_t = np.max(np.abs((truegrad[0] / pinngrad[0]) - 1))


fig = plt.figure()
gs = GridSpec(3, 2)
plt.subplot(gs[0, 0])
vmin, vmax = -2.5, 2.5
plt.pcolormesh(t, x, pinn_x, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('pinn_x')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[0, 1])
vmin, vmax = -2.5, 2.5
```

```
plt.pcolormesh(t, x, pinn_t, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('pinn_t')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[1, 0])
vmin, vmax = -2.5, 2.5
plt.pcolormesh(t, x, truegrad[0], cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('u_x(t,x)')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[1, 1])
vmin, vmax = -2.5, 2.5
plt.pcolormesh(t, x, truegrad[1], cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('u_t(t,x)')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[2, 0])
vmin, vmax = -0.2, 0.2
plt.pcolormesh(t, x, truegrad_x - pinn_x, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vma
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('u_x error')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[2, 1])
vmin, vmax = -0.2, 0.2
plt.pcolormesh(t, x, truegrad_t - pinn_t, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vma
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('u_t error')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()
plt.show()
```

A.2.3 Iteration Error

Plots data collected every 50 iterations to check patterns of convergence for a PINN solution

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.colors import Normalize

true = np.load('../data/waveformTruecis1.npy')
pinn50 = np.load('../data/data_it_mid50.npy')
pinn100 = np.load('../data/data_it_mid100.npy')
pinn150 = np.load('../data/data_it_mid150.npy')
pinn200 = np.load('../data/data_it_mid200.npy')
pinn250 = np.load('../data/data_it250.npy')
pinn300 = np.load('../data/data_it300.npy')
pinn400 = np.load('../data/data_it_mid400.npy')
pinn500 = np.load('../data/data_it_mid500.npy')
pinn600 = np.load('../data/data_it_mid600.npy')
pinn750 = np.load('../data/data_it_mid750.npy')
pinn1000 = np.load('../data/data_it1000.npy')
pinn2000 = np.load('../data/data_it2000.npy')
pinn3000 = np.load('../data/data_it3000.npy')

abserror50 = true-pinn50
abserror100 = true-pinn100
abserror150 = true-pinn150
abserror200 = true-pinn200
abserror250 = true-pinn250
abserror300 = true-pinn300
abserror400 = true-pinn400
abserror500 = true-pinn500
abserror600 = true-pinn600
abserror750 = true-pinn750
abserror1000 = true-pinn1000
abserror2000 = true-pinn2000
abserror3000 = true-pinn3000

errs = [abserror50,
        abserror100,
        abserror150,
        abserror200,
abserror400,
        abserror500,
        abserror600,
abserror750
        ]

titles = [
    '50 iterations',
'100 iterations',
'150 iterations',
'200 iterations',
'400 iterations',
'500 iterations',
'600 iterations',
'750 iterations'
]

# set up grid for plotting
x = np.linspace(0, 2, true.shape[0])
t = np.linspace(0, 4, true.shape[1])

fig = plt.figure()
gs = GridSpec(4, 2)
k=0
for i in range(0, 4):
    for j in range(0, 2):
        plt.subplot(gs[i, j])
        vmin, vmax = -0.05, 0.05
        plt.pcolormesh(t, x, errs[k], cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
        plt.xlabel('t')
        plt.ylabel('x')
        cbar = plt.colorbar(pad=0.05, aspect=10)
        plt.title(titles[k])
        cbar.mappable.set_clim(vmin, vmax)
        plt.tight_layout()
        k = k+1

plt.show()
```

A.2.4 Burgers' Error

Plots the PINN solution and exact solution from the study referenced in chapter 5. Also calculates gradients and calculates the error over the domain.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.colors import Normalize
import scipy.io as scipy

# load data
```

```
output = np.load('grid_u.npy').T
real = scipy.loadmat('burgers_shock.mat')['usol']
print(output.shape)
realgrads_t = np.gradient(real, 1/100)[1]
realgrads_x = np.gradient(real, 1/128)[0]
realgrads_tt = np.gradient(realgrads_t, 1/100)[1]
realgrads_xx = np.gradient(realgrads_x, 1/128)[0]
output_t = np.gradient(output, 1/100)[1]
output_x = np.gradient(output, 1/128)[0]

x = np.linspace(-1, 1, output.shape[0])
t = np.linspace(0, 1, output.shape[1])

fig = plt.figure()
gs = GridSpec(2, 1)

plt.subplot(gs[0, 0])
vmin, vmax = -1, 1
plt.pcolormesh(t, x, real, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title(r'$\phi(x, t)$')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[1, 0])
vmin, vmax = -1, 1
plt.pcolormesh(t, x, output, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('PINN solution')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.show()

fig = plt.figure()
gs = GridSpec(3, 1)

plt.subplot(gs[0, 0])
vmin, vmax = -0.01, 0.01
plt.pcolormesh(t, x, real-output, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title('absolute error')
```

```
cbar.mappable.set_clim(vmin, vmax)

plt.subplot(gs[1, 0])
vmin, vmax = -50, 50
plt.pcolormesh(t, x, realgrads_x, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title(r'$\phi_x$')
cbar.mappable.set_clim(vmin, vmax)
plt.tight_layout()

plt.subplot(gs[2, 0])
vmin, vmax = -10, 10
plt.pcolormesh(t, x, realgrads_t, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
plt.title(r'$\phi_t$')
cbar.mappable.set_clim(vmin, vmax)

plt.show()
```

A.3 1D Wave Equation Series Solution

Calculates the series solution for the boundary value/initial value problem described in 3.2

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.colors import Normalize

# solution for a 1d vibrating membrane u_tt = c^2u_xx
# 2 units long rectangle with c=6 and IC f(x, y)=x(2-x) with u=0 at the edges.
# u_t(t=0)=0

def u_sol(x, t, n):

    u = 0
    for i in range(1, n+1):
        num = (-8*np.pi*i * np.sin(np.pi*i) +
               16 * np.cos(np.pi * i) - 16)
        denom = np.pi**3 * i**3
        coef1 = np.cos(((6)* i * np.pi * t)/2)
        coef2 = np.sin((i * np.pi * x) / 2)
        u = u - ((num / denom) * coef1 * coef2)
    return u


x = np.linspace(0, 2, 200)
t = np.linspace(0, 2, 400)
wave = np.zeros((x.__len__(), t.__len__()))
for i in range(0, x.__len__()):
    for j in range(0, t.__len__()):
        ans = u_sol(x[i], t[j], 100)
        wave[i, j] = ans

    print(i)

np.save('../data/waveformTrue2.npy', np.asarray(wave))

fig = plt.figure(figsize=(7,4))
gs = GridSpec(2, 4)
plt.subplot(gs[0, :])
vmin, vmax = -1.2, +1.2
plt.pcolormesh(t, x, wave, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
cbar.set_label('u(t,x)')
cbar.mappable.set_clim(vmin, vmax)
t_cross_sections = [0, 1, 2, 3]
time = 0
for i, t_cs in enumerate(t_cross_sections):
    plt.subplot(gs[1, i])
    plt.plot(x, wave[:, i * 25])
    plt.title('t={}'.format(time))
    plt.xlabel('x')
    plt.ylabel('u(t,x)')
    plt.ylim([-1.2, 1.2])
    time = time + 0.125
plt.tight_layout()
plt.savefig('1Dwaveproblemtrue.png', transparent=True)
plt.show()
```

A.3.1 2D Wave Equation Series Solution

Calculates the series solution for the boundary value/initial value problem described in 3.2.

```
import numpy as np
import math
```

```
# solution for a rectangular vibrating membrane u_tt = c^2(u_xx + u_yy)
# 2x3 rectangle with c=6 and IC f(x, y)=xy(2-x)(3-y) with u=0 at the edges.
# u_t(t=0)=0

# the estimate the derivative we require a very small value
eps = np.sqrt(np.finfo(np.float32).eps)

# define series solution
def u_sol(x, y, t, n, m):
    u = 0
    coef = 576 / (math.pi ** 6)
    for i in range(1, n + 1):
        sum = 0
        for j in range(1, m + 1):
            termmn = ((1 + (-1) ** (j + 1)) * (1 + (-1) ** (i + 1))) / (j ** 3 * i ** 3)
            termx = math.sin((j * math.pi * x) / 2)
            termy = math.sin((i * math.pi * y) / 3)
            termt = math.cos(math.pi * math.sqrt(9 * j ** 2 + 4 * i ** 2) * t)

            sum = sum + termmn * termx * termy * termt
        u = u + sum

    return coef * u


def g_phys(x, y, t):
    u_2 = 2 * u_sol(x, y, t)
    u_tt = (u_sol(x, y, t+eps) - u_2 + u_sol(x, y, t-eps)) / eps ** 2
    u_xx = (u_sol(x+eps, y, t) - u_2 + u_sol(x-eps, y, t)) / eps ** 2
    u_yy = (u_sol(x, y + eps, t) - u_2 + u_sol(x, y - eps, t)) / eps ** 2
    return u_tt - 6**2 * (u_xx + u_yy)


def u_vol(x, y, t, n):
    u = 0
    coef = math.sin(2 * math.pi * x) / math.pi ** 2
    for i in range(1, n + 1):
        termn = (1 + (-1) ** (i + 1)) / (i * math.sqrt(36 + i ** 2))
        termy = math.sin((n * math.pi) / 3 * y)
        termt = math.sin(2 * math.pi * math.sqrt(36 + i ** 2) * t)

        u = u + termn * termy * termt

    return coef * u


# define the space and preallocate space for series solution
x_int = np.linspace(0, 2, 100)
```

```
y_int = np.linspace(0, 3, 150)
time = np.linspace(0, 0.5, 10)
wave_0 = np.ones((x_int.__len__(), y_int.__len__(), time.__len__()))
wave_vol = np.ones((x_int.__len__(), y_int.__len__(), time.__len__()))
n_it = 20
m_it = 20

for i in range(0, x_int.__len__()):
    for j in range(0, y_int.__len__()):
        for k in range(0, time.__len__()):
            wave_0[i][j][k] = u_sol(x_int[i], y_int[j], time[k], n_it, m_it)
            #wave_vol[i][j][k] = u_vol(x_int[i], y_int[j], time[k], n_it)

    print(i)

# some plotting stuff
# plt.imshow(wave.T, extent=(0, 2, 0, 3), origin='lower', interpolation='nearest')
# plt.show()

# plt.contourf(x_int, y_int, wave[:, :, 0].T)
# plt.show()

#X, Y = np.meshgrid(x_int, y_int)
# # fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
# # surf = ax.plot_surface(X, Y, wave_0.T, linewidth=0, antialiased=False, cmap=cm.plasma)
# # plt.show()

# save the output
np.save("../data/2d_homogeneous_membrane.npy", wave_0)
#np.save("../data/2d_velocity_membrane.npy", wave_0 + wave_vol)
```

A.3.2 Plotting Routine for 2D Wave

Plots the 2D wave solution with data generated from A.2.2. The commented out section creates a 3D animated graph of the wave behaviour for the given time, but is slow to run.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D

wave_0 = np.load("../data/2d_homogeneous_membrane.npy")
x_int = np.arange(0, 2, 2/wave_0.shape[0])
y_int = np.arange(0, 3, 3/wave_0.shape[1])
ims = []
```

```
X, Y = np.meshgrid(x_int, y_int)
"""
fps = 10
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
for i in range(0, wave_0.shape[2]):
    surf = ax.plot_surface(X, Y, wave_0[:, :, i].T, linewidth=0, antialiased=False,
                           shade=True, alpha=0.5, color='black')
    ims.append([surf])
    print(i)

"""
levels = np.arange(-2.5, 2.75, 0.25)
fig, ax = plt.subplots(2, 3)

plt.subplot(2,3,1)
plt.title('Membrane shape at t=0')
cs = plt.contourf(X, Y, wave_0[:, :, 0].T, levels=levels, cmap=cm.jet)
plt.subplot(2,3,2)
plt.title('Membrane shape at t=0.05')
cs = plt.contourf(X, Y, wave_0[:, :, 1].T, levels=levels, cmap=cm.jet)
plt.subplot(2,3,3)
plt.title('Membrane shape at t=0.1')
cs = plt.contourf(X, Y, wave_0[:, :, 2].T, levels=levels, cmap=cm.jet)
plt.subplot(2,3,4)
plt.title('Membrane shape at t=0.15')
cs = plt.contourf(X, Y, wave_0[:, :, 3].T, levels=levels, cmap=cm.jet)
plt.subplot(2,3,5)
plt.title('Membrane shape at t=0.2')
cs = plt.contourf(X, Y, wave_0[:, :, 4].T, levels=levels, cmap=cm.jet)
plt.subplot(2,3,6)
plt.title('Membrane shape at t=0.25')
cs = plt.contourf(X, Y, wave_0[:, :, 5].T, levels=levels, cmap=cm.jet)

fig.subplots_adjust(right=0.8)
cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
CB = fig.colorbar(cs, cax=cbar_ax)
CB.set_label('Membrane height', rotation=270)

plt.show()
```

Plotter for 2D wave

```
import numpy as np
import math

# solution for a rectangular vibrating membrane u_tt = c^2(u_xx + u_yy)
# 2x3 rectangle with c=6 and IC f(x, y)=xy(2-x)(3-y) with u=0 at the edges.
# u_t(t=0)=0
# the estimate the derivative we require a very small value
eps = np.sqrt(np.finfo(np.float32).eps)

# define series solution
def u_sol(x, y, t, n, m):
    u = 0
    coef = 576 / (math.pi ** 6)
    for i in range(1, n + 1):
        sum = 0
        for j in range(1, m + 1):
            termmn = ((1 + (-1) ** (j + 1)) * (1 + (-1) ** (i + 1))) / (j ** 3 * i ** 3)
            termx = math.sin((j * math.pi * x) / 2)
            termy = math.sin((i * math.pi * y) / 3)
            termt = math.cos(math.pi * math.sqrt(9 * j ** 2 + 4 * i ** 2) * t)

            sum = sum + termmn * termx * termy * termt
        u = u + sum

    return coef * u


def g_phys(x, y, t):
    u_2 = 2 * u_sol(x, y, t)
    u_tt = (u_sol(x, y, t+eps) - u_2 + u_sol(x, y, t-eps)) / eps ** 2
    u_xx = (u_sol(x+eps, y, t) - u_2 + u_sol(x-eps, y, t)) / eps ** 2
    u_yy = (u_sol(x, y + eps, t) - u_2 + u_sol(x, y - eps, t)) / eps ** 2
    return u_tt - 6**2 * (u_xx + u_yy)


def u_vol(x, y, t, n):
    u = 0
    coef = math.sin(2 * math.pi * x) / math.pi ** 2
    for i in range(1, n + 1):
        termn = (1 + (-1) ** (i + 1)) / (i * math.sqrt(36 + i ** 2))
        termy = math.sin((n * math.pi) / 3 * y)
        termt = math.sin(2 * math.pi * math.sqrt(36 + i ** 2) * t)

        u = u + termn * termy * termt

    return coef * u


# define the space and preallocate space for series solution
x_int = np.linspace(0, 2, 100)
y_int = np.linspace(0, 3, 150)
time = np.linspace(0, 0.5, 10)
wave_0 = np.ones((x_int.__len__(), y_int.__len__(), time.__len__()))
```

```
wave_vol = np.ones((x_int.__len__(), y_int.__len__(), time.__len__()))
n_it = 20
m_it = 20

for i in range(0, x_int.__len__()):
    for j in range(0, y_int.__len__()):
        for k in range(0, time.__len__()):
            wave_0[i][j][k] = u_sol(x_int[i], y_int[j], time[k], n_it, m_it)
            #wave_vol[i][j][k] = u_vol(x_int[i], y_int[j], time[k], n_it)

    print(i)

# some plotting stuff
# plt.imshow(wave.T, extent=(0, 2, 0, 3), origin='lower', interpolation='nearest')
# plt.show()

# plt.contourf(x_int, y_int, wave[:, :, 0].T)
# plt.show()

#X, Y = np.meshgrid(x_int, y_int)
# # fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
# # surf = ax.plot_surface(X, Y, wave_0.T, linewidth=0, antialiased=False, cmap=cm.plasma)
# # plt.show()

# save the output
np.save("../data/2d_homogeneous_membrane.npy", wave_0)
#np.save("../data/2d_velocity_membrane.npy", wave_0 + wave_vol)
```

A.4 Code for PINNs

A.4.1 ANN to Approximate Solution for ODE

Code initialises random weights and biases for an ANN, and uses the ODE and an IC to converge towards a solution. The plotting routine also checks gradients of the ANN solution against the analytical solution. The code is based on a Julia implementation of a similar problem displayed by MIT. The code generates figure 3.2.

https://mitmath.github.io/18337/lecture3/sciml.html

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.pyplot import figure
import math
from scipy.misc import derivative
```

```
# define the parameters and functions of the ODE
def u(t):
    return math.cos(2 * math.pi * t)


u0 = 1  # create an initial condition

# the estimate the derivative we require a very small value
eps = np.sqrt(np.finfo(np.float32).eps)

# define the learning parameters
rate = 0.01  # when we find the direction of descent, how far will we go in one step
steps = 1000  # how many training steps to complete
batch_size = 500  # how many training points to use each training cycle
display_step = steps / 10  # how often to display training step
points = 100  # number of training points from the ODE used

# define the layout of the network
nn_input = 1  # number of neurons for the input layer
nn_hidden_1 = 32  # number of neurons in hidden layer 1
nn_hidden_2 = 32  # number of neurons in hidden layer 2
nn_output = 1  # number of neurons in the output layer

# initialise the weights and biases for the network as random matrices of the correct size
weights = {
    'h1': tf.Variable(tf.random.normal([nn_input, nn_hidden_1])),
    'h2': tf.Variable(tf.random.normal([nn_hidden_1, nn_hidden_2])),
    'out': tf.Variable(tf.random.normal([nn_hidden_1, nn_output]))
}
biases = {
    'b1': tf.Variable(tf.random.normal([nn_hidden_1])),
    'b2': tf.Variable(tf.random.normal([nn_hidden_2])),
    'out': tf.Variable(tf.random.normal([nn_output]))
}

# we wish to use stochastic gradient descent with the defined learning rate
optimiser = tf.optimizers.SGD(rate)


# create the network
def nn(t):
    t = np.array([[[t]]], dtype='float32')
    # Hidden fully connected layer with 32 neurons
    layer_1 = tf.add(tf.matmul(t, weights['h1']), biases['b1'])
    layer_1 = tf.nn.tanh(layer_1)
    # Hidden fully connected layer with 32 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.tanh(layer_2)
    # Output fully connected layer
    output = tf.matmul(layer_1, weights['out']) + biases['out']
    return output


# define the universal approximator using the network and the initial condition
def g(t):
    return u0 + t * nn(t)


# create the loss function, which compares the derivative of g against u
def loss_function():
    summation = []
    for x in np.linspace(0, 1, points):
        dNN = (g(x + eps) - g(x)) / eps
        summation.append(abs(dNN - u(x)) ** 2)

    ans = tf.reduce_mean(tf.abs(summation))
    return ans


def train_step():
    with tf.GradientTape() as tape:
        loss = loss_function()
    trainable_variables = list(weights.values()) + list(biases.values())
    gradients = tape.gradient(loss, trainable_variables)
    optimiser.apply_gradients(zip(gradients, trainable_variables))


# train the network
for i in range(steps):
    train_step()
    if i % display_step == 0:
        print("Average loss: %f " % (loss_function()))

# display the figure
figure(figsize=(10, 10))


# True Solution
def true_solution(x):
    return 1 + np.sin(2 * np.pi * x) / (2 * np.pi)


X = np.linspace(0, 1, points * 10)
result = []
result_prime = []
S = true_solution(X)
```

```
for i in X:
    with tf.GradientTape() as tape:
        prediction = g(i).numpy()[0][0][0]

    result.append(prediction)

dg = np.gradient(np.array(result), 1/(points * 10))
df = np.gradient(S, 1/(points * 10))
dgg = np.gradient(dg, 1/(points * 10))
dff = np.gradient(df, 1/(points * 10))
plt.plot(X, S, label="Original Function u(t)")
plt.plot(X, result, label="Neural Net Approximation g(t)")
plt.plot(X, dg, label="first derivative approx of g(t)")
plt.plot(X, df, label="first derivative original function")
plt.plot(X, dgg, label="second derivative approx of g(t)")
plt.plot(X, dff, label="second derivative original function")
plt.legend(loc=1, prop={'size': 15})
plt.xlabel("input x")
plt.ylabel("output")
plt.show()
```

A.4.2 Code for PINN wave

Code was adapated from Raissi M. et al to solve the 1D wave equation. Requires latest version of Tensorflow 2

Main script

Here a user may define an initial condition, a boundary condition, domain size and shape, and data distribution.

```
import lib.tf_silent
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
from matplotlib.gridspec import GridSpec
from lib.pinn import PINN
from lib.network import Network
from lib.optimizer import L_BFGS_B
from numpy import save


def u0(tx):
    """
    Initial wave form.

    Args:
        tx: variables (t, x) as tf.Tensor.
        c: wave velocity.
        k: wave number.
        sd: standard deviation.

    Returns:
        u(t, x) as tf.Tensor.
    """

    x = tx[..., 1, None]
    return x * (2-x)

def du0_dt(tx):
    """
    First derivative of t for the initial wave form.

    Args:
        tx: variables (t, x) as tf.Tensor.

    Returns:
        du(t, x)/dt as tf.Tensor.
    """

    with tf.GradientTape() as g:
        g.watch(tx)
        u = u0(tx)
    du_dt = g.batch_jacobian(u, tx)[..., 0]
    return du_dt

if __name__ == '__main__':
    """
    Test the physics informed neural network (PINN) model for the wave equation.
    """

    # number of training samples
    num_train_samples = 300
    # number of test samples
    num_test_samples = 150

    # build a core network model
    network = Network.build()
    network.summary()
    # build a PINN model
    pinn = PINN(network).build()

    # create training input
    tx_eqn = np.random.rand(num_train_samples, 2)
    tx_eqn[..., 0] = tx_eqn[..., 0] * 2          # t =  0 ~ +0.5
```

```
tx_eqn[..., 1] = tx_eqn[..., 1]*2           # x = 0 ~ 2
tx_ini = np.random.rand(num_train_samples, 2)
tx_ini[..., 0] = 0                               # t = 0
tx_ini[..., 1] = tx_ini[..., 1]*2          # x = 0 ~ 2
tx_bnd = np.random.rand(num_train_samples, 2)
tx_bnd[..., 0] = tx_bnd[..., 0] * 2       # t =  0 ~ +0.5
tx_bnd[..., 1] = np.round(tx_bnd[..., 1])*2 # x = 0 ~ 2
# create training output
u_zero = np.zeros((num_train_samples, 1))
u_ini = u0(tf.constant(tx_ini)).numpy()
du_dt_ini = du0_dt(tf.constant(tx_ini)).numpy()

# train the model using L-BFGS-B algorithm
x_train = [tx_eqn, tx_ini, tx_bnd]
y_train = [u_zero, u_ini, du_dt_ini, u_zero]
lbfgs = L_BFGS_B(model=pinn, x_train=x_train, y_train=y_train)
lbfgs.fit()

# predict u(t,x) distribution
t_flat = np.linspace(0, 2, 400)
x_flat = np.linspace(0, 2, 200)
t, x = np.meshgrid(t_flat, x_flat)
tx = np.stack([t.flatten(), x.flatten()], axis=-1)
u = network.predict(tx, batch_size=num_test_samples)
u = u.reshape(t.shape)
save('waveformPINN2.npy', np.asarray(u))
# plot u(t,x) distribution as a color-map
fig = plt.figure(figsize=(7,4))
gs = GridSpec(2, 4)
plt.subplot(gs[0, :])
vmin, vmax = -1.2, +1.2
plt.pcolormesh(t, x, u, cmap='rainbow', norm=Normalize(vmin=vmin, vmax=vmax))
# save('waveformPINN.npy', np.asarray(u))
plt.xlabel('t')
plt.ylabel('x')
cbar = plt.colorbar(pad=0.05, aspect=10)
cbar.set_label('u(t,x)')
cbar.mappable.set_clim(vmin, vmax)
# plot u(t=const, x) cross-sections
t_cross_sections = [0, 1, 2, 3]
time = 0
for i, t_cs in enumerate(t_cross_sections):
    plt.subplot(gs[1, i])
    tx = np.stack([np.full(t_flat.shape, time), x_flat], axis=-1)
    u = network.predict(tx, batch_size=num_test_samples)
    plt.plot(x_flat, u)
    plt.title('t={}'.format(time))
    plt.xlabel('x')
```

```
        plt.ylabel('u(t,x)')
        plt.ylim([-1.2, 1.2])
        time = time + 0.125
    plt.tight_layout()
    plt.savefig('1Dwaveproblem.png', transparent=True)
    plt.show()
```

Network Class

Define the structure of the neural network. Can alter activation function, input nodes, output nodes, and hidden layer structure

```
import tensorflow as tf

class Network:
    """
    Build a physics informed neural network (PINN) model for the wave equation.
    """

    @classmethod
    def build(cls, num_inputs=2, layers=[20,20,20,20,20,20,20,20], activation='tanh', num_ou
        """
        Build a PINN model for the wave equation with input shape (t, x) and output shape u(

        Args:
            num_inputs: number of input variables. Default is 2 for (t, x).
            layers: number of hidden layers.
            activation: activation function in hidden layers.
            num_outpus: number of output variables. Default is 1 for u(t, x).

        Returns:
            keras network model.
        """

        # input layer
        inputs = tf.keras.layers.Input(shape=(num_inputs,))
        # hidden layers
        x = inputs
        for layer in layers:
            x = tf.keras.layers.Dense(layer, activation=activation,
                kernel_initializer='he_normal')(x)
        # output layer
        outputs = tf.keras.layers.Dense(num_outputs,
            kernel_initializer='he_normal')(x)

        return tf.keras.models.Model(inputs=inputs, outputs=outputs)
```

Optimisation class

Defines the optimisation strategy used to minimise the objective function (in this case, L-BFGS). Here a user can alter some the variables of the optimiser, including max iterations and tolerance

```
import scipy.optimize
import numpy as np
import tensorflow as tf

class L_BFGS_B:
    """
    Optimize the keras network model using L-BFGS-B algorithm.

    Attributes:
        model: optimization target model.
        samples: training samples.
        factr: convergence condition. typical values for factr are: 1e12 for low accuracy;
               1e7 for moderate accuracy; 10 for extremely high accuracy.
        m: maximum number of variable metric corrections used to define the limited memory matrix.
        maxls: maximum number of line search steps (per iteration).
        maxiter: maximum number of iterations.
        metris: logging metrics.
        progbar: progress bar.
    """

    def __init__(self, model, x_train, y_train, factr=1e5, m=50, maxls=50, maxiter=5000):
        """
        Args:
            model: optimization target model.
            samples: training samples.
            factr: convergence condition. typical values for factr are: 1e12 for low accuracy;
                   1e7 for moderate accuracy; 10.0 for extremely high accuracy.
            m: maximum number of variable metric corrections used to define the limited memory matrix.
            maxls: maximum number of line search steps (per iteration).
            maxiter: maximum number of iterations.
        """

        # set attributes
        self.model = model
        self.x_train = [ tf.constant(x, dtype=tf.float32) for x in x_train ]
        self.y_train = [ tf.constant(y, dtype=tf.float32) for y in y_train ]
        self.factr = factr
        self.m = m
        self.maxls = maxls
        self.maxiter = maxiter
        self.metrics = ['loss']
        # initialize the progress bar
        self.progbar = tf.keras.callbacks.ProgbarLogger(
            count_mode='steps', stateful_metrics=self.metrics)
        self.progbar.set_params( {
            'verbose':1, 'epochs':1, 'steps':self.maxiter, 'metrics':self.metrics})

    def set_weights(self, flat_weights):
        """
        Set weights to the model.

        Args:
            flat_weights: flatten weights.
        """

        # get model weights
        shapes = [ w.shape for w in self.model.get_weights() ]
        # compute splitting indices
        split_ids = np.cumsum([ np.prod(shape) for shape in [0] + shapes ])
        # reshape weights
        weights = [ flat_weights[from_id:to_id].reshape(shape)
            for from_id, to_id, shape in zip(split_ids[:-1], split_ids[1:], shapes) ]
        # set weights to the model
        self.model.set_weights(weights)

    @tf.function
    def tf_evaluate(self, x, y):
        """
        Evaluate loss and gradients for weights as tf.Tensor.

        Args:
            x: input data.

        Returns:
            loss and gradients for weights as tf.Tensor.
        """

        with tf.GradientTape() as g:
            loss = tf.reduce_mean(tf.keras.losses.mse(self.model(x), y))
        grads = g.gradient(loss, self.model.trainable_variables)
        return loss, grads

    def evaluate(self, weights):
        """
        Evaluate loss and gradients for weights as ndarray.

        Args:
            weights: flatten weights.

        Returns:
            loss and gradients for weights as ndarray.
        """

        # update weights
        self.set_weights(weights)
        # compute loss and gradients for weights
        loss, grads = self.tf_evaluate(self.x_train, self.y_train)
        # convert tf.Tensor to flatten ndarray
        loss = loss.numpy().astype('float64')
        grads = np.concatenate([ g.numpy().flatten() for g in grads ]).astype('float64')

        return loss, grads

    def callback(self, weights):
        """
        Callback that prints the progress to stdout.

        Args:
            weights: flatten weights.
        """
        self.progbar.on_batch_begin(0)
        loss, _ = self.evaluate(weights)
        self.progbar.on_batch_end(0, logs=dict(zip(self.metrics, [loss])))

    def fit(self):
        """
        Train the model using L-BFGS-B algorithm.
        """

        # get initial weights as a flat vector
        initial_weights = np.concatenate(
            [ w.flatten() for w in self.model.get_weights() ])
        # optimize the weight vector
        print('Optimizer: L-BFGS-B (maxiter={})'.format(self.maxiter))
        self.progbar.on_train_begin()
        self.progbar.on_epoch_begin(1)
        scipy.optimize.fmin_l_bfgs_b(func=self.evaluate, x0=initial_weights,
            factr=self.factr, m=self.m, maxls=self.maxls, maxiter=self.maxiter,
            callback=self.callback)
        self.progbar.on_epoch_end(1)
        self.progbar.on_train_end()
```

Gradient Calculator

Custom piece of code that is fed the layers of the network in order to compute automatic differentials

```python
import tensorflow as tf

class GradientLayer(tf.keras.layers.Layer):
    """
    Custom layer to compute 1st and 2nd derivatives for the wave equation.

    Attributes:
        model: keras network model.
    """

    def __init__(self, model, **kwargs):
        """
        Args:
            model: keras network model.
        """

        self.model = model
        super().__init__(**kwargs)

    def call(self, tx):
        """
        Computing 1st and 2nd derivatives for the wave equation.

        Args:
            tx: input variables (t, x).

        Returns:
            u: network output.
            du_dt: 1st derivative of t.
            du_dx: 1st derivative of x.
            d2u_dt2: 2nd derivative of t.
            d2u_dx2: 2nd derivative of x.
        """

        with tf.GradientTape() as g:
            g.watch(tx)
            with tf.GradientTape() as gg:
                gg.watch(tx)
                u = self.model(tx)
            du_dtx = gg.batch_jacobian(u, tx)
            du_dt = du_dtx[..., 0]
            du_dx = du_dtx[..., 1]
        d2u_dtx2 = g.batch_jacobian(du_dtx, tx)
        d2u_dt2 = d2u_dtx2[..., 0, 0]
        d2u_dx2 = d2u_dtx2[..., 1, 1]

        return u, du_dt, du_dx, d2u_dt2, d2u_dx2
```

Physics-Informed Term

The gradients from the layer class are passed into the PINN class in order to construct to physics loss term. Altering the *u_eqn* variable allows a user to alter the physics of the system.

```
import tensorflow as tf
from .layer import GradientLayer

class PINN:
    """
    Build a physics informed neural network (PINN) model for the wave equation.

    Attributes:
        network: keras network model with input (t, x) and output u(t, x).
        c: wave velocity.
        grads: gradient layer.
    """

    def __init__(self, network, c=6):
        """
        Args:
            network: keras network model with input (t, x) and output u(t, x).
            c: wave velocity. Default is 1.
        """

        self.network = network
        self.c = c
        self.grads = GradientLayer(self.network)

    def build(self):
        """
        Build a PINN model for the wave equation.

        Returns:
            PINN model for the projectile motion with
                input: [ (t, x) relative to equation,
                         (t=0, x) relative to initial condition,
                         (t, x=bounds) relative to boundary condition ],
                output: [ u(t,x) relative to equation,
                          u(t=0, x) relative to initial condition,
                          du_dt(t=0, x) relative to initial derivative of t,
                          u(t, x=bounds) relative to boundary condition ]
        """

        # equation input: (t, x)
        tx_eqn = tf.keras.layers.Input(shape=(2,))
        # initial condition input: (t=0, x)
        tx_ini = tf.keras.layers.Input(shape=(2,))
        # boundary condition input: (t, x=-1) or (t, x=+1)
        tx_bnd = tf.keras.layers.Input(shape=(2,))

        # compute gradients
        _, _, _, d2u_dt2, d2u_dx2 = self.grads(tx_eqn)

        # equation output being zero
        u_eqn = d2u_dt2 - self.c*self.c * d2u_dx2
        # initial condition output
        u_ini, du_dt_ini, _, _, _ = self.grads(tx_ini)
        # boundary condition output
        u_bnd = self.network(tx_bnd)  # dirichlet
        #_, _, u_bnd, _, _ = self.grads(tx_bnd)  # neumann

        # build the PINN model for the wave equation
        return tf.keras.models.Model(
            inputs=[tx_eqn, tx_ini, tx_bnd],
            outputs=[u_eqn, u_ini, du_dt_ini, u_bnd])
```

A.4.3 Code for PINN Burger's equation

Code taken directly from the Raissi M. et al paper. Finds an approximate solution the the Burgers' equation, including a solution for noisy data. Requires Tensorflow 1.15, and is incompatible with tensorflow 2.x in its current state.

PINN Class and solver

```
"""
@author: Maziar Raissi
"""

import sys
sys.path.insert(0, '../../Utilities/')

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.interpolate import griddata
from plotting import newfig, savefig
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.gridspec as gridspec
import time

np.random.seed(1234)
tf.set_random_seed(1234)
```

```
class PhysicsInformedNN:
    # Initialize the class
    def __init__(self, X, u, layers, lb, ub):

        self.lb = lb
        self.ub = ub

        self.x = X[:,0:1]
        self.t = X[:,1:2]
        self.u = u

        self.layers = layers

        # Initialize NNs
        self.weights, self.biases = self.initialize_NN(layers)

        # tf placeholders and graph
        self.sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                                     log_device_placement=True))

        # Initialize parameters
        self.lambda_1 = tf.Variable([0.0], dtype=tf.float32)
        self.lambda_2 = tf.Variable([-6.0], dtype=tf.float32)

        self.x_tf = tf.placeholder(tf.float32, shape=[None, self.x.shape[1]])
        self.t_tf = tf.placeholder(tf.float32, shape=[None, self.t.shape[1]])
        self.u_tf = tf.placeholder(tf.float32, shape=[None, self.u.shape[1]])

        self.u_pred = self.net_u(self.x_tf, self.t_tf)
        self.f_pred = self.net_f(self.x_tf, self.t_tf)

        self.loss = tf.reduce_mean(tf.square(self.u_tf - self.u_pred)) + \
                    tf.reduce_mean(tf.square(self.f_pred))

        self.optimizer = tf.contrib.opt.ScipyOptimizerInterface(self.loss,
                                                                method = 'L-BFGS-B',
                                                                options = {'maxiter': 50000,
                                                                           'maxfun': 50000,
                                                                           'maxcor': 50,
                                                                           'maxls': 50,
                                                                           'ftol' : 1.0 * np.finfo(float
        
        self.optimizer_Adam = tf.train.AdamOptimizer()
        self.train_op_Adam = self.optimizer_Adam.minimize(self.loss)

        init = tf.global_variables_initializer()
        self.sess.run(init)
```

```
def initialize_NN(self, layers):
    weights = []
    biases = []
    num_layers = len(layers)
    for l in range(0,num_layers-1):
        W = self.xavier_init(size=[layers[l], layers[l+1]])
        b = tf.Variable(tf.zeros([1,layers[l+1]], dtype=tf.float32), dtype=tf.float32)
        weights.append(W)
        biases.append(b)
    return weights, biases

def xavier_init(self, size):
    in_dim = size[0]
    out_dim = size[1]
    xavier_stddev = np.sqrt(2/(in_dim + out_dim))
    return tf.Variable(tf.truncated_normal([in_dim, out_dim], stddev=xavier_stddev), dty

def neural_net(self, X, weights, biases):
    num_layers = len(weights) + 1

    H = 2.0*(X - self.lb)/(self.ub - self.lb) - 1.0
    for l in range(0,num_layers-2):
        W = weights[l]
        b = biases[l]
        H = tf.tanh(tf.add(tf.matmul(H, W), b))
    W = weights[-1]
    b = biases[-1]
    Y = tf.add(tf.matmul(H, W), b)
    return Y

def net_u(self, x, t):
    u = self.neural_net(tf.concat([x,t],1), self.weights, self.biases)
    return u

def net_f(self, x, t):
    lambda_1 = self.lambda_1
    lambda_2 = tf.exp(self.lambda_2)
    u = self.net_u(x,t)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + lambda_1*u*u_x - lambda_2*u_xx

    return f

def callback(self, loss, lambda_1, lambda_2):
    print('Loss: %e, l1: %.5f, l2: %.5f' % (loss, lambda_1, np.exp(lambda_2)))
    def train(self, nIter):
        tf_dict = {self.x_tf: self.x, self.t_tf: self.t, self.u_tf: self.u}

        start_time = time.time()
        for it in range(nIter):
            self.sess.run(self.train_op_Adam, tf_dict)

            # Print
            if it % 10 == 0:
                elapsed = time.time() - start_time
                loss_value = self.sess.run(self.loss, tf_dict)
                lambda_1_value = self.sess.run(self.lambda_1)
                lambda_2_value = np.exp(self.sess.run(self.lambda_2))
                print('It: %d, Loss: %.3e, Lambda_1: %.3f, Lambda_2: %.6f, Time: %.2f' %
                      (it, loss_value, lambda_1_value, lambda_2_value, elapsed))
                start_time = time.time()

        self.optimizer.minimize(self.sess,
                                feed_dict = tf_dict,
                                fetches = [self.loss, self.lambda_1, self.lambda_2],
                                loss_callback = self.callback)


    def predict(self, X_star):

        tf_dict = {self.x_tf: X_star[:,0:1], self.t_tf: X_star[:,1:2]}

        u_star = self.sess.run(self.u_pred, tf_dict)
        f_star = self.sess.run(self.f_pred, tf_dict)

        return u_star, f_star


if __name__ == "__main__":

    nu = 0.01/np.pi

    N_u = 2000
    layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]

    data = scipy.io.loadmat('../Data/burgers_shock.mat')

    t = data['t'].flatten()[:,None]
    x = data['x'].flatten()[:,None]
    Exact = np.real(data['usol']).T
```

```
np.save('exact_burgers', Exact)

X, T = np.meshgrid(x,t)

X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
u_star = Exact.flatten()[:,None]

x_fine = np.linspace(-1, 1, 1000)
t_fine = np.linspace(0, 1, 1000)
X_fine, T_fine = np.meshgrid(x_fine, t_fine)
fine_mesh = np.hstack((X.flatten()[:, None], T.flatten()[:, None]))

# Doman bounds
lb = X_star.min(0)
ub = X_star.max(0)

######################################################################
######################## Noiseles Data ###############################
######################################################################
noise = 0.0

idx = np.random.choice(X_star.shape[0], N_u, replace=False)
X_u_train = X_star[idx,:]
u_train = u_star[idx,:]

model = PhysicsInformedNN(X_u_train, u_train, layers, lb, ub)
model.train(0)

x_fine = np.linspace(-1, 1, 1000)
t_fine = np.linspace(0, 1, 1000)
X_fine, T_fine = np.meshgrid(x_fine, t_fine)
fine_mesh = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))

u_pred, f_pred = model.predict(fine_mesh)

np.save('predicted_u', u_pred)
np.save('predicted_f', f_pred)


error_u = np.linalg.norm(u_star-u_pred,2)/np.linalg.norm(u_star,2)

U_pred = griddata(fine_mesh, u_pred.flatten(), (X_fine, T_fine), method='cubic')

np.save('grid_u', U_pred)

lambda_1_value = model.sess.run(model.lambda_1)
lambda_2_value = model.sess.run(model.lambda_2)
lambda_2_value = np.exp(lambda_2_value)
```

```
error_lambda_1 = np.abs(lambda_1_value - 1.0)*100
error_lambda_2 = np.abs(lambda_2_value - nu)/nu * 100

print('Error u: %e' % (error_u))
print('Error l1: %.5f%%' % (error_lambda_1))
print('Error l2: %.5f%%' % (error_lambda_2))

######################################################################
########################### Noisy Data ###############################
######################################################################
noise = 0.01
u_train = u_train + noise*np.std(u_train)*np.random.randn(u_train.shape[0], u_train.shape[1])

model = PhysicsInformedNN(X_u_train, u_train, layers, lb, ub)
model.train(10000)

u_pred, f_pred = model.predict(X_star)

lambda_1_value_noisy = model.sess.run(model.lambda_1)
lambda_2_value_noisy = model.sess.run(model.lambda_2)
lambda_2_value_noisy = np.exp(lambda_2_value_noisy)

error_lambda_1_noisy = np.abs(lambda_1_value_noisy - 1.0)*100
error_lambda_2_noisy = np.abs(lambda_2_value_noisy - nu)/nu * 100

print('Error lambda_1: %f%%' % (error_lambda_1_noisy))
print('Error lambda_2: %f%%' % (error_lambda_2_noisy))


######################################################################
############################# Plotting ###############################
######################################################################

fig, ax = newfig(1.0, 1.4)
ax.axis('off')

####### Row 0: u(t,x) ##################
gs0 = gridspec.GridSpec(1, 2)
gs0.update(top=1-0.06, bottom=1-1.0/3.0+0.06, left=0.15, right=0.85, wspace=0)
ax = plt.subplot(gs0[:, :])

h = ax.imshow(U_pred.T, interpolation='nearest', cmap='rainbow',
              extent=[t.min(), t.max(), x.min(), x.max()],
              origin='lower', aspect='auto')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
fig.colorbar(h, cax=cax)
```

```
ax.plot(X_u_train[:,1], X_u_train[:,0], 'kx', label = 'Data (%d points)' % (u_train.shap

line = np.linspace(x.min(), x.max(), 2)[:,None]
ax.plot(t[25]*np.ones((2,1)), line, 'w-', linewidth = 1)
ax.plot(t[50]*np.ones((2,1)), line, 'w-', linewidth = 1)
ax.plot(t[75]*np.ones((2,1)), line, 'w-', linewidth = 1)

ax.set_xlabel('$t$')
ax.set_ylabel('$x$')
ax.legend(loc='upper center', bbox_to_anchor=(1.0, -0.125), ncol=5, frameon=False)
ax.set_title('$u(t,x)$', fontsize = 10)

####### Row 1: u(t,x) slices ##################
gs1 = gridspec.GridSpec(1, 3)
gs1.update(top=1-1.0/3.0-0.1, bottom=1.0-2.0/3.0, left=0.1, right=0.9, wspace=0.5)

ax = plt.subplot(gs1[0, 0])
ax.plot(x,Exact[25,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[25,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.set_title('$t = 0.25$', fontsize = 10)
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])

ax = plt.subplot(gs1[0, 1])
ax.plot(x,Exact[50,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[50,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])
ax.set_title('$t = 0.50$', fontsize = 10)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.35), ncol=5, frameon=False)

ax = plt.subplot(gs1[0, 2])
ax.plot(x,Exact[75,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[75,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])
ax.set_title('$t = 0.75$', fontsize = 10)
```

```
    ####### Row 3: Identified PDE ##################
    gs2 = gridspec.GridSpec(1, 3)
    gs2.update(top=1.0-2.0/3.0, bottom=0, left=0.0, right=1.0, wspace=0.0)

    ax = plt.subplot(gs2[:, :])
    ax.axis('off')
    s1 = r'$\begin{tabular}{ |c|c| }  \hline Correct PDE & $u_t + u u_x - 0.0031831 u_{xx} = 0$ \\  \hlin
    s2 = r'$u_t + %.5f u u_x - %.7f u_{xx} = 0$ \\  \hline ' % (lambda_1_value, lambda_2_value)
    s3 = r'Identified PDE (1\% noise) & '
    s4 = r'$u_t + %.5f u u_x - %.7f u_{xx} = 0$  \\  \hline ' % (lambda_1_value_noisy, lambda_2_value_no
    s5 = r'\end{tabular}$'
    s = s1+s2+s3+s4+s5
    ax.text(0.1,0.1,s)

    # savefig('./figures/Burgers_identification')
```

Plotter

```
"""
@author: Maziar Raissi
"""

import sys
sys.path.insert(0, '../../Utilities/')

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.interpolate import griddata
from plotting import newfig, savefig
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib.gridspec as gridspec
import time

np.random.seed(1234)
tf.set_random_seed(1234)

class PhysicsInformedNN:
    # Initialize the class
    def __init__(self, X, u, layers, lb, ub):

        self.lb = lb
        self.ub = ub

        self.x = X[:,0:1]
        self.t = X[:,1:2]
        self.u = u

        self.layers = layers

        # Initialize NNs
        self.weights, self.biases = self.initialize_NN(layers)

        # tf placeholders and graph
        self.sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                                     log_device_placement=True))

        # Initialize parameters
        self.lambda_1 = tf.Variable([0.0], dtype=tf.float32)
        self.lambda_2 = tf.Variable([-6.0], dtype=tf.float32)

        self.x_tf = tf.placeholder(tf.float32, shape=[None, self.x.shape[1]])
        self.t_tf = tf.placeholder(tf.float32, shape=[None, self.t.shape[1]])
        self.u_tf = tf.placeholder(tf.float32, shape=[None, self.u.shape[1]])

        self.u_pred = self.net_u(self.x_tf, self.t_tf)
        self.f_pred = self.net_f(self.x_tf, self.t_tf)

        self.loss = tf.reduce_mean(tf.square(self.u_tf - self.u_pred)) + \
                    tf.reduce_mean(tf.square(self.f_pred))

        self.optimizer = tf.contrib.opt.ScipyOptimizerInterface(self.loss,
                                                                method = 'L-BFGS-B',
                                                                options = {'maxiter': 50000,
                                                                           'maxfun': 50000,
                                                                           'maxcor': 50,
                                                                           'maxls': 50,
                                                                           'ftol' : 1.0 * np

        self.optimizer_Adam = tf.train.AdamOptimizer()
        self.train_op_Adam = self.optimizer_Adam.minimize(self.loss)

        init = tf.global_variables_initializer()
        self.sess.run(init)

    def initialize_NN(self, layers):
        weights = []
        biases = []
        num_layers = len(layers)
        for l in range(0,num_layers-1):
            W = self.xavier_init(size=[layers[l], layers[l+1]])
            b = tf.Variable(tf.zeros([1,layers[l+1]], dtype=tf.float32), dtype=tf.float32)
            weights.append(W)
            biases.append(b)
        return weights, biases

    def xavier_init(self, size):
        in_dim = size[0]
        out_dim = size[1]
        xavier_stddev = np.sqrt(2/(in_dim + out_dim))
        return tf.Variable(tf.truncated_normal([in_dim, out_dim], stddev=xavier_stddev), dtype=tf.float3

    def neural_net(self, X, weights, biases):
        num_layers = len(weights) + 1

        H = 2.0*(X - self.lb)/(self.ub - self.lb) - 1.0
        for l in range(0,num_layers-2):
            W = weights[l]
            b = biases[l]
            H = tf.tanh(tf.add(tf.matmul(H, W), b))
        W = weights[-1]
        b = biases[-1]
        Y = tf.add(tf.matmul(H, W), b)
        return Y

    def net_u(self, x, t):
        u = self.neural_net(tf.concat([x,t],1), self.weights, self.biases)
        return u

    def net_f(self, x, t):
        lambda_1 = self.lambda_1
        lambda_2 = tf.exp(self.lambda_2)
        u = self.net_u(x,t)
        u_t = tf.gradients(u, t)[0]
        u_x = tf.gradients(u, x)[0]
        u_xx = tf.gradients(u_x, x)[0]
        f = u_t + lambda_1*u*u_x - lambda_2*u_xx

        return f

    def callback(self, loss, lambda_1, lambda_2):
        print('Loss: %e, l1: %.5f, l2: %.5f' % (loss, lambda_1, np.exp(lambda_2)))


    def train(self, nIter):
        tf_dict = {self.x_tf: self.x, self.t_tf: self.t, self.u_tf: self.u}

        start_time = time.time()
        for it in range(nIter):
            self.sess.run(self.train_op_Adam, tf_dict)

            # Print
            if it % 10 == 0:
                elapsed = time.time() - start_time
                loss_value = self.sess.run(self.loss, tf_dict)
                lambda_1_value = self.sess.run(self.lambda_1)
                lambda_2_value = np.exp(self.sess.run(self.lambda_2))
                print('It: %d, Loss: %.3e, Lambda_1: %.3f, Lambda_2: %.6f, Time: %.2f' %
                      (it, loss_value, lambda_1_value, lambda_2_value, elapsed))
                start_time = time.time()

        self.optimizer.minimize(self.sess,
                                feed_dict = tf_dict,
                                fetches = [self.loss, self.lambda_1, self.lambda_2],
                                loss_callback = self.callback)


    def predict(self, X_star):

        tf_dict = {self.x_tf: X_star[:,0:1], self.t_tf: X_star[:,1:2]}

        u_star = self.sess.run(self.u_pred, tf_dict)
        f_star = self.sess.run(self.f_pred, tf_dict)

        return u_star, f_star


if __name__ == "__main__":

    nu = 0.01/np.pi

    N_u = 2000
    layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]

    data = scipy.io.loadmat('../Data/burgers_shock.mat')

    t = data['t'].flatten()[:,None]
    x = data['x'].flatten()[:,None]
    Exact = np.real(data['usol']).T

    np.save('exact_burgers', Exact)

    X, T = np.meshgrid(x,t)

    X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
    u_star = Exact.flatten()[:,None]

    x_fine = np.linspace(-1, 1, 1000)
    t_fine = np.linspace(0, 1, 1000)
    X_fine, T_fine = np.meshgrid(x_fine, t_fine)
```

```
fine_mesh = np.hstack((X.flatten()[:, None], T.flatten()[:, None]))

# Doman bounds
lb = X_star.min(0)
ub = X_star.max(0)

######################################################################
######################## Noiseles Data ###############################
######################################################################
noise = 0.0

idx = np.random.choice(X_star.shape[0], N_u, replace=False)
X_u_train = X_star[idx,:]
u_train = u_star[idx,:]

model = PhysicsInformedNN(X_u_train, u_train, layers, lb, ub)
model.train(0)

x_fine = np.linspace(-1, 1, 1000)
t_fine = np.linspace(0, 1, 1000)
X_fine, T_fine = np.meshgrid(x_fine, t_fine)
fine_mesh = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))

u_pred, f_pred = model.predict(fine_mesh)

np.save('predicted_u', u_pred)
np.save('predicted_f', f_pred)


error_u = np.linalg.norm(u_star-u_pred,2)/np.linalg.norm(u_star,2)

U_pred = griddata(fine_mesh, u_pred.flatten(), (X_fine, T_fine), method='cubic')

np.save('grid_u', U_pred)

lambda_1_value = model.sess.run(model.lambda_1)
lambda_2_value = model.sess.run(model.lambda_2)
lambda_2_value = np.exp(lambda_2_value)

error_lambda_1 = np.abs(lambda_1_value - 1.0)*100
error_lambda_2 = np.abs(lambda_2_value - nu)/nu * 100

print('Error u: %e' % (error_u))
print('Error l1: %.5f%%' % (error_lambda_1))
print('Error l2: %.5f%%' % (error_lambda_2))

######################################################################
########################### Noisy Data ###############################
```

```
######################################################################
noise = 0.01
u_train = u_train + noise*np.std(u_train)*np.random.randn(u_train.shape[0], u_train.shap

model = PhysicsInformedNN(X_u_train, u_train, layers, lb, ub)
model.train(10000)

u_pred, f_pred = model.predict(X_star)

lambda_1_value_noisy = model.sess.run(model.lambda_1)
lambda_2_value_noisy = model.sess.run(model.lambda_2)
lambda_2_value_noisy = np.exp(lambda_2_value_noisy)

error_lambda_1_noisy = np.abs(lambda_1_value_noisy - 1.0)*100
error_lambda_2_noisy = np.abs(lambda_2_value_noisy - nu)/nu * 100

print('Error lambda_1: %f%%' % (error_lambda_1_noisy))
print('Error lambda_2: %f%%' % (error_lambda_2_noisy))


######################################################################
############################# Plotting ###############################
######################################################################

fig, ax = newfig(1.0, 1.4)
ax.axis('off')

####### Row 0: u(t,x) ##################
gs0 = gridspec.GridSpec(1, 2)
gs0.update(top=1-0.06, bottom=1-1.0/3.0+0.06, left=0.15, right=0.85, wspace=0)
ax = plt.subplot(gs0[:, :])

h = ax.imshow(U_pred.T, interpolation='nearest', cmap='rainbow',
              extent=[t.min(), t.max(), x.min(), x.max()],
              origin='lower', aspect='auto')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
fig.colorbar(h, cax=cax)

ax.plot(X_u_train[:,1], X_u_train[:,0], 'kx', label = 'Data (%d points)' % (u_train.shap

line = np.linspace(x.min(), x.max(), 2)[:,None]
ax.plot(t[25]*np.ones((2,1)), line, 'w-', linewidth = 1)
ax.plot(t[50]*np.ones((2,1)), line, 'w-', linewidth = 1)
ax.plot(t[75]*np.ones((2,1)), line, 'w-', linewidth = 1)

ax.set_xlabel('$t$')
ax.set_ylabel('$x$')
ax.legend(loc='upper center', bbox_to_anchor=(1.0, -0.125), ncol=5, frameon=False)
ax.set_title('$u(t,x)$', fontsize = 10)

####### Row 1: u(t,x) slices ##################
gs1 = gridspec.GridSpec(1, 3)
gs1.update(top=1-1.0/3.0-0.1, bottom=1.0-2.0/3.0, left=0.1, right=0.9, wspace=0.5)

ax = plt.subplot(gs1[0, 0])
ax.plot(x,Exact[25,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[25,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.set_title('$t = 0.25$', fontsize = 10)
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])

ax = plt.subplot(gs1[0, 1])
ax.plot(x,Exact[50,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[50,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])
ax.set_title('$t = 0.50$', fontsize = 10)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.35), ncol=5, frameon=False)

ax = plt.subplot(gs1[0, 2])
ax.plot(x,Exact[75,:], 'b-', linewidth = 2, label = 'Exact')
ax.plot(x,U_pred[75,:], 'r--', linewidth = 2, label = 'Prediction')
ax.set_xlabel('$x$')
ax.set_ylabel('$u(t,x)$')
ax.axis('square')
ax.set_xlim([-1.1,1.1])
ax.set_ylim([-1.1,1.1])
ax.set_title('$t = 0.75$', fontsize = 10)

####### Row 3: Identified PDE ##################
gs2 = gridspec.GridSpec(1, 3)
gs2.update(top=1.0-2.0/3.0, bottom=0, left=0.0, right=1.0, wspace=0.0)

ax = plt.subplot(gs2[:, :])
ax.axis('off')
s1 = r'$\begin{tabular}{ |c|c| }  \hline Correct PDE & $u_t + u u_x - 0.0031831 u_{xx} = 0$ \\  \hlin
s2 = r'$u_t + %.5f u u_x - %.7f u_{xx} = 0$ \\  \hline ' % (lambda_1_value, lambda_2_value)
s3 = r'Identified PDE (1\% noise) & '
s4 = r'$u_t + %.5f u u_x - %.7f u_{xx} = 0$  \\  \hline ' % (lambda_1_value_noisy, lambda_2_value_no
s5 = r'\end{tabular}$'
s = s1+s2+s3+s4+s5
ax.text(0.1,0.1,s)

# savefig('./figures/Burgers_identification')
```

A.5 Code for Finite Differences

A.5.1 Example of using FD for Transient Heat Equation

Code was adapted from a snippet used in chemical engineering, designed by Korosh Agha Mohammad Ghasemi from Shiraz University. The BCs, ICs, and time space stepping sequence has been changed, as well as the diffusive properties of the material. Code generates figure 3.1.

```
%----------------------2-D Transient Heat Conduction----------------------
%----------------------No Heat Generation----------------------
%Dr.M.binazadeh
%Korosh Agha Mohammad Ghasemi
%Chemical Engineering at Shiraz University
clc;
clear all;
%% Variable Declaration
n = 50;                 %number of nodes
L = 1;                  %length of domain
W = 1;                  %width of domain
alpha = 1.28*10e-4;          %thermal diffusivity (m^2/s)

m =(n-2)*(n-2);         %construct penta diagonal matrix
sm = sqrt(m);

dx = L/(n-1);           %change in x domain
dy = W/(n-1);           %change in y domain
x = linspace(0,L,n);    %linearly spaced vectors x direction
y = linspace(0,W,n);    %linearly spaced vectors y direction
[X,Y]=meshgrid(x,y);

Tin = 50;              %internal temperature
T  = ones(m,1)* Tin;    %initilizing Space
Ta = zeros(n,n);
A  = zeros(m,m);
Ax = zeros(m,1);
B  = zeros(sm,sm);

dt = 1;                %time step
tmax = 20;              %total Time steps (s)
```

```
t = 0 : dt : tmax;
r = alpha * dt /(dx^2);  %for stability, must be 0.5 or less

Tt = 200;                 %Bottom Wall
Tb = 0;                %Top Wall
Tl = 100;                 %Left Wall
Tr = 25;                 %Right Wall

%% Boundry Conditions
Ta(1,1:n) = Tt;          %Top Wall
Ta(n,1:n) = Tb;          %Bottom Wall
Ta(1:n,1) = Tl;          %Left Wall
Ta(1:n,n) = Tr;          %Right Wall

%% Setup Matrix
for ix = 1 : 1 : m

    for jx =1 : 1 : m

        if (ix == jx)
                A(ix,jx) = (1 + 4*r);

        elseif ( (ix == jx + 1) && ( (ix - 1) ~= sm * round( (ix-1)/sm) ) ) %RHS
                A(ix,jx) = -r;

        elseif ( (ix == jx - 1) && ( ix ~= sm*round(ix/sm) ) )
                A(ix,jx) = -r;

        elseif (ix == jx + sm)
                A(ix,jx) = -r;

        elseif (jx == ix + sm)
                A(ix,jx) = -r;
        else
                A(ix,jx) = 0;
        end
    end
end


for iy = 1 : 1 : sm

    for jy = 1 : 1 : sm

        if (iy == 1) && (jy == 1)
            B(iy,jy) = r * (Tl + Tt);

        elseif (iy == 1) && (jy == sm)
```

```
            B(iy,jy) = r * (Tt + Tr);                           %LHS

        elseif (iy == sm) && (jy == sm)
            B(iy,jy) = r * (Tb + Tr);

        elseif (iy == sm) && (jy == 1)
            B(iy,jy) = r * (Tb + Tl);

        elseif (iy == 1) && (jy == sm)
            B(iy,jy) = r * (Tt + Tr);

        elseif (iy == 1)&&(jy > 1 || jy < sm)
            B(iy,jy) = r * Tt;

        elseif (jy == sm) && (iy > 1 || iy < sm)
            B(iy,jy) = r * Tr;

        elseif (iy == sm) && ( jy > 1 || jy < sm)
            B(iy,jy) = r * Tb;

        elseif (jy == 1) && ( iy > 1 || jy < sm)
            B(iy,jy) = r * Tl;
        else
            B(iy,jy) = 0;
        end

    end
end

 Bx = reshape(B,[],1);  %Convert matrix to vector

%% Solution
for l = 2 : length(t)   %time steps

     Xx = ( T + Bx );

           Ax = A \ Xx;

                 T( 1 : m ) = Ax( 1 : m );

end

Tx = reshape( Ax , sm , sm); %convert vector to matrix

for i = 2 : 1 : n-1

    for j = 2 : 1 :n-1
```

```
        Ta(i,j) = Tx ( i-1 , j-1 );

    end

end

%% Plot

   contourf(X,Y,Ta,50,'edgecolor','none');
        h = colorbar;
        ylabel(h, 'Temperature °C')
        colormap jet
        axis equal

        title(['Top (Tt)= ',num2str(Tb),'°C']);
            xlabel(['Bottom (Tb)= ',num2str(Tt),'°C'])

            yyaxis left
            ylabel(['Left (Tl)= ',num2str(Tl),'°C'])

            yyaxis right
            ylabel(['Right (Tr)= ',num2str(Tr),'°C'])
```