The Applications and Limitations of Physics Informed Neural Networks

Year 3 Project

Noel Howley

Supervised by Tristan Pryer
Mathematical Sciences Department
University of Bath
2023 - 2024

Abstract

This project provides an introduction to Physics Informed Neural Networks (PINNs) and their uses in the real world. Applications in fields such as financial modeling with Black-Scholes and fluid dynamics have been investigated and explored in order to highlight the scope of PINNs. Finally this paper concludes with a discussion, comparing PINNs with more traditional, alternate methods.

Generative AI Mandatory comment

I acknowledge that this work is my own and Generative AI was used as an assistive tool for searching for papers to continue research by myself.

Contents

Chapter 1

Introduction

In this chapter, we will briefly cover the recent booming history of Physics-Informed Neural Networks (PINNs) and some of the example use cases from many papers before diving into the behind the scenes in Chapter 2.

1.1 History

The use of physical laws in Neural Network (NN) models, has lead to the development of what we now refer to as PINNs. This had many implementations in the early explorations of NNs such as for solving differential equations, modelling dynamical systems and approximating solutions to complex relationships. The concept of involving the underlying physical laws to aid the approximater has grown along with the potential means that deep learning methodologies has seen in recent years. As a result, many of the recent papers used in this project are at the cutting edge of the new frontier of the development of Neural Networks and their place in research.

One of the first foundational works PINNs have to offer was by Dissanayake and Phan-Thien in 1994, where they explored the use of NNs for solving Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs), which was a significant breakthrough in NNs in simulating physical systems [3]. Dissanayake was known to describe Neural Networks as a "universal approximator", which is a rather crude way to describe NNs but provides simple intuitiveness into what is seen as a rather incomprehensible, 'black box', model with papers such as [9] offering insights into why NNs are viewed in this way. For readers that would rather choose not to engage in an in-depth exploration, it may be beneficial to take a generalised viewpoint from [3] and choose to naively understand that PINNs and NNs can be viewed as a very computationally expensive mathematical method that has been brought about in recent years due to the improvements in microchips that approximately follows Moore's Law.

The paper by Raissi, Perdikaris, and Karniadakis in 2019 formally introduced the framework of PINNs, showcasing their capability to solve forward and inverse problems involving non linear partial differential equations [12] with examples including the Navier-Stokes equation which describes the motion of fluid mediums, which we will be seeing more of later. Their example showcases PINNs ability to handle modelling the behaviours of fluid flow but also provides a benchmark for PINNs into the field of research for related engineering and scientific projects.

This work also demonstrated the practicality of PINNs in research but also set the stage for the future papers aimed at leveraging PINNs to incorporate domain-specific knowledge into predictive models for many ODEs over time. A key example is highlighted in [21] where Wang and others give example of

the domain of Artificial Intelligence (AI) research papers entering the scene in 1994 by Hutchington [6] with classification using supervised Learning Techniques, through to more advanced Feed-Forward Neural Networks used to predict the value of European call options using a one dimentional Black-Scholes equation [4]. Finally [14] has comprised a literature review with many detailed papers applications of Artificial Neural Networks (ANNs) used to solve variations of Black-Scholes.

More details about PINNs can be found on the recent article [10].

Chapter 2

Introduction to Neural Networks

In this chapter we will cover some basics of the intricacies of NN architectures, linear and non linear activation layers and more, in order to build a level of understanding before introducing PINNs and case examples.

NNs can be thought of as a series of functions aimed at recognising underlying relationships in a set of data through a process that some say mimics the way the human brain operates. As we move from the biological neurons to the artificial networks, it's important to recognise that artificial neurons function through mathematical operations which lay the groundwork for the advanced pattern recognition and data analysis ability.

2.1 Simple Connections

A NNs architecture is built upon a structured framework of interconnected nodes, often compared to biological neurons and organised in layered sequences. The similarity of these computational nodes to biological neurons is generally considered a trivial and rudimentary comparison, suitable for the newcomers to this field but the reader should take caution using this analogy further. Regardless, for newcomers, this comparison might serve as a useful conceptual aid in grasping the fundamentals of NN design.

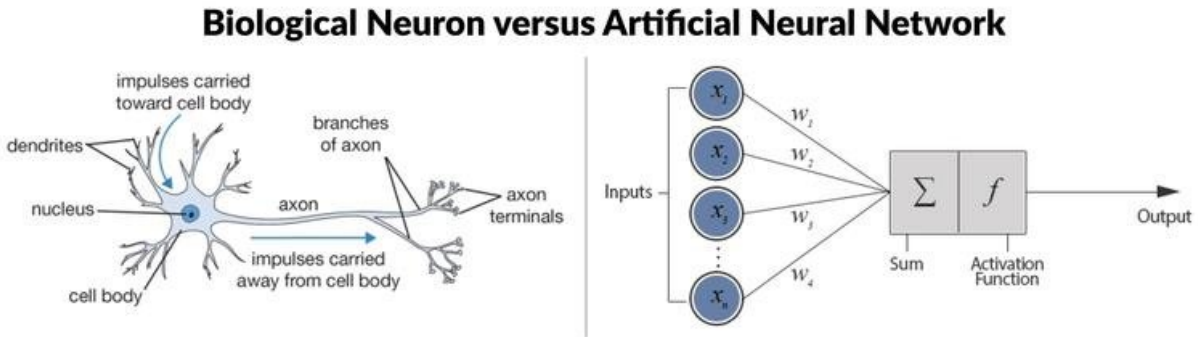

Figure 2.1: Biological Neurons vs. Artificial Neurons [13]

The biological neuron receives signals through its dendrites, processes them in the cell body, and

transmits the output signals through the axon. Similarly, the artificial neuron receives input signals, which is multiplied with the corresponding weight associated with the connection during what is known as the linear phase. Afterwards, the use of activation functions in a non linear phase results in the output.

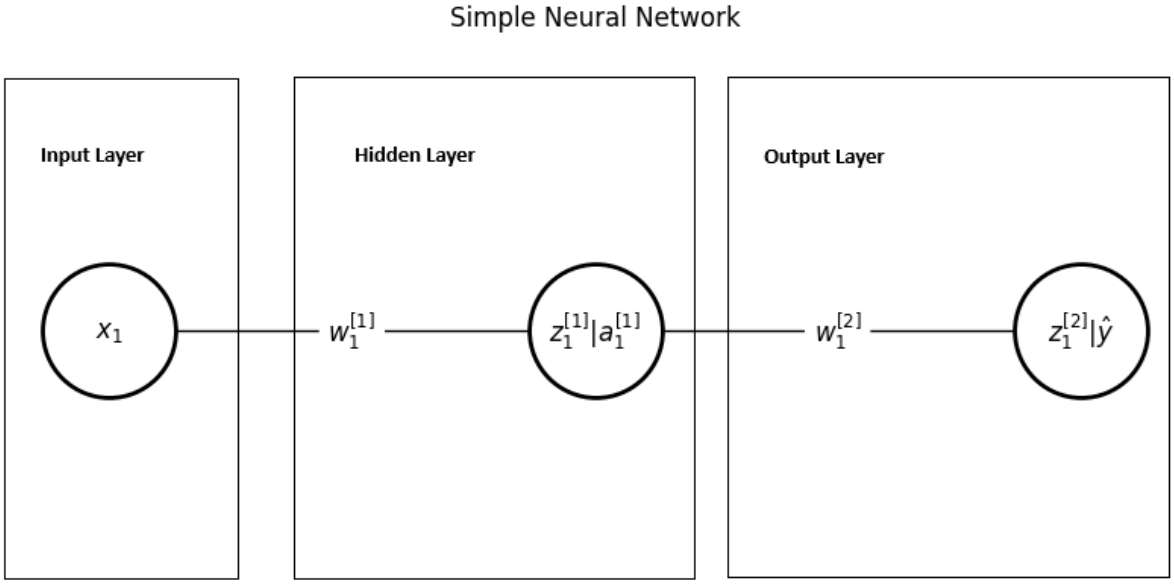

Figure 2.2: Simple Neural Network

2.2 Simple Neural Network

Consider a simple NN as shown in Figure 2.2. This network consists of three layers: an input layer, a hidden layer, and an output layer. The input layer has a single neuron x_1, the hidden layer contains a neuron denoted as $z_1^{[1]}|a_1^{[1]}$, and the output layer includes a neuron represented as $z_1^{[2]}|\hat{y}$.

Definitions and Dimensions

Let:

- $x_1 \in \mathbb{R}$ be the input to the network.
- $w_1^{[1]} \in \mathbb{R}$ and $w_1^{[2]} \in \mathbb{R}$ are the weights, otherwise known as the coefficients of the edges between each node, which are altered during the training phase.
- $b_1^{[1]} \in \mathbb{R}$ and $b_1^{[2]} \in \mathbb{R}$ not seen in the diagram, the biases are coefficients applied to every node and to be altered during the optimisation phase.
- $z_1^{[1]}$ and $z_1^{[2]}$ represent the linear combinations at each layer before applying the activation function.
- $a_1^{[1]}$ and $\hat{y}$ represent the activated outputs of the hidden and output layers, respectively.

The activation a of each neuron in the network is represented by a function of the weighted sum of its inputs. This can be written in mathematical terms as follows:

$$a = f(z) = f\left(w^T x\right) \tag{2.1}$$

where f is the activation function, w is the weight matrix, and x is the input vector. For simplicity, in a 1-1-1 network like this, each layer's activation is calculated using the following linear transformation followed by an activation function:

$$z_1^{[1]} = w_1^{[1]} x_1 + b_1^{[1]} \tag{2.2}$$

$$a_1^{[1]} = f\left(z_1^{[1]}\right) \tag{2.3}$$

$$z_1^{[2]} = w_1^{[2]} a_1^{[1]} + b_1^{[2]} \tag{2.4}$$

$$\hat{y} = f\left(z_1^{[2]}\right) \tag{2.5}$$

Threshold Function

The activation of each node depends on whether the output of the activation function exceeds a certain threshold ϵ. This can be formalised as:

$$G(w, x) = f\left(w^T x\right) - \epsilon = 0 \tag{2.6}$$

Here, $\epsilon > 0$ denotes the threshold value, and G is a function that maps the weights and inputs to the set of real numbers subject to the determined activation function and threshold value chosen. The network is trained to optimise the weights and biases such that this condition is met across the dataset, enabling the model to approximate complex non linear and linear functional relationships inherent in the data.

2.3 Activation Functions

As already referred to as the non linear function f, can also be known more generally as an activation function. An example of this could be the smooth, continuous and infinitely differentiable Sigmoidal function:

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

or otherwise, used just as commonly is the more simple to compute, ReLU function:

$$f(z) = \text{ReLU}(z) = \max(0, z)$$

shown figure 2.3 below.

ReLU and Sigmoidal Functions

As we have seen, the Sigmoid function and the ReLU, otherwise known as the Rectified Linear Unit, activation functions are both non linear transformations. However, they exhibit distinct properties, each

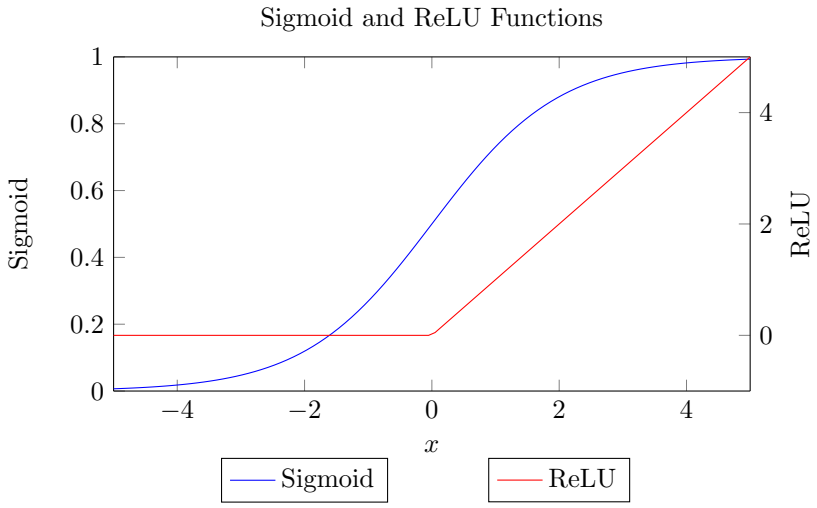

Figure 2.3: Example Activation Functions

with their own advantages and disadvantages.

The Sigmoid function, highlights continuous differentiability, which is very advantageous for gradient-based learning methods, which we touch on later. Unfortunately, it is susceptible to the vanishing gradient problem when considering extreme values of z cause the gradient to tend to zero, which can decelerate or halt the learning process completely. Figure — showcases that for large values of $|z|$ we see that the vanishing gradient effect could be an issue easily. An easy workaround to this issue would be to use the ReLU activation function as mentioned earlier provided $z > 0$, otherwise a specific regularisation or alteration of the Sigmoid function may be necessary to handle the larger values of $|z|$ to stave away the vanishing gradient problem.

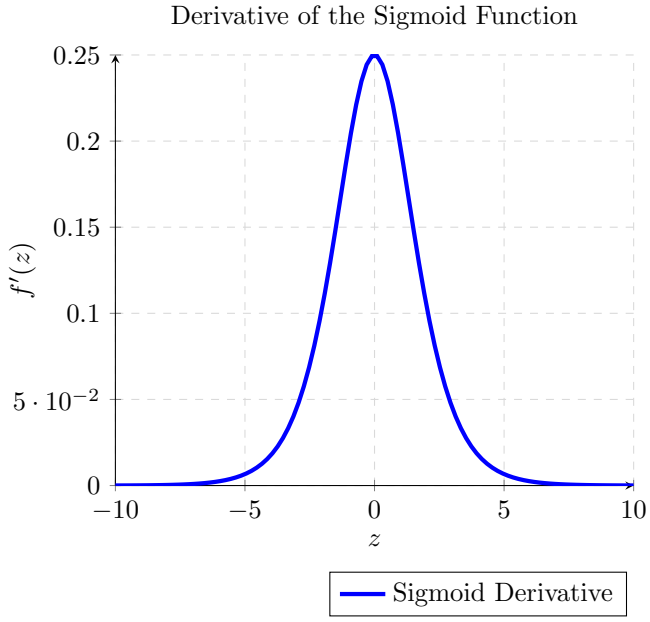

Figure 2.4: The derivative of the Sigmoid function, showing the vanishing gradient problem for large magnitude of z.

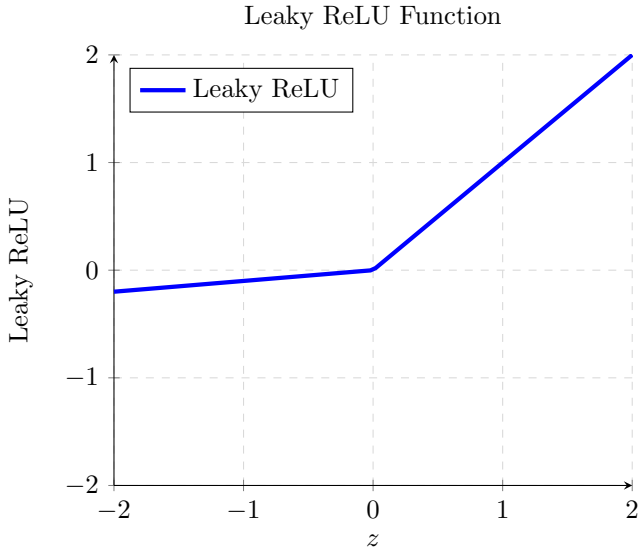

Figure 2.5: The Leaky ReLU function, which allows for a small, non-zero gradient when z is less than zero.

The ReLU function offers a less computationally expensive operation and allows for convergence to happen relatively sooner during training due to its linear nature of half of the domain and lack of the vanishing gradient issue. We note that the gradient for positive inputs is always just 1. However, ReLU is prone to the "dying ReLU" issue, where nodes become completely inactive, meaning z is set to 0 for a particular node, and are unable to "turn back on" due to the gradient for all values of ReLU that are less than the $z = 0$ is also 0. More formally we can write our dying ReLU problem as, $\forall z \leq 0$:

$$\frac{d}{dz}\text{ReLU}(z) = 0 \tag{2.7}$$

This problem does have fixes such as the Leaky Relu:

$$f(z) = \text{Leaky ReLU}(z) = \max(-0.1z, z)$$

Where the gradient for all values of ReLU before the main ReLU condition hold, are set to a very small negative constant, enabling the node to potentially "turn back on" in training and ensure we are not stuck with nodes that are turned off forever, which leads to poor optimisation of the networks weights and biases.

Choosing Between ReLU and Sigmoid Activation Functions

In NN design, selecting the appropriate activation function, such as ReLU or sigmoid, depends on the applications at hand and the NNs architecture. ReLU is commonly preferred in deep NNs due to its ability to accelerate the training process and avoid vanishing gradient problem as already covered. Simply, this function enhances training efficiency and supports effective learning in deep architectures with its straightforward gradient computation.

On the other hand, the Sigmoid function effectively maps input values to a continuous range between 0 and 1. This range is ideal for binary classification tasks where outputs are interpreted as probabilities but also frequently used in the output layers of networks designed for binary classification because of its

probabilistic output. Also, this function is simply to project to further ranges and is commonly chose for such a reason.

Therefore, the selection between ReLU and sigmoid should be tailored to the specific needs of the application, taking into account factors like network depth (size), training efficiency, and the need for probabilistic outputs.

2.4 Loss Functions

The choice of loss function plays a very impactful role in the training dynamics and performance of NNs. Different loss functions have different aspects of the prediction error, leading to variations in training behavior and model effectiveness.

Loss functions quantify the difference between the predicted outputs of NNs and the actual target values. By minimising this loss, the model learns to make predictions that closely match the outcomes and hence allow the models to improve learning. More simply, loss functions quantify the difference between the predicted outputs of the network and the actual target values, quite litteraly a margin of error for the network to work from.

Formally, the loss function $L(\theta) \in \mathbb{R}$, where $\theta \in \mathbb{R}^{N+1}$ represents the parameters of the model (e.g., weights and biases of the nodes), is defined as a function of both the predictions $\hat{y} \in \mathbb{R}^N$ and the true values $y \in \mathbb{R}^N$.

L_1 Loss (Absolute Loss)

The L_1 loss function shown in equation 2.8 is use for its robustness against outliers and its promotion of sparse solutions, which is useful in scenarios where feature selection, otherwise known as identifying the most important or most influential differences in the data, is important. However, its non-differentiability at zero presents optimisation challenges, and its convergence can be slower due to the constant gradient magnitude as errors tend to zero, which hence leaves a trade-off between robustness and optimisation efficiency.

$$L(\theta)_1 = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{2.8}$$

L_2 Loss (Squared Loss)

Shown in 2.9, the L_2 loss function's smoothness and differentiability allow for the use of gradient-based optimisation techniques. It places a greater penalty on larger errors, from above or below, which encourages accuracy in predictions. However, due to this sensitivity there can be issues with overfitting. Overfitting is where the model trains exceptionally well to the "training" data, but so much so that it becomes worse at predicitng the unseen "test" data, implying worse performance in real world environments. Regardless, the L_2 loss is the industry standard to many NN architectures due to the convexity, which allows efficient gradient-based optimisation. Furthermore, its emphasis on penalising larger errors penalises random outliers, making it suitable for regression tasks and contributing to improved model accuracy.

$$L(\theta)_2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.9}$$

Logarithmic Loss (Log Loss)

Finally, in equation 2.10, the logarithmic loss function, useful for classification tasks, is defined for binary classification below. It rewards accurate probabilistic predictions and heavily penalises incorrect classifications, especially those made with high confidence. Despite its potential value in encouraging

model reliability in the binary case, its complexity increases as the number of classifications increase, as seen in multi-class classification problems.

$$L(\theta)_{\log} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.10}$$

Custom Loss Functions

Designing custom loss functions allows for the tailoring of models to specific applications, particullarly important in the case of penalising against certain constraints. This flexibility, can come with heightened risk of overfitting, as the model may become overly optimised for the training data. To counter the overfitting scenario, there are many suitable methods such as the Bias Variance Tradeoff, Cross Validation and Early Stopping. Overall, the correct selection of a loss function is very important for the efficient training and optimal performance of NNs. It requires a careful consideration of the model's application, including the nature of the prediction task, data characteristics, and the presence of outliers.

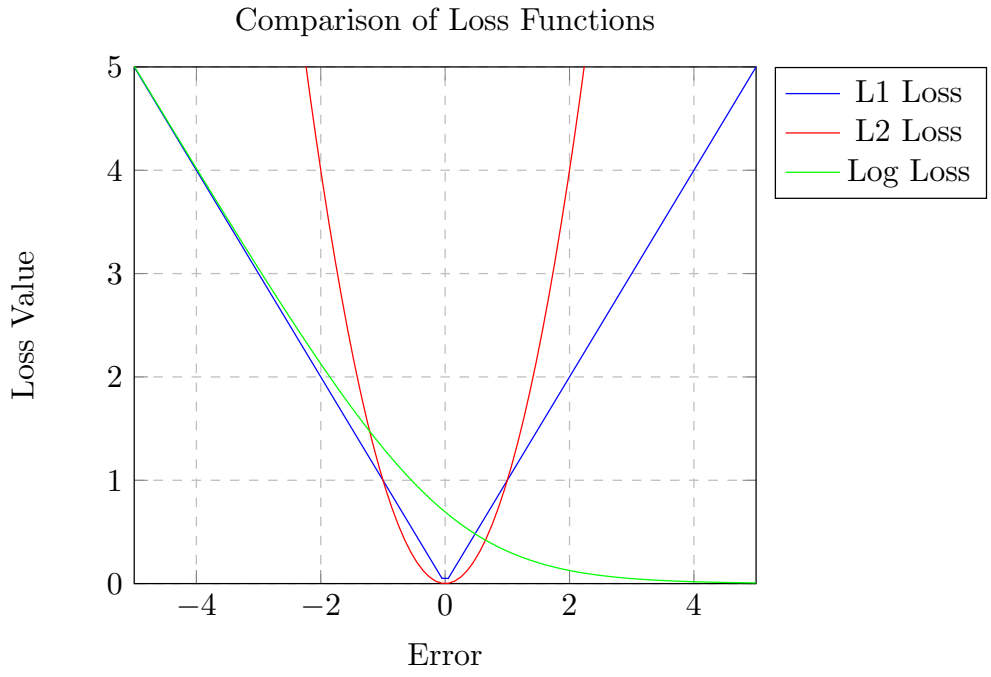

Figure 2.6: Plots of the Loss functions mentioned above

2.5 Training

In the training process of NNs, the minimization of the loss function is achieved over numerous iterations through an optimisation algorithm, typically gradient descent or similar. During each iteration, the algorithm makes adjustments to the model's many parameters, θ, in the direction that reduces the loss function's value, based on the gradient of the loss with respect to these parameters. This iterative process is key to the learning capability of NNs, allowing them to incrementally improve their predictions by refining the weights and biases to better fit the training data. The convergence of this process towards

a minimum loss signifies the network's successful training, equipping it with the generalised ability to make accurate predictions on unseen data.

Gradient Descent in Neural Networks

Consider a simple NN with parameters θ which include all weights and biases across layers. For our Simple NN as defined earlier in Figure 2.2, θ consists of $w_1^{[1]}, b_1^{[1]}, w_1^{[2]}, b_1^{[2]}$.

We will also assume a loss function being the L_2, Loss $L_2(\theta)$, as defined on equation 2.9.

Gradient Descent Update Rule

During each iteration of the training process, the parameters θ are updated with the gradient of the loss function with respect to θ. The update rule can be represented as:

$$\theta_{n+1} = \theta_n - \alpha \nabla_{\theta_n} L_2(\theta_n) \tag{2.11}$$

Here, α represents the learning rate, a hyperparameter that controls the step size during the learning process which is sometimes variable depending on the complexity of the optimisation process. The gradient $\nabla_{\theta_n} J(\theta_n)$ is computed as follows:

$$\nabla_{\theta_n} L_2(\theta_n) = \begin{bmatrix} \frac{\partial L_2}{\partial w_1^{[1]}} \\ \frac{\partial L_2}{\partial b_1^{[1]}} \\ \frac{\partial L_2}{\partial w_1^{[2]}} \\ \frac{\partial L_2}{\partial b_1^{[2]}} \end{bmatrix} \tag{2.12}$$

Further information can be found in the Appendix A.

Supervised Learning and Unsupervised Learning

Supervised learning in the context of NNs involves training models on a labeled dataset where the correct outputs (targets) are known. The primary objective is to develop a model that can accurately predict the output when given new, unseen data. Mathematically, this can be represented by the function $f : X \to Y$, where X is the input data and Y is the output data. The learning process adjusts the model parameters to minimise the difference between the predicted outputs and the actual outputs, typically using a loss function such as L_2 Loss (Mean Squared Error), see Equation 2.9.

Unsupervised learning, on the other hand, deals with input data without explicit labels. The goal is to model the underlying structure or distribution in the data in order to learn more about the data. Common tasks include clustering, where the aim is to group a set of objects in such a way that objects in the same group (or cluster) are more similar to each other than to those in other groups. For example, the K-means clustering algorithm partitions n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This is achieved by minimising the Within-Cluster Sum of Squares (WCSS):

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2$$

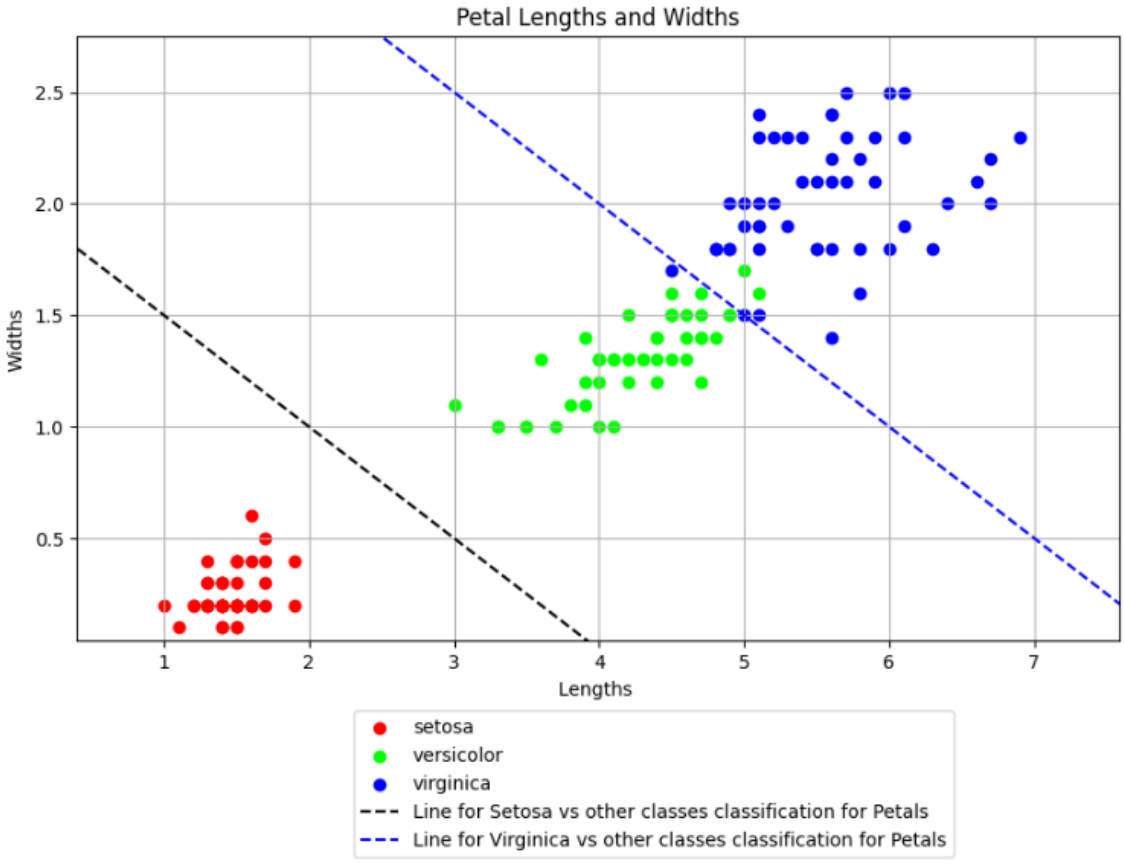

Figure 2.7: Example of the classification of the Petal length and widths unseen by the NN before the optimisation process

where μ_i is the mean of points in S_i.

A simple example of this process can be seen in the case of a network identifying the type of orchid based on the length and width of the Orchid petals only. Providing the training set without the classification of what orchid is a specific type we are ensuring this NN has an unsupervised learning environment for it to complete a classification task based on the WCSS. Figure 2.7 and 2.8 highlight the ability of NNs to correctly classify Orchids based on this fundamental process.

2.6 Multi-Layered Networks

For more advanced and sophisticated networks, we do not just have an output after a single node as shown in figure 2.1, but instead we may have multiple nodes per layer of the network, resulting in a more complex procedure. It is not uncommon to have thousands of nodes and layers resulting in millions of weights between each node. Importantly, this process is similarly defined for the largest networks as it is in the smaller, so understanding the logic as described in the simple connections suffices for the multi-layered case.

The architecture depicted in figure 2.9 represents a multi-layer network with an input layer, a hidden layer, and an output layer. Hidden layers are a substitute name representing for all the layers between the input and output layers of the network, of which there are multiple in most cases.

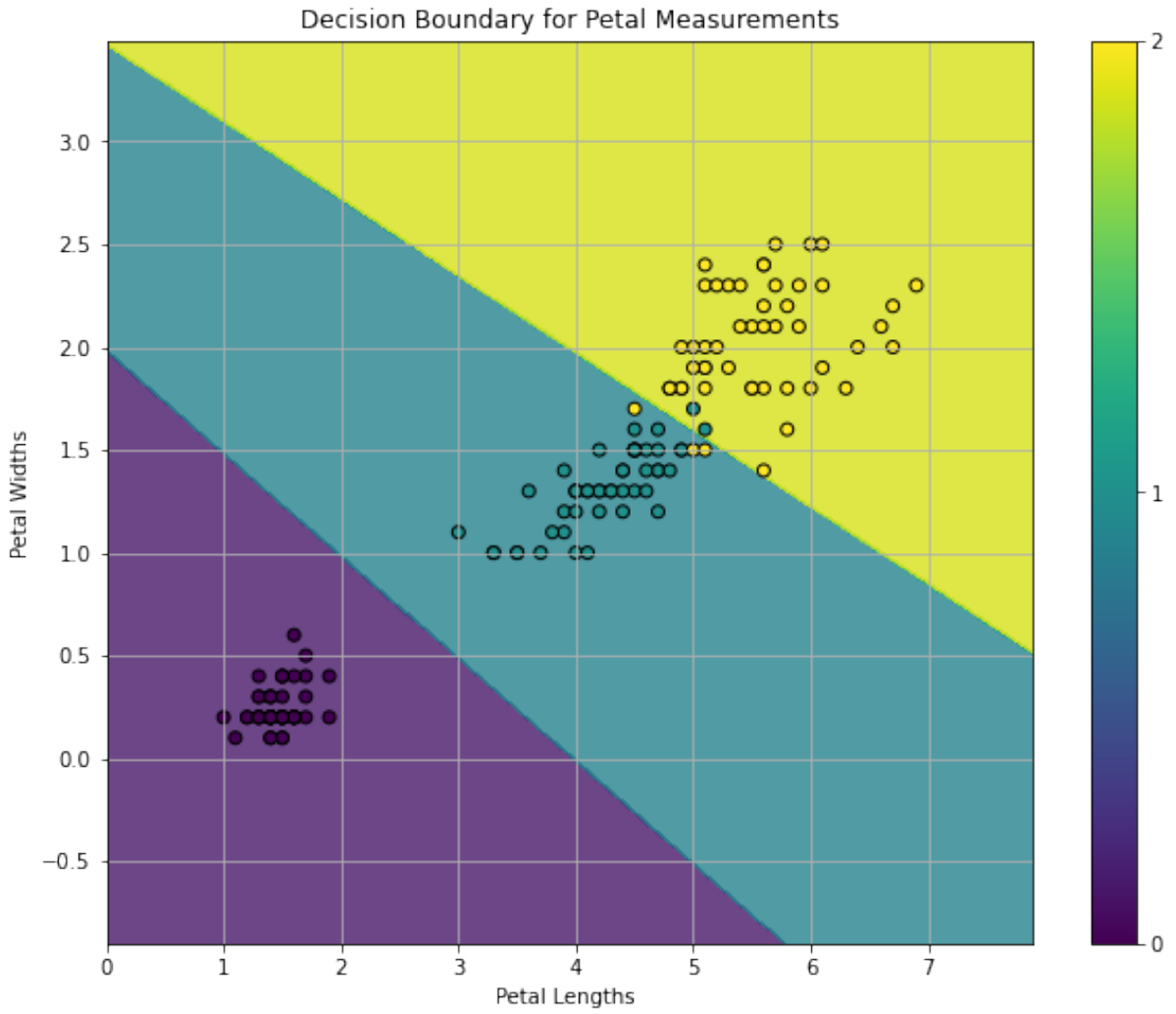

Figure 2.8: Example of the NN correctly classifying the majority of the types of petals with clustering

Figure 2.9 illustrates a classical Multilayer Perceptron (MLP), a type of feedforward NN composed of three primary individual parts: an input layer, at least one hidden layer, and an output layer. This is very similar to the example seen in 2.2 except there are now multiple nodes per layer, and the hidden layer may be home to many interconnected layers which is often a dense interconnected mesh between thousands of node per layer for multiple layers. Each layer is formed by neurons that perform vector-matrix multiplications followed by a non linear transformation as before.

Network Architecture

Given an input vector $\mathbf{x} \in \mathbb{R}^{n_0}$, the MLP processes the data through a series of layers. The first layer, the input layer, consists of n_0 features that are fed directly into the subsequent hidden layers. For each hidden layer l, where $l = 1, \ldots, L-1$, the transformation is defined as follows:

$$\boldsymbol{h}_l = f_l(\boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l) \tag{2.13}$$

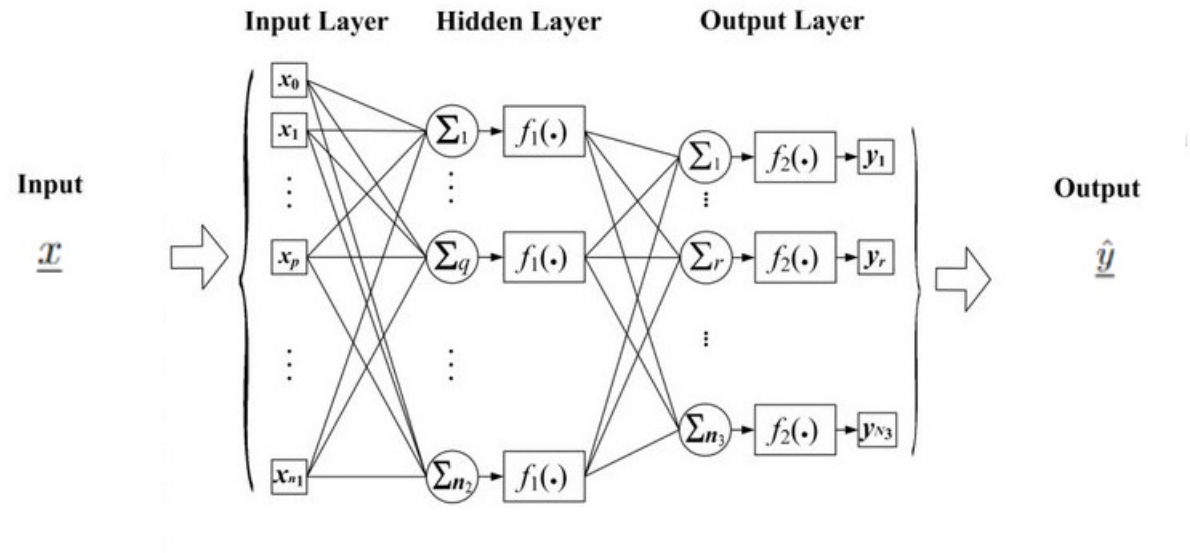

Figure 2.9: Multi Layered Network Architecture from [7]

where $\mathbf{h}_0 = \mathbf{x}$, $\mathbf{h}_l \in \mathbb{R}^{n_l}$ is the output of layer l, $\mathbf{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix, $\mathbf{b}_l \in \mathbb{R}^{n_l}$ is the bias vector, and f_l is a non linear activation function such as ReLU or sigmoid.

The final layer, known as the output layer, generates the prediction $\underline{\hat{\mathbf{y}}} \in \mathbb{R}^{n_L}$:

$$\underline{\hat{\boldsymbol{y}}} = f_L(\boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L) \tag{2.14}$$

Learning Process

The network's learning objective is to find the optimal set of parameters $\{\mathbf{W}_l, \mathbf{b}_l\}_{l=1}^{L}$ that minimize a loss function $\mathcal{L}(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y})$, which measures the difference between the network's output $\hat{\mathbf{y}}$ and the true labels $\mathbf{y}$. This optimisation problem can be stated as:

$$\min_{\{\mathbf{W}_l, \mathbf{b}_l\}} \mathcal{L}(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y}) \tag{2.15}$$

With sufficient training data and proper regularisation techniques, the MLP can approximate a function $N : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ that captures the underlying data distribution, ideally satisfying $N[\mathbf{x}; \{\mathbf{W}_l, \mathbf{b}_l\}] \approx \mathbf{y}$ for all $(\mathbf{x}, \mathbf{y})$ in the training set.

2.6.1 Multi-Layered Network Example: LeNet-5

LeNet-5, developed by Yann LeCun et al. in the late 1990s, is a great example of what convolutional NN's (CNN's) could achieve. This example is designed primarily for the task of classifying handwritten digits, from 0 to 9. An example of what the training dataset looks like can be seen in the figure 2.10.

As one of the first successful applications of CNNs, LeNet-5 aimed to automatically read and interpret handwritten digits, which has many applications including postal mail sorting, bank check processing, and form data entry. It is a prime example of the Network that uses the simplified groundwork covered so far to classify an output, which had been trained during what is known as a supervised learning

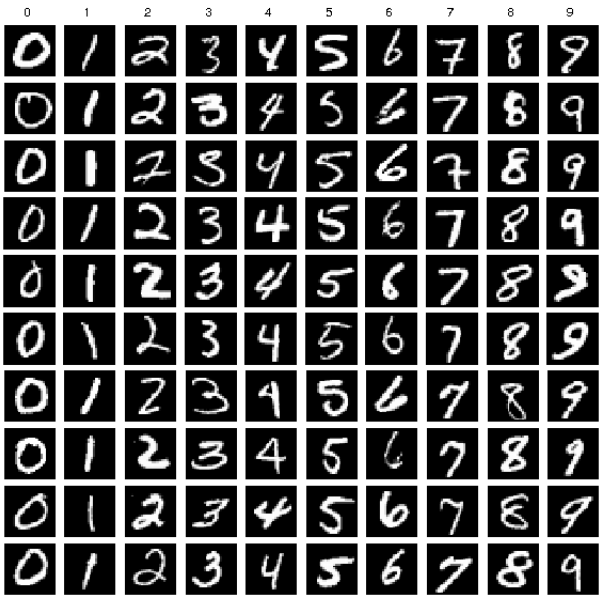

Figure 2.10: Training Data for the Le-Net5 network, known as the MNIST dataset as seen in [1]

environment.

The architecture of LeNet-5 was designed to mimic the way the human eye perceives visuals, using a series of convolutions layers to detect patterns and features in images. Specifically, you can imagine a certain node in a certain layer that is good at identifying completed circles in the data, so for an input of a hand drawn "1", this node would have very low weighting contributing to a non-activation after passing through the linear and non linear phases of the network. Then, once weighing out the outputs, the network would be able to give a certain probability the input is a certain numbered output, so in this instance, due to the hypothetical closed circle node being low weighting, this may imply a low probability of output values of $0, 6, 8, 9$. Other nodes may be able to select other features such as straight lines, bends, loops or perhaps by looking at the intensity of white in each pixel determine other important features. This, as with all nodes in networks, is undetermined by the user but is let to be crafted by the network learning through the training process. I find this acts as a beautiful example of how NNs are once again similar to their biological counterpart. For example, when a young toddler is learning to walk, they may not consciously understand all the necessary connections in each muscle that is needed to allow their body to balance and move forward, much like the system in place by nodes in a network not being defined in advance, but corrected with practice.

The more nodes, layers and computational power given to the architecture of the network, allows for a greater potential accuracy of the network. However, implementation of correct training procedures in the network ensures a high yield of classification accuracy.

As the problem in this case is working with data that has a dimension (inherent feature of image data), this brings in specific pooling layers to reduce spatial resolution while preserving key features. An example of a visualisation of the networks' architecture can be seen in figure 2.11, where we now deal with a 3D component of the architecture due to the extra dimension given from the image data. As we have seen before, fully connected layers to perform classification based on the detected features if rife but importantly, there are 10 output layers in the final layer due to the network completing a classification problem of the digits 0-9, contrasting what we expect to see from non classification neural nets where

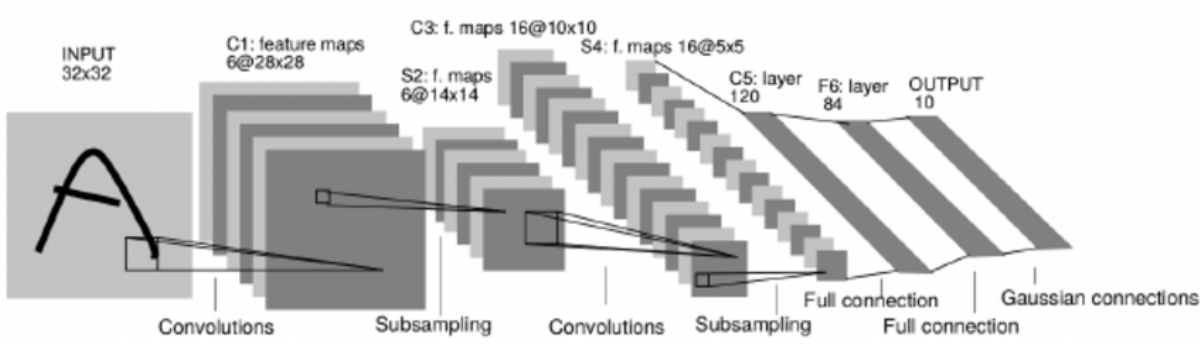

Figure 2.11: Architecture of Le-Net5 [17] and [8]

one output may suffice.

Through this structured method, LeNet-5 could effectively learn to recognise various handwritten digits from pixel data, demonstrating the potential of deep learning for image-based tasks well before the deep learning boom of the 2010s.

This example sheds light onto how NNs are able to classify objects in pictures, with the help of many connected layers and nodes throughout, along with a suitable training set for the network to learn from. Since the creation of these networks,much more sophisticated networks that are able to identify digits, but also pictures of things such as animals, objects and places. Further research into solving CAPTCHA's with deep learning Convolutional Neural Networks (CNNs) are followed up in papers such as [22].

Chapter 3

Introduction to Physics Informed Neural Networks

In this chapter we will seek to clearly define how a PINNs differ to NNs by looking at the fundemental principles of PINNs before diving into a very useful property of PINNs and NNs.

PINNs differ from traditional NNs that learn purely from data, due to PINNs incorporating known physical laws or governing equations directly into the learning process, making them suited for projects where understanding the relationship governed by the PDE's is of most importance.

3.1 Key Principles of PINNs

PINN's Loss Function

The unique selling point of PINNs lies in their ability to encode governing physical laws, typically differential equations, directly into the NN's structure. This is achieved by embedding these laws into the loss function that the network minimises throughout its training process, just like a normal NN. Consider a general physical system described by the following differential equation:

$$\mathcal{N}[u(x);\lambda] = 0, \tag{3.1}$$

where $\mathcal{N}[\cdot]$ denotes a differential operator, $u(x)$ is the solution function, x represents independent variable which sometimes takes the form of spatial coordinates or time, and λ is a vector of parameters within the physical law we believe differential system ($\mathcal{N}[\cdot]$) follows.

In PINNs, the solution $u(x)$ is approximated by a NN $u_\theta(x)$, where θ represents the network's parameters (weights and biases).As we already know, the network is trained to minimize a loss function $\mathcal{L}(\theta)$ comprising two main individual parts: a data fidelity term and a physics-informed term. The physics-informed term, enforcing the physical laws, is formulated by evaluating the differential operator on the NN's output and penalising deviations from zero. Here is an example assuming the L_2 loss:

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{N_p}\sum_{i=1}^{N_p} |\mathcal{N}[u_\theta(x_i);\lambda]|^2, \tag{3.2}$$

where N_p is the number of points enforcing the physical law, and x_i are these points, located within the domain or on its boundaries.

Raissi et al. [12] cover this concept, demonstrating the application of PINNs to solve complex physical problems, such as fluid dynamics governed by the Navier-Stokes equations and other fluid dynamics based ODEs by integrating these differential equations into the NN's learning process [12]. This method ensures that the network's predictions follow the rules of the assumed physical properties, but also makes it possible to tackle inverse problems. Inverse problems are where we are able to approximate the specific input arguments or assumed constants within these physical rules based on the information we gather from data, we will delve more in depth onto this highly import feature momentarily.

This mix of physical laws with PINNs marks a big change in modelling as it combines the ability of NNs to predict outcomes with the core principles of physics. Ultimately helping solve difficult problems from multiple fields in a unique way.

By being able to understand the behaviour of the data, we enable models to not just mimic observed outcomes but to understand the features and use the fundamental processes that produce these outcomes. This depth of understanding allows for models' ability to predict under conditions or in scenarios where direct observations might be sparse or unavailable which is a common challenge in scientific and engineering domains.

A formal definition of the loss function for such a PINNs system looks as follows:

$$\mathcal{L}_{\text{PINNS}}(\theta) = \lambda_1 \cdot \mathcal{L}_{\text{physics}}(\theta) + \lambda_2 \cdot \mathcal{L}_{data}(\theta), \tag{3.3}$$

In the above, $\mathcal{L}_{\text{physics}}(\theta)$ is the loss as defined earlier in equation 3.2 with respect to the fundamental relationships or supposed laws governing the system, enabling the learning process to inherently respect these underlying principles. $\mathcal{L}_{\text{data}}(\theta)$, on the other hand, ensures that the model remains anchored to empirical data from the training set and would take the form as defined before from equations 2.8, 2.9 or 2.10 depending on the type of problem at hand as discussed.

The balancing act performed by λ_1 and λ_2, which are generally predefined variables known as hyperparameters, allows for a tailored method that can be adjusted based on the availability of data or bias the reasearcher may wish to take to the physical model in the system. A nice example of a simple hyperparamter tuning can be seen in Figure 3.1 where the choice between a strong sinusoidal behaviour is over-influencing the loss function in charge of the physical underpinning of the system rather than the data fidelity loss function.

3.2 Forward and Inverse Problems in PINNs

As touched on in the previous section, PINNs have also shown use in addressing both forward and inverse problems, setting them apart from more traditional numerical methods and approximations. For instance, in the field of fluid dynamics, PINNs have been used to predict flow patterns around obstacles based on initial fluid properties and, uniquely, used to infer viscosity coefficients from observed flow data. This ability to infer the parameters in the equation is an extremely valueable aspect as some data collection can either be extremely dangerous or extremely costly.

This ability really does set PINNs apart from their competitors in real-world applications as progress in approximating solutions is possible without much training data. Another example of this can be seen in [2], with their inferred predictions in figure 3.2.

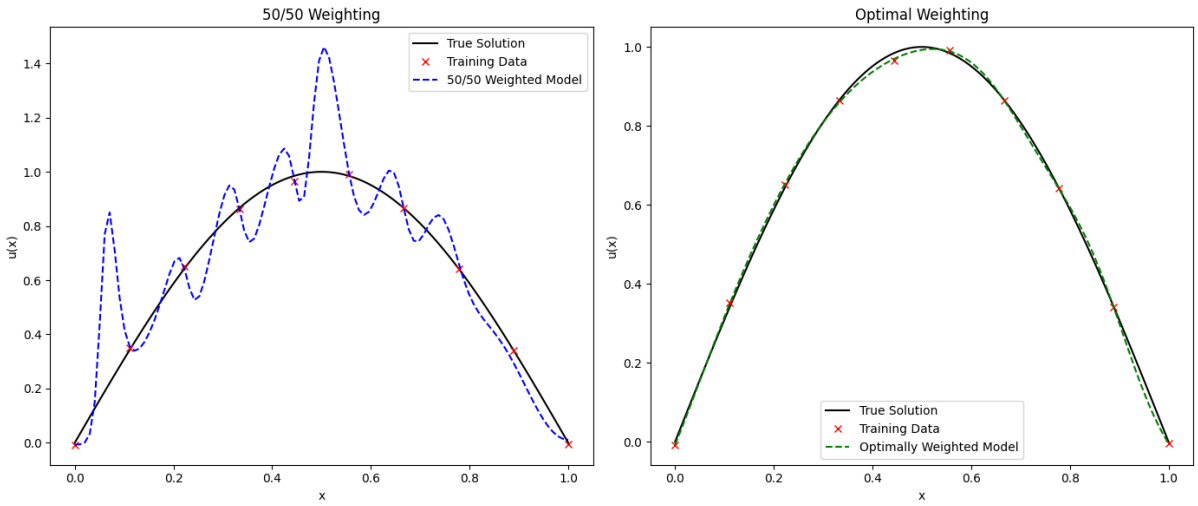

Figure 3.1: An example of optimal weighting of the parameter λ

Forward Problems

In forward problems, the aim is to predict the system's behavior given a set of initial conditions and parameters. The formulation for a forward problem in PINNs can be expressed as seeking the solution $u(x)$ that satisfies the governing physical equations and initial/boundary conditions. This has already defined in equation 3.1.

Inverse Problems

On the other hand, inverse problems focus on determining unknown parameters or initial conditions of a system based on observations of its behavior. This process is known as drawing inference from the data to infer a parameter values and is very beneficial for real world applications when dealing with hard to measure parameters. Put simply, this is similar to solving a puzzle where the outcomes are known, but the pieces (parameters) need to be identified. PINNs approach these problems by adjusting their internal parameters (weights and biases of the network), so the outputs from the network, when fed through the physical model, align with the observed data. Hence, this process estimates the unknown parameters θ but does so in a manner that is consistent with the physical laws of the system. The ability to solve inverse problems is very powerful in scenarios where direct measurement of parameters is challenging or impossible, which is certainly the case throughout the complex fields PINNs are applied to but also for more simple relationships that suffer from a data collection standpiont for ethical or economical reasons. Such reasons could be collecting the Energy-Density relationship for different matter of Proton Beam Radiotherapy on changing density of tissue such as brain, lung or other organs as was the case by the Institute of Mathematical Innovation (2022) or even collecting dangerous and time consuming values of soil moisture and resistivity as was the case from [15].

By addressing both forward and inverse problems, PINNs serve as a bridge between data-driven modeling and traditional physics-based simulation techniques, offering a complete framework for predicting complex systems with the use of inferring variables.

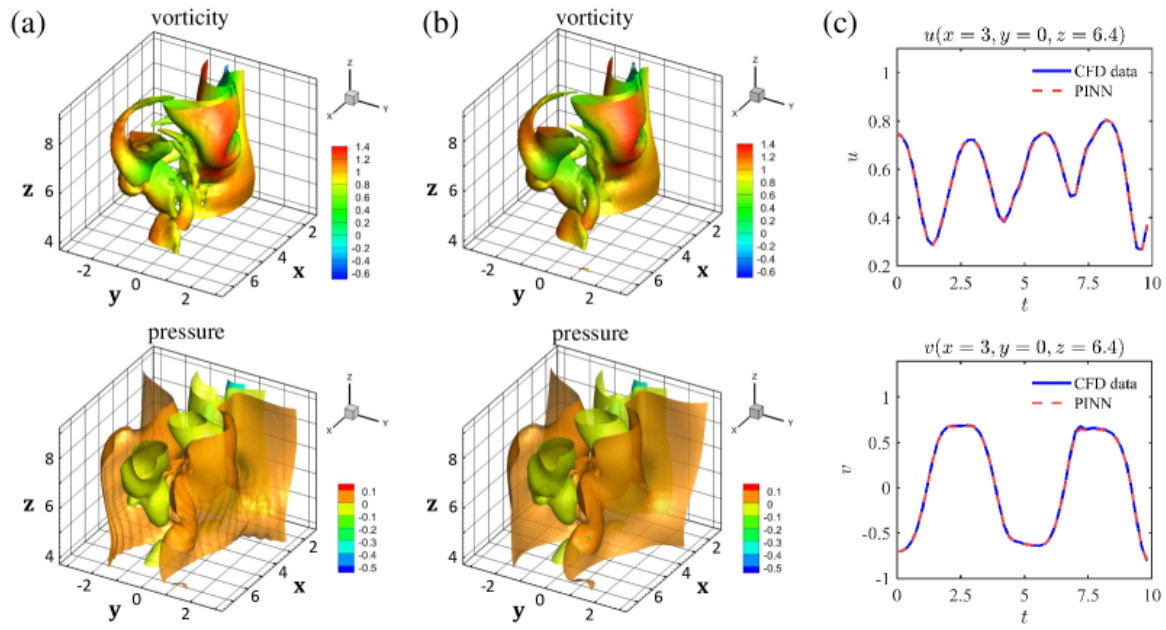

Figure 3.2: Case study of PINNs for incompressible flows: inference result of PINNs for Case 2 from [2]

3.2.1 Potential Drawbacks:

While PINNs offer advantages, they face challenges such as balancing the individual parts of the loss function and ensuring numerical stability during training. Balancing involves adjusting the weight, λ, between fidelity to data and adherence to physical laws.

As already discussed this hyperparameter is a critical factor for the model's success that can be difficult to optimise, but given enough processing power and a correct architecture, the model may be able to infer this too.

A work around may be, early in training, more weight might be given to fitting the data (data fidelity), while later, emphasis could shift towards satisfying the physical laws (physics-informed term). This shift occurs dynamically based on specific criteria such as the predictive power on unseen data in the test set. This ensures that the model learns effectively from both the available data and the underlying physical principles, leading to more accurate and physically plausible predictions. This adaptive method, as shown in the work [19] in PINNs, demonstrates enhanced model performance in fluid dynamics simulations, but can be abstracted to other domians just as easily.

Numerical stability is another key factor in training PINNs, as the numerical methods used within the PINNs framework can impact the convergence of the model.

The computation of derivatives is very susceptible to numerical issues such as exploding or vanishing gradients as mentioned earlier with the use of Sigmoid and ReLU. Exploding gradients can cause instability during training with excessively large updates, whereas vanishing gradients can halt the learning process entirely due to excessively small updates. Work arounds can once again be seen with the use of Stochastic Gradient Descent, Minibatch Gradient Descent and Batch Gradient Descent.

To address these challenges, several strategies are used within the PINNs framework to help the optimisation process along. More advanced activation functions are chosen to maintain the magnitude of gradients and inputs are normalised to constrain the gradient values during backpropagation (see Appendix A for backpropagation). More advanced optimisation algorithms like Adam are often used to

adapt learning rates and scale updates according to the magnitudes of historical gradients.

Chapter 4

Applications

In this chapter we seek to introduce some key applications of PINNs building off some of the topics mentioned so far. This chapter covers a very well known financial model and some equally well known fluid dynamics cases, as well as how PINNs can and have been able to be used in each application.

4.1 Black-Scholes

A financial derivative is a financial instrument whose value is dependant upon the values of other, more fundamental, underlying assets. Common examples of underlying assets are stocks, bonds, and interest rates.

An option, a common type of derivative, serves dual purposes:

- *Hedging:* Reducing risk from potential future changes in market variables.
- *Speculation:* Betting on the anticipated direction of market variables.

The process of pricing these derivatives in what is known as derivative pricing is highly important and has gained attention in both financial industries and academic circles. One well-known model for pricing options is the Black-Scholes model for European options, which is given by the formula:

$$C(S,t) = S_t\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2), \tag{4.1}$$

where

$$d_1 = \frac{\log(S/K) + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}},$$
$$d_2 = d_1 - \sigma\sqrt{T-t}.$$

Here, $C(S,t)$ represents the price of the call option, S_t is the current stock price, K is the strike price, r is the risk-free interest rate, σ is the volatility of the stock, T is the time to maturity, and Φ denotes the cumulative distribution function of the standard normal distribution.

The difference between European and American options primarily lies in their exercise timings and flexibility. European options restrict the holder to exercising the option only at its expiration, which simplifies decisions but could potentially limit profit opportunities. On the other hand, American options allow for exercise at any time before and including the expiration date, offering greater strategic depth

and potential for higher returns, especially in volatile markets. This difference therefore influences the valuation and risk of the options but also their usage in trading strategies.

With this assumption in mind, lets define the terms.

- $C(S, t)$: Price of a European call option.
- S_t: Current stock price.
- Φ: Cumulative distribution function (CDF) of the standard normal distribution.
- d_1 and d_2: Calculated values influencing the likelihood of the option being in-the-money at expiration.
- K: Strike price of the option, the price fixed by the seller.
- e: Mathematical constant.
- r: Risk-free interest rate used for discounting the expected payoff to present value.
- T: Time to expiration of the option.
- σ: Volatility of the stock's returns.
- log: Natural logarithm, used in calculations involving price ratios and adjustments.

Example of an Option Contract

Consider a European call option on a stock. The details of the option contract are as follows:

- Underlying stock price (S_0): \$100
- Strike price (K): \$105
- Expiration (T): 1 year from now
- Risk-free rate (r): 5%
- Volatility (σ): 20%

The holder of this call option has the right, but not the obligation, to buy the stock for \$105 at the end of the year. The cost (premium) of this option can be calculated using the Black-Scholes formula.

Payoff of a Call Option

The payoff of a call option at expiration is given by:

$$\text{Payoff} = \max(S_T - K, 0),$$

where S_T is the stock price at expiration. The payoff function means that if the stock price at expiration (S_T) is above the strike price (K), the payoff is $S_T - K$; otherwise, it is zero. A graphical example can be seen in Figure ——

Figure —— highlights the sensitivity of European call and put options' payoffs near the strike price of \$105. The options' payoffs are shown by the blue line (call option), which depict how the payoff transitions based on the stock price S near K. A more mathematical complete picture can be drawn by assuming the options payout is symmetric, which is not normally the case, and hence we can use the epsilon delta criterion as defined below:

$$\text{Given } \mathrm{x} \in X^{\mathbb{R}N}, f : X^{\mathbb{R}N} \to Y^{\mathbb{R}N}, a, L \in \mathbb{R}^N, N \in \mathbb{N}$$
$$\forall \epsilon > 0, \exists \delta > 0 \text{ such that } 0 < |x - a| < \delta \implies |f(x) - L| < \epsilon \quad (4.2)$$

This criterion can be mathematically expressed as: if $|S - K| < \delta$ then the payoff change must be greater than ϵ. Hence noting the relationship between stock price fluctuations and option profitability.

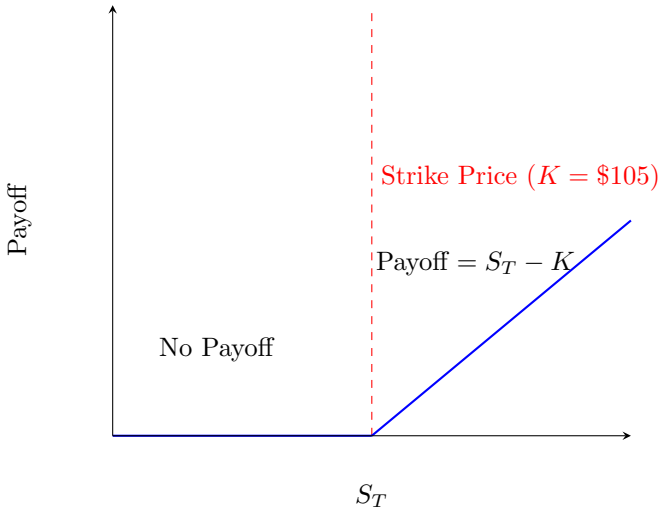

Figure 4.1: Payoff diagram of a European call option

Use of the Black-Scholes Model in Financial Markets

The Black-Scholes model has been a fundamental part of the financial industry for pricing European style options since its introduction in the early 1970s. The model's applicability have made it commonly used amongst various market sectors. Over the years, the model has been adapted to cater to a variety of financial instruments, leading to a broad spectrum of derivative valuation models based on its fundamental principles.

However, the model's assumptions regarding constant volatility and risk-free interest rates have drawn criticism, especially in response to shock responses such as the financial crises that has demonstrated the market's dynamic nature [16]. These challenges have encouraged the development of adapted models that incorporate varying volatility and other market factors [5].

Application of PINNs to Black-Scholes Equation

The paper by [20] represents new improvements in the application of PINNs to financial mathematics, specifically in solving the Black-Scholes partial differential equation (PDE) used for option pricing. This approach covers the core mathematical model but also uses deep learning techniques to test the model.

Black-Scholes Equation as a PDE

The Black-Scholes model is governed by the following PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0, \tag{4.3}$$

where V represents the option price, S the stock price, r the risk-free interest rate, σ the volatility of the stock, and t the time.

PINNs Proposal

As already discussed, PINNs integrate known physics, represented by differential equations, into the learning mechanism of NNs. In the context of the Black-Scholes equation, PINNs approximate the solution $V(S,t)$ by training a NN, denoted as $V_\theta(S,t)$, where θ encapsulates the network parameters. The training involves minimising a loss function with two terms as seen in 3.3:

It is likely that this PINN would be constructed with the $\mathrm{L}_{\mathrm{Physics}}$ term in the overall loss function of the PINN to be defined with the L_2 loss function and hence would be defined as such:

$$\mathrm{L}_{\mathrm{Physics}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial V_\theta}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V_\theta}{\partial S^2} + rS\frac{\partial V_\theta}{\partial S} - rV_\theta\right)^2, \tag{4.4}$$

evaluated at different points in the domain of (S,t).

4.1.1 Structural Benefits of Feed Forward Neural Networks in PDEs

As previously outlined in our discussion on NNs and their applicability to differential equations, the feed-forward Neural Network (FFNN) architecture plays a key role. The FFNN's method, mainly its layer-wise information propagation without feedback loops, makes it very suitable for addressing the computational challenges posed by PDEs.

Architecture of Feed Forward Neural Networks

Considering the complexity involved in solving PDEs, the FFNNs provides a quick and easy method for approximating solutions. The architecture of an FFNNs, as discussed in DeepXDE [11], involves distinct layers, each responsible for progressively abstracting and transforming the input data. The formal structure of an FFNN, adhering to an L-layer configuration, is expressed as:

$$\text{For the input layer } \mathcal{N}_0, \quad \mathcal{N}_0(\mathbf{x}) = \mathbf{x}, \tag{4.5}$$

$$\text{For hidden layers } \mathcal{N}_l, \quad \mathcal{N}_l(\mathbf{x}) = \sigma(\mathbf{W}_l\mathcal{N}_{l-1}(\mathbf{x}) + \mathbf{b}_l), \quad 1 \le l < L, \tag{4.6}$$

$$\text{For the output layer } \mathcal{N}_L, \quad \mathcal{N}_L(\mathbf{x}) = \mathbf{W}_L\mathcal{N}_{L-1}(\mathbf{x}) + \mathbf{b}_L, \tag{4.7}$$

where the layer index l ranges from 1 to $L-1$, denoting the transition from input towards output. Here, $\mathbf{W}_l$ and $\mathbf{b}_l$ are the trainable parameters, signifying the weight matrix and bias vector at the l-th layer, respectively.

Advantages of FFNNs

The advantage of using FFNNs for financial derivatives pricing, is the model's use to approximate complex non linear functions, as is necessary in the Black-Scholes model and other similar financial models. This

is evident in the work of Wang et al. [20], where FFNNs were used to resolve the Black-Scholes equation with hgh efficiency.

The absence of feedback loops in FNNs simplifies the learning process and allows for a clear gradient flow during backpropagation, an understated but very important part of the optimisation process that is used when training the network to adhere to the requirements of PDE solutions.

In summary, FFNNs represent an effective means to an end to solve PDEs, and when coupled with Physics-Informed methodologies, they are well equipped for numerical problem-solving in finance.

Integrating PINNs with the Black-Scholes model showcases a more advanced model for financial institutions seeking to maximize profitability. It also enables improved pricing accuracy by adapting to dynamic market conditions, therefore allowing more competitive pricing for options and bolstering trading profitability through the use of inferred parameters. Moreover, PINNs' power to update model parameters allows for a clearer assessment of risks, outperforming the prior standard Black-Scholes model.

4.2 Fluid Dynamics

Computational fluid dynamics (CFD) is an essential tool in the analysis of fluid flow and related fields across various engineering disciplines. Traditional numerical methods for CFD, such as finite element and finite volume methods, often require more computational resources, especially when dealing with complex boundary conditions and transient behaviour like vortex shedding.

Vortex shedding is a fluid dynamics phenomenon that occurs when a fluid flows past a bluff body, such as a cylinder or a sphere. As the fluid flows around the object, alternating vortices are formed on the downstream side of the body. These vortices are shed in a repeating pattern, creating a swirling flow known as a vortex street. The process of vortex formation and subsequent shedding leads to oscillating forces acting on the body, which can cause it to vibrate.

This phenomenon is very important in many engineering applications because these oscillations can lead to structural fatigue or failure, in structures exposed to fluid flows, like bridges, towers, and offshore platforms. Understanding vortex shedding is an importnt area of study for those who design structures that must withstand these forces or for those who are developing strategies to mitigate their effects, such as modifying the shape of the body to alter the flow pattern or using dampers to absorb the energy of the oscillations. In computational fluid dynamics, accurately simulating vortex shedding is challenging due to its complex, transient nature and the need for high-resolution models to capture the dynamics accurately.

4.2.1 Navier-Stokes Flow Around a Cylinder

In a paper exploring of computational fluid dynamics, a paper [18] investigated the transient behavior of a fluid in a conduit obstructed by a circular barrier, focusing on the phenomenon of vortex shedding — a scenario more commonly challenging for computational models. Prior studies suggested limitations in PINNs for dynamic flow simulations, specifically in capturing vortex shedding without steady-state assumptions. Here, Wang refuted these claims by demonstrating that adequately configured PINNs can effectively simulate vortex shedding dynamics.

Wang considered a fluid with a density $\rho = 1.0$ governed by the time-dependent incompressible Navier-Stokes equations:

$$\mathbf{u}_t + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p - \nu \Delta \mathbf{u} = 0,$$
$$\nabla \cdot \mathbf{u} = 0,$$

where $\mathbf{u} = (u, v)$ represents the velocity field and p the pressure. The kinematic viscosity is $\nu = 0.001$.

PINN Simulation and Results

In Wang's research, the PINNs uses a specially adapted multi-layer perceptron (MLP) architecture, designed to handle the complexity of fluid dynamics simulations. The training process involved an extensive sequence of 2×10^5 training steps for each data segment, this demands a very significant computational resources. This training routine involves a complex balance among over ten different loss terms, each corresponding to different physical conditions and constraints used for the simulation. Manual tuning of these hyperparameters is practically non feasible due to their complex interdependence among them, hence leaving a lot on the table for PINNs.

The potential value of the model is demonstrated through the simulation results at $T = 10$, where the predicted velocity and pressure fields are displayed. These results vividly illustrate the model's capacity to initiate but also accurately trace the development of vortex shedding as the simulation progresses. This capability marks a significant advancement over some traditional numerical models, which often struggle to capture such intricate fluid dynamics . The evidence provided by Wang's work highlights the potential of PINNs to change our approach to simulating and understanding complex fluid behaviors that were previously beyond the reach of conventional numerical techniques.

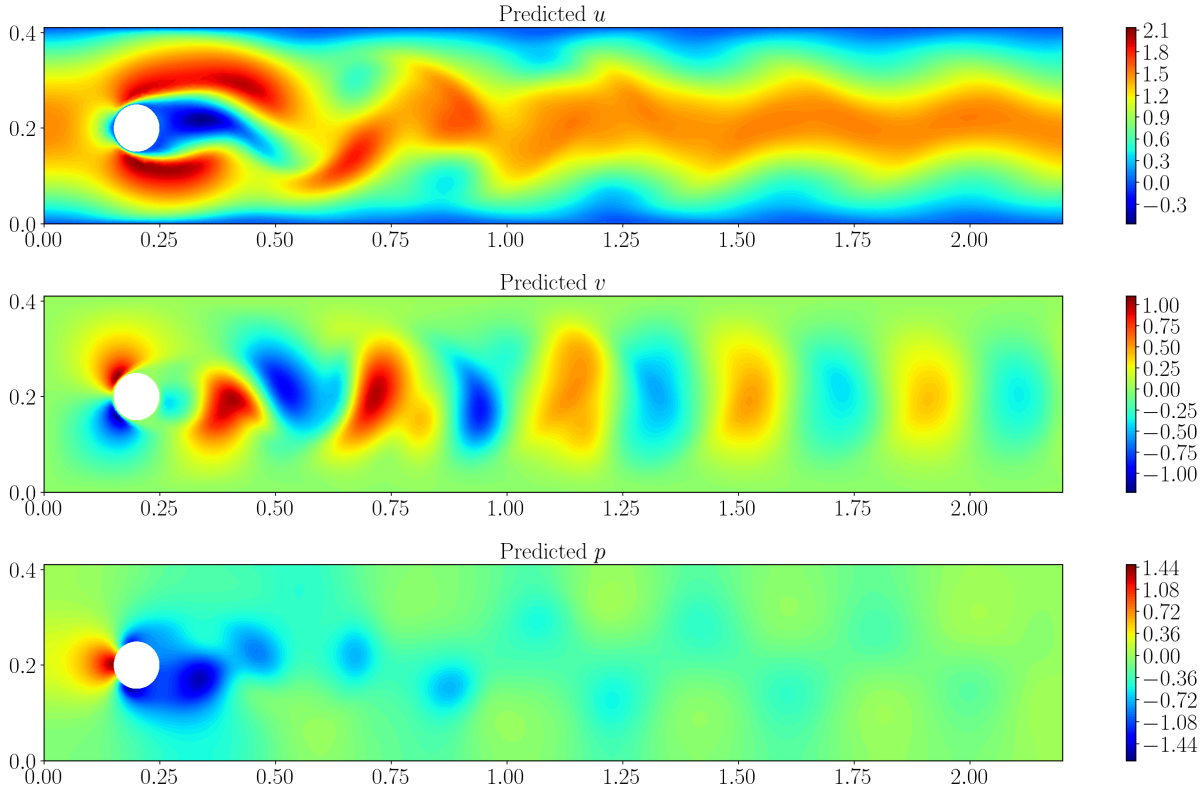

Figure 4.2: Navier-Stokes flow around a cylinder at $T = 10$: Predicted velocity field (top) and pressure (bottom). The visualisations depict the nuanced dynamics and vortex shedding phenomena [18].

The predictions of the PINN from [18] have undeniably given relevancy to the application and research into PINNs with many open doors awaiting further research opportunities.

Simulation of Incompressible Navier–Stokes Flow in a Torus Using PINNs

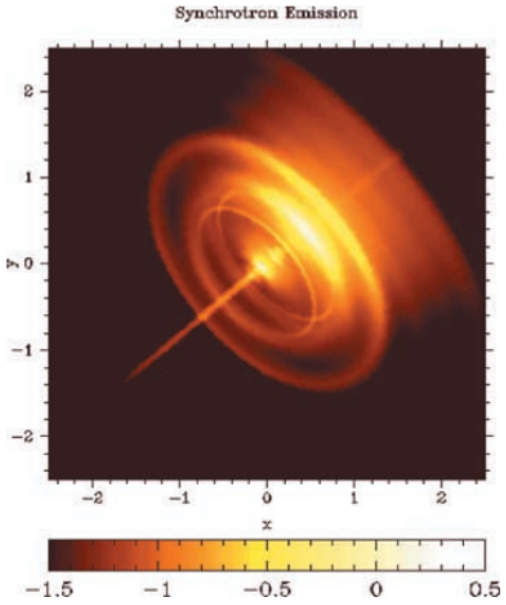

Figure 4.3: Example of a Torus shape: Synchrotron X-ray image for f peculiar jet-torus structure in the Crab nebula

A torus is a donut-shaped surface, visible in 4.3, which presents distinct challenges and opportunities for fluid dynamics studies. The curvature of a torus influence how fluid moves and behaves, which is very important for understanding complex flow dynamics such as turbulence and vortex generation. This field also explores the behaviors of incompressible flows, where the fluid density remains constant despite pressure changes.

Yet again we have practical applications in engineering and physics, where toroidal structures are common, such as in the design of nuclear fusion reactors, development of cyclones and even seen in nebula.

This case study focuses on simulating incompressible Navier–Stokes flow within a toroidal domain from the same paper as the previous example from [18].

The governing equations are given by:

$$\omega_t + \mathbf{u} \cdot \nabla \omega = \frac{1}{\mathrm{Re}} \Delta \omega,$$
$$\nabla \cdot \mathbf{u} = 0,$$

where $\mathbf{u} = (u, v)$ represents the flow velocity field, $\omega = \nabla \times \mathbf{u}$ denotes the vorticity, and Re is the Reynolds number. The domain is defined as $\Omega = [0, 2\pi]^2$, and $\mathrm{Re} = 100$.

PINN Simulation and Results:

The simulation uses a PINN approach to predict the vorticity, the tendency of a fluid to rotate, up to $T = 10$.

The temporal domain is divided into five intervals, and a time-marching strategy is used. Each interval uses a PINN model with a Multilayer Perceptron (MLP) architecture comprising four hidden

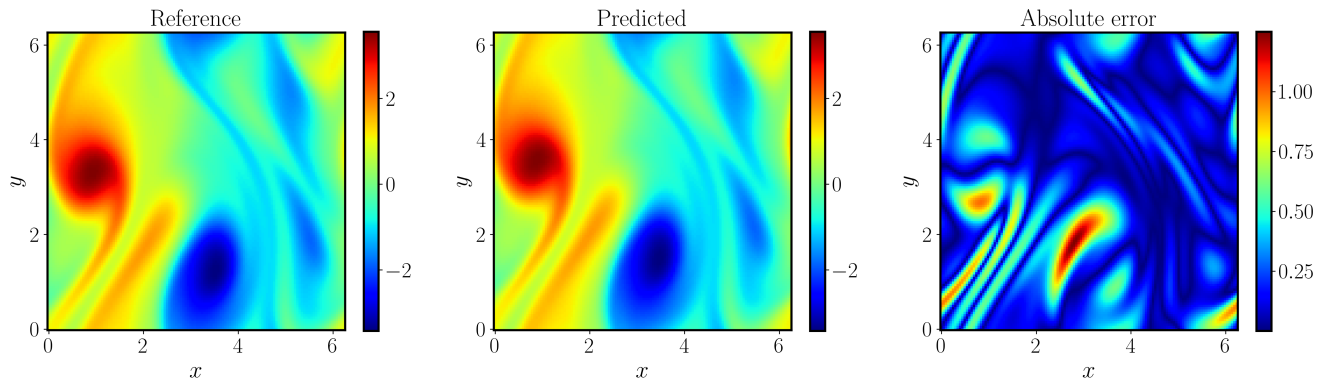

Figure 4.4: Navier-Stokes flow in a torus: Comparison of the best prediction against the reference solution at the last time step. The animation is provided in https://github.com/PredictiveIntelligenceLab/jaxpi. [18]

layers, each with 256 neurons and Tanh activations. Training is conducted for 10^5 iterations of gradient descent using the Adam optimiser.

The results, summarised in the above figure 4.4, illustrate a visual comparison between the reference and predicted vorticity fields at $T = 10$. Although a slight misalignment is observed, highlighted by the Absolute error plot, the model prediction closely matches the corresponding numerical estimations. This demonstrates the capability of PINNs to effectively simulate vortical fluid flows within a toroidal domain, emphasising their potential in capturing complex flow dynamics.

In all, the study successfully demonstrates PINNs in simulating incompressible Navier–Stokes flow within a torus. By leveraging the velocity-vorticity formulation of the equations and using an optimised training regime, PINNs accurately predict the temporal evolution of vorticity, achieving close agreement with numerical solutions. This highlights the promise of PINNs as a valuable tool for simulating complex fluid dynamics problems, offering computational efficiency and accuracy. Predicting such fluid dynamics without the use of PINNs can be very challenging due to the complex interactions and non linear behavior involved, where researchers may even decide to lean on cokriging or resort to empirical correlations. As mentioned, these correlations may not capture the full range of complex behaviour however and have their own assumptions as is covered later.

4.3 The Kuramoto–Sivashinsky Equations

The Kuramoto–Sivashinsky equations belong to a subset of non linear dynamics and fluid mechanics that have applications and in modelling chaotic dynamics. These partial differential equations have left a mark on various fields, from physics to engineering through to fluid dynamics.

Originally, these equations had their use in flame propagation but now the equations have found applications in diverse areas such as pattern formation, surface roughening, and turbulent flows. They capture the intricate interplay of non linear effects of diffusion and more, making them valuable tools for understanding complex behaviour in fluid dynamics.

The Kuramoto–Sivashinsky equations are given by:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} + \frac{\partial^4 u}{\partial x^4} = 0 \tag{4.8}$$

where $u = u(x, t)$ represents the velocity field as a function of space x and time t.

Once again, [18] explored the application of PINNs in chaotic dynamics.

The authors demonstrate the challenges of training PINNs for long-term use of this PDE system without a time-marching strategy. A time-marching strategy, also known as time-stepping, is an approach that is used to solve time-dependent PDEs or ODEs over a specified time interval. Instead of solving the entire problem at once, the solution is computed incrementally over discrete time steps.

Wang showed that as the final simulation time T increases, the relative L^2 error of the PINN solution drastically increases, leading to a failure in capturing the PDE solution accurately. This highlights the need of applying time-marching to improve the accuracy of predictions, although it comes with a higher computational cost.

Furthermore, the authors conduct a study to assess how various components of their proposed method and training process affect the outcomes. They find that all proposed components contribute positively to the overall model performance, with the use of a modified Multilayer Perceptron (MLP) significantly enhancing predictive accuracy. The predicted solution obtained from their best model is in good agreement with the ground truth, although some discrepancies are observed near $t = 1$ due to error accumulation and the chaotic nature of the system.

Overall, once again this work sheds light on the challenges and opportunities of using PINNs for simulating chaotic dynamics, allowing for teaching to be taken on the importance of time-marching strategies and the impact of different model on predictive accuracy.

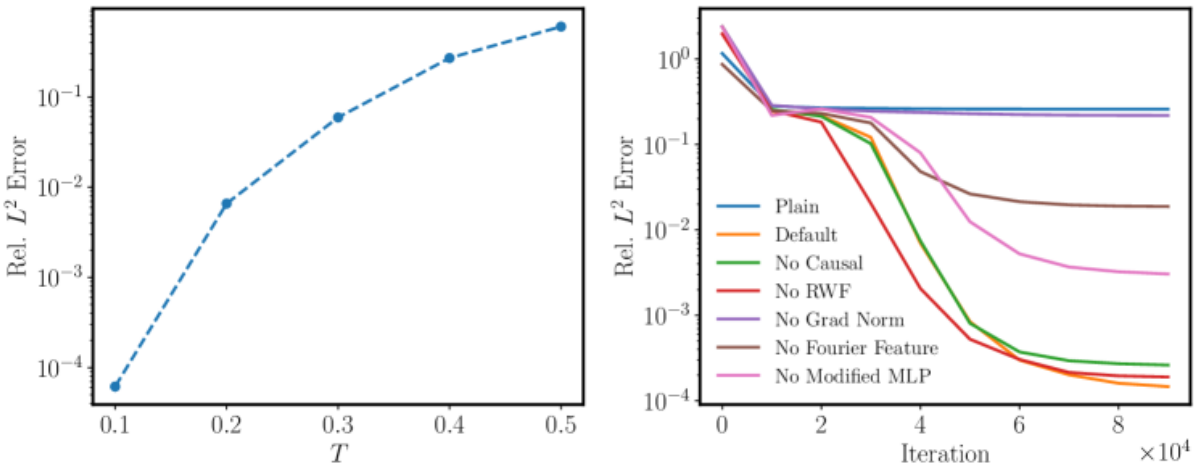

Figure 4.5: Kuramoto–Sivashinsky equation: Relative L^2 errors from one-shot PINN training for different system final time T under the same hyper-parameter setting. [18]

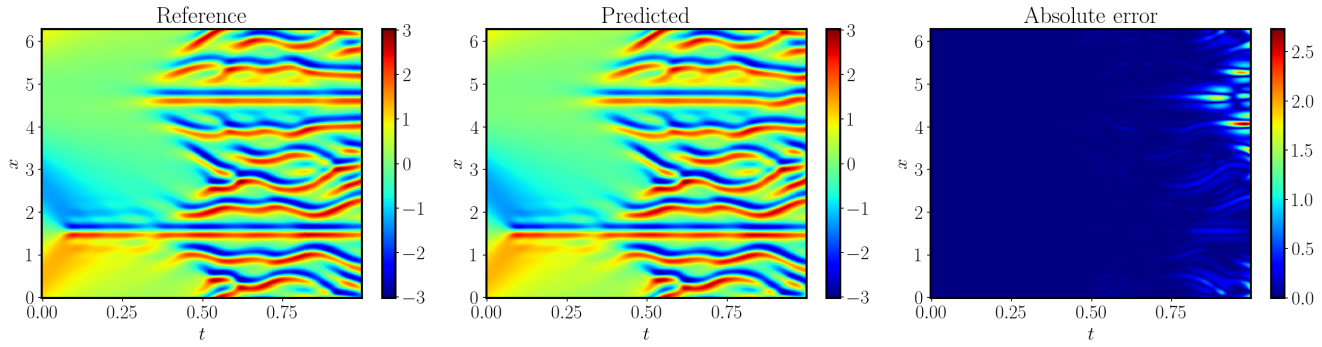

Figure 4.6: Kuramoto–Sivashinsky equation: Comparison of the best prediction against the reference solution. The relative L^2 error of the spatial-temporal predicted solution is 1.61×10^{-1}.[18]

Chapter 5

Further Discussion and Conclusions on the Implications of PINNs

In this chapter we will highlight PINNs advantages, disadvantages and weigh up PINNs against their closest competitors such as traditional interpolants like splines, highlighting their differences and discussing when each method is preferable.

5.1 Advanced Interpolation Methods

Interpolants such as splines are tools in modeling especially in developing predictive models for examples like soil moisture levels, crucial for landslide early warning systems. Splines, especially cubic splines, are used due to their ability to model data with high accuracy through piecewise polynomial functions. These interpolants are very handy in three-dimensional (3D) spatial modeling, where they help capture complex terrain geometries and variations in soil properties over large areas.

Three-Dimensional Applications of Splines

In 3D environmental modeling, cubic splines allow for the smooth interpolation of geographical data points and also allowing for the precise prediction of environmental changes across varying elevations and depths. This ability is critical in constructing reliable soil moisture forecasting models, as shown by the current author in [15].

Integration of Cokriging and Other Advanced Methods

Beyond traditional spline methods, Cokriging offers a sensible statistical approach that incorporates multiple correlated variables into the interpolation process, enhancing the prediction accuracy. Cokriging is very beneficial in scenarios where auxiliary information, such as satellite imagery or topographic data, can augment soil moisture measurements. This is akin to the Data fidelity term and physics informed term in the loss function of PINNs seen in equation 3.3.

Cokriging and other advanced interpolation methods like and Kriging are pivotal in fine-tuning the spatial analysis, enabling more accurate hazard assessments and decision-making in early warning systems. These techniques handle the variability and uncertainty inherent in environmental data more effectively, offering robust frameworks for predicting critical conditions that might lead to landslides.

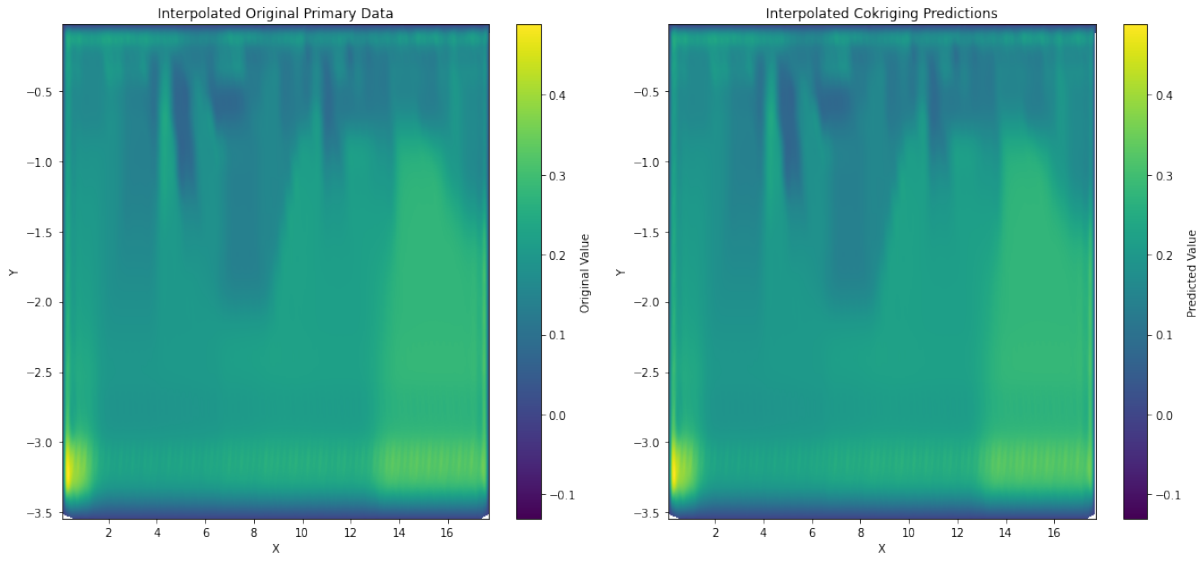

Figure 5.1: True Values of Soil Moisture and Cokriging Interpolated Predictions of Soil Moisture for Early Warning Threat Detection Models

Comparison with Physics-Informed Neural Networks

The use of traditional interpolation methods with cutting-edge techniques like PINNs seems ripe for the picking for future research and advancements in models. This is primarily due to PINNs being able to combine the flexibility of NNs with the rigidity of physical laws, using these laws directly into the learning algorithms. This synergy enables models to leverage sparse data effectively while adhering to the physical dynamics of environmental processes, crucial for applications in geophysics, for example.

5.2 Physics-Informed Neural Networks and Traditional Numerical Methods

Traditional numerical methods for solving ordinary differential equations, such as Euler's method, Runge-Kutta methods, and spline-based interpolants, have long been used due to their simplicity and effectiveness in various engineering and physics applications. Among these are the spline interpolators we have just seen more of, particulaly cubic splines, are widely used for their ability to provide smooth approximations through discrete data points. Cubic splines minimise discontinuities in the first and second derivatives, making them suitable for modeling where smoothness is essential.

Advantages of PINNs: PINNs excel where data is scarce or expensive to obtain. They leverage known physical laws as constraints, thereby reducing the solution space and potentially increasing the accuracy with less data. Furthermore, PINNs can simultaneously solve forward and inverse problems, making them incredibly versatile for research purposes where underlying physics is complex and not wholly understood. Finally once the model is trained and optimised, the model is very efficient and fast to predict future trends, ensuring timely responses to new predictions.

Limitations of PINNs: However, the complexity of PINNs comes with computational challenges when compared to rather simple approximates such as splines. The training process is intensive as it

often requires solving ODEs/PDEs iteratively during network training, which can significantly increase computational time and resource demands. Additionally, the accuracy of a PINN is highly dependent on the correct formulation of the physical laws involved, which can be a limiting factor if the laws are not fully characterised or if the scenario involves uncertainty in parameters. Furthermore, the PINN training and optimisation phase in crucial to the predictive performance of the model leaving small room for error with incorrect choices of architecture, loss function or hyperparameter turning - which can be time consuming.

Strengths of Traditional Numerical Approximators: Traditional numerical methods for solving ordinary differential equations are generally less computationally intensive and do not require the extensive infrastructure that PINNs do. Traditional methods like spline interpolation are very effective in applications where high degrees of smoothness are required, and they are relatively straightforward to implement. Moreover, these methods have a long history of theoretical development and practical application, providing a solid framework for error analysis and stability considerations.

We can improve the results of these methods by making the grid finer or by using more advanced aproximators with less assumptions. These methods have errors that we can usually measure and understand easily because of well-known scientific rules. This is really important in fields like engineering and scientific computing where making sure things are safe and reliable is key and we need to clearly understand how accurate their calculation models are.

Also, these methods don't use as much computer power or memory, which makes them good for real-time projects and for running on less powerful computers. This is helpful when using complex models isn't possible. This feature of traditional calculation methods is especially important in areas when you need to make quick decisions or manage control systems right away.

5.3 Choosing the Right Approach:

The choice between using PINNs and traditional numerical approximators often relies on several critical factors including the specific requirements of the problem, the availability and quality of data, and the necessary precision and reliability of the results. For scenarios where physical laws are well-understood and data is abundant, traditional numerical methods might be preferred for their simplicity, established reliability, and ease of validation. These methods are very effective in controlled environments where the outcomes must be highly predictable and consistent.

On the other hand, in research areas where experimental data is scarce, costly to acquire, or the physical processes involved are complex and not fully understood, PINNs offer a promising alternative. They are capable of using both computational modeling and observational data to uncover new trends and predict behaviour where traditional models may fail or be too rigid to apply. This makes PINNs very valuable in cutting-edge research fields such as complex fluid dynamics, biological processes, and materials science where the underlying physics may be partially known but not sufficiently to model with conventional methods alone.

PINNs can also be more adaptable to changes in the underlying system dynamics without requiring complete reconfiguration or redevelopment of the model. This adaptability is due to their data-driven nature, which can integrate new observations to refine and update the model continually. This is very beneficial in real-world applications where conditions frequently change or in experimental settings where initial models are based on hypothesised physics that may evolve with ongoing research.

However, the computational demands of PINNs also mean that they require substantial computational resources, including specialised hardware such as GPUs, and expertise in both the domain of the application and advanced machine learning techniques. This can limit their use to settings where such resources are available and where the potential gains from their use justify the higher costs and complexity.

Ultimately, the decision to use PINNs over traditional numerical approximators should be guided by a balanced consideration of these factors. It requires a clear understanding of the limitations and potential of each approach and a strategic assessment of their suitability for the task at hand. A careful evaluation will hopefully ensure that the chosen method aligns well with the project's goals, resources available, and desired outcome.

Appendix A

Back Propagation

We have touched on the learning process , but the final part to mention is the procedure of which the weights and biases are updated. Back Propagation is a very important method used in training NNs, especially in deep learning models. It involves two main phases: the forward pass and the backward pass. During the forward pass, input data is passed through the network to generate the output. In the backward pass, the difference between the predicted output and the actual output is computed to determine the error. This error is very important for calculating the gradients of the loss function with respect to each weight by using the chain rule of calculus.

The weights are then updated using the formula:

$$w_{new} = w_{old} - \eta \frac{\partial \text{Loss}}{\partial w}$$

where η is the learning rate, w_{old} represents the current weights, $\frac{\partial \text{Loss}}{\partial w}$ is the gradient of the loss function with respect to the weights, and w_{new} are the updated weights. This updating process is repeated iteratively until the network meets performance goals or a predefined number of training cycles (epochs) is reached.

Example of Back Propagation

Consider a NN with the following structure:

- Input layer: 2 neurons (x_1, x_2)
- Hidden layer: 2 neurons, using sigmoid activation function σ
- Output layer: 1 neuron, using sigmoid activation function σ

Suppose the weights from the input to the hidden layer are $w_{11}, w_{12}, w_{21}, w_{22}$, and the weights from the hidden layer to the output are w_{31}, w_{32}. The biases are b_1 and b_2 for the hidden layer, and b_3 for the output layer.

The forward pass computes the output of the network for given inputs x_1 and x_2. The activation of the hidden layer neurons are:

$$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + b_1)$$

$$h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + b_2)$$

The output of the network is:

$$y = \sigma(w_{31}h_1 + w_{32}h_2 + b_3)$$

Assume a simple loss function, the mean squared error (MSE), to measure the error between the network's prediction y and the actual target output t:

$$L = \frac{1}{2}(t - y)^2$$

During the backward pass, the gradients of the loss function with respect to the weights are computed to update the weights. Using the chain rule, the gradient of the loss with respect to w_{31} is:

$$\frac{\partial L}{\partial w_{31}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_{31}} = -(t - y) \cdot y(1 - y) \cdot h_1$$

This gradient tells how much the weight w_{31} needs to change to minimize the loss. The weights are updated as:

$$w_{31}^{new} = w_{31} - \eta \frac{\partial L}{\partial w_{31}}$$

This process repeats for each weight in the network, moving the weights in the direction that most reduces the loss, iterating until the network's predictions approximate the target values or a maximum number of iterations is reached.

Bibliography

[1] H. Bahi, Z. Mahani, A. Zatni, and S. Saoud. "A robust system for printed and handwritten character recognition of images obtained by camera phone". In: *WSEAS Transactions on Signal Processing* 11 (2015), pp. 9–22.

[2] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. "Physics-informed neural networks (PINNs) for fluid mechanics: A review". In: *Acta Mechanica Sinica* 37.12 (2021), pp. 1727–1738.

[3] M. Dissanayake and N. Phan-Thien. "Neural-network-based approximations for solving partial differential equations". In: *communications in Numerical Methods in Engineering* 10.3 (1994), pp. 195–201.

[4] S. Eskiizmirliler, K. Günel, and R. Polat. "On the solution of the black–scholes equation using feed-forward neural networks". In: *Computational Economics* 58 (2021), pp. 915–941.

[5] S. L. Heston. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". In: *The Review of Financial Studies* 6.2 (1993), pp. 327–343.

[6] J. M. Hutchinson, A. W. Lo, and T. Poggio. "A nonparametric approach to pricing and hedging derivative securities via learning networks". In: *The journal of Finance* 49.3 (1994), pp. 851–889.

[7] F. Kartal and U. Özveren. "Prediction of activation energy for combustion and pyrolysis by means of machine learning". In: *Thermal Science and Engineering Progress* 33 (2022), p. 101346.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[9] Z. C. Lipton. "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery." In: *Queue* 16.3 (2018), pp. 31–57.

[10] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. "DeepXDE: A Deep Learning Library for Solving Differential Equations". In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067. eprint: https://doi.org/10.1137/19M1274067. URL: https://doi.org/10.1137/19M1274067.

[11] M. Penwarden, H. Owhadi, and R. M. Kirby. "Kolmogorov n-Widths for Multitask Physics-Informed Machine Learning (PIML) Methods: Towards Robust Metrics". In: *arXiv preprint arXiv:2402.11126* (2024).

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

[13] S. Rout, V. Dwivedi, and B. Srinivasan. "Numerical approximation in CFD problems using physics informed machine learning". In: *arXiv preprint arXiv:2111.02987* (2021).

[14] J. Ruf and W. Wang. "Neural networks for option pricing and hedging: a literature review". In: *arXiv preprint arXiv:1911.05620* (2019).

[15] I. A. Sousa et al. “Development of a soil moisture forecasting method for a landslide early warning system (LEWS): Pilot cases in coastal regions of Brazil”. In: *Journal of South American Earth Sciences* 131 (2023), p. 104631.

[16] N. N. Taleb. *The Black Swan: The Impact of the Highly Improbable*. Random House, 2007.

[17] H. Wang and B. Raj. “A survey: Time travel in deep learning space: An introduction to deep learning models and how deep learning models evolved from the initial ideas”. In: *arXiv preprint arXiv:1510.04781* (2015).

[18] S. Wang, S. Sankaran, H. Wang, and P. Perdikaris. “An expert’s guide to training physics-informed neural networks”. In: *arXiv preprint arXiv:2308.08468* (2023).

[19] S. Wang, Y. Teng, and P. Perdikaris. “Understanding and mitigating gradient flow pathologies in physics-informed neural networks”. In: *SIAM Journal on Scientific Computing* 43.5 (2021), A3055–A3081.

[20] X. Wang, J. Li, and J. Li. “A Deep Learning Based Numerical PDE Method for Option Pricing”. In: *Computational Economics* 62.1 (2023), pp. 149–164. DOI: 10.1007/s10614-022-10279-x.

[21] X. Wang, J. Li, and J. Li. “A deep learning based numerical PDE method for option pricing”. In: *Computational economics* 62.1 (2023), pp. 149–164.

[22] Q. Zheng, S. Yuan, C. Chen, Y. Wang, J. Liu, and J. Guan. “Breaking the CAPTCHA Using Machine Learning in 0.05 Seconds”. In: *IEEE Access* 6 (2018), pp. 45407–45417.