uc3m | Universidad Carlos III de Madrid

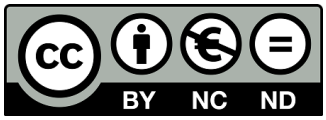
Master's Degree in Applied and Computational Mathematics
2024-2025

Master Thesis

Physics Informed Neural Networks for Fractional and Partial Differential Equations

David Cano Rosillo

Supervisor
Dr. Pedro González Rodríguez
Madrid, June 2025



Abstract

This thesis explores the use of Physics-Informed Neural Networks (PINNs) for solving both partial differential equations (PDEs) and fractional differential equations (FDEs). We address the challenge of low-frequency bias in PINNs using the Neural Tangent Kernel (NTK) framework and benchmark various mitigation strategies, highlighting the effectiveness of the PirateNet architecture. We then apply PINNs to PDEs such as Burgers' and Navier-Stokes equations, achieving promising results. Additionally, we adapt the PINN framework for Fractional Differential Equations (FDEs), proposing key modifications that accelerate convergence.

Dedication

To my supervisor, Dr. Pedro González Rodríguez, for his ongoing support and commitment.

Table of contents

Chapter 1

Introduction

This thesis is about Physics Informed Neural Networks (PINNs), which is an application of neural networks to approximating Partial Differential Equations (PDEs). We will define neural networks and the PINN framework in depth in chapters 2 and 4 respectively. However, for the sake of completeness, we will explain the basics of both of these concepts here. A deeper explanation will follow in subsequent chapters.

A neural network is a function, parametrized by a set of parameters θ that control its behavior. By modifying the set of parameters θ the neural network is able to approximate most functions. The neural network training refers to the optimization of these parameters, by repeatedly quantifying how wrong the output is with a loss function and updating parameters via gradient descent.

Partial Differential Equations (PDEs) are equations in which the solution is an unknown function. The equation gives a condition that the function must fulfill, for PDEs it involves the derivatives of the unknown function. They are widely used in physics and engineering, where the unknown function might be a physical quantity, such as temperature evolving over time. A PDE problem is the combination of the equation with a set of initial data, for example the initial condition and/or values at the boundary, which will determine the unknown function.

The PINN framework gives a procedure for training a neural network to approximate the solution of a given PDE problem. It essentially introduces two loss functions, which are often called the residual loss and the data loss. The residual loss quantifies how much the neural network is not fulfilling the equation of the PDE. The data loss quantifies how much the initial and/or boundary conditions are not being fulfilled. By minimizing the sum of these two loss functions the neural network is able to (hopefully) find an approximated solution to the PDE problem.

Finally, we will deal with Fractional Differential Equations (FDEs). They are similar to PDEs, but they involve fractional derivatives, which, as a matter of fact, are integro-differential operators. This makes them more difficult to solve than standard PDEs.

This thesis aims to

- Understand and tackle the low-frequency bias phenomena, in which neural networks learn first the low-frequency components of a function. As PDE solutions often have

high-frequency components, it is crucial to ensure the neural network is capable of learning those components. (**Chapter 2 and 3**)

- Apply PINNs to challenging PDE problems, like Burgers' equation, the Helmholtz equation and Navier-Stokes in a torus. Along the way we will apply state of the art techniques such as PirateNet's boundary aware initialization. **(Chapter 4)**
- Adapt PINNs to tackle two FDEs, the Relaxation-Oscillation equation and the time fractional diffusion equation. For that purpose we deal with practical considerations such as scaling training to multiple GPUs and (softly) enforcing causality through better sampling strategies. **(Chapter 5)**

1.1 Thesis Structure

This thesis is organized as follows:

- **Chapter 2: Theoretical Foundations**: In this chapter we overview neural networks. We first review the functions they implement, and how they fit data using gradient descent. Finally, we explore the Neural Tangent Kernel (NTK) framework, which is the theoretical justification of the low-frequency bias issue.
- **Chapter 3: Approximating high-frequency Functions**: Before training PINNs we seek to identify which neural network architecture works best for high-frequency functions. To do so we review the state of the art and benchmark the different architectures found in the literature.
- **Chapter 4: Physics Informed Neural Networks**: In this chapter we extensively apply PINNs to PDE problems. The selected equations are challenging, containing nonlinearities and high-frequency components, chosen to take the PINN framework as far as possible.
- **Chapter 5: Application to Fractional Differential Equations:** We finally apply the PINN framework to a more challenging type of equations, Fractional Differential Equations. We first define the fractional derivative in the Caputo sense and then explore how we can apply PINNs to these integro differential equations. Along the way we propose several techniques to increase accuracy.

Chapter 2

Theoretical Foundations

2.1 Neural Networks

We will see two examples of Neural Network architectures, the Multilayer Perceptron (MLP) and Kolgomorov Arnold Network (KAN). They are capable of approximating most functions by modifying their parameters θ. The process of modifying the parameters through gradient descent to perform best on the task given is commonly referred to as training.

2.1.1 Multilayer Perceptron

In this section we will mathematically define the multilayer perceptron. This is a neural network with more than one layer, and can be seen as the composition of layers.

First we will define the transformation on a single layer neural network. Let $x \in \mathbb{R}^n$ be our input vector, then we can define our first layer as:

$$f(x) = h = \sigma(Wx + b) \tag{2.1}$$

Where:

- W is a $\mathbb{R}^{n \times m}$ matrix.
- $b \in \mathbb{R}^m$ is a bias vector.
- σ is a nonlinear activation function.

The resulting $h \in \mathbb{R}^m$ vector is the output of our neural network layer. The parameters optimized will be W and b. We will update these parameters using the information given by the derivative of the network with respect to a loss function, if θ is a vector containing all our parameters then:

$$\theta_{t+1} = \theta_t - t_k \nabla_{\theta_t} \mathcal{L}(\theta_t) \tag{2.2}$$

Where $\mathcal{L}(\theta_t)$ is the loss function evaluated at our current parameters and t_k is our step size or learning rate. In practice the update rule of the parameters is more complicated, and often contains information from previous gradients or even second order information like the Hessian.

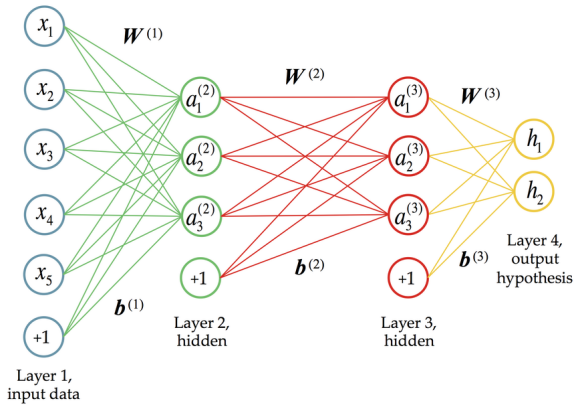


Figure 2.1: Diagram of a neural network parametrized by three weight matrices W and bias vectors b [1].

A multilayer perceptron is just the concatenation of these layers. A layer is defined as a transformation:

$$h^{(i+1)} = f^{(i)}(h^{(i)}) = \sigma(W^{(i)}h^{(i)} + b^{(i)}) \tag{2.3}$$

Where h^i is the output from the previous layer, or x when $i = 1$. Then a N layer network is the composition of these layers, often called a multilayer perceptron:

$$MLP(x) = f^N \circ f^{N-1} \circ ... \circ f^1(x) \tag{2.4}$$

The neural network can be efficiently differentiated using an algorithm known as back-propagation. However, a formula for integration over an input region is not known, and one must resort to numerical methods for that aspect [2].

We will now state the Universal Approximation Theorem, which tells us we can arbitrarily approximate a continuous function on a compact set using only one hidden vector h. However, we might require that the hidden vector is very large, allowing for the use of many parameters.

Theorem 2.1 (Universal Approximation Theorem)**.** *Let $X \subset \mathbb{R}^n$ be a **compact set**, and let $f : X \to \mathbb{R}^m$ be a **continuous function**. Suppose $\sigma : \mathbb{R} \to \mathbb{R}$ is a **nonconstant, bounded, and continuous activation function**. Then, for any $\epsilon > 0$, there exists a feedforward neural network of the form:*

$$F(x) = A\sigma(Wx + b) + c, \quad x \in X \tag{2.5}$$

where:

- $W \in \mathbb{R}^{k \times n}$ *is the weight matrix of the hidden layer (with k neurons),*
- $b \in \mathbb{R}^k$ *is the bias vector,*
- $\sigma(Wx + b)$ *is applied element-wise,*
- $A \in \mathbb{R}^{m \times k}$ *is the output layer weight matrix,*
- $c \in \mathbb{R}^m$ *is the output bias vector,*

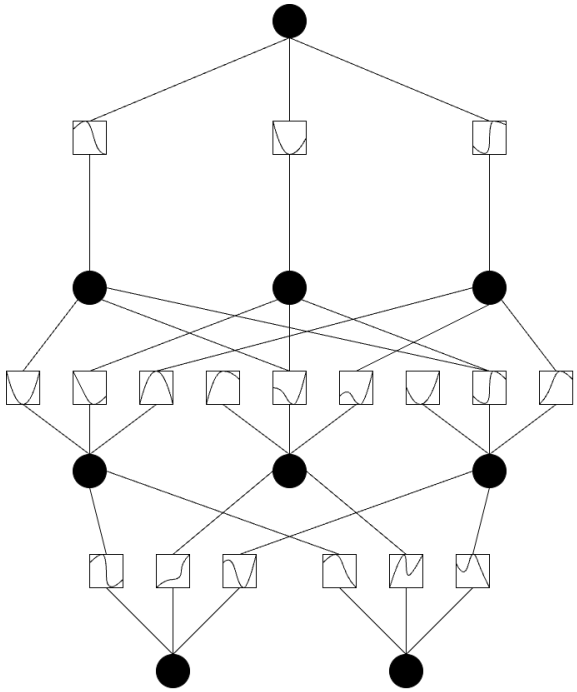


Figure 2.2: The KAN architecture for a 2 layer network. Connections are learnable activation functions, which are then combined via summation on a node.

such that:

$$\sup_{x \in X} \|f(x) - F(x)\| < \epsilon. \tag{2.6}$$

Although in principle any function could be learned by a sufficiently wide one layer network, deep layers turn out to work best in practice. Notice that the theorem applies also to deep networks, as one could concatenate sufficiently wide layers after one another, or have the later layers act as identities.

2.1.2 Kolmogorov Arnold Networks

Kolmogorov Arnold Networks (KANs) are an alternative to standard neural network architectures that has seen a surge in popularity. In the original paper [3] it is claimed that KANs are more parameter efficient, that is they obtain the same accuracy with fewer parameters than multilayer perceptrons.

We will first define one layer KANs and then use composition to define multilayer KANs. We will first define function matrices as:

$$\Phi = \begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{22} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix}$$

Where ϕ_{ij} are univariate functions. We also define multiplication by a function matrix as:

$$h = \Phi x$$

$$\begin{bmatrix} \sum_{i=1}^{n} \phi_{1i}(x_i) \\ \sum_{i=1}^{n} \phi_{2i}(x_i) \\ \vdots \\ \sum_{i=1}^{n} \phi_{mi}(x_i) \end{bmatrix} = \underbrace{\begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{22} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix}}_{\Phi} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where $x \in \mathbb{R}^n$, $h \in \mathbb{R}^m$ and $\Phi \in \mathbb{F}^{n \times m}$ with $\mathbb{F}$ denoting univariate functions.

Using this notation we can define a KAN layer as simply applying a function matrix Φ, to the input vector x:

$$g(x) = h = \Phi x \tag{2.7}$$

each entry $\phi_{ij} \in \Phi$ will be a parametrized function, for example B-splines, which will then be trained using gradient descent. A multilayer KAN network is then just a composition of function matrices:

$$\text{KAN}(x) = \Phi_n \circ \Phi_{n-1} \circ \ldots \circ \Phi_1(x) \tag{2.8}$$

Each Φ_i will have its own parametric univariate functions and will be optimized jointly using gradient descent.

One advantage of KAN's is that once they are trained one can inspect the learned univariate functions. This is more interpretable than neural networks as the function depends only on one input. However using multiple KAN layers may add too much complexity even if each component is univariate.

KANs owe their name to the following representation theorem:

Theorem 2.2 (Kolmogorov-Arnold Representation Theorem)**.** *Let $f : [0,1]^n \to \mathbb{R}$ be a* ***multivariate continuous function****, then f can be written as the finite composition of univariate functions and the sum operation. That is:*

$$f(x) = f(x_1, x_2, ..., x_n) = \sum_{q=0}^{2n} \Phi_q(\sum_{p=0}^{n} \phi_{q,p}(x_p)) \tag{2.9}$$

where $\Phi_q : \mathbb{R} \to \mathbb{R}$ and $\phi_{q,p} : [0,1] \to \mathbb{R}$.

The Kolmogorov-Arnold Representation Theorem is analogous to the Universal Approximation Theorem for multilayer perceptrons.

Notice that the theorem does not specify which 1D functions to use. For some multivariate functions one requires that the 1D functions be fractal, however these are not required in practice.

2.2 Loss functions

The functions that neural networks seek to minimize during training are called loss functions. They aim to quantify the error in the task that we want the neural network to

perform. For example, if we seek to fit some data we would use the mean squared error:

$$\mathcal{L}_{MSE}(\theta) = \sum_{x,y \in D} |\hat{u}(x;\theta) - y|^2 \tag{2.10}$$

Here we are denoting by x the input to the function, and by y the output we desire for each input x. Our dataset D consists of pairs (x, y). Our neural network is parametrized by θ, and is denoted as $\hat{u}$. Notice that the loss function is only dependent on θ. The process of optimizing θ, via some gradient descent related method, is called training in the literature.

It has also been found beneficial to add terms to the loss function that penalize large weights. For example one might modify the previous loss function by adding a term $||\theta||^d$:

$$\mathcal{L}_{MSE}(\theta) = ||\theta||^d + \sum_{x,y \in D} |\hat{u}(x;\theta) - y|^2 \tag{2.11}$$

where d is some positive integer. Usually d is picked with value 1 or 2. In those cases the loss term is called $L1$ and $L2$ regularization respectively.

Regularization is a technique used to improve performance outside the data seen during training. For example one might only care about reducing the mean squared error on unseen data, but find that adding the $L2$ term to the training loss improves performance on test (unseen) data.

2.3 Optimization and loss landscapes

Having discussed the parametric functions defined by Multilayer Perceptrons and Kolgomorov Arnold Networks, we now move to discussing how the parameters are optimized during training to make these functions fulfill a predefined purpose. We will first explore how gradient descent is used to optimize the performance of neural networks. We then move to loss landscape visualization, a technique that will help us visualize the landscape around the neural network parameters.

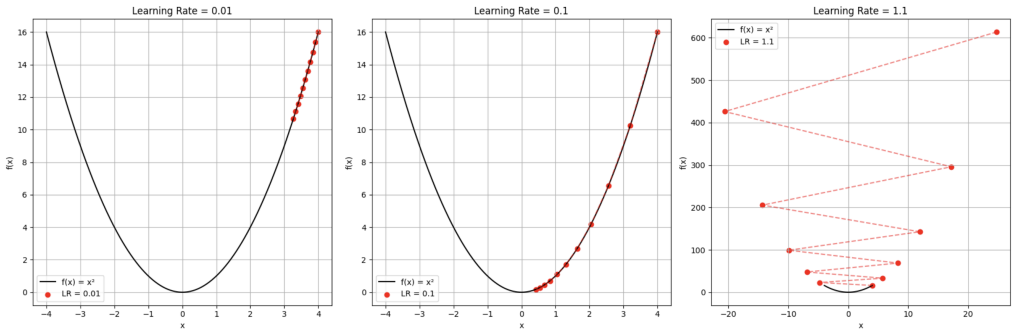


Figure 2.3: The effect of different learning rates. On the left the smallest learning rate convergence is very slow. On the right the learning rate is too high, and the algorithm will not converge. Notice the different scale of the figure on the right.

Throughout this chapter we have mentioned how both KANs and MLPs (Multilayer perceptrons) minimize their loss using optimization techniques on their parameters θ. We know that the loss functions $\mathcal{L}(\theta)$ we will be optimizing will be C^1.

However, in principle we know nothing about the structure of this loss functions. It would be difficult to even formalize the structure of a loss function associated to classification of images, let alone prove useful results about it. This makes the techniques used highly heuristic, only validated by their empirical success.

To illustrate this problem consider the optimization of a KAN or MLP network using gradient descent:

$$\theta_{t+1} = \theta_t - t_k \nabla_{\theta_t} \mathcal{L}(\theta_t)$$

The learning rate coefficient t_k plays a big role in whether we converge to a local minimum. For large values, we will not converge, while for too small values, convergence is extremely slow. So already for one of the simplest optimization algorithm we see heuristics appearing.

Furthermore, evaluating the loss function $\mathcal{L}(\theta)$ is costly, as it often depends on the entire dataset. We might use only a portion of our dataset, sampled at random at each optimization step. This is known as Stochastic Gradient Descent (SGD).

Visualizing the loss values around a set of parameters θ is a good idea to get a sense of how difficult this optimization is in practice, and why we can't rely on the landscape being convex. On figure 2.3 we chose a very simple example where the dimension of our parameters θ was one. In actual applications θ might contain thousands of different parameters and visualization is impossible for more than 2 dimensions. However [4] introduced a technique for visualizing the loss landscape by projecting to two dimensions. Given a trained (or untrained) set of parameters $\theta \in \mathbb{R}^n$ we consider perturbations around it:

$$\tilde{\theta} = \theta + \alpha d_1 + \beta d_2 \tag{2.12}$$

where $d_1, d_2 \in \mathbb{R}^n$ are two random directions and $\alpha, \beta \in \mathbb{R}$ are the magnitude of the perturbation in each direction. It is also suitable to normalize the random directions by the magnitude of each parameter:

$$d_1, d_2 \sim \frac{\mathcal{N}(0, I)}{|\theta|}$$

We can then visualize a slice of the loss landscape by plotting the dependence of the loss $\mathcal{L}(\tilde{\theta})$ on the coefficients α, β. Although this is not the full loss landscape we see an interesting behavior appear. The following are loss landscape visualizations of the neural networks we trained for the chapter on interpolation. We repeat the computations along 3 pairs of random directions to show that the results are not cherry-picked. We provide three examples, of the following neural networks parameters:

1. Successfully trained 2.4: We provide a visualization of a neural network that trained successfully. The terrain around the found minima is relatively convex and flat, making it robust to perturbations introduced by noise in the data.
2. Untrained 2.5: We also show the loss landscape of an untrained network. Indeed, when the neural network is untrained we don't see any structure in the two dimensions plotted.
3. No generalization 2.6: Our last example is a neural network which trained, but got stuck in a bad minima. The minima is characterized by a small basin of attraction, which makes it sensible to perturbations in the data.

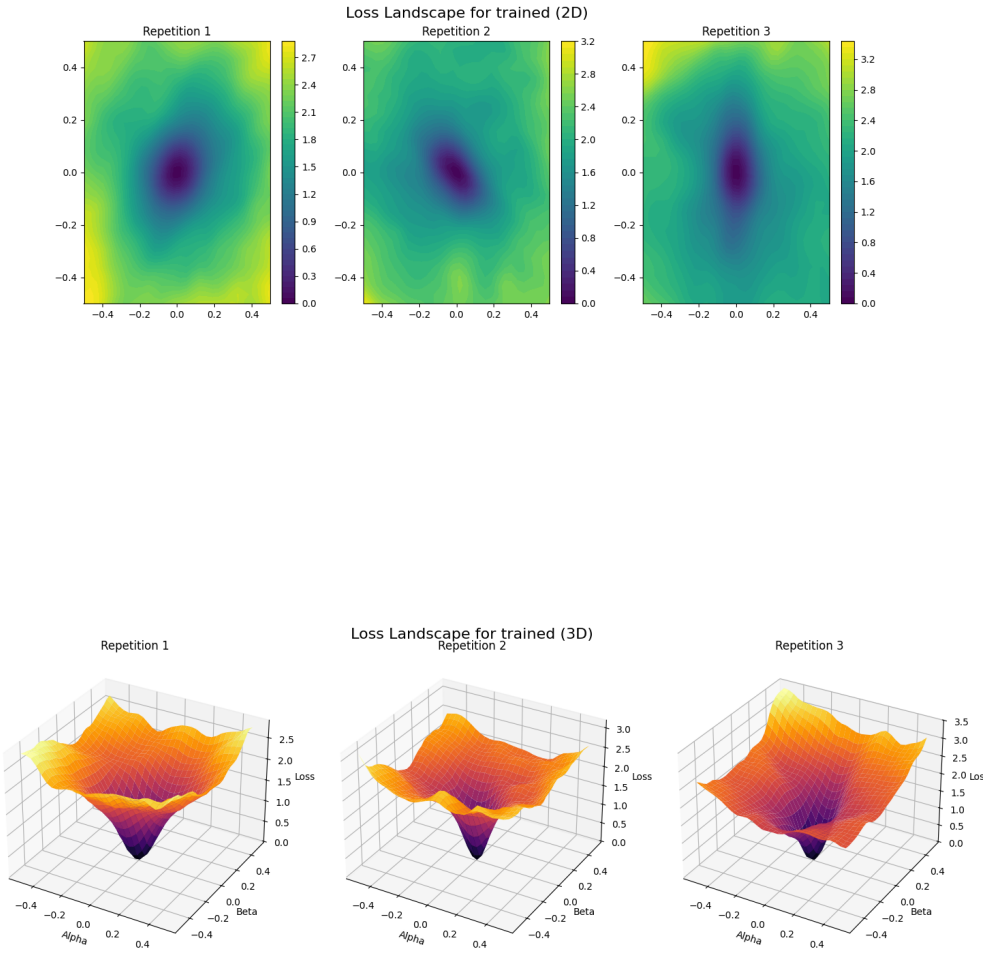
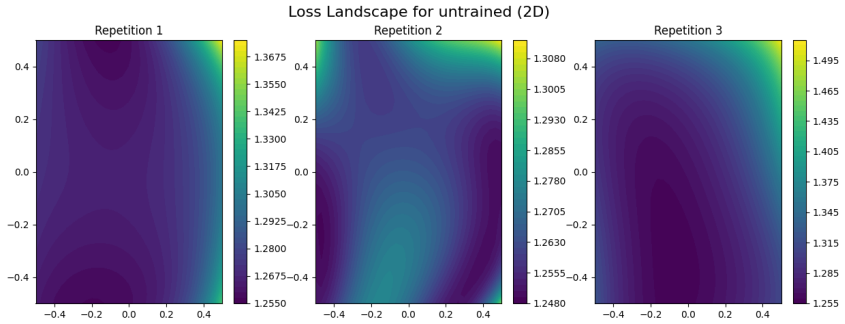


Figure 2.4: Heatmap and 3d plot of projected loss landscape visualization around **trained parameters** of a neural network. We observe local minima in these 2 dimensions, with a smooth terrain around.



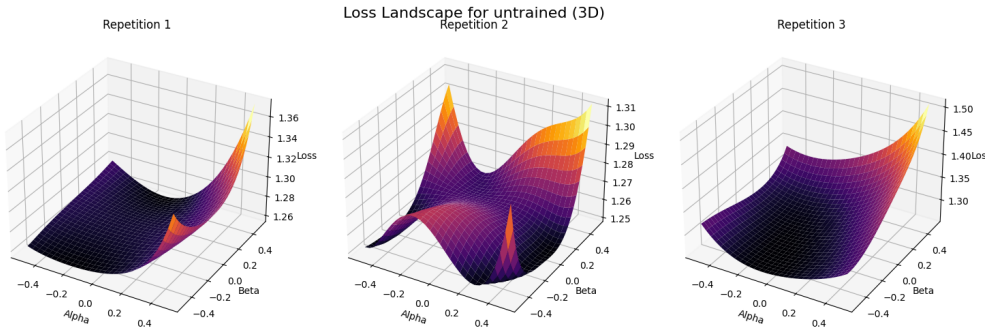


Figure 2.5: Heatmap and 3d plot of projected loss landscape visualization around **initialization parameters** of a neural network. There is no interesting structure, and an overall flatness.

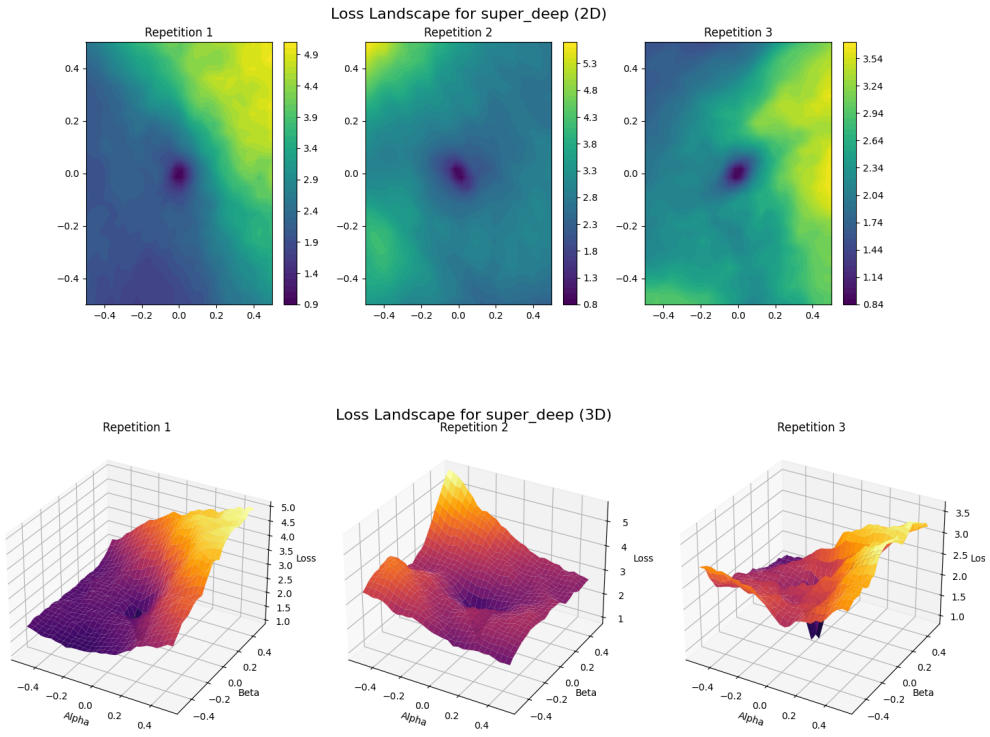


Figure 2.6: Heatmap and 3d plot of projected landscapes visualization around trained parameters of very **deep network**. We see how the terrain around the final parameters is rougher and the minima's basin of attraction is smaller. This is expected for neural networks that perform poorly on unseen data.

2.4 Neural Tangent Kernel

When we try to solve a PDE problem using PINNs one often encounters that the solution of the problem is a function composed of high-frequency components. For the solution to be accurate, the neural network must be able to learn these components. In practice, if one uses the neural networks like MLP or KAN without modifications, one finds that

they do not learn the high-frequency components. This is known as the spectral bias (or low-frequency bias) of neural networks.

We will now delve into the theoretical explanation for why neural networks learn high-frequency components slower than the low-frequency ones. In later sections we will see how to modify our neural networks architecture to make them able to learn high-frequency PDEs solutions.

The Neural Tangent Kernel (NTK) provides an interesting view on neural networks training. It speaks about the asymptotic behavior of neural networks, when we consider infinitely wide layers and infinitely small learning rate.

For a dataset $\{X_{train}, Y_{train}\}$ where $X_{train} = (x_i)_{i=0}^N$ are inputs and $Y_{train} = (y_i)_{i=0}^N$ are targets, we define the Neural Tangent Kernel as:

$$\mathbf{K}_{ij} = \left\langle \frac{\partial f_\theta(x_i)}{\partial \theta}, \frac{\partial f_\theta(x_j)}{\partial \theta} \right\rangle \tag{2.13}$$

where f_θ is a multilayer perceptron instantiated with parameters θ.

Under the limiting behavior where the learning rate t_k is infinitely small, such that it becomes a τ continuous variable, the training evolution of θ parameters evolves according to the differential equation:

$$\frac{\partial f_{\theta(\tau)}(X_{train})}{\partial \tau} \approx -\mathbf{K}(f_{\theta(\tau)}(X_{train}) - Y_{train}) \tag{2.14}$$

where $\theta(\tau)$ are the parameters at time τ. This differential equation can be solved using the change of variable $g_{\theta(\tau)} = f_{\theta(\tau)} - Y_{train}$:

$$\frac{\partial g_{\theta(\tau)}(X_{train})}{\partial \tau} \approx -\mathbf{K}(g_{\theta(\tau)}(X_{train})) \tag{2.15}$$

$$g_{\theta(\tau)} \approx (f_{\theta(0)}(X_{train}) - Y_{train})e^{-\mathbf{K}\tau} \tag{2.16}$$

with $f_{\theta(0)}$ being the parameters at initialization and $e^{-\mathbf{K}\tau}$ the matrix exponential:

$$e^{-\mathbf{K}\tau} = \sum_{n=0}^{\infty} \frac{(-\mathbf{K}\tau)^n}{n!}$$

Suppose that the initialization parameters are such that $f_{\theta(\tau=0)}(X_{train}) = 0$. Then plugging our solution back to the first equation and solving for $f_{\theta(\tau)}$ we find:

$$f_{\theta(\tau)}(X_{train}) \approx (I - e^{-\mathbf{K}\tau})Y_{train} \tag{2.17}$$

Since $\mathbf{K}$ is positive semidefinite we can diagonalize it as $\mathbf{K} = Q\Lambda Q^T$. Then since Q, Q^T are orthogonal applying the property $e^{\mathbf{K}\tau} = Qe^{\Lambda}Q^T$ yields:

$$Q^T(f_{\theta(\tau)}(X_{train}) - Y_{train}) = -e^{\Lambda\tau}Q^T Y_{train} \tag{2.18}$$

which we can expand as:

$$\begin{bmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{bmatrix} (f_{\theta(\tau)}(X_{train}) - Y_{train}) = \begin{bmatrix} e^{\Lambda_1 \tau} & & & \\ & e^{\Lambda_2 \tau} & & \\ & & \ddots & \\ & & & e^{\Lambda_n \tau} \end{bmatrix} \begin{bmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{bmatrix} Y_{train} \tag{2.19}$$

Solving for the error term

$$f_{\theta(\tau)}(X_{train}) - Y_{train} = -Qe^{\Lambda\tau}Q^T Y_{train} \tag{2.20}$$

$$= -\sum_{i=1}^{n} q_i (e^{\Lambda_i \tau} q_i^T Y_{train}) \tag{2.21}$$

This is the main result of this section. It states that under the NTK conditions, the error decreases along the q_i directions, at a rate dictated by its corresponding scalar $(e^{\Lambda_i \tau} q_i^T Y_{train})$.

The network is biased to learn first along the eigendirections with higher eigenvalues. Under further assumptions, one can prove that neural networks decrease first the error corresponding to the low-frequency components of the approximated function [5].

In further chapters we will see how this spectral bias arises even when the NTK conditions are not fulfilled. This can be problematic for some of the functions we will be approximating.

Additionally, we can express the network function using the NTK [6]:

$$f_{\theta(\tau)}(x) = f_{\theta(0)}(x) + \mathbf{K}(x, X_{train})\mathbf{K}^{-1} Y_{train} \tag{2.22}$$

where $\mathbf{K}(x, X_{train})$ is a row vector computed by applying the NTK function to each (x, x_i) $x_i \in X_{train}$ pair.

We implement a numerical experiment to illustrate this result, where we use the previous formula 2.22 to approximate the output of a trained neural network 2.7. We train the neural network to approximate the function in 2.7. We plot both the true function (red), the approximated function by the NN (blue), and the predictions by the approximation of the NTK to the NN (green).

Although our network is not in NTK regime, as we have finite learning rate and finite layer widths, we see that the results hold. Mainly we observe 2.22 is a good approximation of the network, and the low-frequency bias is present when inspecting the spectra of the NTK.

We plot both the eigenvalues of the NTK and the error for the different components of the function over time 2.8. The eigenvalues of the NTK are obtained by computing the NTK matrix on the training data. The error for the different frequencies is logged over time to produce the graph in 2.8.

In both the theory holds, the high-frequency components are learned last, and likely correspond to the lower eigenvalues.

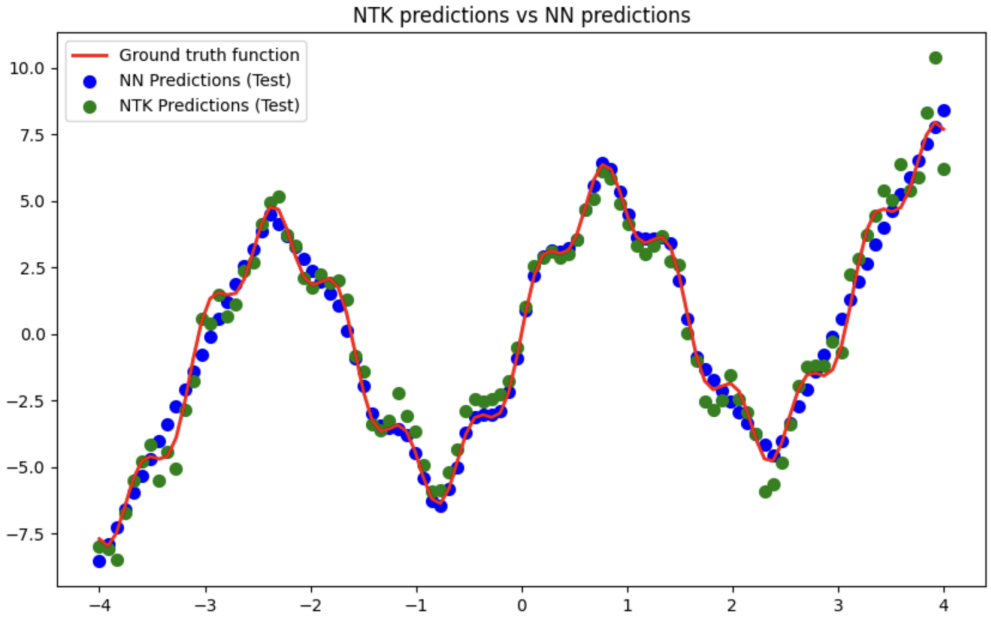
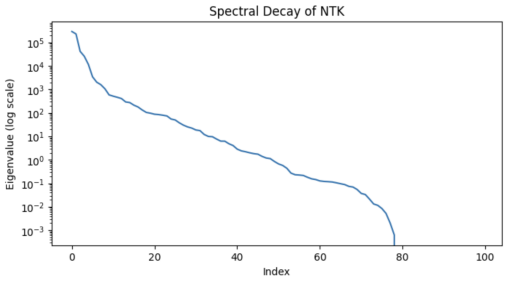
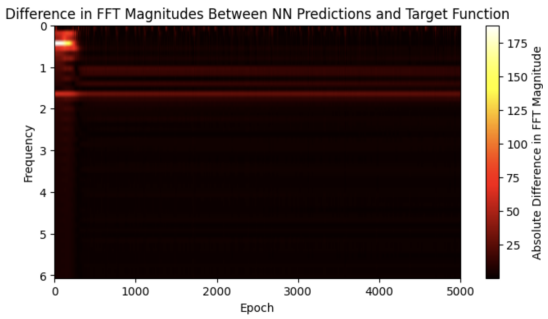


Figure 2.7: Predictions of a neural network (NN) against the Neural Tangent Kernel (NTK) approximation. The approximation holds even for small networks.



(a) Some eigenvalues are much larger than others, biasing the network to learn their corresponding eigendirections first.



(b) The low-frequency components of the approximated function are learned on the first epochs. The high-frequency components may not be approximated at all.

Figure 2.8: Low-frequency bias of neural networks, shown under inspection of the NTK spectra for a trained neural network.

Chapter 3

Approximating high-frequency Functions

Before applying neural networks to solve partial differential equations (PDEs) numerically it is worthwhile to look at which neural networks work best for high-frequency data. Since the solutions of most of the PDE problems we will tackle are composed of high-frequency components, selecting the best method is of great importance.

In order to select the best architecture, we first review the state of the art in approximating high-frequency functions with neural networks. Then we will run numerical experiments where we verify the claims of the papers reviewed, selecting the best architecture for the Physics Informed Neural Network framework we will apply in the next chapter.

3.1 Mitigating the Spectral Bias

In this section we will go over some techniques that can be used to alleviate the low-frequency bias, allowing us to fit high-frequency functions. Later on these techniques will be verified through numerical experiments.

3.1.1 Kolmogorov Arnold Networks

Although KANs also suffer from the low-frequency bias, it has been shown that they are less biased than MLPs [7]. Furthermore, they are claimed to be more parameter efficient, meaning they can model more complex functions with fewer parameters than multilayer perceptrons. Thus, we should see them perform better when comparing them with MLPs for approximating high-frequency functions.

3.1.2 Random Fourier Features

A very powerful technique is to transform the input into high-frequency functions before passing it through the rest of the network [8]. It consists of composing the transformation $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ before the rest of the network:

$$\Phi(x) = \begin{bmatrix} \cos(Bx) \\ \sin(Bx) \end{bmatrix} \tag{3.1}$$

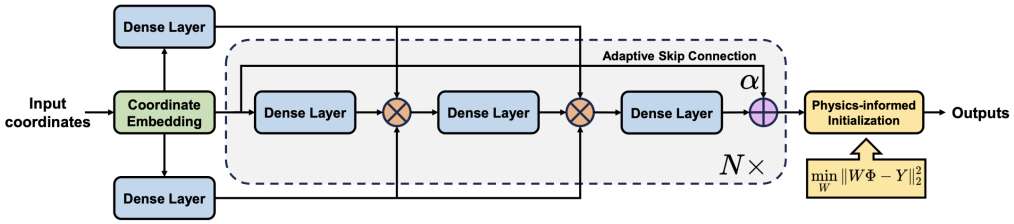


Figure 3.1: A PirateNet block. The Fourier Features are combined after each dense layer in the module. At the end they are passed through another dense layer and combined with the original input through an adaptive skip connection controlled by the α trainable parameter.

where $B \in \mathbb{R}^{d\times n}$ is a matrix, $d = \frac{m}{2}$ is a whole number (m is picked even to use cos and sin and concatenate). The entries of B are sampled randomly to create high frequencies; $B_{ij} \sim \mathcal{N}[0, \sigma^2]$, the usual value range is $\sigma \in [1, 10]$.

There are plenty of modifications, for example one can let the network modify B by adding it as another parameter during gradient descent, and can also add a bias vector b. But the remarkable idea is passing the input through high-frequency functions.

In our implementation we let the network modify B and use $\sigma = 5$.

3.1.3 PirateNets

We will also explore a neural network architecture which was successful for learning solutions to PDE equations (PINN architecture). The block (defined as a composition of various neural network layers) proposed is the following [9].

First the input is passed through the previously explained random Fourier Features layer Φ, and through two separate layers U, V:

$$U = \sigma(\mathbf{W}_1\Phi(x) + \mathbf{b}_1) \qquad V = \sigma(\mathbf{W}_2\Phi(x) + \mathbf{b}_2)$$

Throughout this section, σ is a nonlinear activation function, e.g. tanh. Each PirateNet block consists of the following transformations, here we denote by f, z_1, g, z_2 and h the intermediate states. Thus f is a intermediate value, that stores the value $\sigma(\mathbf{W}_3\mathbf{x} + \mathbf{b}_3)$ and so on:

$$\begin{aligned}
\mathbf{f} &= \sigma(\mathbf{W}_3\mathbf{x} + \mathbf{b}_3),\\
\mathbf{z}_1 &= \mathbf{f} \odot \mathbf{U} + (1 - \mathbf{f}) \odot \mathbf{V},\\
\mathbf{g} &= \sigma(\mathbf{W}_4\mathbf{z}_1 + \mathbf{b}_4),\\
\mathbf{z}_2 &= \mathbf{g} \odot \mathbf{U} + (1 - \mathbf{g}) \odot \mathbf{V},\\
\mathbf{h} &= \sigma(\mathbf{W}_5\mathbf{z}_2 + \mathbf{b}_5),\\
\mathbf{o} &= \alpha \cdot \mathbf{h} + (1 - \alpha) \cdot \mathbf{x}.
\end{aligned}$$

where $\odot$ denotes element-wise multiplication and α is a trainable parameter.

The adaptive skip connection controlled by α lets gradient information flow to previous layers directly, improving convergence. Furthermore, α is initialized to 0, which makes the network behave linearly at initialization.

This is important, because it allows us to initialize the weights of the last layer optimally by solving a least squares problem. Thus we can start training with a set of parameters that fit the data well, and then improve by letting the network introduce nonlinearities.

3.2 Experimental setup

We have implemented the techniques described previously to test how these methods perform when approximating a high-frequency function. We choose the following function $f : \mathbb{R}^2 \to \mathbb{R}$, which we will approximate on $\Omega = [-1, 1]^2$:

$$f(x_1, x_2) = \sin(\omega\pi r_1) + \sin(\omega\pi r_2)$$

where:

i) r_1, r_2 are the euclidean distance to two points, namely:

   a) $r_1 = \sqrt{(x_1 - 0.5)^2 + (x_2 - 0.5)^2}$

   b) $r_2 = \sqrt{(x_1 - 1.5)^2 + (x_2 - 1.5)^2}$

ii) ω is a parameter that controls the frequency, which we will increase in the range $\omega \in [1, 10]$.

As we dial up the ω parameter, the networks will struggle more to approximate the function, as it will contain higher frequencies. We will compare several networks, all with roughly the same number of parameters and training iterations in order to have a fair comparison. Namely, the networks we will test are the following:

- A multilayer perceptron (MLP).
- A Kolmogorov Arnold Network (KAN).
- A MLP with Fourier Features (MLPF)
- A KAN with Fourier Features (KANF)
- A PirateNet (PIRATE).

We will measure the performance of each method in terms of the $L2\%$ error metric:

$$L2\% = 100 \times \frac{\|\hat{u} - u\|}{\|u\|} \tag{3.2}$$

which is just the standard L2 metric, scaled by 100 for percentage representation.

Table 3.1: L2% Values for Different Models Across Frequencies (Best values in **bold**)

Frequency ω	**MLP**	**KAN**	**MLPF**	**KANF**	**PIRATE**
1	0.9658	1.2385	0.4483	1.3996	**0.2945**
2	1.0902	1.1011	0.4160	0.8751	**0.2597**
3	2.0902	1.3618	0.3807	0.9127	**0.2624**
4	2.6374	4.5130	0.4603	1.3593	**0.3667**
5	10.0750	5.6117	0.6294	1.6934	**0.4494**
6	4.4290	5.0446	0.5704	2.3574	**0.4183**
7	24.5845	5.1453	0.6620	2.4075	**0.5991**
8	25.4194	15.7467	0.8581	3.8815	**0.6539**
9	40.6093	16.7734	0.9592	4.8579	**0.8467**
10	44.9243	14.3758	1.0260	9.4300	**0.8977**

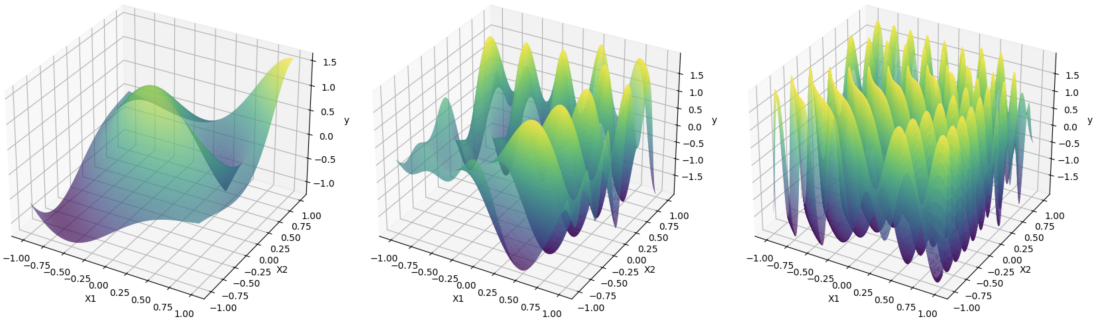


Figure 3.2: The function proposed with frequencies ω valued $1, 5, 10$.

3.3 Results

From the results in the table 3.1 we verify that high-frequency functions are problematic for the standard neural network architecture. We will focus on analysis on the last row of the table, corresponding to the highest frequency function we tested. We draw attention to the following conclusions:

i) A standard multilayer perceptron (MLP) is not capable of correctly approximating the high-frequency function, obtaining an L2% error of 44.92.

ii) We observe that the KAN suffers less from the spectral bias, obtaining a lower error than its MLP counterpart. Notice that KAN does not beat MLP on all low frequencies ω, suggesting that MLPs may be competitive for standard tasks. We also see that the results are noisy, e.g. MLP error suddenly increases at $\omega = 5$.

iii) The addition of Fourier Features (MLPF, KANF) reduces the L2% error significantly, although this feature works best when combined with the MLP architecture.

iv) The PirateNet architecture (PIRATE) obtains the best results at all frequencies. We

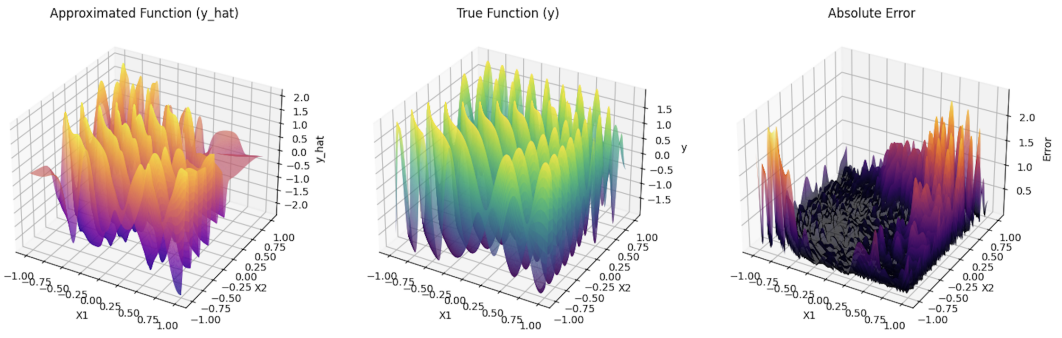


Figure 3.3: MLP model approximation of the frequency $\omega = 10$. The model fails to capture the true function, with complete failure at the extremes.

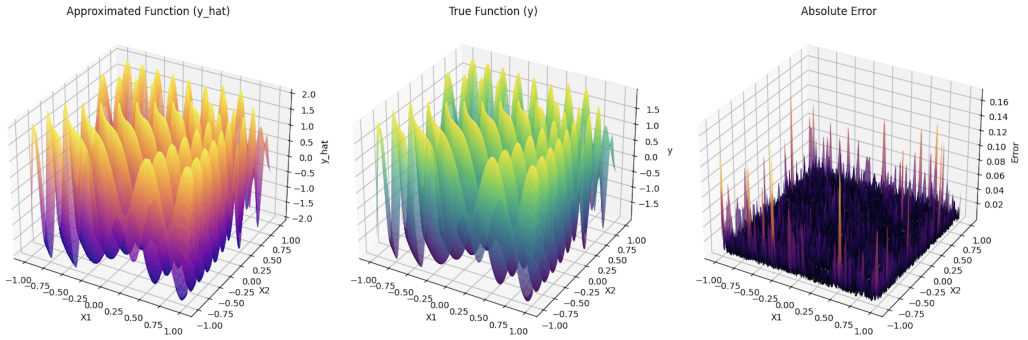


Figure 3.4: PIRATE model approximation of the frequency $\omega = 10$. The error is low, although we observe that it accumulates near the boundaries. Increasing the sampling probability at the boundary may reduce this phenomenon.

remark that all networks have a similar number of parameters (when the same could not be achieved the best network is disfavored, e.g. PIRATE has a bit less parameters than MLP).

The performance of the neural network architectures for approximating high-frequency functions can be ordered as follows, from least to most effective:

1. Multilayer Perceptron (MLP)

2. Kolgomorov Arnold Network (KAN)

3. Kolgomorov Arnold Network with Fourier Features (KANF)

4. Multilayer Perceptron with Fourier Features (MLPF)

5. PirateNet architecture (PIRATE), which also uses Fourier Features.

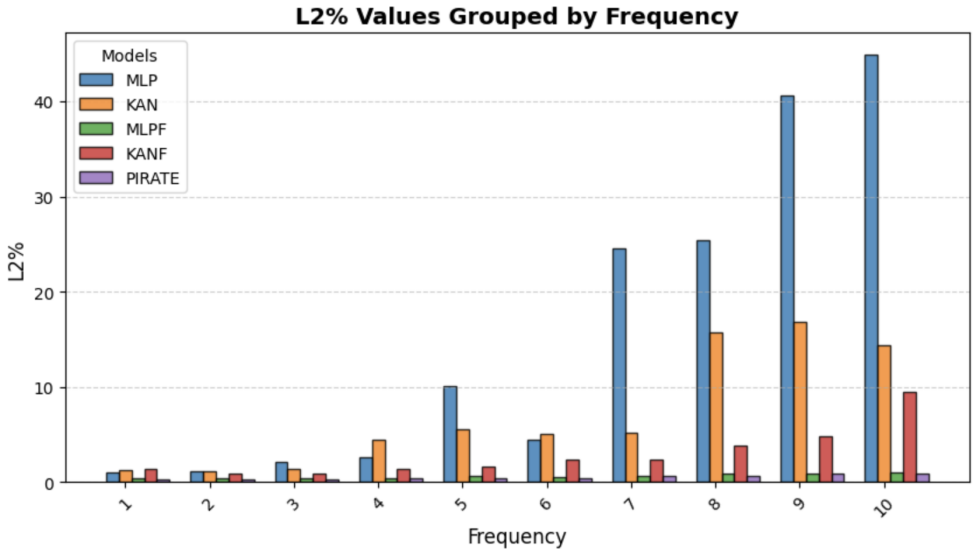


Figure 3.5: Results grouped by frequency, showing the performance in L2% error of each model. Both MLPF and PIRATE have similar errors for high-frequency functions, while KANF lags behind.

3.4 Spectral bias as an inductive bias

Let us conclude our discussion on high-frequency functions by reflecting on the concept of low-frequency bias. Throughout this chapter, we have treated spectral bias as an obstacle and explored methods to mitigate its effects. This naturally leads us to ask: *Why has a biased approximator been so successful in practice?*

One compelling explanation ties low-frequency functions to the principle of Occam's razor. Occam's razor, a philosophical guideline often paraphrased as follows, provides an intuitive perspective for understanding this phenomenon:

> "Among competing hypotheses, the simplest explanation is most likely to be correct."

When interpolating data, using a low-frequency function can be seen as the simplest approach. While this method fails when the underlying function contains high-frequency components, the success of low-frequency bias suggests that such cases are relatively rare. In many real world problems, the most important features of the data are captured by low-frequency functions. This aligns with the idea that simpler, smoother solutions are often sufficient, and even preferable, for modeling the phenomena we care about.

More generally, the strategy of guiding (or biasing) a network toward a specific set of solutions is known as an inductive bias. Spectral bias can be seen as the natural inductive bias of neural networks. Ultimately, interpolation is all about inductive biases, as there are, in principle, infinite possible functions that could have generated the observed data.

Chapter 4

Physics Informed Neural Networks

4.1 General Framework

In this chapter we will use Physics Informed Neural Networks (PINNs) to solve partial differential equation (PDE) problems. We will first define the problem to be solved, and then explain how we leverage neural networks for the task.

A PDE problem consists of finding a function that fulfills a differential equation and a set of data. For example, it may be the heat differential equation:

$$\frac{\partial u(t,x)}{\partial t} = \frac{\partial^2 u(t,x)}{\partial x^2}$$

The set of data is typically the values at the boundaries, for example the initial condition ($t = 0$) and the values on the boundaries:

$$\begin{aligned} u(t=0,x) &= \sin(\pi x) \\ u(t,x=0) &= 0 \\ u(t,x=1) &= 0 \end{aligned}$$

The idea of PINNs is to parametrize the solution $u(x,y)$ through a neural network with parameters θ, which we may denote as $\hat{u}(x,y;\theta)$. We then train the neural network to fit the data and fulfill the differential equation.

To fit the set of data we introduce the usual mean squared loss $\mathcal{L}_{DATA}$, which is big when the neural network does not match the known data:

$$\mathcal{L}_{DATA}(\theta) = \sum_{t,x \in \mathcal{D}} |u(t,x) - \hat{u}(\theta;t,x)|$$

where $\mathcal{D}$ is our set of initial and boundary conditions data. The novelty of PINNs is introducing a residual loss $\mathcal{L}_{PDE}$, that quantifies by how much the network does not fulfill

the differential equation, for our heat PDE:

$$\mathcal{L}_{PDE} = \sum_{t,x \in \Omega} \left| \frac{\partial \hat{u}(\theta; t, x)}{\partial t} - \frac{\partial^2 \hat{u}(\theta; t, x)}{\partial x^2} \right|^2$$

Where Ω is the domain of our PDE problem. Notice that this is just reflecting that if we solved correctly the heat equation, $\mathcal{L}_{PDE}$ should be zero.

Since the PDE often reflects some physical phenomena, the framework is called physics informed neural networks. To summarize: PINNs optimize their parameters to fit the solution of the PDE problem, by minimizing $\mathcal{L}_{PDE}$ and $\mathcal{L}_{DATA}$.

4.2 Application to PDE problems

We will implement the PirateNet neural network architecture and apply it to several PDE problems. We choose PirateNet because of its capability to approximate high frequency functions, as we have concluded from our experiments on chapter 3.

Furthermore, we will initialize the parameters as described in 3.1.3, which allows us to fit initial and boundary conditions at initialization and speeds up training.

The problems were picked to represent a challenge and test PINNs capabilities on approximating solutions with high frequency components, e.g. Helmholtz, Burgers, and learning nonlinear solutions e.g. Navier-Stokes.

4.2.1 Heat Equation

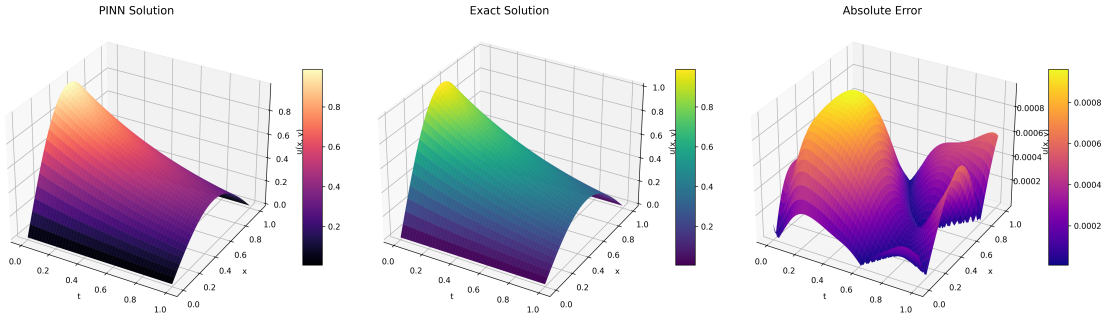


Figure 4.1: PINN solution for the heat equation

We consider the following partial differential equation (PDE) problem for the heat equation

$$\frac{\partial u(t,x)}{\partial t} = \alpha \frac{\partial^2 u(t,x)}{\partial x^2}, \quad x \in [0, L],\ t > 0 \tag{4.1}$$

subject to the initial condition:

$$u(0,x) = f(x), \quad x \in [0, L] \tag{4.2}$$

and Dirichlet boundary conditions:

$$u(t,0) = g_1(t), \quad u(t,L) = g_2(t), \quad t > 0 \tag{4.3}$$

where:

- $u(t,x)$ is the temperature at time t and position x,
- $\alpha > 0$ is the thermal diffusivity,
- $f(x)$ specifies the initial temperature distribution,
- $g_1(t)$ and $g_2(t)$ define the boundary conditions at $x = 0$ and $x = L$, respectively.

We will solve the problem on the domain $\Omega = [0,T] \times [0,L] = [0,1] \times [0,1]$, for the conditions:

- $u(0,x) = \sin(\pi x)$
- $u(t,0) = u(t,L=1) = 0$
- $\alpha = 1$

This problem has an analytical solution which we will use to compare our solution to. It is given by

$$u(t,x) = \sin(\pi x)e^{-t}$$

For our problem, the L^2 error achieved is:

$$L^2 \text{ error} = 1.26 \times 10^{-4} \tag{4.4}$$

The result is visualized in 4.1. The error seems to be evenly distributed across the domain, with a maximum error of 0.0008. This is not a particularly challenging example as the solution decreases smoothly.

4.2.2 Burgers' Equation

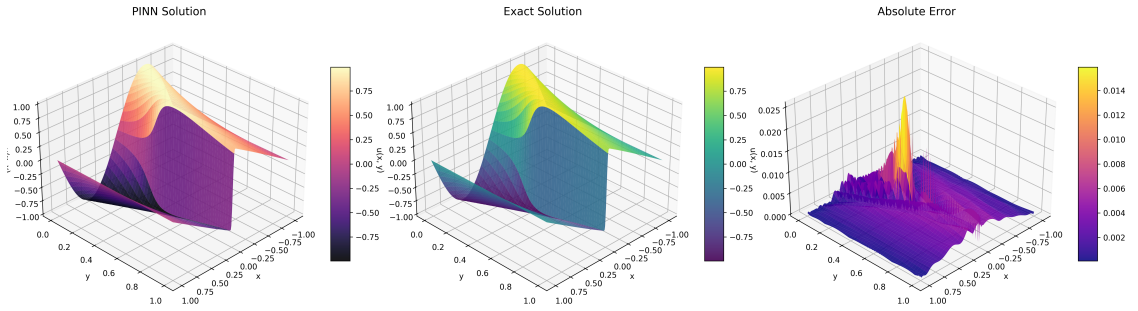


Figure 4.2: PINN solution for burgers equation

The Burgers' PDE problem is given by the following equation and boundary conditions:

$$\frac{\partial u(t,x)}{\partial t} + u\frac{\partial u(t,x)}{\partial x} - \nu\frac{\partial^2 u(t,x)}{\partial x^2} = 0, \quad x \in [-1,1],\, t \in [0,T] \tag{4.5}$$

subject to the initial condition:

$$u(0,x) = f(x), \quad x \in [-1,1] \tag{4.6}$$

and Dirichlet boundary conditions:

$$u(t,0) = g_1(t), \quad u(t,L) = g_2(t), \quad t > 0 \tag{4.7}$$

where:

- $u(t,x)$ is the fluid velocity at time t and position x,
- $\nu > 0$ is the viscosity coefficient,
- $f(x)$ specifies the initial fluid velocity distribution,
- $g_1(t)$ and $g_2(t)$ define the boundary conditions at $x = -1$ and $x = 1$, respectively.

We solve the problem on the domain $\Omega = [0,T] \times [-1,1] = [0,1] \times [-1,1]$, with the parameters:

- Viscosity: $\nu = 0.01/\pi$
- Initial condition: $u(0,x) = -\sin(\pi x)$
- Boundary conditions: $u(t,-1) = u(t,1) = 0$

We apply the same PirateNet neural network architecture with least squares initialization as used for the heat equation.

The L^2 error achieved for our solution is:

$$L^2 \text{ error} = 8.5 \times 10^{-3} \tag{4.8}$$

A comparison between the approximated and true solution is given in 4.1. Notice the strong jump in the true solution at later times. This is why we focused on high frequency functions on earlier chapters, as they are relevant in PDE problems like Burgers' and Helmholtz.

4.2.3 Helmholtz Equation

We solve the following Helmholtz's PDE problem with the given boundary and initial conditions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + k^2 u = f(x,y), \quad x,y \in [-1,1]$$

This equation can model several phenomena. For example in acoustics:

- $u(x,y)$: The pressure amplitude of the sound wave at the point (x,y).

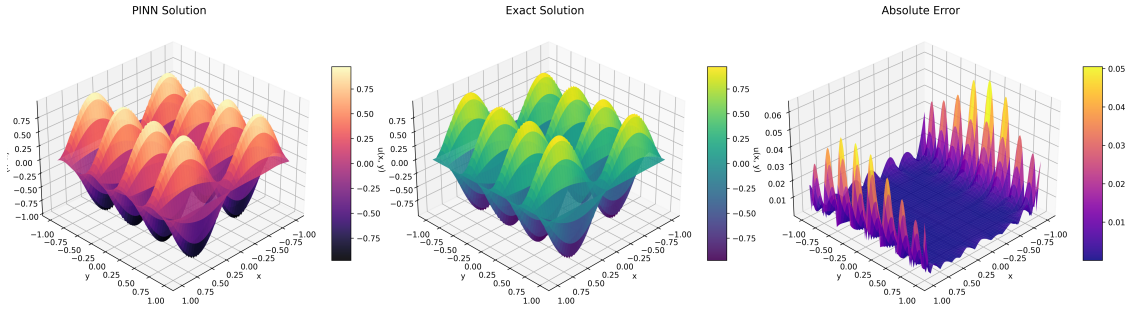


Figure 4.3: PINN solution for helmholtz equation

- k: The wave number, which determines the wavelength and the frequency of the wave. It determines the speed of the oscillation.
- $f(x, y)$: The source term, in our example this could be an instrument emitting sound.

We solve for the following parameters:

- $f(x, y) = (1 - 17\pi^2)\sin(\pi x)\sin(4\pi y)$
- $u(x = -1, y) = u(x = 1, y) = 0$
- $u(x, y = -1) = u(x, y = 1) = 0$

The L^2 error achieved for our solution is:

$$L^2 \text{ error} = 3.7 \times 10^{-3} \tag{4.9}$$

The result is given in figure 4.3. We observe the high frequency structure of the true solution being properly approximated by the neural network. It is interesting that most of the error accumulates near the boundaries. Further experiments could be done where points near the boundary are sampled more, to see if that would help reduce the error.

4.2.4 Navier–Stokes flow in a torus

Finally, we will simulate incompressible Navier-Stokes flow using the velocity-vorticity formulation. The problem is as follows:

$$\begin{aligned} w_t + \mathbf{u} \cdot \nabla w &= \frac{1}{Re}\Delta w, && \text{in } [0, T] \times \Omega \\ \nabla \cdot \mathbf{u} &= 0, && \text{in } [0, T] \times \Omega \\ w(0, x, y) &= w_0(x, y), && \text{in } \Omega \end{aligned}$$

Where $\mathbf{u} = (u, v)$ represents the velocity field x and y components, $w = \nabla \times \mathbf{u}$ and Re denotes the Reynolds number. In our example the domain is $\Omega = [0, 2\pi]^2$ and $Re = 100$. The initial condition is $w_0(x, y)$.

Notice that we did not specify boundary conditions. This is because we will be solving with periodic boundary conditions, that is:

$$u(t,x,y) = u(t,x+2\pi,y), \qquad \forall t,x,y \in [0,T]\times\Omega$$
$$u(t,x,y) = u(t,x,y+2\pi), \qquad \forall t,x,y \in [0,T]\times\Omega$$

We can enforce this periodicity on the neural network by adding the following layers at the beginning of the architecture:

$$\hat{u}(t,\sin(x),\sin(y))$$

where $\hat{u}(t,x,y)$ is our preferred neural network architecture, e.g. PirateNet. This layer is crucial, as it makes the values at the right of the domain smoothly transition to the ones to the left.

We now provide some frames of the simulation in 4.4, we strongly recommend visualizing the full video using the link. The full video demonstrates the flow's periodic behavior as it wraps around the torus.

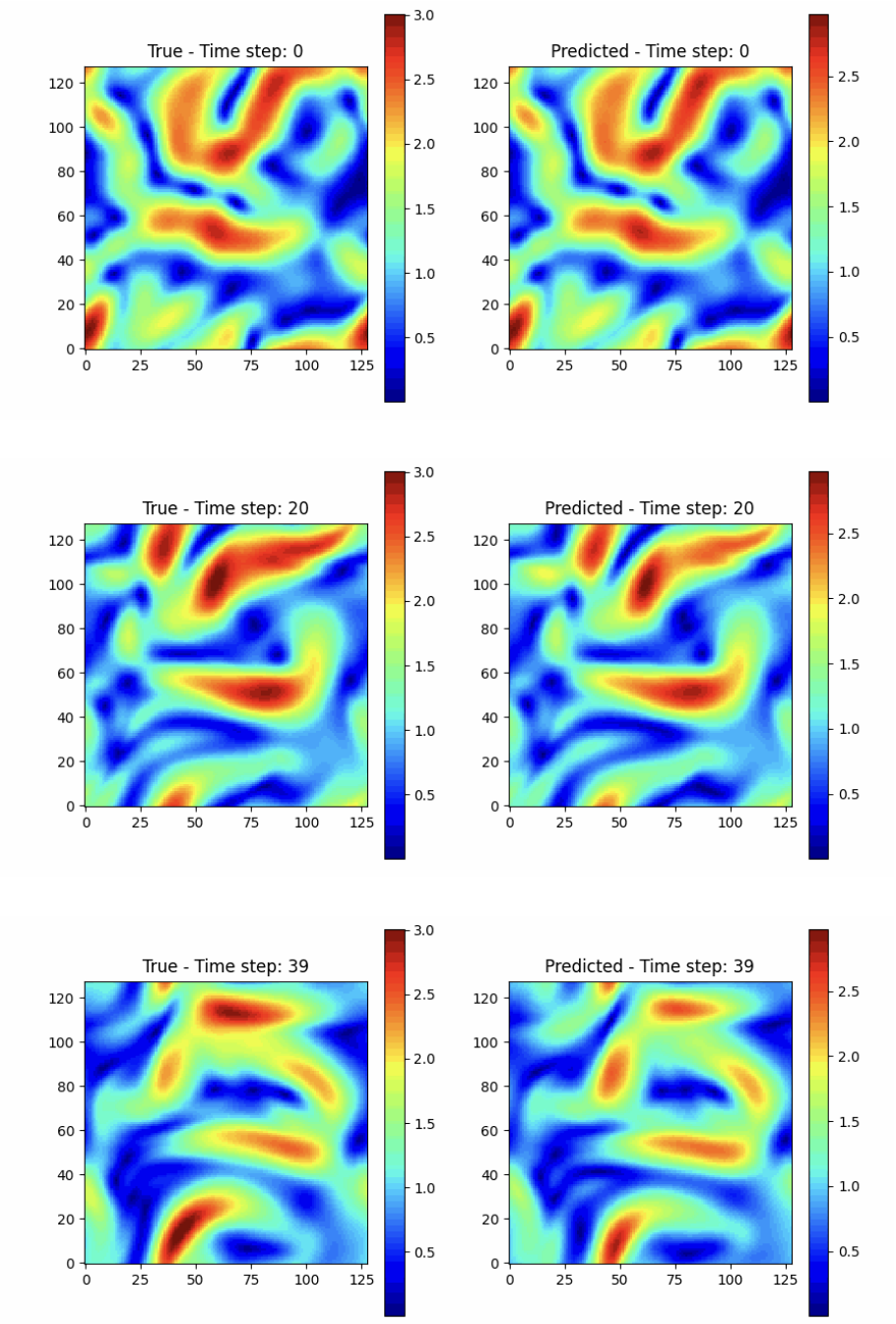


Figure 4.4: Comparison between norm of the velocity field at each point $||u(t,x,y)||^2$ versus PINN approximation $||\hat{u}(t,x,y)||$ corresponding to $t = 0s, 1s, 2s$. Visualization video available at github.com/DavidCanoRosillo/PIKAN

.

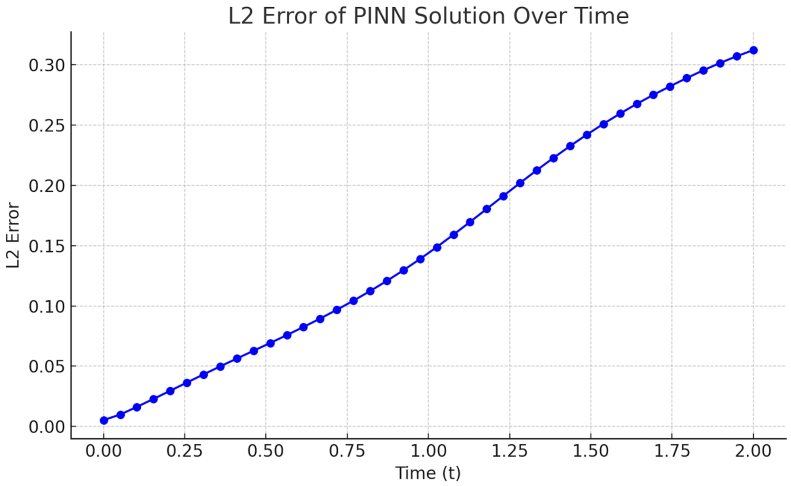


Figure 4.5: L2 Error of the PINN solution as time increases. Since the error accumulates, later times are much harder to predict.

We provide the L2 error results over time 4.5. We first notice we have higher error than in other examples, indeed Navier-Stokes is challenging for PINNs. Furthermore, error increases sharply as time goes on. This is because error accumulates, a small deviation from the true solution can snowball into significant differences in later times.

The challenge becomes clear when considering the network's task: mapping from $u : \mathbb{R}^3 \to \mathbb{R}^2$ while modeling nonlinear dynamics. However the neural network solution matches the true solution general flow structure, even at later times.

Chapter 5

Application to Fractional Differential Equations

In this chapter we will apply the PINNs framework to Fractional Differential Equations (FDE). These equations use the Caputo Fractional Derivative, which we will define in the next section. It is a type of integro-differential operator, which will pose several implementation challenges as we will have to numerically integrate the neural network over an interval to evaluate it. We describe the framework proposed to overcome these challenges, and apply it to a couple of challenging FDE problems.

5.1 Caputo Fractional Derivative

The Caputo Fractional Derivative is defined as:

$$D_t^\alpha f(t) = \frac{1}{\Gamma(\lceil\alpha\rceil - \alpha)} \int_0^t \frac{f^{(\lceil\alpha\rceil)}(\tau)}{(t-\tau)^{\alpha+1-\lceil\alpha\rceil}} d\tau \tag{5.1}$$

Where $\lceil\alpha\rceil$ is the ceiling of α and $f^{\lceil\alpha\rceil}$ is the integer order derivative.

Notice that the (Caputo) fractional derivative consists of an integral over $[0, t]$, which is then scaled by a constant dependent on α. Furthermore, the integrand consists of the values of $f(\tau)$ for $\tau \in [0, t]$ in the numerator, and the denominator increases for τ values near t.

From a modelling perspective, the integral allows the system represented to exhibit strong causality. The time fractional derivative not only depends on the current value $f(t)$, but also on all the values $f(\tau)$, for $0 < \tau < t$.

5.2 Implementation details

Using neural networks to solve FDEs is a challenging task as one of the steps in the PINN framework is more computationally demanding and harder to implement. We will first recap what we know about solving PDEs and how we can adapt the method to FDEs.

From the previous chapter we know that solving a PDE problem with a PINN entails the following key step:

- Program the loss associated with the equation $\mathcal{L}_{PDE}$.

This is the only step that changes for FDEs. How does this step look for an FDE? Say we want to implement the fractional diffusion equation:

$$D_t^\alpha u(t,x) - \frac{\partial^2 u}{\partial x^2} = 0 \tag{5.2}$$

The term with the partial derivatives can be implemented as before. We differentiate the neural network with respect to (wrt.) x and use the output to train the network (differentiating over parameters).

5.2.1 Multi GPU training

The complication is with the Caputo derivative term. To deal with the integral in the definition we will have to use a quadrature, which although conceptually simple will be computationally very demanding.

Suppose we use 100 points for computing the integral. We normally sample batches of so called collocation points, e.g. 2^{14} points per batch. Then to evaluate the loss of the neural network we will need 100 points to calculate each of the 2^{14} integrals required for the loss at each collocation point.

So computing the fractional derivative is costly. Throughout this thesis everything was programmed using parallelization on the GPU. This allowed us to evaluate the loss at all the collocation points at once. We can also use this technique for the fractional derivative, but one quickly runs out of GPU memory and can't do all the collocation points at once.

Using 100 points for computing the integral requires 100 times the memory we previously were using, and in practice we found that one requires about 300 quadrature points for good results. How were we able to overcome these issues?

The solution we found was to scale horizontally, that is distributing the compute along several GPUs. Each GPU computes the loss for the collocation points, and computes how the parameters of the neural network should be updated. Then all the parameters updates are combined, in our case we used average, and sent to each GPU so they can process the next batch of collocation points.

5.2.2 Improving temporal sampling

We have so far been able to introduce more collocation points by using the available hardware better. We can also ask ourselves which collocation points will help the neural network learn the FDE solution faster.

Notice how in the Caputo fractional derivative definition, the integral makes all values before the current collocation point count. If we do not get right all the points before, the solution to the current collocation point is flawed from the start. In other words, the solution exhibits strong causality.

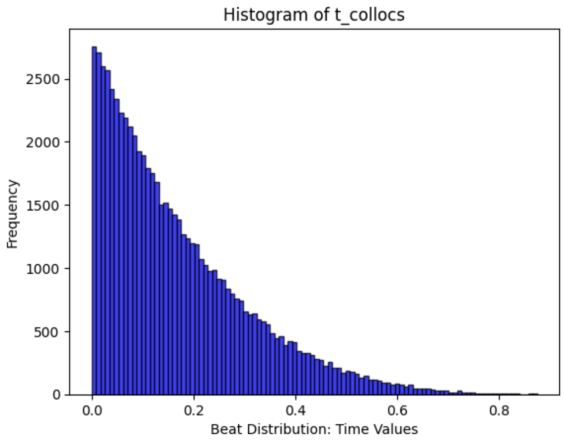


Figure 5.1: Frequency graph of the sampled time values used for training the neural network. Sampling low values more helps enforce causality and avoids getting stuck in unaccurate solutions.

From this we extract the insight that collocation points early in time are the most important. Thus, it makes sense that we sample with higher probability those points, as we want the neural network to approximate well them.

To this end we replace the usual uniform sampling distribution with a beta distribution skewed towards lower time values, see 5.1. From our experiments we find this technique to stabilize training and speed up convergence.

5.3 Fractional Relaxation-Oscillation Equation

The first FDE we will solve is the Fractional Relaxation-Oscillation equation, given by:

$$D_t^\alpha u(t) + u(t) = 0, \quad t \in [0, 1] \tag{5.3}$$

We set the following α and initial conditions

- $u(t = 0) = 1$
- $\alpha = 0.5$

The analytical solution of the fractional differential equation using the Caputo derivative is given by:

$$u(t) = E_\alpha(-t^\alpha) \tag{5.4}$$

where $E_\alpha(\cdot)$ is the Mittag-Leffler function defined as:

$$E_\alpha(z) = \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(\alpha k + 1)} \tag{5.5}$$

For our specific case where $\alpha = 0.5$, the solution becomes:

$$u(t) = E_{0.5}(-t^{0.5}) \tag{5.6}$$

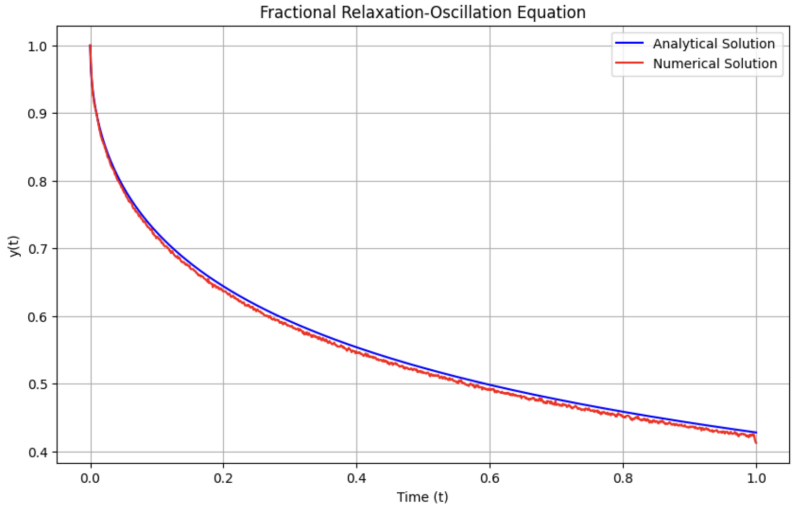


Figure 5.2: Analytical vs numerical solution for the Fractional Relaxation-Oscillation equation.

We obtain an L2 error of:

$$L^2 \text{ error} = 1.2 \times 10^{-2} \tag{5.7}$$

The approximated vs analytical result can be visualized on 5.2. We observe that accurate results are obtained, although the error accumulates in time. Thanks to improved temporal sampling, the PINN approximates earlier collocation points well. This is crucial for getting later times on, as there is high causality.

5.4 Fractional Diffusion Equation

We now consider the time-fractional diffusion equation defined as:

$$D_t^\alpha u(t,x) = \frac{\partial^2 u}{\partial x^2}, \quad (t,x) \in [0,1] \times [0,1] \tag{5.8}$$

subject to the initial condition:

$$u(t=0,x) = \sin(\pi x) \tag{5.9}$$

The analytical solution to the time-fractional diffusion equation with the given initial condition is known and can be expressed using the Mittag-Leffler function:

$$u(t,x) = E_\alpha(-\pi^2 t^\alpha)\sin(\pi x) \tag{5.10}$$

where $E_\alpha(\cdot)$ is the Mittag-Leffler function:

$$E_\alpha(z) = \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(\alpha k + 1)} \tag{5.11}$$

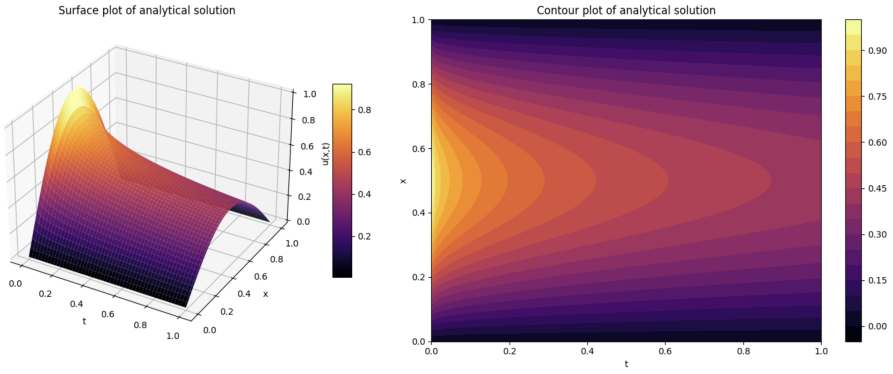


Figure 5.3: Analytical solution of the Fractional Diffusion Equation.

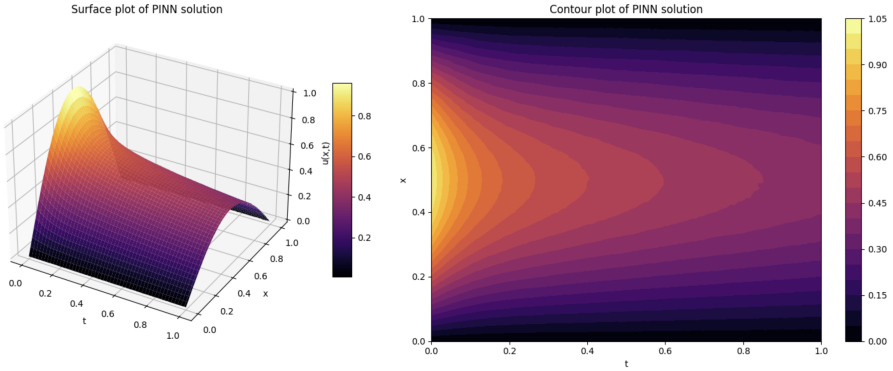


Figure 5.4: PINN-based numerical solution.

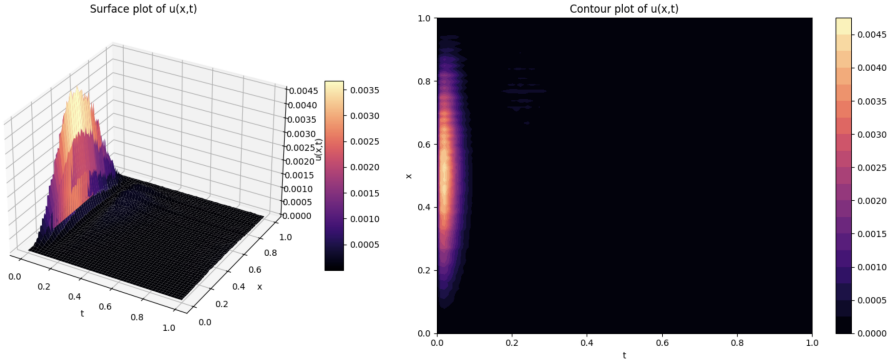


Figure 5.5: Pointwise error between the numerical and analytical solutions.

We provide plots of the analytical 5.3 and PINN 5.4 solution. Furthermore, the absolute error can be visualized in the plot 5.5.

The L^2 error between the numerical and analytical solution is:

$$L^2 \text{ error} = 2.8 \times 10^{-2} \tag{5.12}$$

The PINN solution accurately captures the solution behavior, although we see most of

the error at the beginning. We attribute this result to the solution changing faster at the beginning.

Chapter 6

Conclusion

Throughout this thesis we have given a broad vision of the challenges and state of the art applying PINNs for PDEs and FDEs. More concretely, we have:

- Reviewed neural networks and theoretically motivated the issue of low-frequency bias through the NTK perspective.
- Reviewed state of the art for addressing low-frequency bias, and benchmarked different methods finding the PirateNet architecture to work best.
- Applied the PINN framework to several challenging, high-frequency PDE problems like Burgers' equation and Navier-Stokes.
- Developed a framework for solving fractional differential equations (in the Caputo sense) with PINNs, by scaling GPU training and tackling causality.

While challenges remain, particularly regarding training stability and generalization in complex regimes, the results presented here underscore the viability of PINNs for both classical and fractional differential equations. This thesis contributes to our understanding of the PINN landscape and highlights key areas for further exploration, like fractional differential equations. We hope it inspires continued research at the intersection of physics, mathematics, and machine learning.

All code was developed using the JAX [10] library, which enabled parallelization of neural network training on GPUs. The full source is available at `https://github.com/DavidCanoRosillo/PDEs_and_FDEs_with_PINNs`.

We would like to thank the developers of the software libraries used, particularly the JaxKAN implementation [11], the JAX library [10], and the Matplotlib library [12].

Bibliography

[1] S. Sheehan and Y. Song, "Deep learning for population genetic inference", *PLOS Computational Biology*, vol. 12, e1004845, Mar. 2016. DOI: 10.1371/journal.pcbi.1004845.

[2] Y. Liu, *Neural networks are integrable*, 2024. arXiv: 2310.14394 [math.NA]. [Online]. Available: https://arxiv.org/abs/2310.14394.

[3] Z. Liu *et al.*, *Kan: Kolmogorov-arnold networks*, 2024. arXiv: 2404.19756 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2404.19756.

[4] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, *Visualizing the loss landscape of neural nets*, 2018. arXiv: 1712.09913 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1712.09913.

[5] Y. Cao, Z. Fang, Y. Wu, D.-X. Zhou, and Q. Gu, *Towards understanding the spectral bias of deep learning*, 2020. arXiv: 1912.01198 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1912.01198.

[6] S. Arora, S. S. Du, W. Hu, Z. Li, R. Salakhutdinov, and R. Wang, *On exact computation with an infinitely wide neural net*, 2019. arXiv: 1904.11955 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1904.11955.

[7] Y. Wang, J. W. Siegel, Z. Liu, and T. Y. Hou, *On the expressiveness and spectral bias of kans*, 2025. arXiv: 2410.01803 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2410.01803.

[8] M. Tancik *et al.*, *Fourier features let networks learn high frequency functions in low dimensional domains*, 2020. arXiv: 2006.10739 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2006.10739.

[9] S. Wang, B. Li, Y. Chen, and P. Perdikaris, *Piratenets: Physics-informed deep learning with residual adaptive networks*, 2024. arXiv: 2402.00326 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2402.00326.

[10] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/jax-ml/jax.

[11] S. Rigas, M. Papachristou, T. Papadopoulos, F. Anagnostopoulos, and G. Alexandridis, *Adaptive training of grid-dependent physics-informed kolmogorov-arnold networks*, 2024. DOI: https://doi.org/10.1109/ACCESS.2024.3504962. arXiv: 2407.17611 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2407.17611.

[12] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.