Solving Differential Equations with Physics-Informed Neural Networks

Candidate Number: 1081225

May 5, 2024

1 Introduction

Differential equations are equations involving unknown functions of one or more independent variables and their derivatives. Solving these equations is an extensive topic in various fields such as fluid mechanics and mathematical finance, with new applications continually emerging.

Since most differential equations cannot be solved analytically, the advent of computers in the mid-1900s eventually led to the development of numerical methods for solving complex equations, such as nonlinear ones or those defined over intricate geometries. Despite the popularity and power of traditional numerical methods involving finite differences and finite elements, limitations of these methods still exist, including having difficulty in handling nonlinear problems and obtaining fast solutions that are precise enough for high-dimensional problems.

The recent resurgence in deep neural networks has opened up a brand new track for numerically solving differential equations, especially under circumstances when traditional methods are prone to failure. Specifically, these methods involving neural networks particularly excel in handling nonlinear equations [13] as well as high-dimensional problems [5], and can adapt to complex geometries provided suitable sampling methods are developed [3].

This report will mainly focus on the implementation of solving differential equations with physics-informed neural networks (PINNs). In Section 2, we will introduce some preliminary knowledge relevant to the paper. In Section 3, we will apply the method to solving one-variable ordinary differential equations and discuss the limitations of the approach when solutions with high frequency get involved. The example of a 2D elliptic partial differential equation will be displayed in Section 4 together with an extension of the method for solving 3D wave equations. Finally, Section 5 will provide a simple summary of our work with some possible directions for improvements of the method. All codes in this paper are available through the link given in Appendix A.1.

2 Preliminaries

2.1 Feedforward Neural Networks

Before formally discussing the steps of solving differential equations, we need to briefly review the definition and calculation principles of a simple feedforward neural network. Historically, the artificial neural network was first inspired by its biological counterpart. Biological neural networks are composed of simple neurons, and for each neuron, its input is received as transmitter substances through the dendrites

of the cell, and processed by the internal biological mechanisms. The signal is then transmitted through the axon and triggers the release of new transmitter substances sent to other neurons.

The neuron of an artificial neural network is modelled in a similar way, which receives signals from other previously connected neurons as its input $\mathbf{x}$, performs a weighted summation of these signals $\mathbf{w}^{\mathrm{T}}\mathbf{x}$, subtracts a bias b and then activates the result through some pre-specified activation function $\sigma_A(\mathbf{w}^{\mathrm{T}}\mathbf{x} - \mathrm{b})$ and eventually forms the output of the neuron. The output then becomes the new input for the neurons in the next layer to which the current neuron is connected, as given in Fig 2.1.1.

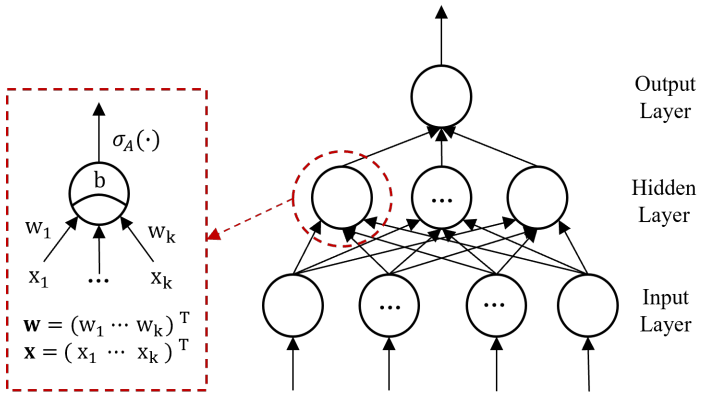

Fig. 2.1.1 A simple three-layer fully connected feedforward neural network.

A network consisting of N neurons can be viewed as a multidimensional function $\varphi_\theta(\mathbf{x})$ with some specified parameters θ, where θ is composed of the weights $\{\mathbf{w}_i\}_{i=1}^N$, and the biases $\{\mathrm{b}_i\}_{i=1}^N$ for each neuron to perform its inside calculation. The learning process of the network is nothing but an optimization problem subject to the selected loss function $\mathcal{L}(\theta)$ for the algorithm specified by the learning rule to minimize. Past theories have proven that, as long as the proper activation function and a sufficient number of neurons are given, even networks with a single hidden layer can have the ability to approximate any continuous functions and their derivatives with arbitrary accuracy [6].

2.2 Automatic Differentiation

Modern optimization algorithms used to minimize the loss function $\mathcal{L}(\theta)$ usually require a computation of gradients $\nabla_\theta \mathcal{L}(\theta)$. In most of the machine learning frameworks, this is realized by a mechanism called automatic differentiation [2]. Before entering into the detailed implementation, we first need to introduce the concept of computational graphs. Consider a scalar function $y = f(\mathbf{x})$ with vectorized independent variables $\mathbf{x} = [\mathrm{x}_1, \cdots, \mathrm{x}_n]^{\mathrm{T}} \in \mathbb{R}^n$, its computational graph is defined as a graph

with nodes representing intermediate values and edges indicating input-output relations in the corresponding computation. For example, the following Fig. 2.2.1 gives a computational graph defined for function $y = f(\mathrm{x}_1, \mathrm{x}_2) = \ln(\mathrm{x}_1) + \mathrm{x}_1\mathrm{x}_2 - \sin(\mathrm{x}_2)$.

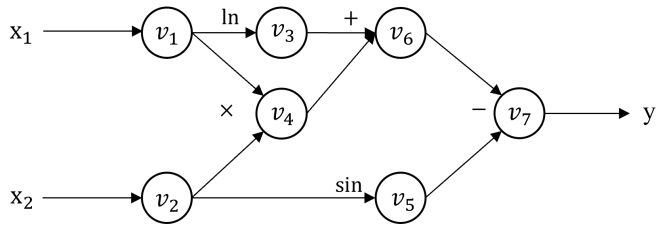

Fig. 2.2.1 Computational graph for function $f(\mathrm{x}_1, \mathrm{x}_2) = \ln(\mathrm{x}_1) + \mathrm{x}_1\mathrm{x}_2 - \sin(\mathrm{x}_2)$.

Let v_i for $i \in \{1, \cdots, N\}$ be the value of the i_{th} node in the computational graph for the function $y = f(\mathbf{x})$, and $\mathrm{in}(i)$, $\mathrm{out}(i)$ be the set of indices for its connected input and output nodes such that $k < i$ strictly for all $k \in \mathrm{in}(i)$. We define the i_{th} "adjoint" to be

$$v_i^* = \frac{\partial y}{\partial v_i}, \tag{2.2.1}$$

and its "partial adjoint" to be

$$v_{i\to j}^* = v_j^* \cdot \frac{\partial v_j}{\partial v_i} \quad \text{for} \quad j \in \mathrm{out}(i), \tag{2.2.2}$$

where for the final node v_N we define $v_{N\to(N+1)}^* = 1$ for $\mathrm{out}(N) = \{N+1\}$. Using the chain rule, it is not hard to deduce the iterative formula to compute the adjoints

$$v_i^* = \textstyle\sum_{j\in\mathrm{out}(i)} v_{i\to j}^*. \tag{2.2.3}$$

Letting $v_p = \mathrm{x}_p$ for independent variables $p \in \{1, \cdots, n\}$, the partial derivatives $\frac{\partial y}{\partial \mathrm{x}_p} = v_p^*$ composing the gradients can then be computed with the following reverse automatic differentiation algorithm as displayed in Algorithm 1.

Algorithm 1: Reverse Automatic Differentiation

Initialize: Dictionary of lists recording partial adjoints for each node
D $\longleftarrow \{N : [1]\}$, target index p, current index $i \longleftarrow N$;

while $i > p$ **do**
  compute the adjoint $v_i^* \longleftarrow \mathrm{sum}(\mathrm{D}[i])$
  for $k \in \mathrm{in}(i)$ **do**
    compute the partial adjoint $v_{k\to i}^* \longleftarrow v_i^* \cdot \frac{\partial v_i}{\partial v_k}$
    append $v_{k\to i}^*$ to D[k]
  $i \longleftarrow i - 1$
$v_p^* \longleftarrow \mathrm{sum}(\mathrm{D}[p])$

2.3 PINNs: Physics-Informed Neural Networks

2.3.1 General Formulation

The Physics-Informed Neural Networks (PINNs) provide a general unsupervised framework for solving differential equations with deep neural networks. Over the recent decades, the method has been widely studied and plays a significant role in solving inverse problems [1] and equation discovery [4].

Given a differential equation with the initial and boundary conditions that the solution satisfies

$$\mathcal{D}[u(\mathbf{x})] = f(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Omega \subset \mathbb{R}^d, \tag{2.3.1.1}$$

$$\mathcal{B}_k[u(\mathbf{x})] = g_k(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Gamma_k \subset \partial\Omega, \tag{2.3.1.2}$$

where $\mathcal{D}$ and $\mathcal{B}_k$ are some differential operators and $u(\mathbf{x})$ is the solution of the equation. PINNs aim to train a neural network $\varphi_\theta(\mathbf{x})$ to approximate the solution $u(\mathbf{x})$ by minimizing the following loss function for a batch of points $\{\mathbf{x}_i : \mathbf{x}_i \in \Omega\}_{i=1}^{N_p}$, $\{\mathbf{x}_{k,j} : \mathbf{x}_{k,j} \in \Gamma_k\}_{j=1}^{N_{b,k}}$

$$\min_\theta \mathcal{L}(\theta) = \mathcal{L}_p(\theta) + \mathcal{L}_b(\theta), \tag{2.3.1.3}$$

$$\mathcal{L}_p(\theta) = \frac{1}{N_p} \sum_{i=1}^{N_p} \Big\| \ \mathcal{D}[\varphi_\theta(\mathbf{x}_i)] - f(\mathbf{x}_i) \ \Big\|^2, \tag{2.3.1.4}$$

$$\mathcal{L}_b(\theta) = \sum_k \frac{\lambda_k}{N_{b,k}} \sum_{j=1}^{N_{b,k}} \Big\| \ \mathcal{B}[\varphi_\theta(\mathbf{x}_{k,j})] - g_k(\mathbf{x}_{k,j}) \ \Big\|^2, \tag{2.3.1.5}$$

where $\lambda_k > 0$ are pre-specified parameters; $\mathcal{L}_p(\theta)$ and $\mathcal{L}_b(\theta)$ are referred to as the **physics loss** and the **boundary loss** respectively.

To avoid the problem of vanishing gradients, the activation function for each hidden neuron in PINNs should be non-linear and infinitely differentiable. Therefore, the **Tanh** activation will be chosen for all neurons in hidden layers throughout the experiments in this paper. The derivatives involved in the loss (2.3.1.4) and (2.3.1.5) can be easily obtained using modern learning frameworks such as PyTorch and TensorFlow with automatic differentiation [11].

2.3.2 Imposing Initial and Boundary Conditions

There are mainly two ways to impose the initial and boundary conditions on the neural network output φ_θ. One way is to apply the general formulation (2.3.1.3) and increase the value of λ_k. Such a method is called PINNs with **soft** conditions. Another choice is to use the neural network as a part of the solution ansatz so that the network's output will always satisfy the required boundary (resp., initial) conditions.

The latter method to impose the conditions of differential equations is called PINNs with **hard** conditions.

Once a hard constraint is asserted on the network's output, the boundary loss $\mathcal{L}_b(\theta)$ will no longer be needed as its contribution to the total loss $\mathcal{L}(\theta)$ will always be zero. Therefore, the problem will become fully unsupervised and the loss that the network aiming to minimize will be reduced to

$$\min_{\theta} \mathcal{L}(\theta) = \mathcal{L}_p(\theta). \tag{2.3.2.1}$$

3 Solving Ordinary Differential Equations

3.1 Examples of Ordinary Differential Equations

3.1.1 Examples with Dirichlet Boundary Conditions

In this section, we will consider solving the following second-order ordinary differential equation with Dirichlet boundary conditions (which is also referred to as the **type 1 condition** throughout this paper for clarity)

$$y''(x) = f(x, y) \quad \text{for} \quad x \in [a, b] \ , \tag{3.1.1.1}$$

$$y(a) = y_a \quad \text{and} \quad y(b) = y_b \ . \tag{3.1.1.2}$$

PINNs with hard boundary conditions will be applied to obtain the solution of the equation. Let $\hat{\varphi}_\theta(x)$ be the original output of the network. The modified output satisfying the Dirichlet boundary conditions is given as

$$\varphi_\theta(x) = \hat{\varphi}_\theta(x) + \frac{b-x}{b-a} \cdot [\ y_a - \hat{\varphi}_\theta(a)\ ] + \frac{x-a}{b-a} \cdot [\ y_b - \hat{\varphi}_\theta(b)\ ], \tag{3.1.1.3}$$

with a reduced loss function defined over a series of sampled points $\{x_i\}_{i=1}^{N_p} \subset [a, b]$

$$\mathcal{L}(\theta) = \frac{1}{N_p} \sum_{i=1}^{N_p} \Big\| \ \varphi''_\theta(x_i) - f(x_i, \varphi_\theta(x_i)) \ \Big\|^2. \tag{3.1.1.4}$$

The networks follow an architecture as displayed in Fig. 3.1.1.1 below. The test examples we used to illustrate the method in Section 3.1.4 are listed in Table 3.1.1.1 for $a = 0$ and $b = 1$.

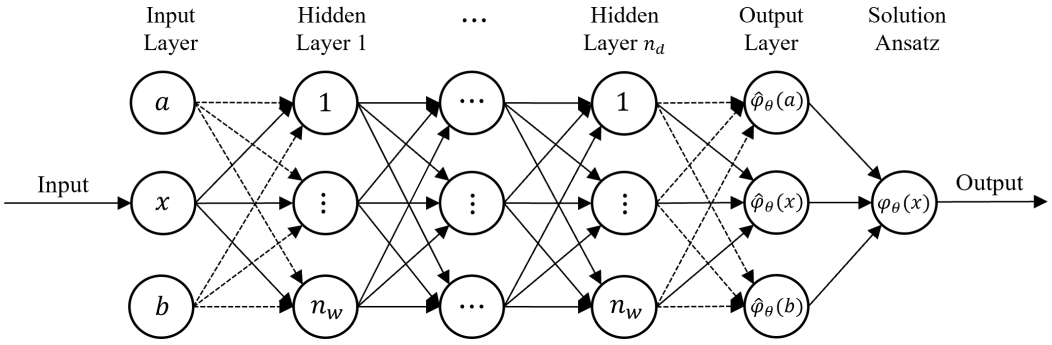

Fig. 3.1.1.1 Architecture for a fully connected PINN with a uniform width n_w and depth n_d, i.e., n_d hidden layers.

Index	Forcing Function	True Solution	Target Interval
i	$f(x,y)$	y_{true}	(a , b)
1	$-y$	$\sin(x)$	(0 , 1)
2	$y^2-[1+x(1-x)]^2-2$	$1+x(1-x)$	(0 , 1)
3	y	e^x	(0 , 1)
4	$e^x[(1-16\pi^2)\sin(4\pi x)+8\pi\cos(4\pi x)]$	$e^x\sin(4\pi x)$	(0 , 1)

Table 3.1.1.1 Known solutions for Dirichlet boundary condition examples.

3.1.2 Examples with Other Boundary Conditions

The method we used in the previous section for equations with type 1 (Dirichlet) conditions can also be easily generalized to those with other boundary conditions. For example, it can be applied to the same equation with the following **type 2** condition

$$y''(x) = f(x,y,y') \quad \text{for} \quad x\in[a,b]\ , \tag{3.1.2.1}$$

$$y(a) = y_a \quad \text{and} \quad y'(b) = y_b\ , \tag{3.1.2.2}$$

with ansatz

$$\varphi_\theta(x) = \hat{\varphi}_\theta(x) + y_a - \hat{\varphi}_\theta(a) + (x-a)\cdot[\ y_b - \hat{\varphi}'_\theta(b)\ ]; \tag{3.1.2.3}$$

and **type 3** condition

$$y(a) + y'(a) = y_a \quad \text{and} \quad y'(b) = y_b\ , \tag{3.1.2.4}$$

with ansatz

$$\varphi_\theta(x) = \hat{\varphi}_\theta(x) + y_a - [\ \hat{\varphi}_\theta(a) + \hat{\varphi}'_\theta(a)\ ] + (x-a-1)\cdot[\ y_b - \hat{\varphi}'_\theta(b)\ ]. \tag{3.1.2.5}$$

The network's structure is exactly the same as the one in Fig 3.1.1.1, with the loss function turning into

$$\mathcal{L}(\theta) = \frac{1}{N_p} \sum_{i=1}^{N_p} \left\| \ \varphi_\theta''(x_i) - f(x_i, \varphi_\theta(x_i), \varphi_\theta'(x_i)) \ \right\|^2. \tag{3.1.2.6}$$

The test examples used are summarized in Table 3.1.2.1.

Index	Forcing Function	True Solution	Target Interval
i	$f(x, y, y')$	y_{true}	(a , b)
5	$-y \cdot [\ y^2 + (y')^2\ ]$	$\cos(x)$	(0 , 1)
6	$-2xy' - 2y$	$2\exp(-x^2)$	(0 , 1)
7	$1 - (y')^2$	$\ln[\cosh(x)]$	(0 , 1)
8	$-y$	$\sin(x)$	(0 , 1)

Table 3.1.2.1 Known solutions for examples involving conditions of type 2 and 3.

3.1.3 An Example of ODE Systems

Finally, we generalize the method to solve systems of second-order ordinary differential equations. Consider

$$y''(x) = f(x, y, y') \quad \text{for} \quad x \in [a, b] \ , \tag{3.1.3.1}$$

$$y(a) = y_a \quad \text{and} \quad y(b) = y_b \ , \tag{3.1.3.2}$$

for some $y, f \in \mathbb{R}^d$. The solution ansatz and the loss function are the same as the one presented in formulas (3.1.1.3) and (3.1.2.6) in vectorized form. The only change we need to specify is that the output layer of the network now consists of d neurons.

For the numerical experiment, we consider the following system of equations

$$u''(x) = x(u-1) - v - \left(\frac{\pi}{2}\right)^2 \sin\left(\frac{\pi}{2}x\right) - 2, \tag{3.1.3.3}$$

$$v''(x) = v - x^2(1-x) - 6x + 2, \tag{3.1.3.4}$$

for $x \in [0, 1]$ with $y(x) = [u(x), v(x)]^{\mathrm{T}}$. Under the Dirichlet boundary condition (3.1.3.2), the solution of the system is given by

$$u(x) = x(1-x) + \sin\left(\frac{\pi}{2}x\right), \tag{3.1.3.5}$$

$$v(x) = x^2(1-x). \tag{3.1.3.6}$$

3.1.4 Numerical Results

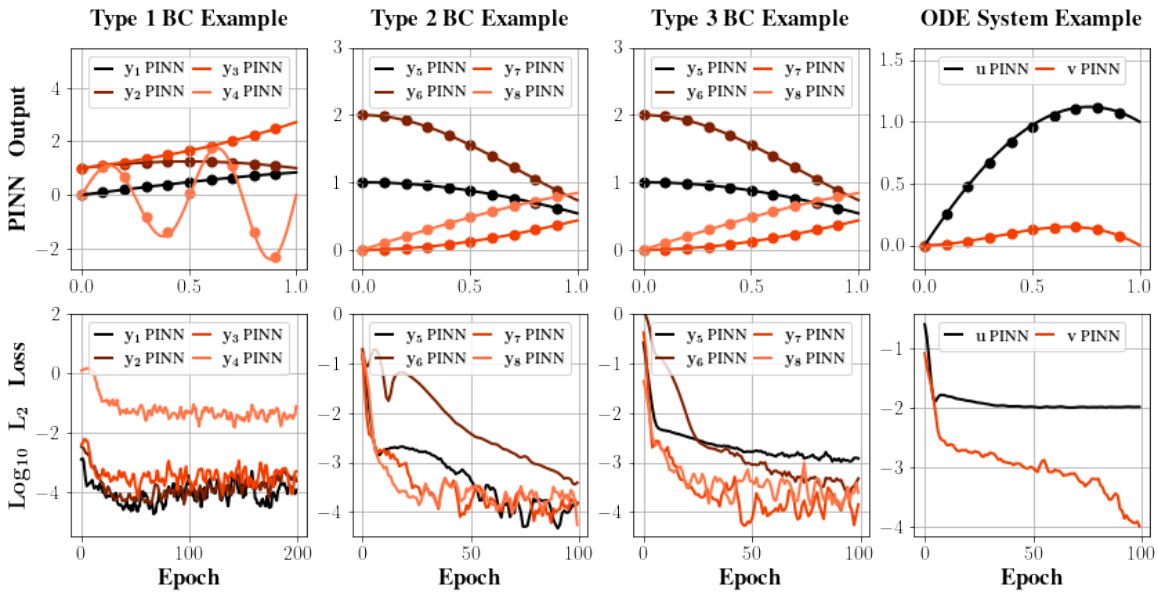

Fig. 3.1.4.1 Solutions of Section 3.1 sample equations generated by PINNs.

Solutions of the sample equations listed in Table 3.1.1.1, Table 3.1.2.1 and equations (3.1.3.3 - 4) are obtained by training a simple 3-layer neural network with only one hidden layer composed of 50 neurons. For each model, only 1000 points (2000 points for solving the ODE system) are sampled from the interval. All networks are trained using the Adam optimizer with a batch size of 32 [7].

For the convenience of comparing the convergence of solving different equations, the L_2 losses are recorded instead of the physics loss $\mathcal{L}_p$. Recall the L_2 loss is defined as

$$L_2(\theta) = \left(\int_a^b \left\| \varphi_\theta(x) - y_{\text{true}}(x) \right\|^2 \mathrm{d}x \right)^{1/2}. \tag{3.1.4.1}$$

The final results are displayed in the above Fig. 3.1.4.1. Note that for most of the examples where the true solutions are monotonic functions over the target interval, the method performs a rapid convergence and achieves an accuracy below 10^{-2} within only 100 training epochs. However, for a solution with a relatively high latent frequency, for instance, y_4, the method tends to converge in a much slower way.

3.2 Scaling to Higher Frequencies

In previous examples, we notice that the PINNs tend to converge faster for functions with lower frequencies. Indeed, literature suggests neural networks prioritise learning lower-frequency functions [12]. Furthermore, the capability of these networks to fit high-frequency functions is also bounded by the number of neurons and trainable parameters.

In fact, once the solution involves high-frequency terms, regular PINNs may take a significant amount of time to converge to adequate accuracy even for relatively simple equations. One solution to this challenge of accelerating convergence when scaling PINNs to higher frequencies is introducing Random Fourier Features (RFF) into the training process [14].

The effect of these Fourier features can be viewed as an additional initialization step for the network input. Instead of passing the value of the independent variables $\mathbf{x}$ directly into the network $\varphi_\theta(\mathbf{x})$, we first convert the input into a series of triangular signals with a pre-specified, fixed random Gaussian matrix G using the map

$$\mathrm{RFF}(\mathbf{x}) = [\ \cos(2\pi \mathrm{G}\mathbf{x})\ ,\ \sin(2\pi \mathrm{G}\mathbf{x})\ ]^{\mathrm{T}}, \tag{3.2.1}$$

where the components of G are drawn independently from a normal distribution $N(0, \sigma^2)$.

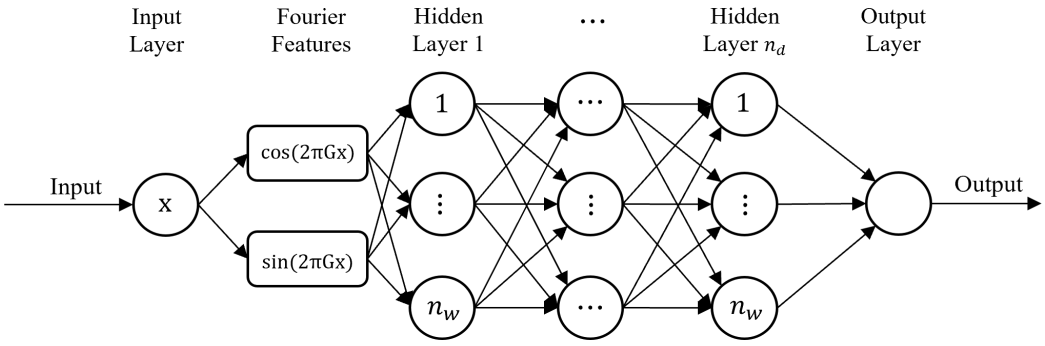

Fig. 3.2.1 PINN with random Fourier features.

We carry out the numerical experiment to study the effect of such an improvement by considering a simple equation

$$y''(x) = -(n\pi)^2 y, \tag{3.2.2}$$

with Dirichlet boundary conditions imposed over the interval $[0, 1]$ so that the solution is given by $y = \sin(n\pi x)$. The results are summarized in Fig 3.2.1.

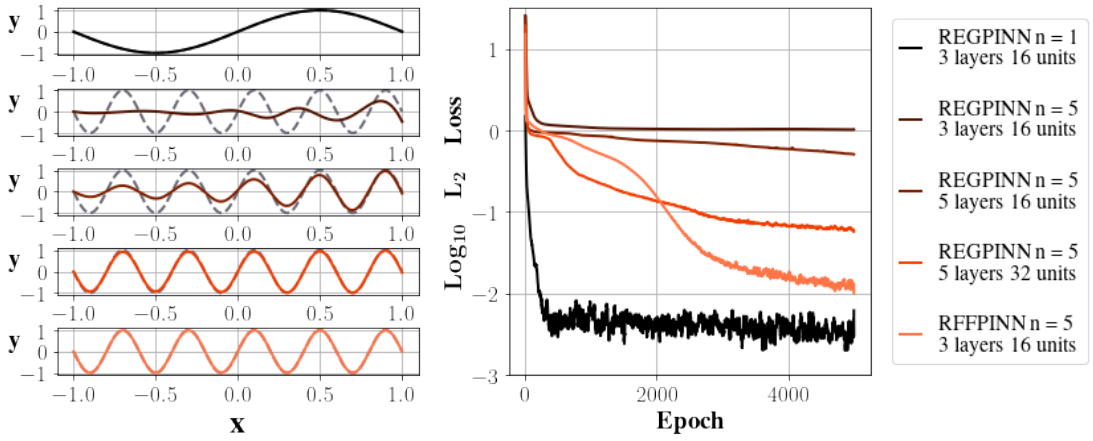

Fig. 3.2.1 Apply PINNs to high-frequency functions. REGPINN stands for regular PINNs, and RFFPINN represents PINNs with random Fourier features.

The experiment mainly reveals several key observations. First, we see the network converges much faster for y with a lower frequency when $n = 1$. To achieve convergence for such a function, only a regular PINN with a structure of 3 layers and one hidden layer containing 16 neurons is needed, which in all provides

$$(1 + 1) \times 16 + (16 + 1) \times 1 = 49$$

trainable parameters. However, the structure soon becomes inexpressive for $n = 5$ as the function's frequency increases. We see the regular network starts to become expressive again when two more hidden layers are added (5 layers in total) to the architecture, and when the number of neurons for each hidden layer is increased to 32. The number of trainable parameters now becomes

$$(1 + 1) \times 32 + (32 + 1) \times 32 \times 2 + (32 + 1) \times 1 = 2209.$$

In comparison, over two thousand new parameters are added to the model to strengthen the expressiveness of the network, while the frequency only increases from 0.5 to 2.5. Luckily, the situation can be alleviated by adding Fourier features to the network structure as displayed in Fig. 3.2.1. With a Gaussian matrix G sampled from $\mathbb{R}^{8\times1}$, the error of the network's output quickly converges to a scale of $\mathcal{O}(10^{-2})$ for a 3-layer network with a hidden layer of width 16. This time, only

$$(2 \times 8 + 1) \times 16 + (16 + 1) \times 1 - 49 = 240$$

new trainable parameters are needed to strengthen the network expressiveness.

4 Solving Partial Differential Equations

4.1 2D Elliptic Equation

In this section, we apply the PINNs to solving 2D Laplace equations with Dirichlet boundary conditions over rectangular regions $\Omega = [a, b] \times [c, d]$,

$$u_{xx}(x, y) + u_{yy}(x, y) = f(x, y) \quad \text{for} \quad (x, y) \in \Omega, \tag{4.1.1}$$

and

$$u(a, y) = u_a(y), \qquad u(b, y) = u_b(y), \tag{4.1.2}$$
$$u(x, c) = u_c(x), \qquad u(x, d) = u_d(x). \tag{4.1.3}$$

To impose hard boundary conditions, the solution ansatz is set as

$$\begin{aligned}
\varphi_\theta(x, y) \;=\; & \hat{\varphi}_\theta(x, y) + \frac{b-x}{b-a} \cdot [u_a(y) - \hat{\varphi}_\theta(a, y)] + \frac{x-a}{b-a} \cdot [u_b(y) - \hat{\varphi}_\theta(b, y)] \\
& + \frac{d-y}{d-c} \cdot [u_c(x) - \hat{\varphi}_\theta(x, c)] + \frac{y-c}{d-c} \cdot [u_d(x) - \hat{\varphi}_\theta(x, d)] \\
& - \frac{(b-x)(d-y)}{(b-a)(d-c)} \cdot [u_a(c) - \hat{\varphi}_\theta(a, c)] - \frac{(x-a)(d-y)}{(b-a)(d-c)} \cdot [u_b(c) - \hat{\varphi}_\theta(b, c)] \\
& - \frac{(b-x)(y-c)}{(b-a)(d-c)} \cdot [u_a(d) - \hat{\varphi}_\theta(a, d)] - \frac{(x-a)(y-c)}{(b-a)(d-c)} \cdot [u_b(d) - \hat{\varphi}_\theta(b, d)],
\end{aligned} \tag{4.1.4}$$

where $\varphi_\theta(x, y)$ is the modified network output. The loss function for this model is

$$\mathcal{L}(\theta) = \frac{1}{N_p} \sum_{i=1}^{N_p} \left\| \frac{\partial^2 \varphi_\theta}{\partial x^2}(x_i, y_i) + \frac{\partial^2 \varphi_\theta}{\partial y^2}(x_i, y_i) - f(x_i, y_i) \right\|^2. \tag{4.1.5}$$

We test the method on the following sample equation over the region $\Omega = [0, 1] \times [0, 1]$,

$$u_{xx}(x, y) + u_{yy}(x, y) = -13\pi^2 \sin(2\pi x) \sin(3\pi y) \quad \text{for} \quad (x, y) \in \Omega, \tag{4.1.6}$$
$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0 \quad \text{for} \quad (x, y) \in \partial\Omega. \tag{4.1.7}$$

The solution is given by

$$u_{\text{true}}(x, y) = \sin(2\pi x) \sin(3\pi y). \tag{4.1.8}$$

We trained a 3-layer network over 2000 sampled points with a batch size of 32 using the Adam optimizer to obtain the solution of the equation (4.1.6 - 7); the loss changes are tracked and recorded in the right column of Fig. 4.1.1. The L_2 loss here is given by

$$L_2(\theta) = \left(\int_0^1 \int_0^1 \left\| \varphi_\theta(x, y) - u_{\text{true}}(x, y) \right\|^2 \mathrm{d}x\mathrm{d}y \right)^{1/2}. \tag{4.1.9}$$

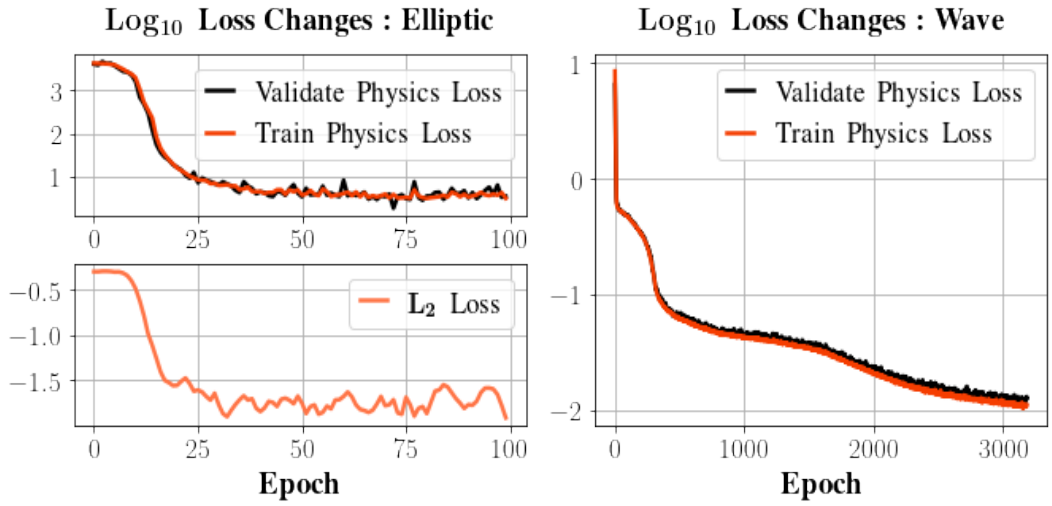

Fig. 4.1.1 The plots on the left column give the changes in both the physics loss and the L_2 loss while training the PINN to solve the sample elliptic equation (4.1.6 - 7); The plots on the right column give the changes in physics losses while training the network to solve the sample wave equation in Section 4.2.

The output of the PINN and its error corresponding to the true solution is presented in Fig. 4.1.2, together with section L_2 errors defined as

$$E(x) = \left(\int_0^1 \left\| \varphi_\theta(x, y) - u_{\text{true}}(x, y) \right\|^2 \, \mathrm{d}y \right)^{1/2}, \tag{4.1.10}$$

$$E(y) = \left(\int_0^1 \left\| \varphi_\theta(x, y) - u_{\text{true}}(x, y) \right\|^2 \, \mathrm{d}x \right)^{1/2}. \tag{4.1.11}$$

For most of the points, the absolute error is controlled within a scale of $\mathcal{O}(10^{-2})$, indicating a relatively good approximation.

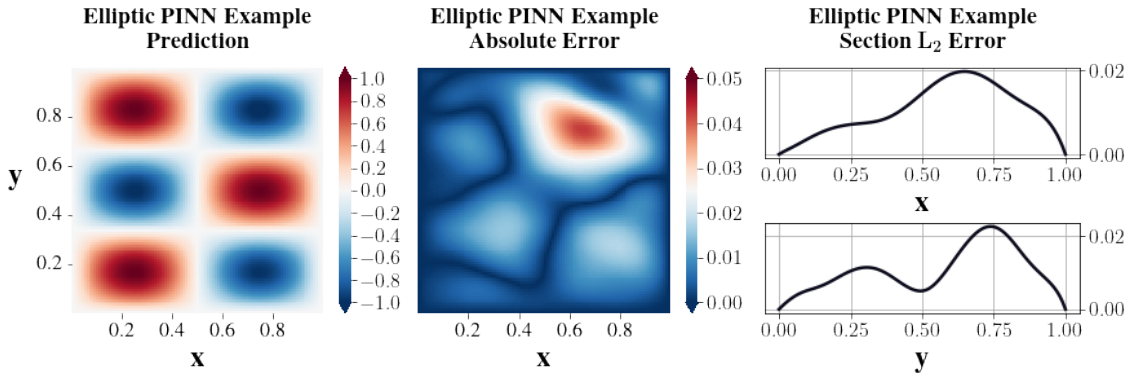

Fig. 4.1.2 The solution of sample elliptic equation (4.1.6 - 7) generated by PINN in plot 1 with prediction error displayed in plot 2 and 3.

4.2 3D Wave Equation

The method is also generalized to solve 3D wave equations with 2 spacial dimensions. Consider solving the equation for $u(x, y, t)$

$$u_{tt} = c^2(u_{xx} + u_{yy}) \quad \text{for} \quad c > 0, \tag{4.2.1}$$

over a target rectangular region $\Omega = [0, a] \times [0, b] \times [0, T]$, where $(x, y, t) \in \Omega$ with homogeneous boundary conditions

$$u(0, y, t) = u(a, y, t) = u(x, 0, t) = u(x, b, t) = 0, \tag{4.2.2}$$

and initial conditions

$$u(x, y, 0) = g(x, y) \quad \text{and} \quad u_t(x, y, 0) = \phi(x, y). \tag{4.2.3}$$

The analytical solution to this problem can be obtained using separation of variables and Fourier series expansions

$$u_{\text{true}}(x, y, t) = \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} \sin(\mu_m x) \sin(\nu_n y)[A_{mn} \cos(\lambda_{mn} t) + B_{mn} \sin(\lambda_{mn} t)], \tag{4.2.4}$$

where $\mu_m = \frac{m\pi}{a}$, $\nu_n = \frac{n\pi}{b}$, $\lambda_{mn} = c\sqrt{\mu_m^2 + \nu_n^2}$, and

$$A_{mn} = \frac{4}{ab} \int_0^a \int_0^b g(x, y) \sin(\mu_m x) \sin(\nu_n y) \, \mathrm{d}y\mathrm{d}x, \tag{4.2.5}$$

$$B_{mn} = \frac{4}{ab\lambda_{mn}} \int_0^a \int_0^b \phi(x, y) \sin(\mu_m x) \sin(\nu_n y) \, \mathrm{d}y\mathrm{d}x. \tag{4.2.6}$$

This time, the conditions are applied softly to train the network $\varphi_\theta(x, y, t)$, with a loss function

$$\begin{aligned}
\mathcal{L}(\theta) = \frac{\lambda_0}{N_p} \sum_{i=1}^{N_p} \left\| \frac{\partial^2 \varphi_\theta}{\partial t^2}(x_i, y_i, t_i) - c^2 \left(\frac{\partial^2 \varphi_\theta}{\partial x^2} + \frac{\partial^2 \varphi_\theta}{\partial y^2} \right)(x_i, y_i, t_i) \right\|^2 \\
+ \frac{\lambda_1}{N_{b,1}} \sum_{j=1}^{N_{b,1}} \left\| \varphi_\theta(0, y_{1,j}, t_{1,j}) \right\|^2 + \frac{\lambda_2}{N_{b,2}} \sum_{j=1}^{N_{b,2}} \left\| \varphi_\theta(a, y_{2,j}, t_{2,j}) \right\|^2 \\
+ \frac{\lambda_3}{N_{b,3}} \sum_{j=1}^{N_{b,3}} \left\| \varphi_\theta(x_{3,j}, 0, t_{3,j}) \right\|^2 + \frac{\lambda_4}{N_{b,4}} \sum_{j=1}^{N_{b,4}} \left\| \varphi_\theta(x_{4,j}, b, t_{4,j}) \right\|^2 \\
+ \frac{\lambda_5}{N_{b,5}} \sum_{j=1}^{N_{b,5}} \left\| \varphi_\theta(x_{5,j}, y_{5,j}, 0) - g(x_{5,j}, y_{5,j}) \right\|^2 \\
+ \frac{\lambda_6}{N_{b,6}} \sum_{j=1}^{N_{b,6}} \left\| \frac{\partial \varphi_\theta}{\partial t}(x_{6,j}, y_{6,j}, 0) - \phi(x_{6,j}, y_{6,j}) \right\|^2.
\end{aligned} \tag{4.2.7}$$

Specifically, we solved the equation for $a = b = c = T = 1$ and trained the network with $\lambda_0 = 5$ and $\lambda_k = 20$ for $k = 1, \cdots, 6$. 2500 points from each of the five boundaries and the interior of the region were sampled to form a training data set of size 15000. The training loss changes were recorded in Fig. 4.1.1, and the predictions of the PINN are given in Fig. 4.2.1. Again, the scale of the absolute errors is approximately $\mathcal{O}(10^{-2})$.

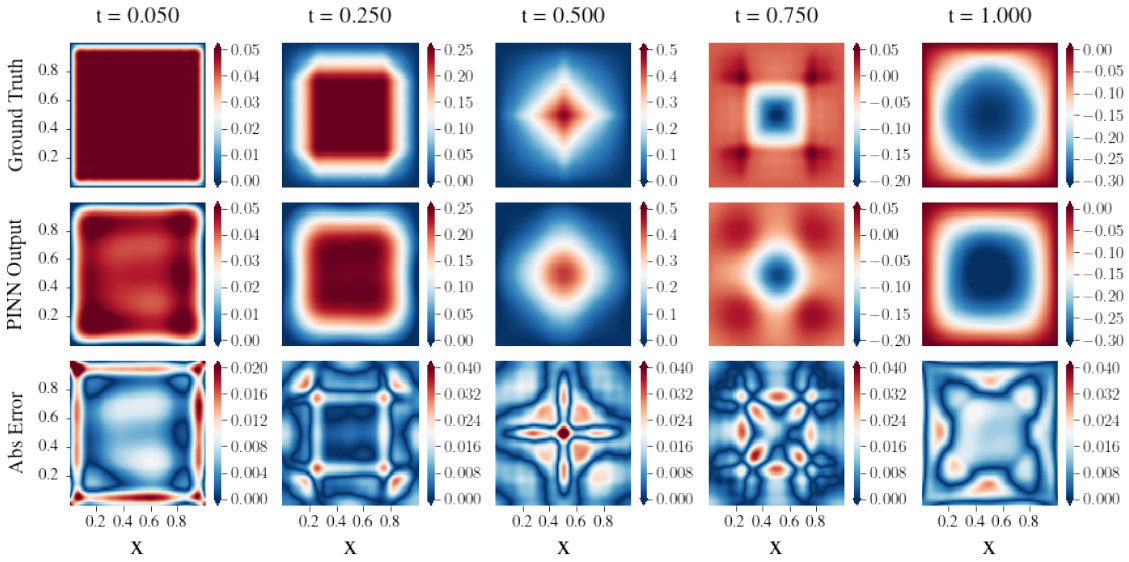

Fig. 4.2.1 The solution of the sample wave equation generated by PINN with absolute errors displayed in the third row.

5 Conclusion

This paper mainly discusses the application of physics-informed neural networks, i.e., PINNs, to solving differential equations. We first apply the method to ordinary differential equations and ordinary differential equation systems under different types of boundary conditions and find that the solution generated tends to converge slower for high-frequency functions. We then fix the problem and improve the convergence pattern for networks fitting high-frequency solutions by adding random Fourier features to the network structure. Finally, we generalize our work to study the behaviour of PINNs over partial differential equations with two and three independent variables. The general absolute errors between the network output and the ground truth are controlled within a scale of $\mathcal{O}(10^{-2})$, indicating an overall accurate prediction.

Finally, there are still many aspects that our study does not cover. For example, most of our example networks are trained under hard conditions. Nevertheless, finding a corresponding solution ansatz can be challenging for complex boundary conditions.

Clearly, employing soft conditions is a more versatile approach compared with the one we use, let alone we can also include lambdas as part of the learnable parameters to achieve a more adaptive strategy [8]. Also, for more complicated practical problems, PINNs usually suffer from problems of high computational cost and slow convergence. For the former, one possible solution is to use conditioned PINNs, where the initial and boundary conditions, as well as other possible features of the equation, are also considered as part of the network inputs to avoid retraining of similar models [10, 15]. For the latter, possible directions may be either employing more advanced network architectures, for instance, using convolutional layers to help networks better understand the structure of the solution [17], or by modifying the training strategies such as letting networks prioritize learning sample points with higher loss values [16] and carrying out the training process under decomposed domains with finite basis [9].

6 References

[1] J. Adler and O. Öktem. "Solving ill-posed inverse problems using iterative deep neural networks". In: *Inverse Problems* 33.12 (Nov. 2017), p. 124007. ISSN: 1361-6420. DOI: 10.1088/1361-6420/aa9581. URL: http://dx.doi.org/10.1088/1361-6420/aa9581 (visited on 05/04/2024).

[2] A.G. Baydin et al. "Automatic differentiation in machine learning: A survey". In: *Journal of Machine Learning Research* 18 (Apr. 2018), pp. 1–43.

[3] J. Berg and K. Nyström. "A unified deep artificial neural network approach to partial differential equations in complex geometries". In: *ArXiv* abs/1711.06464 (2017). URL: https://api.semanticscholar.org/CorpusID:38319575 (visited on 05/04/2024).

[4] Z. Chen, Y. Liu, and H. Sun. "Physics-informed learning of governing equations from scarce data". In: *Nature Communications* 12 (2020). URL: https://api.semanticscholar.org/CorpusID:239455737 (visited on 05/04/2024).

[5] W. Ee and B. Yu. "The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems". In: *Communications in Mathematics and Statistics* 6 (2017), pp. 1–12. URL: https://api.semanticscholar.org/CorpusID:2988078 (visited on 05/04/2024).

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

[7] D.P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[8] L.D. McClenny and U.M. Braga-Neto. "Self-adaptive physics-informed neural networks". In: *Journal of Computational Physics* 474 (2023), p. 111722.

[9] B. Moseley, A. Markham, and T. Nissen-Meyer. *Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations*. 2021. arXiv: 2107.07871 [physics.comp-ph].

[10] B. Moseley, A. Markham, and T. Nissen-Meyer. *Solving the wave equation with physics-informed deep learning*. June 2020. URL: https://arxiv.org/abs/2006.11894 (visited on 05/05/2024).

[11] *PyTorch autograd.grad*. URL: https://pytorch.org/docs/stable/generated/torch.autograd.grad.html (visited on 05/04/2024).

[12] N. Rahaman et al. "On the spectral bias of neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 5301–5310.

[13] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations". In: *ArXiv* abs/1711.10561 (2017). URL: https://api.semanticscholar.org/CorpusID:394392 (visited on 05/04/2024).

[14] M. Tancik et al. "Fourier features let networks learn high frequency functions in low dimensional domains". In: NIPS '20. , Vancouver, BC, Canada, Curran Associates Inc., 2020. ISBN: 9781713829546.

[15] S. Wang, H. Wang, and P. Perdikaris. "Learning the solution operator of parametric partial differential equations with physics-informed DeepONets". In: *Science Advances* 7 (Sept. 2021). DOI: 10.1126/sciadv.abi8605.

[16] C. Wu et al. "A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks". In: *Computer Methods in Applied Mechanics and Engineering* 403 (Jan. 2023), p. 115671. DOI: 10.1016/j.cma.2022.115671.

[17] Y. Zhu et al. "Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data". In: *Journal of Computational Physics* 394 (2019), pp. 56–81. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2019.05.024. URL: https://www.sciencedirect.com/science/article/pii/S0021999119303559 (visited on 05/05/2024).

A Appendix

A.1 Codes

All codes can be found via: https://github.com/abaaba337/MMSC-Computing-Case-Study-PINN.