King Fahd University of Petroleum and Minerals

College of Computing and Mathematics

Mathematics Department

Math 619-Project

Term 232

Final Report

Project: Deep Learning Methods for Partial Differential Equations (PDEs)

Name	KFUPM ID
Hashim Al-Sadah	201578370
Abdulwahab Alghamdi	201734070
Hussain Al-Sinan	202205120

Instructor: Dr. Jamal Al-Smail

Abstract

Partial differential equations (PDEs) are essential components for modelling different processes and systems in various scientific and engineering areas. To predict the behavior of a certain system, one needs to solve or simulate the PDEs that describe that system. However, obtaining an analytical solution to PDEs is a difficult task in most practical situations, especially in the case of nonlinear or high dimensional PDEs. Due to the rapid evolution and advancements in the field of deep learning, researchers are starting to use deep neural networks (DNN) to approximate the solution of PDEs. Different models have been developed to perform this task such as Physics-Informed Neural Networks (PINNs) and Neural Operator. The speed and the efficiency of these models can surpass other common solvers such as Finite Elements (FM), Finite Difference (FD), and spectral methods in certain cases. In this project, we will explore different models of DNN, including their theoretical background and implementation. Furthermore, the models will be used to approximate the solution of various differential equations.

1 Introduction

Due to the increased power of computation and the success of deep learning models in solving different problems in various fields, such as computer vision[3], pattern recognition[12], and natural language and speech processing[2], researchers were motivated to apply deep neural networks to solve problems in the field of scientific computing. One of the widely known and challenging problems is the problem of solving partial differential equations (PDEs). Different systems and processes are modeled by PDEs, whether it is a natural system such as modeling biological and physical phenomena[5, 1], or whether it is not a natural system such as simulating socioeconomic or financial models[1]. Deep learning-based PDEs solvers can surpass classical methods such as finite element or finite difference in certain cases[5]. One of them is the case of high-dimensional PDEs, where classical solvers require discretizing the PDEs' domain into a mesh, causing the number of computations to increase exponentially with the increase of the dimensions, and this is known as the curse. On the other hand, deep learning models are mesh-free, so they only require training data from the PDEs domain. Moreover, deep learning methods are easy to implement and do not require complicated mathematical analysis compared to traditional methods such as finite element method.[5].

Physics-informed Neural Network (PINN) is the most basic and widely used model for approximating PDEs' solutions. PINN is an unsupervised learning model since it does not require any labeled data to learn the approximated function or solution[4]. Despite being an unsupervised technique, it is possible to include experimental data, along with the physical prior knowledge in order to guide the neural network to the optimal approximation by reducing the number of admissible solutions to the problem[6, 11]. Similar to the fully connected neural network, the physics-informed neural network consists of an input layer, hidden layers, and an output layer. First, the input layer feeds the training data into the neural network. Then, the hidden layers map the data to higher dimensions through a series of linear transformations followed by a nonlinear activation function. Finally, the hidden layers map the data to the output layer, which provides the output of the approximated function. Physics-informed neural networks (PINNs) update the learning parameters by minimizing a loss function that takes into account the losses due to the boundary and initial conditions as well as the PDEs' residual. After the training process, the neural network should be able to approximate a function that obeys the laws provided by the PDEs. Therefore, the procedure of including the PDEs' residual helps to restrict the number of possible solutions[4].

Another famous model which has been developed recently is the Neural Operator model. What makes neural operators different from physics-informed neural networks (PINNs) is that the former learns a mapping between infinite function spaces, unlike PINN where the mapping occurs between finite spaces or sets. This enables neural operators to approximate an operator instead of a function, hence the name Neural Operator. The key difference between PINN and neural operators in terms of architecture is that the hidden layers in neural operators consist of linear operators, usually integral operators, followed by a nonlinear activation function. Therefore, neural operators are considered a generalization of physics-informed neural networks[9].

Overall, deep learning methods for PDEs are powerful solvers. They have the potential to resolve various problems and challenges faced by the classical methods. The advantage of neural network-based models is that they are mesh-free methods, which enables them to approximate the solution for nonlinear, non-smooth, and high dimensional PDEs without suffering from the cures of dimensionality. Furthermore, they utilize the automatic differentiation algorithm to compute the residual of the PDEs. The algorithm is implemented by the majority of deep learning libraries, which makes the overall implementation process simple and easy.

2 PINN

In this section, we develop some mathematical notation that we will use throughout the project, and discuss the procedure of approximating ordinary and partial differential equations. All the codes for the obtained results in this section will be available in the GitHub repository https://github.com/HashimAlSadah/MX-Project.git.

2.1 Overview

Let us first consider a general nonlinear differential equation

$$\begin{aligned} \mathcal{F}(u,\gamma)(\mathbf{x}) &= f(\mathbf{x}) \qquad \mathbf{x} \in \Omega, \\ \mathcal{B}(u,\gamma)(x,t) &= g(t) \qquad x \in \partial\Omega, \\ \mathcal{I}(u,\gamma)(x,t_0) &= h(x) \qquad x \in \partial\Omega_0 \end{aligned} \tag{1}$$

Where $u(\mathbf{x})$ is the unknown function that we are trying to approximate, $\mathbf{x} = [x_1, x_2, \ldots, x_{d-1}, t]$ is the vector of space and time in the domain $\Omega \subset \mathbb{R}^d$ with boundary $\partial\Omega$, γ are parameters related to the problem, $\mathcal{F}$ is the nonlinear differential operator, $\mathcal{B}$ is the boundary condition, $\mathcal{I}$ is the initial condition, and $f(\mathbf{x})$, $g(t)$, and $h(x)$ are specified functions for a certain problem.

Since it is established that multilayer feedforward neural networks are universal function approximators[7], this means that a neural network with at least one hidden layer can approximate a function from a finite dimensional set to another with any arbitrary error given that we use a sufficient number of hidden neurons or units.

2.2 The universal function approximator theorem

In this part, we explore the theoretical aspects of the universal approximator theorem. We will follow the definitions and the theorems stated in the paper by Hornik et al. (1988) and try to elaborate and explain as much as possible.

We start by introducing the necessary definitions before stating any theorem.

Definition 2.1. For any $r \in \mathbb{N}$, $\mathbf{A}^r$ is the set of all affine functions from $\mathbb{R}^r$ to $\mathbb{R}$. In other words, it is the set of all functions $A(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$. Where $\mathbf{x}, \mathbf{w} \in \mathbb{R}^r$ and $b \in \mathbb{R}$. $\square$

The affine transformation in Definition 2.1 represents the output of one hidden unit before applying any activation function. $\mathbf{x}$ represents the input to the neural network, $\mathbf{w}$ represents the weights between the input and a particular hidden unit, and b is the bias term.

Definition 2.2. For any Borel measurable function $G(\cdot)$ mapping $\mathbb{R}$ to $\mathbb{R}$ and $r \in \mathbb{N}$ define $\sum^r(G)$ to be the class of functions

$$\left\{ f : \mathbb{R}^r \to \mathbb{R} \,\middle|\, f(x) = \sum_{j=1}^{q} \beta_j G(A_j(\mathbf{x})), \mathbf{x} \in \mathbb{R}^r, \beta_j \in \mathbb{R}, \right. \\ \left. \mathbf{A}_j \in \mathbf{A}^r, \quad j = 1, 2, \ldots \right\} \square$$

When the function G is an activation function, the class of functions in Definition 2.2 becomes the class of neural network with one hidden layer and q hidden units. β_j are the weights from the hidden layer to the output of the neural network. Next, we define what a squashing (activation) function is.

Definition 2.3. For any Borel measurable function $G(\cdot)$ mapping from $\mathbb{R}$ to $\mathbb{R}$ and

$r \in \mathbb{N}$, define $\sum\prod^r(G)$ to be the class of functions

$$\left\{ f : \mathbb{R}^r \to \mathbb{R} \,\middle|\, f(\mathbf{x}) = \sum_{j=1}^{q} \beta_j \cdot \prod_{k=1}^{l_j} G(A_{jk}(\mathbf{x})),\ \mathbf{x}^r \in \mathbb{R}, \beta_j \in \mathbb{R}, \quad \mathbf{A}_j \in \mathbf{A}^r, \quad j = 1, 2, \dots \right\} \square$$

The class of functions defined in Definition 2.3 is a more general class of feedforward neural network, so proving that $\sum\prod$ is a class of universal functions approximators is sufficient to show that the same apply to the class $\sum$ since the latter is a special case of the earlier, which can be obtained by setting $l_j = 1$ for all q.

Definition 2.4. Define C^r to be the set of all continuous functions mapping from $\mathbb{R}^r$ to $\mathbb{R}$ and M^r to be the set of all measurable functions mapping from $\mathbb{R}^r$ to $\mathbb{R}$. Also, denote the Borel σ-field of $\mathbb{R}^r$ by B^r. $\square$

Therefore, by Definition 2.4, if G is any continuous function, then the classes $\sum^r(G)$ and $\sum\prod^r(G)$ belong to C^r. Similarly, if G is any Borel measurable function, then the classes $\sum^r(G)$ and $\sum\prod^r(G)$ belong to M^r.

Note: If a function is continuous, then it is measurable, but the converse is not true in general.

The closeness of functions $f, g \in C^r$ or $f, g \in M^r$ is measured by some metric ρ on C^r or M^r. The closeness of one class of functions to another is measured by the concept of denseness, which will be defined next.

Definition 2.5. A subset S of a metric space (X, ρ) is ρ-dense in a subset T if for every $\epsilon > 0$ and for every $t \in T$ there exists $s \in S$ such $\rho(s, t) < \epsilon$. $\square$

Definition 2.5 implies that an element of S can approximate an element of T to any desired degree of accuracy. In our case, X and T represent C^r or M^r, and S represents $\sum^r(G)$ or $\sum\prod^r(G)$.

Before introducing the main theorem, we need one last definition since we would like to show that $\sum^r(G)$ and $\sum\prod^r(G)$ are uniformly dense on a compacta in C^r or M^r.

Definition 2.6. A subset S of C^r is uniformly dense on a compacta in C^r if for every compact subset $K \subset \mathbb{R}^r$ S is ρ_K-dense in C^r. Where

$$\rho_K(f, g) \equiv sup_{x \in K} \left| f(x) - g(x) \right|, \quad \text{for all } f, g \in C^r$$

A sequence of functions f_n converges to a function f uniformly on compacta if for all compact $K \subset \mathbb{R}^r$ $\rho_K(f_n, f) \to 0$ as $n \to \infty$. $\square$

To prove that $\sum\prod(G)$, for any continuous non-constant G, is uniformly dense on compacta in C^r, we need to use Stone-Weierstrass theorem, which is the following.

Theorem 2.1 (Stone-Weierstrass theorem)**.** Let $\mathbf{A}$ be algebra of real continuous functions on a compact set K. If $\mathbf{A}$ separates points on K and $\mathbf{A}$ vanishes at no point of K, then the uniform closure $\mathbf{B}$ of $\mathbf{A}$ consists of all real continuous functions on K. That is, $\mathbf{A}$ is ρ_K-dense in the space of all real continuous on K. $\square$

Theorem 2.2. Let $G : \mathbb{R} \to \mathbb{R}$ be any continuous non-constant function. Then $\sum\prod^r(G)$ is uniformly dense on compacta in C^r. $\square$

Proof:

Let $K \subset \mathbb{R}^r$ be a compact set and apply Stone-Weierstrass theorem. This means that for any continuous non-constant G, $\sum\prod^r(G)$ is an algebra on K. If $\mathbf{x}, \mathbf{y} \in K$ and $\mathbf{x} \neq \mathbf{y}$, then there exists $A \in \mathbf{A}^r$ such that $G(A(\mathbf{x})) \neq G(A(\mathbf{y}))$.

To see this, choose $a, b \in \mathbb{R}$ such that $a \neq b$ and $G(a) \neq G(b)$. Also, choose $A(\cdot)$ such that $A(\mathbf{x}) = a$ and $A(\mathbf{y}) = b$. Then $G(A(\mathbf{x})) \neq G(A(\mathbf{y}))$, which ensures that $\sum\prod^r(G)$ is separating on K.

Next, there are $G(A(\cdot))$'s that are constant and not equal to zero. To see this, choose $b \in \mathbb{R}$ such that $G(b) \neq 0$ and set $A(\mathbf{x}) = \mathbf{0} \cdot \mathbf{x} + b$. Then for all $\mathbf{x} \in K$, $G(A(\mathbf{x})) = G(b)$, which ensures that $\sum\prod^r(G)$ vanishes at no point of K.

Therefore, Stone-Weierstrass theorem implies that $\sum\prod^r(G)$ is ρ_K-dense in the space of real continuous functions on K. $\square$

Theorem 2.2 implies that the class of feedforward neural network $\sum\prod$ is capable of approximating any continuous real-valued function on a compact set with any desired accuracy. We notice that the choice of the activation function does not influence the result as long as that function is continuous and non-constant.

Since Theorem 2.2 holds true for $\sum\prod$, then it also holds for the class of feedforward neural network $\sum$ as it is a special case of the earlier $(l_j = 1)$, which proves our claim that a shallow neural network with one layer is capable of approximating any real-valued function with an arbitrary degree of accuracy given a sufficient number of hidden units.

The paper by Hornik et al. (1988) extends the result beyond the approximation of any real-valued function to any measurable function. Additionally, they show that the same holds for a multilayer feedforward neural network. Refer to reference [7] for further details.

2.3 Training PINN

Following our notation in section 2.1 and applying Theorem 2.2, we can approximate the target function $u(\mathbf{x})$ by neural network,

$$u_\theta(\mathbf{x}) \approx u(\mathbf{x})$$

Where θ represents the parameters of the neural network, which consist of weights and biases.

As stated earlier, a neural network is composed of input layer, hidden layers, and output layer. Therefore, neural network with L layers can represent u_θ as a compositional function as the following.

$$u_\theta(\mathbf{x}) = f_L(\mathbf{x}) \circ f_{L-1}(\mathbf{x}) \circ \cdots \circ f_1(\mathbf{x})$$

But each layer consists of a linear transformation and a nonlinear scalar activation function

$$f_i(\mathbf{x}) = \sigma_i\left(\mathbf{W}_i \cdot \mathbf{x}_i + \mathbf{b}_i\right),$$

Denote the linear transformation by $T(\mathbf{x})$

$$T(\mathbf{x}) = \mathbf{W}_i \cdot \mathbf{x}_i + \mathbf{b}_i$$

Therefore,

$$u_\theta(\mathbf{x}) = T_L \circ \sigma \circ T_{L-1} \circ \cdots \circ \sigma \circ T_1 \tag{2}$$

We are assuming that the same activation function σ is used for all the layers.

PINN converts the problem of solving a differential equation to an optimization problem since PINN attempts to determine the parameters θ that approximate the function u by minimizing a loss function $\mathcal{L}(\theta)$[4].

The loss function includes the partial differential equation, initial and boundary conditions, and the labeled data if available.

$$\mathcal{L}(\theta) = w_\mathcal{F}\mathcal{L}_\mathcal{F}(\theta) + w_\mathcal{B}\mathcal{L}_\mathcal{B}(\theta) + w_\mathcal{I}\mathcal{L}_\mathcal{I}(\theta) + w_\mathcal{D}\mathcal{L}_\mathcal{D}(\theta) \tag{3}$$

Where $\mathcal{L}_\mathcal{F}(\theta)$, $\mathcal{L}_\mathcal{B}(\theta)$, $\mathcal{L}_\mathcal{I}(\theta)$, and $\mathcal{L}_\mathcal{D}(\theta)$ are the differential equation loss, boundary conditions loss, initial conditions loss, and labeled data loss, respectively, and $w_\mathcal{F}$, $w_\mathcal{B}$, $w_\mathcal{I}$, and $w_\mathcal{D}$ are their corresponding weights. The weights for the losses are usually hyperparameters specified manually.

Considering the loss function to be the mean square error function, then the losses can be defined as the following.

$$\mathcal{L}_\mathcal{F}(\theta) = MSE_\mathcal{F} = \frac{1}{N_c}\sum_{i=1}^{N_c}\left|\mathcal{F}(u_\theta)(\mathbf{x}_i) - f(\mathbf{x}_i)\right|^2 \tag{4}$$

Where N_c are the collocation points, which are points within the domain Ω

(4) is formulated as above since we are requiring the left-hand side of

$$\mathcal{F}(u)(\mathbf{x}) = f(\mathbf{x}) \Rightarrow \mathcal{F}(u)(\mathbf{x}) - f(\mathbf{x}) = 0$$

to be zero.

Similarly, for $\mathcal{L}_{\mathcal{B}}(\theta)$, $\mathcal{L}_{\mathcal{I}}(\theta)$, and $\mathcal{L}_{\mathcal{D}}(\theta)$.

$$\mathcal{L}_{\mathcal{B}}(\theta) = MSE_{\mathcal{B}} = \frac{1}{N_b}\sum_{j=1}^{N_b}\left|\mathcal{B}(u_\theta)(x,t_j) - g(t_j)\right|^2 \tag{5}$$

Note: (5) shows the boundary condition loss for one point in space x at different time t_j.

$$\mathcal{L}_{\mathcal{I}}(\theta) = MSE_{\mathcal{I}} = \frac{1}{N_i}\sum_{i=1}^{N_i}\left|\mathcal{I}(u_\theta)(x_i,t_0) - h(x_i)\right|^2 \tag{6}$$

$$\mathcal{L}_{\mathcal{D}}(\theta) = MSE_{\mathcal{D}} = \frac{1}{N_d}\sum_{i=1}^{N_d}\left|u_\theta(\mathbf{x}_i) - u_i\right|^2 \tag{7}$$

Where u_i is a labeled example or data.

2.4 Continuous model

2.4.1 Linear PDE example

To demonstrate the ability of PINN to approximate the solution of a PDE, we consider the following heat or diffusion equation.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - e^{-t}\sin(\pi x)(1-\pi^2), \quad -1 < x < 1, \quad 0 < t \leq 1 \tag{8}$$

Subject to the following boundary conditions

$$u(x=-1,t) = u(x=1,t) = 0 \tag{9}$$

and the initial condition

$$u(x,t=0) = \sin(\pi x) \tag{10}$$

The exact solution for the problem is

$$u(x,t) = e^{-t}\sin(\pi x), \quad -1 \leq x \leq 1, \quad 0 \leq t \leq 1 \tag{11}$$

To solve the problem, we have discretized the x-axis and the time into 30 points each. Therefore, our loss functions will be the following

$$\mathcal{L}_{\mathcal{F}} = \frac{1}{900}\sum_{i=1}^{30}\sum_{j=1}^{30}\left|\frac{\partial u_\theta}{\partial t}\bigg|_{x_i,t_j} - \frac{\partial^2 u_\theta}{\partial x^2}\bigg|_{x_i,t_j} + e^{-t_j}\sin(\pi x_i)(1-\pi^2)\right|^2$$

$$\mathcal{L}_{\mathcal{B}_1} = \frac{1}{30}\sum_{j=1}^{30}\left|u_\theta(x=-1,t_j)\right|^2, \qquad \mathcal{L}_{\mathcal{B}_2} = \frac{1}{30}\sum_{j=1}^{30}\left|u_\theta(x=1,t_j)\right|^2,$$

$$\mathcal{L}_{\mathcal{I}} = \frac{1}{30}\sum_{i=1}^{30}\left|u_\theta(x_j,t=0) - \sin(\pi x_j)\right|^2$$

The used neural network consists of 4 hidden layers and 10 hidden units or neurons. The learning rate was set to be 0.0001, and we used Adam optimizer, which utilizes the algorithm of stochastic gradient descent.

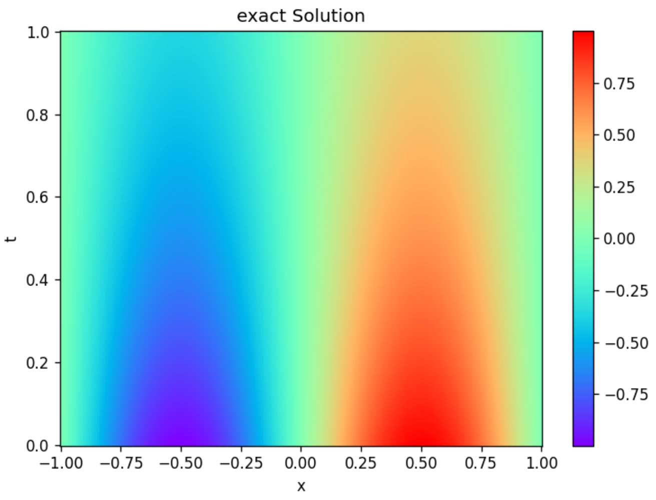

Figure 1: Exact solution $u(x,t)$

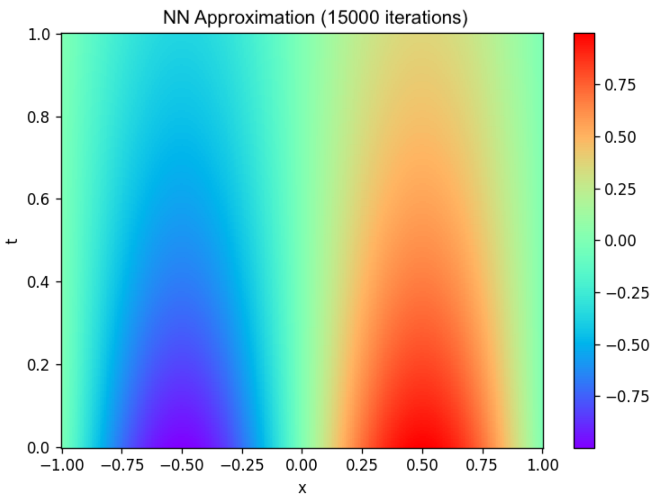

Figure 2: Approximated solution $u_\theta(x,t)$

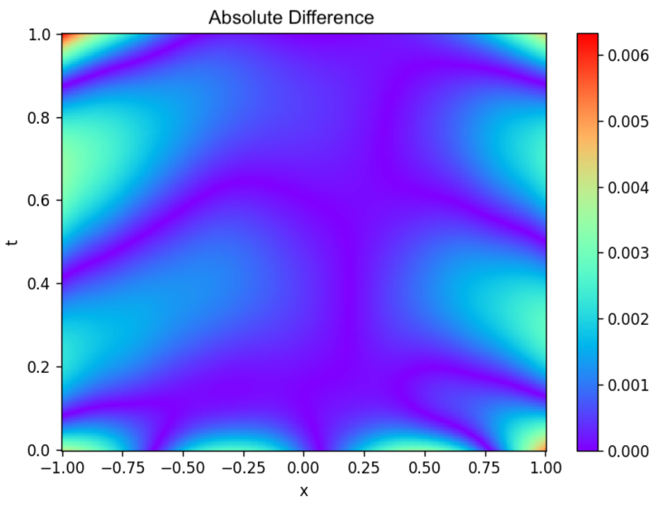

Figure 3: Absolute difference between $u_(x,t)$ and $u_\theta(x,t)$

Figure 3 shows the difference between the exact solution in Figure 1 and the neural network approximation in Figure 2 where it indicates a global error of order 10^{-2} after 15000 iterations. The code and the simulation details are on GitHub.

2.4.2 Nonlinear PDE example

Consider the following nonlinear PDE, Burgers' equation

$$\frac{\partial u(x,t)}{\partial t} = -u(x,t)\frac{\partial u(x,t)}{\partial x} + \nu\frac{\partial^2 u(x,t)}{\partial x^2}, \quad x \in [-1,1], \quad t \in [0,1] \tag{12}$$

Where ν is the diffusion coefficient. For our example, we will use $\nu = 0.01/\pi$.

The PDE is subject to the following initial and boundary conditions.

$$\begin{aligned} u(x,0) &= -sin(\pi x), \\ u(-1,t) &= u(1,t) = 0 \end{aligned} \tag{13}$$

Reference solution

It is possible to obtain the analytical solution for Burgers equation for a restricted set of initial conditions functions, $u(x,t=0) = g(x)$. This can be done by using the Cole-Hopf transformation, which is a nonlinear transformation[10].

$$u(x,t) = -2\nu\frac{\theta_x}{\theta} = -2\nu\log[\theta(x,t)] \tag{14}$$

This transformation turns the nonlinear Burgers PDE into the heat equation

$$\theta_t = \nu\theta_{xx} \tag{15}$$

For the purpose of making the code general when changing the initial conditions (or even the boundary conditions), we are going to use the explicit finite difference (conditionally stable) to approximate the solution for Burgers equation, and that approximation will be our reference solution.

Finite difference discretization

We will discretize the x-dimension into n sub-intervals with $n+1$ points using a step size of $\Delta x = h$

$$x_i = x_0 + ih, \quad x_0 = -1, \quad i = 0,1,2,\ldots,n.$$

The time will be discretized using a step of $\Delta t = k$.

$$t_j = t_0 + jk, \qquad t_0 = 0, \quad j = 0, 1, 2, \ldots$$

For the first derivative in time, we are going to use the forward difference (since it explicit). For the x-axis discretization, we are going to use the central difference for the first and the second derivative.

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{k}, \quad u_x \approx \frac{u_{i+1,j} - u_{i-1,j}}{2h}, \quad u_x \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

The discretized equation is the following

$$\frac{u_{i,j+1} - u_{i,j}}{k} = -u_{i,j}\left(\frac{u_{i+1,j} - u_{i-1,j}}{2h}\right) + \nu\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}\right) \tag{16}$$

Rearrange

$$u_{i,j+1} = u_{i,j} - ku_{i,j}\left(\frac{u_{i+1,j} - u_{i-1,j}}{2h}\right) + k\nu\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}\right)$$
$$\text{for} \quad i = 1, 2, \ldots, n-1, \quad j = 1, 2, \ldots \tag{17}$$

The initial and boundary points are the following

$$u_{i,0} = \sin(\pi x_i), \qquad u_{0,j} = u_{n,j} = 0$$

Figure 4 shows the finite difference approximation of burgers' equation.

Let $\tilde{u}(x,t)$ be the approximated solution by the neural network. Furthermore, rearrange Burgers equation and define the following function

$$f(x,t) = \tilde{u}_t + \tilde{u}\tilde{u}_x - \nu\tilde{u}_{xx} \tag{18}$$

Since we know that $f(x,t) = 0$, we can use this in our loss function, which is usually

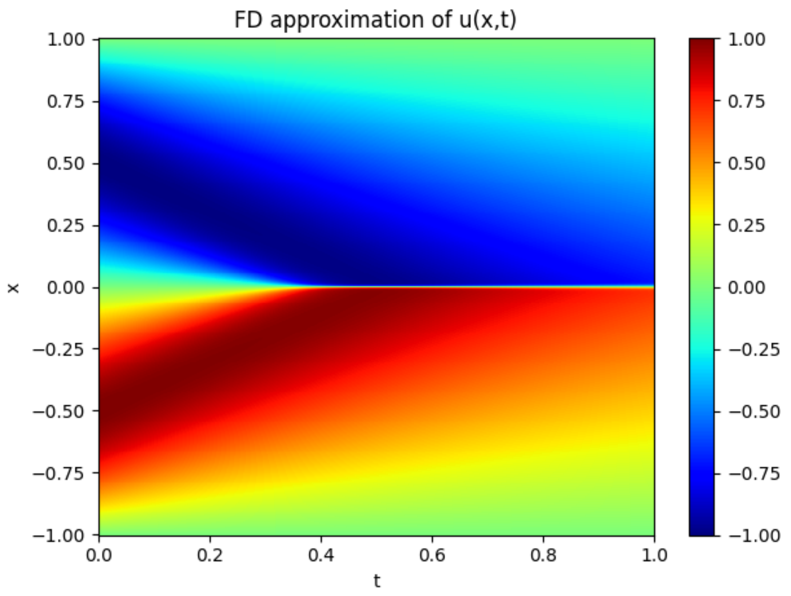

Figure 4: Finite difference approximation

taken as the mean square error between the predicted and the true values.

$$MSE_{PDE} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left| f(x_i, t_i) \right|^2 \tag{19}$$

Since $x \in [-1, 1]$ and $t \in [0, 1]$ then the collocation points are defined over the domain $(x_i, t_i) \in [-1, 1] \times [0, 1]$. N_f is the number of collocation points.

Similarly, we can define the loss due to the initial and boundary points.

$$MSE_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \left| \tilde{u}(x_i, t = 0) + \sin(\pi x_i) \right|^2 \tag{20}$$

Where N_0 is the number of initial points.

$$
\begin{aligned}
MSE_{lb} &= \frac{1}{N_{lb}} \sum_{i=1}^{N_{lb}} \left| \tilde{u}(x=-1, t_i) \right|^2 \\
MSE_{rb} &= \frac{1}{N_{rb}} \sum_{i=1}^{N_{rb}} \left| \tilde{u}(x=1, t_i) \right|^2
\end{aligned} \tag{21}
$$

Where N_{lb} and N_{rb} are the number of left boundary and right boundary points, respectively.

The loss function that we need to minimize is the sum of PDE, initial, and boundary losses. Therefore,

$$
MSE = MSE_{PDE} + MES_{init} + MSE_{lb} + MSE_{rb} \tag{22}
$$

The neural network that was used to approximate the solution consists of 4 hidden layers with each layer containing 32 neurons. The total number of collocation or training points is $N_f = 2500$. 50 points from the spatial domain $[-1, 1]$ and 50 points from the temporal domain $[0, 1]$ and hence $N_f = 50 \times 50 = 2500$. Similar to the example of the diffusion problem, the optimizer that was used to minimize the loss function is the Adam optimizer with a learning rate of 10^{-4}. Figure 5 shows the model approximation after training the neural network for 20000 iterations and Figure 6 shows the absolute difference between the FD solution and the PINN approximation.

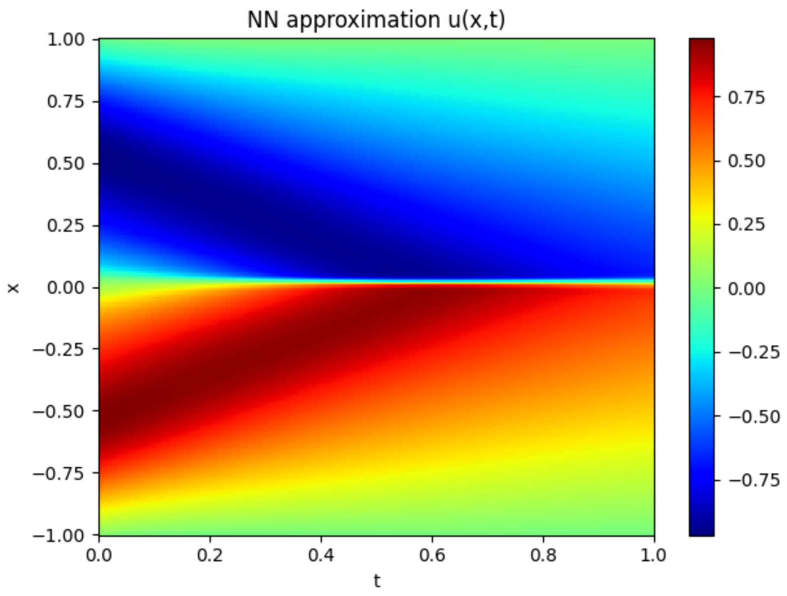

Figure 5: PINN approximation

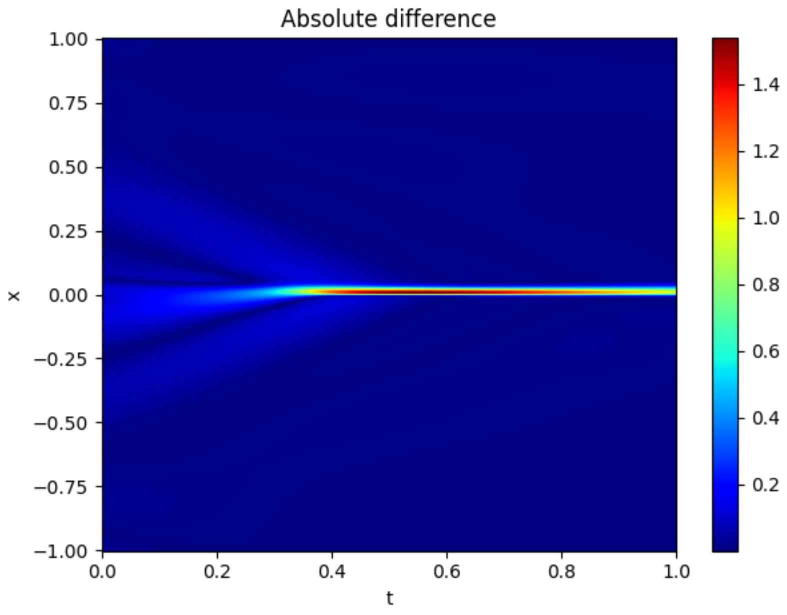

Figure 6: Absolute difference between FD and PINN

Since we have not used the exact solution as our reference solution, we used the mean square error to compare the neural network approximation with reference solution and the mean square error is approximately 0.0122. The code and the details of the simulation are on GitHub.

2.5 Discrete-time model

Instead of using the PDE immediately as we did in the case of the continuous model, we can perform a temporal discretization to the PDE using explicit/implicit Runge-Kutta method[11]. Let us consider the following general PDE.

$$\frac{\partial u(x,t)}{\partial t} = g[u(x,t)], \quad x \in \Omega, \quad t \in [0,T] \tag{23}$$

Where $u(x,t)$ is the desired solution, and $g[\cdot]$ is a spatial differential operator.

Next, we apply a general Runge-Kutta scheme with q stages as with the following Butcher's tableau

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1q} \\ c_2 & a_{21} & a_{22} & \dots & a_{2q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_q & a_{q1} & a_{q2} & \dots & a_{qq} \\ \hline & b_1 & b_2 & \dots & b_q \end{array}$$

This would result in the following equations as defined in [8].

$$\begin{aligned} u_{n+c_i} &= u_n + \Delta t \sum_{j=1}^{q} a_{ij}\, g[u_{n+c_j}] \qquad \text{for } i = 1, 2, \dots, q \\ u_{n+1} &= u_n + \Delta t \sum_{j=1}^{q} b_j\, g[u_{n+c_j}] \end{aligned} \tag{24}$$

Where

$$u_{n+c_j} = u(t + c_j \Delta t, x) \qquad \text{for} j = 1, 2, \dots, q$$

u_{n+c_j} are the Runge-Kutta solutions or the intermediate solution and u_{n+1} is the solution at the next step in time $u_{n+1} = u(t + \Delta t, x)$. The neural network will take training points from the spatial domain as an input and will output the intermediate solutions along with the solution at the next time step t_{n+1}. Therefore, the output layer will have $q+1$ neurons corresponding to the number of stages and the next solution.

$$NN_{out} \rightarrow [u_{n+c_1}(x), u_{n+c_2}(x), \dots,\ u_{n+c_q}(x), u_{n+1}(x)] \tag{25}$$

Given the approximated output of the neural network, we can compute the approximated previous solution u_n and the error between the approximated u_n and the exact u_n, which is known.

We can rearrange (24), so that we have our neural network output on one side and the previous solution on the other side.

$$\begin{aligned} u_n &= u_{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij}\, g[u_{n+c_j}] \qquad \text{for } i = 1, 2, \dots, q \\ u_n &= u_{n+1} - \Delta t \sum_{j=1}^{q} b_j\, g[u_{n+c_j}] \end{aligned} \tag{26}$$

Since the previous steps in (26) are the approximated, we will assign each one them a unique identifier as the following.

$$\begin{aligned} u_n^{(i)} &= u_{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij}\, g[u_{n+c_j}] \qquad \text{for } i = 1, 2, \dots, q \\ u_n^{(q+1)} &= u_{n+1} - \Delta t \sum_{j=1}^{q} b_j\, g[u_{n+c_j}] \end{aligned} \tag{27}$$

Therefore, the physics-informed output is

$$NN_{phys} \rightarrow [u_n^{(1)}, u_n^{(2)}, \dots, u_n^{(q)}, u_n^{(q+1)}] \tag{28}$$

Note: (25) is the neural network output without applying any equation and (28) is the physics-informed output since we are applying (27) to obtain (28).

For the purpose of implementation, it is better to write (27) using vectors and matrix notation.

$$\begin{bmatrix} u_n^{(1)} \\ u_n^{(2)} \\ \vdots \\ u_n^{(q)} \\ u_n^{(q+1)} \end{bmatrix} = \begin{bmatrix} u_{n+c_1} \\ u_{n+c_2} \\ \vdots \\ u_{n+c_q} \\ u_{n+1} \end{bmatrix} - \Delta t \begin{bmatrix} a_{11} & a_{12} & \dots & \dots & a_{1q} \\ a_{21} & a_{22} & \dots & \dots & a_{2q} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{q1} & a_{q2} & \dots & \dots & a_{qq} \\ b_1 & b_2 & \dots & \dots & b_q \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{q-1} \\ g_q \end{bmatrix} \tag{29}$$

Where

$$g_j = g[u_{n+c_j}], \qquad j = 1, 2, \dots, q$$

Finally, we define the loss function, which is the mean square error between the exact and the approximated solution of the previous step in time.

$$MSE_n = \frac{1}{N_x(q+1)} \sum_{j=1}^{N_x} \sum_{i=1}^{q+1} \left| u_n^{(i)}(x_j) - u_n(x_j) \right|^2 \tag{30}$$

Where N_x is the number of nodes or training points from the domain Ω and q is the number of stages of Runge-Kutta method.

Note: We have only defined a loss function for the inner points and not for the boundary points since we have not specified the geometry of the problem and the type of boundary conditions. However, the loss function for the boundary points MSE_b can be defined depending on the problem's specifications and the total loss will be $MSE = MSE_n + MSE_b$ as we will see in the following example.

After training the neural network or the model we should be able to predict the solution $u(x, t + \Delta t)$ at the next step in time t_{n+1}. To find the value of u at t_{n+2}, we need to train the neural network again using the approximation that we have acquired at t_n. In other words, we have to implement a time marching scheme.

An advantage of this method is that we could approximate the solution at the last time step in time T immediately if we use an implicit Runge-Kutta. An implicit Runge-Kutta scheme with large number of stages will have a larger stability region and hence one can use a greater step size to obtain the solution at the final step in one shot. The same advantage can be a limitation in certain cases where we would like to observe how the simulated system evolves in time, which may requires us to compute the solution using a small step. However, we can always use a time marching scheme to resolve this issue.

2.5.1 Linear PDE example

Consider the following diffusion equation

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2} - u(x,t), \quad 0 < x < 1, \quad 0 < t \leq 1 \tag{31}$$

Subject to the following boundary and initial conditions

$$u(x=0,t) = u(x=1,t) = 0, \quad t \geq 0 \tag{32}$$

$$u(x,t=0) = sin(\pi x), \quad x \in [0,1] \tag{33}$$

The exact solution is

$$u(x,t) = sin(\pi x)e^{-(\pi^2+1)t} \tag{34}$$

We write (31) in the form of (23) by defining the following differential operator

$$g[u(x,t)] = \frac{\partial^2 u(x,t)}{\partial x^2} - u(x,t) \tag{35}$$

For this problem, we are going to apply an implicit Runge-Kutta with three stages with following Butcher's tableau.

$$\begin{array}{c|ccc}
0 & \frac{1}{9} & \frac{-1-\sqrt{6}}{18} & \frac{-1+\sqrt{6}}{18} \\
\frac{3}{5} - \frac{\sqrt{6}}{10} & \frac{1}{9} & \frac{11}{45} + \frac{7\sqrt{6}}{360} & \frac{11}{45} - \frac{43\sqrt{6}}{360} \\
\frac{3}{5} + \frac{\sqrt{6}}{10} & \frac{1}{9} & \frac{11}{45} + \frac{43\sqrt{6}}{360} & \frac{11}{45} - \frac{7\sqrt{6}}{360} \\
\hline
 & \frac{1}{9} & \frac{4}{9} + \frac{\sqrt{6}}{26} & \frac{4}{9} - \frac{\sqrt{6}}{26}
\end{array} \tag{36}$$

Therefore, our linear system is

$$\begin{bmatrix} u_n^{(1)} \\ u_n^{(2)} \\ u_n^{(3)} \\ u_n^{(4)} \end{bmatrix} = \begin{bmatrix} u_{n+c_1} \\ u_{n+c_2} \\ u_{n+c_3} \\ u_{n+1} \end{bmatrix} - \Delta t \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} g[u_{n+c_1}] \\ g[u_{n+c_2}] \\ g[u_{n+c_3}] \end{bmatrix} \tag{37}$$

Where

$$\text{NN}_{out} = [u_{n+c_1}, u_{n+c_2}, u_{n+c_3}, u_{n+1}]$$

$$\text{NN}_{phys} = [u_n^{(1)}, u_n^{(2)}, u_n^{(3)}, u_n^(4)]$$

The loss function is defined as the following

$$MSE = MSE_n + MSE_b \tag{38}$$

MSE_n is the loss function for the points in the domain, $x \in (0, 1)$.

$$MSE_n = \frac{1}{N_x(q+1)} \sum_{j=1}^{N_x} \sum_{i=1}^{q+1} \left| u_n^{(i)}(x_j) - u_n(x_j) \right|^2 \tag{39}$$

MSE_b is the loss function for the boundary points, $x = 0$ and $x = 1$.

$$MSE_b = \frac{1}{q} \left(\sum_{i=1}^{q} \left| u_{n+c_i}(x=0) \right|^2 \right) + \left| u_{n+1}(x=0) \right|^2 + \frac{1}{q} \left(\sum_{i=1}^{q} \left| u_{n+c_i}(x=1) \right|^2 \right) + \left| u_{n+1}(x=1) \right|^2 \tag{40}$$

Where N_x is the number of training points and $q = 3$ is the number of Runge-Kutta stages.

The neural network consists of 4 hidden layers and 64 hidden layers, and the optimizer used to minimize the loss function is the limited-memory BFGS (LBFGS), which is quasi-Newton method since it approximates the Hessian matrix (the second partial derivative of the loss function). The time marching scheme applies a time step of

$\Delta t = 0.01$ to obtain a smooth approximation and the number of training points is $N_x = 200$.

Figure 7 shows the neural network approximation and the exact solution and Figure 8 shows the absolute difference between the exact and the approximated solutions, which indicates a global error of the order 10^3. The details of the simulation are on GitHub repository.

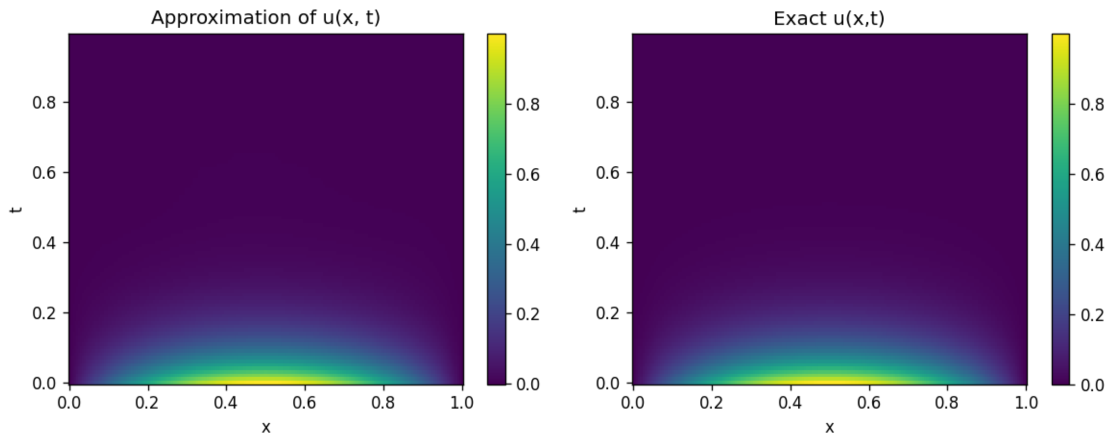

Figure 7: Approximation and Exact solution of $u(x, t)$

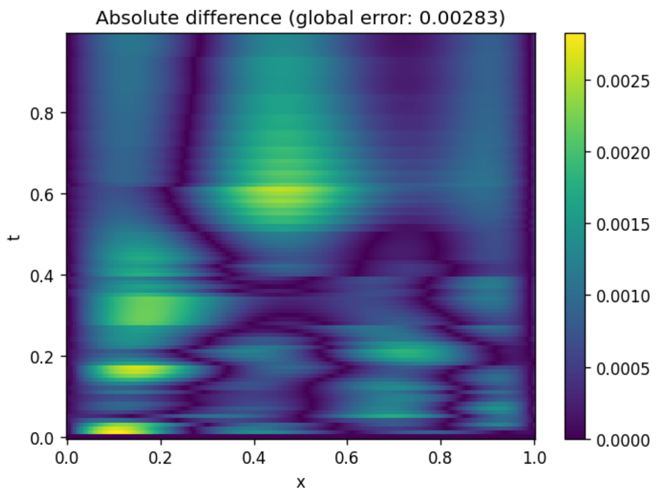

Figure 8: Absolute difference of $u_{exact} - u_{approx}$

3 PI-DeepONet

Physics Informed Deep Neural Operator (PI-DeepONet) is a DNN model for approximating the solutions of parametric partial differential equations (PDEs) where it incorporates physics informed knowledge in the architecture of the DeepONet model. DeepONet is a neural network architecture used to learn an operator between infinite dimensional function spaces.

This section provides a concise overview of the DeepONet model architecture [13], focusing specifically on the learning of solution operators for parametric Partial Differential Equations (PDEs). The term "parametric PDEs" refers to PDE systems where certain parameters are permitted to vary within a specified range. These parameters could include factors such as the shape of the physical domain, initial or boundary conditions, coefficients (constant or variable), and source terms. To address such problems comprehensively, let (U, V, S) be a triplet of Banach spaces, and $N : U \times S \to V$ be a differential operator, either linear or nonlinear. We examine parametric PDEs of the form $N(u, s) = 0$ [13] where $u \in U$ represents the parameters (input functions), and $s \in S$ is the corresponding unknown solution to the PDE system. It is assumed that for any $u \in U$, there exists a unique solution $s = s(u) \in U$ to the equation $N(u, s) = 0$ (subject to appropriate initial and boundary conditions). Consequently, a solution operator $G : U \to S$ is defined as $G(u) = s(u)$.

Following the original formulation by Lu et al. [13], we represent the solution map G using an unstacked DeepONet denoted as G_θ, where θ encompasses all trainable parameters of the DeepONet network. As depicted in Figure 9 DeepONet comprises two separate neural networks, known as the "branch net" and "trunk net", respectively.

The branch net takes u as input and produces a feature embedding $[b_1, b_2, ..., b_q]^T \in \mathbb{R}^q$ as output, where $u = [u(x_1), u(x_2), ..., u(x_m)]$ denotes a function $u \in U$ evaluated at a set of fixed locations $\{x_i\}_{i=1}^m$. On the other hand, the trunk net accepts continuous coordinates y as inputs and generates a feature embedding $[t_1, t_2, ..., t_q]^T \in \mathbb{R}^q$ as output. The final output of the DeepONet is obtained by merging the outputs of the branch and trunk networks via a dot product. Specifically, the DeepONet prediction $G_\theta(u)(y)$ for a function u evaluated at y is expressed as:

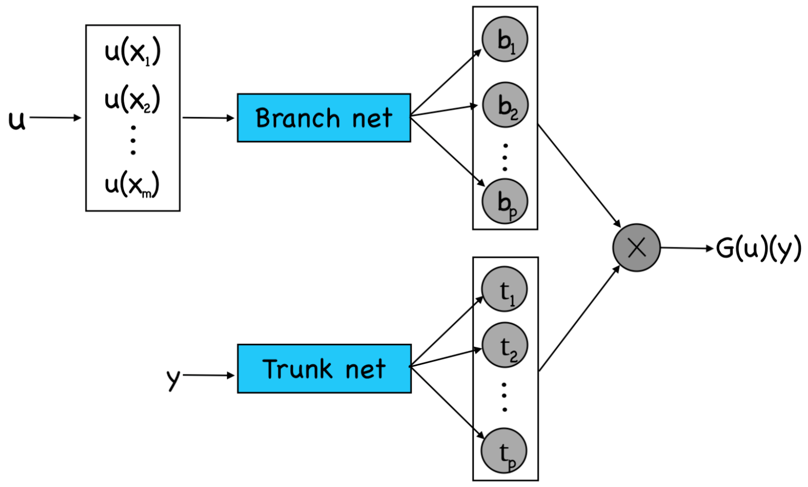

Figure 9: DeepONet architecture

$$G_\theta(u)(y) = \sum_{k=1}^{q} b_k(u(x_1), u(x_2), ..., u(x_m)) \cdot t_k(y), \tag{41}$$

where θ represents the collection of all trainable weight and bias parameters in the branch and trunk networks. These parameters are optimized by minimizing the mean square error loss given by:

$$\mathcal{L}_{ODE} = \frac{1}{m}\sum_{j=1}^{m}\left(-\frac{d^2 G_\theta(f)(p_j)}{dp^2} - f(p_j)\right)^2_{p=x}$$

Where $G_\theta(f) = u_\theta$ and p_j is an input point to the trunk NN.

3.1 IMPLEMENTATION

We will consider the 1D Poisson equation as a study case to illustrate the implementation of the PI-DeepONet method. The 1D Poisson equation is given by:

$$-\frac{d^2u}{dx^2} = f(x), \quad 0 \leq x \leq 1$$

Subject to the boundary conditions

$$u(0) = u(1) = 0$$

Where $f(x)$ is the forcing term. Our goal is to determine the function $u(x)$ for different forcing terms$f(x)$. In other words, we are trying to determine an operator G such that $G : f \to u$.

The operator G will be approximated by the branch neural network in the case of the DeepONet model, $G_\theta \approx G$. Where θ represents the parameter of the branch NN. The model will consist of two neural networks, the branch NN and the trunk NN.

3.1.1 Branch Net

In the branch net, we will consider the input to be the forcing term $f(x)$. The input function space will be limited to a set of random polynomials of degree 3 since it is not possible to implement an infinite function space. The space of polynomials of degree 3 is given by

$$\mathcal{P}^n(x) = a_0 + a_1x + a_2x^2 + \ldots\cdots + a_nx^n = \sum_{i=0}^{n} a_ix^i \tag{42}$$

Where $a_i \in \mathbb{R}$ and n is the degree of the polynomial.

The branch net will take the forcing term $f(x)$ as input and output the approximated function $u(x)$. In our work, we consider 100 different forcing terms $f(x)$ as the following

$$branch_{in} = \begin{pmatrix} f_1(x_1) & f_1(x_2) & f_1(x_3) & \ldots & f_1(x_{10}) \\ f_2(x_1) & f_2(x_2) & f_2(x_3) & \ldots & f_2(x_{10}) \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ f_{100}(x_1) & f_{100}(x_2) & f_{100}(x_3) & \ldots & f_{100}(x_{10}) \end{pmatrix} \tag{43}$$

Which yields an output of:

$$branch_{out} = \begin{pmatrix} u_1^{(1)} & u_2^{(1)} & u_3^{(1)} & \dots & u_m^{(1)} \\ u_1^{(2)} & u_2^{(2)} & u_3^{(2)} & \dots & u_m^{(2)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ u_1^{(100)} & u_2^{(100)} & u_3^{(100)} & \dots & u_m^{(100)} \end{pmatrix} \tag{44}$$

$$G_\theta : \begin{bmatrix} f_i(x_1) & f_i(x_2) & f_i(x_3) & \dots & f_i(x_{10}) \end{bmatrix} \to \begin{bmatrix} u_1^{(i)} & u_2^{(i)} & u_3^{(i)} & \dots & u_m^{(i)} \end{bmatrix} \tag{45}$$

for each $f_i(x)$ where $i = 1, 2, 3, \dots, 100$

$$u_i(x) = \begin{bmatrix} u_1^{(i)} & u_2^{(i)} & u_3^{(i)} & \dots & u_m^{(i)} \end{bmatrix} \tag{46}$$

$u_i(x)$ is not yet evaluated at any point. For the evaluation, we will use the output of the trunk neural network.

3.1.2 Trunk Net

The input to the trunk neural network are the points where we want to evaluate the target function $u(x)$ at, and we represent by the following column vector

$$trunk_{in} = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{100} \end{pmatrix} \tag{47}$$

Note: We used p instead of x, so there is no confusion with between these points and the sensor points.

Since $0 \leq x \leq 1$, then $0 \leq p_j \leq 1$ for $j = 1, 2, 3, \dots 100$

The trunk neural network will map each point to a higher dimension and in this case, we require that the output dimensions of the branch and the trunk networks must be the same. Therefore, the output of the trunk neural network is the following.

$$trunk_{out} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} & \dots & b_m^{(1)} \\ b_1^{(2)} & b_2^{(2)} & b_3^{(2)} & \dots & b_m^{(2)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ b_1^{(100)} & b_2^{(100)} & b_3^{(100)} & \dots & b_m^{(100)} \end{pmatrix} \tag{48}$$

For each p_j where $j = 1, 2, 3, \dots, 100$,

$$\mathbf{NN}_{\text{trunk}}(p_j) = \begin{bmatrix} b_1^{(j)} & b_2^{(j)} & b_3^{(j)} & \dots & b_m^{(j)} \end{bmatrix} \tag{49}$$

3.1.3 Evaluation

To evaluate the output u_i at the trunk points, we take the dot product of the branch output and the trunk output and that is why we require the output dimensions of both networks to be the same. For example, to evaluate $G(f_i)$ at p_j we perform the following operation

$$G_\theta(f_i)(p_j) = u_i(p_j) = \begin{bmatrix} u_1^{(i)} & u_2^{(i)} & u_3^{(i)} & \dots & u_m^{(i)} \end{bmatrix} \cdot \begin{bmatrix} b_1^{(j)} & b_2^{(j)} & b_3^{(j)} & \dots & b_m^{(j)} \end{bmatrix} = \sum_{n=1}^{m} u_n^{(i)} b_n^{(j)} \tag{50}$$

We are going to use only 1 hidden layer with 32 parameters for both the branch and trunk networks. Also, we will use LBFGS optimizer since it converges to the minimum faster.

3.2 RESULTS

We trained the model and are going to test it with $f(x)$ to be in the same space that we trained it on and on functions outside the domain to see how general the model is.

Case 1 - Constant

For $f(x) = -1$ which is a simple case with an exact solution $u(x) = \frac{1}{2}(x^2 - x)$ we get the following results

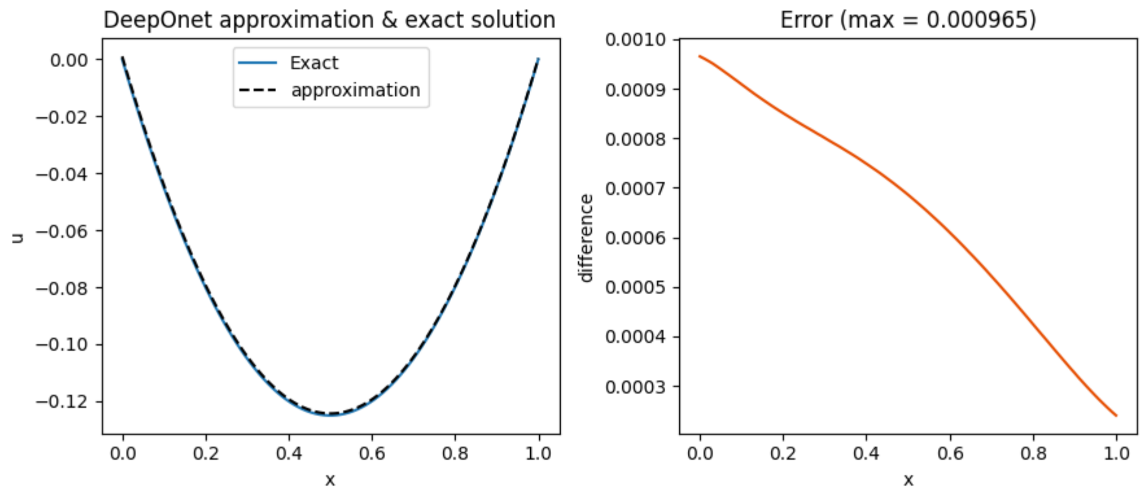

Figure 10: difference between the exact solution and the predicted solution for $f(x) = -1$

As we see in 3.2, the max error is very small, and the model was able to predict the solution very well.

Case 2 - Quadratic
In this case, we will consider the following forcing term

$$f(x) = x + 3x^2$$

Which has an exact solution of

$$u(x) = \frac{1}{6}x^3 - \frac{1}{4}x^4 + \frac{10}{24}x.$$

The results are shown in the following figure

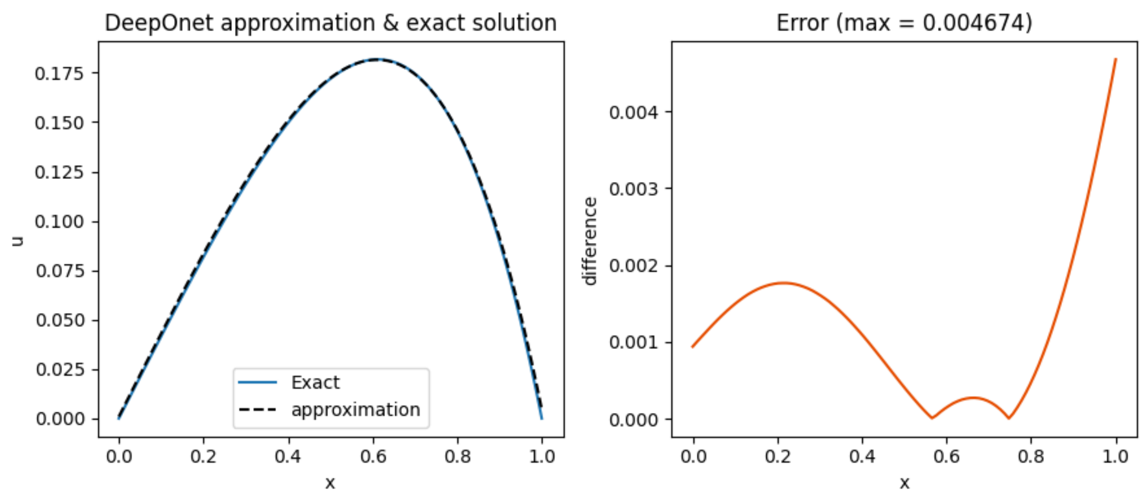

Figure 11: difference between the exact solution and the predicted solution for $f(x) = x + 3x^2$

Case 3 - Out of Space Functions (Exponential)

In this case, we will consider the following forcing term

$$f(x) = e^x$$

Which has an exact solution of

$$u(x) = -e^x + xe - x + 1$$

Since the forcing term is from a function space that the model hasn't been trained on, we expect the model to be less accurate. The results are shown in the following figure

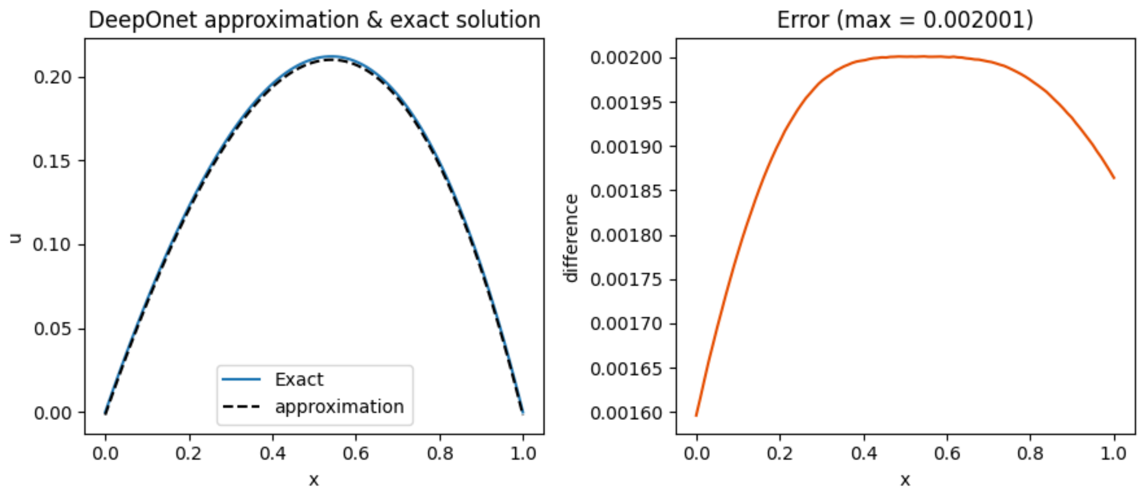

Figure 12: difference between the exact solution and the predicted solution for $f(x) = e^x$

4 Conclusion

The project shows the ability of Neural networks with different architecture to approximate the solutions of PDEs. We employed physics informed neural network (PINN) model to approximate the solutions for a nonlinear PDE (Burgers equation) and a linear PDE (diffusion equation), and the results match the reference solution with an acceptable accuracy. Furthermore, we implemented the two variations of PINN, which are the time-continuous model and the time-discrete model, where the latter utilizes Runge-Kutta to discretize the temporal part of the PDE and thus reducing the number of input data to the neural network. Finally, we looked into the physics informed DeepONet model, which approximates the solutions of parametric PDEs by learning a functional between two infinite dimensional function space. For this model, we only considered the time independent (one-dimensional) Poisson equation. Despite the promise that Deep Learning methods show for approximating PDEs, they have certain limitations, such as the need for a large amount of training points, especially for high dimensional and multi-physics problems, the low accuracy of the approximated solution, and the long training time for neural network model.

Future Work

Due to the time limit of this project we could not explore other interesting aspects of deep learning methods for approximating PDEs, so we suggest here few directions to continue the project.

- Using PINNs to solve a multi-physics problems such as the cavity flow problem represented by the Navier-Stocks equations or high dimensional ones like many-body problems (for example, the simulation of electron gas).
- Implement the DeepONet to solve parametric PDEs such as approximating the solution for burgers equation with different boundary or initial conditions.
- Using deep learning methods to approximate the solution of PDEs with irregular and complex geometry.
- Combining classical and deep learning approaches to create more robust methods.

References

[1] Christian Beck, Martin Hutzenthaler, Arnulf Jentzen, and Benno Kuckuck. An overview on deep learning-based approximation methods for partial differential equations. *arXiv preprint arXiv:2012.12348*, 2020.

[2] Junyi Chai and Anming Li. Deep learning in natural language processing: A state-of-the-art survey. In *2019 International Conference on Machine Learning and Cybernetics (ICMLC)*, pages 1–6, 2019.

[3] Junyi Chai, Hao Zeng, Anming Li, and Eric WT Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[4] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

[5] Tamara G Grossmann, Urszula Julia Komorowska, Jonas Latz, and Carola-Bibiane Schönlieb. Can physics-informed neural networks beat the finite element method? *arXiv preprint arXiv:2302.04107*, 2023.

[6] Zhongkai Hao, Songming Liu, Yichi Zhang, Chengyang Ying, Yao Feng, Hang Su, and Jun Zhu. Physics-informed machine learning: A survey on problems. *Methods and Applications. arXiv*, 2211, 2022.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[8] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[9] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

[10] SELÇUK Kutluay, AR Bahadir, and A Özdeş. Numerical solution of one-dimensional burgers equation: explicit and exact-explicit finite difference methods. *Journal of computational and applied mathematics*, 103(2):251–261, 1999.

[11] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[12] Joel Serey, Miguel Alfaro, Guillermo Fuertes, Manuel Vargas, Claudia Durán, Rodrigo Ternero, Ricardo Rivera, and Jorge Sabattin. Pattern recognition and deep learning technologies, enablers of industry 4.0, and their role in engineering research. *Symmetry*, 15(2):535, 2023.

[13] Sifan Wang, Hanwen Wang, and Paris Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *Science advances*, 7(40):eabi8605, 2021.