USING NEURAL NETWORKS TO SOLVE PARTIAL DIFFERENTIAL EQUATION

Joshua Joseph George

Course Supervisor: Professor Lili Mou

Department of Mathematical and Statistical Sciences
Department of Computing Science
University of Alberta
Edmonton, Alberta

Abstract. This project concerns the introduction of some techniques used to solve partial differential equations (PDEs) using neural networks. We provide some background, introduce the methods and the solutions obtained by each method, compare results, talk about limitations and lastly mention applications.

University of Alberta
E-mail address: jjgeorge@ualberta.ca.
Date: April 25, 2023.

Contents

1. Introduction

The topic of using neural networks to PDEs has gained significant attention in recent years due to its potential to provide accurate solutions for complex PDEs. We use the approach, known as physics-informed machine learning (PIML) [1] [2], which involves incorporating the governing PDE into the neural network architecture as a constraint. This allows for the network to learn the underlying physics of the system and provide accurate solutions.

2. Background

2.1. **PDEs.** A partial differential equation (PDE) is an equation of the form:

$$F(x_1, x_2, ..., x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, ..., \frac{\partial^2 u}{\partial x_1^2}, \frac{\partial^2 u}{\partial x_1 \partial x_2}, ..., \frac{\partial^2 u}{\partial x_n^2}, ...) = 0$$

where u is the unknown function of n independent variables $x_1, x_2, ..., x_n$, and F is a function of u and its partial derivatives with respect to the independent variables. PDEs are used to model a wide range of physical, biological, and social phenomena, from fluid dynamics to population dynamics to finance. One such example is the heat equation which is a PDE that describes the distribution of heat in a medium over time. It is mathematically given by

$$\partial_t u = \kappa \Delta u$$

where $k > 0$ is a constant, $u = u(t, x), x \in \mathbb{R}^3$ is the temperature function at time t and position x on a physical body, and Δ is called the Laplacian operator. Here Δu is

$$\Delta u = \partial_{x_1 x_1} u + \partial_{x_2 x_2} u + \partial_{x_3 x_3} u.$$

2.2. **Neural Networks.** A neural network is essentially a transformation that maps input data to output data. The function consists of multiple layers of neurons, which represent the basic computational unit that processes and transforms input information. The output of each neuron is determined by a weighted sum of the inputs, followed by the application of an activation function. The weights and biases of the neurons are learned through a process called back-propagation, which involves minimizing a loss function that measures the error between the predicted output and the actual output. The so-called activation functions are used to introduce non-linearity into the network, allowing it to model complex relationships between the input and output. Mathematically, the output from the neural network is

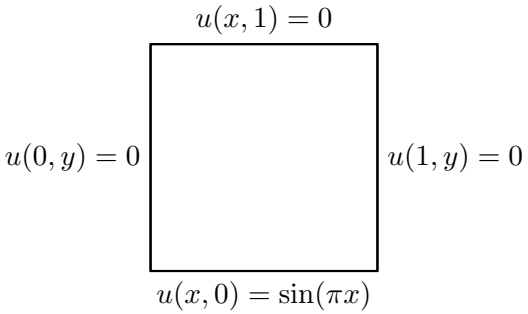
$$f_1 \left(f_2 \left(\dots f_{H-1} \left(f_H \left(\mathbf{x}\mathbf{W}^{(H)} \right) \mathbf{W}^{(H-1)} \right) \dots \right) \mathbf{W}^{(1)} \right)$$

where $\mathbf{x}$ is the input denoting each activation function $f_1, \dots, f_H$, ordered with f_1 as the output activation, and $p_1, \dots, p_{H-1}$ as the hidden dimensions with $H-1$ hidden layers and $\mathbf{W}^{(1)} \in \mathbb{R}^{p_1 \times m}, \mathbf{W}^{(2)} \in \mathbb{R}^{p_2 \times p_1}, \dots, \mathbf{W}^{(H)} \in \mathbb{R}^{d \times p_{H-1}}$ the weight matrices.

3. The Laplace Equation

For our project, we only study and solve the classic Laplace Equation $\Delta u = 0$ on the domain $(0,1) \times (0,1)$ with **Dirichlet** boundary conditions (BCs) $u(x,0) = \sin(\pi x), u(x,1) = u(0,y) = u(1,y) = 0$. This is written as,

$$\begin{cases} \Delta u = 0 \text{ on } \Omega : (0,1) \times (0,1) \\ u = u_0 \text{ on } \partial\Omega \end{cases} = \begin{cases} \Delta u = 0 \text{ on } \Omega : (0,1) \times (0,1) \\ u(x,0) = \sin(\pi x), \\ u(x,1) = u(0,y) = u(1,y) = 0 \end{cases}$$



We now present the analytic solution of the above PDE:

3.0.1. *Analytic Solution.* Consider $\Delta u = 0 \implies u_{xx} + u_{yy} = 0,\ 0 < x, y < 1$. Let $u(x, y) = X(x)Y(y)$. Substituting this into $u_{xx} + u_{yy} = 0$ gives us $X''Y + XY'' = 0 \implies \frac{X''}{X} + \frac{Y''}{Y} = 0$. If either of X or Y are zero then u is zero by definition but then it would not satisfy the BC $u(x, 0) = \sin(\pi x)$. Therefore let $\frac{X''}{X} = -\lambda$ and $\frac{Y''}{Y} = \lambda$ for some constant $\lambda \geq 0$. The sum $\frac{X''}{X} + \frac{Y''}{Y}$ is zero. Now consider $\frac{X''}{X} = -\lambda \implies X'' + \lambda X = 0$. By the BCs we have, $X'' + \lambda X = 0,\ X(0) = X(1) = 0$. The solution to the ODE: $X'' + \lambda X = 0$ is well known and it's just $A_n \cos(\sqrt{\lambda}x) + B_n \sin(\sqrt{\lambda}x)$ for some constants $A_n, B_n \in \mathbb{R}$.. Plugging in the BCs we get $X_n = \sin(n\pi x),\ n = 1, 2, 3, \ldots$ with $\lambda_n = n^2\pi^2$. Similarly consider $\frac{Y''}{Y} = \lambda \implies Y'' - \lambda Y = 0$. Using $\lambda = n^2\pi^2$, we get $Y'' - n^2\pi^2 Y = 0$. The solution to this ODE is also well known and it's $C_n \cosh(\sqrt{\lambda}x) + D_n \sinh(\sqrt{\lambda}x)$ for some constants $C_n, D_n \in \mathbb{R}$. Using the BCs we get, $Y_n = \sinh(n\pi(1 - y))$. Because of the non-homogenous BC at $y = 0$ we divide $Y_n(0)$ by its value at $y = 0$; thus, we get $Y_n = \frac{\sinh(n\pi(1-y))}{\sinh(n\pi)} = \operatorname{csch}(n\pi)\sinh(n\pi(1 - y))$. Therefore our solution is $u_n = \sin(n\pi x)\operatorname{csch}(n\pi)\sinh(n\pi(1 - y))$. Therefore, we define the formal solution as $u(x, y) = \sum_{n=1}^{\infty} \sin(n\pi x)\operatorname{csch}(n\pi)\sinh(n\pi(1 - y))$. Now we have $u(x, 0) = \sin()$. Comparing this and the formal solution we get that the infinite series is zero for all terms except for $n = 1$, this our solution for the above PDE is $u(x, y) = \operatorname{csch}(\pi)\sin(\pi x)\sinh(\pi(1 - y))$. □

To show the validity of our solution we compare it to the solution given by Wolfram's Mathematica [3] and the plot for the solution is given in Figure 1 and then we plot it in python [3].

3.1. **The Finite Difference Method.** The Finite Difference Method (FDM) is a numerical method used to approximate solutions to differential equations. It is commonly used in engineering and physics applications to solve problems in fluid dynamics, heat transfer, and structural mechanics, among others.

While the finite difference method is not typically used in machine learning algorithms, it can be used in certain types of numerical simulations that are relevant to machine learning applications. For example, the finite difference method can be used in simulations of physical systems that are used to generate training data for machine learning models.

In particular, finite difference simulations can be used to generate synthetic training data for machine learning models that are designed to model physical systems, such as weather forecasting or fluid dynamics. By using simulations to generate training data, it is possible to generate large data-sets that capture a wide range of possible scenarios and conditions, which can improve the accuracy and generalization of the machine-learning model.

The idea of finite difference method is to discretize the PDE by replacing the partial derivatives with their approximations i.e finite differences. We now solve the above PDE using the finite difference method:

3.1.1. *Finite Difference Method solution.* We have $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$. Once we discretize the grid of which the PDE is defined on and let each point on the grid be designated by a numbering scheme i and j, where i indicates x increment and j indicates y increment. The second derivatives at the point (i, j) can be approximated (derived from the Taylor series) as

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} \quad \text{and} \quad \left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2}$$

Our Laplace equation then gives us,

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} = 0.$$

Assuming $\Delta x = \Delta y$, the finite difference approximation of Laplace's equation for interior regions can be expressed as $u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$ and this is the solution for the PDE. □

3.1.2. *Explanation of the code.* [3] The code defines a rectangular domain with dimensions $a \times b$, and a grid of $n \times m$ points is created within this domain. The initial solution is set to zero everywhere except on the left boundary, where the solution is set to $\sin(\pi x)$ (`u[:, 0] = np.sin(np.pi*x)`), and the boundary conditions are specified. The update function computes a new estimate of the solution by taking the average of the four neighboring grid points. This is repeated until the solution converges to a specified tolerance or until a maximum number of iterations is reached. The for loop iterates over the update function until the solution converges or the maximum number of iterations is reached. In each iteration, the update function is called to compute a new estimate of the solution. The solution is updated only in the interior points of the grid, and the boundary conditions are not updated. Once the solution converges to within the specified tolerance, the loop is exited, and the final solution is plotted given in Figure 2 .

4. Neural Networks

Neural networks can be used to PDEs by learning a mapping between the input variables, which represent the domain of the PDE, and the output variables, which represent the solution to the PDE. To put it another way, the neural network uses the input variables as input and outputs the PDE solution. Making ensuring that the neural network fulfills the PDE at every point in the domain is the main problem when utilizing neural networks to solve PDEs. By including the PDE in the loss function used to train the network, this can be accomplished. Specifically, the loss function should penalize deviations from the PDE at each point in the domain.

One common approach is the discretization of the PDE using the method of finite differences that yields a set of equations that connect the solution at each point in the domain to its surrounding points. The loss function that penalizes departures from the PDE at each location in the domain can then be built using these equations. Using gradient descent or another optimization approach, the network trains to minimize this loss function. When the network has been trained, it can be utilized to produce PDE solutions at fresh input locations within the domain.

4.1. **Simple Feed Forward Neural Network.** A Simple Feed Forward Neural Network (FNN) only sends data in one direction, from the input layer through one or more hidden layers to the output layer. Nodes make up each layer and are joined to nodes in the layer below and/or the layer above by weighted connections. Each layer's nodes produce their output by applying a mathematical function to the weighted total of their inputs, which is then transmitted to the following layer. Because there are no loops or cycles in the connections of a FNN, the network's output is only dependent on its inputs and the weights and biases of its nodes. These weights and biases are learned during training by minimizing a loss function using back-propagation.

We first show the simplest case of just training our data on the solution of our PDE and show the results i.e in this code we are just using the solution of the PDE at the grid points as training data for a neural network, with the input to the neural network being the coordinates of the grid points and the output being the PDE solution at those points. We are not testing it on any new data. We talk about our results in the result section.

4.1.1. *Explanation of the code.* [3] The FNN is trained to approximate a solution to the Laplace Equation. The rectangular domain is first discretized into a grid with $n \times m$ points. The FNN is defined to take in as input the x and y coordinates of each point on the grid, and output is the solution at that point which we got using the FDM. The architecture consists of an input layer, a hidden layer with 32 neurons and a `ReLU` activation function, and an output layer with a single neuron and no activation function. The loss function used is mean squared error, and the Adam optimizer is used to minimize it. The network is trained for 25000 epochs. The final output of the trained FNN is plotted as a 3D surface. The variable $'u'$ contains the solution obtained from a finite difference element code run before, and is used to generate the training data for the FNN. The training data consists of the x and y coordinates of the grid points, and the corresponding values of u. Here our loss is just MSE.

The final solution is plotted given in Figure 3.

4.1.2. *Predictions.* [3] Now we finally implement the above but using training, validation and testing data sets. Essentially we train the PDE on the solution but only using a subset. Then we use it to predict the solution to the PDE at new input points within the domain.

For example, if we have a PDE that describes the temperature distribution in a room, we can train a neural network to predict the temperature at any point in the room given the relevant input variables such as the position, time of day, outside temperature, and so on. Once the network is trained, we can use it to predict the temperature at any new point in the room by providing the relevant input variables.

This ability to generate solutions to the PDE at new input points is a key advantage of using neural networks for PDEs, as it allows us to obtain a continuous approximation to the solution over the entire domain rather than just at a finite set of grid points.

The final testing plot is given in Figure 3.

4.2. **Physics Informed Neural Networks.** [1], [2] Class of neural network-based models that can be used to solve PDEs. The idea behind PINNs is to use neural networks to learn the solution to a PDE, while also incorporating any known physical laws or constraints that govern the behavior of the system being modeled. The basic approach in PINNs is to use a neural network to approximate the solution to the PDE, and then incorporate the PDE as a constraint in the training process. This is done by adding a loss function that penalizes any deviation from the PDE, which encourages the neural network to learn a solution that satisfies the PDE.

In our case we have the Laplace Equation with Dirichlet BC which is one of the simplest PDE. Our additional loss (along with MSE) is just the going to be called the *residual* loss (r_{loss}) which in our case is just,

$$r_{\text{loss}} = \|\Delta u + f(x,y)\|_{L_2}^2 + \|u(x,0) - \sin(\pi x)\|_{L_2}^2$$

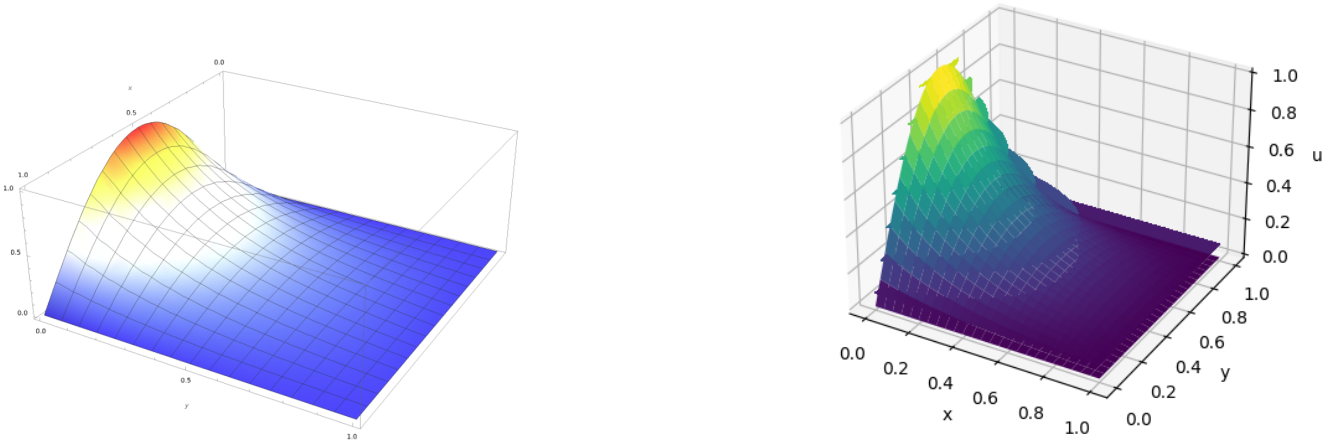
- Δu is the Laplace operator applied to the predicted solution $u(x,y)$ of the PINN.
- $f(x,y) = 0$ is the source term in the Laplace equation.
- $u(x,0)$ is the predicted solution at the bottom boundary, which is defined to match the sine function at the BC.

4.2.1. *Explanation of the code.* [3] We essentially have the same code as before 4.1.1 plus r_{loss} on which our NN is trained on. In the `residual` function, we calculate the second-order partial derivatives using finite difference method . In the `residual_loss` function, we calculate the residual using the residual function and return the mean squared residual. We then incorporate our BCs in our residual loss r_{loss}.

4.2.2. *Predictions.* [3] Like 4.1.2 we implement the above but using training, validation and testing data sets. The final plots are given below in the Results Section in Figure 4.

5. Results

5.1. **Analytic Solutions.** First we plot the solution in Wolfram alpha's Mathematica and the corresponding solution in python to verify our solution.



(A) Mathematica Solution (B) Python Solution

Figure 1. Analytic Solutions.

5.2. **FDM Solution.** We solve the PDE using FDM and we get the following plot:

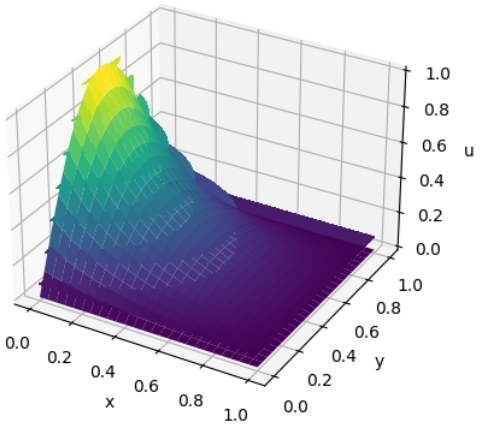


Figure 2. FDM solution

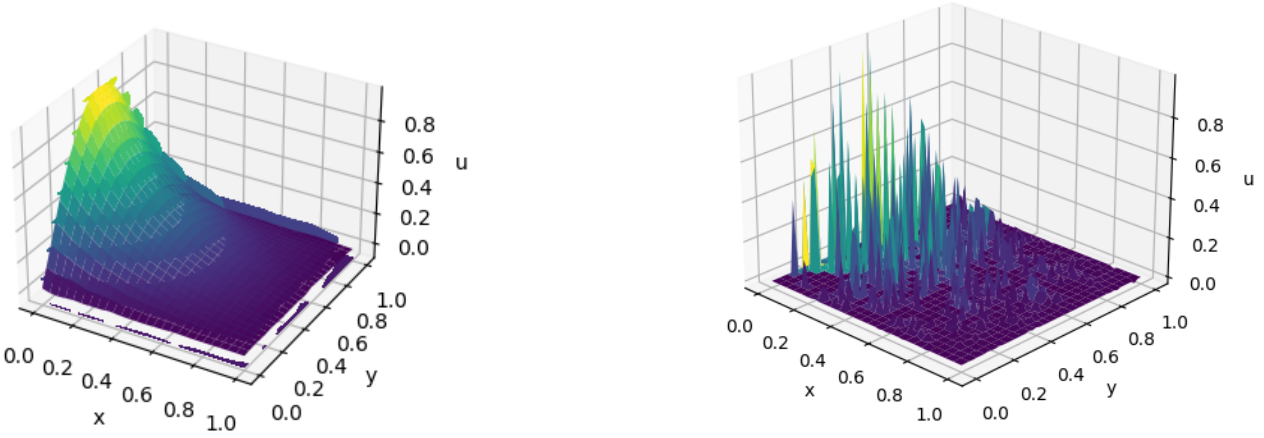
From the plot we see no difference between this solution and the analytic solution and this can be verified by `np.linalg.norm(u- u_fdm)` which gives us $9.995295922238655e-05$ which shows our FDM method is very accurate.

5.3. **FNN Solution.** We first do hyper-parameter tuning for 1000 epochs for our activation function and hidden layer sizes. We get the following:

```
Best parameters: ['activation': 'relu', 'hidden_layer_size': 64]
L2 norm difference for best model: 0.1720.
```

Then we first plot the solution we got by doing this on the entire data-set which is shown below in Figure (A) and then we implement the Train- Test- Validation framework using the best hyper-parameters. Essentially in this case the test data are unseen points in the meshgrid which we predict and we plot below in Figure (B). We get the following losses:

```
Training Loss: 0.0098, Validation Loss: 0.0121, Testing Loss: 0.0092
```



(A) FNN Solution (B) FNN Test Data

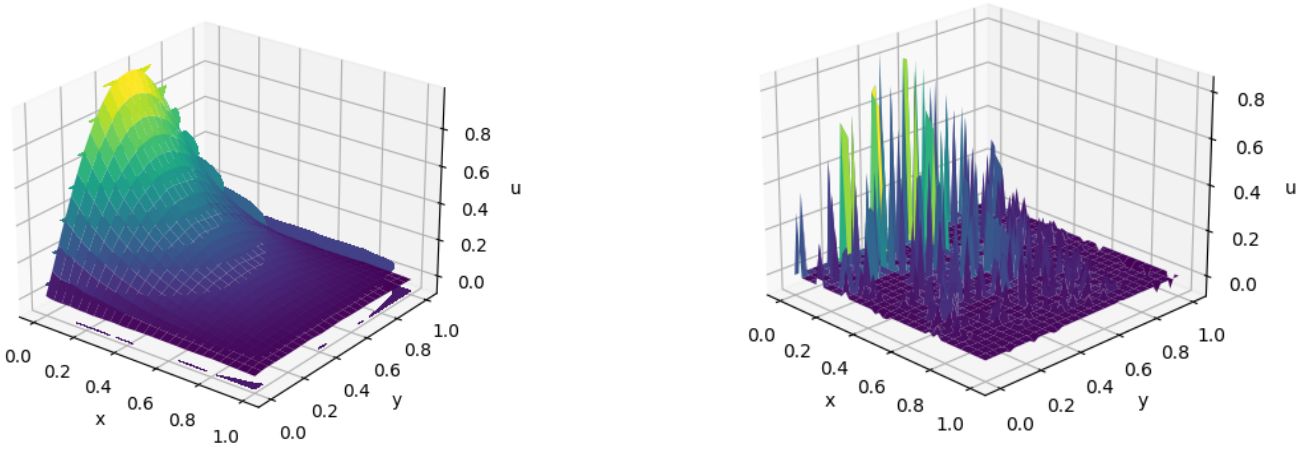
FIGURE 3. FNN Solutions.

5.4. **PINN Solution.** Similar to 5.3, by hyper-parameter tuning for 1000 epochs we get:

```
Best parameters: ['activation': 'relu', 'hidden_layer_size': 64]
L2 norm difference for best model: 0.1801.
```

Similar to 5.3, we get the following losses and the following plots:

```
Training Loss: 0.0018, Validation Loss: 0.0023, Testing Loss: 0.0022
```



(A) PINN Solution (B) PINN Test Data

FIGURE 4. PINN Solutions.

5.5. **Comments.** We directly see that FDM has best approximated our PDE as it has lowest `L2 norm difference` from the analytic solution. This is probably because we are considering a very simple PDE i.e the Laplace Equation with simple BC. We also see that the NN and PINN have comparable results which doesn't tell us much about the performance. This is because our source term i.e $f(x, y)$ in $\Delta u = f(x, y)$ is 0 and we have simple BC so the residual loss r_{loss} is not that large, so when we add it to the MSE loss it doesn't make much of a difference. However, it is noticed that the PINN always does slightly better in approximating the analytic solution.

Note. We also additionally plotted the contours in the plots which explains the horizontal lines.

6. FDM vs FNN vs PINN?

Finite Difference Method (FDM), Neural Networks (NN), and Physics-Informed Neural Networks (PINN) are the three different approaches for solving partial differential equations that we have discussed.

By discretizing the PDE and solving the resulting system of algebraic equations using finite differences to approximate the derivatives, FDM is a numerical technique. Althought it can be limited by its accuracy, stability, and convergence rate. NN and PINN are machine learning-based methods that use artificial neural networks to approximate the solution of the PDE. NN uses a feedforward neural network to learn the mapping between the input coordinates and the corresponding output solution values. On the other hand, PINN incorporates the PDE into the neural network's loss function, which allows it to learn the solution and the underlying physical laws simultaneously.

When comparing these approaches, FDM is typically quicker and more effective at solving PDEs with straightforward boundary conditions and geometries (as we saw in the Results section). However, when solving PDEs with complicated geometries and high dimensionalities, FDM can become computationally expensive. In summary, the choice of method depends on the problem's complexity, the available data, and the required accuracy and computational resources

7. Limitations

One main limitation we encountered while doing the project was the lack of computing power required to run the code with more finer mesh-grids. Finer meshes helps improve the discretization of the solution and provide better approximations of the PDE, resulting in more accurate solutions. Other limitation was since using NNs to solve PDEs is a relatively new field in Machine Learning, there is a lack of data online which could help in facilitating the learning of PDEs. Due to this we had to generate our own data.

8. Applications

Due to the recent advances in this field using Neural Networks to solve PDEs has shown promise in various fields such as:

- Fluid dynamics: Neural networks have been used to solve the Navier-Stokes equations, which describe the motion of fluids. This has been applied to problems such as turbulent flow, flow around obstacles, and fluid-structure interactions.
- Quantum mechanics: Neural networks have been used to solve the Schrödinger equation, which describes the behavior of quantum particles. This has been applied to problems such as molecular dynamics and quantum control.
- Heat transfer: Neural networks have been used to solve the heat equation, which describes the transfer of heat in a material. This has been applied to problems such as heat conduction in composite materials and cooling of electronic devices.

9. Discussion

We first start off introducing the topic in Section 1. Then in Section 2 we introduce the concepts of Partial Differential Equations 2.1 and Neural Networks 2.2. Then in Section 3 we introduce the Laplace Equation and present the Analytic Solution 3.0.1 and the Finite Difference Method 3.1, it's solution 3.1.1 and Code Explanation 3.1.2. Then we move onto Neural Networks in Section 4 and we talk about the simplest Neural Network i.e the Simple Feed Forward Neural Network 4.1, explain the code 4.1.1 and present predictions 4.1.2. Then we finally talk about Physics Informed Neural Networks 4.2 and similarly it's code 4.2.1 and predictions 4.2.2. Then we discuss the results which are all mentioned in Section 5. We present the Analytic Solution 5.1, Finite Difference Method Solution 5.2, Feed Forward Neural Network Solution 5.3 and the Physics Informed Neural Network Solution 5.4 and leave some comments for the

results 5.5. Lastly we comment on the general machine learning frameworks/ algorithms we discussed in Section 6 and talk about some Limitations 7 and Applications 8 in the last two Sections.

10. Acknowledgements

This project has been a fulfilling experience, thanks to the incredible individuals I have had the privilege of meeting. I would like to express my deep gratitude to them here. Firstly, I would like to thank God for giving me the wisdom, knowledge and understanding to undertake this project and course. Second, I would like to thank Professor Lili Mou for blessing me with the opportunity to undertake this project and I would like to thank my parents, brother and friends for their love, support, and encouragement during the process of writing this project and have supported me throughout my academic and personal journey.

References

[1] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). *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. (ICML 2017).

[2] Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations.* Journal of Computational Physics, vol. 378, 2019, pp. 686-707. (IEEE ICMLA 2018)

[3] Hey, Joshua. CMPUT-466-Final-Project. GitHub, 28 Apr. 2021, https://github.com/joshuahey/CMPUT-466-Final-Project.