Solving PDE's with Physics Informed Neural Networks: A comparison with Finite Difference Methods

Kiran Dsouza

Dartmouth College, Hanover, NH 03755
kiran.dsouza.th@dartmouth.edu

Abstract. The emerging field of Scientific Machine Learning combines methods from both machine learning and scientific computing. In this study we explore solving partial differential equations (PDE's) with Physics Informed Neural Networks (PINN's), which is a promising research direction within this field. To evaluate the benefits of PINN's, we solve three one-dimensional evolutionary PDE's using PINN's and compare the results with Finite Difference (FD) methods. We also look at periodic boundary conditions and investigate how to improve the PINN results by changing hyperparameters as well as the network architecture.

Keywords: Partial Differential Equations, Deep Learning, Physics Informed Neural Networks, Scientific Machine Learning

1 Introduction

Neural Networks have been around for more than eighty years, and they have seen many waves of interest in that time. The most recent wave has huge momentum, starting with the breakthrough success of the neural network based AlexNet in the 2013 ImageNet competition to recognize images. Today, this technology is routinely used in speech recognition, vision, machine translation and most recently, chatbots. There are many other machine learning methods, but neural networks are dominant, and their use is often referred to as Deep Learning because of the multiple layers of artificial neurons used today. It is natural to explore their application to other areas as well, such as techniques for solving partial differential equations (PDE's) that may complement existing approaches.

In this study, we solve three PDE's with PINN's and again with FD methods as baseline for comparison. We implemented both as scripts in Matlab so that we have full control to better understand how each method works and how PINN's might be improved. In particular, we explore changes to the neural network architecture and hyperparameters for tuning. We did not find much existing work comparing PINN's with other methods, mainly [3] that compare PINN's with finite element methods for steady-state equations but they use off the shelf solvers as black-boxes on a Poisson equation, namely the commercial Abaqus tool for FEM and the DeepXDE tool [7] for PINN's. Their main conclusion was that DeepXDE was slower and less accurate than Abaqus, and tuning hyperparameters did not help to improve results.

2 Neural Networks

Neural networks are loosely inspired by the architecture of the brain, which consists of a network of connected neurons that propagate signals at their outputs based on activation functions fed by a set of inputs. As explained in [9], a very simple trainable neural network is the perceptron, comprising a single neuron that outputs a 1 if a weighted sum of the inputs and a bias value is positive and 0 otherwise. This step function is called its activation function, and the weighted sum is called a *logit*. The perceptron training algorithm will adjust the weights based on a set of labeled inputs: if the output is 1 when it should be 0, or vice versa, the weights are reduced; otherwise, they are increased. For recognizing multiple classes such as classifying 28x28 images of handwritten digits, an array of ten perceptrons with 28x28=784 input values are used, with the digit recognized based on the highest logit. The problem with perceptrons is that some simple functions like XOR cannot be represented in one layer.

This perceptron model can be extended to a single layer neural network and trained using a relatively simple optimization technique such as gradient descent. This requires a differentiable loss function, so the logits are converted into a probability distribution using the SoftMax function, which for our example for digit *d* and input *x* would be:

$$\sigma(d,x) = \frac{e^{l_d(x)}}{\Sigma_i e^{l_i(x)}}$$

The loss function, *X*, is based on the probability of the correct label (digit) being picked for any input image *x*:

$$X(\omega,x) = -\ln\left(\sigma(d,x)\right)$$

An optimization method such as conjugate gradient descent will minimize this loss function, in effect minimizing the difference between the true probability and the current neural net. The architecture of this one-layer neural network from [9] is shown in Figure 1, where input *x* is combined into weighted sum logits and then each logit is converted into a probability by a SoftMax application σ. The input layer is of width 784, the next two layers are of width 10, each arrow represents connections between all elements at adjacent layers.

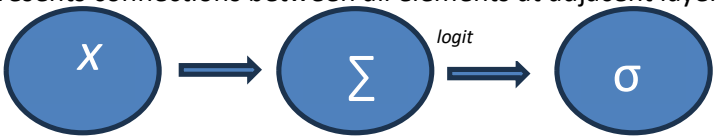

Figure 1 Single layer neural network

The training of this model involves two passes: a forward pass from inputs *x* to get the logits, *l(x)*, and then a back propagation pass using gradient descent to update the weights using gradient descent. The *i*th weight for digit *j* is updated with learn rate $\mathcal{L}$ using:

$$\Delta w_{i,j} = -\mathcal{L} x_i \frac{\partial X}{\partial l_j}$$

After training on 70% of the MNIST database of digit images, this single layer system is 90% accurate[9]. To get further improvement, multiple layers of neurons must be used, and the layers must be connected using a nonlinear activation function to increase the range of functions the network can represent. Commonly used activation functions include the rectified linear unit (ReLU), the sigmoid function, and *tanh*, which have derivatives that are easy to compute. To handle back propagation across multiple layers, we do it one layer at a time, working from outputs to inputs. Figure 2 shows graphs of these three activation functions.

In our work on PDE's, the neural network is computing a function and not acting as a classifier as in the digit image classification example. A universal approximation theorem says that any continuous function on a closed bounded subset of $\mathbb{R}^n$ is representable by a multi-layered neural network[10]. However, it is not known what class of functions an arbitrary given network can represent.

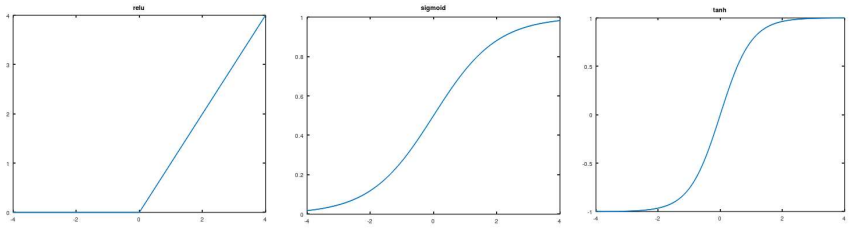

Figure 2: THREE ACTIVATION FUNCTIONS: RELU, SIGMOID, TANH

3 Physics Informed Neural Networks

Though neural networks can represent any continuous function with the right setup, training a network to do this can be ineffective without making use of more information about the problem. Therefore, when solving a PDE, we can make use of the differentiability of a neural network to embed the PDE itself into the loss function when training the neural network, in addition to the initial and boundary conditions. Furthermore, the ability of the neural network to represent nonlinear functions suggests that it could provide another technique to solve PDE's. These ideas are explored in an extensive study[8], though this only considered Laplace and Poisson (stead-state) PDE's. The approach was modernized and expanded as

PINN's[1] twenty years later, following the AlexNet success. In this work, they consider PDE's of the form:

$$f = u_t + N[u]$$

where N is a differential operator. They then define the loss function for the neural network as follows: They pick N_u points for guiding the network towards the initial and boundary conditions, and N_f points for guiding it towards the PDE itself. The loss function is the mean square error (*mse*) sum of the deviation from these two constraints on the function the network is to learn:

$$mse = mse_u + mse_f$$

$$mse_u = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u\left(t_u^i, x_u^i\right) - u^i\right|^2$$

$$mse_f = \frac{1}{N_f}\sum_{i=1}^{N_f}\left|f\left(t_f^i, x_f^i\right)\right|^2$$

Besides introducing PINN's, there are two other contributions in [1]: they use a semi-discretization approach with Runge-Kutta time stepping as in conventional methods, and they explore learning the parameters of a PDE from data. We do not explore these techniques in this study. The limitations of the PINN approach are not explored in [1], but the Wikipedia page on PINN's lists the following issues:

1. Hard to approximate translation and discontinuous behavior.
2. Fail on differential equations with slight advective dominance
3. Fail on systems of dynamical/chaotic equations
4. Are based on optimization and therefore prone to getting stuck on local optima

We based our PINN implementation on a version we found on the Matlab website[4], designed around a standard example used in PINN expository work (Burger's equation). The following table summarizes the default settings we used in our work, but we also report the results of with varying these defaults.

Activation Function	*Layer Count*	*Layer Neurons*	*Training Epochs*	*Learn Rate*	*Optimizer*	N_u	N_f
tanh	9	20	200	0.01	Adam	75	10000

4 Finite Difference Method

The finite difference method (FD) is a widely used technique for solving PDE's by approximating derivatives with finite differences[10]. The time and space domains of the PDE are discretized by dividing them into a finite number of intervals. At each discretization point the solution is approximated using the finite differences and values from nearby points. This converts the problem of solving a PDE into manipulating linear systems that are very computationally efficient, though scalability can be an issue if the discretization produces a huge number of points for evaluation.

5 Experimental Methodology

We solve the equation with PINN's and FD, and then evaluate the solution at 5 values of *t* between 0 and 1. At each time, we evaluate the solution at *N* = 1000 equally spaced *x* values. This is different from the mesh spacing in general, so we interpolate the result for FD. This interpolation is not needed for the PINN since the method is mesh free and we can use the trained network at any point. Both the PINN and FD solutions are compared with the exact solution and the error for each value of *t* is computed as the relative L_2 error, *rle,* over the *N* values of *x*. We calculate the relative L_2 error as the L_2 norm of the differences between the computed solution, u_c, and the exact solution, u_e , divided by the L_2 norm of u_e. To compare FD and PINN, we report this relative L_2 error at *t* values 0, 0.25, 0.5, 0.75 and 1.0.

$$norm(t,u) = \sqrt{\sum_{i=1}^{N} |u(t,x_i)|^2}$$

$$rle(t) = \frac{norm(t, u_e - u_c)}{norm(u_e)}$$

Of course, the implementation of the exact solution also involves approximation but here we take it as a common reference. In our experiments where we vary the hyperparameters of

the PINN, we also use the relative L_2 norm to decide which settings are beneficial. We do not consider the effects of changing combinations of hyperparameters in this work, and we just one at a time.

6 Simple Transport Equation

As a first sanity check for our PINN implementation, we solved the following first order equation from p9 of the textbook by Strikwerda[2]. The BC's are the boundary condition, and the IC's are the initial condition. We picked a this simple example to help debug our PINN implementation and experiment with the PINN hyperparameters. We computed solutions up to time 1.

PDE	$u_t + u_x = 0$	$0 < x < 1$
BCs	$u(t,0) = 0$	$0 < t < \infty$
ICs	$u(0,x) = sin(\pi x)$	$0 \leq x \leq 1$

For our PINN, we used the default settings described in Section 3. The BC was specified at 50 equidistant *x* values between 0 and 1. The IC was described at 25 equidistant *t* values between 0 and 1. The PINN trained for 150s. The figure below shows the solution, the loss function being optimized by the PINN in Matlab, and the relative L2 error at times 0, 0.25, 0.5 and 0.75.

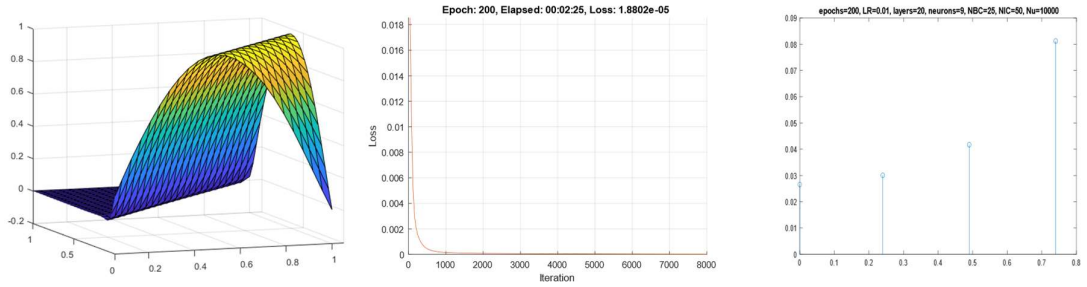

Figure 3 PINN solution, Loss optimization graph, and relative L2 error at given times.

This basic transport equation is solved in under a second with a Lax-Friedrich FD implementation descibed in Strikweda's textbook [2]. The following figure shows the relative error norm using this method for three mesh sizes (100x100, 1000x1000 and 10000x10000) at times 0, 0.25, 0.5, and 0.75. The error was always 0 at time 1. The 100x100 result took 1s and is much better than the the PINN accuracy for about the same number of points (100000). Changing the mesh to 1000x1000 we get a more accurate results in 2s, and using a 10000x10000 mesh we further improve the error in 5s. Clearly, the simple explicit Lax-Friedrich FD is a much better solver here compared to the default PINN setup we have.

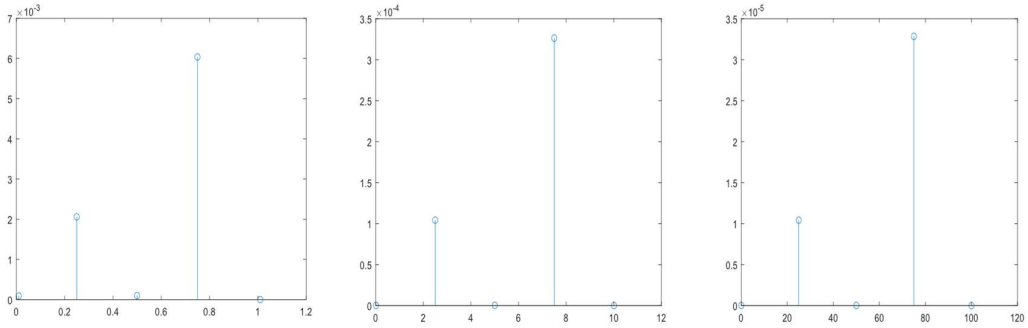

Figure 4 Relative L_2 error for Lax-Friedrich at given times

6.1 Extrapolation

Since the PINN we used is mesh free, we did not have to interpolate values between mesh points. A related question is whether the PINN is useful outside of the range of time values where we trained the PINN. To evaluate this, we trained the PINN for time *t* up to 0.8, and then evaluated it at time 9 against the exact solution using the relative L_2 norm. The error was much higher than for interpolation. We didn't use an extrapolation method for FD because that method would also be applicable to PINN.

`

6.2 Experiment to vary the training epochs

Increasing the number of training epochs (passes over the data) steadily reduces the error but the run time also increases significantly. The gains seem to taper off, so using about 800 epochs seems like a good compromise, higher than our default of 200 epochs.

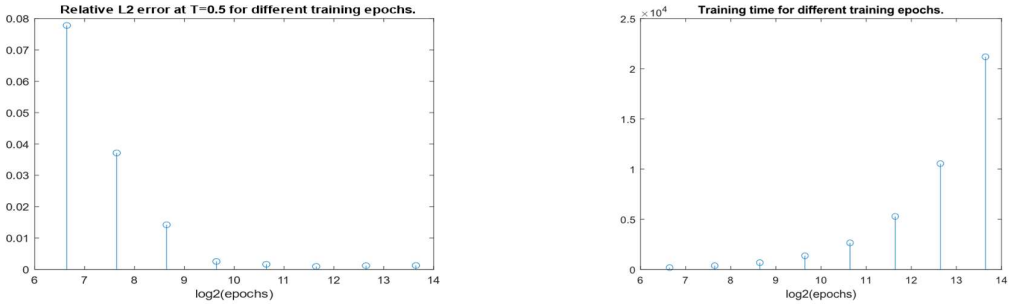

Figure 5 Relative L2 error and training time for increasing epochs.

6.3 Experiment to vary the learning rate

We varied the learning rate for the optimizer with values 0.1, 0.05, 0.01, 0.005, and 0.001. The results show the run time was about the same, but the accuracy was best around 0.005. So our default learning rate of 0.01 was not optimal.

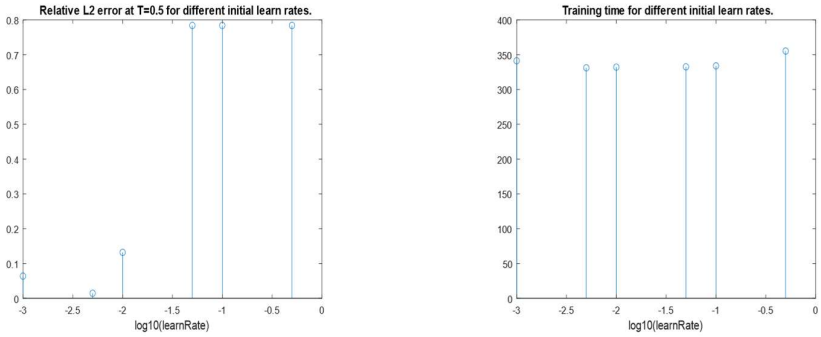

Figure 6 Relative L2 error and training time for learning rates

6.4 Experiment to vary the number of PINN layers

Our default value of 9 neurons per layer is optimal for this example, since the error increa if we go higher or lower.

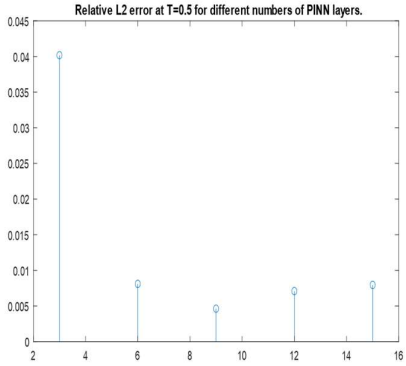

Figure 7 Relative L2 error vs PINN layers

6.5 Experiment to vary the number neurons in a PINN layer

The results of varying the neurons in a layer show the default value of 20 was fine, though 30 was very slightly better, but probably not enough to justify the higher compute cost. The behavior is erratic.

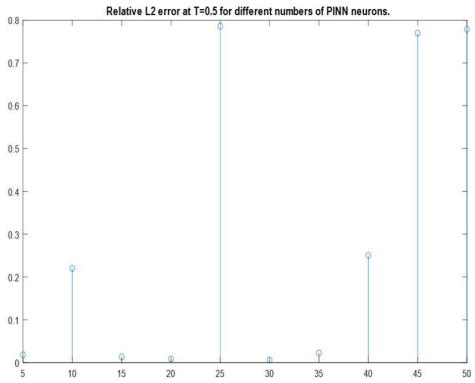

Figure 8 Relative L2 error vs neurons per layer

6.6 Experiment to vary the number of training points

Increasing the number of training points for the differential equation reduces the error but the run time increases linearly. It seems 25000 is a good compromise between run time and error, which is higher than our default of 10000.

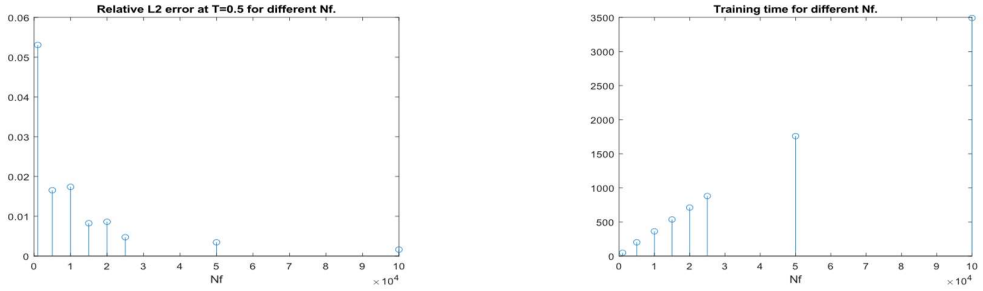

Figure 9 Relative L2 error, and training time vs PDE training points

Increasing the number of boundary used for training seems to increase the error. Our default value of 25 seems relatively good. The run time is not affected by the number of boundary points.

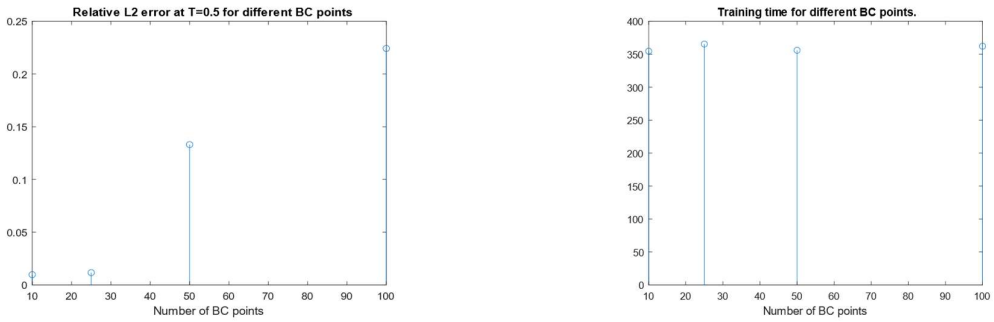

Figure 10 Relative L2 error, and training time vs number of boundary condition points for training

Increasing the number of initial condition training points also seems to increase the error initially but then it drops. This is also hard to explain. Our default value of 50 seems bad. The run time is not affected.

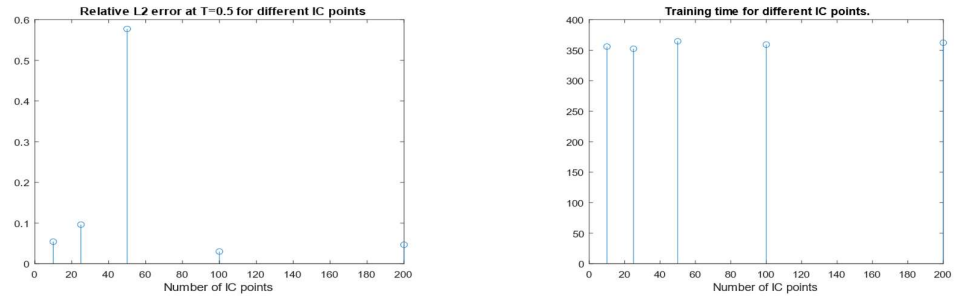

Figure 11 Relative L2 error, and training time vs number of initial condition points for training

6.7 Experiment to vary the activation function

As shown in the figure below, the default activation, *tanh*, performed far better than the other widely used functions: sigmoid, relu and leaky relu. There is no reason to change it.

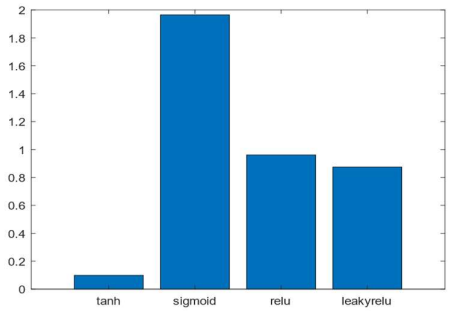

Figure 12 Relative L2 error for different activation functions

7 Another Transport Equation

The next equation we tried was the transport equation on p18 of the textbook by Strikwerda[2]. The solution is not smooth, and it is a little tricky even for the FD method we tried (Lax-Friedman) because the ratio of the time step to the space step must be small. The FD solution again was computed in under 1s and was more accurate than the PINN solution that took several minutes. The two figures below show the FD solution (right) and the PINN solution (left) at time 0.13.

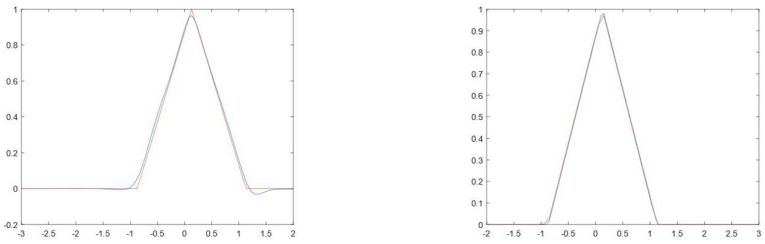

Figure 13 Solution to transport equation at t=0.13. Left: PINN Right: FD

The relative L2 error for the FD method was 0.02, but for the PINN method with default parameters it was 4.3112. Even though PINN's are slower and less accurate, his experiment

showed us that they can cope with discontinuity in the derivatives. The red lines in both figures are the exact solution.

8 Transport Equation with periodic boundary condition

Our final experiment with first order transport equations was to explore something we didn't initially find in existing work: handling periodic boundary conditions in PINN's. Our solution was to add a third term to the *mse* loss function for this boundary condition as the mean square error of the deviation of the boundaries from each other. The changes to the original loss function are as follows:

$$mse = mse_u + mse_f + mse_p$$

$$mse_p = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u\left(t_u^i, 1\right) - u\left(t_u^i, 0\right)\right|^2$$

Subsequently we discovered a similar method used with a PINN-based semidiscretization method, but it's still interesting to discover if it works with the PINN method that we are investigating where time-stepping is not handled outside of the PINN. Interestingly, the Matlab pdepe solver does not support periodic boundary conditions.

The problem we used for this experiment was taken from HW5 of Math 116.

PDE	$u_t = u_x$	$0 < x < 1$
BCs	$u(0,t) = u(1,t)$	$0 < t < \infty$
ICs	$u(x,0) = sin(2\pi x)$	$0 \leq x \leq 1$

The results of this experiment is shown in the following diagrams as snapshots at different times. On the left is the result of the PINN at various time steps and on the right is the FD result using Lax-Friedrichs.

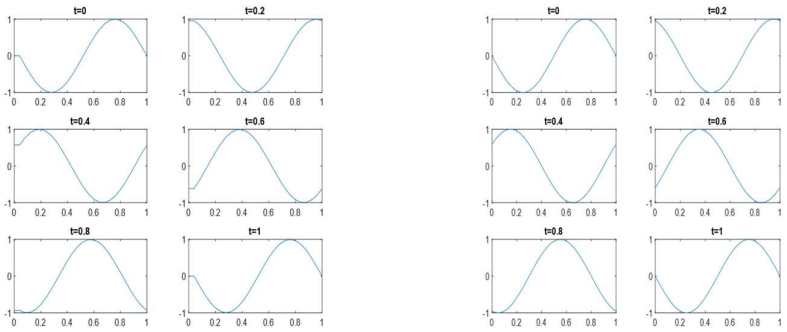

Figure 14 Solution to transport equation with periodic boundary conditions. Left: PINN Right: FD

The FD solution very much more accurate and completed in less than 1s. The PINN took 6m. The relative L2 errors are shown below.

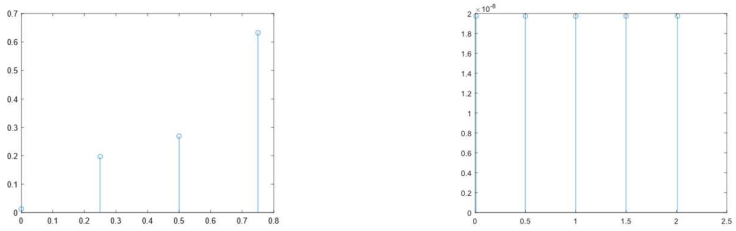

Figure 15 Relative L2 errors at given times with periodic boundary condition using PINN (left) and FD (right).

9 Diffusion Equations

9.1 Nonhomogenous Diffusion Equation

We solve the PDE from page 69 of Farlow's textbook[5], with initial and boundary conditions as follows:

PDE	$u_t = \alpha^2 u_{xx} + sin(3x)$	$0 < x < 1$
BCs	$u(0,t) = 0$ $u(1,t) = 0$	$0 < t < \infty$

ICs	$u(x,0) = sin(\pi x)$	$0 \leq x \leq 1$

The PINN solution after training for 1000 epochs took 5m and is shown below.

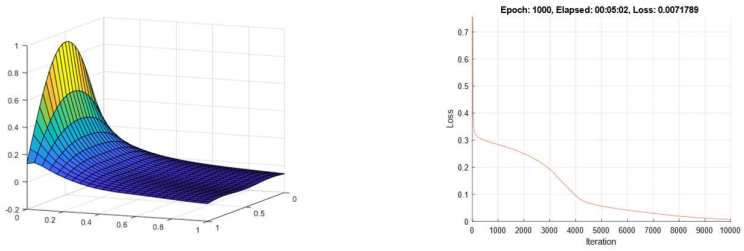

Figure 16 PINN solution for nonhomogenous diffusions equation, and loss optimization graph.

The PINN error is worse than for the transport equation when compared with the exact solution. The FD method used (Crank Nicholson) is more accurate and faster with just a 25x25 grid as shown below.

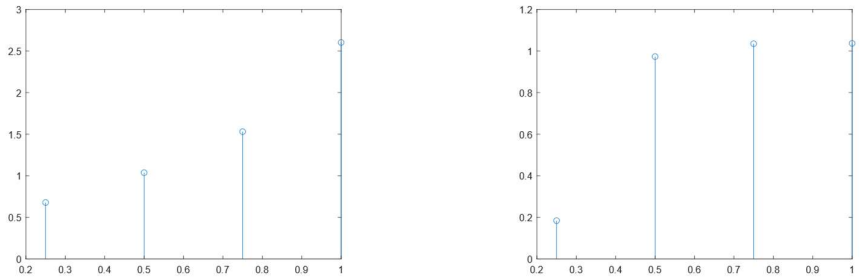

Figure 17 Relative L2 error at given times for PINN (left) and FD (right)

10 Wave Equation

We consider the following wave PDE:

PDE	$u_{tt} = \alpha^2 u_{xx}$	$0 < x < 1$ $0 < t < \infty$

BCs	$u(0,t)=0$ $u(1,t)=0$	$0<t<\infty$
ICs	$u(x,0)=sin(\pi x)$ $u_t(x,0)=0$	$0<x<1$

The work that formed the basis for this study restricted the method to only first derivatives in time[1], but we didn't fully understand the need for this restriction other than the need to handle the initial condition for the first time derivative. Since we had some success with extending the loss function to periodic boundary conditions, we decided we could add the initial condition for the first derivative to the loss function as well. However, the results were very bad. In the following figure, the graph on the left shows the PINN solution where the results deteriorates as time progresses. The FD solution is on the right (oriented a bit differently) and closely matches the exact solution. A semidiscretization approach with PINN's might perform better here.

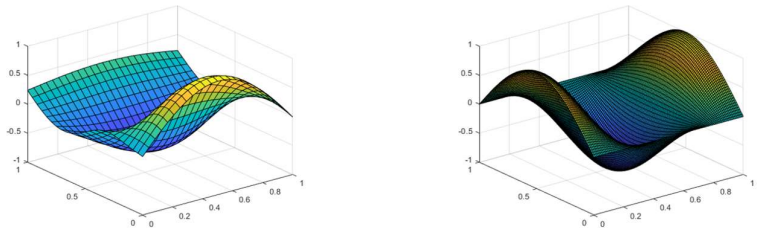

Figure 18 PINN solution to wave equation (left) and FD solution (right).

11 Conclusions

We have demonstrated that PINN methods can solve first order transport PDE's and also diffusion equations. Both run times and accuracy are much worse that using FD methods at present, a similar conclusion to an earlier study comparing finite element methods with PINN's for a Poisson equation. However, unlike that study, we found that changing some hyperparameters do improve the PINN accuracy. However, even then, there is a limiting accuracy that is worse than what FD methods achieve in even a fraction of the time. It may be that semidiscretization methods used in conjunction with PINN's, or PINN's applied to experimental data to help learn the parameters of a PDE may be a better direction than using

PINN's as a mesh-free alternative to FD methods. However, given the huge amount of research in Deep Learning, there may be more scope to improve PINN's across the board.

References

1. Raissi, M., Perdikaris P., Karniadakis, G.E.: Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, Journal of Computational Physics, Volume 378,2019, Pages 686-707, ISSN 0021-9991, https://doi.org/10.1016/j.jcp.2018.10.045.
2. Strikwerda J.: Finite Difference Schemes and Partial Differential Equations Second Edition (ISBN: 0-89871-567-9) .
3. Sacchetti, A. , Bachmann, B., Löffel, K., Künzi, U., and Paoli, B: Neural Networks to Solve Partial Differential Equations: A Comparison With Finite Elements, in IEEE Access, vol. 10, pp. 32271-32279, 2022, doi: 10.1109/ACCESS.2022.3160186.
4. Mathworks: "Solve Partial Differential Equations Using Deep Learning."

   https://www.mathworks.com/help/deeplearning/ug/solve-partial-differential-equations-using-deep-learning.html
5. Farlow, S: Partial Differential Equations for Scientists and Engineers, Dover Books, 1993
6. Skeel, R. D. and M. Berzins: A Method for the Spatial Discretization of Parabolic Equations in One Space Variable, SIAM Journal on Scientific and Statistical Computing, Vol. 11, 1990, pp.1–32.
7. Lu, L., Meng, X., Mao, Z., and Karniadakis G., DeepXDE: A Deep Learning Library for Solving Differential Equations SIAM Review 2021 63:1, 208-228
8. Lagaris, I, Likas, A, and Fotiadis, D: Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *EEE Trans. Neural Networks* **9**(5), 987–1000 (1998).
9. Charniak. E: Introduction to Deep Learning. MIT Press, 2018
10. Goodfellow, I, Bengio Y, and Courville, A: Deep learning, MIT Press, 2016. (Deeplearningbook.org)