A Machine Learning Approach to Partial Differential Equations

Nathaniel Jones, Brendan Keller, Samar Mahmood, Chuan Li

MAT 455/555: Industrial Mathematics Practicum

10 May 2024

1 Introduction

Neural networks have become something of a hot topic in recent years. With the rise of large language models, image classification models, text-to-image models, and more, neural networks have captured the imaginations of the scientific community and the public at large. The key strength of neural networks is their flexibility: given enough training data, a neural network can eventually learn to model any phenomenon, even if the underlying phenomenon is not well understood by humans. In this investigation, we studied how physics-informed neural networks (PINNs) can be applied to solve fluid dynamics problems involving partial differential equations.

Neural networks are a machine learning model inspired by the human brain. A set of nodes called neurons are organized into a series of layers. The inputs to the network are recursively passed through the layers, the output of each layer's neurons being used as input for the next layer's neurons. Each neuron stores a set of real numbers called weights, and another real number called a bias; to compute an output, a neuron creates an affine combination of its inputs using the weights and the bias, and plugs this combination into an activation function. Activation functions are the key to a neural network's flexibility, as they transfer the linear combination of weights and inputs plus bias into a nonlinear output. Despite its simplicity, this architecture is extremely powerful, and can be used to approximate an enormous variety of functions.

To study how neural networks can be used to solve partial differential equations, our research used the one-dimensional viscous Burgers' Equation. Burger's Equation is a nonlinear partial differential equation (PDE) originating in fluid mechanics. As one of the simplest models of turbulence, it combines characteristics of the first-order wave equation and the heat conduction equation, and can be thought of as a simplified form of the Navier-Stokes equation. Its applications include jet flows, growth of molecular interfaces, shockwaves, gas mechanics, and more.

The specific initial and boundary value problem (IBVP) we studied is given as follows:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2}$$

$$u(x, 0) = -sin(\pi x)$$

$$u(-1, t) = u(1, t) = 0$$

u refers to velocity, *x* to space, *t* to time, and v to viscosity, which is a constant. This specific problem was chosen for two reasons. First, this particular IBVP is well-known and has an exact solution to which we can compare the approximate solution by the model. Second, it is interesting to study because there is a jump discontinuity in the solution, which increases the difficulty in modeling.

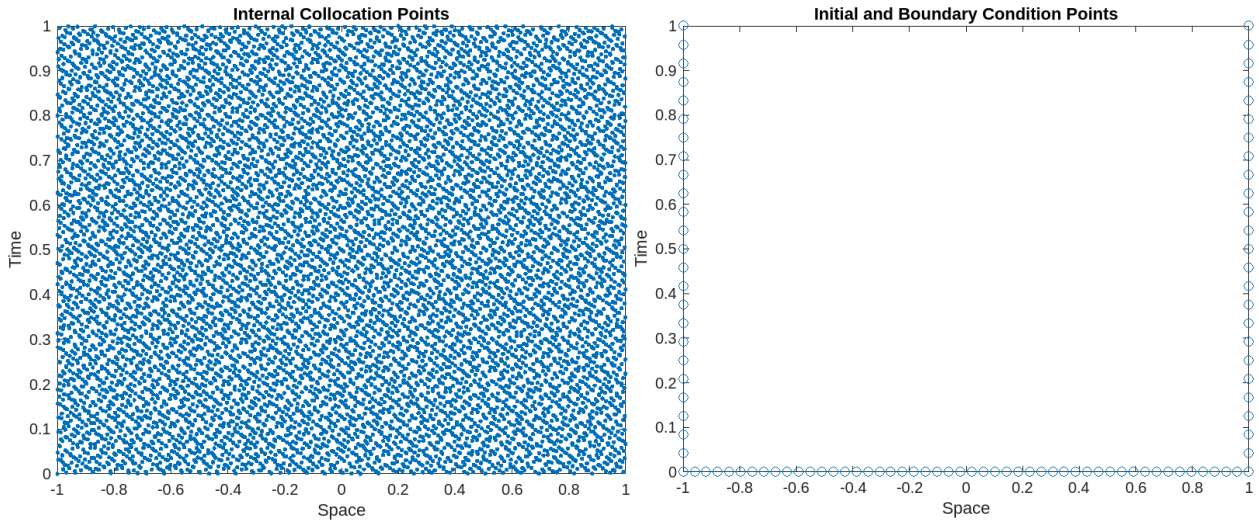
The goals of this research were twofold. First, we wanted to understand how altering various parameters affects the network output. In particular, we were interested in the effects of network size, the choice of activation function, and the structure of the training data. Furthermore, we wanted to know if there is any way to determine the accuracy of the network's solution without directly comparing it to the exact solution, since in real-world applications an exact solution would not be known. Second, we wanted to know how PINNs compare to a standard finite difference method in speed and accuracy. Furthermore, we wanted to test the claim that PINNs can be

used to make predictions even for ill-posed problems, such as problems with incomplete initial conditions and inverse problems.

2 Methodology

2.1 Loss Function

As mentioned previously, a neural network contains a set of weights and biases (henceforth known as "learnable parameters") that affect the shape of the final output. However, these learnable parameters are not encoded by a programmer, but rather found through an optimization process. Specifically, we want to find the values of the learnable parameters that minimize a function of the network output known as the loss function. The loss function is a critical component of any machine learning technique, but its structure is especially important for PINNs.



$$L_{\text{Burgers}}(\hat{u}) = mse(\frac{\partial \hat{u}}{\partial t} + \hat{u}\frac{\partial \hat{u}}{\partial x}, v\frac{\partial^2 \hat{u}}{\partial x^2}) \qquad L_{\text{Data}}(\hat{u}) = mse(\hat{u}, u_0)$$

$$L(\hat{u}) = L_{\text{Burgers}}(\hat{u}) + L_{\text{Data}}(\hat{u})$$

The loss function consists of two parts, L_{Burgers} and L_{Data}, both of which need to be minimized. L_{Burgers} is the mean squared error between the left and right sides of Burgers' equation across all of the internal collocation points, which are a quasi-randomly distributed set of points in the domain. We use quasi-random points as opposed to a uniform grid to avoid systematic error caused by aliasing. The goal is to get both sides of the governing equation to be as close to equal as possible throughout the domain; we need L_{Burgers} because without it the solution produced by the network would not follow Burgers' equation. L_{Data} is set up similarly to L_{Burgers}; we take the mean squared error of the predicted u and the given initial and boundary conditions across the initial and boundary points. L_{Data} is necessary to ensure that the network outputs a solution to the specific IBVP we are trying to solve. Additionally, if function values are known inside the domain (for example, from experimental measurements), these values can be incorporated into L_{Data} to find a solution that matches them.

2.2 Training Process

The process of finding the learnable parameter values in a neural network is known as training. To minimize the loss function, stochastic gradient descent is performed with the learnable parameters as input. Training is an iterative process in which the data set is split into a set of mini-batches and the learnable parameters are adjusted with respect to only one batch at a time. In our case, the data set is the set of collocation points. A full pass through all the collocation points is called an epoch. A smaller batch size allows the model to learn more from each individual point but it takes more epochs to converge. A larger batch size requires fewer epochs to converge but may result in the model failing to capture the nuances in the solution due to falling into a local minimum.

2.3 Experimental Setup

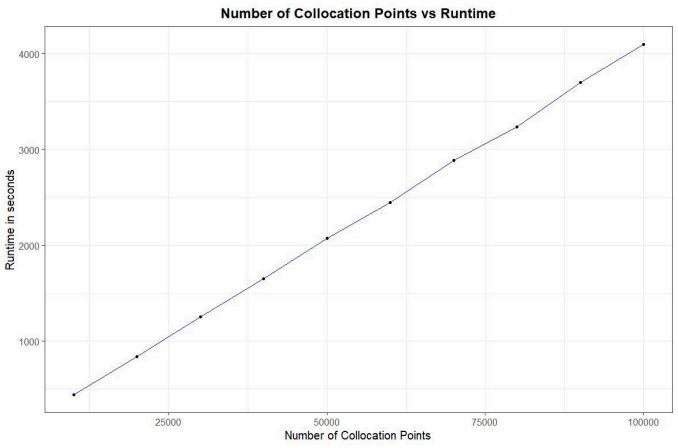
In order to run large numbers of experiments, we created a system for generating, storing, and loading large sets of parameters. A Python program was created that could generate all possible combinations of different parameter values. For example, the number of neurons per layer and the number of layers could be varied simultaneously to examine the interaction between them. The Python program produced a MATLAB file that would generate an array of parameter value structures. Finally, this array was iterated through and a new network was trained for each set of parameters. Each trained network's performance metrics and experimental parameters

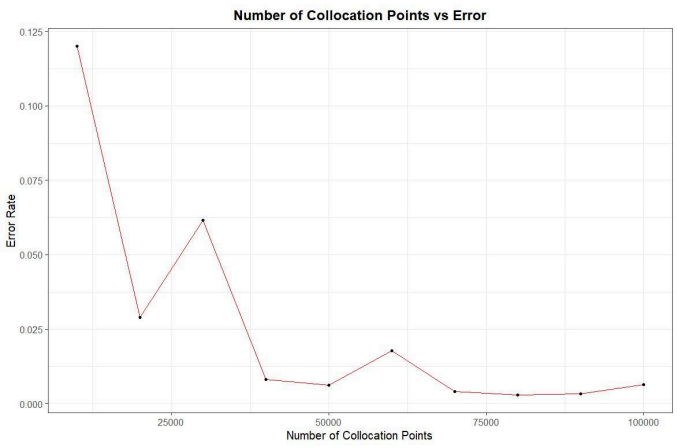
were written to a CSV file for further analysis. Additionally, each trained network's weights and biases were also saved. To measure a network's performance, we used the training time in seconds and the L2 error between the network output and the exact solution at the final time t = 1. Each experiment was run at least three times and the performance metrics were averaged across the runs. The experiments were performed on a dual-socket system with two 12-core Intel Xeon E5-2670V3 processors running at 2.3 GHz each, however each experiment was only run on a single core. Additionally, the networks were all trained for 1,000 epochs with 100 initial condition points and 200 boundary condition points. Viscosity was set to 0.0031 unless otherwise specified.

3 Results

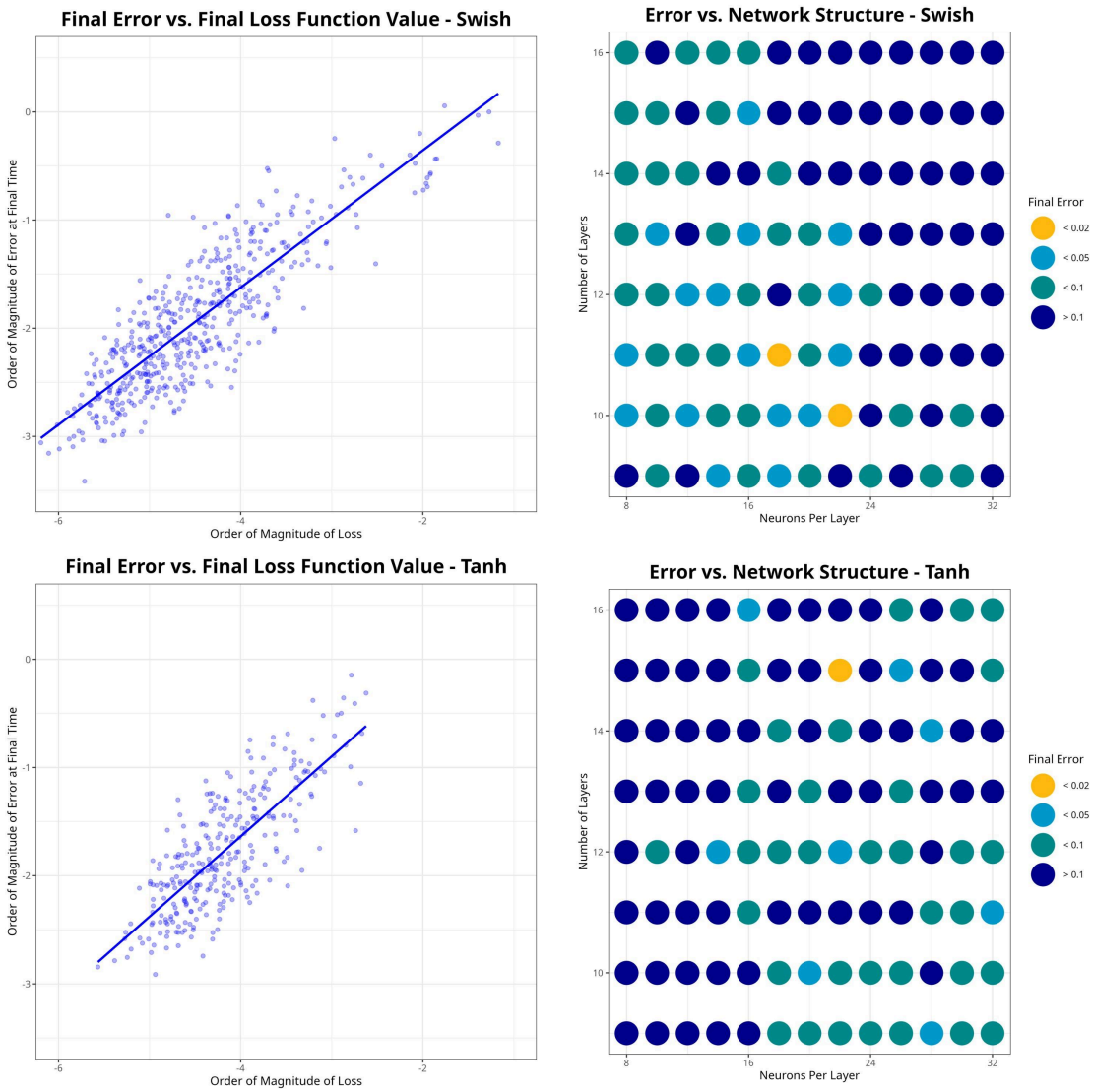
3.1 Variation of Number of Collocation Points

Starting from 10,000 to 100,000 collocation points, we see a general trend that error decreases as the number of collocation points increase. However, runtime increases, which is not surprising since more points requires more computational work.





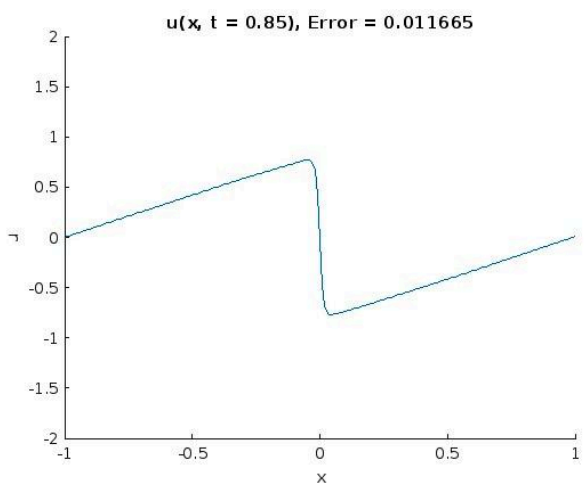
3.2 Swish vs. Tanh Activation

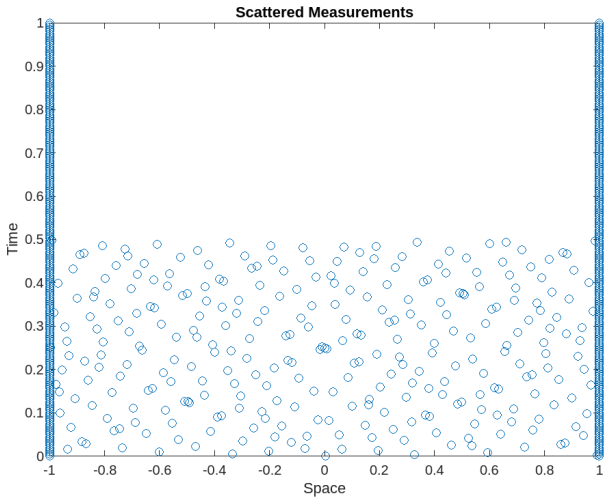


One important component of our research was the performance comparison between the tanh and swish activation functions. The graphs on the right show final error vs. number of neurons and number of layers. The scale goes from yellow

corresponding to the least error to dark blue indicating the most error. For the swish, the right side of the graph is mostly dark blue. This means that swish works best for a small number of layers and a small number of neurons. The lowest error for swish was achieved with 18 neurons per layer with 11 layers and 22 neurons per layer with 10 layers. The final loss vs. error graph for swish shows a strong positive correlation in the log-log scale, with a slope of 0.634. This means reducing loss by a factor of 100 reduces error by a factor of a little more than 10. The error vs. network structure graph for tanh shows an opposite trend of swish, with the left side being mostly dark blue circles. This shows that tanh works best for a larger number of layers and a larger number of neurons, with the minimum error achieved with 22 neurons per layer with 15 layers. The final error vs. loss graph is similar to the graph for swish, showing a strong positive correlation between loss and error in the log-log scale. The error for tanh decreases slightly more rapidly as loss decreases compared to swish with a slope of 0.71, but we found that swish is able to achieve lower loss overall. One thing to note is that when we have the same number of layers and neurons, swish takes fewer epochs to reach the same level of error as tanh, which means that swish is the faster activation function.

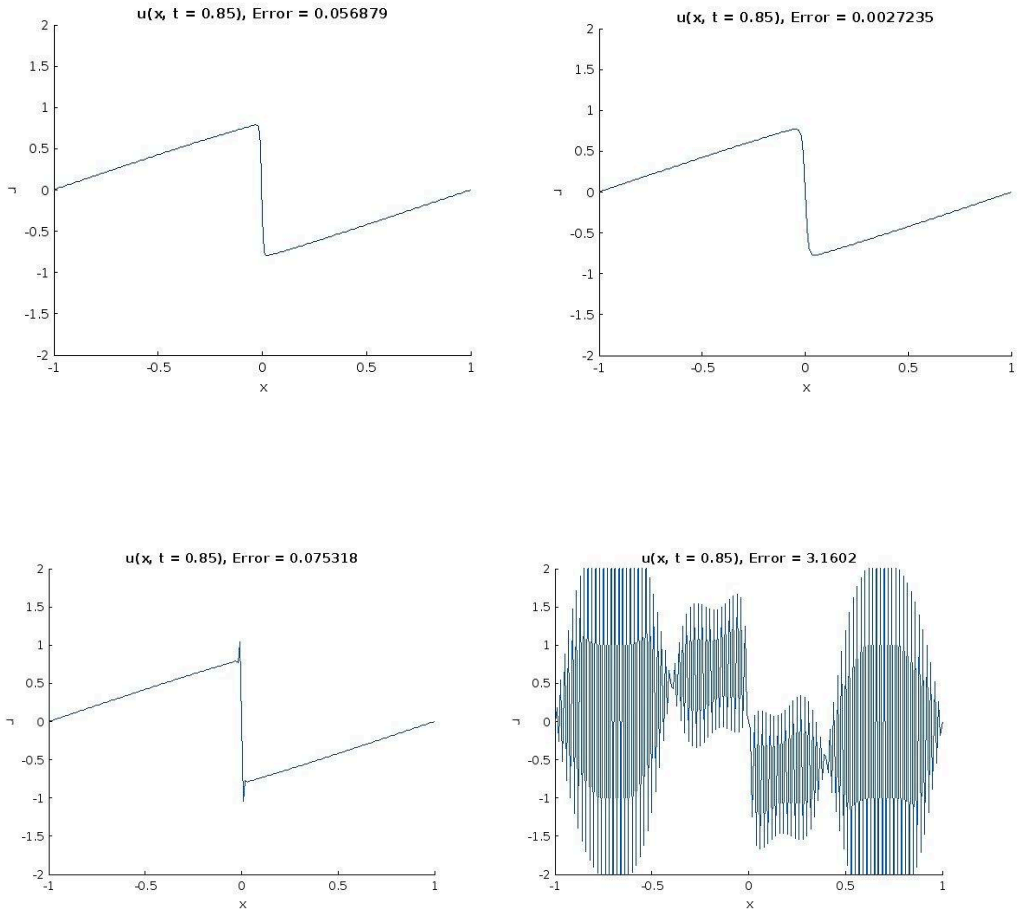
3.3 Incomplete Initial Condition





In this experiment, we tested to see if PINNs can produce reasonably accurate predictions even in the face of an incomplete initial condition. On the right is a graph of the points incorporated into L_{Data}, and on the left is the output produced by the neural network at time t = 0.85. The error at t = 1 was 0.012, which is reasonably low given the ill-posed nature of the problem. This indicates a possible application of PINNs for obtaining plausible solutions to ill-posed problems in differential equations.

3.4 PINNs vs Standard Finite Difference



In this investigation, we compared PINNs to a standard point-to-point finite difference method. The finite difference method used is first order in time, second order in space, with the timestep and spacestep both set to 0.01. The viscosity was altered from 0.0031 (left) to 0.0062 (right) and the behavior of the PINNs and finite difference method were compared. The PINNs curve (the top two graphs) still remains highly stable when the viscosity constant is doubled. However, the finite difference method curve (bottom two graphs) becomes less defined and exhibits erratic behavior when the

same change in viscosity is made. This highlights that PINNs is unconditionally stable, meaning that it can adapt to changing parameters well due to its machine learning backbone. On the other hand, standard finite difference is only conditionally stable, and requires extra care when adjusting parameters in the differential equation.

V = 0.0031	Error at Final Time	Runtime (seconds)
PINNs	0.0618	**490.732**
Finite Difference	0.0589	0.002

This table illustrates the differences in runtime and error for PINNs vs. the finite difference method. While PINNs have proven to offer unconditional stability under changing conditions, it comes at the cost of an extraordinarily high runtime. To achieve the same error, it takes 490.732 seconds to train the neural network, while the finite difference method takes only 0.002 seconds. This substantial difference in runtime is a critical weakness of PINNs, and presents an opportunity for future improvement.

4 Conclusion and Future Work

PINNs are a promising new technique for tackling difficult problems in differential equations. However, they are not a panacea. Their robustness even in the face of ill-posed problems is impressive, but the training time leaves something to be desired. Still, there is something to be said for a method that will nearly always work eventually; after all, computing time is much cheaper than engineering time. However, this research has just barely scratched the surface, with many more possibilities to explore. For example, training time could potentially be decreased substantially by using parallel computing, where each mini batch is processed simultaneously. It would also be interesting to try learning transfer, where a network already trained to solve one problem is used as a starting point on a different problem. It would also be worth seeing how PINNs compare to traditional numerical methods on higher dimensional problems. There is a lot of potential in this field, and we are excited to see how PINNs are used in the future.

5 References

Burgers, J. M. (1948). A mathematical model illustrating the theory of turbulence. Advances in applied mechanics, 1, 171-199.

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., & Yang, L. (2021). Physics-informed machine learning. Nature Reviews Physics, 3(6), 422-440.

MathWorks. (2023). Solve Partial Differential Equations Using Deep Learning. MATLAB and Simulink Help Center.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. arXiv preprint arXiv:1710.05941.