NNFDA - Neural Networks for Finite Difference Approximations

Hunter Johnson and Clayton Kramp
December 13, 2018

***Abstract*— For our final project, we propose NNFDA, a neural network approach to approximating differential equations using finite difference methods. As finite difference methods are extremely computationally expensive, and time exhaustive, a neural network can be trained to represent a model more efficiently. From our analysis and experiments, we have concluded three significant results: 1) NNFDA can store data for a model more efficiently, 2) NNFDA is quick to compute and 3) NNFDA predicts values within a training region well. These properties make NNFDA a great candidate for approximate modeling.**

I. Introduction

For this project, we purpose NNFDA, a neural network solution to Ordinary Differential Equations (ODE) and Partial Differential Equations (PDE). Scientists and Engineers work with models involving ODEs and PDEs every day, and this process can be time consuming. With scientific models, we often seek high accuracy, but this comes with a price of high computation time. Therefore, we introduce NNFDA, which while takes $\mathcal{O}(n)$ time to train, where n is the sample size, it is highly efficient as it only takes $\mathcal{O}(1)$ time to predict given a trained model. In this way, we can use NNFDA to sacrifice some accuracy for tremendous improvement in computation time. We first test our model's accuracy and efficiency with simple ODEs, and later move on to apply it to the Heat Equation and the Navier-Stokes Model. These are more complex equations that are frequently used by the Computational Fluidics community.

II. Description of Solution

We have developed a neural network that takes as input a linear spacing of x values for testing and training, and the corresponding y values for the given input. For ODEs, each x value represents a single corresponding y value, whereas for our PDE models, we have each x value represents a grid y. We have created differential equation (DE) classes, that for each DE function we are interested in, contains information of its initial value and a formula to compute its true solution. For ODEs with analytical solutions, the true solution is calculated simply with the analytical equation. For example, the differential equation:

$$\frac{dy}{dx} = x$$

Has the analytical solution:

$$y = \frac{x^2}{2}.$$

Now, if the ODE does not have an analytical solution (most ODEs), we evaluate the true solution by using `scipy.integrate.odeint`, a fast ODE solver for `python`. An example of a differential equation that we would use for this is:

$$\frac{dy}{dx} = x|\sin(x)|$$

Thus its true solution is computed using `odeint`. For our PDEs, we generated the true solutions using `MATLAB`, saved the outputs, and read them in with our `python` program to parse the values. Our decision to use `MATLAB` for solving PDEs is mostly due to the fact that `MATLAB` is optimized to work and handle scientific computing problems, and is a better environment to work with PDEs.

Now that we have the inputs for the neural network, we can customize our network to appropriately approximate our solutions. The parameters we can adjust are:

- Testing x values range and data size
- Training x values range and data size
- Depth of network
- Number of nodes at each layer
- Activation functions at each layer
- Epoch
- Batch size

III. Neural Network Architecture

While conducting our experiments, we have understood that the depth, nodes per layer, and epoch time necessary to reach good results varies significantly from model to model. For example, suppose we begin with a shallow network, that has the first layer with one node, a hidden layer with 10 nodes, and an output layer with 1 node. Then, for the equation $\frac{dy}{dx} = x|\sin(x)|$, we get the results as seen in Figure 1. With a naive implementation and fitting for a line with many curves, we have a horribly fit curve (this is as expected). However, we can improve our results by changing the network to have the topology 100-10-10-1 with relu activation functions. With this topology, we get the output as seen in Figure 2. Thus, we see here that the combination of adding nodes and increasing the depth of our model we can improve our results. From this example we learned that the optimal configurations of layers and depth depend on what equations we are solving. Since, ideally, we want to see maximum efficiency with high accuracy, we plan on using different hyper parameters for different models. Since efficiency is also important, if a model does not require a deep architecture, we should not add excess layers. Furthermore, we can adjust batch size and epochs as parameters. Toggling those values can also improve our

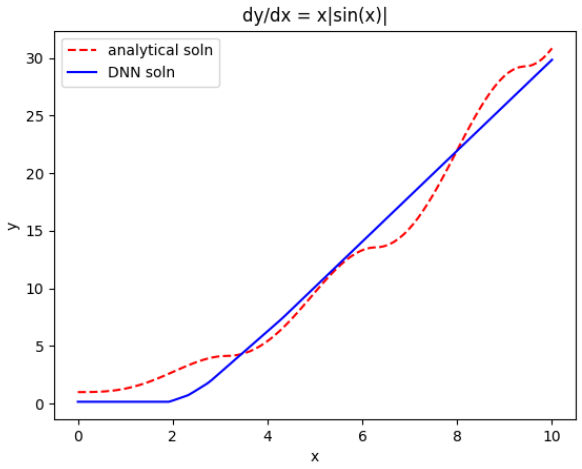


Fig. 1: Shallow network architecture

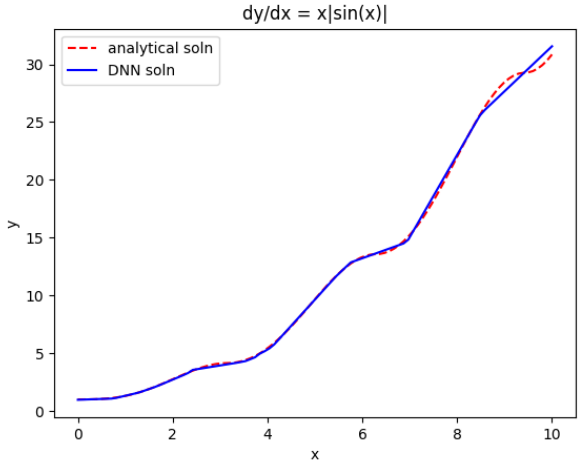


Fig. 2: Deep network architecture

accuracy without excessively deepening our network and should adjusted before adding extra layers.

IV. Multi-Parameter Adaptability

When working in simulations, we often have parameters found in equations. The following is an example:

$$\frac{dy}{dx} = kx$$

in which k is our parameter. Often times in mathematical biology, we have parameters that have a certain range that it can exist in. For example, in blood flow simulation, the density of blood is about 1060 kg/M^3, whereas the density of blood plasma is 1025 kg/M^3 and the density of blood cells are 1125 kg/M^3. Since this type of parameter variability exists naturally in simulations, we wanted to be able to construct one neural network that can train on multiple parameter constants so that it can adapt to inputs. Thus, after training on some constants, we can use the same model to get approximate computation for any parameter value in that range. We will discuss our results of multi-param adaptability in the next section.

In our implementation for this solver we include a way to input multiple constant values for training as well as multiple constant values for testing. Due to the need to train on a great number of constant values we introduced further parameters to our code. These parameters are the lower bound, and upper bound for the training constants as well as the number of training constants to use in producing the training data. We also include a parameter that consists of the testing constant values. Since it is possible that any given differential equation has a need for a greater number of constants our program is smart enough to alter the input dimension of the neural network accordingly. Once the constants are produced for the function they are used in the production of the training data as well as the ground truth data for testing. It is important to note, that the number of different constants as they are added are just another dimension in the feature space of the model. This means that, for example, if our function has has one constant as a parameter, as does $\frac{dy}{dx} = kx$, our network now accepts a 2 dimensional vector as its input.

V. Results: ODE

First, we will examine our results for the differential equation

$$\frac{dy}{dt} = x(1 + |x|)$$

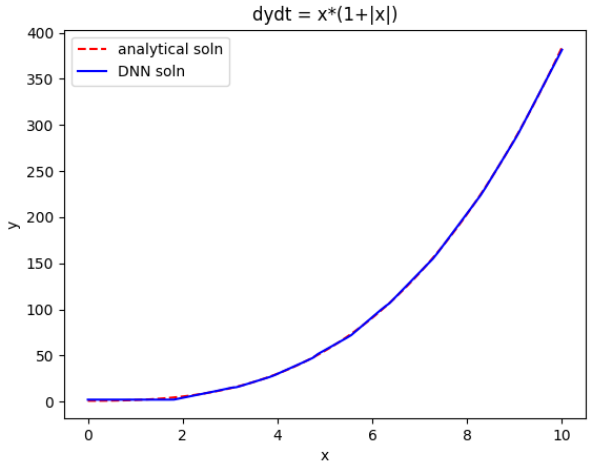


Fig. 3: Neural network approximation over the true solution

The graph in Figure 3 reflects the solution we get when we solve this ODE with our neural network. Using `odeint`, we were able to compute the values for y in 0.00035 seconds, and our neural network took 0.0234 seconds to run the prediction, 66.85 times slower. For a simple problem as this one, we definitely see that the neural network approach is much more costly. In this example, the average error along the curve was 1.045%. Now, we examine a second order ODE:

$$y'' + y' + 2y = 0$$

To solve this using `odeint`, we must first convert this into a system of first order ODEs. Below is our result:

$$\begin{aligned} x_1' &= x_2, \quad x_1(0) = 1 \\ x_2' &= -2x_1 - x_2, \quad x_2(0) = 0 \end{aligned}$$

We solve the system, and approximate it with our neural network. The graph is shown in Figure 4.

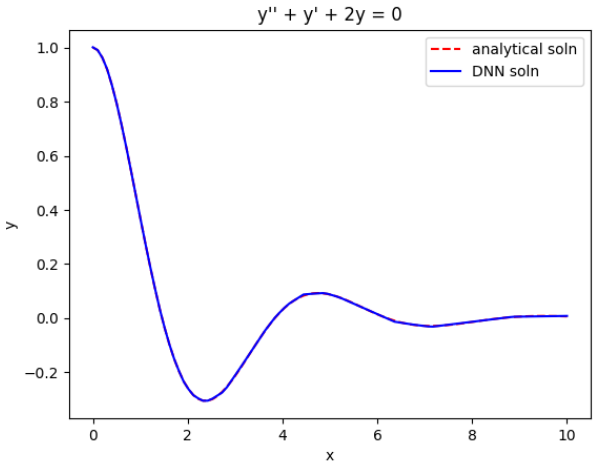


Fig. 4: Neural network aligns curvature along the true solution

For this problem, `odeint` took 0.001677 seconds to compute, whereas our neural network took 0.03097 seconds to predict, 18.46 times slower than `odeint`. However, notice that when we moved from first order to second order, `odeint`'s computation time increased ≈ 5 times whereas the neural network took ≈ 1.3 times more time (due to more layers). Thus, we immediately see that neural network computation times can be faster for multi-dimensional, multi-parameter problems. Therefore, we suspect that we will see opportunities of performance improvements when applying to PDEs, as they will have multiple variables to solve for. Particularly, we hope to see improvements while analyzing a 2D grid in x and y with time component t for our Navier-Stokes model.

Next, we analyze our results for multi-parameter adaptability. Figure 5 shows promising results for adding constants to the simple differential equation $\frac{dy}{dx} = kx$.

In this example we trained on 15 linearly spaced constants withing the range [0, 10] and shown is the actual output versus the DNN approximation for several different constants in the training domain. Although, the constants that were tested were part of the training domain, they were not the actual constant values used in training. Due to these results it is reasonable to say that our model can adequately represent functions containing constants in the training range.

One of our hopes was that we could train on a certain range of data and then be able to produce decent approximations for domains outside of this domain. For example, we would like to be able to train on x values of [0,10], and constant values from [0,10] and then be able to produce

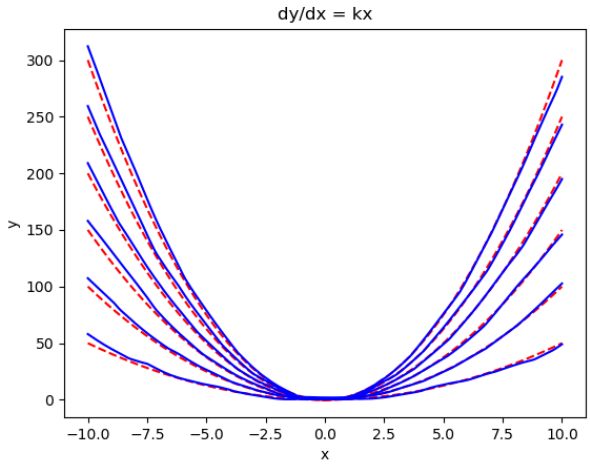


Fig. 5: An example of approximating $\frac{dy}{dx} = kx$ within the training range.

accurate results when testing on x values from [90,100] and constant values from [25,30]. Our current results, unfortunately, do not exhibit this behavior to the extent that we have hoped. Figure 6 shows what happens when we step out of the domains for both x values and constant values.

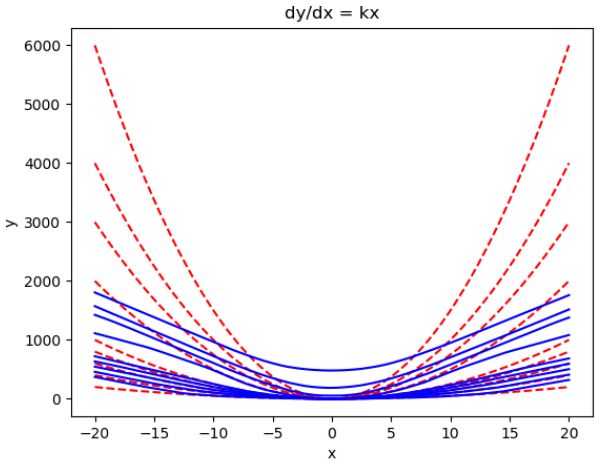


Fig. 6: An example of approximating $\frac{dy}{dx} = kx$

In Figure 6 above the function was trained on the x domain [-10,10] and the constant domain from [0,20]. The testing data for this figure was produced on the x domain [-20,20] and with k values 2,4,6,8,10,20,40,60. Figure 6 shows several important things. The first of which is that when we leave the domain for any of our inputs by to great of an amount the approximation gets terrible. This happened for both the x domain and the k domain. The top two blue lines in the figure correspond to the k values of 40 and 60 respectively. As these values correspond to constants that are far out of what we trained them on they never produce results that are accurate enough to use even when they are in the domain

of x values that were trained on. Correspondingly, when we have constant values that are within the k training domain when we leave the x training domain, values from [-20,-10] and [10,20] the approximation breaks down. Given those two observations, of course, when we leave the domains for both what we have trained x on and what we have trained k on the approximation gets terrible. This can be seen when looking at the top blue line towards the edges of the plot in Figure 6.

The former analysis does not give our model quite enough credit, as we have analyzed how bad it is when we leave our training domains by vast amounts. Figure 6 also shows that when we leave our training domains by smaller factors of their size our approximations still remain reasonably accurate.

We can also examine the effect that data size has on our model. In machine learning problems, data is what allows us to get highly accurate approximations. With large amount of data, our network has more information to use to apply appropriate weights to its edges. We experiment with Figure 4, and reduce the testing data size from 9000 to 1000, and we receive the result seen in Figure 7.

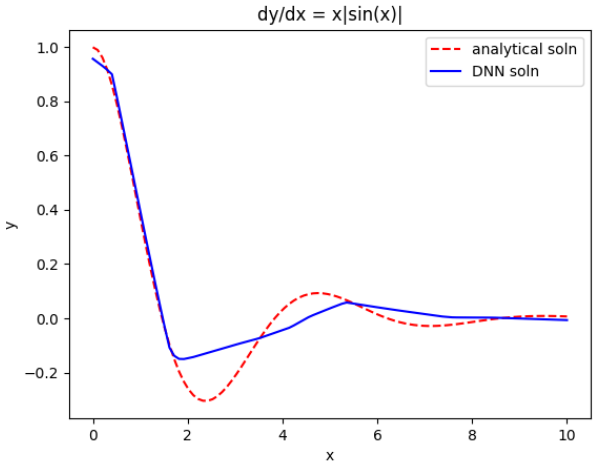


Fig. 7: Neural network with low data size

With low data size, we see a significant reduction in accuracy. However, we do see by increasing our epoch times sufficiently, our model has more time and iterations to improve its solution and actually provides similar results with Figure 4 after enough time. However, this can be very time costly, thus understanding a good balance of data size and epoch time is critical for high performance.

The great component about working with equations, is that if we want more data, we can simply increment our data size parameter which decreases the spacing between values to increase the total number of inputs.

VI. Results: Heat Equation

The heat equation is a parabolic PDE that describes the distribution of heat over time. We examine the Heat Equation as it is an important model for any scientific computing engineer, and it is frequently used in Physics, Biology, and Mathematics. The equation is the following:

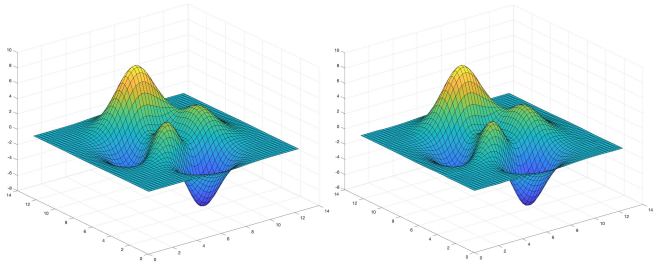
$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \tag{1}$$

Where u is the heat, t is time, α is the diffusion coefficient, and ∇ represents the gradient over our space. For our implementation, we used a two dimensional space, thus the model is governed with the following equation:

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0 \tag{2}$$

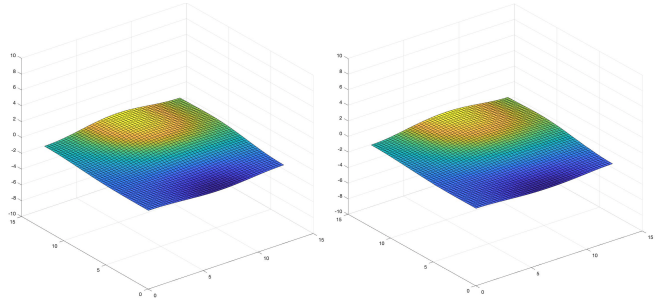
In order to evaluate this surface for each time step, we make small incremental time steps of $\Delta t = 0.01$ over the interval $[0, 10]$ with a diffusion coefficient of 0.5. We solve the heat equation given the above parameters on `MATLAB`'s `ode15s` solver, an efficient system of DE solver. We use `MATLAB`'s default `peak` function, obtained by translating and scaling Gaussian distributions, to create an initial temperature distribution.

After running and evaluating the model on `MATLAB`, we save its outputs to be parsed by NNFDA. We then train the model, and test its accuracy at the trained points using mean squared error. We then also test its accuracy at non-trained points inside of the training region (inside $[0, 10]$) and on points outside of the training region. The following is an image result of our graphs at $t = 0$.



(a) ODE15s Result (b) NNFDA Result

We can also examine the resulting graph at $t = 10$ to ensure that they have both reached similar conclusions as to what the model should become:



(a) ODE15s Result (b) NNFDA Result

We indeed see from a graphical and structural perspective the results are extremely similar.

A. Analysis

In order to analyze our results of NNFDA for the heat equation, we will individually analyze its accuracy, time, and space performance.

1) Accuracy: We begin by analyzing the model's accuracy on trained points. We will use Mean Squared Error in order to judge the quality of our NNFDA approximation. We select five linearly spaced points across the trained region to test the accuracy of our model.

Time Point	MSE
0	6.057e-04
2.5	2.101e-03
5	2.170e-05
7.5	1.822e-04
10	1.202e-03

From the above table, we see the MSE value ranges vastly for different time points. However, we do see that overall, the accuracy is high, and from the graphs we saw previously, the structure and shape of the graphs are maintained. We can also examine the difference graph shown in Figure 10: The difference graph shows that the maximal errors come

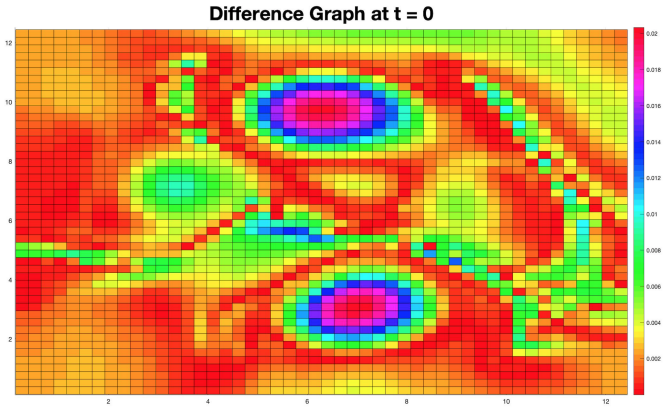


Fig. 10: Difference Graph at $t = 0$

from the peaks that we have in our heat distribution. At the extrema, we see that we have high degree of error, while on the relatively flat locations the error is not very high. This shows that 1) our model can handle heat equations with multiple extrema 2) our model has very little error when our surface is relatively flat.

2) Time: For our purposes, we have three methods of retrieving heat distribution information at a particular time point.

- Recompute from scratch up until desired t
- Write to memory grid values at every time step to be read in at a later time
- Predict using NNFDA

We shall examine the runtimes of each of these methods for our heat equation for our time value $t = 10$. We arrive at the following results:

Method	Time (seconds)
Recompute	93.51
I/O	0.00157
NNFDA Train	396.4983
NNFDA Predict	0.101132

As expected, we do see that saving all time step information to memory and retrieving it later for future use is indeed the fastest operation. However, as we will see later, this is perhaps the most inefficient method to store data, and is not really a viable option. Then, we are left with running from scratch and our NNFDA model. We see that while NNFDA takes a decent amount of time to train, that it is extremely fast to predict provided a model. Pair that with the fact that NNFDA provides accurate results shows that our model satisfies both the time and accuracy constraints.

3) Space: In our current implementation, we have our `MATLAB` code save the grid information at each time step. The grid size is 50×50 with doubles, and there are 1000 of said files in order to store the exact grid values at each time step. It is important to note that even with the Finite Difference Method, it requires a past and a present grid to be saved. Thus, we reach the following storage outcomes:

Method	Space
Recompute	40 KB
I/O	19.7 MB
NNFDA	2.9 MB

From the table, we see that NNFDA can store data with only 14.72% of the space required by I/O. It is also important to note, that as long as we do not change the topology and hyperparameters of our NNFDA, its byte size will never change. Therefore, in this specific problem, its size will always be 2.9 MB regardless of the training samples, batch sizes, and epochs we use to train the model. This means that this model is fully adjustable and flexible to be applied to any heat equation problem with any surface. We do see that the recompute method has the smallest storage requirements, however it is important to remember the tremendous computation time that recompute takes. Therefore, we show that NNFDA is an excellent method to store approximate information of a model that is fast to compute and small in storage.

B. Non-Trained Points

We also test our model's accuracy on points within $[0, 10]$ but not trained on it. We can evaluate this by reducing our time steps even further to $\Delta t = 0.005$ to develop a 'true' solution, and approximate with NNFDA. Below are some random points we have selected to evaluate the result:

Time Point	MSE
1.565	7.6730e-04
2.345	4.9030e-04
8.675	3.6334e-04

From the above table, we see that our model continues to perform extremely well even for values that it has not been

trained on. This makes NNFDA a great candidate for better storage of models.

C. Out of Range Points

We shall also test our model to see how well it can handle out of range values (values outside of the trained $[0, 10]$ region). We arrive at the following results:

Time Point	MSE
11	0.0042
12	0.0093
15	0.0451
20	0.2565

Our model shows that it performs relatively well if we are only slightly out of our trained region, however our model begins to fail greatly as we move further away from the trained region. This is a rather expected result as we had a similar experience when modelling ODEs. However, as we were able to show that NNFDA provides strong results when inside the training region and outperforms traditional methods by being small and storage and fast in computation, NNFDA becomes a reliable opetion for approximate modelling of the Heat Equation.

VII. Results: Navier-Stokes Model

The Navier-Stokes Model is one of the most important equations in modern engineering and physics. The model is used in applications such as weather simulations, ocean current simulations, flow simulations for chemical engineering, and more. In this paper we use NNFDA to model the incompressible Navier-Stokes equations in two space dimensions. These are given by:

$$u_t + p_x = -(u^2)_x - (uv)_y + \frac{1}{Re}(u_{xx} + u_{yy}) \tag{3}$$

$$v_t + p_y = -(uv)_x - (v^2)_y + \frac{1}{Re}(v_{xx} + v_{yy}) \tag{4}$$

$$u_x + v_y = 0 \tag{5}$$

For the sake of our model this is defined on a rectangular domain $\Omega = [0, l_x] \times [0, l_y]$. The domain is fixed in time, and the following define no slip boundary conditions on each wall:

$$u(x, l_y) = u_N(x), v(x, l_y) = 0 \tag{6}$$

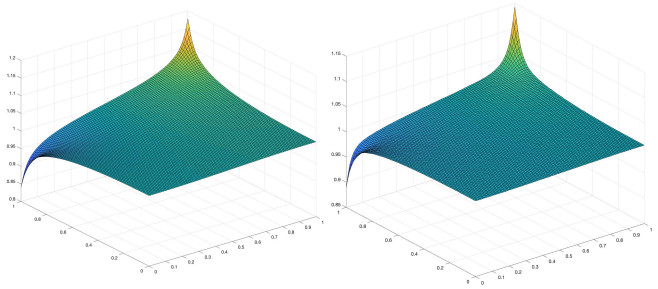
$$u(x, 0) = u_S(x), v(x, 0) = 0 \tag{7}$$

$$u(0, y) = 0, v(0, y) = v_W(y) \tag{8}$$

$$u(l_x, y) = 0, v(l_x, y) = v_E(y) \tag{9}$$

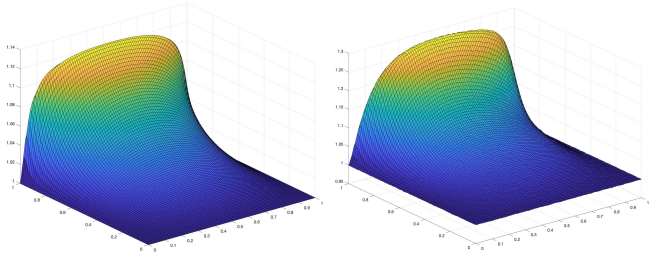
Equations 3 and 4 are the momentum equations and describe the pressure and velocity fields within space and time. Note that in equations 3 and 4 the term p represents pressure, the terms (u, v) represent the velocity field.

In the following figures we present the results from the output of the model given in the paper [1] versus our NNFDA approximation. At a given time in the simulation the points solved for are presented in a 90×90 mesh in which any grid point has an associated magnitude of the stream function, along with a pressure value. The code producing the ground truth data for the problem uses finite difference methods to do this approximation. The goal of our NNFDA model is to take a time in the simulation and then output the stream function magnitude and pressure, as does the original model. Due to our interest in two variables, flow and pressure we composed our NNFDA model with two networks. The first network predicts the pressure and the second predicts the stream function.



(a) True Result (b) NNFDA Result

In Figure: 11b we see the visual results for the stream function. From this plot we see that visually they appear to be quite similar and structurally identical. From this graph we can see that this function is quite hard to model due to the fact that it has massive spikes in magnitude at its top two corners and many close to zero values elsewhere. When solving around these points of high gradients, finite difference methods generally have problems when their time steps become too large. It should be noted that the MIT model used to produce our training data used time steps that were small enough to not cause this breakdown in accuracy. We hypothesize that because a DNN does not use slope values to do its approximating for future time steps it fundamentally will not have this pitfall.



(a) True Result (b) NNFDA Result

In Figure: 12b we see a plot of the pressure of the original data versus the output from our model. From this we see that, similarly to Figure: 11b, visually the NNFDA approximation is quite good.

A. Analysis

To analyze our results for NNFDA for the Navier Stokes Equations, we must look at the accuracy, time, and space performance.

1) Accuracy: To begin we analyze the results of our NNFDA on trained points in the model. Below, in the table, we see the MSE of both the flow and pressure values at five linearly spaced points on the trained domain. Interestingly, as the MSE for pressure goes down the MSE for the stream function goes up. The MSE for flow is also significantly higher than the MSE for Pressure. This could be a result of the stream function having gradients that are significantly higher than the change in pressure.

Time Point	MSE Pressure	MSE Flow
0	1.3710e-04	0.0037
0.25	8.2628e-05	0.0288
0.50	5.7743e-05	0.0371
0.75	4.2168e-05	0.0506
1	3.3164e-05	0.0739

2) Time: For our time experiments we test to see how much computation time it takes to retrieve data from $t = 1$. The recompute method will run finite difference methods from $t = 0$, the I/O method will retrieve data from memory, and our NNFDA model will predict provided $t = 1$.

Method	Time (seconds)
Recompute	5.925
I/O	0.00249
NNFDA Train	42.30
NNFDA Predict	0.0567

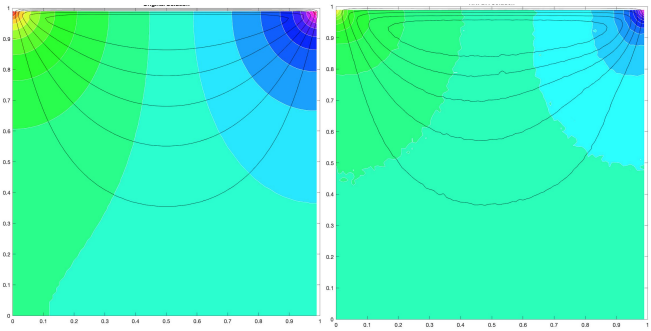
From our above results, we see that while it takes time to train our model, it is extremely fast to predict values, over 104 times faster than to recompute from scratch. This alone is a significant detail as the model allows for quick retrieval of information.

3) Space: For this problem, we constructed two NNFDA models for pressure and for flow. We saved to memory 100 time samples of the 90×90 grid for both pressure and flow of our Navier-Stokes model. Note that with finite difference methods it requires the grid information of the current and previous iteration. We get the following results:

Method	Space
Recompute	226 KB
I/O	11.3 MB
NNFDA	4.2 MB

From the above table, we see that similarly to our heat equation model, NNFDA was able to achieve good results using only 37% of the memory required by the I/O method. This is a significant reduction in memory requirements, especially for a complicated and involved model like Navier-Stokes. As expected we do see that the recomputation method takes the least amount of space in exchange for significantly more computation time.

4) Visualization Analysis: Now that we have numerical analyses of the results, we can examine the visual results that we get.



(a) True Result at $t = 0$ (b) NNFDA Result at $t = 0$

From examining Figures 13a and 13b, we see that they have very similar features. We see that the flow model's structure (contours with colors) is maintained with very similar values (colors). The pressure (parabolic contours) are also very closely matched. However, we do notice some jagged edges in our graph. We have discovered that as we increase our node count for our model, while it does increase the byte size of our model significantly, it does reduce the jagged edges that we see in 13b. While smoothening the jagged edges does not lead to much accuracy improvement numerically, visually and structurally this could be an important detail.

We can also examine these 'bumps' in the surface plot as well:

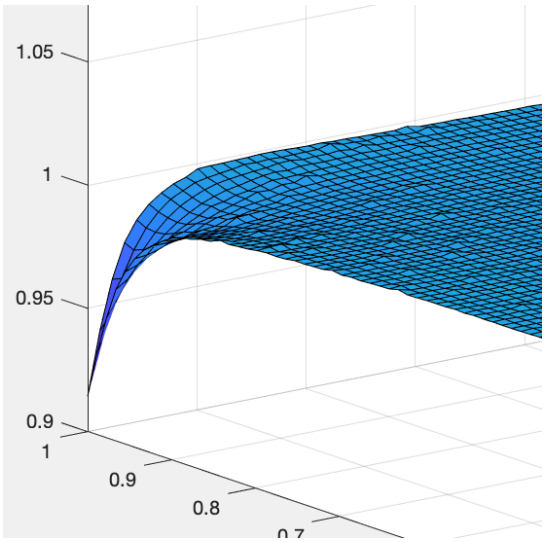


Fig. 14: Zoomed in Surface Graph with bumps

From Figure 14, we see that the neural network adds random noise, and inaccuracies to the model. While the overall structure is maintained, it is important to note that our model incorporates these types of unwanted details when we limit the byte size of the model.

VIII. Future Work

A. Out of Range Prediction

One goal of our project that we were not fully able to achieve is the out of range prediction. Our NNFDA model

performs well for values within the training region, however begins to fail as we extend out of the training region. We can improve our model's range by simply training on a larger domain, however this does lead to more computation time as our finite difference methods takes a significant amount of time to train for extended time periods. We would like to do further work on our model to be able to predict outside of the training region by implementing interpolation methods, or other history-based approximation schemes.

B. Accuracy and Space

We believe that with additional time, we can improve the accuracy of our model by finding better hyperparameters, and topologies that reflect the behavior of the original equation. With this, we would also like to develop a function to determine the minimum number of parameters to achieve some ϵ approximation to the model in order to limit the byte size of the model.

IX. Related Work

A. Parrot Transformation

The Parrot Transformation (PT) is a neural acceleration method proposed by University of Washington and Microsoft Research [3]. PT selects a region of imperative code and creates a neural network to simulate it. It produces approximate results in significantly faster speeds. This transformation works very well for code that is repeated heavily and has high computation. In our project, NNFDA has a similar behavior, where it mimics the DE model that we are interested in, and produces approximate results. Model simulation is extremely costly and boils down to a `for` loop of high cost, thus makes it a great candidate for PT and implementation of NNFDA.

B. Cellular Neural Networks

In the book Cellular Neural Networks [4], the authors discuss the potential of Cellular Neural Networks (CNN) in a variety of fields including modelling and nonlinear dynamics. They have shown that using CNNs can drastically reduce the computation time of traditional finite difference methods, and that a neural network can provide good approximation in less time. They discuss the process of breaking down a PDE or model to a system, and how to implement the model on a CNN. They do not discuss much further than one air-quality modelling example.

X. Conclusion

With our purposed NNFDA model, we were successfully able to approximate a PDE model with little space, and low computation time. We first began with modelling ODEs, which proved that a neural network approach to solving DEs could provide significant results for more complicated models. After testing with the Heat Equation and the Navier-Stokes Model, we have shown that NNFDA provides significant speedup in prediction time with relatively low storage needs. Furthermore, NNFDA is completely independent of models, coefficients, and time steps used, and is a flexible model to approximate any mathematical model. With our multi-param input, NNFDA becomes very adaptable as it can provide quick approximation feedback for any trained model. Overall we believe that our implementation of NNFDA lead to significant time and storage improvement of PDE models, and that it has potential to provide a better approach for scientific computing.

XI. Code

Accessible at https://bit.ly/2QLmpna

XII. Citations

[1] Benjamin Seibold. (2008). A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains. Massachusetts Institute of Technology

[2] Dissanayake, M. and Phan-Thien, N. (1994). Neural-network-based approximations for solving partial differential equations. Communications in Numerical Methods in Engineering, 10(3), pp.195-201.

[3] Esmaeilzadeh, Hadi and Sampson, Adrian and Ceze, Luis and Burger, Doug. (2012). Neural Acceleration for General-Purpose Approximate Programs. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 449–460.

[4] Gabriele Manganaro, Paolo Arena, Luigi Fortuna. (2012). Cellular Neural Networks: Chaos, Complexity and VLSI Processing.

[5] Google Research. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.

[6] Lagaris, I., Likas, A. and Fotiadis, D. (1998). Artificial neural networks for solving ordinary and partial differential equations. IEEE Transactions on Neural Networks, 9(5), pp.987-1000.

[7] Lee, H. and Kang, I. (1990). Neural algorithm for solving differential equations. Journal of Computational Physics, 91(1), pp.110-131.

[8] Parisi, D., Mariani, M. and Laborde, M. (2003). Solving differential equations with unsupervised neural networks. Chemical Engineering and Processing: Process Intensification, 42(8-9), pp.715-721.

[9] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud. (2018). Neural Ordinary Differential Equations.