4-19-2021

Estimating the Heat Equation using Neural Networks

Nathan Jordre
College of Saint Benedict/Saint John's University, njordre001@csbsju.edu

Estimating the Heat Equation using Neural Networks

College of Saint Benedict/Saint John's University

by Nate Jordre
April 2021

Project Title: Estimating the Heat Equation using Neural Networks

Approved by:

DocuSigned by:
Michael Heroux
A5DAF32937404F9...

Mike Heroux
Thesis Advisor, Scientist in Residence

DocuSigned by:
Jeremy Iverson
F7B1847F6A6647D...

Jeremy Iverson
Faculty Reader, Assistant Professor of Computer Science

DocuSigned by:
Robert Hesse
70E183EC63084BF...

Robert Hesse
Faculty Reader, Associate Professor of Mathematics

DocuSigned by:
Noreen Herzfeld
993392756B3A48B...

Noreen Herzfeld
Chair, Department of Computer Science

DocuSigned by:
Lindsey Gutsch
DA28EB4CB48D4FA...

Lindsey Gunnerson Gutsch
Program Director, CSB/SJU Distinguished Thesis

Abstract

Partial Differential Equations have important uses in many fields including physics and engineering. Due to their importance, heavy research has been done to solve these problems efficiently and effectively. However, some PDEs are still challenging to solve using classical methods, often due to the dimensionality of the problem. In recent years, it has been thought that neural networks may be able to solve these problems effectively. This research assesses how well a neural network can estimate a simple example PDE, the heat equation, as well as the practicality of doing so.

Contents

1 Introduction

Partial Differential Equations (PDEs) have many important applications in scientific fields including physics, engineering, finance and more. They are often used to model physical phenomena such as the diffusion of heat or sound, the flow of fluids, electrodynamics, and other physical problems [9]. Due to the importance of PDEs, there already exists a number of methods to approximate the solutions to these problems. However, many of these methods become less practical as the dimensionality of the problems increase. Due to this, there is still a large amount of research being done on additional methods for solving PDEs. One research area that has received a lot of attention lately is the use of nerual networks for solving PDEs. Neural networks are an important sector of machine learning and are modeled after the human brain. Neural networks are promising for approximating the solutions of PDEs due to their strong function approximation capabilities [8].

There are many important PDEs, but this paper will focus on the heat equation. The heat equation describes the temperature distribution and diffusion of heat within an object. The majority of my research dealt with the 1-dimensional heat equation because data was most readily available in this dimension. Using the 1-dimensional heat equation also serves as a starting point for assessing the ability of neural networks to approximate the heat equation. Because the 1-dimensional version is the most simple, it should also be the easiest to model with a neural network, providing a base estimate for how well neural networks can approximate the solution to PDEs.

This paper will report how accurately different types of neural networks can estimate the heat equation. A variety of networks are used, including standard artificial neural networks with a variety of activation functions, a radial basis function network (RBF network), and simple example of a "physics inspired neural network".

2 Gathering Training Data

In order to estimate the heat equation with neural networks, first you must have sample solutions to use as training data. Fortunately, in a series of lecture videos, University of Utah professor Kody Powell

explained and showed how to solve the 1-Dimensional heat equation numerically using python [10, 11]. By making some small modifications to his script, I was able to use this to generate the training data I needed to train my model. Taking a closer look at the particular problem this script is solving, it is important to take note of some characteristics. The script simulates the flow of heat in a wall where heat only flows in the x direction. Throughout the different simulations, some things are held constant: the wall is always 10 cm thick, and it is always split into 20 uniformly sized nodes (each which will have its own unique temperature). Additionally, each node begins the simulation with an initial temperature of zero degrees. The thermal diffusivity of the wall is held constant. The wall has heat applied at both boundaries, and the temperature at either boundary is held constant throughout a single simulation. Each simulation was run for a total of 30 seconds, with the temperature of each node updated every 0.1 seconds. So for this set of conditions, the simulation finds a temperature at each of the 20 nodes at every 0.1 second time step, up to 30 seconds. The neural network will seek to do the same: give a prediction for the temperature of a given node, at a given time, with the given initial boundary temperatures. To gather training data, a number of simulations were run, each with unique pairings of boundary temperatures.

3 Artificial Neural Networks

The first model used with the 1-Dimensional data was a basic Artificial Neural Network, also known as a Multi Layer Perceptron Network (MLP Network). This model consists of a number of layers, each layer containing a number of nodes/neurons. The nodes are densely interconnected between layers, that is, each node has a connection from every node in the layer before it, and each node has a connection to every node in the layer after. Figure 1 shows the basic architecture of a neural network; this particular network has three layers with eight nodes in the input and middle layers and one node in the output layer.

Each node receives an input and produces an output. The type of input depends on the layer, the first layer will receive the training/testing data. All other layers will receive input that is based on the outputs of nodes in the previous layer. The values are passed

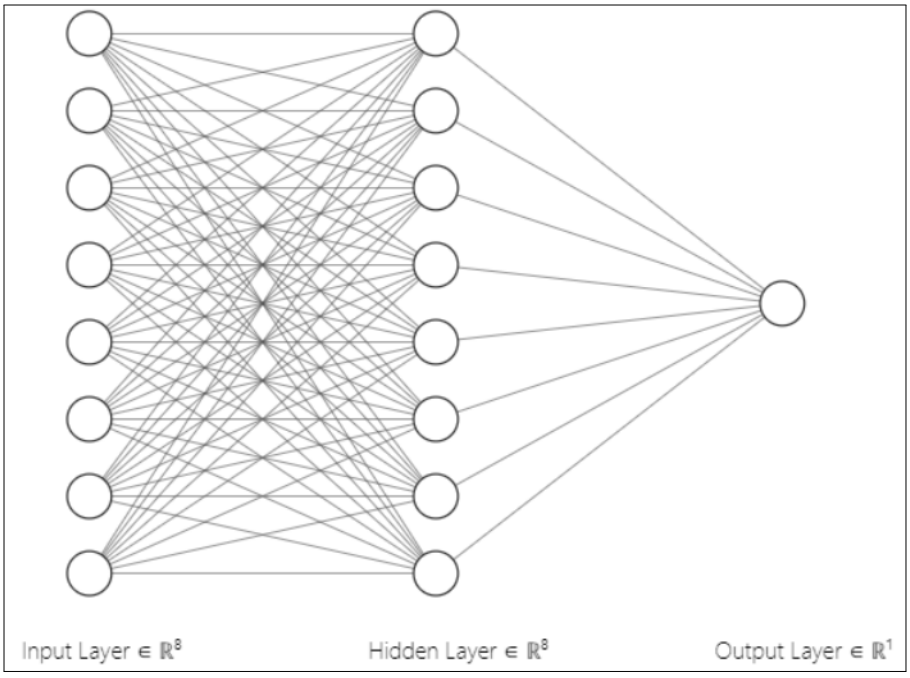


Figure 1: A basic three layer neural network, with 8 nodes in the input and hidden layers, and one node in the output layer.

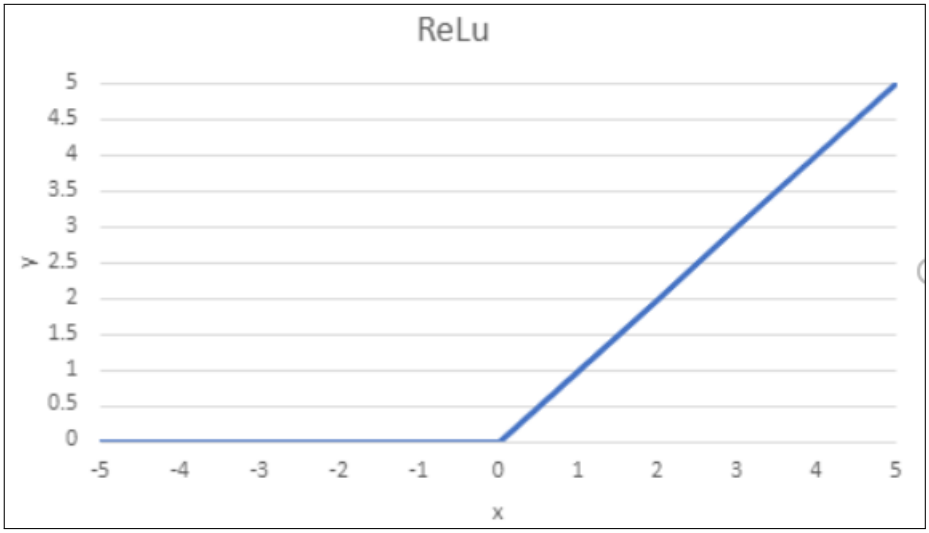


Figure 2: The ReLu activation function.

between layers through connections, where each incoming connection is associated with a weight [6]. Weights essentially control the strength of the connection between two nodes [4]. Once the node has received all inputs, it must calculate the value of the node and determine whether the value meets the threshold for the neuron to fire or be considered activated. The value is calculated as follows:

$$x = \Sigma(Weight * Input) + bias \tag{1}$$

The bias value is simply used to shift the result either positively or negatively [3]. Since x has infinite possible values, an activation function is applied to determine whether the neuron is activated or not. There are many different functions used, I experimented with three: ReLu, Sigmoid, and tanh.

ReLu is likely the most commonly used activation function. One of ReLu's greatest benefits is its simplicity; the equation is as follows (Figure 2):

$$A(x) = max(0, x) \tag{2}$$

If x is greater than zero, then there is no change to the value of the neuron, and it is activated with its full value. If the value is less than zero, then the neuron is not activated at all. This can also provide benefit due to the concept of "Sparse Activation". Since such a large range of values ($x <= 0$) will cause a neuron to not be

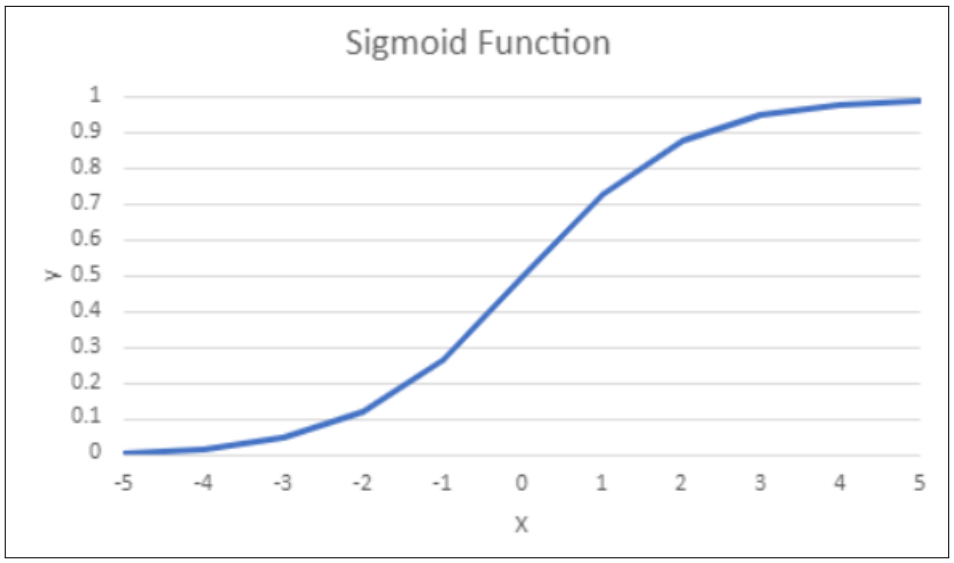


Figure 3: The Sigmoid activation function.

activated, layers further on in the network will receive less inputs because the network will have less activated neurons. This sparse activation results in lighter computations[14].

Sigmoid is another common activation function, and it acts very differently compared to ReLu. While ReLu is uncapped in the positive direction, and bounded to zero for all negative inputs, sigmoid is different in both of these cases. It has a strict range, (0, 1). The formula is for the sigmoid function is this (Figure 3):

$$A(x) = 1/(1 + e^{-x}) \tag{3}$$

The tight range offers the advantage that activations will not "blow up", reducing the influence of the more extreme inputs.

Tanh is the final activation function I experimented with during this phase of the project. It is very similar to the sigmoid function (Figure 4):

$$A(x) = tanh(x) = 2/(1 + e^{-2x}) - 1 \tag{4}$$

Like sigmoid, the activation range is tightly bound, but tanh allows for values between (-1, 1). Sigmoid and tanh both lack the characteristic of ReLu where high positive values are represented at their full intensity.

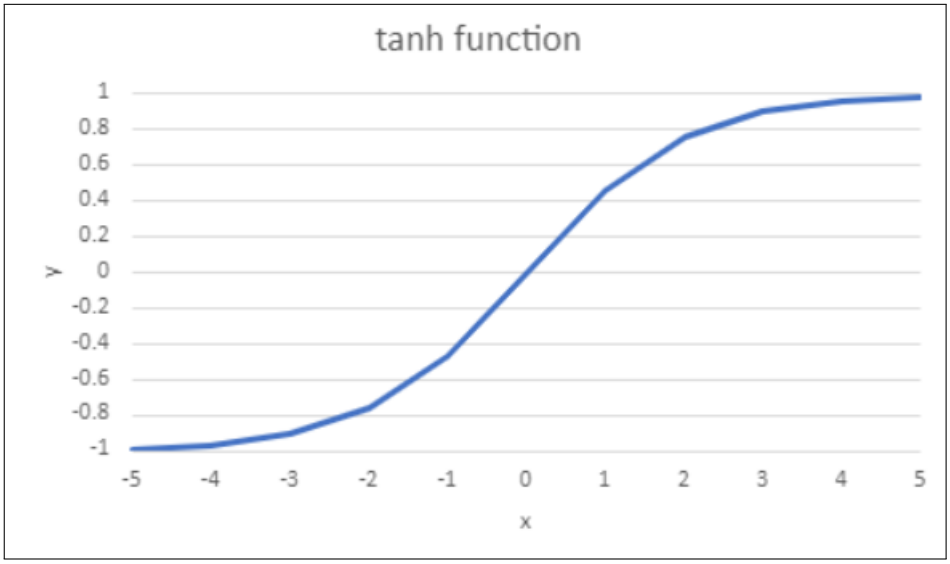


Figure 4: The Tanh activation function.

After the basic construction of the network, it is not already skilled at solving the heat equation or any other problems it is fed. In fact, the weights within the model are often entirely random to begin with [13]. For the network to learn, it must be trained. Training a neural network requires significant access to training data; without enough data, the network will likely not perform to its full potential. In the simplest terms, the training process works like this: the model is fed sample data, and makes a prediction for each example it is given. The network is fed labeled data, meaning the correct result is known, so after making its own prediction, it is compared to the expected output. The model's weights are then adjusted to improve the model's accuracy based on the new information it has "learned". More concretely, the training process is about minimizing a loss function. A loss function indicates how well an algorithm performs on a given data set. A high loss value signals poor performance, that is, the predicted results are not close to the expected. By minimizing this function, we can expect better predictions. There are a number of loss functions, and they differ based on whether the algorithm is performing a classification or regression task. Modeling the heat equation is a regression task. In this case, a common loss function,

and the one I used, is Mean Squared Error (MSE),

$$MSE = 1/n \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{5}$$

where n is the number of data points, Y_i is the observed values and $\hat{Y}_i$ is the predicted values.

This loss function is minimized by adjusting and optimizing the weights of the neural network. This is done through a process called back-propagation. While predictions are made by the neural network in a feed-forward fashion, back-propagation is the opposite as it begins in the final layer an propagates backwards. While moving backwards, the algorithm calculates how much the final output, and therefore the error, has been affected by each weight, essentially determining which weights are to blame for the error in the previous prediction. Then, new optimal weights are calculated and the weights are updated to reflect this [1]. The weights are update continuously throughout the training phase as the network alternates between feeding data forward and back-propagation.

After the training phase is complete, the weights of the neural network are solidified and no longer change during the testing phase. The purpose of the testing phase is to evaluate how accurate the model is. This is accomplished by providing new data to the model, so the testing samples must be kept separate from the training data set. Again, the network's predictions will be compared to the expected output. But this is done only with the goal of evaluating the model's accuracy, rather than as part of the process for improving the model itself. Two common metrics that I used to evaluate the accuracy are the same as the common loss functions, MSE and Mean Average Error (MAE). MAE is even simpler than MSE, it is simply the average distance between the predicted and expected values.

Testing how well my neural network estimated the heat equation involved these same steps. The first step involved generating a dataset to use for training and testing. As noted earlier, I used a slightly modified python script originally created by Kody Powell to generate the data. My dataset included 119600 data points in the training set and had 29900 points in the testing set. Each data point included the following features: HeatTempLeft, which was the temperature of the heat applied to the left side of the rod, HeatTempRight, which was the temperature of the heat applied to

the right side of the rod, InitialTemp, which was the initial temperature of the non-boundary points, time, which indicates the time within the simulation that a given point represents, TestPointPosition which can be used to measure the distance of the given test point to the leftmost point of the rod, and ResultingTemp, which is the simulated temperature of the point at that position and time. The resulting temp is what the network will be predicting.

To create the neural network, I used the Python library TensorFlow as well as the Keras library which acts as an interface for the TensorFlow library. The model I used in this portion of the experiment was quite simple, it consisted of three explicit layers: two 64 node layers and a one node output layer; Keras also adds one implicit layer that handles the shape of the input based on the number of features in the dataset. I had the best results using sigmoid as my activation function, but I also tested the network with ReLu and tanh functions. The model used MSE as its loss function. The training time for the neural network was relatively short. On average, it took about 80 seconds to complete an epoch. One epoch means that the entire training set has been passed through the network one time. Generally, the training phase lasts multiple epochs as the model will be underfit if there are too few epochs. I typically had my network complete 20 epochs during the training phase. Testing follows the training phase. The model makes predictions for all 29900 data points in the test set and then the testing metrics are calculated. These metrics are the primary way to evaluate how well the neural network can estimate the 1-dimensional heat equation. The main model, which had the three layers described above, and used the sigmoid function as its activation function performed best. Over the course of five complete runs, the neural network had an average MAE of 0.24494. For context, the actual temperatures varied between 0 and 45 degrees, so for any given point in time and any given point within the wall, the model was an average of only 0.24494 degrees from the expected temperature. The average MSE throughout the five runs was only 0.12984. The other models using different activation functions performed well but had worse results by a noticeable margin. Using ReLu, the model had an MAE of 0.35638 and an MSE of 0.31504. The model using tanh as the activation function had an average MAE of 0.4359 and an average MSE of 0.3450. The results from each activation function's five runs are

shown in figure 5. These results were somewhat surprising to me. When I was doing my initial building and testing of the network, I had used ReLu by default, as it is often considered to perform consistently well, more so than the other two activation functions used. It was not until much later when I began to experiment with the construction of the model that I tried using the sigmoid function for the activation and realized it performed much better on this particular problem.

I also experimented with adding layers to the model, as well as expanding layers by adding more nodes to them, but these changes led to marginal decreases in accuracy and also increased the training and testing times of the model. From these experiments, sigmoid using more neurons per layer performed best, but it was still not as good as the 64-neuron original. With 128 neurons per layer, the neural network averaged an MAE of 0.27106 and an MSE of 0.16332, so it still performed significantly better than the other activation functions (Figure 6).

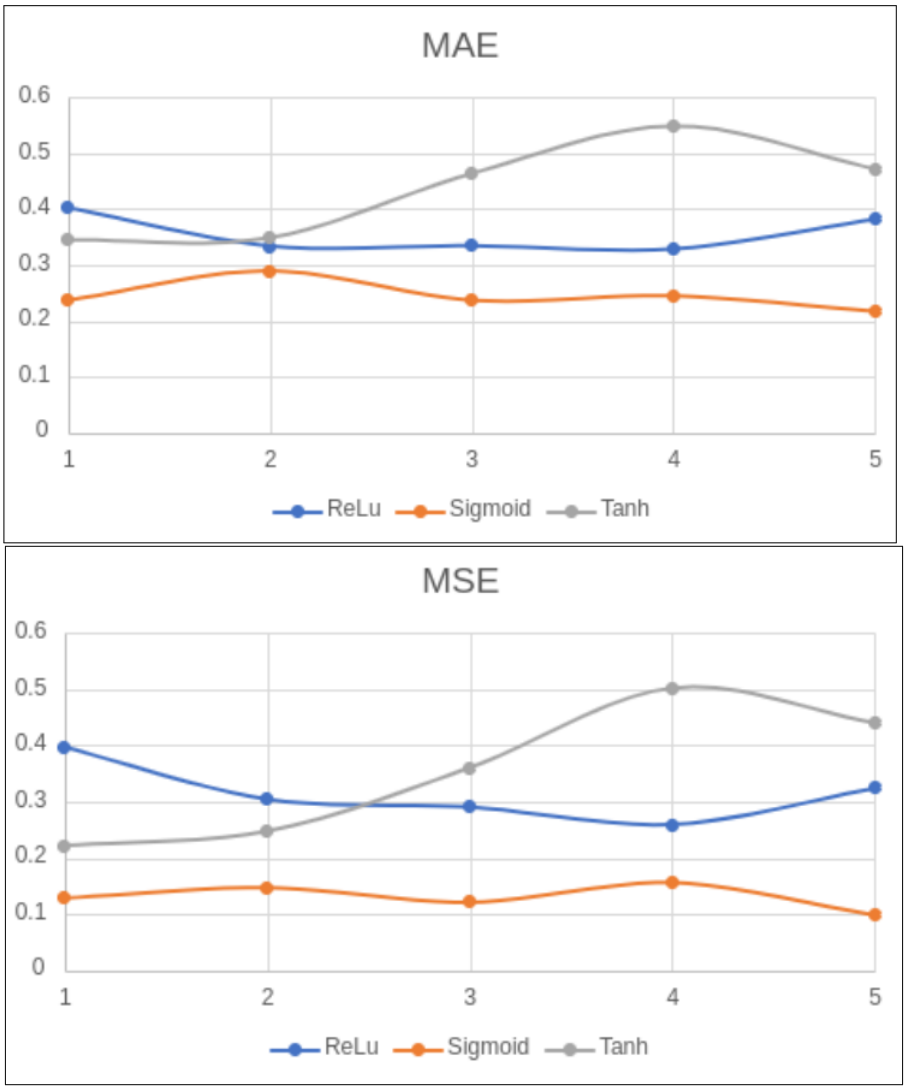


Figure 5: The MAE and MSE of each activation function througout 5 runs.

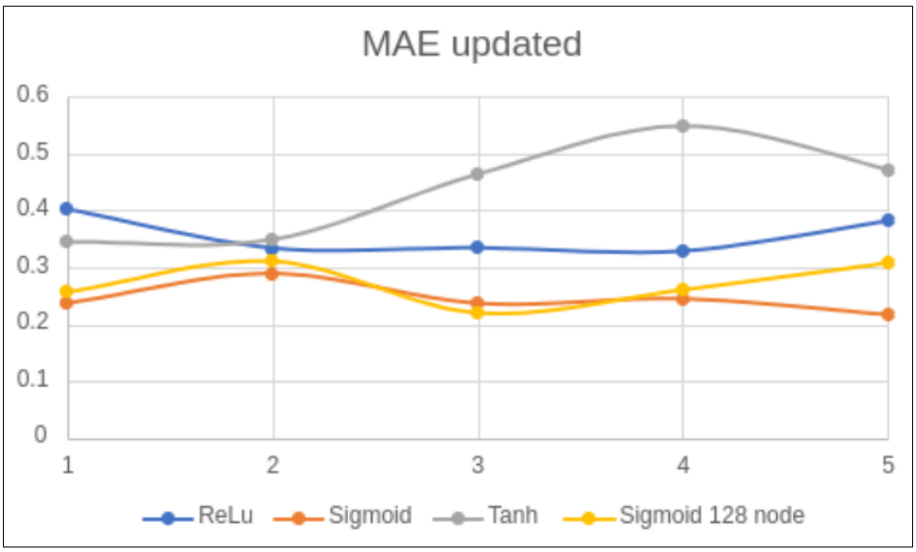
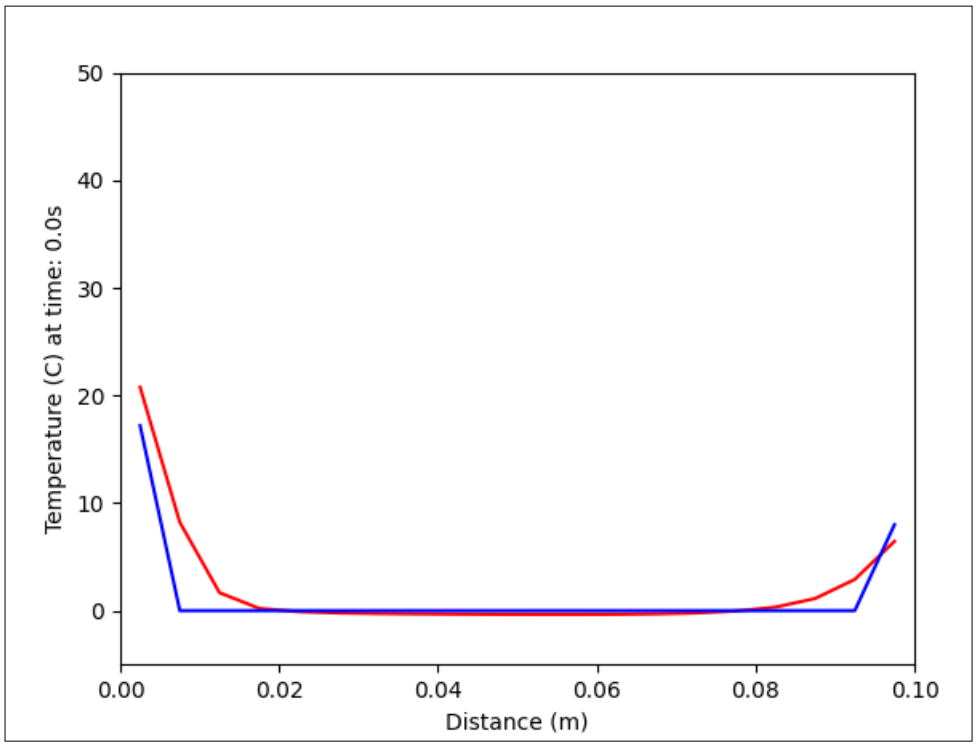
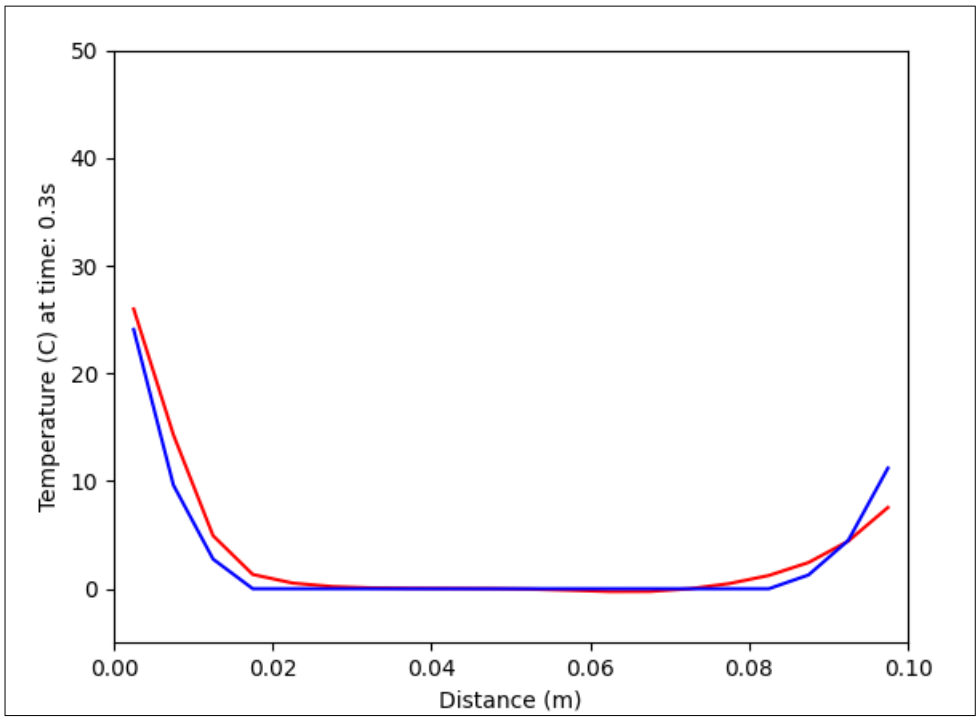


Figure 6: Updated MAE chart to include 128 node Sigmoid model

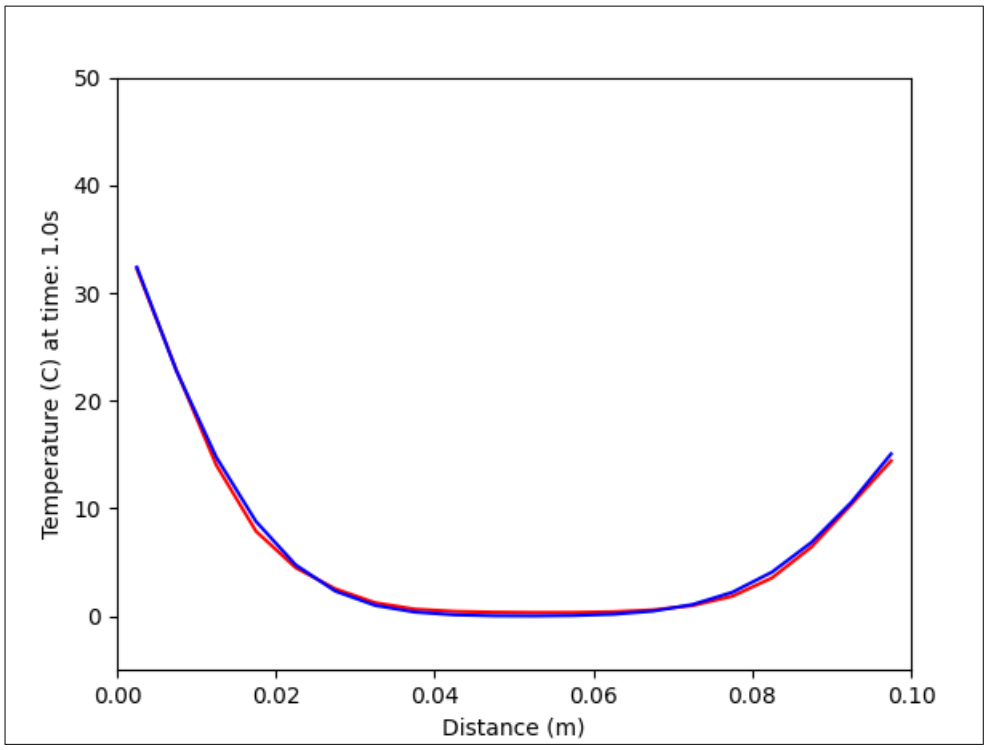
Another method for evaluating these results were graphs and visualizations. I used matplotlib to create a graph that represented the changes in heat over time. Both the expected and predicted values were plotted so it was easy to compare the visual differences between the lists. The visualizations also gave context to the error results, and illustrated where and when the biggest errors occurred. The first tenths of the first second has by far the most error. Here is what the graph looks like at very beginning. Red is used to show the neural network's predictions, and blue is the expected results.



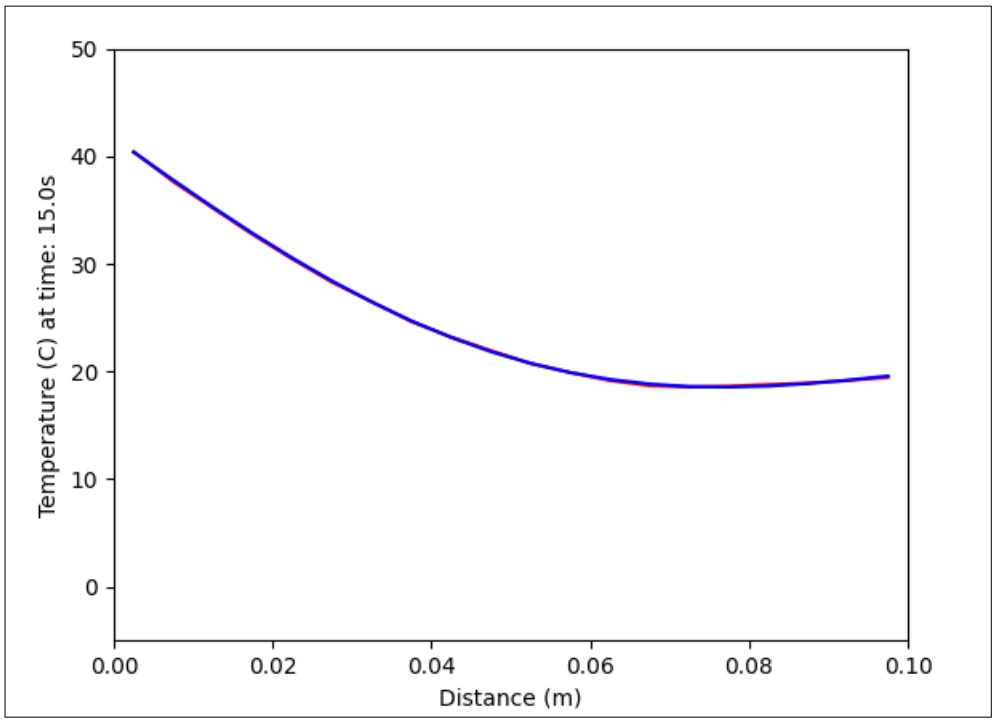
Here, there is some very clear error, especially at either end of the wall. By 0.3 seconds, the results begin to look a little better:



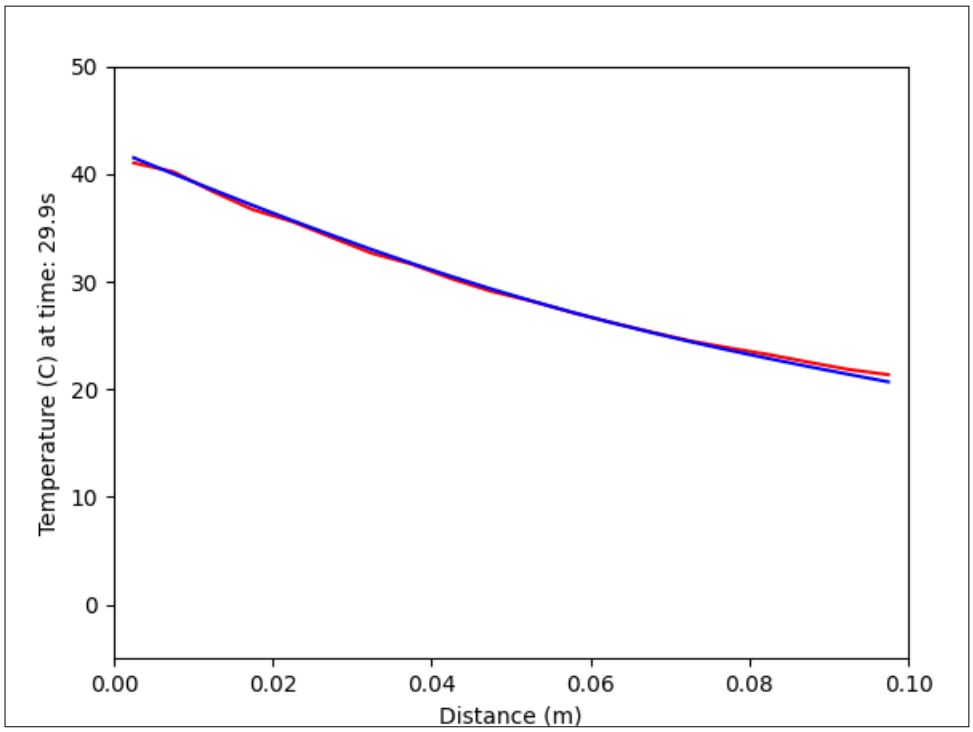
At one second, the lines begin to match up closely:



And from this point forward, the predicted results and the actual results are typically a near match, here it is again at 15 seconds where the lines are almost indistinguishable from one another:



The simulation lasts 30 seconds; this is what the graph looks like at the very end:



Based on these visualizations, it was clear that the first second was the weakest part of the network's predictions. I decided to investigate that further, and found over the course of five more runs (with the sigmoid based neural network) that during the first second, the average MAE was 0.612688. During the rest of the simulation, 29 seconds, the average MAE was only 0.175114. This is a major discrepancy, and something that would definitely need to be addressed if future research took a similar approach.

4 Radial Basis Function Network

The second model used is called a radial basis function network (RBF Network). They are capable of universal function approximation, just like MLP networks. The architecture of an RBF network is also very similar to traditional neural networks; they consist of an input layer, a hidden middle layer, and an output layer [16]. However, an RBF network has only one middle layer, while an MLP network may have multiple. This simplicity means that RBF net-

works typically have faster training speeds compared to other neural networks. Additionally, RBF networks operate differently than MLP networks because they are based on clustering using ellipses and circles rather than an MLP network which is based on linear separation [16]. Within the hidden layer, each node uses a radial basis function (RBF) as a nonlinear activation function [17]. The RBF is denoted as ϕ(r). Each node also has a center due to the clustering nature of RBF networks. Picking the center of each node is typically the first part of the learning phase for a RBF network. Two common ways to select RBF centers are to select them randomly from the training sets, or more commonly, to use k-means clustering to choose RBF centers. I tried both of these approaches. Like traditional neural networks, there are also weights associated with the connections between layers. The second part of the learning phase focuses on optimizing the weights by minimizing the MSE.

There are many different functions that can be used as the RBF, but the most common, and the one I used was the Gaussian function:

$$\phi(r) = e^{-r^2/2\sigma^2} \tag{6}$$

where $r > 0$ is the distance from a data point x to a center c, and σ is the standard deviation that is used to "control the smoothness of the interpolating function" [17].

The implementation of a RBF network is not as straightforward as the neural networks used in the previous section because it is not a default option in the Keras library like the other models were. Fortunately, a faculty member of the Czech Academy of Sciences, Petra Vidnerová, had created a custom RBF keras layer which was available on her GitHub [15]. Using this RBF layer, I was able to test a basic RBF network on the same data used in the previous section. The model I used had 64 nodes in the hidden layer. Due to the high training speed attributed to RBF networks, the network completed 8000 epochs during the training phase. Those 8000 epochs lasted about as long as 150 epochs for the models in the previous section. I tested the RBF network with random centers as well as with centers initialized by k-means. The results were unsurprising; the model with centers initialized by k-means was more consistent than the random centers models, but both were able to come close to the sigmoid network's accuracy during the best runs. The average MAE over the course of five runs for the RBF network with

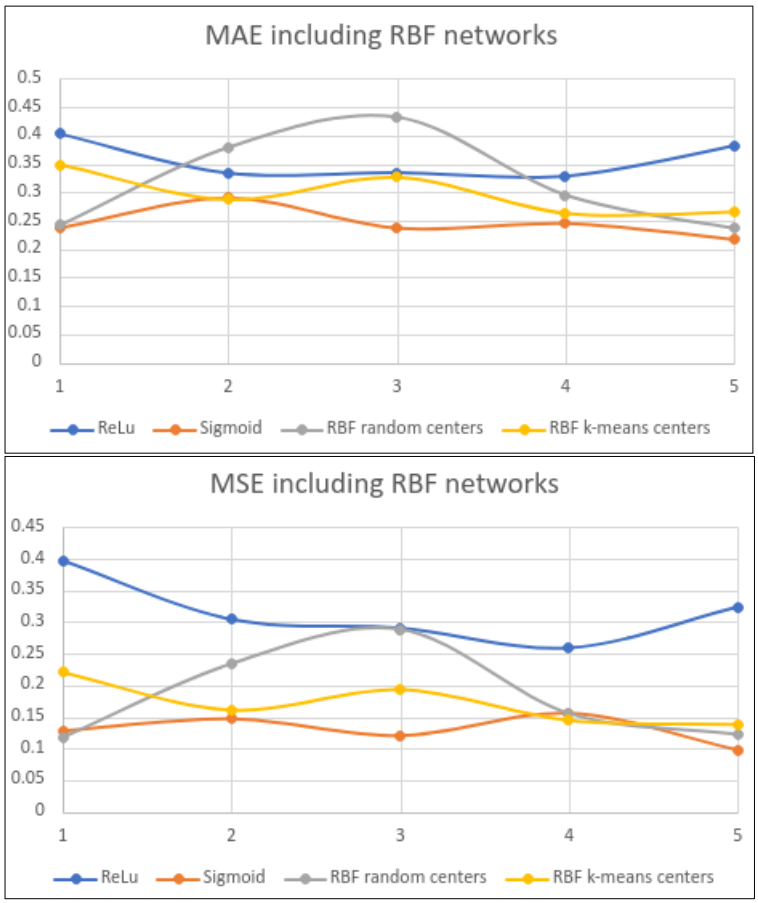


Figure 7: Updated MAE and MSE charts to include both types of versions of the RBF networks.

random initial centers was 0.3171. The average MSE was 0.18364. For the RBF network which used k-means to determine centers, the average MAE was 0.29818 and the average MSE was 0.17128. Both models compare favorably to ReLu and tanh, but they are not quite as accurate as a sigmoid based model (Figure 7).

5 Physics-Informed Neural Networks

The final class of neural networks that I used was a little more abstract. Physics-Informed Neural Networks (PINNs) are described as "neural networks that are trained to solve supervised learning tasks while respecting any given laws of physics" [12]. These models are considered "data-efficient" because they are typically used with problems where data acquisition is especially difficult. To fill in the gaps caused by the lack of data, PINNs encode physical laws as prior information which is then used to make better predictions [12]. This concept is a very recent development, the first paper published on physics informed neural networks or deep learning that I found was published in late 2017 by Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Their research and subsequent research has primarily been focused on approximating nonlinear partial differential equations in high dimensions. The "curse of dimensionality" has caused there to be very few practical high-dimensional alrgorithms which have been developed [5]. This has provided the opportunity for deep learning algorithms to offer new and competitive methods for approximating these PDEs, though it is made clear that the new methodologies should not be considered replacements of classical methods for solving PDEs [12]. Some example PDEs where neural networks have the potential to provide the most benefit are brought up by Jiequn Han, Arnulf Jentzen, and Weinan E. They include: the Schrodinger equation, where the dimensionality is about three times the number of electrons or quantum particles in the system, the nonlinear Black-Scholes equation which is used for pricing financial derivatives where the dimensionality is based on the number of financial assets under consideration, and the Hamilton-Jacobi-Bellman equation, a game theory or resource allocation problem where the dimensionality increases linearly based on the number of actors or resources [5]. All this is to say that while these problems are very interesting, and the research done so far appears very promising, it is not all that similar to my own as there are no dimensionality issues within my problem scope and there are already practical and effective solutions to the heat equation. However, the basic premise of a "physics-informed" neural network was interesting and I expected it to improve my results with a simple addition to account for prior information and physical laws.

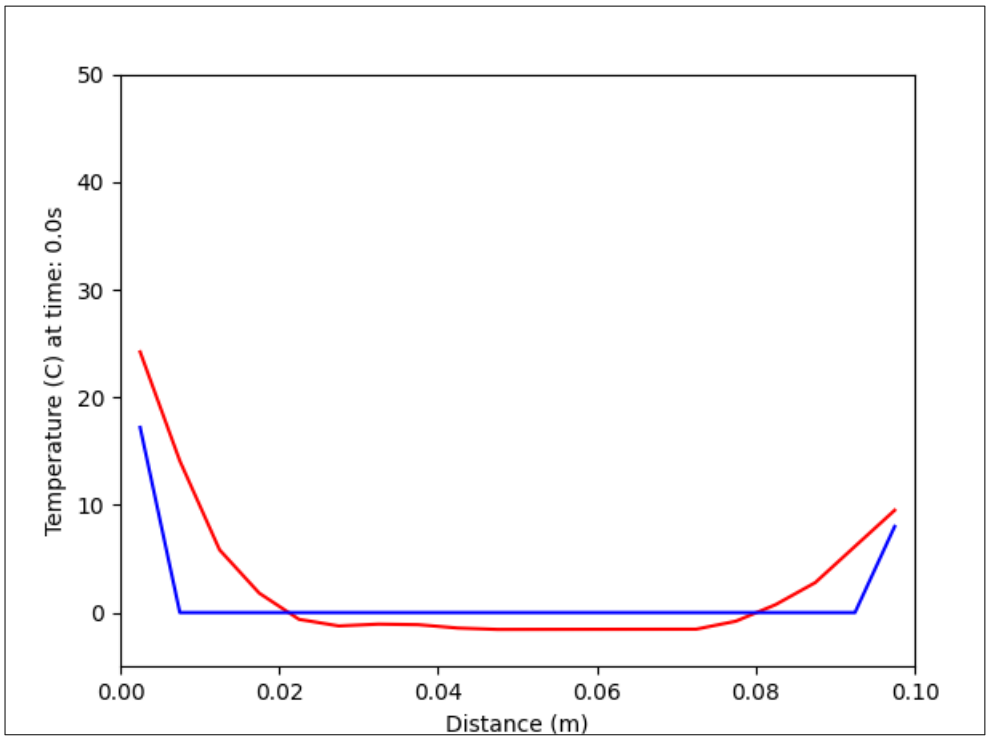


Figure 8: With the ReLu model, many of the initial predictions are for temperatures below zero.

One of my earliest observations while testing the networks performance on estimating the heat equation was the inaccuracy that was most prevalent during the initial seconds of the simulation. The simulated results I used as training and test data had the interior points of the wall begin with initial temperatures of zero degrees while the boundary points, to which the heat was applied, were much higher. This effectively created a range of physically possible temperatures throughout the simulation that was between zero and the highest applied temperature. However, the neural network is not informed of this implicit range and would make predictions which fell outside of the possible values. Over time as time as the testing error was reduced through further improvements, this flaw became less noticeable, especially when using sigmoid as the activation function. When using ReLu, it is still clearly visually apparent (Figure 8). When using sigmoid, it is sometimes visible, but usually to a much smaller degree, shown in figure 9.

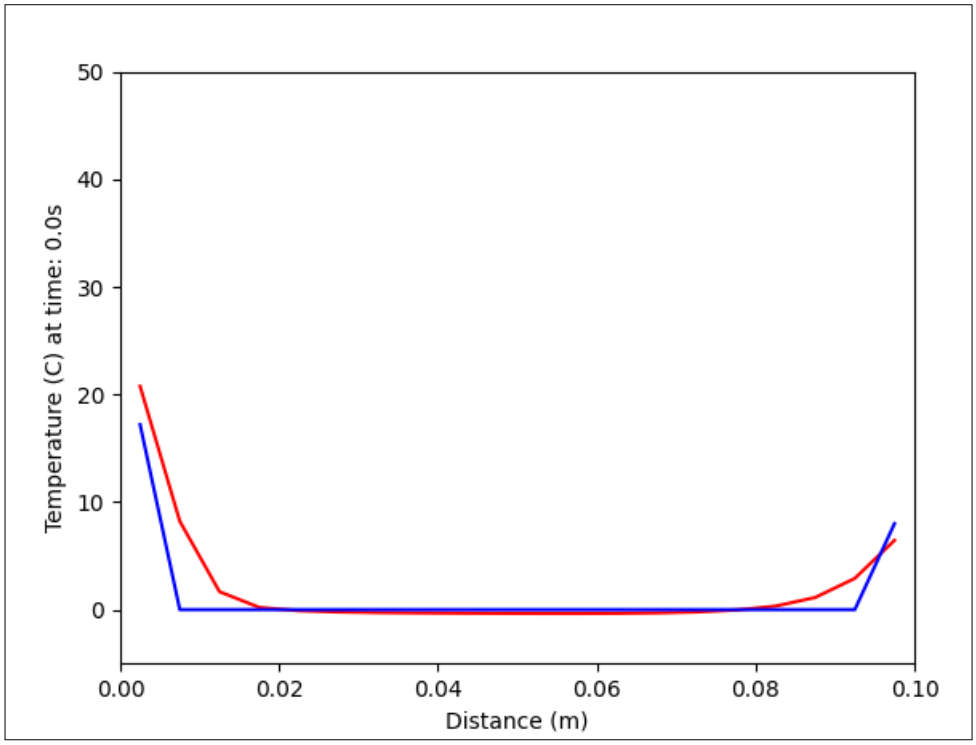


Figure 9: The sigmoid model sometimes made slightly negative initial predictions, but to a much smaller degree.

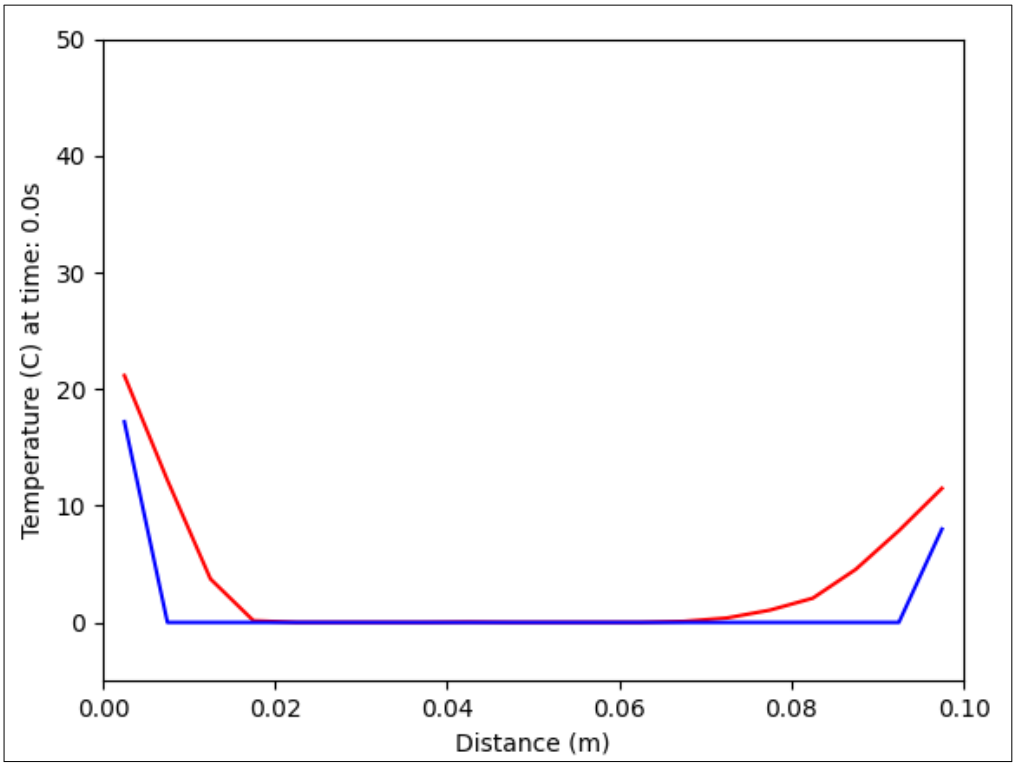


Figure 10: The ReLu model's predictions again, this time with a range enforced.

To combat this error, which is especially jarring (particularly for the ReLu network) because it is immediately noticeable as the predictions begin, I limited the range of guesses the neural network could make to the physically possible range. This eliminated the negative guesses and improves the initial visual of the networks predictions, if only by a small amount. Here is the graph for the ReLu neural network again, this time with a range enforced (Figure 10).

Clearly, there are still major inaccuracies within the first fractions of the first second, especially for the non-sigmoidal networks, as the vast majority of the error still comes from predictions greater than zero. Still, the negatives were the lowest hanging fruit, and still provided some small improvement. For the model which used ReLu as the activation function, the average MAE over five runs went from 0.331084 using the original model, to 0.315684 when enforcing the range. The predictions were made using the same training and test set for both the standard and range-enforced results in a single

run. While this improvement appears to be quite small, it is nearly a five percent improvement in accuracy. These changes had less of an impact on the sigmoid based network because this network did not tend to predict negative values nearly as often or as extreme. Still, there was a slight improvement. Using the predictions from the same test set, the original network had an average MAE of 0.230186 over five runs and the updated version had an MAE of 0.224988. This is only an improvement of just over two percent.

These are certainly not drastic improvements, but they can only improve the results of the predictions, so there is really no downside to implementing them. This is probably the simplest possible "physics informed" idea that can be implemented, so it does not show even a small amount of the potential improvement that physics informed nerual networks can offer. This is only a small example of how a basic concept can be useful and provide a small, but guaranteed improvement.

6 Going Beyond the First Dimension

The results discussed up to this point have only dealt with the one dimensional heat equation, but it is important to confirm that the performance will hold up beyond the simplest problem. Fortunately, I was again able to find an online resource to generate training data for the two dimensional diffusion equation [2]. This is done in a two dimensional plane where one rectangular area has heat applied (Figure 11). Then over the course of the simulation, the heat diffuses over the plane (Figure 12). It is important to note that in this simulation, unlike the one dimensional data, the area that is initially heated is not held to that temperature throughout the simulation. Instead, it is initially heated and then diffuses over time.

Neural networks using the sigmoid function proved again to provide the best approximation for this problem. The training set generated consisted of 323200 samples and took about 50 minutes to complete 15 epochs of training. The test set had 80800 samples. For the 2-dimensional problem, the network predicted the temperature of a point given by its x and y coordinates, the time, the initial temperature of the applied heat, along with some other constants. Like the 1-dimensional experiment, the range of expected temperatures ranged between 0 and 45 degrees so the scale of the loss

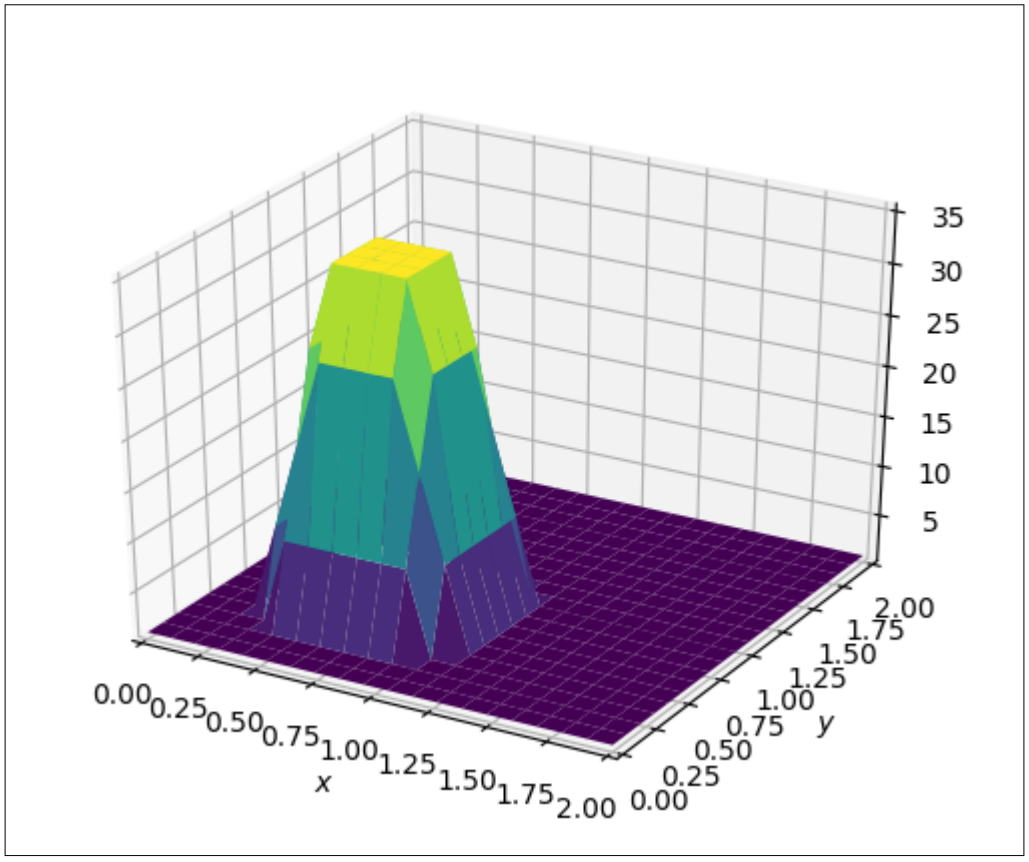


Figure 11: The initial state of the 2D simulation.

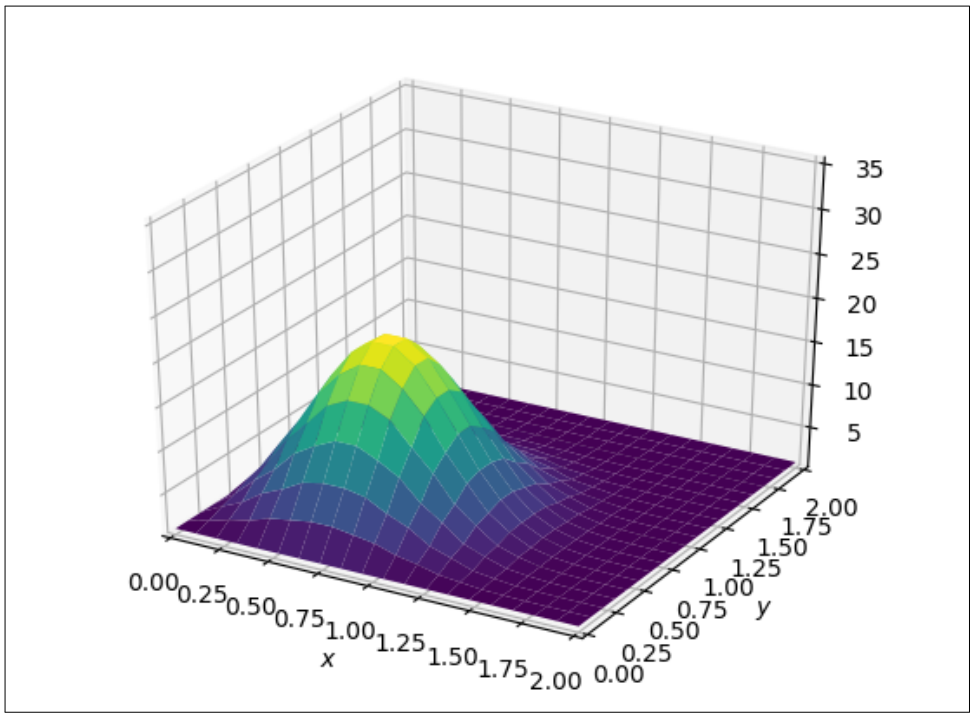


Figure 12: The state of the 2D simulation after beginning the diffusion.

functions should not be significantly different. The model used with the 2-dimensional data was very similar to the network used with the original 1-dimensional set. It was the same three layer model with a 64 node input layer, a 64 node middle/hidden layer, and a single node in the output layer. The main difference is that the data inputted to the second model was shaped differently, containing a few additional features.

The neural network performed well with the new data set. The test set predictions had a MAE of 0.26268 and a MSE of 0.26008. While this is slightly worse than the same metrics for the top performing model on the 1-dimensional data set, the difference is small. Assuming this trend continues, a neural network should continue to be capable of estimating the heat equation at higher dimensions with small margin of error.

7 Discussion of Results and Future Work

The results obtained show that neural networks can estimate the heat equation with a reasonably small error. However, when looking specifically at the heat equation, it would be a stretch to conclude there is practical application for *these* networks. There already exists a number of ways to solve the heat equation, both quickly and accurately. Given this, there is not much need for a neural network which can approximate the PDE more slowly and with less accuracy. To put this more concretely, the basic feed-forward neural network took 1.1794 seconds to predict a full 30 second 1-dimensional simulation. For the simulation script that was used to generate data, it took only 0.00876 seconds. Classic methods for solving the heat equation are certainly not too slow in the lower dimensions. However, this is not to say speedup is not possible. In a recent Stanford study, the authors solved the 2D Poisson equation using a deep neural network and had 2-3 times speedup in comparison to standard iterative solvers [7]. This model is built on top of existing linear solvers so one limitation to this approach is that it cannot be used solve some of the nonlinear problems stated earlier. They do believe the method can be extended to PDEs beyond the Poisson equation in future work.

My results are more a demonstration of how proficient machine learning can be on a challenging problem. Even with very simple

networks, the predictions have a relatively small margin of error, especially after the first few time steps. As my first experience with neural networks, I did not stray far from the keras library when creating the models; these are certainly not hand-crafted, high performance neural networks. I think it would be interesting to see just how far the error could be taken down by someone with more experience using neural networks.

Additionally, this research did not venture into the deep learning sector of machine learning as I was immediately met with reduced accuracy after increasing the number of layers within the network. I would be curious if deep learning could be applied in a more sophisticated way to this problem and have serious performance boosts. Again, applying these more advanced techniques to the heat equation and similarly simple PDEs may not have as many practical uses, but they are still interesting problems to solve.

Where there is real potential for practical use is the problems that are currently challenging to due dimensionality and non-linearity concerns. Recent studies have already shown impressive results for estimating the Burgers' equation and the nonlinear Schrodinger Equation, both problems which have proved to be difficult for classical methods to solve [12]. However, as seen in my own results, there still exists the error and uncertainty associated with the predictions of neural networks. Reducing or even eliminating this problem would make neural networks a far more practical choice for solving PDEs, and I expect this will continue to be studied in the near future.

8 Contribution

The research detailed in this paper, while not presenting groundbreaking results, is still a worthwhile contribution. For other undergraduate students who may be interested in working on the same, or a similar problem, this paper provides a good starting point. Within the associated repository I have consolidated a number of useful materials, including the scripts I used to generate training, and simple neural networks that approximate the solution to the heat equation with generally good results. Future students interested in this research can expand on this research and focus on producing better models without needing to spend as much time setting up the

problem. Additionally, for more experienced researchers, the results presented within this paper can be seen as a baseline for how well simple neural networks can predict the solution for the heat equation. It should be expected that the models they are building will perform better than mine.

9 Conclusion

Neural networks can be a powerful tool when used for the right problems. Even further, these results show that neural networks still perform well on problems where they are not entirely necessary. Even though it is unlikely that anyone will be using neural networks to estimate the 1 or 2-dimensional heat equations for any practical purposes, the fact that the predictions are good enough to the point of being almost indistinguishable to the expected results visually (after the first second) only reaffirms the function approximation capabilities of neural networks.

Within my own results it is clear that the standard neural network using the sigmoid function as an activation function had the best results. The RBF network also performed competitively, but was not quite as consistent or as accurate as the best network. Adding physics informed constraints to the model also was shown to be effective in reducing the error, though only in a small way.

Given the amount of work done in the past five years on topics like this, I expect much progress to be made in the near future. While my research and results are certainly not cutting edge, it was an interesting way to begin looking at and working with a challenging problem.

10 References

[1] Backpropagation in neural networks: Process, example code.

[2] Lorena A. Barbara and Gilbert F. Forsythe. 12 steps to navier–stokes. 2017.

[3] Jaron Collis. Glossary of deep learning: Bias. Apr 2017.

[4] Satya Ganesh. What's the role of weights and bias in a neural network? Jul 2020.

[5] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[6] Larry Hardesty. Explained: Neural networks. *MIT News*, Apr 2017.

[7] Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning neural PDE solvers with convergence guarantees, 2019.

[8] Isaac Lagaris, Aristidis Likas, and Dimitrios Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *Neural Networks, IEEE Transactions on*, pages 987 – 1000, 10 1998.

[9] A. D. Polyanin, W. E. Schiesser, and A. I. Zhurov. Partial differential equation. *Scholarpedia*, 3(10):4605, 2008. revision #121514.

[10] Kody Powell. Derivation of the heat diffusion equation (1d) using finite volume method, August 2017.

[11] Kody Powell. Solving the heat diffusion equation (1d pde) in python, August 2017.

[12] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[13] Daniel Svozil, Vladimír Kvasnicka, and Jirí Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43–62, 1997.

[14] Avinash Sharma V. Understanding activation functions in neural networks. Mar 2017.

[15] Petra Vidnerová. Rbf-keras: an rbf layer for keras library available at https://github.com/petravidnerova/rbfkeras, 2017.

[16] Tiantian Xie, Hao Yu, and Bogdan Wilamowski. Comparison between traditional neural networks and radial basis function networks. *Proceedings - ISIE 2011: 2011 IEEE International Symposium on Industrial Electronics*, pages 1194 – 1199, 07 2011.

[17] Biaobiao Zhang Yue We, Hui Wang and K.-L. Du. Using radial basis function networks for function approximation and classification. *ISRN Applied Mathematics*, 2011.

11 Source Code

Below is the Keras/TensorFlow implementation of the basic neural network used to make predictions as well as the visualizations. Any additional code, including the scripts to generate training and testing data and the custom RBF layer can be found at https://github.com/Collegeville/Jordre-Nathan-Work or as they are cited.

```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics



print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
datafile = open("heatData.csv", 'r')

df = pd.read_csv(datafile, index_col=None)

target = df.pop('ResultingTemp')

full_dataset = tf.data.Dataset.from_tensor_slices((df.values, target.values))
unshuffled = full_dataset.batch(1)
# shuffle data set to ensure randomness of training and test sets
```

```
full_dataset = full_dataset.shuffle(len(df)).batch(1)
# take 20% of dataset to use as testing data
test_dataset = full_dataset.take(tf.data.experimental.cardinality(full_dataset).numpy() * .2)
# skip the 20% of data used for testing and take the other 80% to use for training
train_dataset = full_dataset.skip(tf.data.experimental.cardinality(full_dataset).numpy() * .2)

# can use this option if creating a visualization
# must manually create the training and test sets in this case
# train_dataset = full_dataset


def get_compiled_model():
    tf.keras.backend.set_floatx('float64')
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation="sigmoid"),
        tf.keras.layers.Dense(64, activation="sigmoid"),
        tf.keras.layers.Dense(1)
    ])

    model.compile(optimizer=tf.keras.optimizers.RMSprop(0.001),
                  loss='mse',
                  metrics=['mae', 'mse'])
    return model


model = get_compiled_model()
#can use following line to switch between cpu and gpu
with tf.device('cpu:0'):
    model.fit(train_dataset, batch_size=128, epochs=20)

loss, mae, mse = model.evaluate(test_dataset, verbose=2)


# To properly create visuals without mixing data sets,
# you must manually create the training and testing data sets
datafile = open("smallHeatData.csv", 'r')

df = pd.read_csv(datafile, index_col=None)
```

```
target = df.pop('ResultingTemp')

full_dataset = tf.data.Dataset.from_tensor_slices((df.values, target.values))
unshuffled = full_dataset.batch(1)

predictions = model.predict(unshuffled)

x = np.linspace((.1/20)/2, .1-(.1/20)/2, 20)

T = []
N = []
i = 0
time = 0.0
while i < len(predictions):
    T = []
    N = []
    plt.ion()
    plt.clf()
    targ = i + 20
    while i < targ:
        T.append(predictions[i])
        N.append(target[i])
        i += 1
    plt.figure(1)
    plt.plot(x, T, 'r')
    plt.plot(x, N, 'b')
    plt.axis([0, .1, -5, 50])
    plt.xlabel('Distance (m)')
    plt.ylabel('Temperature (C) at time: ' + str(round(time + 0.1, 1)) + 's')
    plt.show()
    plt.pause(0.05)
    time += 0.1
```