Dylan Smith
PHY480 Final Project

Solving Partial Differential Equations Numerically Using Neural Networks

I. Introduction

The Solutions of partial differential equations (PDEs) are highly sought after to determine the dynamics of complex systems [1]. However, often times, these equations do not have known or possible analytic solutions, so numerical methods such as finite elements or finite differences must be used. However, these methods only solve PDEs weakly over a specified domain, and the solutions are discrete and usually have limited differentiability [1].

Neural networks are a class of algorithm within the field of machine learning that are composed of nonlinear, repeated transformations to input data in an attempt to find a mapping between the input data and a desired label. Neural networks can also act as universal function approximators; according to the Universal Approximation Theorem, a single hidden layer neural network with an arbitrary number of hidden nodes can approximate any continuous function over some bounded domain under $\mathbb{R}^n$ [1].

Compared to numerical solutions using finite element analysis or Runge-Kutta integration, solving differential equations with neural networks has a large number of benefits. Many routes of numerically solving PDEs yield discrete solutions or solutions of limited differentiability, while neural network solutions are closed and analytic. Additionally, the number of hyperparameters to tune is far less than other contemporary solution methods, and models can be obtained with very low constraints on system memory [1]. Finally, the neural network architecture is very easy to augment to parallel architectures.

II. Methods

The basic method to solve PDEs with shallow NNs is to assume the solution of the PDE being studied has a trial solution that both satisfies the boundary conditions (BCs) and/or initial conditions (ICs) and has the output of the NN as an input. This trial solution has the form

$$u_t(\vec{x}) = A(\vec{x}) + B(\vec{x}) * N(W, \vec{x}),$$

with W being the neural network weights that get iteratively updated during training, and the functions $A(\vec{x})$ and $B(\vec{x})$ being functions that guarantee that the BCs and ICs are satisfied. The cost function that is used to train the network is the mean-squared error of the differential equation, or

$$C(W, \vec{x}\,) = f(\vec{x}, D\vec{x}, \dots, D^{(n)}\vec{x})^2,$$

where $f(\vec{x}, D\vec{x}, \dots, D^{(n)}\vec{x})$ is the differential equation being studied.

The wave equation will be solved using a single hidden layer neural network, where the error will be assessed as the difference between the known, analytic solution and the numerical solution determined by the neural network. The one-dimensional wave equation is given by

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \frac{\partial^2 u(x,t)}{\partial x^2}$$

where $u(x,t)$ is the scalar function to be solved for, and c is the wave speed. The ICs and BCs are given by

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

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

$$\frac{\partial u(x,t)}{\partial t} = -\pi \sin(\pi x) \text{ at } t = 0,$$

$$c = 1.$$

A potential trial solution is given by

$$u_t(x,t) = (1-t^2)\sin(\pi x) - \pi t * \sin(\pi x) + x(1-x)t^2 * N(x,t,W),$$

and the analytic solution to be compared with is

$$u_a(x,t) = \sin(\pi x)\cos(\pi t) - \sin(\pi x)\sin(\pi t).$$

The domain to be integrated over is $D = [0,1] \times [0,1]$, with a mesh spacing of 0.1. The optimizer being used is called Adam, which adaptively changes parameters in the network to increase efficiency and accuracy. The learning rate, a hyperparameter that controls how big a step the optimizer takes during each iteration, is set to 0.01. The number of neurons in the hidden layer will be varied to see how it affects the error, as well as the number of iteration steps.

The network architectures used will be programmed in TensorFlow, a powerful, end-to-end machine learning platform. Initially, Autograd, an automatic fast differentiation library for Python, was used. However, solving the same equation in TensorFlow was much faster compared to Autograd; Autograd took up 15-20 minutes to compute the solution over 10,000 iterations, while TensorFlow only took seconds. TensorFlow also has an easy-to-use, modular API that allows for quick changes to neural network architecture.

III. Results

Shown below is a plot of the loss versus the iteration step. This was for a single hidden layer network with 90 hidden neurons.

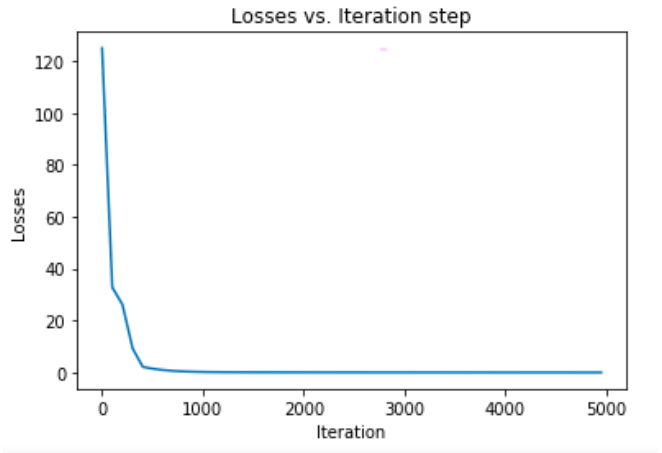

Fig. 1: Loss function value at several iteration steps for an NN with 90 hidden neurons.

Shown below are plots of the analytic and NN solutions to the wave equation problem described above for the same network. The solution was calculated after 10,000 iterations, which took 35.9 seconds.

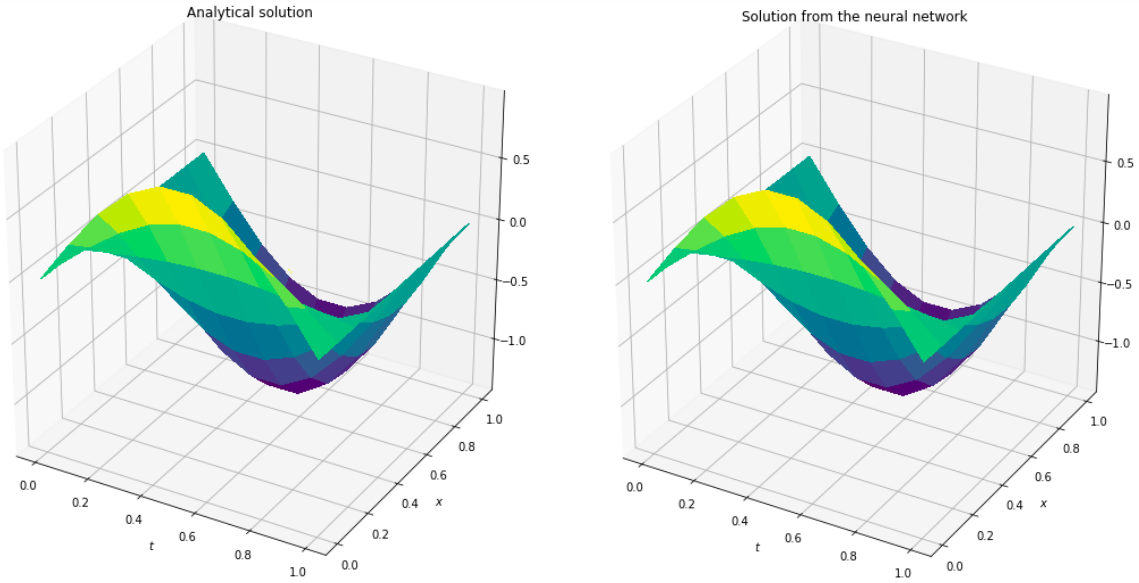

Fig. 2: (a) Analytic solution to the wave equation within given BCs (left). (b) Solution given by the neural network (right)

Shown below are cross sections to each solution shown at $t = 0$ and $t \approx 0.5$.

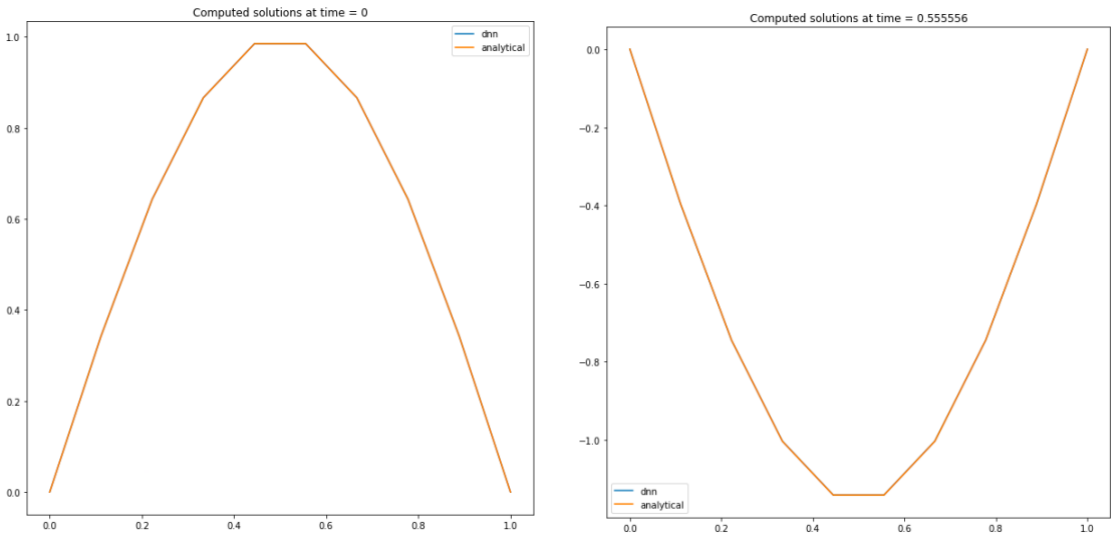

Fig. 3: (a) computed solutions at $t = 0$ *and* $t = 0.555556$

Shown below in Fig. 3 is the absolute difference over the domain between the analytic and numerical solutions. The max absolute difference was calculated to be $7.86 * 10^{-4}$.

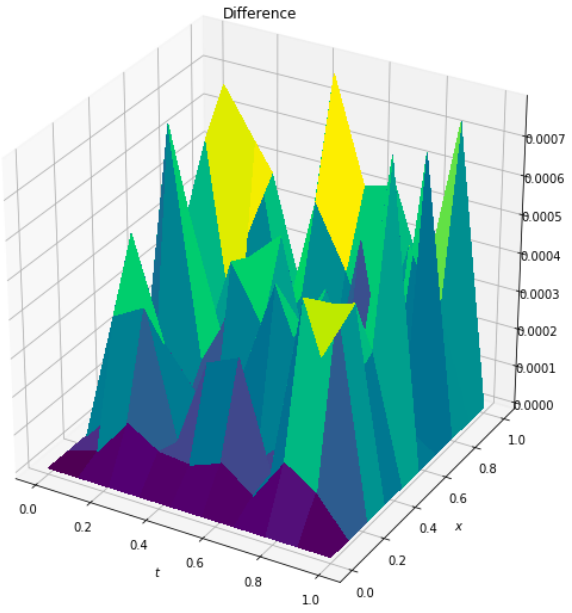

Fig. 4: Absolute difference between numerical and analytic solutions

Finally, shown below are the errors vs. the number of iteration steps and the number of hidden neurons, respectively. The first graph was generated using an NN with a constant hidden neuron size of 90 neurons, and the second graph was generated over 2,000 iteration steps.

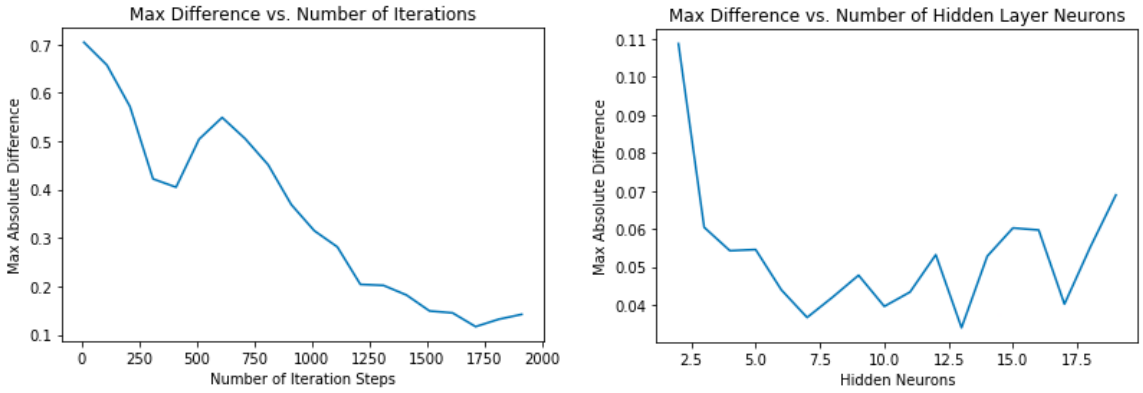

Fig. 5: (a) Maximum difference vs. number of iteration steps. (b) Max difference vs. number of hidden layer neurons

IV. Discussion

As shown by Fig. 1, the loss generally starts very high, as the network weights are randomly initialized. However, the network quickly learns during the optimization, and approaches single digits after approximately 500 iterations. For this same network of 90 hidden neurons, the Adam optimizer worked the best; the same network trained instead using gradient descent took 34.3 seconds, but produced an absolute max difference of 0.028, nearly 40 times larger than the max difference using Adam.

As shown in Fig. 3 and Fig. 4, the NN solution is very close to the analytic solution, showing close to no deviation in the cross-sectional solutions to the unaided eye. The error is fairly uniform over the domain, with the lowest error being at the boundary, which makes sense given that this was set before training began.

It seems that increasing the number of iteration steps taken during optimization tends to decrease the max difference until a point where it stagnates just above 0.1. Training for any longer than 5,000 seems to have marginal returns in lowering the error, while taking significantly longer to train; for a network trained over 50,000 iteration steps, the error dropped to $2.59 * 10^{-4}$, only roughly 0.0005 less than with 10,000 iterations, and it took 123 seconds to compute – nearly four times as long. The number of hidden layer neurons seems to minimize the difference at 13 neurons; increasing the number past this point seems to gradually increase the loss. Previous work [2] suggests that the number of neurons should increase as the mesh spacing decreases, but for the fairly large spacing of 0.1 used here, increasing the number of neurons too much produces marginal improvements to the error, if not increasing the error altogether.

V. Conclusions

In conclusion, neural networks provide a powerful method of solving PDEs over a given domain. With just one hidden layer, neural networks can produce accurate solutions within a very short amount of time. Additionally, these neural networks are modular in nature, and so

they can be easily augmented to experiment with different architectures and differential equations.

In this work, the only domain considered was the rectangular region $[0,1] \times [0,1]$. Future studies could look at nonuniform domains, perhaps with irregular boundaries.

VI. Citations

[1] Lagaris, A. Likas and D. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations", *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987-1000, 1998. Available: 10.1109/72.712178

[2] Chiaramonte, M., and M. Kiener. Solving differential equations using neural networks. http://cs229.stanford.edu/proj2013/ChiaramonteKiener-SolvingDifferentialEquationsUsingNeuralNetworks.pdf