# Finding Approximate Analytic Solutions to Differential Equations using Neural Networks

**Abiyaz Chowdhury**
Department of Electrical Engineering
New York-The Cooper Union
New York, NY 10003

**Kevin Yao**
Department of Electrical Engineering
New York-The Cooper Union
New York, NY 10003

## Abstract

In this project, we will look at a neural network approach to analytically solve ordinary and partial differential equations. We will show that this addresses problems related to common methodologies. Primarily, we are able to discover an algebraic expression comprised of the input variables for our solution. Our goal is to demonstrate the current effectiveness of the neural network model and several ways to improve it.

## 1 Motivation

Analytically solving ordinary and partial differential equations has had fulfilling uses in numerous fields of science. Many techniques have been developed and are continuously used to find these solutions, but they are generally inefficient due to the discrete nature of their approach. These approaches, including finite elements and Runge-Kutta, are generally efficient and accurate, but require repeated computation for every new calculation. This poses a problem for large scale simulations in engineering such as computational fluid dynamics, as the memory space and time become pressing constraints.

## 2 Methodology

In this section, we will briefly demonstrate the original working neural network model used to solve differential equations. This study focuses primarily on second order differential equations of this form:

$$G(\vec{x}, \Psi(\vec{x}), \nabla\Psi(\vec{x}), \nabla^2\Psi(\vec{x})) = 0, \vec{x} \in D \tag{1}$$

subject to certain boundary conditions where $\vec{x} = (x_1, ..., x_n) \in R^n, D \subset R^n$. $D$ denotes the definition domain, and $\Psi(\vec{x})$ is the solution developed. In our implementation, we will use the collocation method to discretize the domain $D$ and boundary $S$ to $\hat{D}$ and $\hat{S}$. Our new domain becomes $\hat{D} = \vec{x_i} \in D, i \in (1, ..., m)$, and our equation becomes:

$$G(\vec{x_i}, \Psi(\vec{x_i}), \nabla\Psi(\vec{x_i}), \nabla^2\Psi(\vec{x_i})) = 0, \forall i \in (1, ..., m) \tag{2}$$

To approximately solve for $\Psi(\vec{x})$, we use a trial solution of the form:

$$\hat{\Psi}(\vec{x_i}) = A(\vec{x}) + F(\vec{x})N(\vec{x}, \vec{p}) \tag{3}$$

where $N(\vec{x}, \vec{p})$ is a feedforward neural network with parameters $\vec{p}$. Note that $A(\vec{x})$ contains no adjustable parameters. One of the reasons this model is so effective is because of the boundary conditions being "hard-coded" into the model. The term $A(\vec{x})$ and the function $F(\vec{x})$ are chosen beforehand such that boundary conditions are satisfied and will not be affected by the output of the neural network. Note that $A(\vec{x})$ contains no adjustable parameters. This approach reduces the problem from a constrained optimization problem to an unconstrained optimization problem, which is simply easier to handle. We seek to minimize the following cost function with respect to model parameters $\vec{p}$:

$$J(\vec{p}) = \sum_{\vec{x_i} \in \hat{D}} G(\vec{x_i}, \hat{\Psi}(\vec{x_i}), \nabla\Psi(\vec{x_i}), \nabla^2\Psi(\vec{x_i}))^2 \tag{4}$$

subject to boundary conditions.

After optimizing parameters $\vec{p}$, the solution $\hat{\Psi}(\vec{x})$ will be a smooth approximation of the true solution and can be evaluated continuously in the domain. We test our results by comparing our optimized trial solution to the actual solution.

### 2.1 Network Model

The basic neural network architecture can be represented as the following:

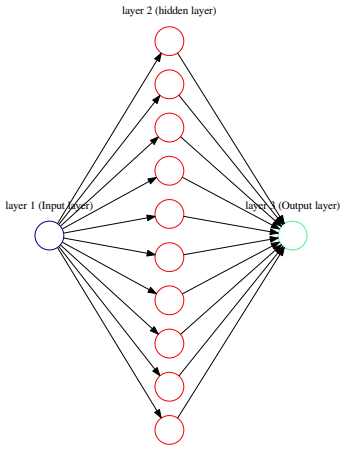


Figure 1: Schematic of Basic Neural Network Architecture

This is a simple three layer multilayer perceptron (MLP) with one hidden layer that takes 1 input, has 10 nodes in the hidden layer, and returns one output.

## 3 Results

We implement the basic three layer MLP applied to both an ordinary and partial differential equation. Note that this model is trained without batch and layer normalization and a single 10 unit hidden layer with sigmoid activation functions.

### 3.1 Example - Ordinary Differential Equation

Our first example will be an attempt to solve the ordinary differential equation:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = e^{-x/5}\cos x - \frac{y}{5} \tag{5}$$

where $y(0) = 0$. To solve this equation, we assume a trial solution of the form:

$$\hat{y} = xN(x, \vec{p}) \tag{6}$$

where $N(x, \vec{p})$ is the output of the neural network. Our optimization problem then becomes the following:

$$\min \sum_{\vec{x_i} \in \hat{D}} \left[ \frac{\mathrm{d}\hat{y}}{\mathrm{d}x} - (e^{-x/5} \cos x - \frac{1}{5y}) \right]^2 \tag{7}$$

Note that the derivative of the trial solution involves the derivative of the neural network model with respect to input values $x$. This implementation utilizes automatic differentiation to quickly and accurately calculate derivatives of the neural network model.

To measure the accuracy of the solution, we compute the absolute distance from the actual solution:

$$y = e^{-x/5} \sin x \tag{8}$$

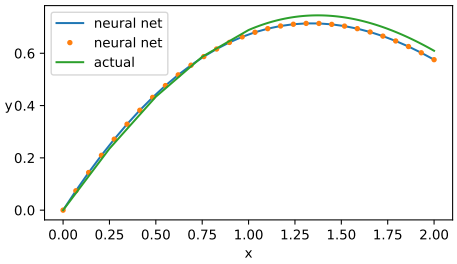


Figure 2: Comparison of approximation and actual solutions

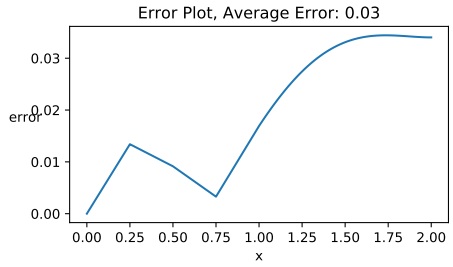


Figure 3: Absolute Error of Trial Solution

### 3.2 Example - Partial Differential Equation

We will look at the Laplace's equation in two dimensions:

$$\frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} = 0 \tag{9}$$

with Dirichlet boundary conditions:

$$\Psi(0, y) = 0, \quad \Psi(1, y) = 0, \quad \Psi(x, 0) = 0, \quad \text{and} \quad \Psi(x, 1) = \sin \pi x \tag{10}$$

Just as for ordinary differential equations, the assumed trial solution is formed such that boundary conditions are inherently accounted for. In this case, the trial solution is the following:

$$\hat{\Psi}(x, y) = y \sin \pi x + xy(x-1)(y-1)N(x, y, \hat{p}) \tag{11}$$

Second order derivatives are taken again using automatic differentiation to calculate the loss function. For checking the accuracy of the approximated solution after optimization, the actual solution used is

$$\Psi(x, y) = \frac{\sin \pi x \sinh \pi y}{\sinh \pi} \tag{12}$$

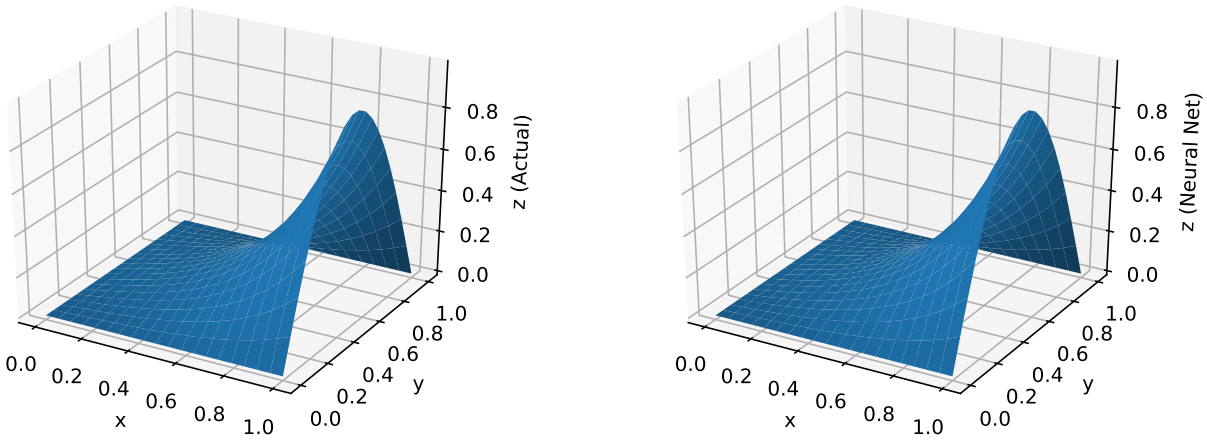


Figure 4: Visual aid for comparing approximation (right) and actual (left) solutions

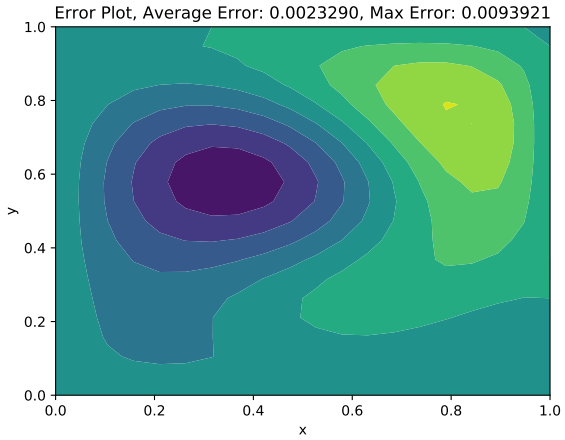


Figure 5: Absolute Error of Trial Solution

## 4 Improvements

Since the publication of the original paper, computational power has improved drastically. As a result, it is no longer as expensive to increase the number of model hyperparameters. Intuitively, increasing the number of layers, as well as nodes per layer, is the first thing to look at for potential improvements. Furthermore, many new techniques have been developed to improve model fitting for neural networks. In particular, we will look at different activation functions, L2 regularization, batch normalization, and layer normalization.

### 4.1 Increasing Nodes and Layers

The current neural network model has just one hidden layer. Upon trying multiple combinations of layers and nodes according to neural network standard procedure, it was found that multilayer perceptrons with 2 hidden layers performed better than others.

| Average Absolute Error | | | |
|---|---|---|---|
| Number of Nodes | 1 Hidden Layer | 2 Hidden Layers | 3 Hidden Layers |
| 5 | 0.0024 | 0.0012 | 0.0023 |
| 10 | 0.0035 | 0.0021 | 0.0056 |
| 15 | 0.0074 | 0.0106 | 0.0087 |
| 20 | 0.0082 | 0.0085 | 0.0084 |

From here on, we will run our model using the architecture demonstrated in Figure 6, which seemed to garner the best results in terms of absolute error.

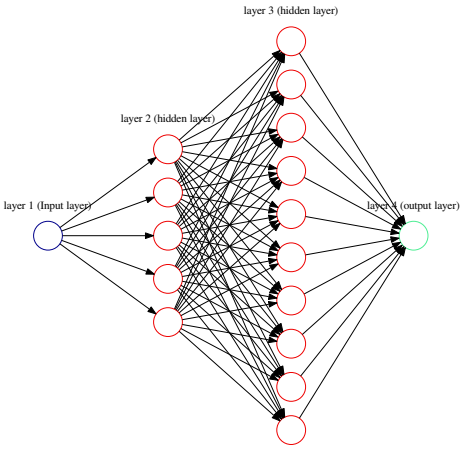


Figure 6: Schematic of Proposed Neural Network Architecture

This changes the network of this model from a single hidden layer to a double hidden layer with five and 10 nodes in the first and second hidden layer, respectively. The mathematical intuition here should be that each individual hidden layer combines its inputs linearly, and therefore cannot add any sense of nonlinearity to the model without additional layers. Generally speaking, solutions to interesting differential equations tend to be nonlinear, and therefore it is natural that a model for generating these solutions should account for nonlinearity.

Figures 7-10 aim to demonstrate the superiority of this architecture with respect to the original single hidden layer architecture.

### 4.2 Activation Functions

Sigmoid functions used to be the standard activation function for neural networks in general. However, there are other effective activation functions that have been proven to solve several common training problems in this field. Specifically, these activation functions tend to address vanishing or exploding gradients.

### 4.3 $L^2$ Regularization

Among various techniques for reducing over-fitting, we tried $L^2$ Regularization. As regularization allows a network to penalize large values for the weights, it allows a network to generalize better

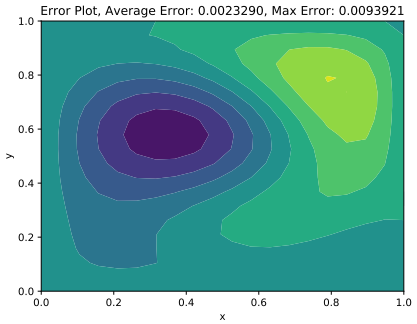


Figure 7: Absolute Error in Original Model

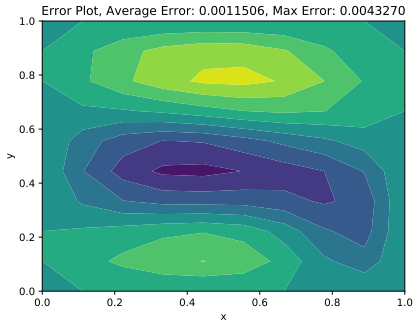


Figure 8: Absolute Error in Modified Model

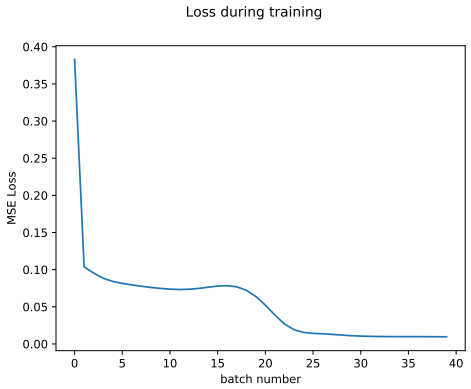


Figure 9: Loss (MSE) in Original Model

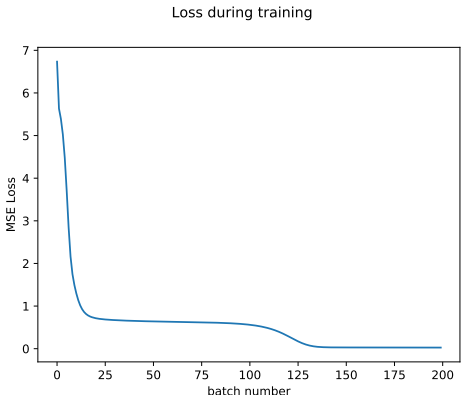


Figure 10: Loss (MSE) in Modified Model

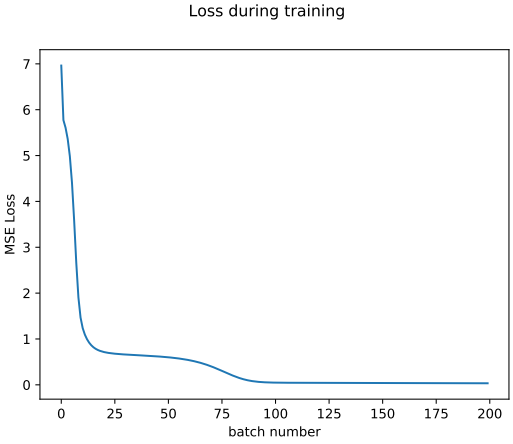


Figure 11: Loss (MSE) in L2 Regularized Model

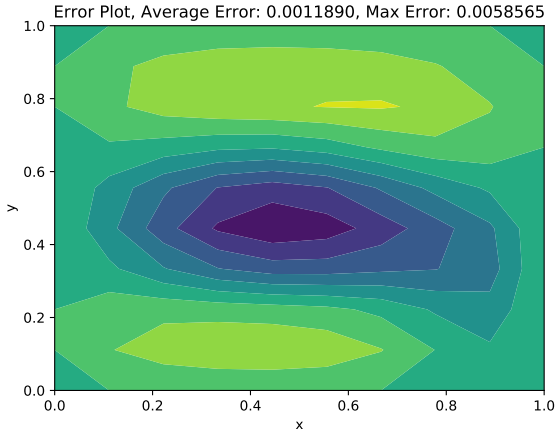


Figure 12: Absolute Error in L2 Regularized Model

to various test sets drawn from the same distribution. The disadvantage of regularization is that it increases bias, which is the error of a model with respect to a particular test set, though the advantage of better generalization is often far more important that regularization becomes an important tool in the process of reducing over-fitting.

### 4.4 Minibatch Training

In typical on-line learning, the gradients are computed and subsequently updated per training example. In batch training, the gradients are computed for a batch of examples, and then updated once for the whole batch. The training set is therefore divided into batches. A batch size of 1 is equivalent to on-line training, whereas a batch size consisting of the entire training set is equivalent to batch training. Our network was trained with various batch sizes in an effort to improve accuracy. Unfortunately,, mini-batch training with a batch size of 5 didn't improve accuracy by much. In some cases, it degrades performance and in others, it improves performance. As a general rule however, as the learning rate decreases to 0, both on-line learning and batch training start to converge in terms of how quickly they converge.

### 4.5 Various Optimization Algorithms

Standard methodology utilizes the Stochastic Gradient Descent algorithm for optimization through backpropagation. However, there are a few others to try that might have advantages, as well as disadvantages.

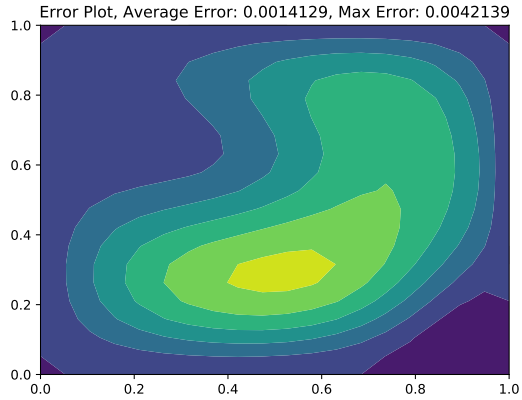


Figure 13: Absolute Error in Minibatch Model

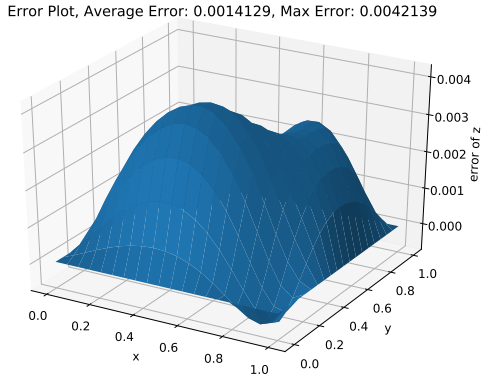


Figure 14: Absolute Error in Minibatch Model

The Adaptive Moment Estimation Optimizer, or Adam, is a method which computes adaptive learning rates for each parameter. It takes from similar optimizers such as Adadelta and RMSprop by storing an exponentially decaying average of past squared gradients, which is likened to the second moment (uncentered variance) of the gradient. It also takes into account the exponentially decaying average of past gradients, which is like an estimate of the first moment.

After repeated trials, it seems that gradient descent continuously returns the results with lowest error. While Adam is the next best, it is generally worse by an absolute error of about 0.002. As a result, we will continue to use stochastic gradient descent as our optimizer.

## 5 Conclusion

Overall, the neural network model for finding analytic solutions to both ordinary and partial differential equations is acceptable with a fairly high accuracy. The improvements to the original model were mostly found in increasing dimensions of the model architecture as well as implementing training through minibatches. While regularization and normalization techniques did sometimes show minor improvements, they were not significant with respect to other changes made to the model.

### Acknowledgments

We would like to thank Professor Chris Curro for guiding us along the project. Thanks to him, we have achieved a deeper understanding of neural networks and its applications to science and technology. Our work is available online in the following repository: `https://github.com/keyao21/diffeq_nn`.

## References

[1] Lagaris, I.E. & Likas, A. & Fotiadis, D.I. (1997) Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *Department of Computer Science, University of Ioannina*. Ioannina, Greece.

[2] Meade Jr, A.J. & Fernadez, A.A. (1994) *Solution of Nonlinear Ordinary Differential Equations by Feedforward Neural networks.* Math. Comput. Modelling, vol. 20, no. 9, pp. 19-44, 1994.

[3] Goodfellow et al (2016) Deep Learning. MIT Press

[2] Tompson, J. & Schlachter et al (2017) *Accelerating Eulerian Fluid Simulation With Convolutional Networks.* Google Inc. *New York University*. New York, NY.