Using Neural Networks to Solve Initial Value Problems

NICK BAUMGARTNER, University of Wyoming, USA
MELISSA BUTLER, University of Wyoming, USA
PAUL SANSAH GYREYIRI, University of Wyoming, USA

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: differential equations, neural networks, initial value problem, boundary value problem, ODE

ACM Reference Format:
Nick Baumgartner, Melissa Butler, and Paul Sansah Gyreyiri. 2026. Using Neural Networks to Solve Initial Value Problems. 1, 1 (February 2026), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM STATEMENT

In the study of differential equations there are many techniques, both analytical and numerical, that are used to find unique solutions. Most of the more complicated equations do not have direct analytic solutions and numerical techniques are the standard to approximate solutions. An emerging field applies deep learning to solving differential equations through Physics Informed Neural Networks (PINNS). These are neural networks that are trained using a set of points in the domain and rely on the physics of the equation to inform the loss function. The goal of this work is to explore this relatively new technique for finding the solution of Ordinary Differential Equations (ODEs), Initial Value Problems (IVPs), and Boundary Value Problems (BVPs). We will look at current literature, prevalent methodology, and showcase the results of PINNs on various ODEs.

2 SIGNIFICANCE

Most natural phenomena can be modelled through differential equations, e.g. fluid dynamics, mechanical vibrations, electrical systems, etc.. Although these equations are elegant and essential for all physical sciences, they are often extremely difficult to solve analytically and numerical approximations are needed to use these necessary equations. However, classic numerical methods have many drawbacks that machine learning is working to overcome, including computation time, domain discretization, high-dimensionality, and non-linearity. There is an increasing interest and research for PINNs approximation for partial differential equations that have addressed many of theses issues, but literature is still lacking for ordinary differential equations. This paper explores the use and benefits of deep learning approximations for ODEs.

Authors' addresses: Nick Baumgartner, University of Wyoming, Laramie, WY, USA; Melissa Butler, University of Wyoming, Laramie, WY, USA; Paul Sansah Gyreyiri, University of Wyoming, Laramie, WY, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Association for Computing Machinery.
Manuscript submitted to ACM

3 LITERATURE SURVEY

Using deep learning to solve differential equations has recently been given significant attention and many literary works have shown the success of this method for partial differential equations (PDEs). Despite the success of neural networks in solving PDEs, there is a lack of scholarly articles applying deep learning to ordinary differential equations (ODEs). There is a plethora of online material and textbook examples, but few articles devoted to ODEs. Knoke et al. explores the possible failures in *Solving differential equations via artifical neural networks: Findings and failures in the model problem* [5] and Dufera uses neural networks for a system of ODEs in *Deep neural network for system of ordinary differential equations: Vectorized algorithm and simulation* [2]. Two papers addressing PDEs that were reviewed are *DeepXDE: A Deep Learning Library for Solving Differential Equations* by Lu et al. [6] and *Solving high-dimensional partial differential equations using deep learning* by Han et al. [4].

DeepXDE is a Python library for solving PINNs, that is not only a research tool but also has the added features of being a classroom ready learning tool. It can solve initial value problems, boundary value problems, inverse problems, time dependent problems, multiphysics problems, and problems with complex domains. The DeepXDE library avoids tedious boundary discretization tasks by supporting constructive solid geometry (CSG) [6]. It also has the ability to solve inverse problems without significant changes to the code used for the associated forward problem [6]. There are several numerical examples using DeepXDE presented, including a 2D Burgers equation using a Residual-Based Adaptive Refinement (RAR). The RAR technique distributes an increased amount of training data points where the loss function is high, to better capture equations with extreme slopes and high variability. Adaptive data from the domain is one benefit of deep learning.

Neural networks do not rely on partitioning of the domain as do current numerical methods, such as finite element, finite volume, and discrete differences. Other benefits of PINNs are discussed in /citelu and /citehan. Since neural networks are compositions computing derivatives becomes a simple exercise in chain rule. There are four common methods for computing the necessary derivatives in PINNs, analytically by hand, numerical approximation, symbolic software (e.g., Mathematica, Maple, etc.), and automatic differentiation (AD) [6]. A specialized technique of AD using backpropagation is used in deep learning and takes advantage of the composite function nature of neural networks. The forward pass computes values and the backward pass computes any needed derivatives, using chain rule reduces computations to only one forward pass and one backward pass. The number of required passes for previous numerical methods are dependent on the mesh used and can get computationally expensive. PINNs can be applied to the strong form of the PDE, instead of a corresponding energy functionals and Galerkin projections, to eliminate the errors in truncation and numerical integration of the variation forms, that are common in classical methods. Some methods still use a varitional formulation.

Han et al. [4] creates a varitational formulation of the PDE using backward stochastic differential equations. This overcomes the difficulty of solving PDEs of high dimension. Numerical results are presented for the Black-Scholes equation, the Hamilton-Jacobi-Bellman equation, and the Allen-Cahn equation, all of which have unique difficulties including non-linearity. Many PDEs become impractical due to the curse of dimensionality, i.e. the computation cost increases exponentially as we increase the dimensions [4]. Han et al. explains that the compositional nature of neural networks approximates allows complicated functions to be approximated via a composition of simpler functions where classic techniques require an additive method, greatly decreasing computation expense. However, we need more research into the "theoretical framework" explaining the extreme effectiveness of multi-layer neural networks [4].

Quantifying the error of a PINNs approximation is also a open research topic. Lu et al. decomposing the total error into the approximation error, the generalization error, and the optimization error so we can balance a good approximation versus overfitting. An analysis is presented of the existence of a neural network that satisfies the boundary value problem, simultaneously and uniformly. They refer to a theorem of derivative approximation using neural networks, which relies on dense sets, to show that feed-forward neural nets can "simultaneously and uniformly approximate any function and its partial derivatives," if enough neurons are used [6]. However, we risk overfitting if the network is too large and must also consider the generalization error.

4 DATASET DESCRIPTION

For each IVP and BVP that we investigate our dataset consists of a set of points X in the domain and a set of zeros, F, of the same size. Although we use equispaced points for X, it is not necessary and many problems benefit from a nonuniform distribution. Since all ODEs can be written as $F(x, y) = 0$ we set all desired outputs to 0.

5 METHODOLOGY

There are several methods and loss functions to approximate solutions to ODEs using neural networks. We have adapted the method and code presented in [3]. Our goal is to find the function $y : \Omega \subseteq \mathbb{R} \to \mathbb{R}$ that satisfies,

$$\begin{cases} y'(x) = f(x, y) \\ y(x_0) = y_0 \in \mathbb{R} \end{cases},$$

for a given function $f : \mathbb{R}^2 \to \mathbb{R}$ and initial value y_0. We will set up and train a neural network to approximate y with a network containing one hidden layer. We let P be the parameters. For our network we define P by a collection of $w_0 \in \mathbb{R}^M, b_0 \in \mathbb{R}^M, w_1 \in \mathbb{R}^M$, and $b_1 \in \mathbb{R}$, where M is the number of nodes in our hidden layer. Let $w_0[j]$ represent the j^{th} element of w_0, for $j = 1, \ldots, M$, and similarly for $b_0[j], w_1[j]$. With this we explicitly define P as

$$P = (w_0[1], w_0[2], \ldots, w_0[M], b_0[1], b_0[2], \ldots, b_0[M], w_1[1], w_1[2], \ldots, w_1[M], b_1),$$

and can see $P \in \mathbb{R}^{3M+1}$. Our approximate solution at any given point $x \in \Omega$ is

$$\hat{y}(x; P) = \sum_{j=1}^{M} h(x w_0[j] + b_0[j]) w_1[j] + b_1,$$

where $h : \mathbb{R} \to \mathbb{R}$ can be various functions. We used a soft plus function defined by

$$\phi(z) = \log(1 + e^z)$$

and a sigmoid function defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

To set up the network we begin by defining a set of points in our domain, $X = \{x_i\}_{i=1}^n \subset \Omega$; these will be the input data for training our neural network and do not need to be equispaced. Let $H : \mathbb{R}^n \to \mathbb{R}^n$ be a vector valued function where

$$H[j] = [h(x_1 w_0[j] + b_0[j]), h(x_2 w_0[j] + b_0[j]), \ldots, h(x_n w_0[j] + b_0[j])]^\top,$$

for $j = 1, \ldots, M$. Our neural network can be visualized in Figure 1. We define our loss function $L : \mathbb{R}^n \times \mathbb{R}^{3M+1} \to \mathbb{R}$, as

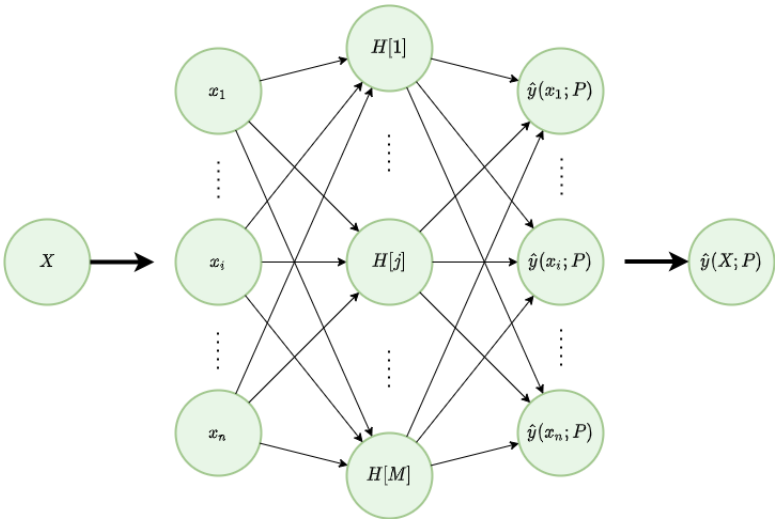

Fig. 1. Single Layer Neural Network for $\hat{y}(X;P)$

a linear combination of the sum of squares error of our approximation and the error in the initial condition, where

$$L(\hat{y}(X;P)) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\partial}{\partial x_i}\hat{y}(x_i;P) - f(x_i,\hat{y}(x_i;P))\right)^2 + (\hat{y}(x_0;P) - y_0)^2,$$

and $\frac{\partial L}{\partial P} : \mathbb{R}^n \to \mathbb{R}^m$ is a vector valued function defined as,

$$\frac{\partial L}{\partial P} = \left(\frac{\partial L}{\partial P_1}, \frac{\partial L}{\partial P_2}, \ldots, \frac{\partial L}{\partial P_{3M+1}}\right).$$

To train the network, we initialize P and several training variables, $m, \eta \in \mathbb{R}$ and $v \in \mathbb{R}^{3M+1}$. For each forward pass we calculate G

$$G = \frac{\partial L}{\partial P}(\hat{y}(X, P + mv)).$$

Then for the backward pass we update,

$$v = mv - \eta G$$
$$P = P + v.$$

6 RESULTS

We solve a series of ordinary differential equations of a variety of orders and conditions. The two main conditions that are looked at in problems are that of an initial value problem, IVP, and a boundary value problem, BVP. Both of these conditions provide their own hurdles to finding the solution of equation, but merely act as constraints on the solution of the problem.

The primary goal of this work is to analyse the speed at which a Neural Network will solve these ODEs, and in the instances where we can find an analytic solution to the differential equation, how accurate our approximation is. This later condition can be replicated by looking at the plots of the loss functions as we defined in the previous section.

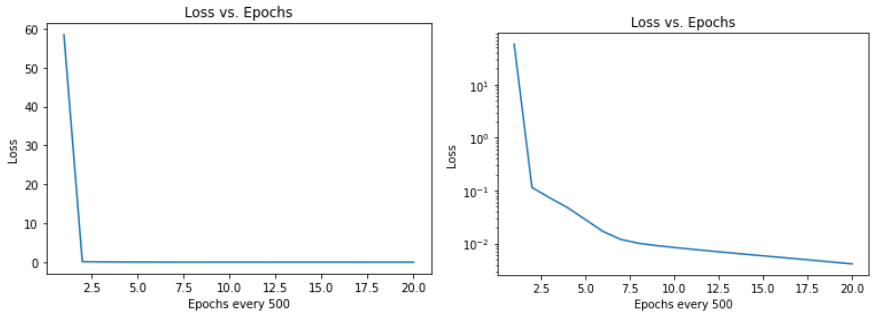

Fig. 2. Plots of the Loss versus the number of Epochs our neural network has run. (left) Standard, (right) semi log plot.

For our purposes we will use a single hidden layer with 80 nodes, unless otherwise stated.

6.1 First Order: Initial Value Problem

The first problem under investigation is a classic in the testing of solving N.N.ODEs. We look at the first order IVP

$$\begin{cases} \dot{y} = 2xy \\ y(0) = 1 \end{cases}$$

which we know has an analytic solution that satisfies the initial conditions as

$$y = e^{-2x}.$$

While this is a simplistic equation, its analytic solution makes it ideal for looking at the accuracy of the solution. As we can see from Figure 2, when looking at the loss found using the parameters derived from the neural network there is an immediate drop as the solution is making its initial operations to approach the solution. However, once we reach an "approximate" solution we can see that there is a fairly linear decrease in the loss with respect to the number of epochs run.

6.2 Second Order: Boundary Value Problem

We now look at the solutions to Boundary Value Problems, which are considered slightly more complex to solve numerically using classical techniques. Instead of using a time marching method for solutions, it is common to create an IVP that matches the same general characteristics and then use a root finding method to shrink the distance between the initial condition and boundary condition. This technique, while effective, can often be extremely time consuming and costly.

Using the neural network we are able to directly solve the BVP and avoid the conversion to such an IVP. We will examine the system

$$\begin{cases} -u'' - \sin(2x)\cos(u) + 2u = -2 - \sin(2x)\cos(x^2 - 1) + 2(x^2 - 1) \\ u(-1) = 0, \quad u(1) = 0 \end{cases}$$

which has been constructed to have the analytic solution

$$u(x) = x^2 - 1.$$

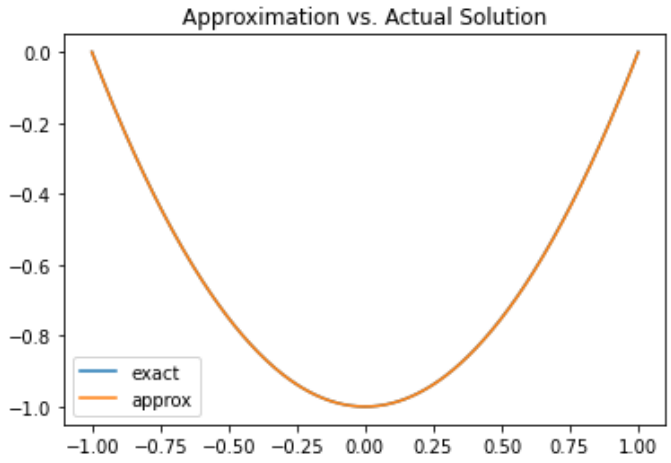

Fig. 3. Actual and Approximate solution to the BVP with analytic solution $u(x) = x^2 - 1$.

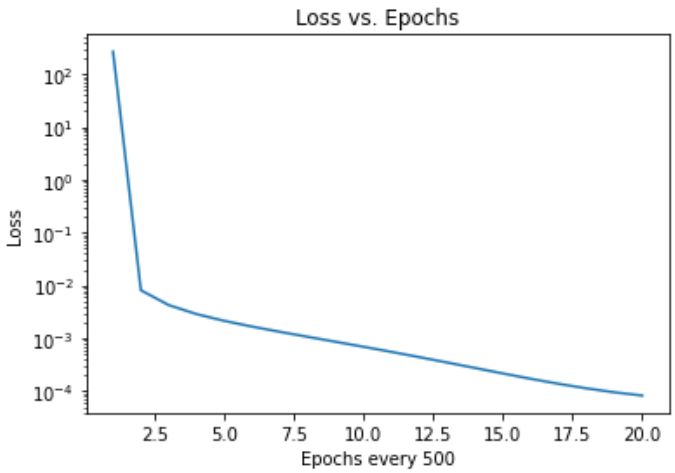

Fig. 4. Convergence of the loss for the BVP with analytic solution $u(x) = x^2 - 1$.

We can see in Figure 3 that our actual and approximate solution are near identical after the 10,000 epochs that we've run. The convergence of the loss, found in Figure 3, shares that of the other problems where after a brief period where the model finds an approximate solution quickly and then spends time refining the solution.

Additionally we may consider problems that do not have direct analytic solutions. The wave equation is a well known partial differential equation, however when we fix our equation with a constant acceleration we arrive at an equation whose added complications do not allow an analytic solution.

$$\begin{cases} u'' + \lambda u = 0 \\ u(0) = 0, \quad u(L) = 0 \end{cases}$$

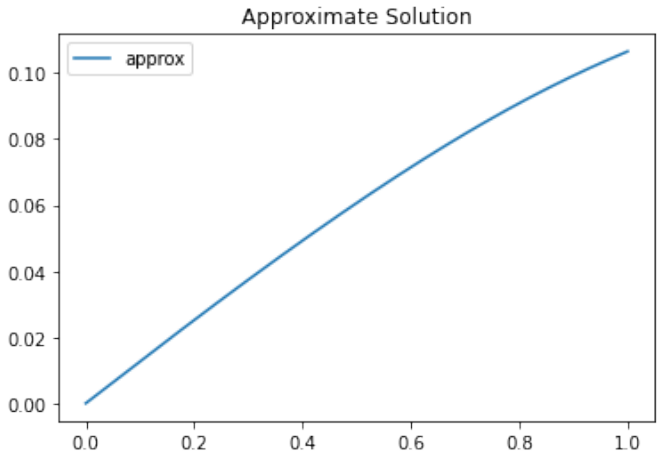

Fig. 5. Approximate solution for the wave equation with constant acceleration using the neural network

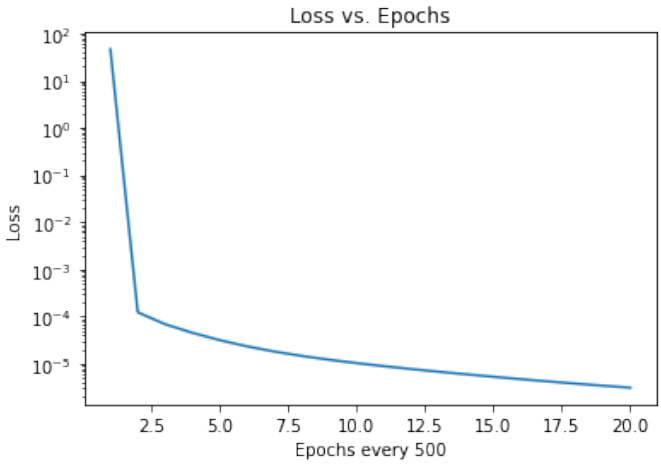

Fig. 6. Convergence rate of the loss for the wave equation

So instead we use our neural network to solve the resulting system. The value of λ is defined as $n^2\pi^2/L$ as the solution would be expressed as an infinite series if done by hand. As we can see in fig. 6 the notion of using a neural network to find a solution for a problem is extremely fast in terms of the convergence of the system.

6.3 Performance Analysis

We explored the performance of the system by considering the different activation functions, the number of neurons, and the number of training points for the first order IVP in section 6.1. With 20000 epochs and 20 neurons with a single hidden layer and using the sigmoid activation function our model attains a loss of 0.0.0006799.

6.3.1 Activation Function. We evaluated performance of the system for the sigmoid activation function and the soft plus activation function. For a model with 80 neurons, even though the soft plus function converges early, the sigmoid function turn out to have a better overall performance after the 20000 epochs. The sigmoid function results in a reduced error when trained on 20 neurons as shown in table 1. fig. 8 shows the plot of the losses for sigmoid and soft plus for every 1000 epochs.

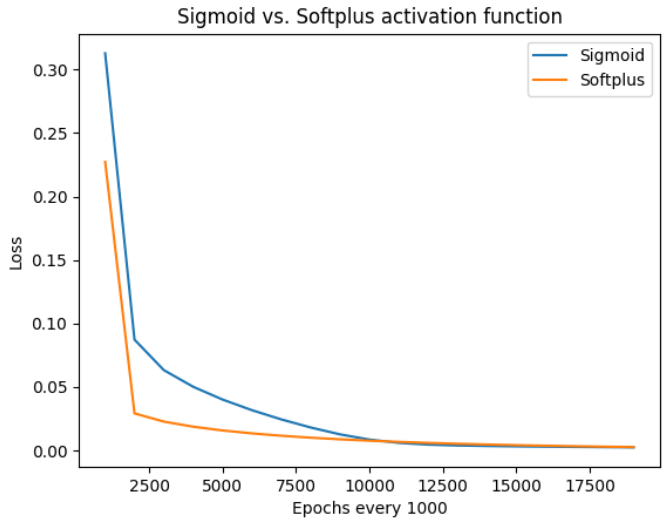

Fig. 7. Sigmoid vs Soft Plus Activation Functions

6.3.2 Number of Neurons. We evaluated the performance of the system based on the number of neurons. We trained the model on 20, 40, 80 and 160 neurons. Though there were no significant differences in the final results, however, there is a trade-off between the initial error and the number of neurons. **??** shows the plot of the loss against every 2000 epoch of the number of neurons. The model with 20 neurons started with a high initial error, followed by 40, 80 and 160 with the lowest initial error. It is also important to note that, even though the number of neurons affected the initial error, however, the overall performance as shown in the table 1 does not guarantee higher neurons had the lowest error. The model with 40 neurons performed better when the soft plus activation function was used with an error of 0.0008159. When the sigmoid activation function was used, the model with 20 neurons performed better with a loss of 0.0006799.

Table 1. Model Performance

Number of Neurons	Sigmoid	Soft Plus
20	0.0006799	0.0010989
40	0.0026600	0.0008159
80	0.0023749	0.0024706
160	0.0033131	0.0107603

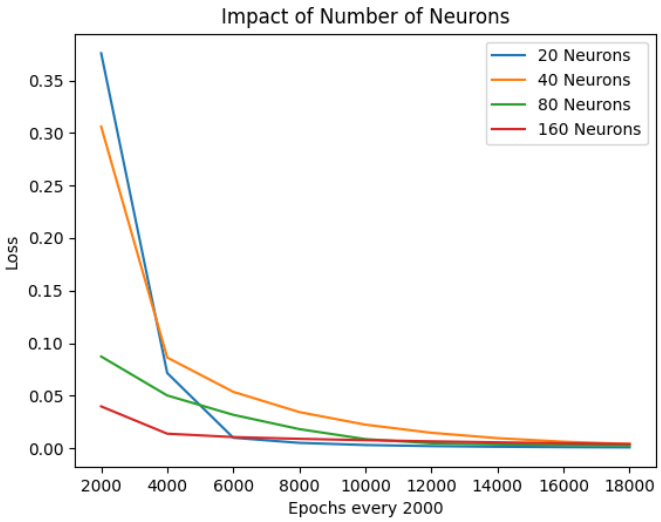

Fig. 8. Impact of Number of Neurons

6.3.3 Number of Training Points. A major challenge for classic numerical solvers, is their accuracy is dependent on the discretization of the domain. A neural network can handle a much courses mesh for its training data. We have tested the standard IVP given above for $n = 11, 101, 401$ training points and show the results in Figure 9 - Section 6.3.3. We see a higher accuracy for a larger number of training points for early passes, which is expected. However, even with $n = 11$, we see the same convergence rate and a loss of 0.009 by pass 8000. This shows the unnecessary need for a fine discretization in training the neural network.

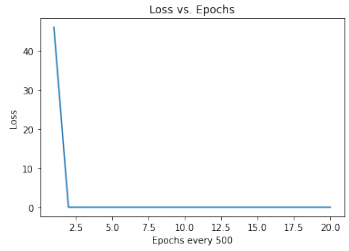

Fig. 9. n=11

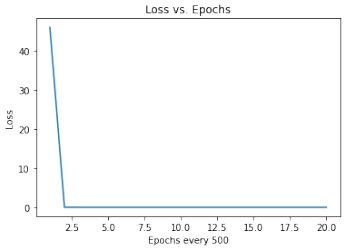

Fig. 10. n=101

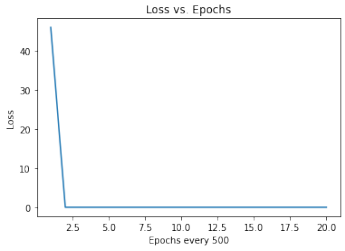

Fig. 11. n=401

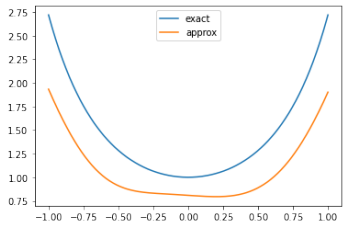

Fig. 12. n=11, epochs = 2000

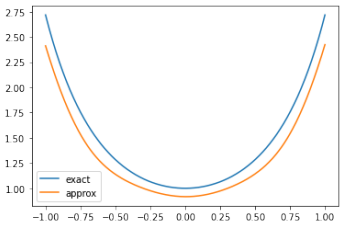

Fig. 13. n=11, epochs = 4000

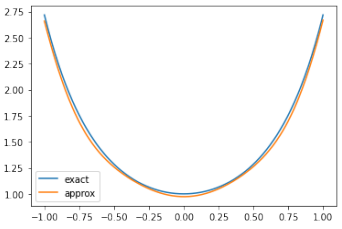

Fig. 14. n=11, epochs = 8000

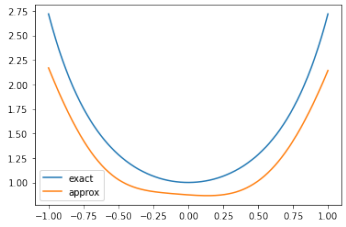

Fig. 15. n=101, epochs = 2000

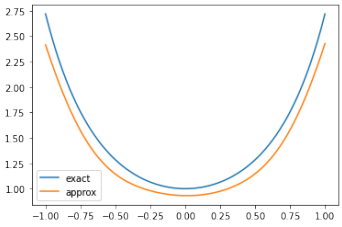

Fig. 16. n=101, epochs = 4000

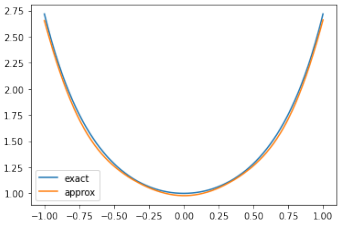

Fig. 17. n=101, epochs = 8000

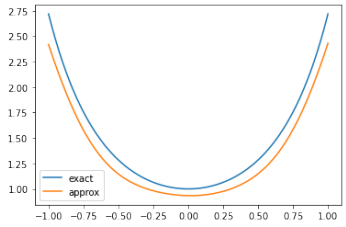

Fig. 18. n=401, epochs = 2000

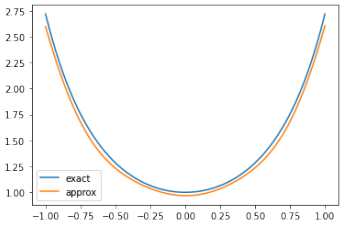

Fig. 19. n=401, epochs = 4000

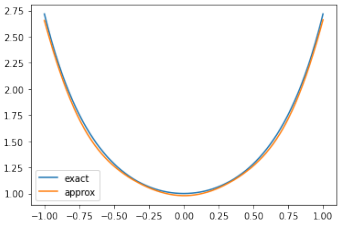

Fig. 20. n=401, epochs = 8000

7 CHALLENGES

The primary package that was used for the vast majority of the code is a relatively new package and while it is gaining popularity, it is still constantly being updated. This unfortunately means that many of the examples that we used as a starting point were out of date and it required a significant amount of reading to determine which code to work with.

Further challenges became present as we modified the style of the problem. While a traditional first order IVP was relatively easy to solve with the code, it became more challenging to solve these problems when we focused on other initial conditions, such as a first derivative condition on the boundary. After working with the code we determined to investigate two point boundary problems as the main focus to avoid this issue.

Additional challenges presented themselves in the solutions of second order boundary value problems and the convergence of such problems. The primary issue arose during the training stage of the model where the results of the training approached infinity, this was primarily due to the value of the learning rate and the momentum, which is a similar error we might find from a solution that uses a classic step size.

8 CONCLUSIONS

Overall we see that the accuracy of the solutions for these Neural Network Differential Equations are exceptional even with very few epochs and nodes used in the training. There are still some issues to investigate regarding the various boundary conditions, such as using an initial condition where the derivative is defined. We find that all of the differential equations that were investigated for this project, including the equations that do not have compact analytic solutions, were able to reach a solution that was close to the actual in approximately 1500 epochs, further epochs always reduced the remaining error to an insignificant point.

Future work would be to apply this same technique to Partial Differential Equations. While the majority of ODEs can be solved analytically, the opposite is true for PDEs and as such it is important to derive techniques that work for finding the solution of such problems after one proves the existence and uniqueness of such a solution. The techniques that are commonly used for finding such solutions typically require heavy discretization and refinement of the domain which slows down the computation time greatly. A technique that would reduce the computation time would be invaluable to the community.

9 POSSIBLE VENUES

With further work, we could present this paper at University of Wyoming, Department of Mathematics, CAM Seminar; University of Wyoming, School of Computing, CAM Seminar; SIAM Conference on Mathematics of Data Science.

10 CONTRIBUTIONS

Nick Baumgartner: Project Statement, ODE construction, Image Generation, Plot Generation, Challenges, Conclusion, Slides, Presentation

Melissa Butler : Project Statement, Significance, Literary Survey, Dataset Description, Methodology, Number of Training Points Results, Slides, Presentation

Paul Sansah: Performance analysis, no of neurons, activation function and plots under this section

Additional support, help, and resources from Silba Dowell.

11 PRESENTATION

Video Presentation

REFERENCES

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[2] Tamirat Temesgen Dufera. 2021. Deep neural network for system of ordinary differential equations: Vectorized algorithm and simulation. *Machine Learning with Applications* 5 (2021), 100058. https://doi.org/10.1016/j.mlwa.2021.100058

[3] Andreas Gillgren and Simon pf. 2020. *Solving Differential Equations with Neural Networks.* https://github.com/Gillgren/Solving-Differential-Equations-with-Neural-Networks

[4] Jiequn Han, Arnulf Jentzen, and Weinan E. 2018. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences* 115, 34 (2018), 8505–8510. https://doi.org/10.1073/pnas.1718942115 arXiv:https://www.pnas.org/content/115/34/8505.full.pdf

[5] Tobias Knoke and Thomas Wick. 2021. Solving differential equations via artificial neural networks: Findings and failures in a model problem. *Examples and Counterexamples* 1 (2021), 100035. https://doi.org/10.1016/j.exco.2021.100035

[6] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. 2021. DeepXDE: A deep learning library for solving differential equations. *SIAM Rev.* 63, 1 (2021), 208–228.