Demonstrating Neural Networks as Universal Approximators

Completed for the Certificate in Scientific Computation and Data Sciences
Spring 2024

Reuben Rapose
Bachelor of Science in Mathematics
Department of Mathematics
College of Natural Sciences

Oscar Gonzalez
Professor of Mathematics
Department of Mathematics

Table of Contents

1. Introduction

It is important to first understand the motivation behind this study. Differential Equations show up in modeling almost every aspect of this world, including finance, population, biology, and especially physics. There have been notable famous named Differential Equations that show up everywhere, with the heat and wave equations perhaps being some of the most well studied. Due to their popularity, mathematicians have been working on numerical methods to solve these equations, or rather, approximate their solutions. Common methods to solve such equations include Finite Difference Method, Finite Element Method (FEM), Finite Volume, Collocation, and more. Given some domain, these methods involve discretizing the domain, and then use a series of computations to approximate the function's value at each point in the discretization. These methods have a few drawbacks. For one, as the domain becomes more complex, the discretization becomes more complex. This involves significant overhead from the user, and such overhead becomes tedious and takes a lot of effort. Also, the process often falls short of a wholistic understanding of the solution in the sense that the approximation is only found at certain points or intervals. This makes plotting a continuous function dependent on how many steps are used. It is not always guaranteed that choosing a different mesh will automatically translate into the same solution. Furthermore, these methods require a notable amount of knowledge and understanding of the mathematics that surround the boundary conditions of the differential equation and its domain, especially in Partial Differential Equations (PDEs). In the spirit of increasing speed and accessibility, one would want to try and reduce the overhead and prior knowledge necessary to as large of an extent as possible. The situation is often as follows. There is a differential equation to be solved of which the solution is known to exist and be unique, but the solution itself is unknown. Additionally, the necessary boundary or initial conditions are observed. Ideally, there exists a calculator such that the equation itself, its domain, and the boundary or initial conditions are input and the solution is output. The neural network would serve as this calculator.

Neural networks were first introduced in 1943 and were studied sparsely through the following decades[1]. However, its recursive nature requires large computing power, and thus remained almost stagnant. In the past decade however, research in this specific area has exploded, but it first rose to prominence in 1989 when a paper proved that feedforward neural networks serve as universal approximators[3]. This theorem, among others, will be explored further in Section 2.3. Although it is only in recent developments that the true mathematics behind how a neural network achieves success are being well understood[2], the theorem provides the crucial existence property which encouraged further research to be done.

Solving differential equations is a huge part of scientific computing, and in general this study will show how most problems in scientific computing can be transformed to a problem solvable by a neural network. It will also provide an explanation of the relation to Kolgomorov's Superposition Theorem[2], which provided a mathematical baseline foundation for understanding neural networks as universal approximators, a concept given by Hornik's paper[3]. This provided strong motivation to even consider this type of method. Specifically, it is based on the idea that a feedforward network is no different than an arbitrarily large sum of compositions of functions, a notion that will be thoroughly explained in Section 2.1. Finally, the study will also highlight the challenges one often comes across when using neural networks to solve differential equations. Apart from the aforementioned robust domain handling, neural networks also shine in high order PDEs. Such equations usually have less theoretical understanding, and so the traditional numerical methods are less developed. Because whether a neural network will converge is some-

what independent of the equations themselves, they end up being reliable, hindered only by the complexity of the solution itself and derivative calculations.

2. Neural Network

Artificial Neural Networks are loosely based on the human brain, and the neurology behind the way thoughts are developed. The brain has countless pathways that connect neurons, and such neurons pass along information to each other and build upon each other to approach coherency. Similarly, an artificial neural network is built up of nodes, and each node acts as a neuron that holds information. Connecting these neurons are edges, or paths, that help affect the information that is passed on. Finally, each node has a pre-existing bias that acts on the information before it is finalized, in a sense. Weights and biases of a network are scalar values that carry out the aforementioned effects on information that is passed through nodes. While these boil down to mostly sums, products, and a so-called activation function, there is a remarkable effect that takes place as the network trains. The following sections rigorously explain the process and how this architecture seamlessly translates to mathematically defined functions.

2.1 Architecture

A classic neural network is orchestrated by its layers and nodes in each layer, in addition to input and output nodes. Each node has some numerical dimension embedded into it, but the options are quite expansive. Traditional neural networks can be generalized to accept any input in $\mathbb{R}^{n\times m}$, and each node can be any dimension. For the purpose of function approximations, specifically functions $f : \mathbb{R}^n \to \mathbb{R}$, the conventional approach is for each node to be a scalar value, and the multiple variables are handled at the input layer, which would feature n nodes.

Let L denote the number of layers in a network. This implies the network has L hidden layers, denoted $\{\ell_i\}_{i=1}^{L}$ where ℓ_0 is the input layer and ℓ_{L+1} is the output layer. It now makes sense to consider each $\ell_i \in \mathbb{R}^{n_i}$ where each n_i indicates the number of nodes in each layer. Each node will be referred to as $v_i^{(k)} \in \mathbb{R}$ indicating it belongs to layer ℓ_k and it is the ith node in that layer (where the ordering is arbitrary but must be held consistent). This is, $\ell_k = [v_1^{(k)}, \ldots, v_{n_k}^{(k)}]$. Each node has a bias, $b_i^{(k)} \in \mathbb{R}$ where $i = 1, \ldots, n_k$ and $k = 1, \ldots, L+1$. The input nodes do not have biases. Finally, between every two layers exist the edges that connect every node in one layer to every node in the following layer. These will be denoted $w_{i,j}^{(k)}$, which indicates the weight in between ℓ_{k-1} and ℓ_k that connects $v_i^{(k-1)}$ to $v_j^{(k)}$, where $k = 1, \ldots, L+1$, $i = 1, \ldots, n_{k-1}$ and $j = 1, \ldots, n_k$. For each of the examples, the convention $n_1 = n_2 = \cdots = n_L$ will be used, such that each hidden layer will have the same number of nodes. In terms of function approximation, the network in Figure 1 would approximate $f : \mathbb{R}^3 \to \mathbb{R}$.

Lastly, a neural network designer must choose an activation function, $\sigma : \mathbb{R} \to \mathbb{R}$. This function has two main purposes. It serves to help stabilize the values, and depending on the choice of activation function, it helps the network learn nonlinear patterns[2]. The network calculates its node values recursively as follows:

$$v_i^{(k)} = \sigma\left(\sum_{j=1}^{n_{k-1}} w_{j,i}(v_j^{(k-1)}) + b_i^{(k)} \right) \tag{2.1}$$

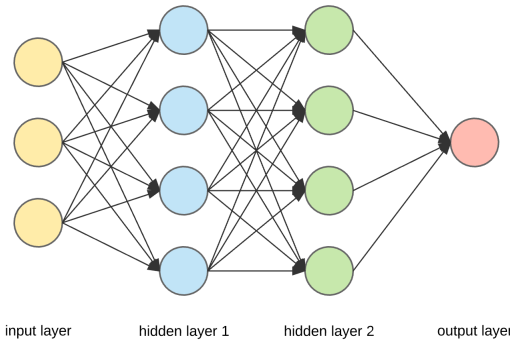


Figure 1: 4x2 Neural Network with 3 inputs, 1 output

2.2 Application to Functions

The output of the neural network will be denoted by $\mathcal{N}_p : \mathbb{R}^n \to \mathbb{R}^m$, where p can be generalized as the set of all weights and biases. Assuming the number of nodes in each hidden layer is n_0, the input $z \in \mathbb{R}^n$ and $N_p(z) \in \mathbb{R}^m$, it follows that $p \in \mathbb{R}^q$, where q, the total number of parameters (weights and biases), is given by

$$q = n_0 n + n_0 m + n_0^2(L-1) + n_0 L + m$$

the first two terms calculating number of weights for the first layer and output layer, the third term calculating number of weights between hidden layers, and the fourth and fifth term calculating the number of biases for all nodes. The examples in this study will use the convention that each p_i will begin as a value in $[-1, 1] \subset \mathbb{R}$.

Now, given any mathematically defined function $f : X \to Y$, we require the domain to be restricted such that $X \subset \mathbb{R}^n$ is compact and $Y \subset \mathbb{R}^m$, where $f|_X$ will contain the values of interest. In other words, define $\mathcal{N}_p : X \to Y$ by $x \in X$ with $x = \ell_0$ and $\mathcal{N}_p(x) \in Y$ has $\mathcal{N}_p(x) = \ell_{L+1}$. It can be helpful to extend the domain beyond values of interest so that the neural network smooths over any boundary points as well. Specifically, one has $f(x) = \mathcal{N}_{p^{(i)}}(z) + \epsilon_i$ where ϵ_i denotes the error term at the ith iteration and $p^{(i)}$ is the vector of parameters used at the ith iteration. Note, from equation (2.1), $\mathcal{N}_p$ is defined recursively, culminating in the node(s) on the output layer.

2.3 Universal Approximators

In order to justify using neural networks to approximate a function or solution, there have been many results that manage to place weak enough conditions to bring about the concept of "universal". These results are strongly backed by Kolgomorov's Superposition Theorem, which essentially proves that a multidimensional continuous function can be broken down into a compositions of lower dimensional functions. Kolgomorov found this result in an attempt to answer Hilbert's 13th question. Specifically, this result justified using a neural network to approximate any n-dimensional function, as a neural network is composed of compositions of functions on $\mathbb{R}$. This relation is thoroughly explained in [5]. The quest to find weaker conditions continued.

The theorem with the weakest conditions is found in Hornik's paper[3]. We summarize Hornik's result below. Let M^n represent the class of Borel measurable functions defined on $\mathbb{R}^n$ as M^n.

This set contains "virtually all functions relevant in applications"[3], and it includes the general function $\mathcal{N}$ that represents any neural network. The paper then presents the result, stated slightly differently:

Universal Approximator Theorem (Version 1). *For any $n \in \mathbb{N}$, the set of functions on compact subset $X \subset \mathbb{R}^n$ that can be approximated by one layer in a feed forward neural network with a continuous activation function is dense in M^n.*

Compactness in $\mathbb{R}^n$ is equivalent to being closed and bounded. Density is best understood with some notion of distance, which, for functions, can be given by

$$d(f,g) = \left(\int_{\mathbb{R}^n} |f-g|^2 \, dx \right)^{\frac{1}{2}} \tag{2.2}$$

This is given by the L^2-norm. We have the following definition.

Definition. *Given a metric space (X,d), $A \subset X$ is **dense** in X is for every $\epsilon > 0$ and $x \in X$, there exists $a \in A$ such that $d(x,a) < \epsilon$.*

Now to apply this to M^n, let $\Pi \subset M^n$ be the set of functions on $\mathbb{R}^n$ that can be approximated by one layer in a feed forward neural network. We get the following restating of the theorem:

Universal Approximator Theorem (Version 2). *For each $f \in M^n$ and each $\epsilon > 0$, there exists $g \in \Pi$ such that on a compact set the following conditions hold: i) $d(f,g) < \epsilon$, ii) there exists $p \in \mathbb{R}^q$ for some $q \in \mathbb{N}$ such that $\mathcal{N}_p(x) = g(x)$ where $\mathcal{N}$ uses a continuous activation function.*

The first condition comes from density, and the second comes from the definition of the set Π. The conclusion follows that given any $f \in M^n$, there exists a set of parameters such that a one layer feed forward network of these parameters with a continuous activation function can get arbitrarily close to f. It is now clear that theoretically a network of more than one layer can also successfully approximate the function, assuming the first layer has the necessary size.

These specific results do not put constraints nor do they provide guidance on the "width" of the network i.e. the number of nodes in each layer. This could leave room for error in network construction, in that designing a network too small might miss the necessary size of vector p and therefore do a poor job in approximating the function. Therefore, it makes logical sense to always err on the side of a larger network. This minimizes the risk that the network's parameter vector is too small, and as previously mentioned, having more parameters than necessary cannot worsen the approximation. However, this paper will show that for less complex functions, the size of p need not be large (relative to conventional sizes for categorical determinations) while still outputting accurate approximations.

2.4 Algorithms and Optimization

Over the years, there have been several algorithms employed to calculate the output of a network. The custom algorithm used throughout much of this paper is a brute force feed forward algorithm, for which the pseudo code is below. Writing and using this algorithm is helpful to truly understanding the logic behind how a neural network trains. A custom algorithm also helps significantly in data collection. Considering the notation from before, p is the vector of weights and biases, z is the input, σ is the activation function, and $n_0 = \cdots = n_L$ is the size of each layer.

Algorithm 1 Neural Network Output

$output, pind \leftarrow 0$ ▷ $pind$ = parameter vector index ▷ construct first layer
$prev\_nodes \leftarrow []$
for $nnind$ in range(n_0) **do** ▷ $nnind$ = new layer's node index
    $weighted \leftarrow 0$
    for zi in z **do**
        $weighted \leftarrow weighted + p[pind] * zi$
        $pind \leftarrow pind + 1$
    end for
    $node \leftarrow \sigma(weighted + p[pind])$
    $pind \leftarrow pind + 1$
end for
for $lind$ in range($L - 1$) **do** ▷ construct hidden layers
    $new\_nodes \leftarrow []$
    for $nnind$ in range(n_0) **do**
        $weighted \leftarrow 0$
        for $pnind$ in range(n_0) **do** ▷ $pnind$ = previous layer's node index
            $weighted \leftarrow weighted + (p[pind] * prev\_nodes[pnind])$
            $pi \leftarrow pi + 1$
        end for
        $node \leftarrow \sigma(weighted + p[pi])$
        append $node$ to new_nodes
        $pi \leftarrow pi + 1$
    end for
    $prev\_nodes \leftarrow new\_nodes$
end for
for $pnind$ in range($nodes\_per\_layer$) **do** ▷ get $output \in \mathbb{R}$
    $output \leftarrow output + (p[pind] * prev\_nodes[pnind])$
    $pi \leftarrow pi + 1$
end for
return $output + p[pi]$

Some of these approximations are done in c++, using two-point central finite difference for any gradient calculations. There were a few attempts at performing autodifferentiation using a c++ package, however this proved to be far more time consuming, and the accuracy to efficiency ratio was not strong enough to warrant its usage.

The optimization here is Steepest Gradient Descent. It is an iterative method given by the following equation, $p_{new} = p_{old} - \alpha \nabla L(z)$, where α is the learning rate, and where the function is evaluated at z. The gradient is given by

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial p_1}(z) \\ . \\ . \\ . \\ \frac{\partial L}{\partial p_q}(z) \end{bmatrix}$$

Some of the differential equation examples utilizes PyTorch, a Python package for machine learning. The algorithms here are optimized in the backend, and Python's large accessibility to autodifferentiation is very handy. Specifically, the examples use the optimizer Stochastic Gradient Descent. This differs slightly from standard Gradient Descent in that rather than using all values in the input, it takes a sample. This saves on the computational resources of each gradient calculation, but lowers the convergence rate.

3. Demonstrations

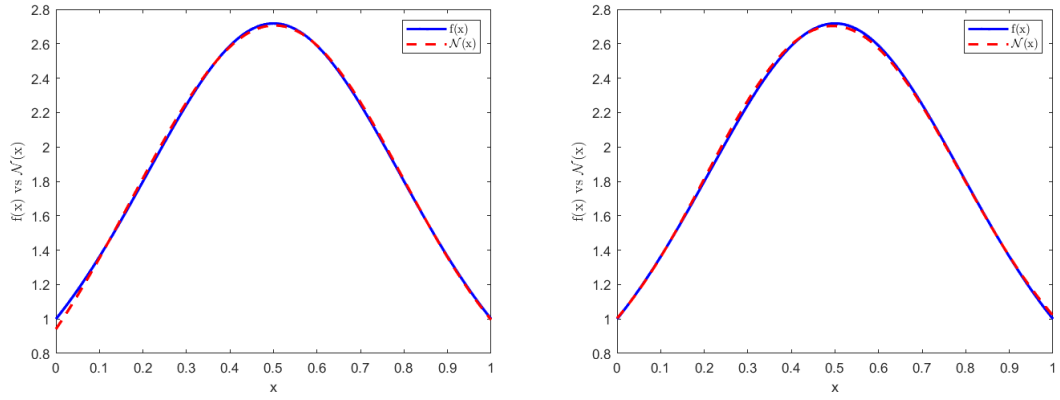
For function $f : X \to Y$, the domain X must be discretized for training. This process can be uniform or randomly chosen. For this study, the majority of examples utilize a random mesh. As will be shown later, both have advantages and disadvantages, but more specifically a random mesh struggles with the end points of the domain. One way to circumvent this is to manually add the endpoints after a random mesh has been generated. Uniformity can push the network to interpolate rather than generally approximate, which can lead to high accuracy given by the loss function, but with more complex functions, this can result in a less effective plot, and so a finer mesh would expose such inaccuracies.

3.1 Function Approximation

The first and perhaps most straightforward task for a neural network is to approximate a known function. For example, take $f : \mathbb{R} \to \mathbb{R}$ to be defined $f(x) = e^{\sin(\pi x)}$. Then the goal is to minimize the difference between $\mathcal{N}_p(x)$ and $f(x)$ by finding p. There are many formulas for this minimization, most popular being the Mean Squared Error. For a mesh z of size N, this gives the loss function: $\mathcal{L}(z) = \frac{1}{N} \sum_{i=1}^{N} (\mathcal{N}_p(z_i) - f(z_i))^2$. It would be easiest to approximate a smooth function (discussed in Example 1.4), although smoothness is not a necessary condition according to Hornik's theorem. Every example that follows were tested multiple times, and any outliers in convergence will be noted. We start with "simple" smooth functions, and extend to functions with multiple extrema, and eventually demonstrate a piecewise continuous function.

Example 1.1 Let $f : \mathbb{R} \to \mathbb{R}$ be defined $f(x) = e^{\sin(\pi x)}$. Here we use two networks, a 2x1

(2 nodes on each hidden layer, 1 hidden layer) and a 2x2, and restrict f to $X = [0, 1]$. On this domain, f has a single bump. The graphs are shown below.



(a) Approximating $f(x) = e^{\sin(\pi x)}$ on $[0, 1]$ with 2x1 network

(b) Approximating $f(x) = e^{\sin(\pi x)}$ on $[0, 1]$ with 2x2 network

Figure 2

These are both tiny networks, yet they approximate this function extremely well. There was a general trend with larger networks taking fewer iterations to converge, however this trend plateaued relatively quickly as the network sized increased significantly. Also, as noted earlier, the network tends to struggle with the endpoints with random input domains, and in more complex examples, it became necessary to alter the randomized domains. Indeed, the domain $[.1, .9]$ is fit perfectly by both networks, and so extending the domain would likely avoid the potential hassle of weighting certain domain points. Unfortunately, in differential equations, this concept is far less convenient since the physics are often reported on a specific domain, and so extending the domain would usually require some kind of change of variables.

Example 1.2 Let $f : \mathbb{R} \to \mathbb{R}$ be defined $f(x) = .1x^5 + 2x^4 - x^3 - 2x^2 + x$. Define $\mathcal{N}_p$ to have 2 hidden layers and 6 nodes on each layer. We skip straight to a 6x2 network due to the higher complexity in f (Figure 3).

Example 1.3 Let $f : \mathbb{R} \to \mathbb{R}$ be defined $f(x) = \sin(x^3)x^{-1.1}$. The goal of this example is to test how the network handles multiple changes in direction. This was tested on a 8x2 network. As expected, it took longer to converge than the previous two examples, but nevertheless approximated the function well (Figure 4).

Example 1.4 We now move on to a piecewise continuous function, and the results are far less promising. The network successfully trains to the function on just one subset of the domain, and then struggles to change to fit the function on the rest (Figure 5). Let $f : \mathbb{R} \to \mathbb{R}$ be defined $f(x) = \begin{cases} -3x + 1 & x \in [0, \frac{1}{6}] \\ \sin(\pi x) & x \in [\frac{1}{6}, 1] \end{cases}$ (1.4.1). After significant tuning to parameters, as well as weighting the loss function, the model only slightly improved. Theoretically there exists a network that converges to this piecewise function since it meets the criterion of Theorem 1 (since f is borel measurable), but finding the network proved far more difficult. In terms of understanding

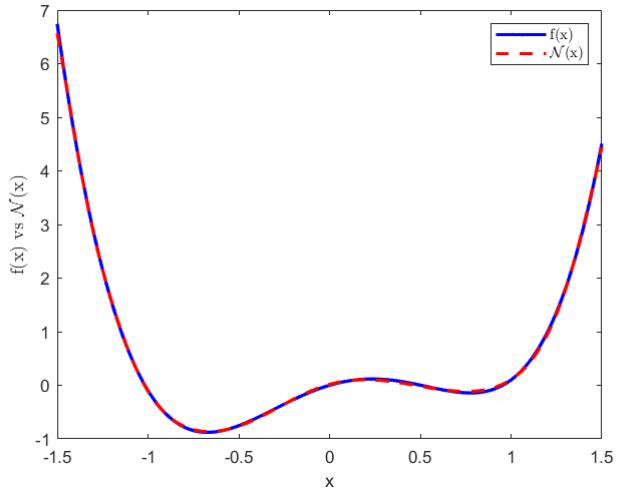


Figure 3: Approximating $f(x) = .1x^5 + 2x^4 - x^3 - 2x^2 + x$ on $[-1.5, 1.5]$

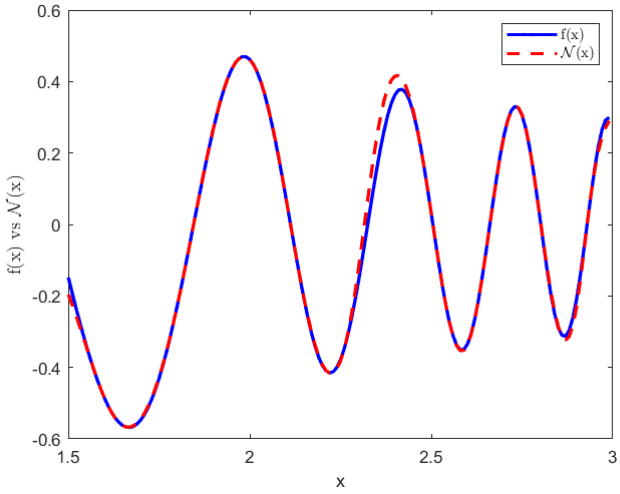
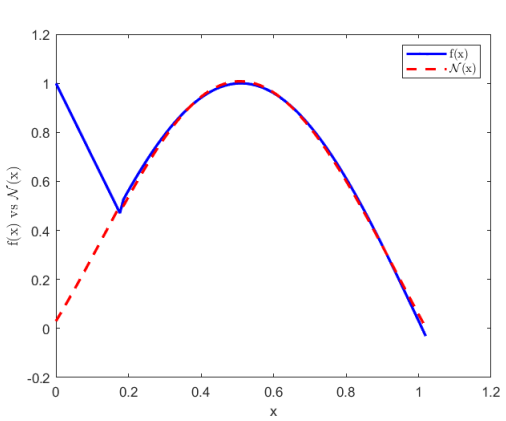
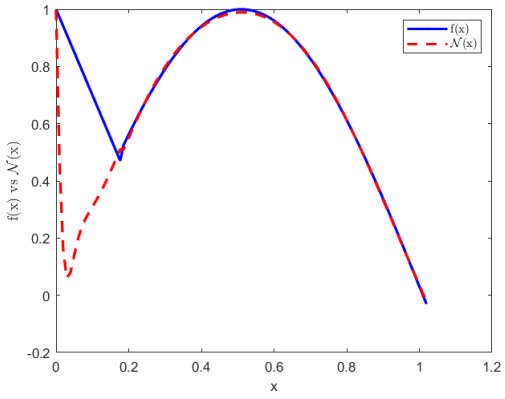


Figure 4: Approximating $f(x) = \sin(x^3)x^{-1.1}$ on $[1.5, 3]$

why the model struggled, this function differs significantly from the three previous examples due to its lack of smoothness. A function is "smooth" (informally) if it has continuous derivatives over a domain. This function is not differentiable at $x = \frac{1}{6}$, and so it is not smooth on $[0, 1]$. In general, many numerical methods rely on a function's smoothness for convergence, and so although neural networks are unique in their universality, it seems that the dependence of their efficacy and convergence rate has overlaps with those of traditional methods.



(a) Approximating $f(x) = (1.4.1)$ on $[0, 1]$



(b) Approximating $f(x) = (1.4.1)$ on $[0, 1]$ with significant tuning

Figure 5

3.2 Physics Informed Neural Networks

When considering the problem of solving differential equations, the notion Physics Informed Neural Networks[6] comes into play. A differential equation is any equation that relates "unknown" functions to their derivatives. Take for instance the simple differential equation, $(*)$ $\frac{d^2y}{dx^2} = 0$ with conditions $y(0) = 1, y(1) = 2$. One can easily verify the solution is $y = x + 1$. This implies that the neural network $\mathcal{N}_p$ has the goal of approximating $\mathcal{N}(x) = x + 1$. However, if the goal is to solve the ODE, one cannot simply use a known solution. Rather, the loss function must be built around the ODE itself with its initial conditions. This is done by solving for 0 in the ODE and its initial conditions respectively. In the above example, it is already in the form $\mathcal{A}(x, y, y', y'', \dots) = 0$ where $\mathcal{A}$ represents some operator as a function of x, $y(x)$, and the derivatives of y with respect to x. Let z be an $N \times n$ dimensional matrix, where n denotes the input dimension of the function to be approximated (for ODEs, $n = 1$), and N denotes the size of the sample input. This, along with the general form where $y(x_0) = y_0$ and $y(x_1) = y_1$, the loss function becomes

$$\mathcal{L}(z, p) = \frac{1}{N} \sum_{i=1}^{N} \left[\left(\mathcal{A}(z_i, \mathcal{N}_p(z_i), \mathcal{N}_p'(z_i), \dots) \right)^2 + (\mathcal{N}_p(x_0) - y_0)^2 + (\mathcal{N}_p(x_1) - y_1)^2 \right]$$

The first term indicates the ODE itself is satisfied by the network, and the rest of the terms force the neural network to satisfy the initial/boundary conditions. The descriptor Physics Informed was coined because the network trains in an untraditional method; rather than being given inputs and corresponding true outputs, it is given inputs and a corresponding equation to solve, the equation usually being motivated by physics (i.e. heat equation, wave equation, etc.).

With this problem comes the computational challenge of solving the derivatives. The traditional way of calculating derivatives is using Finite Difference. For the common two central difference, the calculation is as follows. Given a function f, fixing tunable parameter h, $f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$. The choice of h is usually $h = 1e - 5$. This formula implies the

error is of order h^2, which is small but not always small enough. Taking $\mathcal{L}$ from the example $(*)$,

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\frac{d^2\mathcal{N}_p}{dx^2}(z_i)\right)^2 + (\mathcal{N}_p(0)-1)^2 + (\mathcal{N}_p(1)-2)^2\right]$$

$$\Rightarrow \frac{\partial L}{\partial p_i} = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{\partial}{\partial p_i}\left(\frac{d^2\mathcal{N}_p}{dx^2}\right)^2 + \frac{\partial}{\partial p_i}(\mathcal{N}_p(0)-1)^2 + \frac{\partial}{\partial p_i}(\mathcal{N}_p(1)-2)^2\right]$$

it now requires a calculation of a third derivative from that first term. This compounds the accuracy. To solve this problem, automatic differentiation is employed. This is a method of harnessing the chain rule to derive any function. Using the above example, each arithmetic operation (including multipication, addition, activation function, etc.) on p_i can be defined as its own function (many of which lie in the deep recursion definition of $\mathcal{N}_p$). Then the entire function $\mathcal{L}$ becomes a composition of such functions. With this string of compositions, the derivative is computed by carefully applying the chain rule to each composition.

Lastly, when considering differential equations, the choice of activation function becomes quite important. Specifically, for an nth order differential equation, it is first required that the activation function $\sigma \in C^{n+1}$. This is, the function must have $n+1$ continuous derivatives. Furthermore, the function must avoid non uniform spikes in values within the $n+1$ derivatives. The famous ReLU activation function, for instance, commonly used in reinformence learning, is a poor choice since its derivative does not exist at 0. Since the tanh function requires many derivatives for its range to become "unstable", it tends to be reliable[2].

3.3 Ordinary Differential Equations

An Ordinary Differential Equation (ODE) is any differential equation where the unknown function is defined on $\mathbb{R}$, or a function of one independent variable. Famous ODEs include the motion of the oscillator, derived from Newton's Second Law: $m\frac{d^2x}{dt^2} + kx = 0$. The following examples test the network's ability with slight differences. Example 3.1 uses finite difference for the derivatives. Due to the second derivative in Example 3.2, it was deemed preferable to skip straight to an automatically differentiating environment.

Example 3.1 Beginning with a simple example, consider the differential equation

$$\frac{dy}{dx} + xy = 0 \quad \text{with} \quad y(0) = 1 \tag{3.1}$$

One can solve this analytically to get the solution $y = e^{-\frac{1}{2}x^2}$. Using a 2x2 Network, this network was trained without any adjustments to the domain. This bare bones method was used in part to test the convergence of a network "without help", trained without a known solution. This is a testament to the robustness of the network to converge to simpler functions even when using an implicit loss function.

Example 3.2 We will now move on to the second order differential equation, $\frac{d^2x}{dt^2} + \frac{k}{m}x = 0$ (3.2) where k, m are constants. This is a 2nd order linear homogeneous equation, which we can solve analytically. It has the general solution $x(t) = A\cos(\omega t) + B\sin(\omega t)$, where $\omega = \sqrt{\frac{k}{m}}$ and A, B are constants. For simplicity, we will use $k = m = 1$. This is typically turned into an initial

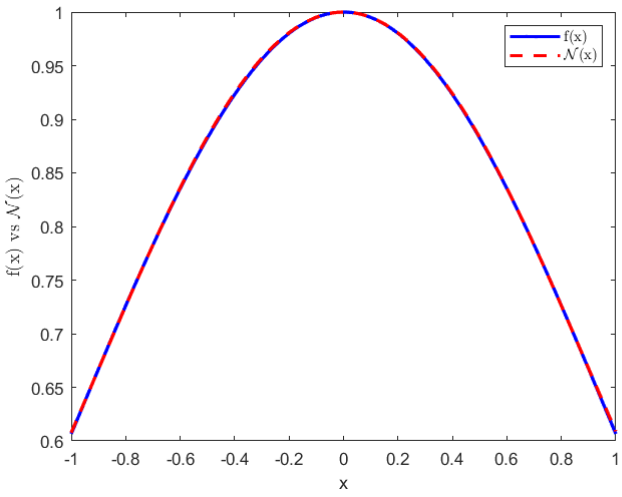
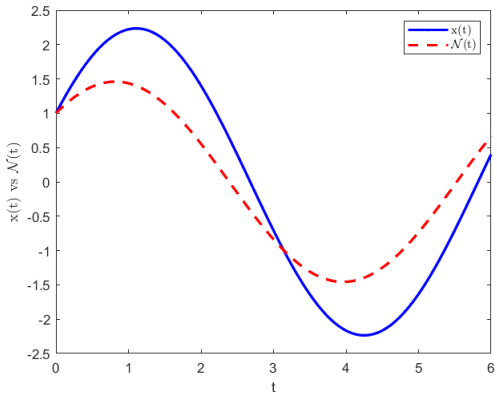


Figure 6: Solution to ODE (1.1) with initial condition (1.2) on $[-1, 1]$

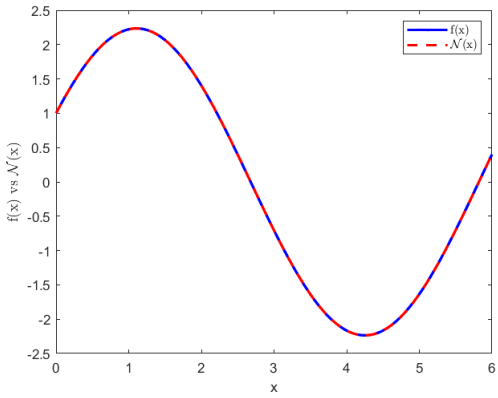
value problem:

$$\frac{d^2x}{dt^2} + x = 0 \quad \begin{cases} x(0) = 1 \\ x'(0) = 1 \end{cases} \tag{3.2}$$

It is easy to check that the solution is then $x(t) = \cos(t) + 2\sin(t)$. This problem used PyTorch, a learning rate of .03, and Stochastic Gradient Descent was the optimization algorithm.



(a) Solving ODE (3.2) on $[0, 6]$



(b) Solving ODE (3.2) on $[0, 6]$ with significant tuning

Figure 7

Here the network converged to the general solution $x(t) = \cos(t) + \sin(t)$ with constants $A = B = 1$. Some tuning was necessary in the form of lowering the tolerance level as well as weighting the initial conditions slightly higher in the loss function. This was done by adding multiple elements of 0 into the sample input.

3.4 Partial Differential Equations

Partial differential equations (PDEs) are the general form of ordinary differential equations, such that they include functions of any number of variables. Solving PDEs are essential to many areas of science, finance, general modeling, and more. Trying to find a balance between accuracy and efficiency has been the focus of numerical analysis for more than half a century. However, a universal approximator can theoretically surpass any of the traditional problems that arise for complicated problems (FEM and other traditional methods still outperform in efficiency for lower order problems). That being said, with the added dimensions comes an added challenge. Reconsidering Kolgomorov's Superposition Theorem, a Neural Network solving differential equations of more than one variable fully embellishes his notion. The network now needs to "learn" in two different directions, and as one might expect, this increases the overall cost.

The biggest obstacle in this method is calculating derivatives of varying order and of different variables. Even with automatic differentiation, this can be computationally expensive. Furthermore, the nature of the domain is less straight forward. There are usually more complex initial and boundary conditions to satisfy. In fact, most PDEs have at least one initial or boundary condition that becomes a function on a lower dimension. This is essentially a function approximation problem embedded into the PDE problem, but only making up a small percentage. These challenges showed up in various ways which will be discussed. The main PDE we consider is the one spatial-dimensional heat equation. This is a function of space and time, and we study the equation under the Dirichlet Conditions:

$$u_t - c^2 u_{xx} = 0, \quad u(x,0) = g(x), \quad u(0,t) = u(1,t) = 0$$

where our domain is $X = [0,1] \times [0,1]$, where $g(x) \in C^2(X)$, a twice differentiable function, represents the initial condition. In this case, the loss function becomes:

$$L(z,p) = \frac{1}{N_{pde}} \sum_{i=1}^{N} \left(\frac{\partial \mathcal{N}_p}{\partial t}(x_i,t_i) - c^2 \frac{\partial^2 \mathcal{N}_p}{\partial x^2}(x_i,t_i) \right)^2$$

$$+ \frac{1}{N_{ic}} \sum_{i=1}^{N} \left(\mathcal{N}_p(x_i,0) - g(x) \right)^2$$

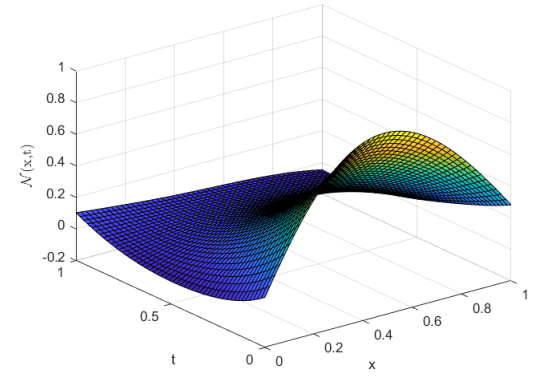
$$+ \frac{1}{N_{bc1}} \sum_{i=1}^{N} \left(\mathcal{N}_p(0,t_i) \right)^2 + \frac{1}{N_{bc2}} \sum_{i=1}^{N} \left(\mathcal{N}_p(1,t_i) \right)^2$$

where z is the input sample. We choose $N_{pde}, N_{ic}, N_{bc1}, N_{bc2}$ to be the size of the input dedicated to each specific term, and ideally N_{pde} is dominating in order for the network to train most specifically against the PDE itself.

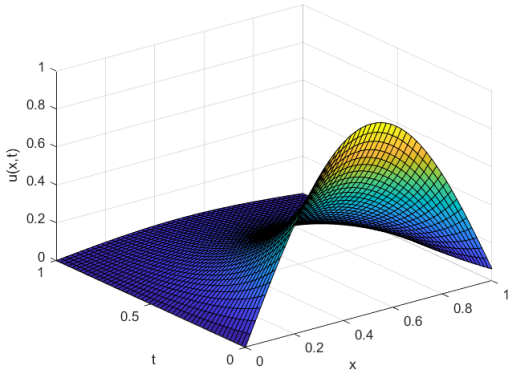
Example 4.1 Consider the heat equation

$$u_t - \frac{1}{\pi} u_{xx} = 0 \text{ with } \begin{cases} u(x,0) = \sin(\pi x) \\ u(0,t) = u(1,t) = 0 \end{cases} \tag{3.3}$$

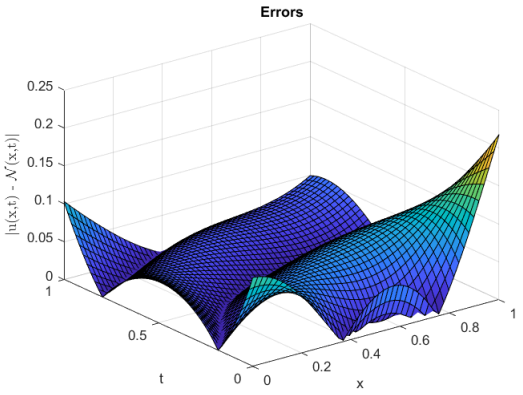
Using an 8x2 network with learning .001, we get the following approximation. Clearly, the network struggled significantly (Figure 8). This approximation was done using finite difference for all the derivatives, and it is likely that this hurt the accuracy.



(a) Approximating solution to (3.3) on $[0,1]$



(b) True solution to (3.3) on $[0,1]$



(c) Errors: $|u(x,t) - \mathcal{N}(x,t)|$ on $[0,1]$

Figure 8

Example 4.1 In this example, we consider a very similar heat equation but test the network to handle the rest of the sin period.

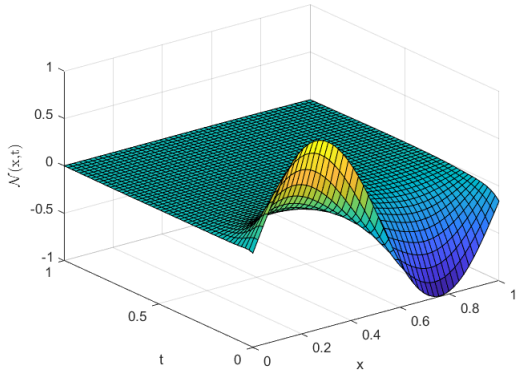
$$u_t - \frac{1}{\pi} u_{xx} = 0 \quad \text{with} \quad \begin{cases} u(x,0) = \sin(2\pi x) \\ u(0,t) = u(1,t) = 0 \end{cases} \tag{3.4}$$

Additionally, with this algorithm, small parts of the loss function derivative was analytically calculated like so:

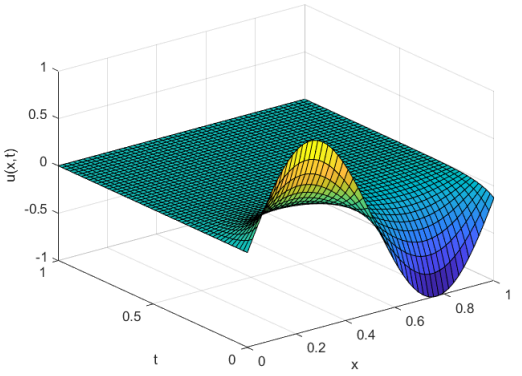
$$L(z,p) = \frac{1}{N_{pde}} \sum_{i=1}^{N} \left(\frac{\partial \mathcal{N}_p}{\partial t}(x_i,t_i) - c^2 \frac{\partial^2 \mathcal{N}_p}{\partial x^2}(x_i,t_i) \right)^2$$

$$+ \frac{1}{N_{ic}} \sum_{i=1}^{N} \left(\mathcal{N}_p(x_i,0) - g(x) \right)^2$$

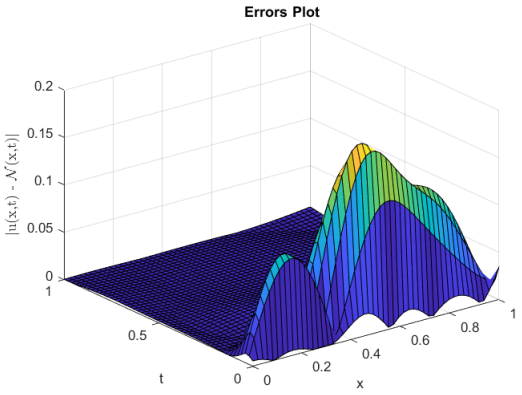
$$+ \frac{1}{N_{bc1}} \sum_{i=1}^{N} \left(\mathcal{N}_p(0,t_i) \right)^2 + \frac{1}{N_{bc2}} \sum_{i=1}^{N} \left(\mathcal{N}_p(1,t_i) \right)^2$$



(a) Approximating solution to (3.4) on $[0, 1]$



(b) True solution to (3.4) on $[0, 1]$



(c) $|u(x,t) - \mathcal{N}(x,t)|$ on $[0, 1]$

Figure 9

$$\Rightarrow \frac{\partial L}{\partial p_k}(z,p) = \frac{2}{N_{pde}} \sum_{i=1}^{N} \left(\frac{\partial \mathcal{N}_p}{\partial t}(x_i,t_i) - c^2 \frac{\partial^2 \mathcal{N}_p}{\partial x^2}(x_i,t_i) \right) \cdot \frac{\partial}{\partial p_k} \left(\frac{\partial \mathcal{N}_p}{\partial t}(x_i,t_i) - c^2 \frac{\partial^2 \mathcal{N}_p}{\partial x^2}(x_i,t_i) \right)$$

$$+ \frac{2}{N_{ic}} \sum_{i=1}^{N} \left(\mathcal{N}_p(x_i,0) - g(x) \right) \cdot \left(\frac{\partial}{\partial p_k} \mathcal{N}_p(x_i,0) - g'(x) \right)$$

$$+ \frac{2}{N_{bc1}} \sum_{i=1}^{N} \left(\mathcal{N}_p(0,t_i) \cdot \frac{\partial}{\partial p_k} N_p(0,t_i) \right) + \frac{2}{N_{bc2}} \sum_{i=1}^{N} \left(\mathcal{N}_p(1,t_i) \cdot \frac{\partial}{\partial p_k} N_p(1,t_i) \right)$$

using chain rule and that derivative operators agree with addition. Since $g'(x)$ has a known derivative, it does not require approximation. Either way, it is likely that this change played a part in the significantly more accurate plot, as the accuracy is less compounded than if finite difference were to be applied to the whole function.

4. Closing Remarks

In general, we see that neural networks are very capable of solving equations. The difficulty lies in derivative calculation, choosing the right algorithm and optimization, and the ever present tuning of parameters. Throughout the project, the speed difference between C++ and Python became very apparent. It would stand to reason that if the same machine learning packages that are available in Python were available in C++, or perhaps similar new languages like Rust, the computational expense of machine learning would come down. There was a steep jump in difficulty of approximation when considering PDEs, but even still, the machine learning models came close to the solution. Given more computational power, it is likely the errors would have been reduced even more. It is unclear right now if neural networks will replace the traditional numerical methods of solving differential equations, but the case is building. Indeed, strategies are being developed to deal with the heavy memory and computational resources that machine learning tends to bring[6]. However, with the rise of supercomputers, the differences in computational power between the methods will likely become trivialized, and then neural networks' ability to *predict* values at any point, rather than the specificied descritization, will really shine.

In all, this study shows the power of the standard Artificial Neural Network, but there is still knowledge to be gained. Most importantly, mathematicians are interested in understanding the convergence rate of neural networks. Heavy research is being done into this, and as a result, other types of networks, including Convolutional Neural Networks and Transformer Neural Networks, have started to shine[7]. The progress is being made in theoretical analysis of neural networks, and the combination of this with the advances in technology is likely to put Neural Networks at the forefront of Numerical Methods and Scientific Computing.

5. References

1. Bohdan Macukow. Neural Networks – State of Art, Brief History, Basic Models and Architecture. (2023)
2. J. Watt, R. Borhani, A. Katsaggelos: Machine Learning Refined. Cambridge University Press, (2016) Pages 307-328
3. K. Hornik, M. Stinchcombe, H. White: Multilayer feedforward networks are universal approximators, Neural Networks, Volume 2, Issue 5, (1989), Pages 359-366.
4. M. Lai and Z. Shen: The Kolmogorov Superposition Theorem can Break the Curse of Dimensionality When Approximating High Dimensional Functions, (2021).
5. R. Hecht-Nielson. Kolmogorov's Mapping Neural Network Existence Theorem, (1987).
6. Sifan Wang, Shyam Sankaran, Hanwen Wang, Paris Perdikaris. An Expert's Guide to Training Physics-informed Neural Networks. (2023)
7. M. Villani, P. McBurney, The Topos of Transformer Networks, (2024).