Numerical Solution of Differential Equations Using Neural Networks

Case Study in Scientific Computing

Candidate Number: 1060612

1 Introduction

In this paper, we discuss how to solve systems of partial differential equations on the domain $\Omega = [0,1]^n$ and with boundary constraints $\mathcal{B}$ using neural networks. We introduce our notation

$$\begin{cases} G_1(\vec{x}, \vec{u}, \nabla\vec{u}, \nabla^2\vec{u}) = 0, \\ G_2(\vec{x}, \vec{u}, \nabla\vec{u}, \nabla^2\vec{u}) = 0, \\ \dots \\ G_N(\vec{x}, \vec{u}, \nabla\vec{u}, \nabla^2\vec{u}) = 0. \end{cases} \tag{1}$$

Where $\vec{u} = (u^{(1)}, u^{(2)}, \dots, u^{(N)})$ and $\vec{x} = (x_1, x_2, \dots, x_n)$. The notation in (1) can be simplified to $\vec{G}(\vec{x}, \vec{u}, \nabla\vec{u}, \nabla^2\vec{u}) = \vec{0}$. Also, for simplicity we will use the notation $G_i(\vec{x}, \vec{u}) = G_i(\vec{x}, \vec{u}, \nabla\vec{u}, \nabla^2\vec{u})$ for $i \in \{1, 2, 3, \dots, N\}$.

We will define a neural network with one hidden layer and derive the derivative of the loss function with respect to the weights and biases. In this paper, we will only consider neural networks with one hidden layer because a one hidden layer neural network with sufficient neurons in the hidden layer can approximate any continuous function this is discussed in [1], this is known as the Universal Approximation Theorem.

Also, we will discuss different approaches to satisfy the boundary constraints $\mathcal{B}$. Then a derivation of calculating the n^{th} derivative of the sigmoid function. Additionally, we will implement the theory in Python to demonstrate this is a valid method for solving systems of PDEs.

$n \in \mathbb{N}_+$	Number of inputs to the neural network.
$m \in \mathbb{N}_+$	Number of neurons in hidden layer.
$N \in \mathbb{N}_+$	Number of outputs from neural network.
$\vec{x} \in \mathbb{R}^n$	The input to the neural network.
$\vec{y} \in \mathbb{R}^N$	The output from the neural network.
$\vec{u} \in \mathbb{R}^N$	The solution to (1).

Table 1: Variables in our Neural Network

Furthermore, we will cover multiple examples for calculating numerical solutions to a system of differential equations using neural networks in Python. Examples include solving a PDE in one and two dimensions and extending the neural network to solve a coupled ODE and a heat equation in one spatial dimension. Therefore, the reader can use the code to solve a custom system of PDEs. In this paper, we will use the notation from the table (1).

2 Loss Function

We need to define the loss function. The neural network aims to minimize the loss function. Then unseen data should be calculated by the neural network correctly given there is a pattern in the data set. Any point $\vec{x} \in \Omega$ can be used for training the neural network to solve PDEs. We will use N_p^n points $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(N_p^n)}$ to train our neural network with N_p^n equal to N_p to the power of n where n is the number of inputs to the neural network . Also, we use the notation $\mathbf{x} = \begin{bmatrix} \vec{x}^{(1)} & \vec{x}^{(2)} & \ldots & \vec{x}^{(N_p^n)} \end{bmatrix}$.

We need our output from the neural network to agree with the boundary conditions. There are two approaches for implementing boundary conditions into the loss function.

2.1 Method 1: Use γ

In this approach the output of the neural network is the solution to the PDE. We will use notation $\vec{y}(\vec{x}) = \vec{y}(\vec{x};\Theta)$ and $\vec{G}(\vec{x}) = \vec{G}\left(\vec{x}, \vec{y}(\vec{x}), \frac{\partial \vec{y}}{\partial x}|_{\vec{x}}, \frac{\partial^2 \vec{y}}{\partial x^2}|_{\vec{x}}\right)$thus for one PDE in one spatial dimension with Dirichlet boundary conditions we get the loss function

$$\mathcal{L}(\mathbf{x};\Theta) = \frac{1}{2N_p}\sum_{k=1}^{N_p} G_1(x^{(k)})^2 + \frac{\gamma}{2}((y(0) - u_0^{(1)})^2 + (y(1) - u_1^{(1)})^2). \tag{2}$$

Where $u_0^{(1)}$ is the boundary value at $x = 0$ and $u_1^{(1)}$ is the boundary value at $x = 1$. Now taking the derivative with respect to Θ gives us

$$\begin{aligned}\frac{\partial \mathcal{L}(\mathbf{x};\Theta)}{\partial \Theta} = \frac{1}{N_p}\sum_{k=1}^{N_p} G_1(x^{(k)}) \frac{\partial G_1}{\partial \Theta}\bigg|_{x^{(k)}} &+ \gamma(y(0) - u_0^{(1)}) \frac{\partial y}{\partial \Theta}\bigg|_{x=0} \\ &+ \gamma(y(1) - u_1^{(1)}) \frac{\partial y}{\partial \Theta}\bigg|_{x=1}. \end{aligned} \tag{3}$$

Now we generalise the loss function for a coupled PDE system in multiple dimensions. With $\vec{y}^{(k)} = \vec{y}(\vec{x}^{(k)}, \Theta)$

$$\mathcal{L}(\mathbf{x};\Theta) = \frac{1}{2NN_p^n}\sum_{k=1}^{N_p^n}\sum_{i=1}^{N} G_i(\vec{x}^{(k)}, \vec{y}^{(k)})^2 + \frac{\gamma}{2N_p^{n-1}}\sum_{k=1}^{N_p^{n-1}}\sum_{i=1}^{N_\mathcal{B}} \mathcal{B}_i^2(\vec{x}_R^{(k)}). \tag{4}$$

Where $\mathcal{B}$ gives an array of the boundary constraints, $N_{\mathcal{B}}$ represents the number of boundary constraints. We have $\vec{x}_R^{(k)} \in \mathbb{R}^{n-1}$. By differentiating with respect to Θ we get

$$\frac{\partial \mathcal{L}(\mathbf{x};\Theta)}{\partial \Theta} = \frac{1}{NN_p}\sum_{k=1}^{N_p}\sum_{i=1}^{N} G_i(\vec{x}^{(k)},\vec{y}^{(k)})\frac{\partial G_i}{\partial \Theta}\bigg|_{x^{(k)}} + \frac{\gamma}{N_p^{n-1}}\sum_{k=1}^{N_p^{n-1}}\sum_{i=1}^{N_{\mathcal{B}}} \mathcal{B}_i(\vec{x}_R^{(k)})\frac{\partial \mathcal{B}_i}{\partial \Theta}\bigg|_{\vec{x}_R^{(k)}} \tag{5}$$

For (3) and (5) we need $\frac{\partial G_i}{\partial \Theta}$ and $\frac{\partial \mathcal{B}_i}{\partial \Theta}$, thus we use the chain rule to get

$$\frac{\partial G_i}{\partial \Theta} = \frac{\partial G_i}{\partial \vec{y}}\frac{\partial \vec{y}}{\partial \Theta} + \frac{\partial G_i}{\partial \vec{y}_x}\frac{\partial \vec{y}_x}{\partial \Theta} + \frac{\partial G_i}{\partial \vec{y}_{xx}}\frac{\partial \vec{y}_{xx}}{\partial \Theta}, \tag{6}$$

$$\frac{\partial \mathcal{B}_i}{\partial \Theta} = \frac{\partial \mathcal{B}_i}{\partial \vec{y}}\frac{\partial \vec{y}}{\partial \Theta} + \frac{\partial \mathcal{B}_i}{\partial \vec{y}_x}\frac{\partial \vec{y}_x}{\partial \Theta} + \frac{\partial \mathcal{B}_i}{\partial \vec{y}_{xx}}\frac{\partial \vec{y}_{xx}}{\partial \Theta}. \tag{7}$$

In (2) and (3) we have $\mathcal{B} = \left[y(0) - u_0^{(1)},\ y(1) - u_1^{(1)}\right]$.

The advantage of this method is the output of the neural network is the solution. Therefore, it doesn't require any additional steps to get the solution.

2.2 Method 2: Redefine Problem With Substitution

In this approach the neural network does not give us the solution to the PDE. Instead we use a substitution to remove the boundary constraints $\mathcal{B}$. The substitution will be of the form

$$\vec{u}(\vec{x}) = \vec{P}(\vec{x}) + \vec{Q}(\vec{x}) \odot \vec{y}, \tag{8}$$

where $\odot$ denotes element-wise multiplication. Therefore, the substitution involves finding a $\vec{Q}(\vec{x}) : \mathbb{R}^n \mapsto \mathbb{R}^N$such that $\vec{Q}(\vec{x})\odot\vec{y}$ can not impact the boundary constraints $\mathcal{B}$ and finding a $\vec{P}(\vec{x}) : \mathbb{R}^n \mapsto \mathbb{R}^N$ such that $\vec{P}$ satisfies the boundary constraints $\mathcal{B}$. This substitution leads to training an unconstrained neural network and for learning we use the loss function in (4) with $\gamma = 0$. In this paper, anytime we state $\gamma = 0$ this means we have used a substitution to enforce our boundary constraints $\mathcal{B}$.

The main advantage of this approach is that the solution is guaranteed to fit the boundary conditions and the optimisation problem becomes easier as there are no boundary constraints. However, the substitution for higher spatial dimensions gets complicated and the derivatives of the substitution need to be derived. Thus substitution of the form (8) leads to changing the system of PDEs the neural network must solve. This can be seen in our examples. For example 1 in section 5.1, we have a PDE (48) which gets turned into the PDE (51) which have additional terms but the neural network is now unconstrained.

It is worth noting that $\vec{P}$ and $\vec{Q}$ are not unique, multiple functions can be found to satisfy their requirements. We now discuss how to find $\vec{P}$ and $\vec{Q}$ for commonly found boundary conditions.

2.3 Zero Dirichlet Boundary Conditions

When we have a system of N PDEs with zero Dirichlet boundary conditions in n spatial dimensions on domain $\Omega = [0,1]^n$. We can do the substitution

$$\vec{u}(\vec{x}) = \vec{y}(\vec{x};\Theta)\prod_{i=1}^{n} x_i(1-x_i). \tag{9}$$

Where $\vec{y}(\vec{x};\Theta)$ is the output from the neural network in Figure 3. Also, with (9) there are no constraints on the neural network. In this case $\vec{Q}_k(\vec{x}) = \Pi_{i=1}^{n} x_i(1-x_i)$ for $k \in \{1,2,\dots,N\}$ and $\vec{P}(\vec{x}) = \vec{0}$.

2.4 Dirichlet Boundary Conditions In 1D

Now we consider the case when we have arbitrary Dirichlet boundary conditions. In one spatial dimension we can use the substitution

$$\vec{u}(x) = (1-x)\vec{u}(0) + x\vec{u}(1) + x(1-x)\vec{y}(x;\Theta). \tag{10}$$

Therefore, we have $Q(x) = x(1-x)$ and $P(x) = (1-x)\vec{u}(0) + x\vec{u}(1)$.

2.5 Neumann Boundary Conditions In 1D

Now we consider the case when we have Neumann boundary conditions $\frac{\partial u}{\partial x}|_{x=0} = u_x(0)$ and $\frac{\partial u}{\partial x}|_{x=1} = u_x(1)$. In one spatial dimension we can use the substitution

$$\vec{u}(x) = \frac{(\vec{u}_x(1) - \vec{u}_x(0))x^2}{2} + \vec{u}_x(0)x + A + x^2(1-x)^2\vec{y}(x;\Theta), \tag{11}$$

where $A \in \mathbb{R}$. The problem with this method is that the neural network can not contribute to the Dirichlet boundary conditions. This is why we have the constant A in (11), further research is needed to find the out how the neural network can calculate A. The problem comes from the fact that we want $\frac{\partial}{\partial x}(Q(x)y(x)) = y\frac{\partial Q}{\partial x} + Q\frac{\partial y}{\partial x} = 0$ at $x = 0$ and $x = 1$ thus to have no constraints on the neural network we must have $Q_x(0) = Q_x(1)$ and $Q(0) = Q(1) = 0$. However, if we want the neural network to contribute to the Dirichlet boundary conditions then we must have $Q(0) \neq 0$ and $Q(1) \neq 0$. Thus a contradiction is formed.

2.6 Robin Boundary Conditions In 1D

Now we consider the case when we have the boundary conditions

$$\begin{cases} \alpha_0 u(0) + \beta_0 \frac{\partial u}{\partial x}|_{x=0} = 1, \\ \alpha_1 u(1) + \beta_1 \frac{\partial u}{\partial x}|_{x=1} = 1, \end{cases} \tag{12}$$

where we can multiple the equations in (12) to get arbitrary robin conditions. By assuming $P(x)$ takes the form $P(x) = ax + b$ where $a, b \in \mathbb{R}$ and by fitting the boundary conditions we find

$$P(x) = \frac{(\alpha_1 - \alpha_0)x + \beta_0 - (\alpha_1 + \beta_1)}{\beta_0\alpha_1 - \alpha_0(\alpha_1 + \beta_1)}. \tag{13}$$

If $\beta_0\alpha_1 - \alpha_0(\alpha_1 + \beta_1) = 0$ then we find another form for $P(x)$. We get $Q(x) = x^2(1 - x^2)$.

2.7 Dirichlet Boundary Conditions In 2D

We introduce variable $\vec{p} = \vec{p}(x_1, x_2)$. Now we fit the x_1 boundary conditions. This means we get

$$\vec{u}(x_1, x_2) = (1 - x_1)\vec{u}(0, x_2) + x_1\vec{u}(1, x_2) + \vec{p}(x_1, x_2). \tag{14}$$

We want to find $\vec{p}$ such that the boundary conditions

$$\begin{cases} \vec{p}(x_1, 1) & = \vec{u}(x_1, 1) - (1 - x_1)\vec{u}(0, 1) - x_1\vec{u}(1, 1), \\ \vec{p}(x_1, 0) & = \vec{u}(x_1, 0) - (1 - x_1)\vec{u}(0, 0) - x_1\vec{u}(1, 0), \\ \vec{p}(1, x_2) & = 0, \\ \vec{p}(0, x_2) & = 0, \end{cases} \tag{15}$$

are satisfied. Where the values from (15) are calculated by making $\vec{p}$ the subject in equation (14). Then by fitting the x_2 boundary conditions for $\vec{p}$ we get

$$\vec{p}(x_1, x_2) = (1 - x_2)\vec{p}(x_1, 0) + x_2\vec{p}(x_1, 1) + x_1x_2(1 - x_1)(1 - x_2)\vec{y}(x_1, x_2; \Theta). \tag{16}$$

We substitute the values from equation (16) and (15) into (14) to get

$$\begin{aligned} \vec{u}(x_1, x_2) = \ & (1 - x_1)\vec{u}(0, x_2) + x_1\vec{u}(1, x_2) \\ & + (1 - x_2)[\vec{u}(x_1, 1) - (1 - x_1)\vec{u}(0, 1) - x_1\vec{u}(1, 1)] \\ & + x_2[\vec{u}(x_1, 0) - (1 - x_1)\vec{u}(0, 0) - x_1\vec{u}(1, 0)] \\ & + x_1x_2(1 - x_1)(1 - x_2)\vec{y}(x_1, x_2; \Theta). \end{aligned} \tag{17}$$

when implementing (17) in code it could be considered easier to implement the equation formed by substituting $\vec{p}$ from (16) into the equation (14). This gives us

$$
\begin{aligned}
\vec{u}(x_1, x_2) =& (1 - x_1)\vec{u}(0, x_2) + x_1\vec{u}(1, x_2) \\
&+ (1 - x_2)\vec{p}(x_1, 0) + x_2\vec{p}(x_1, 1) \\
&+ x_1x_2(1 - x_1)(1 - x_2)\vec{y}(x_1, x_2; \Theta).
\end{aligned} \tag{18}
$$

2.8 Dirichlet Boundary Conditions In 3D

Now we find a substitution for three spatial dimensions. We introduce $\vec{p} = \vec{p}(x_1, x_2, x_3)$ and $\vec{q} = \vec{q}(x_1, x_2, x_3)$. We first remove the boundary conditions for x_1 which leads to

$$
\vec{u}(x_1, x_2, x_3) = (1 - x_1)\vec{u}(0, x_2, x_3) + x_1\vec{u}(1, x_2, x_3) + \vec{p}(x_1, x_2, x_3). \tag{19}
$$

We now want to find $\vec{p}$ such that the boundary conditions

$$
\begin{cases}
\vec{p}(x_1, x_2, 1) = \vec{u}(x_1, x_2, 1) - (1 - x_1)\vec{u}(0, x_2, 1) - x_1\vec{u}(1, x_2, 1), \\
\vec{p}(x_1, x_2, 0) = \vec{u}(x_1, x_2, 0) - (1 - x_1)\vec{u}(0, x_2, 0) - x_1\vec{u}(1, x_2, 0), \\
\vec{p}(x_1, 1, x_3) = \vec{u}(x_1, 1, x_3) - (1 - x_1)\vec{u}(0, 1, x_3) - x_1\vec{u}(1, 1, x_3), \\
\vec{p}(x_1, 0, x_3) = \vec{u}(x_1, 0, x_3) - (1 - x_1)\vec{u}(0, 0, x_3) - x_1\vec{u}(1, 0, x_3), \\
\vec{p}(0, x_2, x_3) = 0, \\
\vec{p}(1, x_2, x_3) = 0,
\end{cases} \tag{20}
$$

are satisfied. It is worth noting we could use the first part of (17) to fit the x_2 and x_3 boundary conditions. However, we will fit the x_2 boundary conditions for $\vec{p}$. Thus we get

$$
\vec{p}(x_1, x_2, x_3) = (1 - x_2)\vec{p}(x_1, 0, x_3) + x_2\vec{p}(x_1, 1, x_3) + \vec{q}(x_1, x_2, x_3), \tag{21}
$$

with $\vec{q}$ satisfying boundary conditions

$$
\begin{cases}
\vec{q}(x_1, x_2, 1) = \vec{p}(x_1, x_2, 1) - (1 - x_2)\vec{p}(x_1, 0, 1) - x_2\vec{p}(x_1, 1, 1), \\
\vec{q}(x_1, x_2, 0) = \vec{p}(x_1, x_2, 0) - (1 - x_2)\vec{p}(x_1, 0, 0) - x_2\vec{p}(x_1, 1, 0).
\end{cases} \tag{22}
$$

This means we have

$$
\begin{aligned}
\vec{q}(x_1, x_2, x_3) =& (1 - x_3)\vec{q}(x_1, x_2, 0) + x_3\vec{q}(x_1, x_2, 1) \\
&+ x_1x_2x_3(1 - x_1)(1 - x_2)(1 - x_3)\vec{y}(x_1, x_2, x_3; \Theta).
\end{aligned} \tag{23}
$$

And for our code we would use

$$
\begin{aligned}
\vec{u}(x_1, x_2, x_3) = (1 - x_1)\vec{u}(0, x_2, x_3) + x_1\vec{u}(1, x_2, x_3) \\
+ (1 - x_2)\vec{p}(x_1, 0, x_3) + x_2\vec{p}(x_1, 1, x_3) \\
+ (1 - x_3)\vec{q}(x_1, x_2, 0) + x_3\vec{q}(x_1, x_2, 1) \\
+ x_1x_2x_3(1 - x_1)(1 - x_2)(1 - x_3)\vec{y}(x_1, x_2, x_3; \Theta).
\end{aligned} \tag{24}
$$

Additionally, when the problem has mixed boundary conditions instead of Dirichlet boundary conditions, we can use a similar technique. Also, this method can be continued to higher spatial dimensions.

3 Activation Function

We will discuss the sigmoid activation function and calculating its derivatives. We start with the sigmoid function which is

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{25}$$

From this we get $\dot{\sigma} = \sigma(1 - \sigma)$. However, for finding the derivatives of the neural network which are (56), (57), (58) and (59). We need to calculate σ to an arbitrary derivative i.e. calculate $\sigma^{(k)} = \frac{d^k\sigma}{dx^k}$.

3.1 Calculating Derivatives of Sigmoid Function

From $\dot{\sigma} = \sigma(1 - \sigma)$ we get the fact that the derivatives of σ can be written in terms of σ. This means we get

$$\sigma^{(n)} = \sum_{k=1}^{n+1} c_k^{(n)}\sigma^k. \tag{26}$$

From this definition it follows

$$\sigma^{(n)} = \frac{d}{dx}\sigma^{(n-1)} = \frac{d}{dx}\left(\sum_{k=1}^{n} c_k^{(n-1)}\sigma^k\right) = \sum_{k=1}^{n} c_k^{(n-1)}k\sigma^k - \boxed{\sum_{k=1}^{n} c_k^{(n-1)}k\sigma^{k+1}}. \tag{27}$$

For the sum boxed in red in equation (27) we do the substitution $\xi = k + 1$ then we replace ξ with k thus getting the sum boxed in blue in equation (28). This leads to

$$\sigma^{(n)} = \sum_{k=1}^{n} c_k^{(n-1)}k\sigma^k - \boxed{\sum_{k=2}^{n+1} c_{k-1}^{(n-1)}(k - 1)\sigma^k}, \tag{28}$$

$$= c_1^{(n-1)}\sigma^1 - c_n^{(n-1)}n\sigma^{n+1} + \sum_{k=2}^{n}(c_k^{(n-1)}k - c_{k-1}^{(n-1)}(k - 1))\sigma^k. \tag{29}$$

Thus by equating coefficients of σ^k with (26) we get the system of equations

$$\begin{cases} c_1^{(n)} &= c_1^{(n-1)}, \\ c_k^{(n)} &= c_k^{(n-1)}k - c_{k-1}^{(n-1)}(k-1), \;\; k \in \{2,3,\dots,n\}, \\ c_{n+1}^{(n-1)} &= -c_n^{(n-1)}n. \end{cases} \tag{30}$$

Therefore, this is a system of linear equations with $c_1^{(0)} = 1$. Thus we put this into matrix form getting

$$\begin{bmatrix} c_1^{(n)} \\ c_2^{(n)} \\ \vdots \\ \vdots \\ c_{n+1}^{(n)} \end{bmatrix} = \left[\begin{array}{ccccc|c} 1 & & & & & \\ -1 & 2 & & & & \\ & -2 & 3 & & & \\ & & \ddots & \ddots & & \\ & & & & -n & n+1 \end{array}\right] \begin{bmatrix} c_1^{(n-1)} \\ c_2^{(n-1)} \\ \vdots \\ c_n^{(n-1)} \\ 0 \end{bmatrix}. \tag{31}$$

Where $\mathbf{A}_{n+1}$ denotes the $(n+1)\times(n+1)$ matrix in (31). Thus it is clear to see that we get

$$\vec{c}^{(n)} = \mathbf{A}_{n+1}^n e_1, \tag{32}$$

where $e_1 = \begin{bmatrix} 1 & 0 & \dots \end{bmatrix}^T$. This can be further simplified by noting the matrix $\mathbf{A}_n$ has n distinct eigenvalues and eigenvectors. Thus the we can diagonalise $\mathbf{A}_n$. However, in our code we use (32). Our calculation of the n^{th} derivative of the sigmoid function uses the ideas stated in [2].

3.2 Numerical Example for Calculating $\boldsymbol{\sigma^{(3)}}$

We check this method works by calculating $\sigma^{(3)}$. Thus from (32) we get

$$\begin{bmatrix} c_1^{(3)} \\ c_2^{(3)} \\ c_3^{(3)} \\ c_4^{(3)} \end{bmatrix} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & -2 & 3 & 0 \\ 0 & 0 & -3 & 4 \end{array}\right] \begin{bmatrix} c_1^{(2)} \\ c_2^{(2)} \\ c_3^{(2)} \\ 0 \end{bmatrix}, \quad \begin{bmatrix} c_1^{(2)} \\ c_2^{(2)} \\ c_3^{(2)} \\ 0 \end{bmatrix} = \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & -2 & 3 & 0 \\ \hline 0 & 0 & -3 & 4 \end{array}\right] \begin{bmatrix} c_1^{(1)} \\ c_2^{(1)} \\ 0 \\ 0 \end{bmatrix}, \tag{33}$$

$$\begin{bmatrix} c_1^{(1)} \\ c_2^{(1)} \\ 0 \\ 0 \end{bmatrix} = \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ \hline 0 & -2 & 3 & 0 \\ 0 & 0 & -3 & 4 \end{array}\right] \begin{bmatrix} c_1^{(0)} \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} c_1^{(3)} \\ c_2^{(3)} \\ c_3^{(3)} \\ c_4^{(3)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & -2 & 3 & 0 \\ 0 & 0 & -3 & 4 \end{bmatrix}^3 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{34}$$

Therefore we evaluate $\mathbf{A}_4^3 e_1$, leading to

$$\vec{c}^{(3)} = \mathbf{A}_4^3 e_1 = \begin{bmatrix} 1 \\ -7 \\ 12 \\ -6 \end{bmatrix}. \tag{35}$$

This leads us to

$$\sigma^{(3)}(x) = \sigma(x) - 7\sigma^2(x) + 12\sigma^3(x) - 6\sigma^4(x). \tag{36}$$

Which is the same answer we would get if calculated by hand.

4 Optimisation

In this paper, we optimise our neural network by calculating the gradient of the loss function $\mathcal{L}$ with respect to Θ. Then we use stochastic gradient descent to update our Θ. We investigated and implemented other approaches to optimisation including momentum methods and Adam. We got our best results by using stochastic gradient descent. Now we discuss the different methods of optimisation which are discussed at [3].We have a momentum method

$$\begin{cases} m_t &= \beta m_{t-1} + (1-\beta)\frac{\partial}{\partial \Theta}\mathcal{L}(\mathbf{x}, \Theta_{t-1}), \\ \Theta_t &= \Theta_{t-1} - \alpha m_t. \end{cases} \tag{37}$$

Where we have a stochastic method when $\mathbf{x}$ is randomly selected at each epoch cycle to contain points in the domain. And when $\beta = 0$ we get the gradient descent method. Furthermore, [4] has an algorithm for Adam which we tested on our examples but found stochastic gradient descent produced better results. Thus the Adam method is not implemented in our code. However, we do have a secant method approach which is

$$\frac{\partial \mathcal{L}}{\partial \Theta_i} \approx \frac{\mathcal{L}(\mathbf{x}, \Theta + h\vec{e}_i) - \mathcal{L}(\mathbf{x}, \Theta)}{h}, \tag{38}$$

where h is sufficiently small and $\vec{e}_i$ is the zero vector but the i^{th}component is one. This can be proven by using Taylor expansion.

Additionally, when we initialise our neural network the weights and biases are randomly generated from a normal distribution with a mean zero and standard deviation of two.

5 Basic Neural Network

We first consider a basic neural network to solve a PDE in two spatial dimensions. Therefore, we need two neurons for the input layer since the PDE is in two spatial dimensions. We need one output neuron since there is one PDE. We will use two neurons in the hidden layer. The resulting neural network is shown in Figure 1.

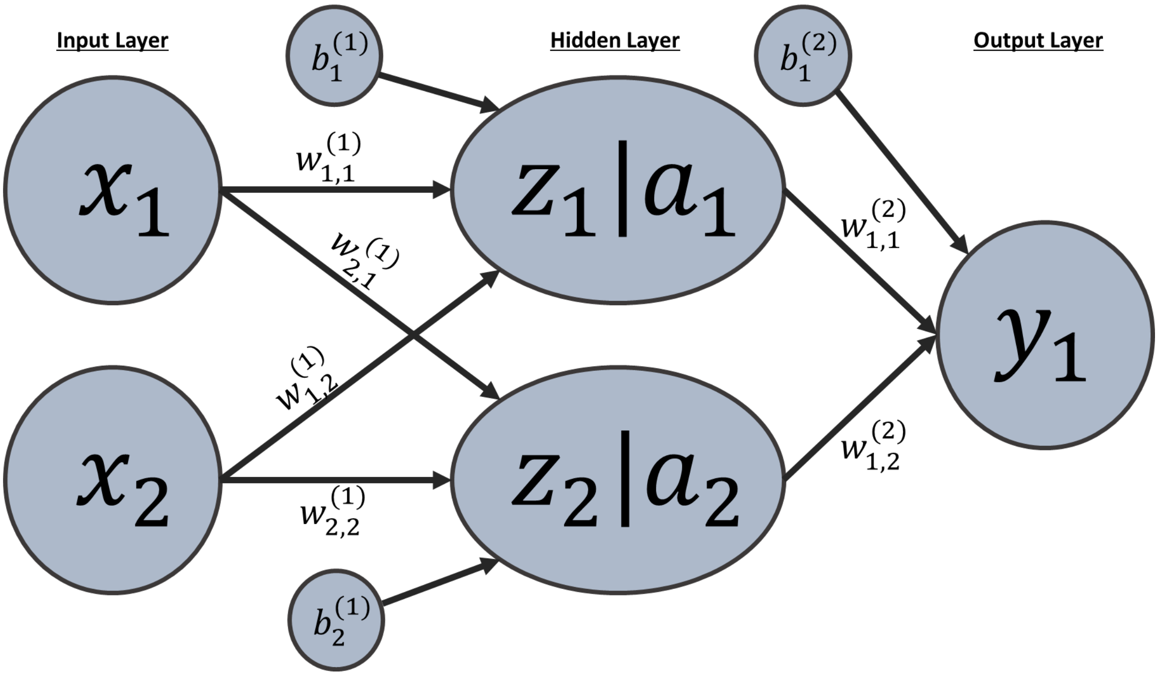


Figure 1: Basic Neural Network, with $a_1 = \sigma(z_1)$ and $a_2 = \sigma(z_2)$.

We now calculate the output y_1 of the neural network in Figure 1. We start by calculating $\vec{z}$

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{1,1}^{(1)}x_1 + w_{1,2}^{(1)}x_2 + b_1^{(1)} \\ w_{2,1}^{(1)}x_1 + w_{2,2}^{(1)}x_2 + b_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}, \tag{39}$$

thus we get $\vec{z} = \mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}$. Now with $a_1 = \sigma(z_1)$ and $a_2 = \sigma(z_2)$ we get

$$y_1 = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + b_1^{(2)} = \mathbf{W}^{(2)}\vec{a} + b_1^{(2)}. \tag{40}$$

For concise notation we use $\sigma^{(k)}(z_i) = \sigma_i^{(k)}$ where $\cdot^{(k)}$ denotes the k^{th} derivative. In the derivative of the loss function (5) we need the spatial derivatives of the neural network, this leads to

$$\frac{\partial y_1}{\partial x_j} = \sigma_1^{(1)} w_{1,1}^{(2)} w_{1,j}^{(1)} + \sigma_2^{(1)} w_{1,2}^{(2)} w_{2,j}^{(1)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix} \begin{bmatrix} \sigma_1^{(1)} & 0 \\ 0 & \sigma_1^{(1)} \end{bmatrix} \begin{bmatrix} w_{1,j}^{(1)} \\ w_{2,j}^{(1)} \end{bmatrix}. \tag{41}$$

And for the 2^{nd} partial derivative we get

$$\frac{\partial^2 y_1}{\partial x_i \partial x_j} = w_{1,1}^{(2)} \sigma_1^{(2)} w_{1,i}^{(1)} w_{1,j}^{(1)} + w_{1,2}^{(2)} \sigma_2^{(2)} w_{2,i}^{(1)} w_{2,j}^{(1)}. \tag{42}$$

From this we can see

$$y_1^{(g)} = \frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_1}} \frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}} y_1 = w_{1,1}^{(2)} \sigma_1^{(\Lambda)} P_1 + w_{1,2}^{(2)} \sigma_2^{(\Lambda)} P_2, \tag{43}$$

with $\Lambda = \lambda_1 + \lambda_2$ and $P_i = w_{i,1}^{(1),\lambda_1} w_{i,2}^{(1),\lambda_2}$. In the neural network shown in Figure 1 we have eight parameters we need to optimise. We denote the group of parameters to optimise by Θ. In this case we have $\Theta = \Theta(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \vec{b}^{(1)}, b_1^{(2)})$. For methods like stochastic gradient descent we need to find $\frac{\partial y_1^{(g)}}{\partial \Theta}$, from (43) we have

$$\frac{\partial y^{(g)}}{\partial w_{1,i}^{(2)}} = \sigma_i^{(\Lambda)} P_i, \tag{44}$$

$$\frac{\partial y^{(g)}}{\partial b_i^{(1)}} = w_{1,i}^{(2)} \sigma_i^{(\Lambda+1)} P_i, \tag{45}$$

$$\frac{\partial y^{(g)}}{\partial b^{(2)}} = \delta_{\Lambda,0}, \tag{46}$$

$$\frac{\partial y^{(g)}}{\partial w_{i,j}^{(1)}} = x_j w_{1,i}^{(2)} \sigma_i^{(\Lambda+1)} P_i + \lambda_j w_{1,i}^{(2)} \sigma_i^{(\Lambda)} \frac{1}{w_{i,j}^{(1)}} P_i, \tag{47}$$

where δ denotes the Kronecker delta.

5.1 Example 1: Solving A Basic PDE

We want to solve the following ODE

$$u_{x_1 x_1} + u_{x_2 x_2} = -2\pi^2 \sin(\pi x_1) \sin(\pi x_2), \tag{48}$$

with zero Dirichlet boundary conditions on domain $\Omega = [0,1] \times [0,1]$ i.e. $u(0, x_2) = u(1, x_2) = u(x_1, 0) = u(x_1, 1) = 0$. It has an analytic solution

$$u(x_1, x_2) = \sin(\pi x_1) \sin(\pi x_2). \tag{49}$$

We use the neural network structure shown in Figure 1 to solve this PDE. We will implement the boundary conditions by using the substitution

$$u = x_1 x_2 (1 - x_1)(1 - x_2) y, \tag{50}$$

where y is the output of the neural network. This means we must have the neural network solve the unconstrained PDE

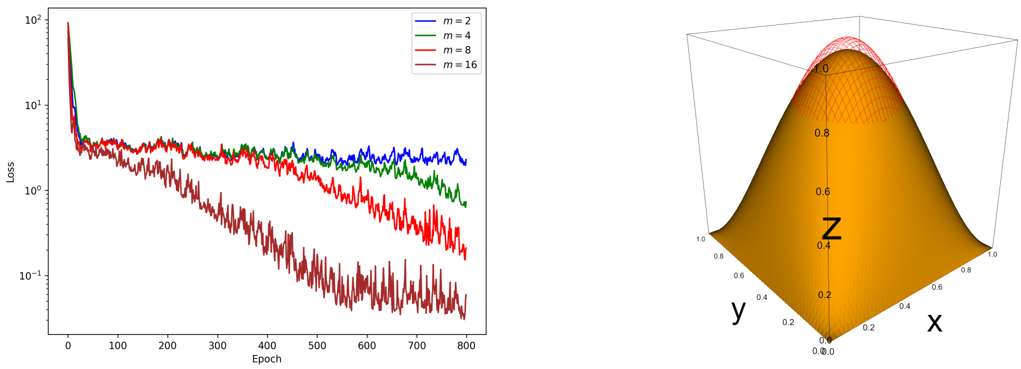
$$\begin{aligned}2\pi^2\sin(\pi x_1)\sin(\pi x_2) = x_1x_2(1-x_1)(1-x_2)(y_{x_1x_1}+y_{x_2x_2})\\ +2x_2(1-x_2)((2x_1-1)y_{x_1}+y)\\ +2x_1(1-x_1)((2x_2-1)y_{x_2}+y).\end{aligned} \tag{51}$$

Therefore, we can implement the solution of the PDE with the code

```
#Example_1.py
from Neural_Network import *; from functools import partial
import numpy as np; np.random.seed(42)
NN1 = Neural_Network_One(2,2,1)
def ODE(x):
    x_1=x[0,0]; x_2=x[1,0]; y=partial(NN1.y_hat_g, x)
    total = x_1*x_2*(1-x_1)*(1-x_2)*(y(lams=[2,0])+y(lams=[0,2]))
    total += 2*x_2*(1-x_2)*((2*x_1-1)*y(lams=[1,0])+y(lams=[0,0]))
    total += 2*x_1*(1-x_1)*((2*x_2-1)*y(lams=[0,1])+y(lams=[0,0]))
    return np.array([total-2*np.pi**2 * np.sin(x_1*np.pi)*np.sin(x_2
   *np.pi)])
def dODE(x):
    x_1=x[0,0]; x_2=x[1,0]; y=partial(NN1.d_y_hat_g, x)
    total = x_1*x_2*(1-x_1)*(1-x_2)*(y(lams=[2,0])+y(lams=[0,2]))
    total += 2*x_2*(1-x_2)*((2*x_1-1)*y(lams=[1,0])+y(lams=[0,0]))
    total += 2*x_1*(1-x_1)*((2*x_2-1)*y(lams=[0,1])+y(lams=[0,0]))
    return np.array([total])
NN1.updates(50,ODE,dODE,N_p=4,alpha=0.1,gamma=0, Secant = True)
NN1.plot(Q=lambda x_1, x_2: x_1*x_2*(1-x_1)*(1-x_2), HTML=True)
print(NN1.Loss(ODE, N_p=4, gamma=0))
```

The last line of "Example_1.py" outputs the loss of approximately 2.61. In Figure 2a we can see having more neurons in the hidden layer leads to a smaller loss value. Additionally, on average the loss decreases when epochs increase, this makes intuitive sense because the longer the model trains the better the model will become. However, when $m = 2$ additional epoch cycles after 50 add little benefit. We hypothesise it is because the neural network does not have enough neurons to capture the dynamics of the system or it has encountered a local minimum.

In Figure 2b we have the orange surface represents the numerical solution of (48). And the red wireframe represents the exact solution (49). From visual inspection, the neural network makes a good approximation of the solution.



(a) Loss decreasing over training cycles for PDE (48) with $\gamma = 0$.

(b) Neural network solution to PDE (48) with $m = 2$. Red wire-frame is exact solution.

Figure 2: Neural Network for Example 1.

6 Neural Network With One Hidden Layer

This will now be extended to a neural network with one hidden layer which has n inputs, m nodes in the hidden layer and N output nodes. Therefore, the neural network will be able to solve a system of N PDEs in n spatial dimensions given there are a sufficient number of neurons in the hidden layer. If m is too small underfitting occurs which means the neural network can not capture all dynamics of the solution. The resulting neural network is shown in Figure 3, for an elegant figure the weights and biases are not shown. From Figure 3 we have

$$\vec{z} = \mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}, \tag{52}$$

$$\vec{a} = \sigma(\vec{z}), \tag{53}$$

$$\vec{y} = \mathbf{W}^{(2)}\vec{a} + \vec{b}^{(2)}, \tag{54}$$

with $\mathbf{W}^{(1)} \in \mathbb{R}^{m\times n}, \mathbf{W}^{(2)} \in \mathbb{R}^{N\times m}, \vec{b}^{(1)} \in \mathbb{R}^m$ and $\vec{b}^{(2)} \in \mathbb{R}^N$. Where (53) means σ is applied element wise therefore $a_i = \sigma(z_i)$. Also, we again use the notation $\frac{\partial^k \sigma(z_i)}{\partial z_i^k} = \sigma_i^{(k)}$. This leads us to the spatial derivatives of the neural network

$$y_\ell^{(g)} = \frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_1}} \frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}} \cdots \frac{\partial^{\lambda_n}}{\partial x_n^{\lambda_n}} y_\ell = \sum_{k=1}^{m} w_{\ell,k}^{(2)} \sigma_k^{(\Lambda)} P_k, \tag{55}$$

with $\Lambda = \sum_{k=1}^n \lambda_k$ and $P_i = \Pi_{k=1}^n w_{i,k}^{\lambda_k}$. And the derivatives of the parameters being

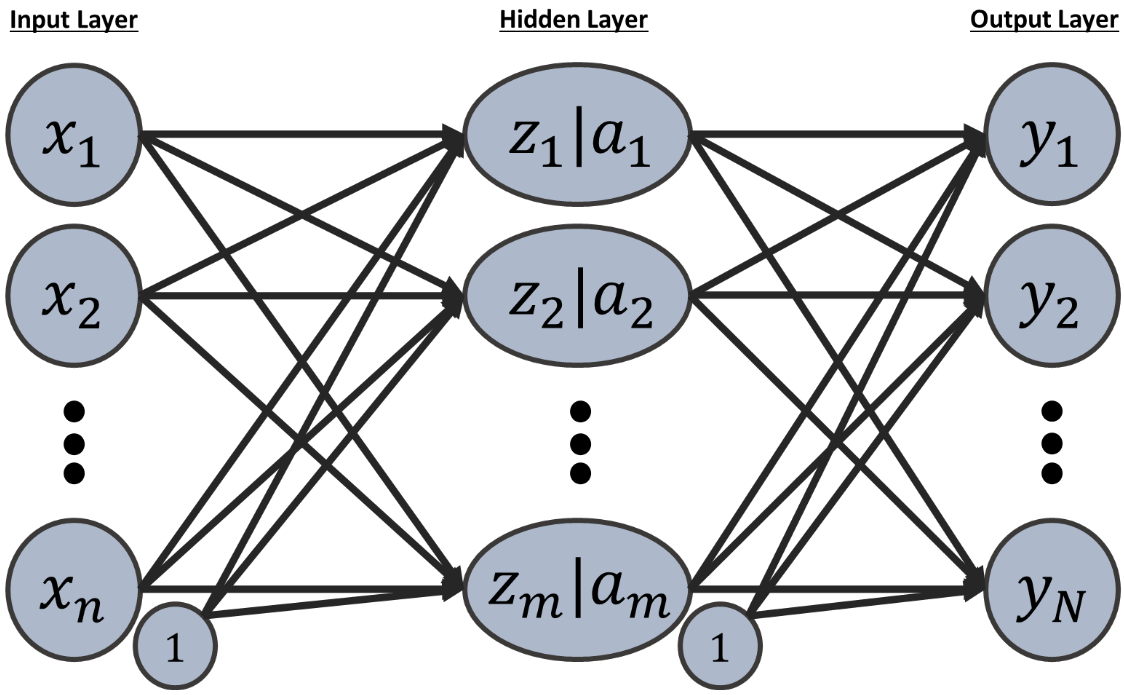


Figure 3: Neural Network, with one hidden layer.

$$\frac{\partial y_\ell^{(g)}}{\partial w_{k,i}^{(2)}} = \sigma_i^{(\Lambda)} P_i \delta_{k,\ell} \ , \tag{56}$$

$$\frac{\partial y_\ell^{(g)}}{\partial b_i^{(1)}} = w_{\ell,i}^{(2)} \sigma_i^{(\Lambda+1)} P_i \ , \tag{57}$$

$$\frac{\partial y_\ell^{(g)}}{\partial w_{i,j}^{(1)}} = x_j w_{\ell,i}^{(2)} \sigma_i^{(\Lambda+1)} P_i + \lambda_j w_{\ell,i}^{(2)} \sigma_i^{(\Lambda)} \frac{1}{w_{i,j}^{(1)}} P_i \ . \tag{58}$$

And for $\vec{b}^{(2)}$

$$\frac{\partial y_\ell^{(g)}}{\partial b_i^{(2)}} = \begin{cases} \delta_{i,\ell}, & \text{when } \Lambda = 0 \\ 0, & \text{otherwise} \end{cases} , \tag{59}$$

where δ denotes Kronecker delta. In our Python implementation we update our Θ for $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \vec{b}^{(1)}$and $\vec{b}^{(2)}$.

The derivation of derivatives takes ideas from [5] and is extended to include multiple output nodes and biases for the output layer denoted by $\vec{b}^{(2)}$.

7 Examples of Numerically Solving ODEs

7.1 Example 2: Simple ODE

We will use our neural network code to solve the following ODE

$$u_{xx} = x, \tag{60}$$

with boundary conditions $u(0) = u(1) = 0$. This has a solution $u(x) = \frac{x}{6}(x^2 - 1)$. We will use the neural network structure shown in Figure 3 to solve this ODE. We will restrict $b_1^{(2)} = 0$ to show the solution can be solved without a bias for the output layer. We implement the boundary conditions using the loss function (4) with $\gamma = 1$.

```python
#Example_2_with_gamma.py
from Neural_Network import *; from functools import partial
import numpy as np; np.random.seed(42)
NN2_g = Neural_Network_One(1,4,1, b_2_Exsist=False); Epochs = 150
def ODE_g(x):
    y_xx = NN2_g.y_hat_g(x, [2]); x_1 = x[0,0]
    return np.array([y_xx-x_1])
def dODE_g(x):
    return np.array([NN2_g.d_y_hat_g(x, [2])])
def BC_g(x):
    y = partial(NN2_g.y_hat_g, lams = [0])
    a=[[0]]; b=[[1]]; y_a=0; y_b=0;
    return np.array([y(a)-y_a, y(b)-y_b])
def dBC_g(x):
    dy = partial(NN2_g.d_y_hat_g, lams = [0])
    a=[[0]]; b=[[1]];
    return np.array([dy(a), dy(b)])
NN2_g.updates(Epochs,ODE_g,dODE_g,BC_g,dBC_g,N_p=6,alpha=0.1,
              gamma=1)
NN2_g.plot(); print(NN2_g.Loss(ODE_g, BC_g, N_p=6, gamma=1))
```

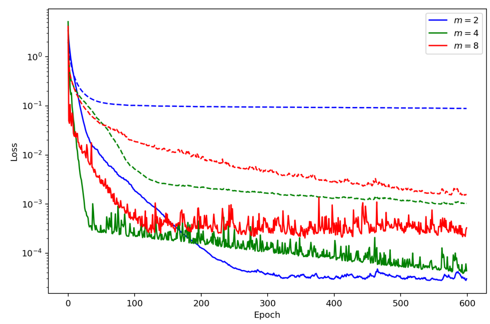
With the last line of code creating the brown coloured line in Figure 5b and outputting the loss 0.00241. We now solve (60) but we implement the boundary conditions by the substitution

$$u = x(1 - x)y, \tag{61}$$

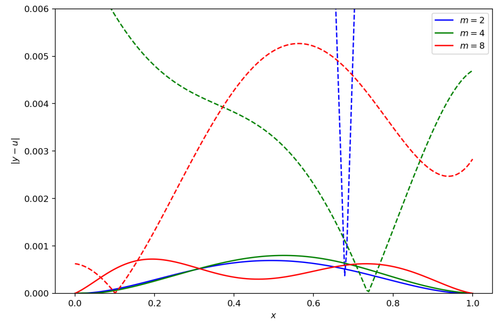
where y is the output of the neural network. Therefore, the neural network must solve the unconstrained ODE

$$x(x - 1)y_{xx} + 2(2x - 1)y_x + 2y + x = 0. \tag{62}$$

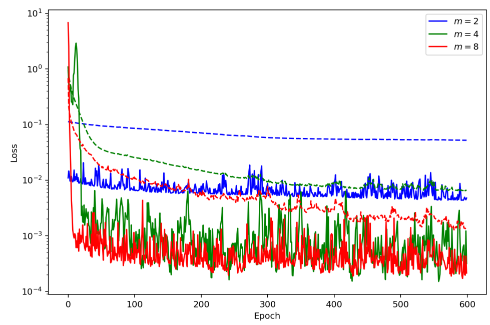
```python
#Example_2.py
from Neural_Network import *; from functools import partial
import numpy as np; np.random.seed(42)
NN2 = Neural_Network_One(1,2,1, b_2_Exsist=False); Epochs = 60
def ODE(vec_x):
    y = partial(NN2.y_hat_g, vec_x); x = vec_x[0,0]
    yc_xx = x*(x-1); yc_x = 2*(2*x-1); yc = 2
    return np.array([yc_xx*y(lams=[2])+yc_x*y(lams=[1])
                     +yc*y(lams=[0])+x])
def dODE(vec_x):
    y = partial(NN2.d_y_hat_g, vec_x); x = vec_x[0,0]
    yc_xx = x*(x-1); yc_x = 2*(2*x-1); yc = 2
    return np.array([yc_xx*y(lams=[2])+yc_x*y(lams=[1])
                     +yc*y(lams=[0])])
NN2.updates(Epochs,ODE,dODE,N_p=6,alpha=0.1,gamma=0)
NN2.plot(Q = lambda x: np.array([x*(1-x)]))
print(NN2.Loss(ODE, N_p=6, gamma=0))
```



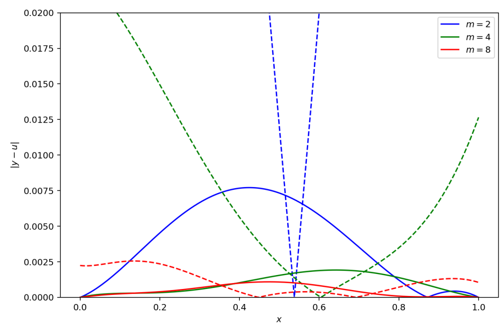
(a) Loss decreasing over training cycles.



(b) Difference plot, shows $|y - u|$.



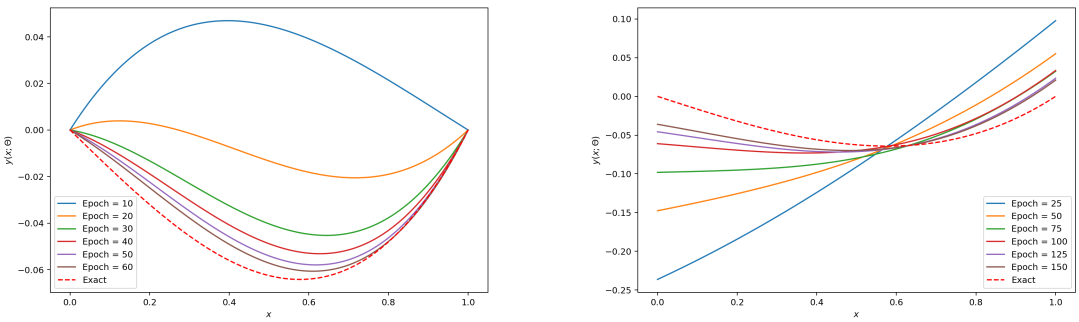
(c) Another simulation of loss decreasing over training cycles.



(d) Difference plot, shows $|y - u|$.

Figure 4: Numerical Error from Training, dashed lines denote training with $\gamma = 1$.

Line 16 in "Example_2.py" creates the brown coloured line in Figure 5a. In Figures 4 and 5 we have visualised the results from the code segments. However, more complex visualisation and analysis of the numerical solution of the Python implementation is left to the reader. Figure 4 visually suggests that applying a substitution to enforce



(a) Boundary conditions enforced by substitution (61), with $m = 2$.

(b) Boundary conditions enforced by $\gamma = 1$, with $m = 4$.

Figure 5: Solution of neural network at different Epochs. With exact solution denoted by a dashed red line.

the boundary constraints is better than using the loss function (4) with $\gamma = 1$ (denoted by dashed lines) to enforce the boundary constraints. Also, it suggests that having a neural network with more neurons in the hidden layer leads to a lower loss function for the same amount of epochs. But each epoch cycle requires more computations. Additionally, in Figure 4a the lowest loss at epoch 800 is $m = 2$ this could be because the random initial conditions for Θ lead to a good neural network.

7.2 Example 3: Coupled ODE

We will use our Python neural network implementation to solve the following coupled ODE

$$\begin{cases} u^{(1)}_{xx} + u^{(2)} + u^{(3)} = 2, \\ u^{(2)}_{xx} + u^{(1)}_{x} = -2x, \\ u^{(1)}_{x} + u^{(2)}_{x} + u^{(3)}_{x} = 0, \end{cases} \qquad (63) \qquad \begin{cases} u^{(1)}(0) = 0, u^{(1)}(1) = 0, \\ u^{(2)}_{x}(0) = 1, u^{(2)}_{x}(1) = 0, \\ u^{(3)}(0) = 1. \end{cases} \qquad (64)$$

From doing some algebra manipulation we come to the analytic solution

$$\begin{cases} u^{(1)}(x) = \frac{1}{2e}(e^x + e^{1-x} - e - 1), \\ u^{(2)}(x) = x(1 - \frac{x^2}{3}) + \frac{1}{2e}(e^{1-x} - e^x + x(1+e)), \\ u^{(3)}(x) = x(\frac{x^2}{3} - 1) - e^{-x} - \frac{x(e+1)}{2e} + 2. \end{cases} \tag{65}$$

We will use the neural network shown in Figure 3 with $n = 1, N = 3$ and $m = \{2, 4, 8\}$. To implement the boundary conditions (64) for the substitution approach we need to find a valid $\vec{Q}$ and $\vec{P}$. Thus we get an unconstrained neural network $\vec{y}$ when

$$\vec{u}(x) = \begin{bmatrix} 0 \\ A + x(1 - \frac{x}{2}) \\ 1 \end{bmatrix} + \begin{bmatrix} x(1-x) \\ x^2(1-x)^2 \\ 1-x \end{bmatrix} \odot \vec{y}. \tag{66}$$

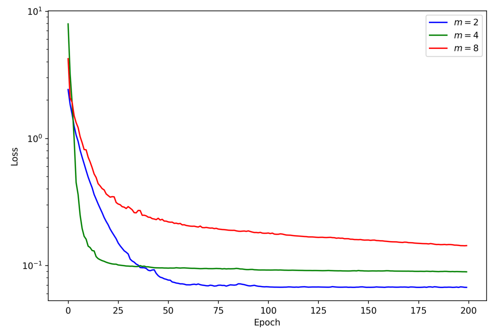
We can see the substitution (66) satisfies the boundary conditions (64). We will take $A \approx 0.31606$, this is acquired from substituting 0 into the exact solution. Further research is required for the neural network to deal with Neumann boundary conditions. Now we will calculate the partial derivatives used in the coupled ODE (63).

$$\begin{cases} u^{(1)}_x &= -x(x-1)y^{(1)}_x - (2x-1)y^{(1)}, \\ u^{(1)}_{xx} &= -x(x-1)y^{(1)}_{xx} - 2(2x-1)y^{(1)}_x - 2y^{(1)}, \\ u^{(2)}_x &= x^2(x-1)^2 y^{(2)}_x + 2x(x-1)(2x-1)y^{(2)} - x + 1, \\ u^{(2)}_{xx} &= x^2(x-1)^2 y^{(2)}_{xx} + 4x(x-1)(2x-1)y^{(2)}_x + (12x^2 - 12x + 2)y^{(2)} - 1, \\ u^{(3)}_x &= xy^{(3)}_x + y^{(3)}. \end{cases} \tag{67}$$

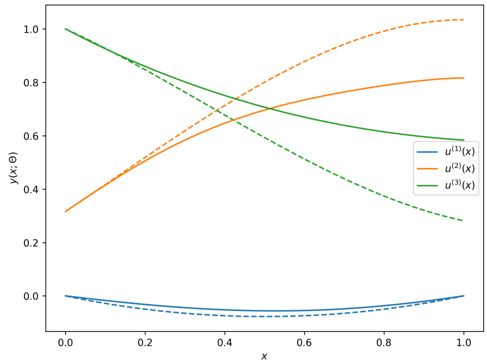
Now we put (67) into the ODE (63) to get the unconstrained ODE the neural network must solve. We implemented the neural network in Python with the code in appendix B.

Figure 6a gives us a contradiction as it states that having fewer neurons in the hidden layer leads to the neural network having a better approximation to the solutions.

In Figure 6b we have the numerical solution to the differential equation at epoch 75 against its analytic solution denoted by a dashed line. From visual inspection, we can see the numerical solution gives an approximation of the exact solution. From Figure 6a further training cycles would give little to no benefit to the numerical solution. However, if we used $m = 8$ we would get a worse solution but after more training cycles it would get better.



(a) Loss decreasing over training cycles for ODE (63) with $\gamma = 0$.



(b) Solution to the system of coupled ODEs (63) with $m = 2$. Dashed line is exact solution.

Figure 6: Neural Network for Example 3.

8 Conclusion

In this paper, we used neural networks to solve systems of PDEs. We discussed different methods for implementing boundary conditions. From our numerical solutions of the example problems solved in this paper, we found the best approach was to use substitution to apply boundary conditions. Also, we derived substitutions for dealing with common boundary constraints. And a formula for calculating the n^{th} derivative of the sigmoid function.

Other findings included that having more neurons in the hidden layer leads to a smaller value for the loss function but the training time was considerably longer. And since this approach has few parameters it requires less storage space than methods like the finite element method.

Finally, multiple examples were covered thus the reader can modify the code to solve a custom system of partial differential equations. In the examples we found, our best results were from an ordinary differential equation. However, for neural networks with more than one input or output training the neural network was slow and it was difficult to decrease the loss function to a satisfactory level in a suitable time frame. Due to the difficulty of optimising a non-convex problem, it strongly suggests that in a commercial setting it is better to use the finite element method.

We discussed the general concepts of the project as a group, therefore all the sections in this paper are my individual extension. Additionally, all the Python code shown in this paper was written by me for this project. But the file "IPV_Show.py" was submitted for another coursework module.

Further research and development would be focused on deriving better optimisation techniques. This could include coding and implementing second-order optimisation algorithms like BFGS. Also, deriving better methods for implementing Neumann boundary conditions when we have $\gamma = 0$.

Additionally, translating into a compiled language like C++ would lead to faster execution. Also, a heterogeneous computing implementation in CUDA (Compute Unified Device Architecture) would take advantage of calculating the derivatives of the loss function with respect to weights and biases in parallel.

References

[1] George V. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314. URL: `https://cognitivemedium.com/magic_paper/assets/Cybenko.pdf`.

[2] Joe Mckenna. *Derivatives of the Sigmoid Function.* Last accessed 10.05.2022. 20.01.2018. URL: `https://joepatmckenna.github.io/calculus/derivative/sigmoid%20function/linear%20albegra/2018/01/20/sigmoid-derivs/`.

[3] Sanket Doshi. *Various Optimization Algorithms For Training Neural Network.* 2019. DOI: `10.48550/ARXIV.1412.6980`. URL: `https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6`.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. DOI: `10.48550/ARXIV.1412.6980`. URL: `https://arxiv.org/abs/1412.6980`.

[5] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: `10.1109/72.712178`. URL: `https://www.researchgate.net/publication/220279855_Artificial_neural_networks_for_solving_ordinary_and_partial_differential_equations`.

[6] Ronald Orozco López. *Product and Quotient rule for Ward's differential calculus.* 2021. DOI: `10.48550/ARXIV.2107.00137`. URL: `https://arxiv.org/abs/2107.00137`.

A Heat Equation

Here we use neural networks to model a heat equation. We discuss an example with one PDE. We train our neural network on the interval $0 \leq t \leq T$ and $0 \leq x \leq 1$.

A.1 Example 4: Heat Equation In One Spatial Dimension

We wish to solve

$$\begin{cases} u_t = u_{xx}, \\ u(t,0) = 0, \\ u(t,1) = \cos(t), \\ u(0,x) = x. \end{cases} \tag{68}$$

Which has analytic solution

$$u(t,x) = x\cos(t) - \sum_{n=1}^{\infty} \frac{2(-1)^n}{\pi n(1+\zeta^2)}[\zeta\sin(t) - \cos(t) + e^{-\zeta t}]\sin(n\pi x). \tag{69}$$

With $\zeta = n^2\pi^2$. Since we are training a neural network we will restrict the time interval to $0 \leq t \leq T = 2$. By using a variant of (14) we get the substitution $u(t,x) = x\cos(t) + x(1-x)ty$ where y is the output of a neural network. This substitution has derivatives

$$\begin{cases} u_t &= tx(1-x)y_t + x(1-x)y - x\sin(t), \\ u_{xx} &= tx(1-x)y_{xx} + 2t(1-2x)y_x - 2ty. \end{cases} \tag{70}$$

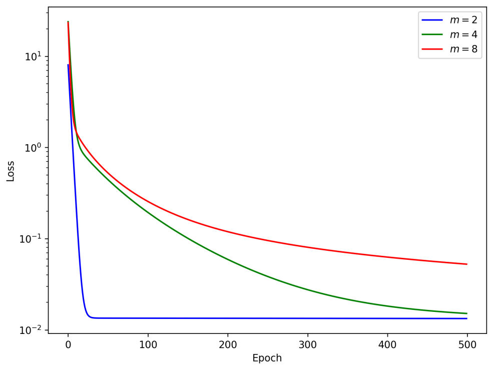
Therefore, we have y must solve the PDE

$$-tx\,(x-1)\,y_t = tx\,(1-x)\,y_{xx} + 2t(1-2x)y_x + (x(x-1) - 2t)y + x\sin(t) \tag{71}$$

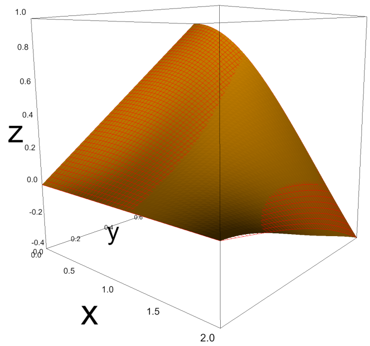
There are no boundary condition constraints because they are enforced in the substitution. This leads us to our code

```python
#Example_4.py
from Neural_Network import *; from functools import partial
import numpy as np; np.random.seed(42)
NN4 = Neural_Network_One(2,8,1); Epochs = 200
def ODE(vec_x):
    #x is [[x_1],[x_2],[x_3], ...]
    y = partial(NN4.y_hat_g, vec_x); D=1.0
    t = vec_x[0,0]; x = vec_x[1,0]
    u_t = t*x*(1-x)*y(lams=[1,0])+x*(1-x)*y(lams=[0,0])-x*np.sin(t)
```

```
    u_xx = t*x*(1-x)*y(lams=[0,2]) + 2*t*(1-2*x)*y(lams=[0,1])
    u_xx += -2*t*y(lams=[0,0]); return np.array([u_t-D*u_xx])
def dODE(vec_x):
    #x is [[x_1],[x_2],[x_3], ...]
    y = partial(NN4.d_y_hat_g, vec_x); D=1.0
    t = vec_x[0,0]; x = vec_x[1,0]
    u_t = t*x*(1-x)*y(lams=[1,0])+x*(1-x)*y(lams=[0,0])
    u_xx = t*x*(1-x)*y(lams=[0,2]) + 2*t*(1-2*x)*y(lams=[0,1])
    u_xx += -2*t*y(lams=[0,0]); return np.array([u_t-D*u_xx])
NN4.updates(Epochs,ODE,dODE,N_p=3,alpha=0.01,gamma=0, Time = 2)
Q = lambda t, x: t*x*(1-x); P = lambda t, x: x*np.cos(t)
NN4.plot(Q=Q,P=P, Time = 2)
print(NN4.Loss(ODE, N_p=3, gamma=0, Time=2))
```



(a) Loss decreasing over training cycles for PDE (68) with $\gamma = 0$.



(b) Neural network solution to PDE (68) with $m = 8$ at Epoch 200. Red wire-frame is exact solution.

Figure 7: Neural Network for Example 4.

In Figure 7a it can be seen that the neural network with $m = 2$ performs the best. This suggests that the solution for PDE (68) is simple enough for it to be approximated by a few neurons in the hidden layer.

In Figure 7b we have the orange surface represents the numerical solution of (68). And the red wireframe represents the exact solution (69). From visual inspection, the neural network makes a good approximation of the solution. Additionally, in this figure we have labels for the axis $X = t, Y = x$ and $Z = y$.

B Code File: Example_3.py

```
#Example_3.py
from Neural_Network import *; from functools import partial
import numpy as np; np.random.seed(42)
NN3 = Neural_Network_One(1,2,3); Epochs = 75
def Diff(x, y_):
    #y_=lambda diff, l: partial(y, [[x]], lams=[diff], l=l)
    u1_x = -x*(x-1)*y_(1,1)-(2*x-1)*y_(0,1)
    u1_xx = -x*(x-1)*y_(2,1)-2*(2*x-1)*y_(1,1)-2*y_(0,1)
    u2_x = x**2*(x-1)**2*y_(1,2)+2*x*(x-1)*(2*x-1)*y_(0,2)
    u2_xx = x**2*(x-1)**2*y_(2,2) + 4*x*(x-1)*(2*x-1)*y_(1,2)
    u2_xx += (12*x**2-12*x+2)*y_(0,2)
    u3_x = (x)*y_(1,3)+y_(0,3)
    return [u1_x,u1_xx,u2_x,u2_xx,u3_x]
def ODE(vec_x):
    y=lambda diff, l: NN3.y_hat_g(vec_x, [diff], l); x=vec_x[0,0];
    u2 = 0.31606027941427883+x*(1-x/2)+x**2*(1-x)**2*y(0,2)
    u3 = 1+(x)*y(0,3); d = Diff(x,y); u1 = x*(1-x)*y(0,1)
    return np.array([d[1]+u2+u3-2,
                     d[3]+d[0]+2*x-1,d[0]+d[2]+d[4]-x+1])
def dODE(vec_x):
    y=lambda diff, l: NN3.d_y_hat_g(vec_x, [diff], l); x=vec_x[0,0];
    u1 = x*(1-x)*y(0,1); u2 = x**2*(1-x)**2*y(0,2)
    u3 = (x)*y(0,3); d = Diff(x,y)
    return np.array([d[1]+u2+u3,d[3]+d[0],d[0]+d[2]+d[4]])
NN3.updates(Epochs,ODE,dODE,N_p=6,alpha=0.1,gamma=0)
Q = lambda x: np.array([x*(1-x),x**2*(1-x)**2,x])
P = lambda x: np.array([0,0.31606027941427883+x*(1-x/2),1])
NN3.plot(Q=Q, P=P); print(NN3.Loss(ODE, N_p=6, gamma=0))
```

C Code File: Neural_Network.py

```
# Neural_Network.py
import numpy as np
import matplotlib.pyplot as plt
from functools import partial
import ipyvolume as ipv


# Custom Modules
from Math_Tools import *
from IPV_Show import IPV_Show_Solution


```

```
class Neural_Network_One:  # One hidden layer
    """
    Class that is used to solve the a system of PDEs using
    neural networks with one hidden layer.

    Attributes
    ----------
    n : int
        The number of inputs to the neural network
    m : int
        The number of neurons in the hidden layer
    N : int
        The number of outputs from the neural network
    b_2_Exsist : bool, optional

    Methods
    -------
    y_hat_g(self, x, lams, l=1)
        calculates the derivative of the neural network
        with respect to the input parameters.
    d_y_hat_g(self, x, lams, l=1)
        calculates the derivative of the neural network
        with respect to the weights and biases.
    Loss(self, ODE, BC=None, N_p=6, gamma=0, Time=0)
        calculates the loss of the neural network
    updates(self, Epochs, ODE, dODE, BC=None, dBC=None,
                N_p=6, alpha=0.1, beta=0.0, gamma=0,
                Secant=False, Time=0)
        Trains the neural network
    plot(self, Time=1, P=0, Q=0, HTML=True)
        Output a basic plot
    """
    def __init__(self, n, m, N, b_2_Exsist = True):
        #b_2_Exsist tells us if b_2 can be non_zero
        self.n = n
        self.m = m
        self.N = N
        self.b_E = b_2_Exsist

        self.W_1 = np.random.normal(0, 2, size=(m, n))
        self.W_2 = np.random.normal(0, 2, size=(N, m))

        self.b_1 = np.random.normal(0, 2, size=(m, 1))
```

```
        self.b_2 = np.zeros((N,1))
        if self.b_E:
            self.b_2 = np.random.normal(0, 2, size=(N, 1))

    def y_hat(self, x):
        # x is [[x_1],[x_2],[x_3], ...]
        z = np.dot(self.W_1, x)+self.b_1
        a = sigma(z)
        return np.dot(self.W_2, a) + self.b_2

    def P_(self, i, lams):
        total = 1
        for k in range(self.n):
            total *= self.W_1[i][k]**lams[k]
        return total

    def y_hat_g(self, x, lams, l=1):
        """
        Parameters
        ----------
        x : numpy.array
            x is [[x_1],[x_2],[x_2],...]
        lams : list
            lams is [lam_1, lam_2, ...]
        l : int, optional
            take the l^th component of the output from the neural net

        Returns
        -------
        the value of the (d^lams y_l)/(dx^lams)
        """
        Lam = np.sum(lams)
        z = np.dot(self.W_1, x)+self.b_1
        total = 0
        for k in range(self.m):
            total += self.W_2[l-1][k]*dsigma(Lam, z[k, 0])*self.P_(k, lams)
        return total + (Lam==0)*self.b_2[l-1,0]

    def d_y_hat_g_W_1_ij(self, x, lams, i, j, l=1):
        l -= 1
        Lam = np.sum(lams)
```

```
        z = np.dot(self.W_1, x)+self.b_1
        part_1 = x[j][0]*self.W_2[l, i]*dsigma(Lam+1, z[i, 0])
        part_2 = lams[j]*self.W_2[l, i]*dsigma(Lam, z[i, 0])/(self.
W_1[i, j])
        return (part_1+part_2)*self.P_(i, lams)

    def d_y_hat_g_W_1(self, x, lams, l=1):
        d_W_1 = np.array([[self.d_y_hat_g_W_1_ij(x, lams, i, j, l=1)
                                for j in range(self.n)] for i in range(self
.m)])
        return d_W_1

    def d_y_hat_g_W_2(self, x, lams, l=1):
        l -= 1
        Lam = np.sum(lams)
        z = np.dot(self.W_1, x)+self.b_1
        d_W_2 = np.array([[(k == l)*dsigma(Lam, z[i, 0])*self.P_(i,
lams)
                                for i in range(self.m)] for k in range(self
.N)])
        return d_W_2

    def d_y_hat_g_b_1(self, x, lams, l=1):
        l -= 1
        Lam = np.sum(lams)
        z = np.dot(self.W_1, x)+self.b_1
        d_b_1 = np.array([[self.W_2[l, i]*dsigma(Lam+1, z[i, 0])
                                * self.P_(i, lams) for i in range(self.m)
]]).T
        return d_b_1

    def d_y_hat_g_b_2(self, x, lams, l=1):
        l -= 1
        Lam = np.sum(lams)
        d_b_2 = np.array([[(i==l)] for i in range(self.N)])*(Lam==0)
        return d_b_2

    def d_y_hat_g(self, x, lams, l=1):
        """
        Parameters
        ----------
        x : numpy.array
            x is [[x_1],[x_2],[x_2],...]
```

```
        lams : list
            lams is [lam_1, lam_2, ...]
        l : int, optional
            take the l^th component of the output from the neural
    net

        Returns
        -------
        the value of the d((d^lams y_l)/(dx^lams))/(dTheta)
        """
        d_W_1 = self.d_y_hat_g_W_1(x, lams, l=l)
        d_W_2 = self.d_y_hat_g_W_2(x, lams, l=l)
        d_b_1 = self.d_y_hat_g_b_1(x, lams, l=l)
        d_b_2 = self.d_y_hat_g_b_2(x, lams, l=l)

        return np.array([d_W_1, d_W_2, d_b_1, d_b_2], dtype=object)

    def Loss(self, ODE, BC=None, N_p=6, gamma=0, Time=0):
        """
        Parameters
        ----------
        ODE : function
            the ODE we are trying to solve
        BC : function
            Contains the boundary constraints
            only needed if gamma != 0
        N_p : int
            Number of points that span dimension.
        gamma : float
            Constant stating how important boundary constraints
            are. If gamma=0, the substition method used.
        Time : float
            The limit of the x_1 axis
            if Time = 0 solving for 0<x_1<1

        Returns
        -------
        None
        """
        total = 0
        a = 0
        b = 1
        x_pos, x_pos_BC = get_x_pos(self.n, a, b, N_p, random=False,
```

```
 Time=Time)
        for x in x_pos:
            total += np.sum(ODE(np.array([x]).T)**2)
        total /= 2*(N_p**self.n)*self.N
        # BC Conditions
        # 1D reduction of x_pos
        if gamma != 0:
            Gamma = gamma/(2*N_p**(self.n-1))
            for x in x_pos_BC:
                BC_now = BC(np.array([x]).T)
                for index in range(len(BC_now)):
                    total += Gamma*BC_now[index]**2
        return total

    # Training Section
    def updates(self, Epochs, ODE, dODE, BC=None, dBC=None,
                N_p=6, alpha=0.1, beta=0.0, gamma=0, Secant=False,
 Time=0):
        """
        Parameters
        ----------
        Epochs : int
            number of training cycles
        ODE : function
            the ODE we are trying to solve
        dODE : function
            the derivative with respect to theta
            of the ODE we are trying to solve
        BC : function
            Contains the boundary constraints
            only needed if gamma != 0
        dBC : function
            Contains the derivative of the boundary
            constraints wrt Theta
            only needed if gamma != 0
        N_p : int
            Number of points that span dimension.
        alpha : float
            Step size of stochastic gradient descent
        beta : float
            Momentum term in momentum gradient descent.
            if beta =0 then we use stochastic gradient descent
        gamma : float
```

```
                Constant stating how important boundary constraints
                are. If gamma=0, the substition method used.
            Secant : bool
                Calculation of gradient via differences
            Time : float
                The limit of the x_1 axis
                if Time = 0 solving for 0<x_1<1

            Returns
            -------
            None
            """
            a = 0
            b = 1
            # Gradient Desent with momentum
            m_t = np.array([np.zeros((self.m, self.n)),
                            np.zeros((self.N, self.m)),
                            np.zeros((self.m, 1)),
                            np.zeros((self.N, 1))], dtype=object)

            for Epoch in range(Epochs):
                x_pos, x_pos_BC = get_x_pos(self.n, a, b, N_p, Time=Time)
                if Secant:
                    m_t = beta*m_t + (1-beta)*self.Secant_Grad(ODE, BC, N_p, gamma)
                else:
                    m_t = beta*m_t + (1-beta)*self.calc_grad(ODE,
                                                             dODE, x_pos, N_p, BC,
                                                             dBC, x_pos_BC, gamma)
                self.W_1 = self.W_1 - alpha*m_t[0]
                self.W_2 = self.W_2 - alpha*m_t[1]
                self.b_1 = self.b_1 - alpha*m_t[2]
                self.b_2 = self.b_2 - alpha*m_t[3]*self.b_E

        def Secant_Grad(self, ODE, BC=None, N_p=6, gamma=0):
            h = 0.00001
            Loss_b = self.Loss(ODE, BC, N_p, gamma)  # Loss before change
            total = np.array([np.zeros((self.m, self.n)),
                              np.zeros((self.N, self.m)),
```

```
                       np.zeros((self.m, 1)),
                       np.zeros((self.N, 1))], dtype=object)

     # W_1
     for i in range(self.m):
         for j in range(self.n):
             self.W_1[i, j] += h
             total[0][i, j] = (self.Loss(ODE, BC, N_p, gamma)-Loss_b)/h
             self.W_1[i, j] -= h

     # W_2
     for i in range(self.N):
         for j in range(self.m):
             self.W_2[i, j] += h
             total[1][i, j] = (self.Loss(ODE, BC, N_p, gamma)-Loss_b)/h
             self.W_2[i, j] -= h

     # b_1
     for j in range(self.m):
         self.b_1[j, 0] += h
         total[2][j, 0] = (self.Loss(ODE, BC, N_p, gamma)-Loss_b)/h
         self.b_1[j, 0] -= h

     # b_2
     for j in range(self.N):
         self.b_2[j, 0] += h
         total[3][j, 0] = (self.Loss(ODE, BC, N_p, gamma)-Loss_b)/h
         self.b_2[j, 0] -= h

     return total

 def calc_grad(self, ODE, dODE, x_pos, N_p,
               BC=None, dBC=None, x_pos_BC=None, gamma=0):
     # W_1, W_2, b_1
     total = np.array([np.zeros((self.m, self.n)),
                       np.zeros((self.N, self.m)),
                       np.zeros((self.m, 1)),
                       np.zeros((self.N, 1))], dtype=object)
     for x in x_pos:
```

```
            ODE_now = ODE(np.array([x]).T)
            dODE_now = dODE(np.array([x]).T)
            for index in range(len(ODE_now)):
                total += ODE_now[index]*dODE_now[index]
        total /= N_p**self.n*self.N

        # 1D reduction of x_pos
        if gamma != 0:
            Gamma = gamma/(N_p**(self.n-1))
            for x in x_pos_BC:
                BC_now = BC(np.array([x]).T)
                dBC_now = dBC(np.array([x]).T)
                for index in range(len(BC_now)):
                    total += Gamma*BC_now[index]*dBC_now[index]
        return total

    # Basic Visualization of Neural Network
    # for one & two spatial dimensions
    def plot(self, Time=1, P=0, Q=0, HTML=True):
        """
        Parameters
        ----------
        Time : float
            The limit of the x_1 axis
        P : function
            Add this to each position in plot
        Q : function
            Multiplies the neural network

        Returns
        -------
        A Plot of the solution
        """
        if P == 0:
            if self.n == 1:
                def P(x): return np.array([0]*self.N)
            if self.n == 2:
                def P(x_1, x_2): return np.array([0]*self.N)
        if Q == 0:
            if self.n == 1:
                def Q(x): return np.array([1]*self.N)
            if self.n == 2:
                def Q(x_1, x_2): return np.array([1]*self.N)
```

```python

        if self.n == 1:
            x_pos = np.linspace(0, 1, 110)
            plt.xlabel(r'$x$')
            plt.ylabel(r'$\vec{y}$')
            for i in range(self.N):
                y_pos = np.array(
                    [P(x)[i]+Q(x)[i]*self.y_hat([[x]])[i] for x in x_pos])
                plt.plot(x_pos, y_pos)
            plt.show()

        if self.n == 2:  # Only for self.N = 1
            a = np.linspace(0, 1, 20)
            X, Y = np.meshgrid(Time*a, a)
            Z = np.array([[P(T_i, X_i)+Q(T_i, X_i)*self.y_hat([[T_i], [X_i]])
                         for T_i in Time*a] for X_i in a])
            ipv.figure()
            ipv.plot_surface(X, Y, Z, color="orange")
            IPV_Show_Solution("Solution", save_HTML=HTML, open_HTML=HTML)
```

D Code File: Math_Tools.py

```python
# Math_Tools.py
import numpy as np
import ipyvolume as ipv


def sigma(z):
    return 1/(1+np.e**(-z))


def dsigma(n, x):
    """nth derivative of sigmoid function"""
    # compute coeffs
    c = np.zeros(n + 1)
    c = Mult(A(n+1), n)[:, 0]
    # compute derivative as series
    res = 0.0
    sig = sigma(x)
    for i in range(n, -1, -1):
```

```
        res = sig * (c[i] + res)
    return res


def A(n):
    return np.diag(np.arange(1, n+1))-np.diag(np.arange(1, n), -1)


def Mult(A, k):
    A_ = A
    for i in range(k-1):
        A_ = np.dot(A_, A)
    return A_


def get_x_pos(n, a, b, N_p, random=True, Time=0):
    # Random option
    x_grid = [np.linspace(a, b, N_p+2)[1:-1]]*n
    x_pos = np.reshape(np.meshgrid(*x_grid), (n, -1)).T

    if random:
        x_pos = np.random.random((N_p**n, n))

    if n > 1:
        x_grid_BC = [np.linspace(a, b, N_p)]*(n-1)
        x_pos_BC = np.reshape(np.meshgrid(*x_grid_BC), (n-1, -1)).T
    else:
        x_pos_BC = [[0]]

    if Time != 0:
        x_time = np.linspace(0, Time, N_p+2)[1:-1]
        x_grid = [np.linspace(a, b, N_p+2)[1:-1]]*(n-1)
        x_pos = np.reshape(np.meshgrid(x_time, *x_grid), (n, -1)).T
    return x_pos, x_pos_BC
```

E Code File: IPV_Show.py

```
#IPV_Show.py
import webbrowser
import ipyvolume as ipv


def IPV_Show_Solution(file_name, save_HTML = True, open_HTML = True,
```

```
    open_offline = True):
    """
    Displays 2D surface plots, by saving them as HTML files and then
    opening,
    or if in juypter notebook displays graphs directly below code.

    Parameters
    ----------
    file_name : str
        The name of the html to be saved, .html not needed added in
   code
    save_HTML : bool, optional
        If true views Ipyvolume plot via html, if false views plot
   below code
        Default is True
    open_HTML : bool, optional
        If true opens HTML after it has been created.
        Default is True
    open_offline : bool, optional
        if True, use local urls for required js/css packages and
   download all
        js/css required packages (if not already available), such
   that the html
        can be viewed with no internet connection. Online version
   doesn't work,
        as can't fetch data.
        Default is True
    Returns
    -------
    None
    """
    if save_HTML:
        ipv.save(file_name + ".html", offline = open_offline)
        if open_HTML:
            print("Opening in default webbrowser")
            webbrowser.open(file_name + ".html")
            print("Opened " + file_name + ".html" + " check default
   browser")
    else:
        ipv.show()
```

F General Leibniz Rule

This is covered because it was meant to help calculate the n^{th} derivative of the sigmoid function. But a better method was found thus this is no longer required.

We start by considering the product rule in higher dimensions this is known as the general Leibniz rule which is stated in [6]. The Leibniz rule is

$$\frac{d^k}{dx^k}(f(x)g(x)) = \sum_{m=0}^{k} \binom{k}{m} f^{(k-m)}(x)g^{(m)}(x). \tag{72}$$

Where $\cdot^{(k)}$ denotes the k^{th} derivative and $\binom{k}{m} = \frac{k!}{m!(k-m)!}$. We discuss a simple way of finding an assumption for the product rule in higher dimension (72). When we calculate the first four derivatives of $f(x)g(x)$ we notice the coefficients are the same as binomial coefficients thus the expansion of $(f(x) + g(x))^k$ gives us the coefficients of (72). This gives us an assumption that the formula for the product rule in higher dimensions is of the form seen in (72).

Now we show by induction that the Leibniz rule (72) is correct. It is trivial to see the general Leibniz rule (72) is correct when for $k = 0$. Assuming (72) is true we now show

$$\frac{d^{k+1}}{dx^{k+1}}(f(x)g(x)) = \sum_{m=0}^{k+1} \binom{k+1}{m} f^{(k+1-m)}(x)g^{(m)}(x). \tag{73}$$

We denote $f = f(x)$ and $g = g(x)$ thus we get

$$\begin{aligned}\frac{d^{k+1}}{dx^{k+1}}(fg) &= \frac{d^k}{dx^k}(f^{(1)}g^{(0)} + f^{(0)}g^{(1)}),\\ &= \sum_{m=0}^{k} \binom{k}{m} f^{(k+1-m)}g^{(m)} + \sum_{m=0}^{k} \binom{k}{m} f^{(k-m)}g^{(m+1)}.\end{aligned} \tag{74}$$

Doing the substitution $\xi = m+1$ on the 2^{nd} sum and then replacing ξ with m we get

$$\begin{aligned}\frac{d^{k+1}}{dx^{k+1}}(fg) &= \sum_{m=0}^{k} \binom{k}{m} f^{(k+1-m)}g^{(m)} + \sum_{m=1}^{k+1} \binom{k}{m-1} f^{(k+1-m)}g^{(m)},\\ &= f^{(k+1)}g^{(0)} + f^{(0)}g^{(k+1)} + \sum_{m=1}^{k} \left[\binom{k}{m-1} + \binom{k}{m}\right] f^{(k-m+1)}g^{(m)}.\end{aligned} \tag{75}$$

By using $\binom{k}{m-1} + \binom{k}{m} = \binom{k+1}{m}$ which is proved in appendix G we get equation (73). Therefore by induction the general Leibniz rule (72) is true for $k \in \mathbb{N}_0$.

G Binomial Coefficient Identity

Here we prove the binomial coefficient identity $\binom{k}{m-1} + \binom{k}{m} = \binom{k+1}{m}$.

$$\begin{aligned}
\binom{k}{m-1} + \binom{k}{m} &= \frac{k!}{(m-1)!(k-m+1)!} + \frac{k!}{m!(k-m)!} \\
&= \frac{k!}{m!(k-m)!}\left(1 + \frac{m}{k-m+1}\right) \\
&= \frac{k!}{m!(k-m)!}\frac{k+1}{k-m+1} \\
&= \frac{(k+1)!}{m!(k+1-m)!} = \binom{k+1}{m}.
\end{aligned} \tag{76}$$