Tel - Aviv, Israel, June 20th, 2021

Multilayer Perceptron for Solving Differential Equations

Authors: Michael (Miki) Hayek-Raz & Amit Sirak, Hemda

Correspondent Author: Mr. Shlomo Rozenfeld, Hemda

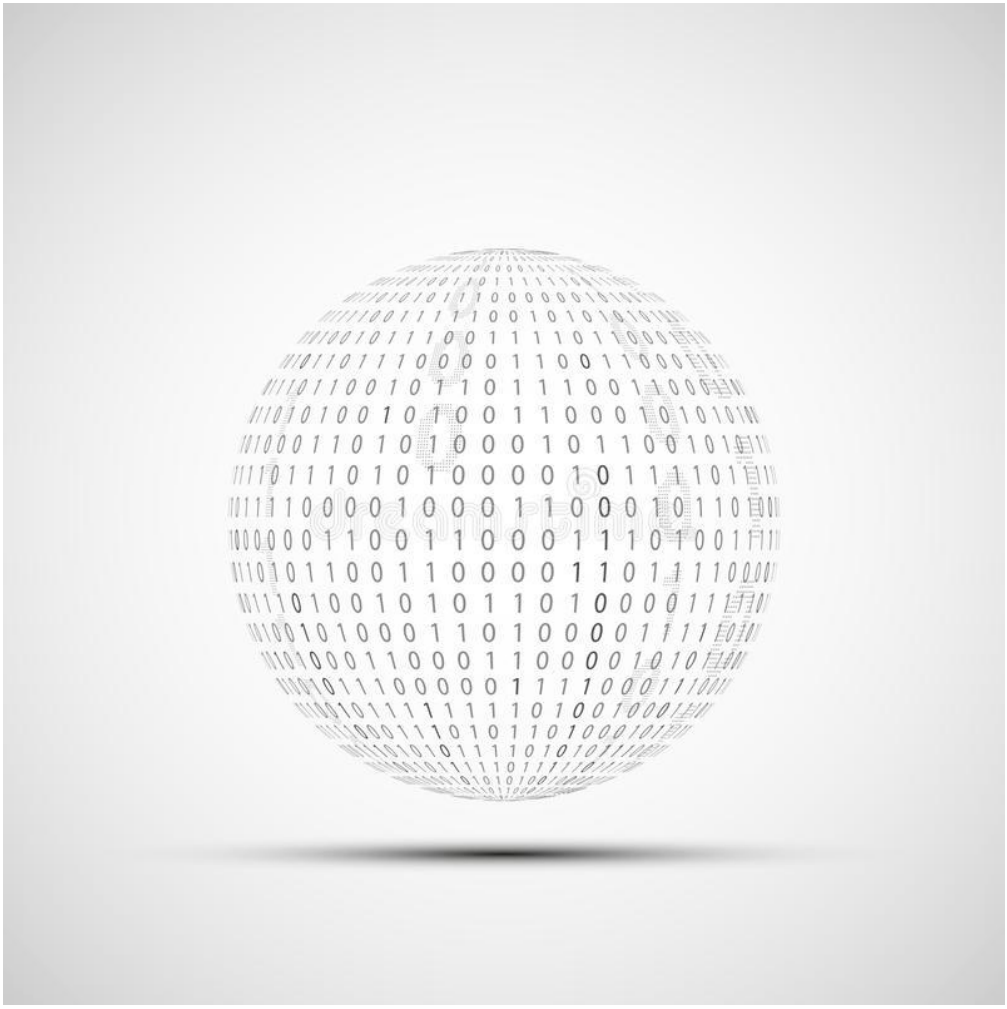


TABLE OF CONTENTS

INTRODUCTION

In this project we will check the validity of using an artificial neural network (ANN) for solving differential equations and compare the solution to a few common numerical methods. We will check multiple different types of differential equations and change the neural network to optimize the solving method and maximize the accuracy of the solution. Changing the neural network will entail changing the number of neurons, the number of layers and the activation function. In addition, we will analyze the relationship between the form of the specific differential equation and the optimal build of the neural network.

THEORETICAL BACKGROUND

1. Differential Equations

In mathematics, a differential equation is an equation that relates one or more functions and their derivatives. In applications, the functions generally represent physical quantities, the derivatives represent their rates of change, and the differential equation defines a relationship between the two. Such relations are common; therefore, differential equations play a prominent role in many disciplines including engineering, physics, economics, and biology. The order of the differential equation is the order of the highest order derivative present in the equation and its degree is the power of the highest order derivative, where the original equation is represented in the form of a polynomial equation in derivatives. Just as biologists have a classification system for life, mathematicians have a classification system for differential equations. We can place all differential equation into two types: ordinary differential equation and partial differential equations. An ordinary differential equation (ODE) is an equation containing an unknown function of one real or complex variable x, its derivatives, and some given functions of x.

In general, ODEs can be written as the following (n is the order of the equation):

$$F\left(x, y, y', \ldots, y^{(n-1)}\right) = y^{(n)}$$

A partial differential equation (PDE) is a differential equation that contains unknown multivariable functions and their partial derivatives. (This contrasts with ordinary differential equations, which deal with functions of a single variable and their derivatives.)

For example (Poisson's equation):

$$\nabla^2 \varphi = f$$

2. Numerical Methods

Numerical methods for differential equations are methods used to find numerical approximations to the solutions of differential equations. In this section of the theoretical background, we will provide an overview of some common numerical methods. For ODEs we will look at the Euler Method and for PDEs we will examine the Finite difference method.

Euler Method

Let f (x) be a real or complex-valued function. The Taylor series of the function, assuming that is infinitely differentiable at a real or complex number a is the following power series:

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 +$$

Let us assume we have a first order ODE of the form:

$$\frac{dy}{dx} = f(x,y)$$

Where y_0 is known and $f(x,y)$ is some function of the variable x and the function y.

Euler's Method assumes our solution is written in the form of a Taylor's Series.

Thus, we have:

$$y(x+h) \approx y(x) + hy'(x) + \frac{h^2 y''(x)}{2!} + \frac{h^3 y'''(x)}{3!} + \frac{h^4 y^{iv}(x)}{4!} +$$

If we only take the first two terms we have:

$$y(x+h) \approx y(x) + hy'(x)$$

Euler's method can be written as the following:

$$y(x+h) \approx y(x) + hf(x,y)$$

Finite Difference Method

The basic idea of the finite difference method is to replace the partial derivatives by approximations obtained by Taylor expansions near the points of interest. Let f (x, t) be a function of several real variables.

for small Δt, using the Taylor expansion at point (x,t) and truncating the Taylor polynomial we get:

$$f(x,t+\Delta t) = f(x,t) + \frac{\partial f(x,t)}{\partial t}\Delta t + O((\Delta t)^2)$$

Solving for $\frac{\partial f(x,t)}{\partial t}$ we get:

$$\frac{\partial f(x,t)}{\partial t} = \frac{f(x,t+\Delta t) - f(x,t)}{\Delta t} - O(\Delta t)$$

This is, not coincidentally, like the definition of derivative, which is given by:

$$\frac{\partial f(x,t)}{\partial t} = \lim_{\Delta t \to 0} \frac{f(x,t+\Delta t) - f(x,t)}{\Delta t}$$

3. Neural Networks

Neural nets are a means of doing machine learning, in which a computer learns to perform some tasks by analyzing training examples. Usually, the examples have been hand-labeled in advance. An object recognition system, for instance, might be fed thousands of labeled images of cars, houses, coffee cups, and so on, and it would find visual patterns in the images that consistently correlate with labels. Modeled loosely on the human brain, a neural net consists of thousands or even millions of simple processing nodes that are densely interconnected. In this section we will describe the basics of neural networks.

The Perceptron

A perceptron is a mathematical function conceived as a model of biological neurons, a neural network. Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs (representing excitatory postsynaptic potentials and inhibitory postsynaptic potentials at neural dendrites) and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing, continuous, differentiable, and bounded.

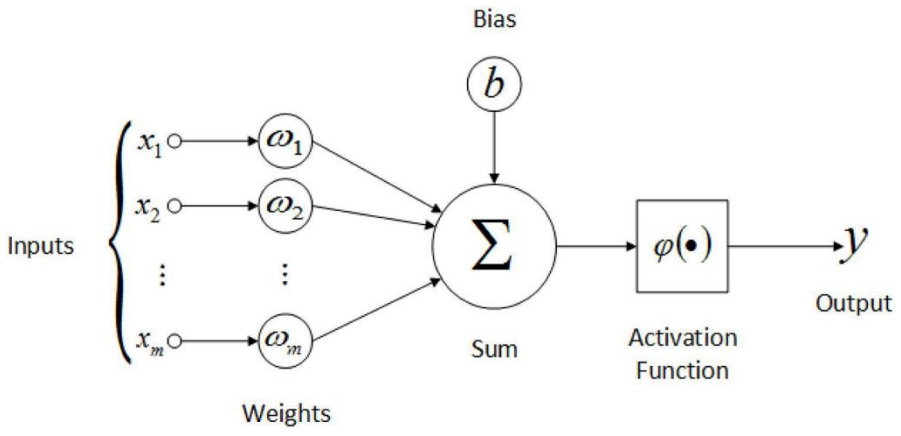


Fig. 1. Perceptron Model

Loss Functions

In mathematical optimization and decision theory, a loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. Loss functions are used in regression when finding a line of best fit by minimizing the overall loss of all the points with the prediction from the line, and they are used while training perceptrons and neural networks by influencing how their weights are updated. The larger the loss is, the larger the update. By minimizing the loss, the model's accuracy is maximized.

Multilayer Perceptron

The multilayer perceptron (MLP) consists of three types of layers—the input layer, output layer and hidden layer. The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP. In a MLP the data flows in the forward direction from input to output layer.Learning occurs in the perceptron by changing connection weights and biases after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning and is carried out through backpropagation. MLPs are designed to approximate any continuous function and can solve problems which are not linearly separable. The major use cases of MLP are pattern classification, recognition, prediction, and approximation.

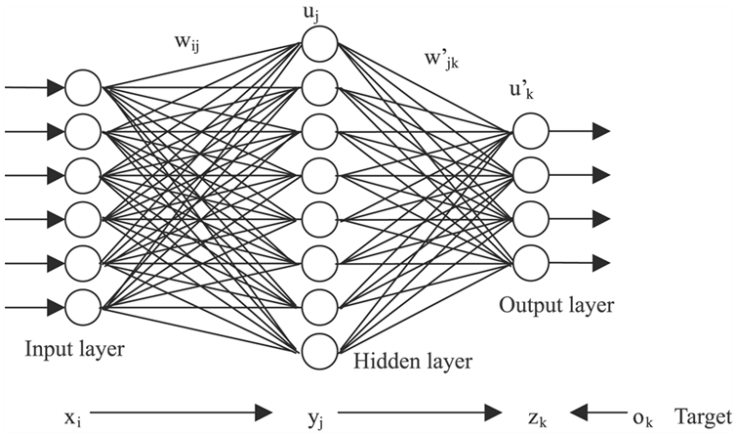


Fig. 2. Multilayer Perceptron Model

THE ALGORITHM

1. Mathematical Model

An algorithm is a set of mathematical instructions or rules that, especially if given to a computer, will help to calculate an answer to a problem. In this case, the problem is how to solve differential equations using a multilayer perceptron.

Cost Function Derivation For ODE

We will initially consider a first order differential equation of the form:

$$\frac{d\psi}{dt} = f(\psi, t)$$

To approximate a solution to this equation, we will first define the relationship between our neural network and the function $\psi(t)$. To do so we shall use The Taylor series of the function:

$$\psi(t) = \psi_0 + \Delta t N(t)$$

Where $N(t)$ is equal to the neural network. Assuming we have the values of ψ_0 (The initial condition) and $f(\psi, t)$, we can now define a cost function. For the sake of simplicity, we will first define delta:

$$\delta_i = \dot{N}(t)\Delta t + N(t) - f(\psi, t)$$

Now we can write the cost function:

$$L = \frac{1}{n}\sqrt{\sum \delta_i^2}$$

Assuming we have a neural network with one hidden layer and its activation function is the ReLU (Rectified Linear Unit) activation function we can write it as the following:

$$N(t) = ReLU(w^0 t + b^0)w^1 + b^1$$

We can now find its derivative:

$$\dot{N}(t) = ReLU(w^0 t + b^0)w^0 w^1$$

We can also approximate its derivative:

$$\dot{N}(t) \approx \frac{N(t + \Delta t) - N(t - \Delta t)}{2\Delta t}$$

Cost Function Derivation For PDE

We will now consider Poisson's equation:

$$\nabla^2\psi = f$$

Let us assume:

$$x \in [0,1]$$

$$y \in [0,1]$$

With Dirichlet BC:

$$\psi(0,y) = f_0(y)$$

$$\psi(1,y) = f_1(y)$$

$$\psi(x,0) = g_0(x)$$

$$\psi(x,1) = g_1(x)$$

To approximate a solution to this equation, we will first define the relationship between our neural network and the function $\psi(x,y)$ while also considering the boundary conditions (A term):

$$\psi_t(x,y) = A(x,y) + x(1-x)y(1-y)N(x,y,\vec{p})$$

Where:

$$A(x,y) = (1-x)f_0(y) + xf_1(y) + (1-y)\{g_0(x) - [(1-x)g_0(0) + xg_0(1)]\} + y\{g_1(x) - [(1-x)g_1(0) + xg_1(1)]\}$$

We will now define the delta:

$$\delta_i = \nabla^2\psi - f$$

Now we can write the cost function:

$$L = \frac{1}{n}\sqrt{\sum \delta_i^2}$$

Gradient Descent

In our algorithm we will use a method called gradient descent to achieve our optimal weights and biases for the neural network. This algorithm is a method of reaching a minimum of a given multi-variable function. It is important to note that the algorithm can only find a local minimum and not the overall minimum of the given function. We use the following equation to do so:

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

Where γ is the step size for each "descent" step (learning rate) and the gradient function is the partial derivative of the function in each direction. We do this repeatedly until a_n stabilizes to a certain value (which in our case will indicate that we reached our optimal weights and biases) or until it reaches a given number of iterations.

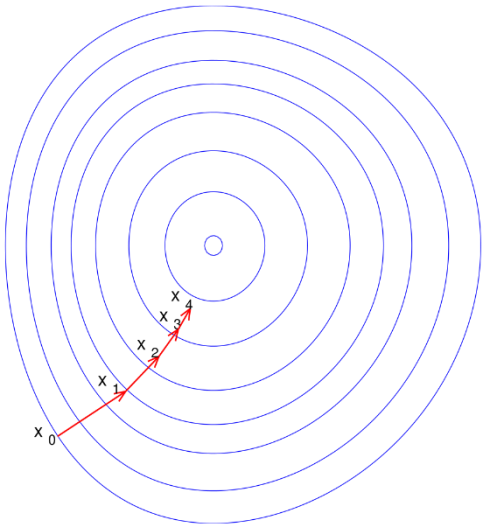
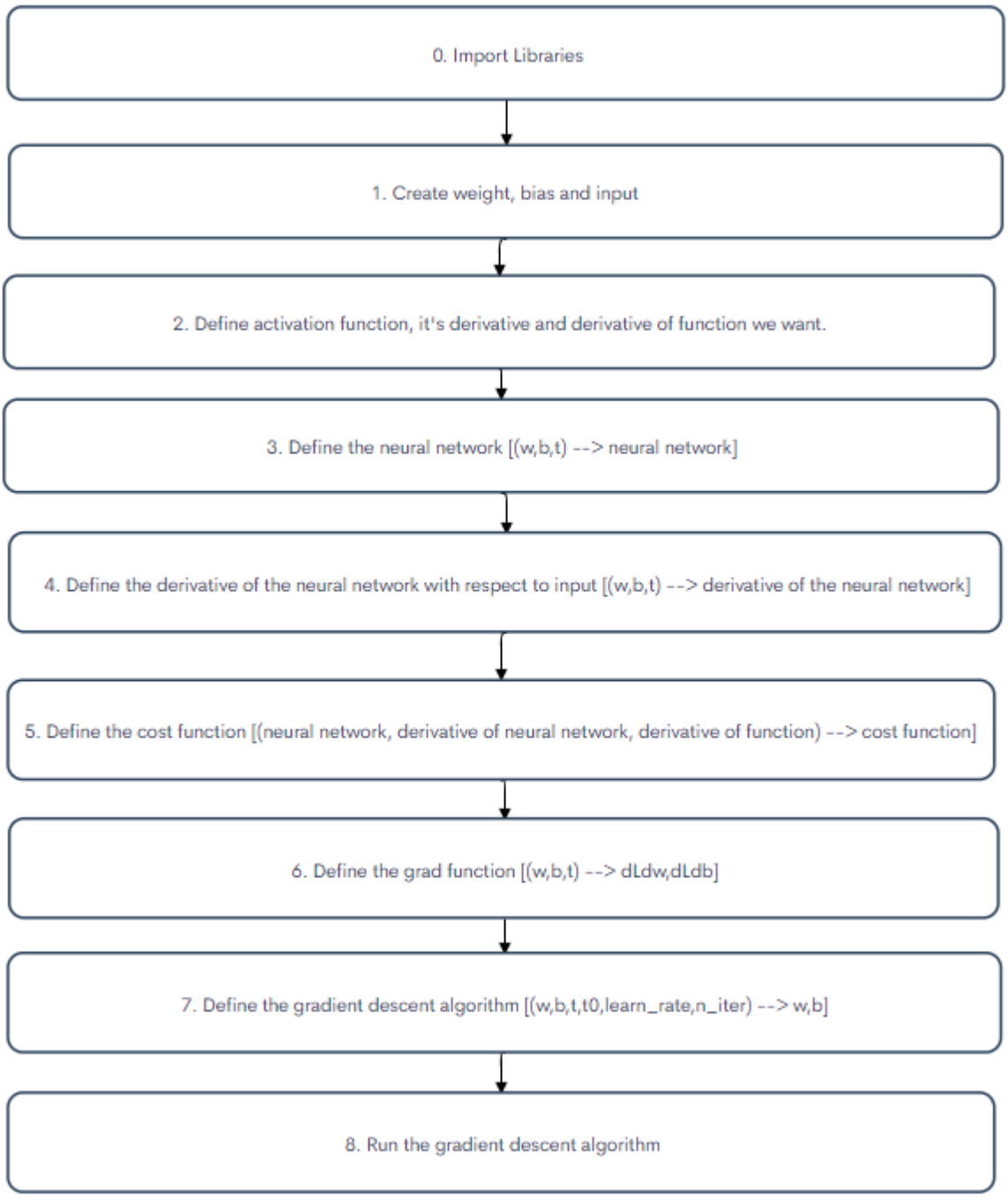


Fig. 3. Gradient Descent Example

In our program we use gradient descent for the optimization of our neural networks weights and biases, therefore, a_n will be the vector of our weights or biases in a certain layer: w^n, b^n.

2. Program Structure



ORDINARY DIFFERENTIAL EQUATIONS

1. ODE's

In this section of the paper, we shall attempt to change parameters of a neural network to optimize the solving method and maximize the accuracy of the solution for ordinary differential equations. We will then compare the results to numerical solutions of the same equations and to analytical results of some of them. All the code used can be found on the GitHub repository in the ODE Jupyter notebook. Unless specified otherwise assume that we have one hidden layer (for the sake of simplicity) and two neurons (percptrons) in the hidden layer.

In addition, assume:

Number of iterations = $I = 10^4$

Learning rate = $\gamma = 10^{-3}$

Number of points in training set = $P_T = 10$

Number of points in output = $P_0 = 10^2$

Number of neurons in hidden layer = $\kappa = 2$

Activation function = $ReLU$

Number Of Neurons In The Hidden Layer

The first parameter we will examine will be the accuracy of the neural network as a function of the number of neurons in the hidden layer.

1. $f(\psi,t) = 2t,\ \psi(0) = 1,\ t \in [0,1]$

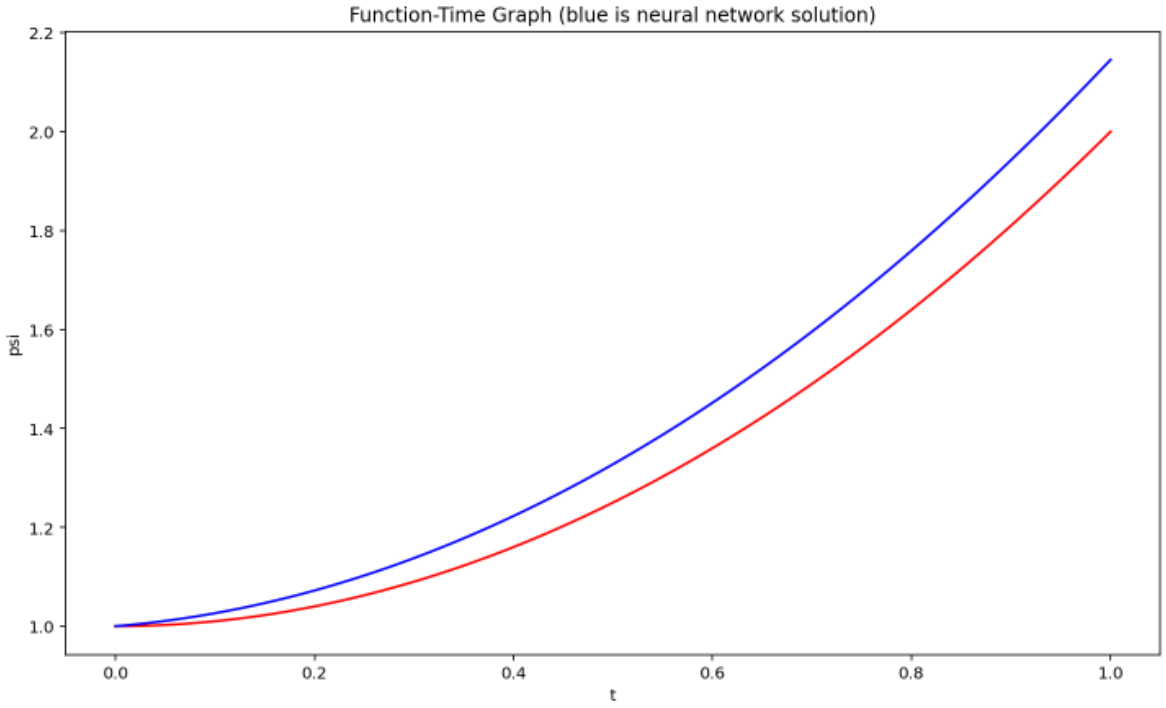


Fig. 4. Function-Time Graph Output

The following is a discussion of the results gained from an example run. As we can see the error is not at all small towards the end of the graph, but the shape of the blue curve (neural net solution) seems to be consistent.

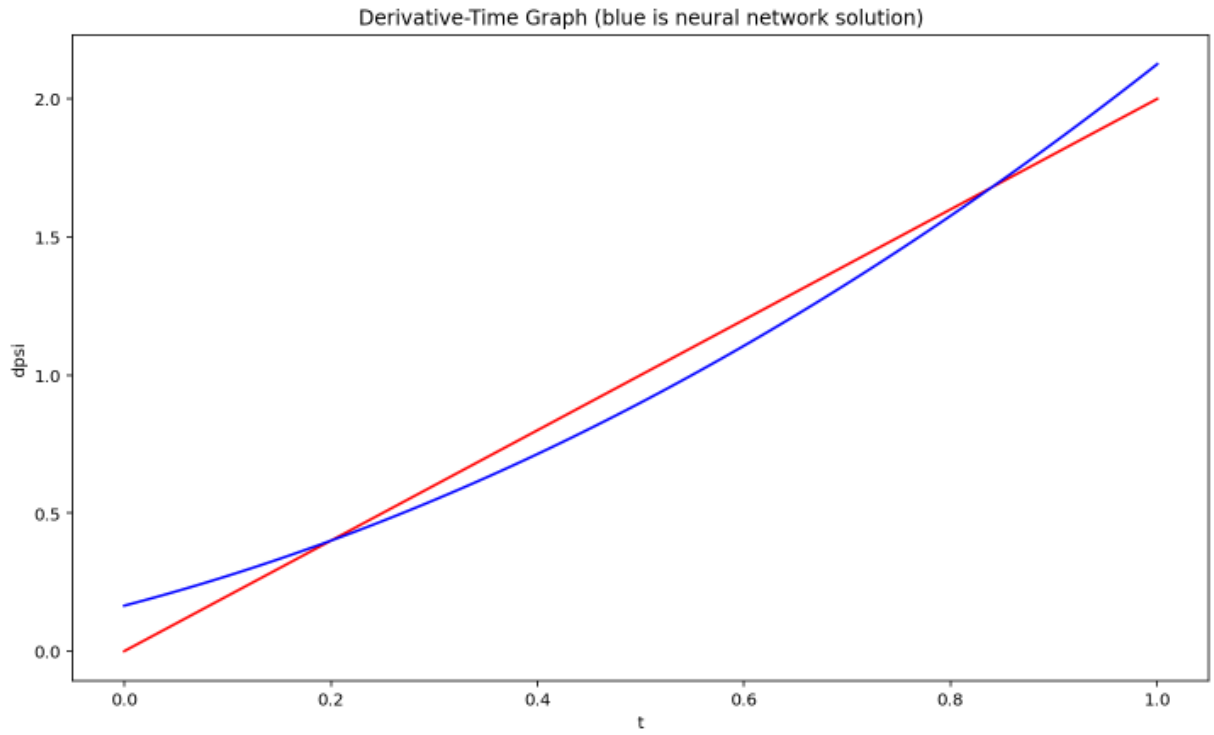
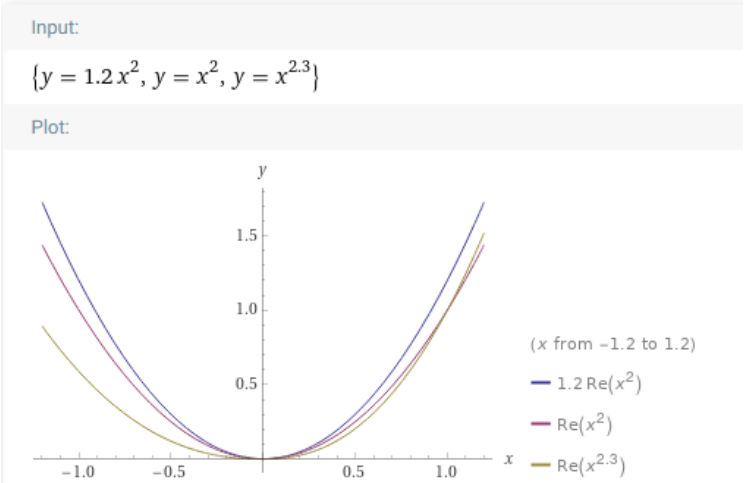


Fig. 5. Derivative-Time Graph Output

The neural net solution derivative is not a straight line as it should be, but rather it is slightly curved. It seems as if the neural network is generating an equation (fig 4) which is of the sort x to the power of y. Where y is a number that is slightly larger than 2. If y were equal to something less than 2 and greater than 1, the neural networks solution derivative would curve in the opposite direction (it would curve in concave fashion). Another possibility is that the neural network solution was multiplied by some factor. The graphs described can be viewed in the following plot from wolfram alpha:



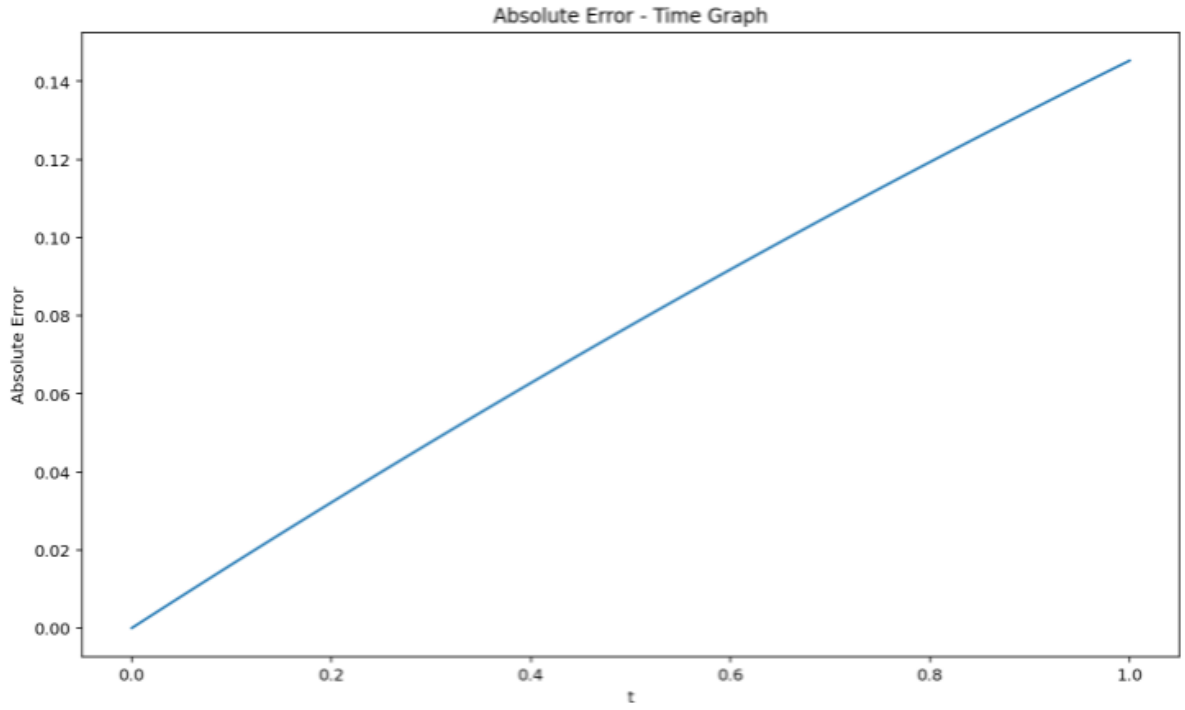


Fig. 6. Absolute Error - Time Graph

During this run we got an L value of approximately 0.281 and an MSE of approximately 0.00755.

The graphs presented in the next page will be from data collected from five runs with a different number of neurons in each of them.

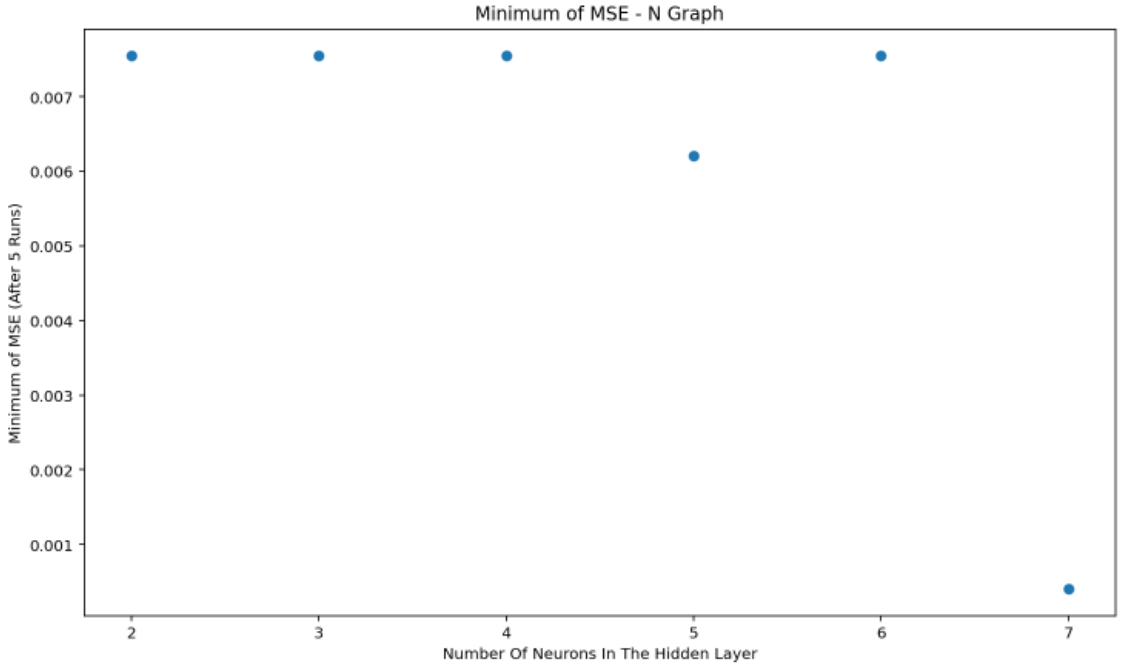


Fig. 7. MSE Min - N Graph

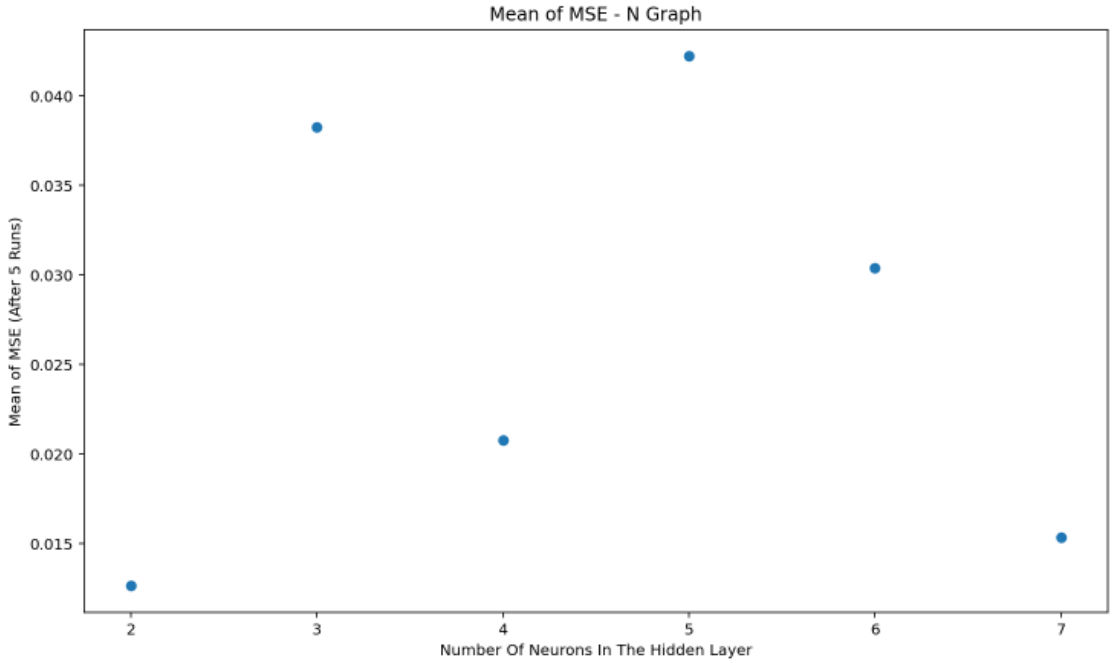


Fig. 8. MSE Mean - N Graph

The results seem to indicate that an increase in the number of neurons can possibly give a significant increase is the accuracy of the model. From the minimum of mean squared error graph, we can see that an increase in the number of neurons is correlated with a lower minimum mean squared error. However, from the mean of mean squared error graph we can see that after five runs, the mean of the mean squared error seems to become more erratic as the number of neurons grows. This is probably a result of an increase in the number of weights and biases that need to be adjusted. If this is indeed the case, to get better accuracy as a result of an increase in the number of neurons, it is imperative to also increase the number of iterations. As such, like most problems we wish to solve with a neural network, when deciding on the architecture of the neural network a balance must be found between the accuracy of the result and training run time.

Activation Function Of Hidden Layer

We shall now study what effect the activation function of the hidden layer has on the solution of the neural network.

1. $f(\psi,t)=\frac{1-\psi \sin t}{\cos t},\ \psi(0)=1,\ t\in[0,1.6]$

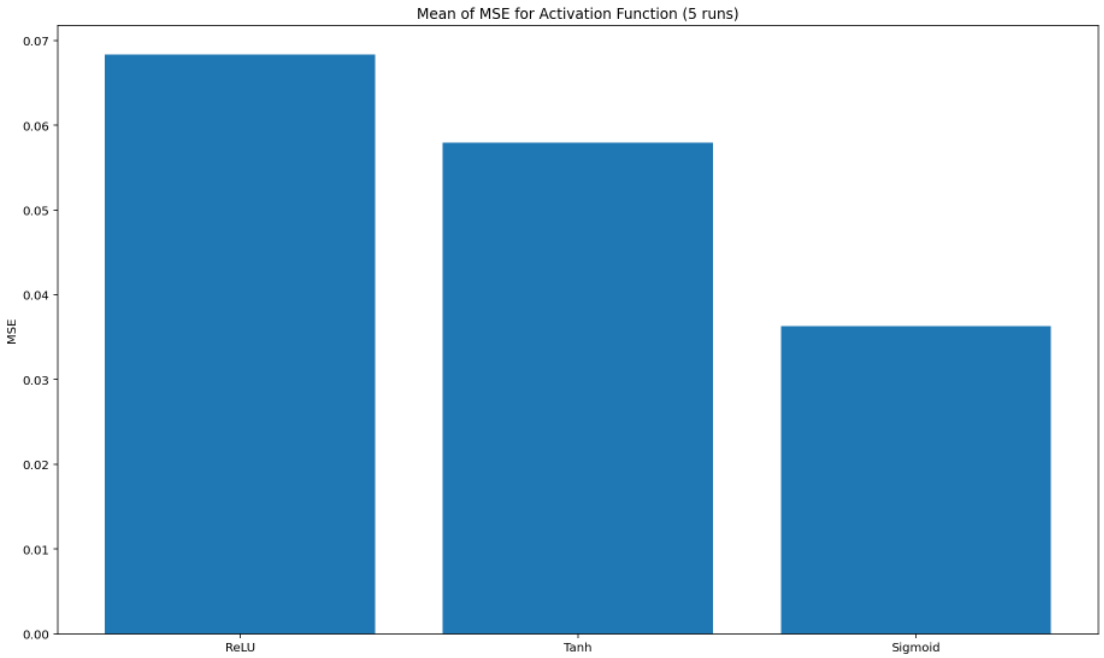


Fig. 9. MSE Mean - Activation Plot

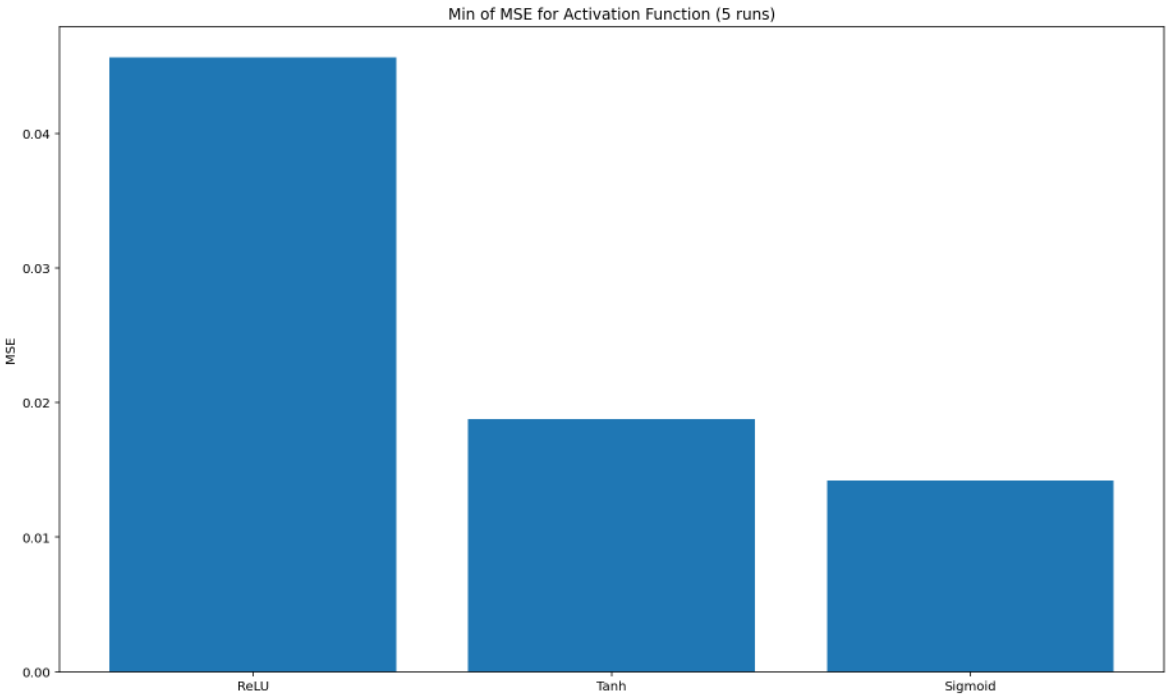


Fig. 10. MSE Min - Activation Plot

For this differential equation, we found that the most effective activation function is the sigmoid, with the ReLU activation function being much worse than the rest.

2. $f(\psi,t) = 9.8 - 0.196\psi,\ \psi(0) = 48,\ t \in [0,1]$

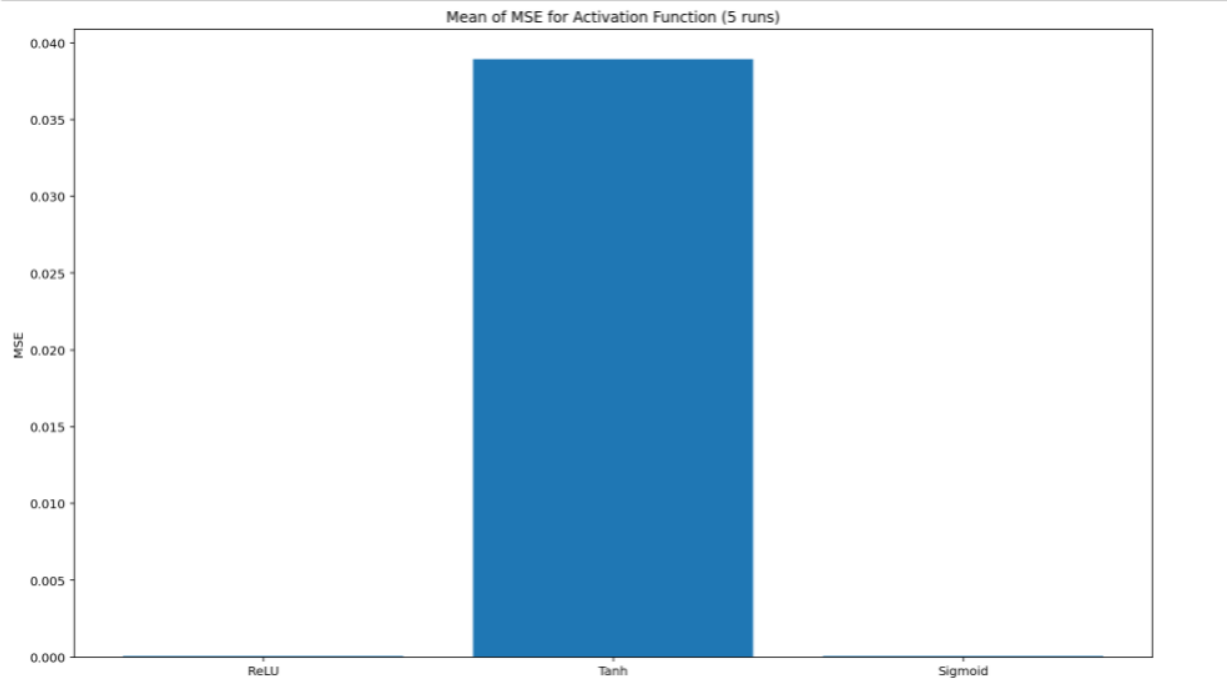


Fig. 11. MSE Mean - Activation Plot 2

In this plot (figure 11) the mean of the MSE for the sigmoid function is larger than the mean of the MSE for the ReLU function.

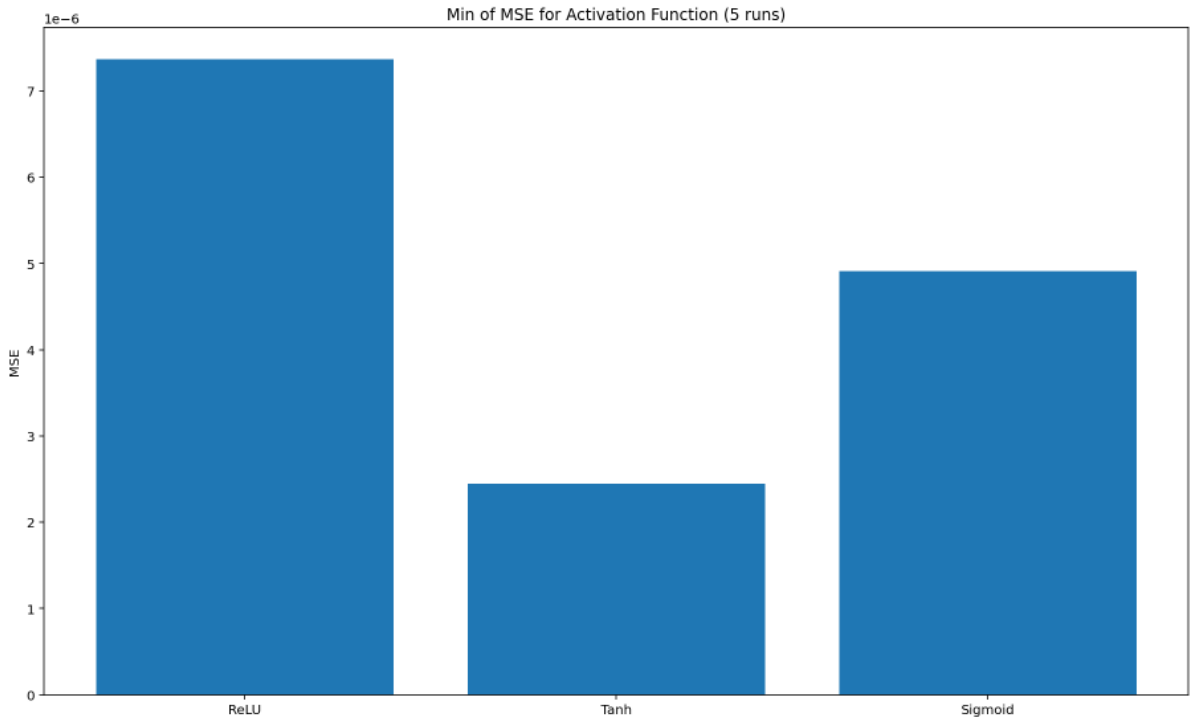


Fig. 12. MSE Min - Activation Plot

For the second differential equation, we get some remarkably interesting behavior. The smallest minimum value for the MSE came when using the tanh activation function, it's mean of the MSE however was significantly larger than the other activation functions. From this we can conclude that the tanh activation function has the most variability out of the three. While the ReLU function has the largest minimum value for the MSE, it's mean of the MSE is the smallest. This would indicate that the ReLU activation function has the least amount of variability and outputs consistent results. The sigmoid function seems to find a good balance between the min of the MSE and the mean of the MSE.

Run Time As Number Of Points Increases

1. $f(\psi, t) = 2t,\ \psi(0) = 1,\ t \in [0,1]$

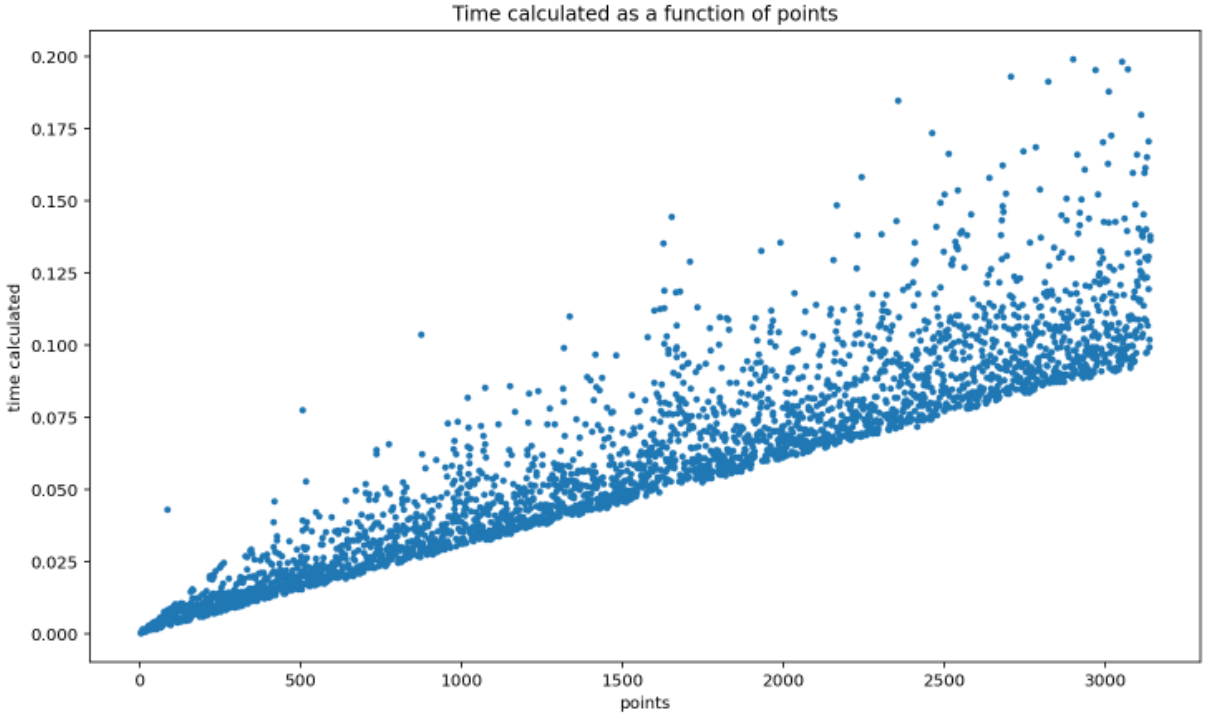


Fig. 13. Time Calculated – Number of Points Plot

2. $f(\psi,t) = 9.8 - 0.196\psi,\ \psi(0) = 48,\ t \in [0,1]$

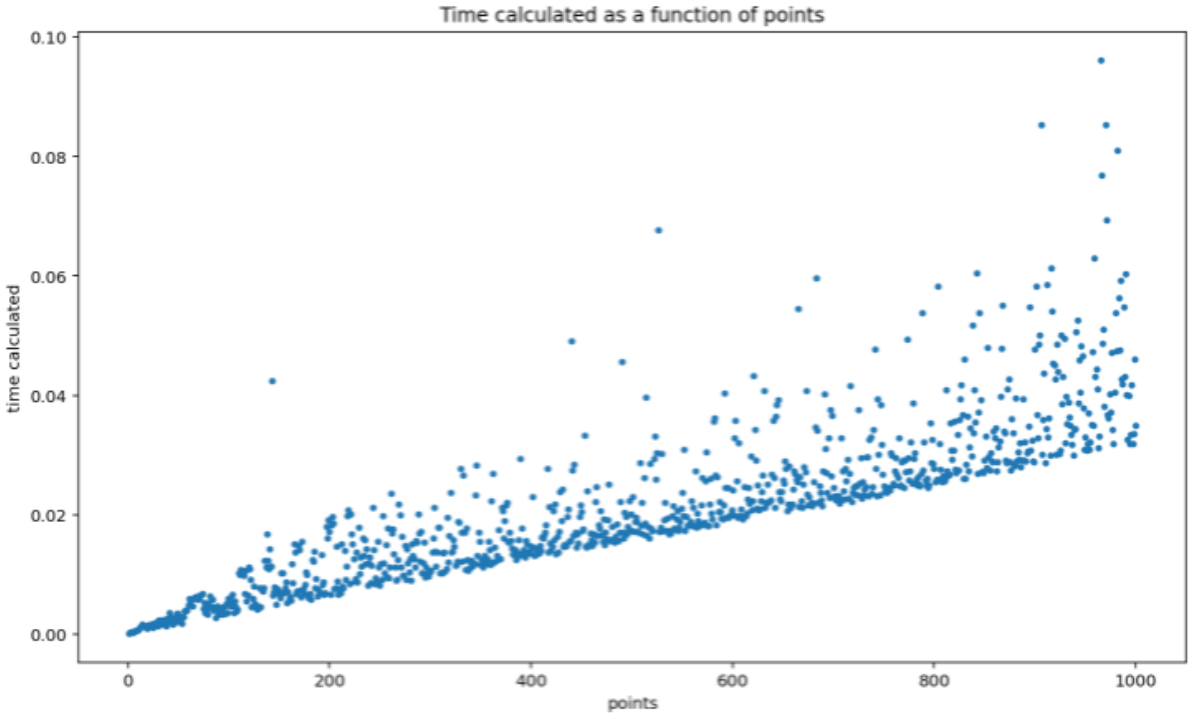


Fig. 14. Time Calculated – Number of Points Plot

The run time is measured in seconds. From Fig 13 and 14 we see that the time increases in a linear way, however there is much variation in the execution time. This probably stems from daemons, the OS-scheduler, changes in the CPU clock rate etc. It is important to note that as we add more points the variation seems to be more chaotic while at the start it looks as if the variation is confined in between two linear lines.

MSE As Number Of Points Increases

1. $f(\psi,t) = 2t,\ \psi(0) = 1,\ t \in [0,1]$

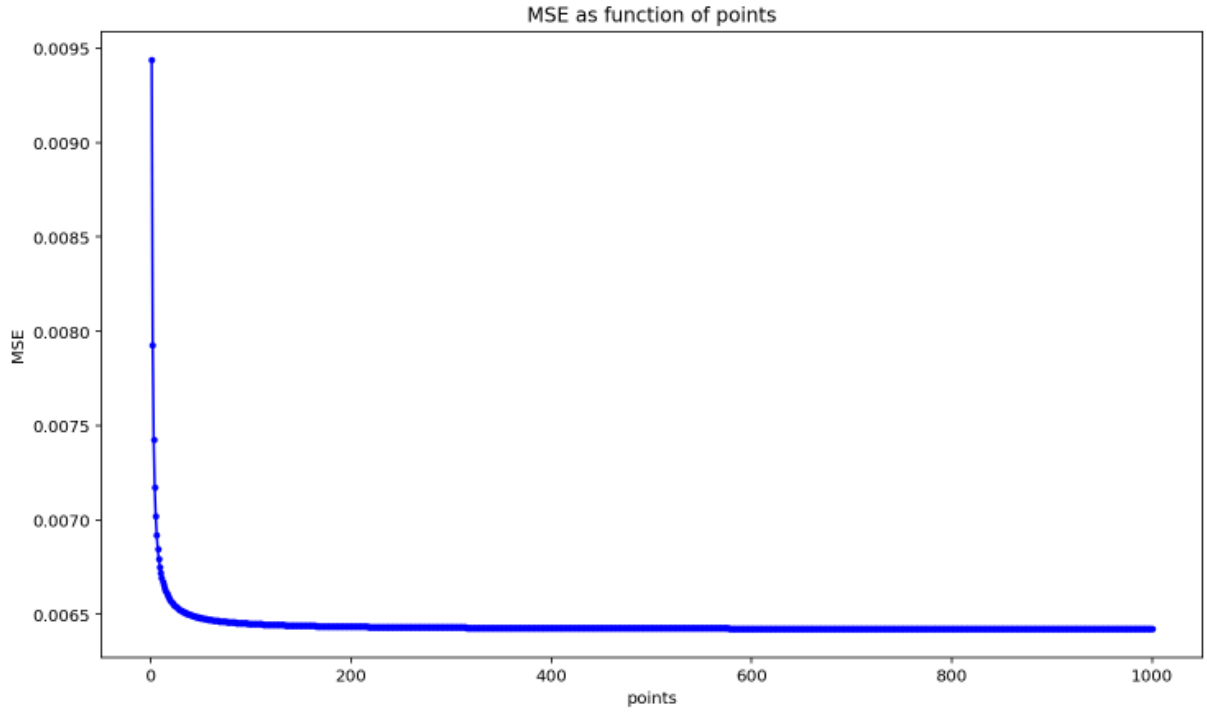


Fig. 15. MSE – Number of Points Plot

2. $f(\psi, t) = 9.8 - 0.196\psi,\ \psi(0) = 48,\ t \in [0,1]$

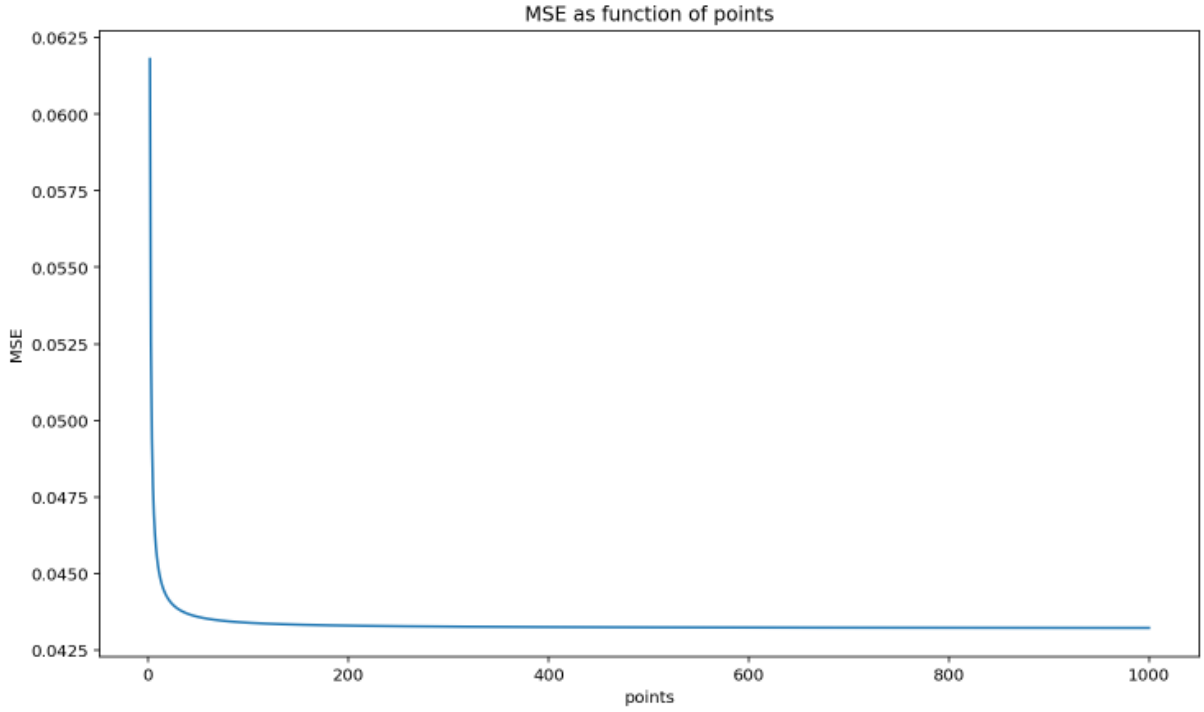


Fig. 16. MSE – Number of Points Plot

The graphs show that the mean squared error rapidly stabilizes to a constant value as the number of points increases.

Comparison With Numerical Solution (Euler Method)

1. $f(\psi,t) = 2t,\ \psi(0) = 1,\ t \in [0,1]$

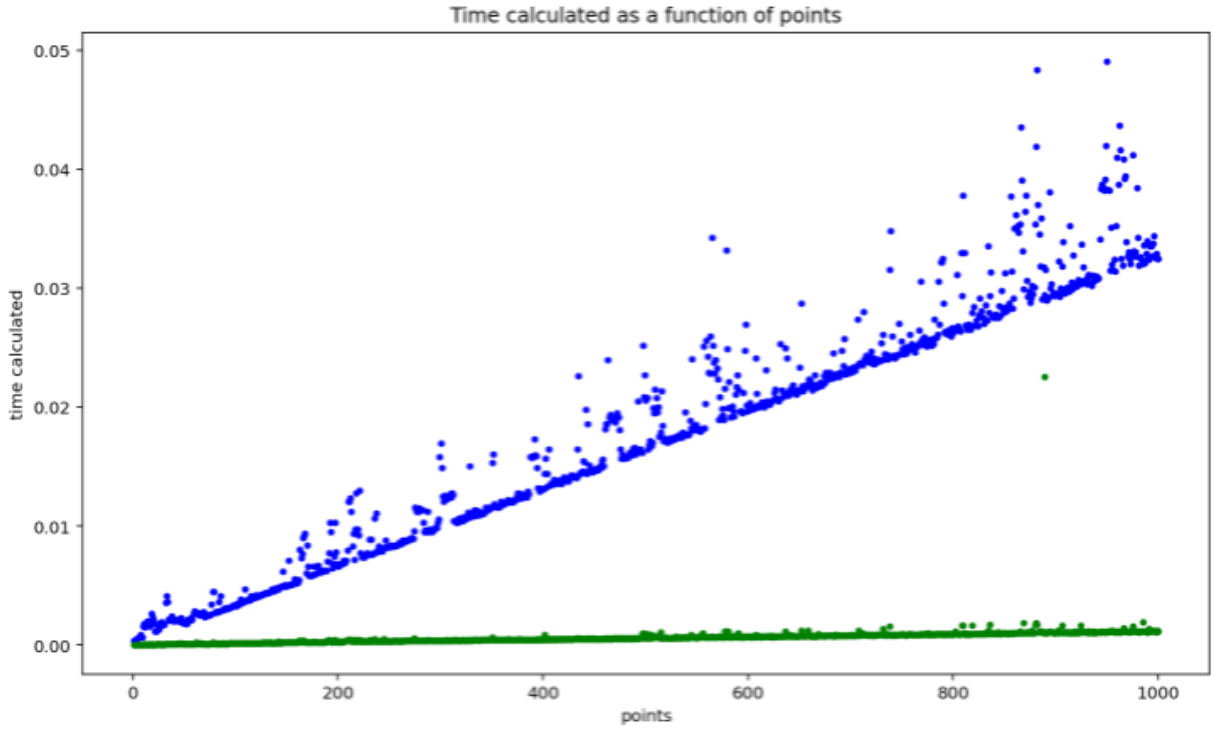


Fig. 17. Comparison of Run-Time

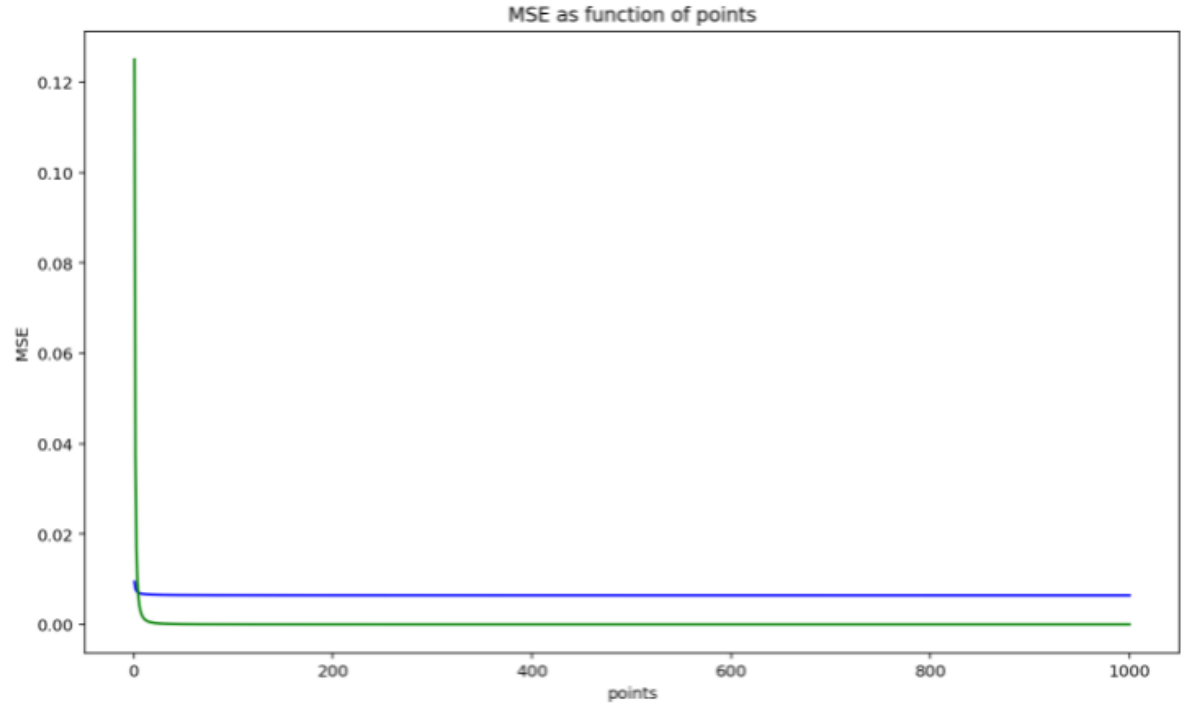


Fig. 18. Comparison of MSE

2. $f(\psi, t) = 9.8 - 0.196\psi,\ \psi(0) = 48,\ t \in [0,1]$

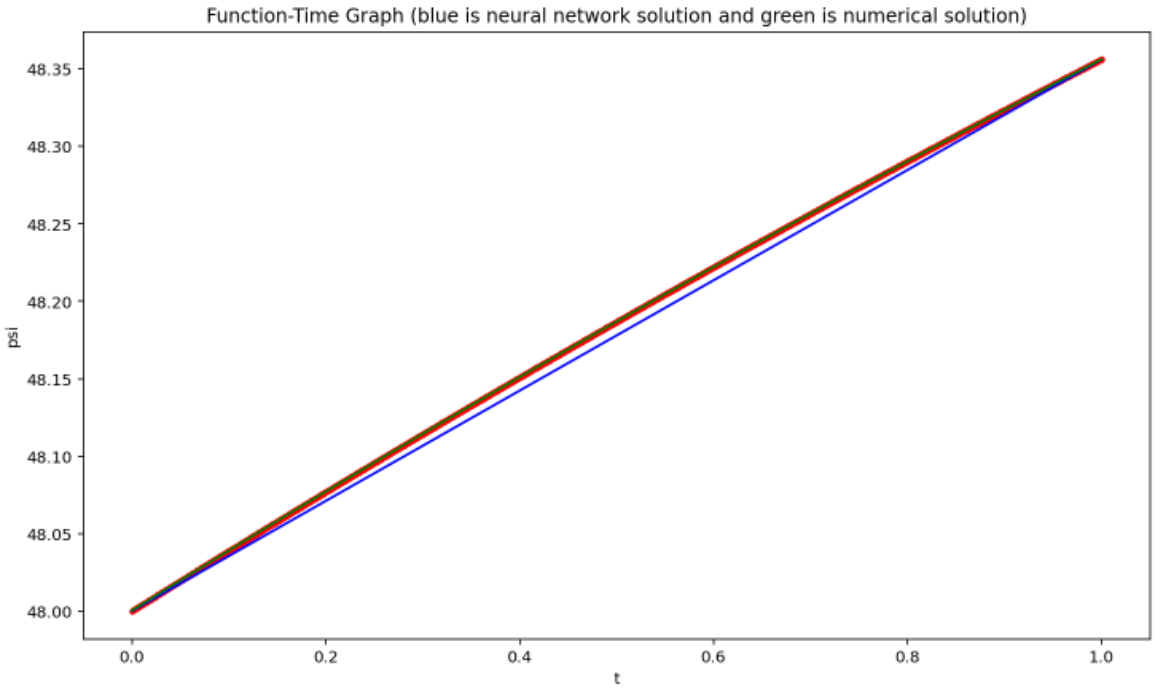


Fig. 19. Example Output For 1000 points

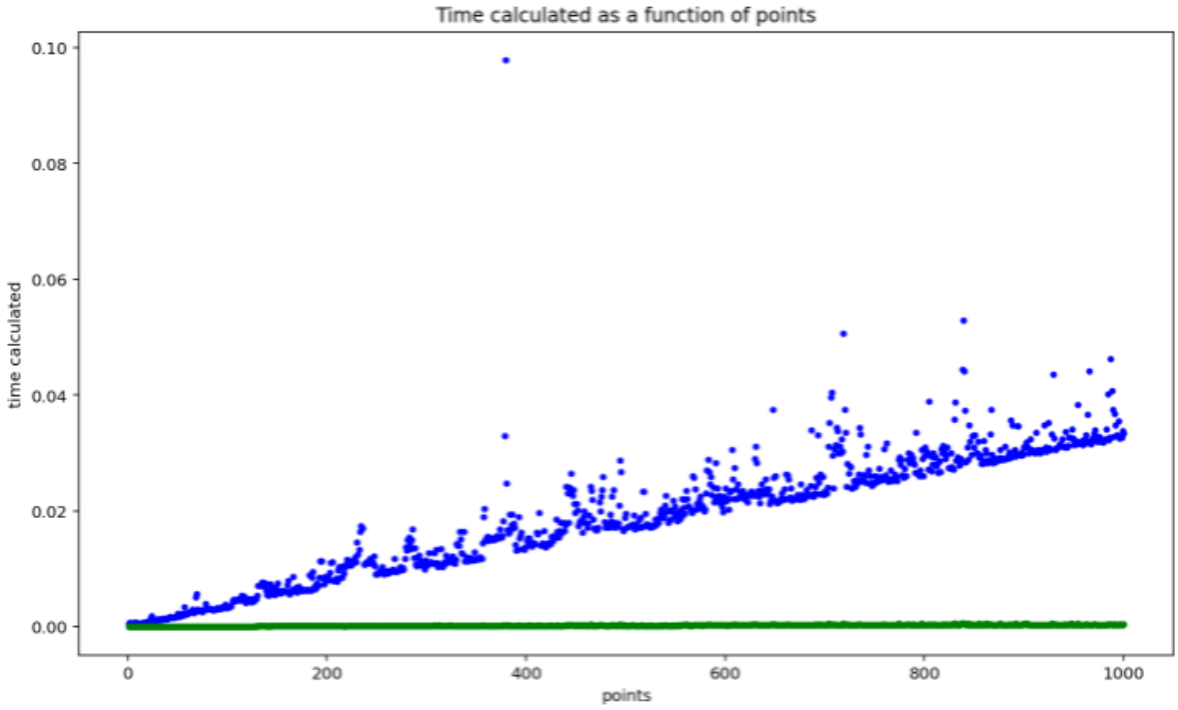


Fig. 20. Comparison of Run-Time 2

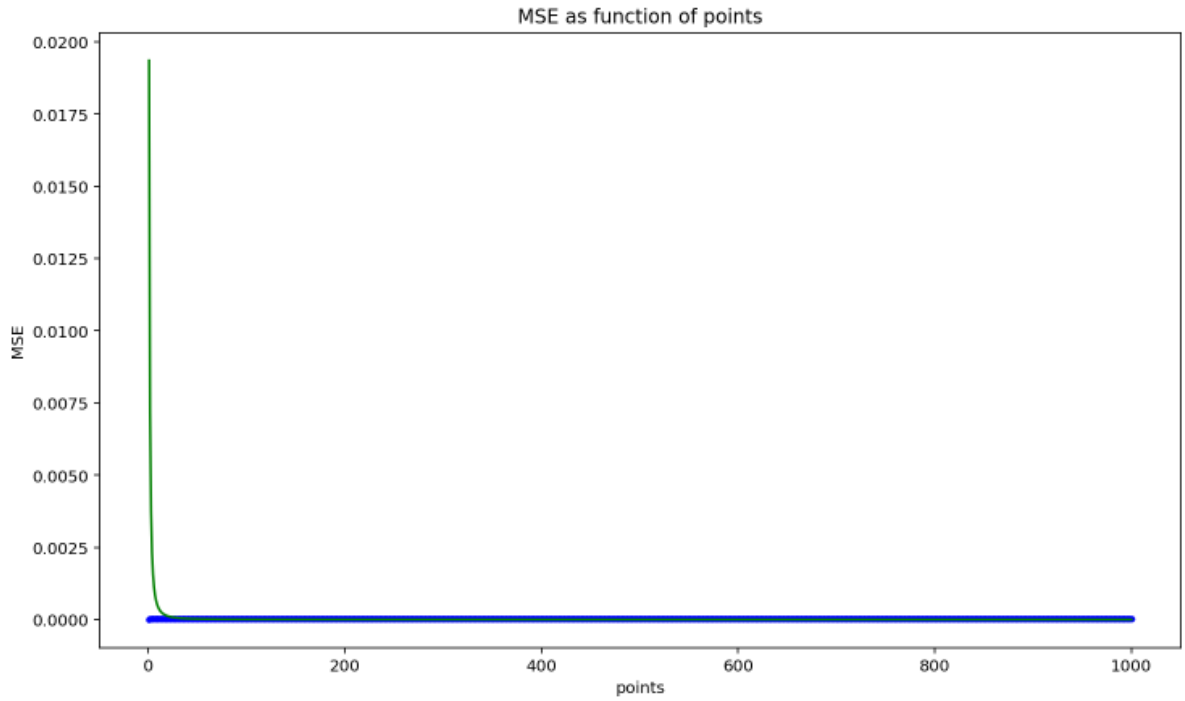


Fig. 21. Comparison of MSE 2

The graphs indicate that the numerical solution (Euler's method) is far more effective in approximating the solution to a given ordinary differential equation. The numerical solution seems to be both faster (about 5 orders of magnitude faster) and more accurate. It is important to note however that the code we wrote is unoptimized (Including the gradient descent and derivative approximations). Furthermore, results may vary significantly between different neural network architectures.

PARTIAL DIFFERENTIAL EQUATIONS

1. PDE's

In this section of the paper, we shall attempt to change parameters of a neural network to optimize the solving method and maximize the accuracy of the solution for partial differential equations. We will then compare the results to numerical solutions of the same equations and to analytical results of some of them. All the code used can be found on the GitHub repository in the PDE Jupyter notebook. Unless specified otherwise assume that we have one hidden layer (for the sake of simplicity) and two neurons (percptrons) in the hidden layer.

In addition, assume:

Number of iterations = $I = 10^2$

Learning rate = $\gamma = 10^{-3}$

Number of points in training set = $P_T = 10x10$

Number of points in output = $P_0 = 20x20$

Number of neurons in hidden layer = $\kappa = 2$

Activation function = $S(x)$

Example Run

1. $\nabla^2\psi(x,y) = e^{-x}(x - 2 + y^3 + 6y)$

With Dirichlet BC:

$\psi(0,y) = y^3,\ \psi(1,y) = (1+y^3)e^{-1},\ \psi(x,0) = xe^{-x},\ \psi(x,1) = e^{-x}(x+1)$

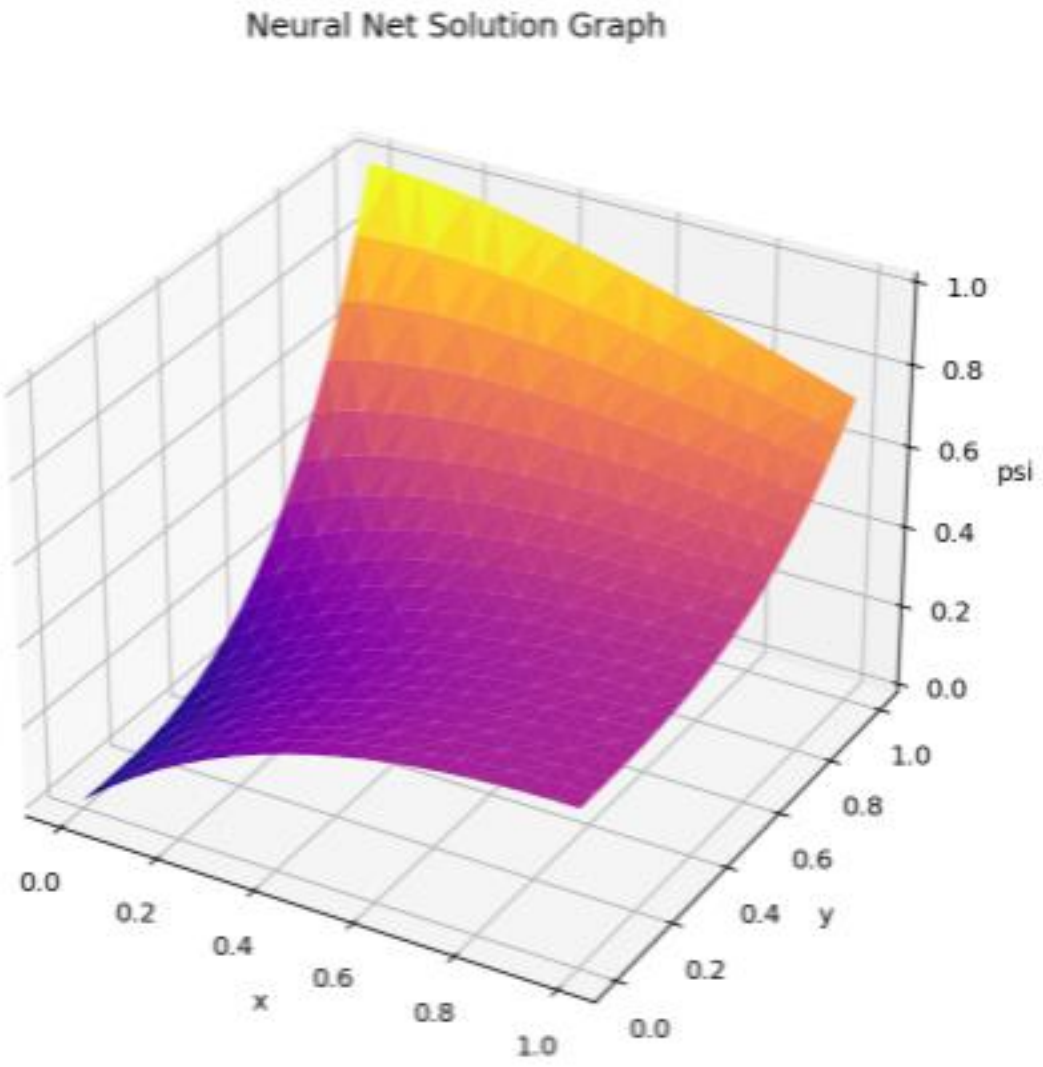


Fig. 22. Example Run Neural Network Solution

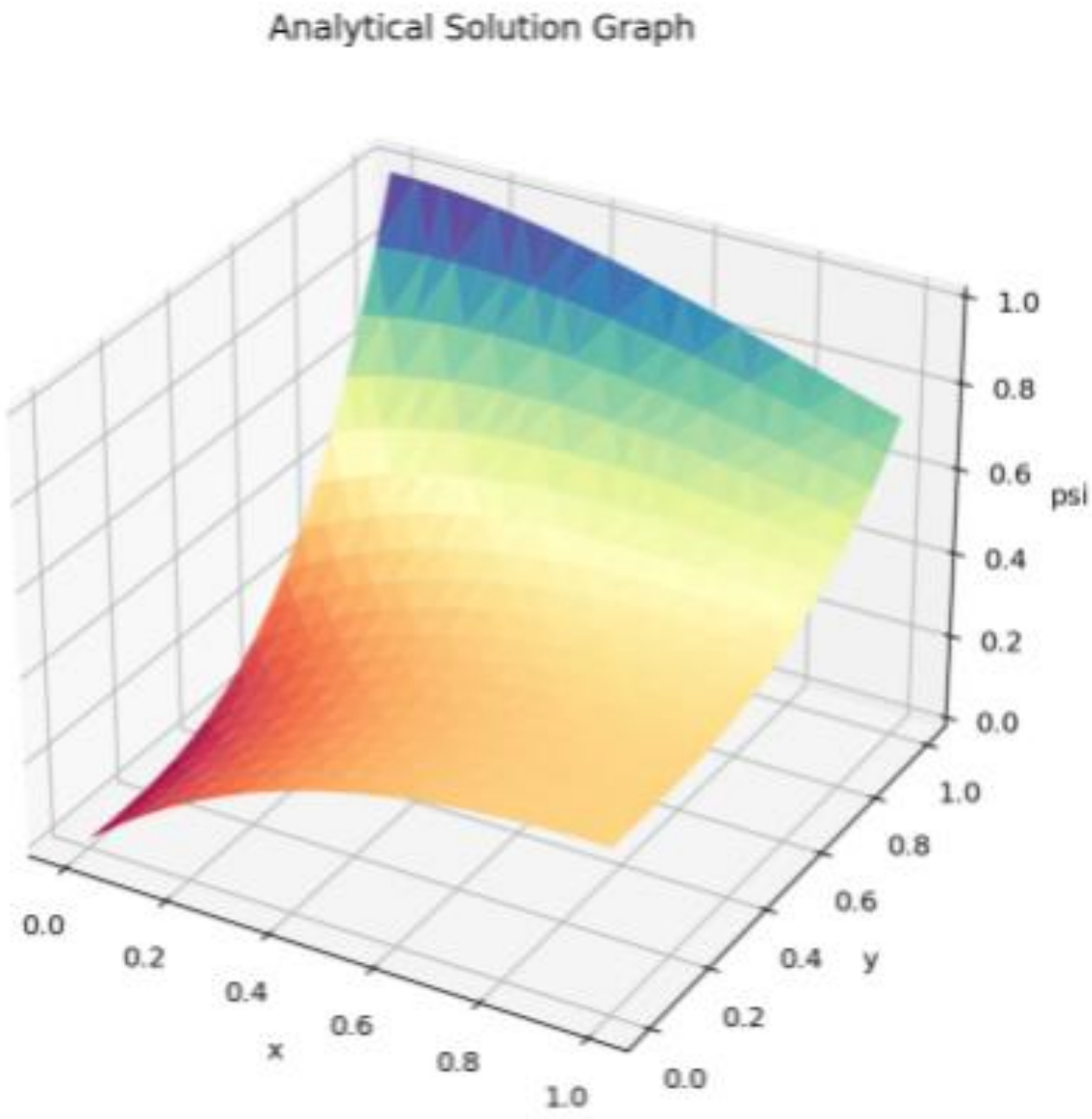


Fig. 23. Example Run Analytical Solution

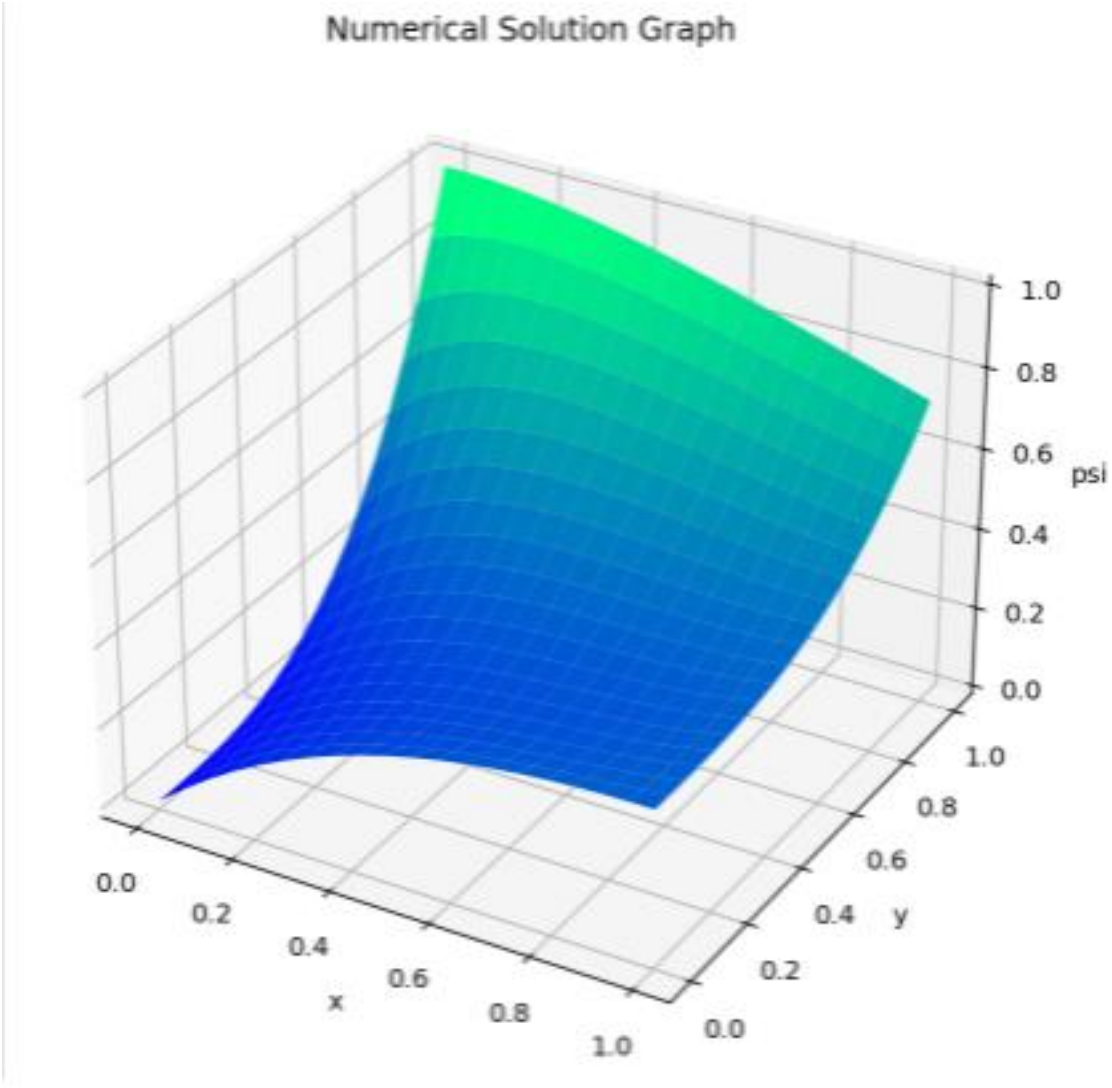


Fig. 24. Example Run Numerical Solution

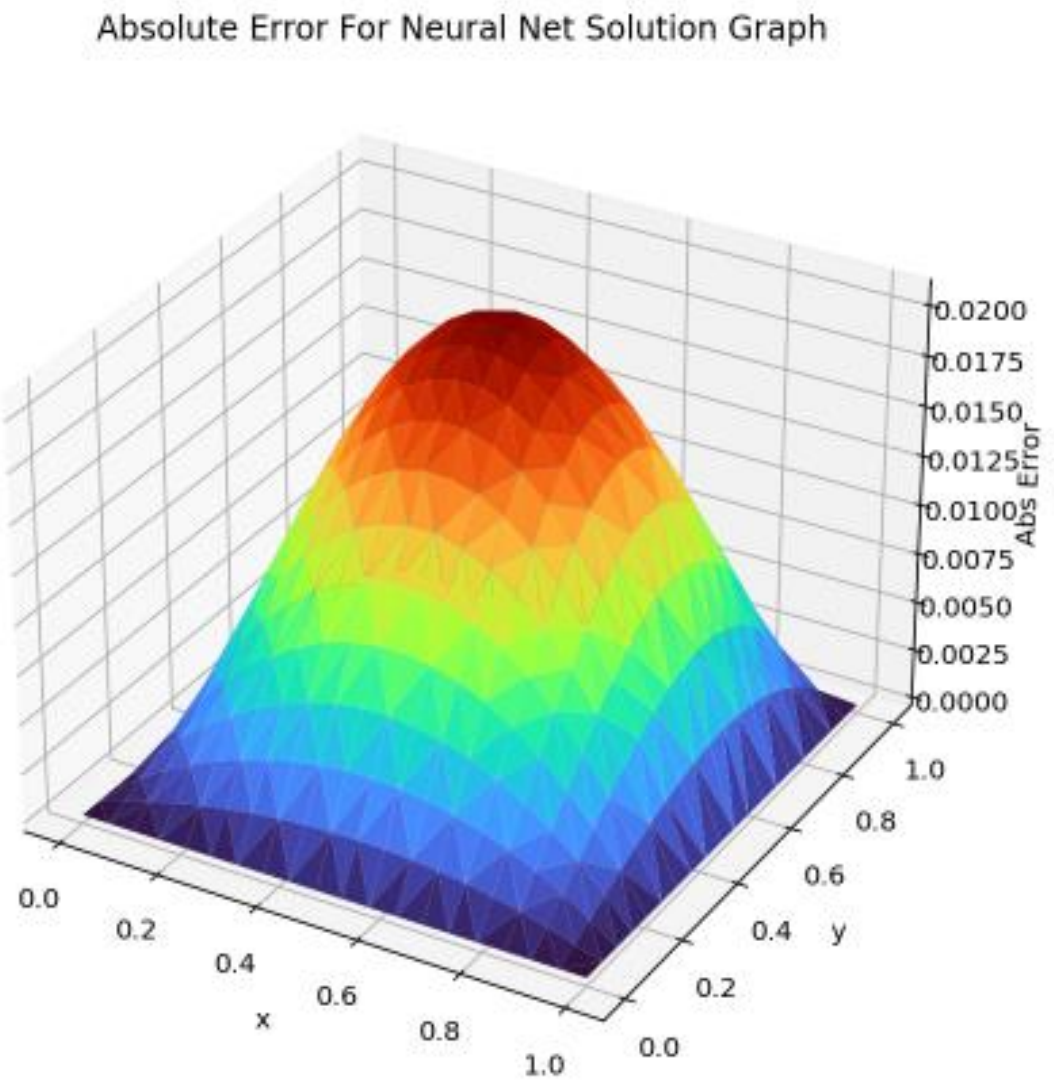


Fig. 25. Example Run Absolute Error Neural Net

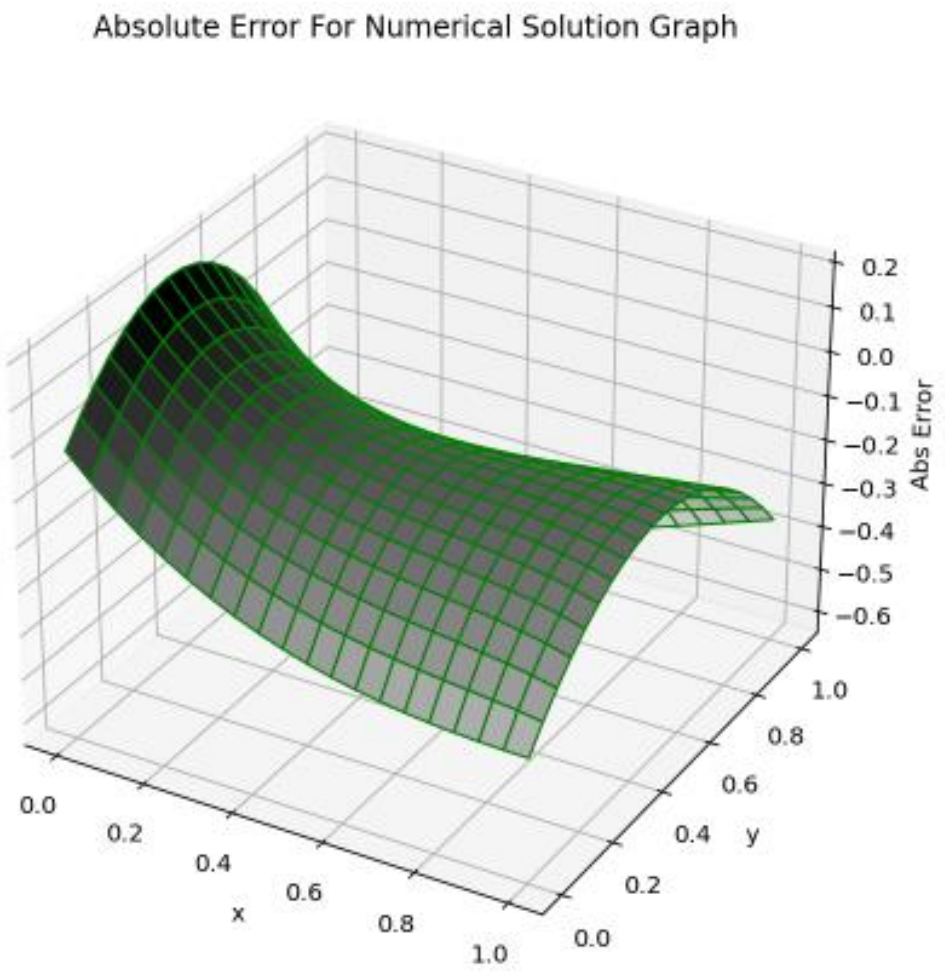


Fig. 26. Example Run Absolute Error Numerical

Types Of Graphs For Absolute Error (Neural Network)

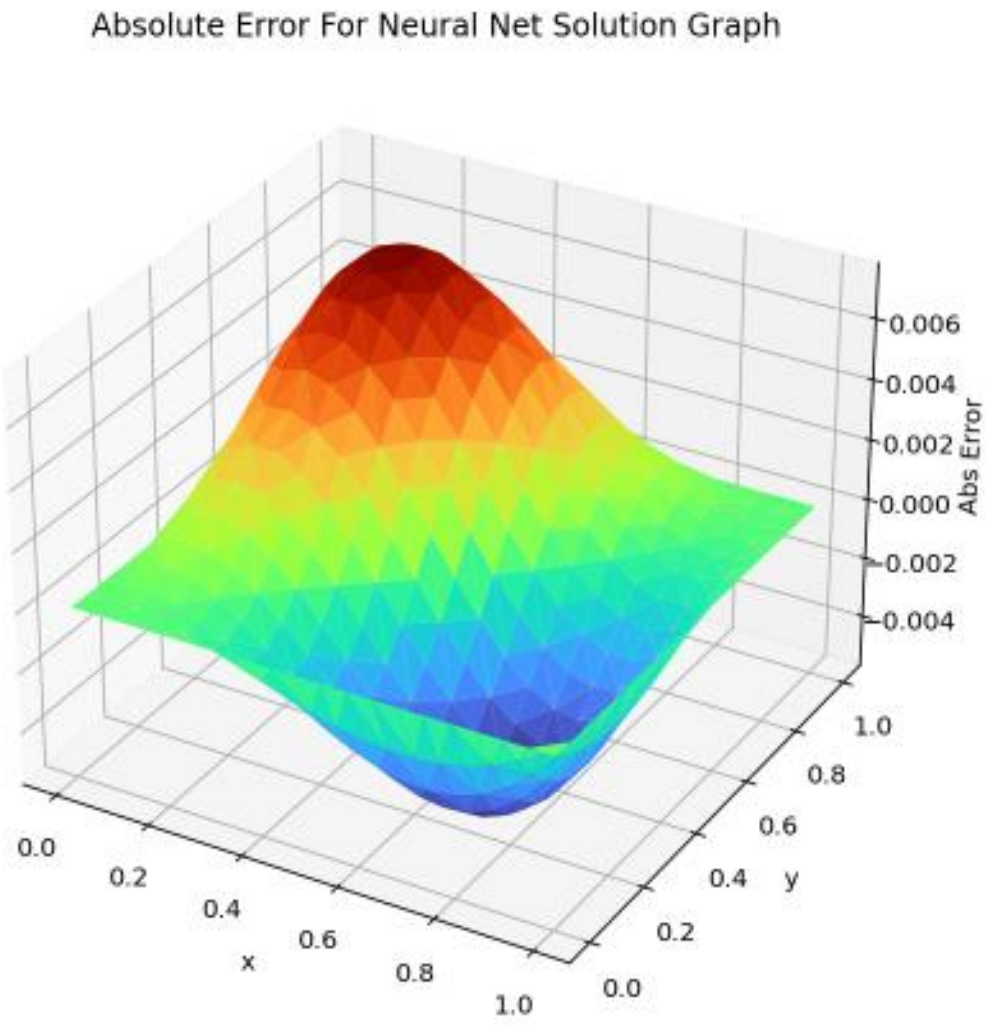


Fig. 27. Absolute Error Neural Network 2

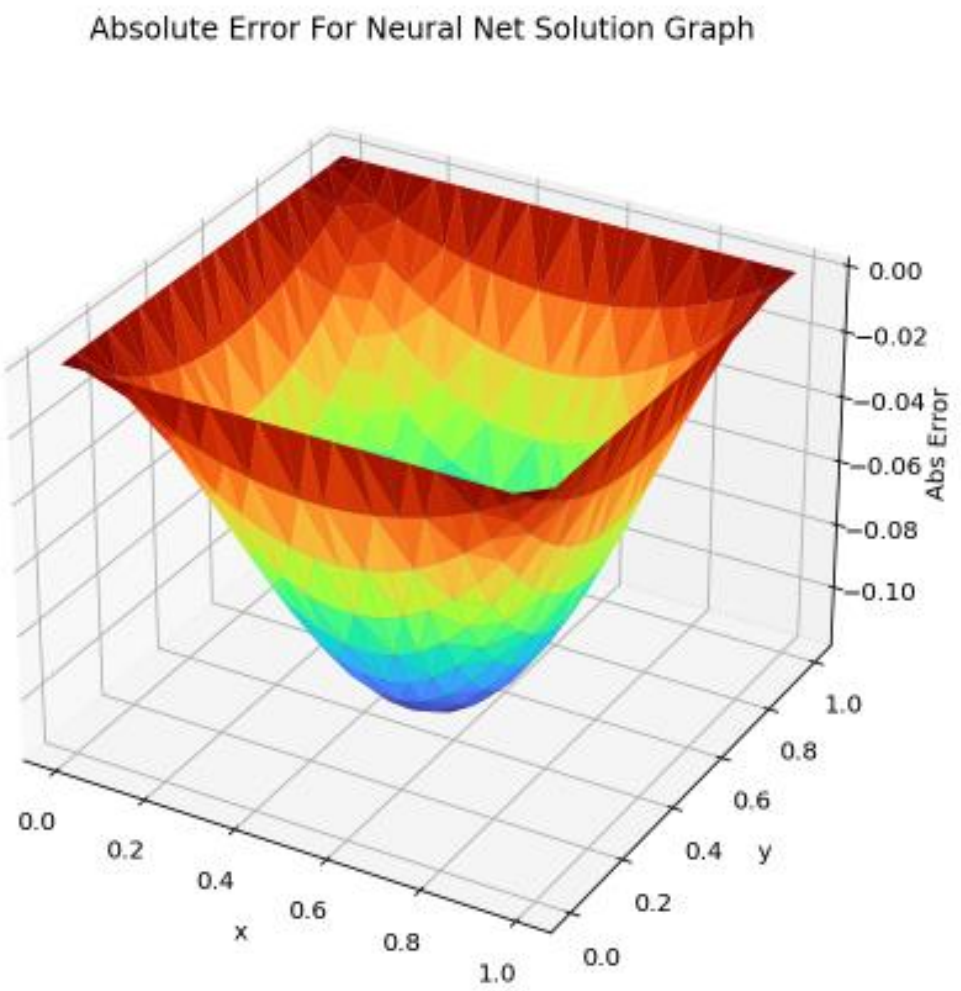


Fig. 28. Absolute Error Neural Network 3

The reason for the drastic change in accuracy is due to the neural network finding different weights and biases each time. This is a result of the neural network finding a different local minimum each run. We ran the code multiple times. Each time we get one of three “types” of plots. The first being the ″hill” as seen in fig 25, the second being the “inverse hill” as seen in fig 28, and the third being the “hill and ditch”, as seen in fig 27.

Run Time As Number Of Points Increases

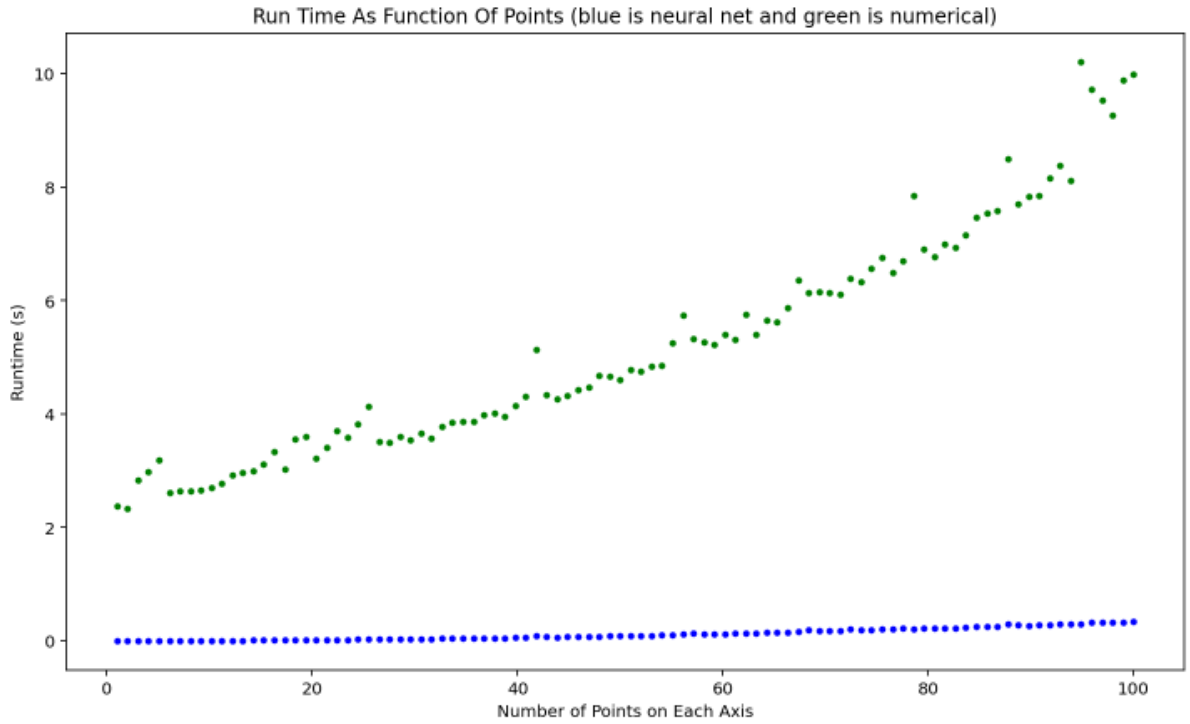


Fig. 29. Run Time as Function of Points

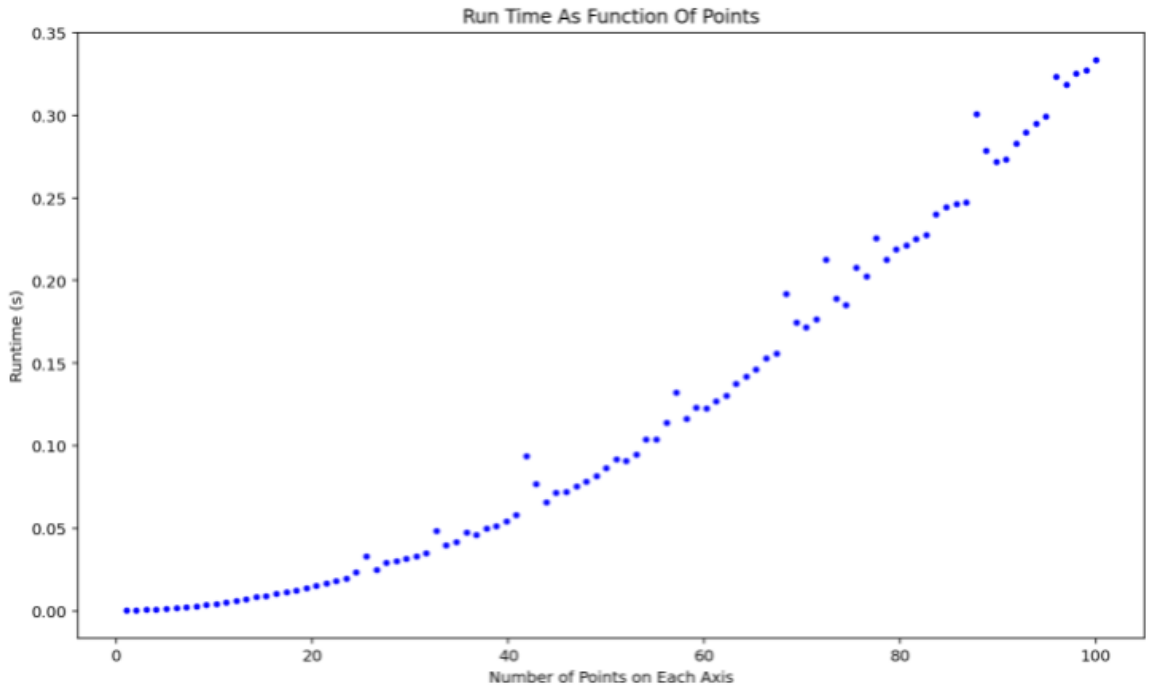


Fig. 30. Run Time as Function of Points (Neural Net Solution)

The graphs indicate that the neural network solution is far more effective in approximating the solution to a given partial differential equation. The neural network solution seems to be both faster and more accurate. It is important to note however that the code we wrote is unoptimized.

CONCLUSION

1. ODEs

The neural network solution for ordinary differential equations appears to be reasonable. Even more so when we consider that no training data set was required. Throughout the ordinary differential equation section, we examined how different parameters of the neural network changed the accuracy of the solution. The first thing we checked was how changing the number of neurons in the hidden layer affected the solution accuracy. We found that as one would expect, the accuracy increases. Along with the increase in accuracy there was also an increase in runtime and so when deciding on the number of neurons, it is important to also take the increase into account. In the following sub-section, we studied what effect the activation function of the hidden layer has on the solution of the neural network. We did this for multiple differential equations. To do an in-depth analysis we ran the solution five times with each activation function for each differential equation (In order to minimize the effect of local minima). When doing so we could not find a clear winner. We did however find that the sigmoid function provides a good balance between minimum accuracy and average accuracy over multiple runs. In the next two sub-sections, we looked at how the runtime and accuracy of the solution changes when we add points to the interval on which we trained the network. We did this for multiple differential equations. We found that the increase in runtime seemed to be linear. In addition, the accuracy did not decrease but rather remained constant. The final sub-section was a comparison of the neural network solution with a numerical solution given by Euler's method. We found that the numerical solution is far more effective in approximating the solution to an ODE. The numerical solution seems to be both faster and more accurate. It is important to note however that the code we wrote is unoptimized (Including the gradient descent and derivative approximations). Furthermore, results may vary significantly between different neural network architectures.

2. PDEs

The initial neural net solution we got for the PDE was quite accurate for most of the runs. The absolute error varied a lot, sometimes we got maximum errors as small as 0.006, and sometimes we got maximum errors as large as 0.1. This is probably due to the neural network settling on a local minimum which can cause great variation in the results. The comparison to the numerical solution was very surprising, we found that the neural network had a smaller runtime. In addition, it was more accurate than the numerical solution as the number of points increased. The only large disadvantage the neural network had is that we needed to train it beforehand.

3. Neural Network Solution vs. Numerical Solution

The comparison to the numerical solution is particularly interesting when considering the results, we obtained for partial differential equation to the ones we got for ordinary differential equations. The results seem to indicate that for ODEs a numerical solution is better while for PDEs the neural network solution would be preferred. It is important to note however that all the code was unoptimized as we wrote it ourselves, with the only exceptions being the libraries we utilized (NumPy and matplotlib) and the programming language itself. In addition, we only examined a limited number of differential equations.

BIBLIOGRAPHY

Lagaris, Isaac Elias, et al. “Artificial Neural Networks for Solving Ordinary and Partial Differential Equations.” *IEEE TRANSACTIONS ON NEURAL NETWORKS*, vol. 9, no. 5, Sept. 1998, doi:https://faculty.sites.iastate.edu/hliu/files/inline-files/Anns_lagaris1998_0.pdf.

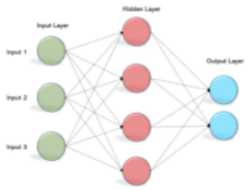
Ananthaswamy, Anil, and substantive Quanta Magazine moderates comments to facilitate an informed. “Latest Neural Nets Solve World’s Hardest Equations Faster Than Ever Before.” *Quanta Magazine*, 19 Apr. 2021, www.quantamagazine.org/new-neural-networks-solve-hardest-equations-faster-than-ever-20210419/.

APPENDIX

ODE CODE

In [455]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
```

In [456]:
```
#2
f = lambda x:2*x #ODE 1 derivative of function we are searching for
func0 = 48

def ftrig(t,psi): #ODE 2
    numerator = 1-psi*np.sin(t)
    denominator = np.cos(t)
    return numerator/denominator

def f_ode3(t,psi): #ODE 3
    return 9.8 - 0.196*psi

sig = lambda x:1/(1 + np.exp(-x)) #activation function
dsig = lambda x:(np.exp(-x))/(1+np.exp(-x))**2 #activation function derivative

def ReLU(x): #second activation function
    return x * (x > 0)

def dReLU(x):#second activation function derivative
    return 1. * (x > 0)
```



In [457]:
```
#3
def N(w0,w1,b0,b1,t): #neural network
    return np.dot(ReLU(np.dot(w0,t)+b0),w1) + b1
```

In [458]:
```
#4
def dN(w0,w1,b0,b1,t): #neural network derivative
    alpha = 1e-3 #term that helps find delta w where delta w = alpha*w
    return (N(w0,w1,b0,b1,t+alpha*t)-N(w0,w1,b0,b1,t-alpha*t))/(2*alpha)
    #return np.dot(np.dot(dsig(np.dot(w0,t) + b0).T,w0),w1)
```

In [459]:
```
#5
def L(w0,w1,b0,b1,t,func0): #cost function
    err_sum = 0 #sum of cost for each input
    for t_i in t:
        net_value = N(w0,w1,b0,b1,t_i)
        dnet_value = dN(w0,w1,b0,b1,t_i)
        psi = t_i*net_value+func0
        err_sum += (dnet_value*t_i + net_value - f_ode3(t_i,psi))**2
    return np.mean(np.sqrt(err_sum))
```

In [460]:
```
#6
def grad(w0,w1,b0,b1,t,func0): #gradient of the cost function
    alpha = 1e-3 #term that helps find delta x where delta x  = alpha*x

    #find derivative of cost function with respect to each variable
    dLdw0 = (L(w0+alpha*w0,w1,b0,b1,t,func0) - L(w0-alpha*w0,w1,b0,b1,t,func0))/(2*alpha*w0)
    dLdw1 = (L(w0,w1+alpha*w1,b0,b1,t,func0) - L(w0,w1-alpha*w1,b0,b1,t,func0))/(2*alpha*w1)
    dLdb0 = (L(w0,w1,b0+alpha*b0,b1,t,func0) - L(w0,w1,b0-alpha*b0,b1,t,func0))/(2*alpha*b0)
    dLdb1 = (L(w0,w1,b0,b1+alpha*b1,t,func0) - L(w0,w1,b0,b1-alpha*b1,t,func0))/(2*alpha*b1)

    return dLdw0,dLdw1,dLdb0,dLdb1
```

In [461]:
```
#7
def gradient_descent(w0,w1,b0,b1,t,learning_rate,n_iter,func0): #gradient descent algorithm
    s0 = w0
    s1 = w1
    s2 = b0
    s3 = b1
    for _ in range(n_iter):
        diff = np.multiply(grad(s0,s1,s2,s3,t,func0),-learning_rate) #find out how much we need to change each variable
        s0 = np.add(s0,diff[0]) #change w0
        s1 = np.add(s1,diff[1]) #change w1
        s2 = np.add(s2,diff[2]) #change b0
        s3 = np.add(s3,diff[3]) #change b1
        #print("Loss: " + str(L(s0,s1,s2,s3,t))) #calculate the new value of the loss
    return s0,s1,s2,s3,L(s0,s1,s2,s3,t,func0)
```

In [462]:
```
#number of nodes in each layer
n_input = 1
n_hidden = 2
n_output = 1

#values for training
learning_rate = 0.001
n_iter = 10000
t_min = 0
t_max = 1
t = np.linspace(t_min,t_max,10)

#1 - create the weight and bias vectors
w0 = np.random.randn(n_input,n_hidden)
w1 = np.random.randn(n_hidden,n_output)
b0 = np.random.randn(n_hidden)
b1 = np.random.randn(n_output)
```

In [463]:
```
values = gradient_descent(w0,w1,b0,b1,t,learning_rate,n_iter,func0)
w0 = values[0]
w1 = values[1]
b0 = values[2]
b1 = values[3]
print(values[4])
```

```
0.07050512085106553
```

In [464]:
```
#function we are approximating
def func(x,func0):
    return 50 - 2*np.exp(-0.196*x)
```

In [465]:
```
def mean_squared_error(true, pred):# outputs the mean squared error of the nerual networks solution in relation to other functions (numer
ical or analytical)
    mean_squared_error = np.square(np.subtract(true,pred)).mean()
    return mean_squared_error
```

In [466]:
```
#Euler method
def Euler(f,x0,y0,x1,dx = 0.1):
    start_time = time.time()
    x_array = []
    y_array = []
    x = x0
    y = y0
    n = int(round((x1 - x0) / dx + 1,0))
    x_array.append(x0)
    y_array.append(y0)
    for i in range(1, n):
        y_array.append(y_array[i-1]+f_ode3(x_array[i-1],y_array[i-1])*dx)
        x_array.append(x_array[i-1] + dx)

    runtime = time.time() - start_time

    y_array = np.delete(y_array, 0)

    return y_array,runtime
```

In [467]:
```
#plot to test results

num_p = 100 #Number of points in interval

t = np.linspace(0,1,num_p) #add points in the interval to test solution

psi = [] #holds function approximation
dpsi = [] #contains function derivative approximation


for t_i in t:
    neural_net_value = N(w0,w1,b0,b1,t_i).tolist()[0][0] #calculate point in neural network
    dneural_net_value = dN(w0,w1,b0,b1,t_i).tolist()[0][0] #calculate point in neural network derivative
    psi.append(t_i*neural_net_value+func0) #calculate and add the point in function approximation
    dpsi.append(dneural_net_value*t_i+neural_net_value) #calculate and add the point in function derivative approximation


dx = 1/num_p
numerical_boi = Euler(f,0,func0,1,dx)[0] #find numerical solution

#plot graph
plt.title("Function-Time Graph (blue is neural network solution and green is numerical solution)")
plt.xlabel("t")
plt.ylabel("psi")

plt.scatter(t,func(t,func0),marker = ".", color = 'r')
plt.plot(t,psi,'b')
plt.plot(t,numerical_boi,'g')
```

PDE CODE

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
```

In [2]:
```
sig = lambda x:1/(1 + np.exp(-x)) #activation function
dsig = lambda x:(np.exp(-x))/(1+np.exp(-x))**2 #activation function derivative
```

In [3]:
```
def N(w0,w1,b0,b1,x,y): #neural network
    inputs = np.array([x,y])
    input_hidden = sig(np.dot(w0,inputs)+b0)
    hidden_output = np.dot(input_hidden,w1) + b1
    return hidden_output
```

In [4]:
```
def dNdx(w0,w1,b0,b1,x,y): #neural network derivative with respect to x
    alpha = 1e-3 #term that helps find delta w where delta w = alpha*w
    return (N(w0,w1,b0,b1,x+alpha*x,y)-N(w0,w1,b0,b1,x-alpha*x,y))/(2*alpha)
```

In [5]:
```
def dNdy(w0,w1,b0,b1,x,y): #neural network derivative with respect to y
    alpha = 1e-3 #term that helps find delta w where delta w = alpha*w
    return (N(w0,w1,b0,b1,x,y+alpha*y)-N(w0,w1,b0,b1,x,y-alpha*y))/(2*alpha)
```

In [6]:
```
def A(x,y): #Calculated using boundary conditions
    return (1-x)*f0(y) + x*f1(y) + (1-y)*(g0(x) - ((1 - x)*g0(0) + x*g0(1))) + y*(g1(x) - ((1 - x)*g1(0) + x*g1(1)))
```

In [7]:
```
def L(w0,w1,b0,b1,x,y): #cost function
    err_sum = 0 #sum of cost for each input
    for y_i in y:
        for x_i in x:
            net_value = N(w0,w1,b0,b1,x_i,y_i)
            dxnet_value = dNdx(w0,w1,b0,b1,x_i,y_i)
            dynet_value = dNdy(w0,w1,b0,b1,x_i,y_i)
            dnet_value = dxnet_value + dynet_value
            psi = A(x_i,y_i)+x_i*(1-x_i)*y_i*(1-y_i)*net_value
            err_sum += (dnet_value - f_pde(x_i,y_i,psi))**2
    return np.mean(np.sqrt(err_sum))
```

In [8]:
```
def gradw0(w0,w1,b0,b1,x,y): #gradient of the cost function
    alpha = 1e-3 #term that helps find delta x where delta x  = alpha*x
    dLdw0 = (L(w0+alpha*w0,w1,b0,b1,x,y) - L(w0-alpha*w0,w1,b0,b1,x,y))/(2*alpha*w0)
    return dLdw0

def gradw1(w0,w1,b0,b1,x,y):
    alpha = 1e-3 #term that helps find delta x where delta x  = alpha*x
    dLdw1 = (L(w0,w1+alpha*w1,b0,b1,x,y) - L(w0,w1-alpha*w1,b0,b1,x,y))/(2*alpha*w1)
    return dLdw1

def gradb0(w0,w1,b0,b1,x,y):
    alpha = 1e-3 #term that helps find delta x where delta x  = alpha*x
    dLdb0 = (L(w0,w1,b0+alpha*b0,b1,x,y) - L(w0,w1,b0-alpha*b0,b1,x,y))/(2*alpha*b0)
    return dLdb0

def gradb1(w0,w1,b0,b1,x,y):
    alpha = 1e-3 #term that helps find delta x where delta x  = alpha*x
    dLdb1 = (L(w0,w1,b0,b1+alpha*b1,x,y) - L(w0,w1,b0,b1-alpha*b1,x,y))/(2*alpha*b1)
    return dLdb1
```

In [9]:
```
def gradient_descent(w0,w1,b0,b1,x,y,learning_rate,n_iter): #gradient descent algorithm
    s0 = w0
    s1 = w1
    s2 = b0
    s3 = b1
    for _ in range(n_iter):
        diffw0 = np.multiply(gradw0(s0,s1,s2,s3,x,y),-learning_rate) #find out how much we need to change
        diffw1 = np.multiply(gradw1(s0,s1,s2,s3,x,y),-learning_rate) #find out how much we need to change each weight
        diffb0 = np.multiply(gradb0(s0,s1,s2,s3,x,y),-learning_rate)
        diffb1 = np.multiply(gradb1(s0,s1,s2,s3,x,y),-learning_rate)
        s0 = np.add(s0,diffw0) #change w0
        s1 = np.add(s1,diffw1) #change w1
        s2 = np.add(s2,diffb0) #change b0
        s3 = np.add(s3,diffb1) #change b1
        #print("Loss: " + str(L(s0,s1,s2,s3,t))) #calculate the new value of the loss
    return s0,s1,s2,s3,L(s0,s1,s2,s3,x,y)
```

In [10]:
```
def f_pde(x,y,psi): #PDE 1
    return np.exp(-1*x) * (x-2+y**3+6*y)
```

In [11]:
```
#boundary condition functions PDE 1 (dirichlet boundary condition)
def f0(y): #BC at (0,y)
    return y**3

def f1(y): #BC at (ny,y)
    return (1 + y**3)*np.exp(-1)

def g0(x): #BC at (x,0)
    return x * np.exp(-1*x)

def g1(x): #BC at (x,nx)
    return (x + 1) * np.exp(-1*x)
```

In [12]:
```
def pde_sol(x,y): #PDE 1 solution
    return np.exp(-1*x)*(x+y**3)
```

In [13]:
```
#Numerical Solution

#Initial conditions for source term (RHS)
def source_term(f,x,y,nx,ny):
    f[:,:] = np.exp(-1*x) * (x-2+y**3+6*y)

#Boundary conditions for solution
def boundary(psi,x,y,nx,ny):
    psi[:,0] = y**3
    psi[:,nx-1] = (1+y**3)*np.exp(-1)
    psi[0,:] = x*np.exp(-1*x)
    psi[ny-1,:] = np.exp(-1*x)*(x+1)

#Solve the poisson equation (using discrete form)
def discrete_poisson(psi,f,x,y,t):
    for i in range(0,t):
        prev_psi = psi.copy()
        psi[1:-1,1:-1] = (((prev_psi[1:-1, 2:] + prev_psi[1:-1, :-2]) * dy**2 + (prev_psi[2:, 1:-1] + prev_psi[:-2, 1:-1]) * dx**2 - f[1:
-1, 1:-1] * dx**2 * dy**2) / (2 * (dx**2 + dy**2)))
        boundary(psi,x,y,nx,ny)

#Plot the solution on a 3d graph as a wireframe
def plot_wireframe(psi,x,y):
    fig = plt.figure()
    ax = plt.axes(projection="3d")
    X, Y = np.meshgrid(x, y)
    ax.plot_wireframe(X, Y, psi, color='green')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('psi')
    plt.show()

#Plot the solution on a 3d graph
def plot(psi,x,y):
    fig = plt.figure()
    ax = plt.axes(projection="3d")
    X, Y = np.meshgrid(x, y)
    ax.plot_surface(X, Y, psi, rstride=1, cstride=1, cmap='winter', edgecolor='none')
    plt.title("Numerical Solution Graph")
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('psi')
    plt.show()
```

In [14]:
```
#number of nodes in each layer
n_input = 2
n_hidden = 2
n_output = 1

#Number of points on (x,y) grid space
nx = 10
ny = 10

#values for training
learning_rate = 0.001
n_iter = 100
x_min = 0
x_max = 1
x = np.linspace(x_min,x_max,nx)
y_min = 0
y_max = 1
y = np.linspace(y_min,y_max,ny)

#1 - create the weight and bias vectors
w0 = np.random.randn(n_input,n_hidden)
w1 = np.random.randn(n_hidden,n_output)
b0 = np.random.randn(n_hidden)
b1 = np.random.randn(n_output)
```

In [15]:
```
values = gradient_descent(w0,w1,b0,b1,x,y,learning_rate,n_iter)
w0 = values[0]
w1 = values[1]
b0 = values[2]
b1 = values[3]
print(values[4])
```

In [16]:
```
#plot to test results

num_p = 20 #Number of points in interval

x = np.linspace(0,1,num_p) #add points in the interval to test solution
y = np.linspace(0,1,num_p) #add points in the interval to test solution

psi = [] #holds function approximation
sol = [] #hold function

for y_i in y:
    for x_i in x:
        neural_net_value = N(w0,w1,b0,b1,x_i,y_i) #calculate point in neural network
        psi_value = (A(x_i,y_i)+x_i*(1-x_i)*y_i*(1-y_i)*neural_net_value)[0]
        psi.append(psi_value) #calculate and add the point in function approximation
        sol.append(pde_sol(x_i,y_i))
```

In [17]:
```
mesh_x = []
mesh_y = []
for add in range (0,len(y)):
    for add2 in range(0,len(x)):
        mesh_x.append(y[add2])
        mesh_y.append(y[add])
```

In [18]:
```
#Plot the neural network solution on a 3d graph
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.title("Neural Net Solution Graph")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('psi')
ax.plot_trisurf(mesh_x,mesh_y,psi,cmap=plt.cm.plasma)
```

In [19]:
```
#Plot the solution on a 3d graph
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.title("Analytical Solution Graph")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('psi')
ax.plot_trisurf(mesh_x,mesh_y,sol,cmap=plt.cm.Spectral)
```

In [20]:
```
difference = []

zip_object = zip(sol, psi)
for list1_i, list2_i in zip_object:
    difference.append(list1_i-list2_i)

fig = plt.figure()
ax = fig.gca(projection='3d')
plt.title("Absolute Error For Neural Net Solution Graph")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Abs Error')
ax.plot_trisurf(mesh_x,mesh_y,difference,cmap=plt.cm.turbo)
```

In [21]:
```
#Numerical Solution of the equation

#time we will run solution over (not real time, ideally it would go to infinity)
t = 100000

#Number of points on (x,y) grid space
nx = 20
ny = 20

#Max values for x and y
xmin = 0
xmax = 1
ymin = 0
ymax = 1

#Finding the delta value in each direction
dx = (xmax - xmin) / (nx - 1)
dy = (ymax - ymin) / (ny - 1)

#Creating axis for x and y
x  = np.linspace(xmin, xmax, nx)
y  = np.linspace(xmin, xmax, ny)

#Creating grid for potential values
psi_numerical = np.zeros((ny,nx))
prev_psi_numerical = np.zeros((ny,nx))

#Creating grid for function values
f = np.zeros((ny,nx))
```

In [22]:
```
source_term(f,x,y,nx,ny)
discrete_poisson(psi_numerical,f,x,y,t)
plot(psi_numerical,x,y)
```

In [48]:
```
difference2 = []

zip_object = zip(sol, psi_numerical)
for list1_i, list2_i in zip_object:
    difference2.append(list1_i-list2_i)

difference2 = np.array(difference2)

fig = plt.figure()
ax = plt.axes(projection="3d")
X, Y = np.meshgrid(x, y)
ax.plot_surface(X, Y, difference2, rstride=1, cstride=1, cmap='binary', edgecolor='g')
plt.title("Absolute Error For Numerical Solution Graph")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Abs Error')
plt.show()
```

GITHUB LINK FOR CODE