Solving Differential Equations Using Neural Networks, With Applications to Eigenvalue Problems

Jan Egil Ødegård

(Dated: December 14th, 2020)

The source code files can be found in my GitHub Repository

In this project we have taken a closer look at the application of neural networks when solving differential equations of different types. First, we have looked closer at a partial differential equation (PDE), specifically the heat equation, and compared this method with a standard forward Euler / explicit scheme algorithm of solving PDEs. We have specifically looked closer at comparing them in terms of efficiency, accuracy, and critically assess the pros and cons of the different methods. Further, we have also looked at the possibility of solving a specific non-linear differential equation with the added application of finding a symmetric, real matrix's eigenvectors and corresponding eigenvalues. We will compare this to regular linear algebra numerical diagonalization schemes both in terms of accuracy and efficiency. The explicit method seems to outperform the neural network in terms of accuracy and computational performance, in addition to overall being a simpler algorithm to implement. Training up a neural network over roughly an hour gives an order of magnitude worse results than the explicit scheme gives using 1 millisecond. As for the eigenvalue solver, there is no eigenvalue solver (that functions). As such, this report is only a shell of what could have been achieved. Overall, the neural networks seem to perform more poorly, and are vastly less intuitive in setting up, compared to their direct numerical counterpart. The possibilities are however somewhat undiscovered, as finding the ideal parameters is an entire task in itself.

I. INTRODUCTION

Certain problems in mathematics, statistics, physics, and most other sciences, can be stated using differential equations of varying complexities. Some are simple to solve analytically, some are more tedious, and some are downright unsolvable using pen and paper. Especially for the latter ones, different numerical methods to approximate solutions can be achieved using smart algorithms. These methods usually come with a catch, either being methods that scale poorly in terms of run time and accuracy, or with severe limitations in terms of stabilities of solutions. We will here take a look closer at an alternative approach to solving differential equations, namely one of neural networks.

First, we will look closer at the prospect of solving Partial Differential Equations (PDEs) using a neural network. Specifically, we will be looking closer at the heat equation. For this, we will compare both a straightforward approach of numerically solving the PDE using the explicit scheme based on forward Euler, together with our Neural Network approach. We will take a comparative look, both in terms of efficiency, scaling, accuracy, and overall assert a critical assessment of the methods and their advantages or disadvantages as compared to each other.

We will also be looking at the possibility of solving a non-linear differential equation using neural network. The differential equation we will be looking closer at is one proposed by Yi et al., which has the added benefit in which we could also compute eigenvectors and eigenvalues of a symmetric, real matrix[1]. By solving this differential equation and finding said eigenvalues and eigenvectors, we could compare our results to that of ordinary eigenvalue solvers using other aspects of linear algebra to compare both efficiency and accuracy. We will here as well also give a critical assessment of the methods presented compared to the standard linear algebra ones.

CONTENTS

II. THEORY

This theory section will first look closer at the specific PDE we will be solving. This is followed by an introduction to the numerical explicit scheme we will use to solve this equation. Following this, we will discuss the notion of solving a differential equation using a neural network. Finally, we will introduce a non-linear differential equation which we can solve for eigenvalues and eigenvectors for a given symmetric matrix.

A. The Partial Differential Equation to solve

For this project, we will look closer at the diffusion equation, with an emphasis on the interpretation that it represents the temperature distribution of a rod of length L. The diffusion equation we will look into is on the form:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \quad , \quad t \geq 0 \quad , \quad x \in [0, L]. \tag{1}$$

We will specifically look closer at the case of the rod having length $L = 1$. Our boundary conditions will be defined as:

$$u(0,t) = u(L,t) = 0 \quad , \quad t \geq 0. \tag{2}$$

Finally, we will define our initial condition to be on the form:

$$u(x,0) = \sin(\pi x). \tag{3}$$

The PDE in equation 1, together with boundary conditions in equation 2 and the initial condition in equation 3, has the following analytic solution:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t}. \tag{4}$$

For derivation of equation 4, see appendix A.

B. The Explicit Forward Euler

When attempting to solve the PDE in equation 1 numerically, one could be inclined to approximate them using the forward-Euler algorithm. The first derivative in time is then given as:

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x_i, t_j + \Delta t) - u(x,t)}{\Delta t}. \tag{5}$$

To ease notation, knowing that the index "i" corresponds to the spatial component and the index "j" corresponds to the temporal one, we write this as:

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u_{i,j+1} - ui,j}{\Delta t}. \tag{6}$$

This has an approximation error running as $\mathcal{O}(t)$. Likewise when using the forward-Euler scheme on the spatial double derivative, this would become:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2}. \tag{7}$$

This having an approximation error running as $\mathcal{O}(x^2)$. Exchanging the derivatives in equation 1 with the numerical approximations of equations 6 and 7, we get:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} = \frac{u_{i,j+1} - ui,j}{\Delta t}. \tag{8}$$

Given that we know the state of our system at every point at a time step t_j, the only unknown term in this equation would be $u_{i,j+1}$. Solving for this, defining the quantity $\alpha \equiv \frac{\Delta t}{(\Delta x)^2}$ yields:

$$u_{i,j+1} = \alpha\big(u_{i+1,j} + u_{i-1,j} - 2u_{i,j}\big) + u_{i,j}. \tag{9}$$

Equation 9 is the expression we will have to solve for all t, given our initial conditions and boundary conditions. This scheme does however have a stability criterion requiring that $\alpha = \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$[2]. This ultimately means that for a choice $\Delta x = \frac{1}{100}$, Δt must be of size $\Delta t = 5 \cdot 10^{-5}$ or smaller. This is a strong suppressor of an optimal code, and will tend to slow down computations quadratically for larger spatial resolutions.

C. PDE Solving Using Neural Networks

Another way one can attempt to solve the PDE is by training a neural network to solve it. For this, we will be using a fully-connected, feed forward neural network, using back-propagation to train the associated weights and biases. For a more general introduction to the structure and conceptual ideas of this type of neural network, see our earlier report[3]. For the specific problem at hand, we must then define an appropriate input layer, output layer, and a fitting cost function. We also need to discuss appropriate activation functions, ideal learning rates, ideal depths of the hidden layers and the respective sizes for each individual layer. This is to be discussed now.

For our specific problem, we wish to approximate the function $u(x,t)$. As such, the input layer will necessarily consist of 2 nodes, corresponding to the spatial coordinate x and the temporal coordinate t. Our output layer corresponding to $u(x,t)$ will then have a single node. Further, we rewrite our PDE from equation 1 to be of a more suggesting form:

$$\frac{\partial^2 u(x,t)}{\partial x^2} - \frac{\partial u(x,t)}{\partial t} = 0. \tag{10}$$

Considering that these two expressions are to be equal, we can define our cost function as the squared difference between these:

$$\mathcal{C}(x,t,P) = \left(\frac{\partial^2 u(x,t)}{\partial x^2} - \frac{\partial u(x,t)}{\partial t} - 0 \right)^2. \tag{11}$$

Here, P denotes the parameters associated with our neural network, specifically being our weights and biases. The values of $u(x,t)$ will be calculated using the neural network, and is as such also in reality dependent on the parameters P, this has however been suppressed in the notation. We want to minimize the cost function in equation 11, using our free parameters P. We denote this mathematically, with $\hat{P}$ being the optimal parameters, as:

$$\mathcal{C}(x,t,\hat{P}) = 0 = \min_P \left\{ \left(\frac{\partial^2 u(x,t,P)}{\partial x^2} - \frac{\partial u(x,t,P)}{\partial t} \right)^2 \right\} \tag{12}$$

Here, we underline that this notation implies minimisation of the cost function with regard to the free parameters P. To determine what the derivatives inside this cost function should be, we need to have an expression on a function form which we can evaluate. We therefore create an arbitrary trial function u_{trial} on the following form:

$$u_{trial}(x,t,P) = u_1(x,t) + x(1-x)t \cdot N(x,t,P). \tag{13}$$

Here $N(x,t,P)$ is the output from our neural network, and u_1 is a function which will satisfy the boundary and initial conditions. We pick u_1 to be the following:

$$u_1(x,t) = (1-t)\Big(u(x,0) - \big((1-x)u(0,t) + xu(L,t) \big) \Big). \tag{14}$$

Here $u(x,0)$ is our initial condition (defined in equation 3), while $u(0,t)$ and $u(L,t)$ are our boundary conditions (stated in equation 2). Inserting for these into equation 14, and inserting this expression into equation 13, we get our trial solution to be:

$$u_{trial}(x,t,P) = \Big[(1-t)\sin(\pi x) + x(1-x)t \cdot N(x,t,P) \Big]. \tag{15}$$

This is on a functional form such that we can numerically determine the derivatives in equation 11. We can then determine the new parameters P using a gradient method as follows:

$$P_i = P_{i-1} - \lambda \cdot \frac{\partial \mathcal{C}}{\partial P_{i-1}} \tag{16}$$

Where the subscript i denotes the i'th iteration of parameters, where we train the model over many subsequent iterations. Given an optimal choice of learning rate parameter λ, this should ideally converge towards a local (and hopefully global) minima of our cost function.

Theoretically, having trained our network over N total iterations with the scheme described above, we could feed in arbitrary values for both x and t into our network to retrieve a function value $u(x,t)$.

D. Expanding Differential Equation Neural Networks to Matrix Eigenvalue Problems

The notion of solving differential equations with neural networks can be extended into the domain of solving for eigenvalues. For instance, we look to the non-linear differential equation:

$$\frac{d\mathbf{x}(t)}{dt} = \dot{\mathbf{x}}(t) = -\mathbf{x}(t) + f(\mathbf{x}(t)) \;\; , \;\; t \geq 0 \tag{17}$$

Where we define $f(x)$ as:

$$f(\mathbf{x}) = \big[\mathbf{x}^T \mathbf{x} \mathbf{A} + (1 - \mathbf{x}^T \mathbf{A} \mathbf{x}) \mathbb{I} \big] \mathbf{x}. \tag{18}$$

In this expression, $\mathbf{x} = (x_1, \cdots, x_n)^T \in \mathbb{R}^n$ is a vector representing the "state" of the system, $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a symmetric matrix, and $\mathbb{I}$ is the $n \times n$ identity matrix. It can be shown that the equilibrium points for the differential equation 17, i.e. the points where $\dot{\mathbf{x}}(t) = 0$, correspond to the eigenvector of the symmetric matrix A[1]. This inclines us to define our cost function as:

$$\mathcal{C} = \Big[\dot{\mathbf{x}}(t) + \mathbf{x}(t) - f(\mathbf{x}(t)) \Big]^2. \tag{19}$$

Here, as the values inside the bracket is a vector, we interpret squaring this bracket as the scalar product of the vector with itself. As this cost function requires us to calculate a derivative, we are in need of a trial function to determine this value numerically. This is on the following form:

$$\mathbf{x}_{trial}(t,P) = e^{-t}\mathbf{x}_0 + (1 - e^{-t}) N(t,P) \tag{20}$$

The first term makes sure that initial conditions are satisfied, while the second term follows the somewhat expected structure of the solution of a first order ODE. After our neural network has trained for a couple of iterations until our cost function in equation 19 has reached an acceptable low value ($\implies$ high accuracy), we can conclude that our vector $\mathbf{x}$ corresponds to an eigenvector of our matrix $\mathbf{A}$. As shown by Yi et al., this eigenvector corresponds to the vector related to the largest eigenvalue of our matrix A.

Having determined the eigenvector $\mathbf{x}$ corresponding to the matrix $\mathbf{A}$, we can thus calculate the eigenvalue λ as:

$$\lambda = \frac{\mathbf{x}^T \mathbf{A} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} \tag{21}$$

Further, by training our neural network on the matrix $-\mathbf{A}$ instead of $\mathbf{A}$, the vector $\mathbf{x}$ will also converge to an eigenvector of the matrix $\mathbf{A}$. This eigenvalue can again be calculated using equation 21, and it has been shown by Yi et al. that this eigenvalue in fact is the minimum eigenvalue of our matrix $\mathbf{A}$.

III. ALGORITHM

A brief discussion of the numerical implementation of each of the method is to be discussed here. First for the explicit scheme, where we introduce the stencil representation, as well as look at the stability requirement of the method. This is followed by a minor discussion of free parameters to pick from in the neural network when solving for PDEs. Finally, we will briefly introduce how we will implement the eigenvalue solver with a neural network.

A. Explicit Scheme & Stencil Representation

To calculate the full expression $u(x,t)$ from the PDE in equation 3, we have, as already shown, discretized it into a set of points $u_{i,j}$. To calculate all of these points, we then simply use the relation as stated in equation 9. A nice illustration to visulize how this helps us calculate things can be seen in figure 1.

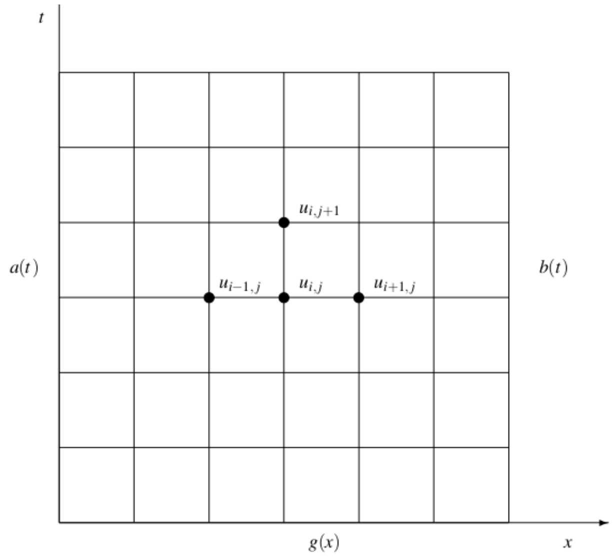

Figure 1. An illustration of how we will calculate for all (Figure from M. Hjorth-Jensen "Computational Physics" compendium, figure 10.1[4].)

The illustration seen in figure 1 is often called a stencil representation, or a calculational molecule. Our boundary terms $a(t)$ and $b(t)$ are locked by our boundary conditions, while $g(x)$ in this illustration is our initial condition, locking all of the temporal

Using our already-discussed stability limitation of $\alpha \equiv \frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$, and choosing either $\Delta x = 1/10$ or $\Delta x = 1/100$, the stability requirement therefore require that our temporal resolution is *at least* of size given in table I.

As is apparent from table I, this requires exponentially many more calculations for better resolutions of x, and as such scales quite badly.

Given Δx	Maximal Δt
$\frac{1}{10}$	$\frac{1}{200}$
$\frac{1}{100}$	$\frac{1}{20000}$

Table I. Limits for Δt given different Δx, as dictated by our stability requirement $\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$

B. Neural Networks for PDEs

The Neural Network we will be running will have 2 hidden layers, each of the hidden layers having 100 nodes. We will be running through 250 epochs trying to train our model, using a learning rate of $\lambda = 0.01$. Ideally, we would like to scout through several learning rates to find the ideal one, but due to the computational load (to be discussed in the discussion section), this was not a realistic endeavour.

As for the activation function, we will be using the Sigmoid function. This choice has just been made haphazardly, and is not justified other than being a relatively fine choice. Given that our expected values for the neural network is $u(x,t) \in [0,1]$ and Sigmoid $\in [0,1]$, this choice can be justified. Whether or not other activation functions are preferable will not be discussed here.

C. Neural Networks for Eigenvalue Problems

We will produce the real, symmetric matrix $\mathbf{A}$ in the same manner as discussed in the article by Yi et al. By use of a randomly generated, real matrix $\mathbf{Q}$, a real and symmetric matrix can be constructed as follows:

$$\mathbf{A} = \frac{1}{2}\Big(\mathbf{Q}^T + \mathbf{Q}\Big) \tag{22}$$

If $\mathbf{Q} \in \mathbb{R}^{n\times n}$, then we obviously have $\mathbf{A} \in \mathbb{R}^{n\times n}$. As such, to generate a 6×6-matrix $\mathbf{A}$, we need $\mathbf{Q} \in \mathbb{R}^{6\times 6}$.

As for the design of the hidden neurons of the Neural Network, we will here also use 2 hidden layers, each consisting of 100 neurons which will be fully connected. The input layer will be of size N and the output layer will be of size N as well, where N corresponds to the length of the vectors $\mathbf{x}$. As such, we will update both the vectors $\mathbf{x}$ when running through the neural network, but the neural network's weights and biases as well. Having done this for a couple of iterations, we expect that the vector will converge to an eigenvector of A. This is clearly an RNN (Recurrent Neural Network).

IV. RESULTS

This results section will be split into ones discussing the numerical solutions to the Partial Differential Equation, namely the heat equation, and one discussing the non-linear differential equation with eigenvalue solving properties.

A. Partial Differential Equations

The presentation of the results for the PDE will be split into the explicit scheme results, and the results from the neural network.

1. *Explicit Scheme*

First, we present the results using the explicit scheme in the following figures 2-6.

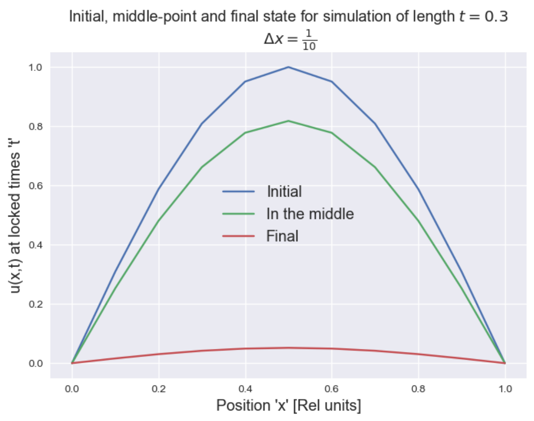

Figure 2. $u(x,t)$ for 3 different times t using the explicit scheme with $\Delta x = \frac{1}{10}$. The blue curve is the initial condition at $t = 0$, while the red curve is at $t = 0.3$. The green curve in the middle is at an arbitrary time relatively close to $t = 0$. For this simulation, we have chosen Δt as close to the stability criteria as possible.

In figure 2, we have plotted the function value of $u(x,t)$ for different times t, first at the initial time, followed by an arbitrary time in the middle, and finally at the simulations end. Here, we have chosen $\Delta x = \frac{1}{10}$ and $\Delta t = \frac{1}{200}$.

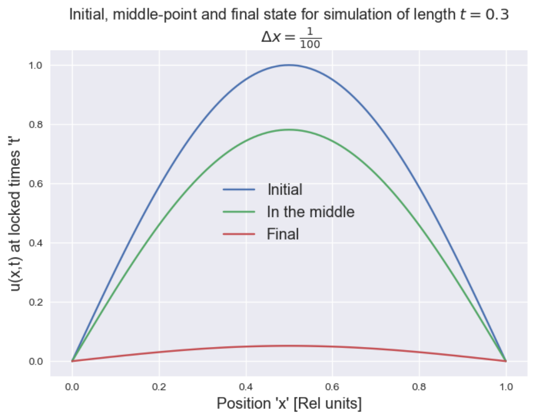

Figure 3. $u(x,t)$ for 3 different times t using the explicit scheme with $\Delta x = \frac{1}{100}$. The blue curve is the initial condition at $t = 0$, while the red curve is at $t = 0.3$. The green curve in the middle is at an arbitrary time relatively close to $t = 0$. For this simulation, we have chosen Δt as close to the stability critera as possible.

In figure 3, we have plotted the function value of $u(x,t)$ for different times t, first at the initial time, followed by an arbitrary time in the middle, and finally at the simulations end. Here, we have chosen $\Delta x = \frac{1}{100}$ and $\Delta t = \frac{1}{20000}$.

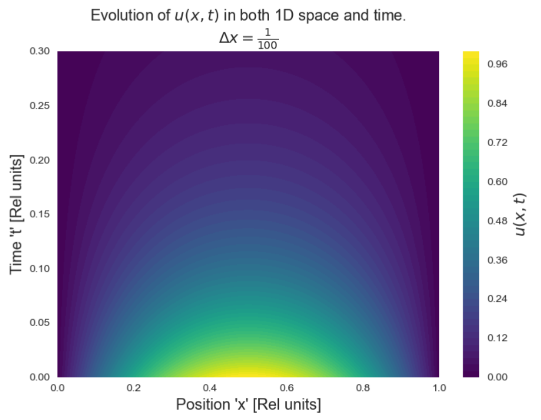

Figure 4. A color contour plot of the calculated values of $u(x,t)$ using the explicit method with $\Delta x = \frac{1}{100}$. Time t increases as we move up in this plot. For this simulation, we have chosen Δt as close to the stability criteria as possible.

In figure 4, we have plotted a color contour plot of function values $u(x,t)$ for an entire plane of point values (x,t), calculated using the explicit scheme. Here, we have chosen $\Delta x = \frac{1}{100}$ and $\Delta t = \frac{1}{20000}$.

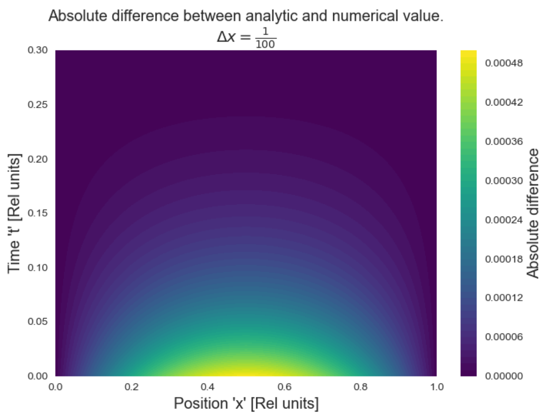

Figure 5. A color contour plot of the absolute difference between the analytic and numerically calculated values of $u(x,t)$, using the explicit method with $\Delta x = \frac{1}{100}$ for the latter. For this simulation, we have also chosen Δt as close to the stability criteria as possible.

In figure 5, we have plotted a color contour plot of the absolute difference of the function values $u(x,t)$ calculated using the explicit scheme (seen in figure 4), and the true analytic function value (as shown in equation 4). We observe that the largest differences to the analytic values can be seen at the points where $u(x,t)$ has the largest gradient / largest change between function values. The absolute difference is however small, at ~ 0.0001 in orders of magnitude.

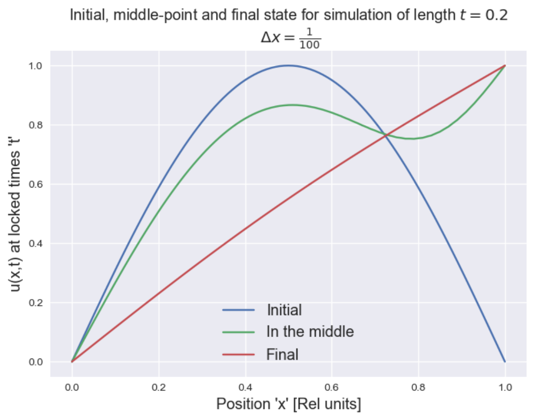

Figure 6. $u(x,t)$ for 3 different times t using the explicit scheme with $\Delta x = \frac{1}{100}$, but this time with the boundary condition $u(x=1,t) = 1$ instead. The blue curve is the initial condition at $t = 0$, while the red curve is at $t = 0.2$. The intermediate green curve is set at an arbitrary time relatively close to $t = 0$. For this simulation, we have chosen Δt as close to the stability criteria as possible.

In figure 6, we have a similar plot as in figure 2, but having changed the boundary condition at $x = 1$ to $u(x = 1,t) = 1$. This was simply added to see if the behaviour of the model behaved as expected for other values of boundary conditions.

2. Neural Networks

The specific neural network we trained was 2 layers deep, with each layer having 100 nodes, and being fully connected. We used a learning rate of $\lambda = 0.01$, running through 250 iterations of 400 data points to train on. The results presented in figures 7 and 8 is only for this one case of a neural network.

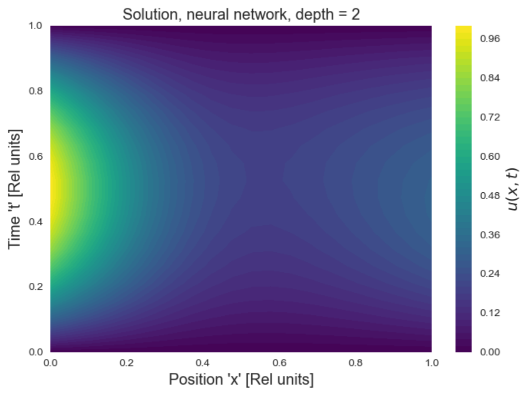

Figure 7. A color contour plot of $u(x,t)$ according to our neural network.
NOTE: the y-axis and x-axis is interchanged, and time is on the x-axis, while position is on the y-axis. It would have taken another 50 minute computation to adjust this, so it was left in.

In figure 7, we see our neural network's attempt at recreating the actual function value at all points (x,t). We see that the initial drop in time is less steep than the correct one (seen in figure 4), and it also seems to increase in value when t grows large, which also seems to contradict the expected behaviour of this function.

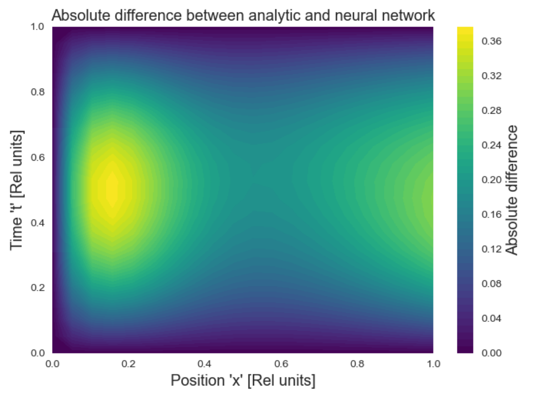

Figure 8. A color contour plot of the absolute difference between the analytic value of $u(x,t)$ and the neural network approximation.
NOTE: the y-axis and x-axis is interchanged, and time is on the x-axis, while position is on the y-axis. It would have taken another 50 minute computation to adjust this, so it was left in.

In figure 8, we see the absolute difference between the analytic solution and our neural network's attempt at approximating it. Right off the bat we can see that the absolute error is larger here than for the explicit scheme method, getting as large as ~ 0.36. For an analytic solution $u(x,t) \in [0,1]$, that is a quite large potential error.

Method	**Time spent**	**$\sim$ Max Error**
Explicit, $\Delta x = \frac{1}{10}$:	$\sim$ 1 millisecond	~ 0.05
Explicit, $\Delta x = \frac{1}{100}$:	$\sim$ 1 second	~ 0.0005
NN, 250 iterations:	49:32min	~ 0.36

Table II. Comparison between methods in terms of simulation length ("Time spent") and the approx max error of each method.
NOTE: This was only done for 1 value of learning rate $\lambda = 0.01$, and as such is potentially not the ideal one. The time to scout the parameter space with this computational load was too time consuming at the time of writing to optimize for.

In table II, we have a direct comparison between the explicit scheme and our specifically trained neural network. When all is said and done, the explicit scheme wins over the neural network for our choice of neural network parameters. This is not to say that this will always be the case, but it is the case for our specific model.

B. Eigenvalue problems

Ahh yes, considering the eigenvalue solver, the program itself proved too difficult to properly put together. The program was mostly based on the program of solving the PDE above, but with the added complexity that 6 nodes in the input and output layer layer brought forth were several bugs. In solving these bugs, new bugs appeared which made the class structure more messy and the bugs grew insurmountable after a while.

If we were successful in making the logic of the program work out, we would ideally attempt to create a plot somewhat similar to those presented in the article by Yi et al. This would be followed by a comparison of this method to ones of numerical diagonalization, or other eigenvalue-finding methods from linear algebra, to compare these both in terms of efficiency, accuracy, and how well they would scale for larger symmetric matrices. We could also potentially test some borderline things like non-symmetric matrices, or almost symmetric-matrices, to see if the stability holds for these cases as well.

The program, however little it actually functions, is however possible to debug, placed neatly onto the GitHub repository linked at the top of this report.

V. DISCUSSION

We will first discuss the solving of our PDE, both using the explicit scheme and neural network. This is followed by a discussion of the neural network approach to solving eigenvalue problems.

A. A PDE Solving Comparison Between Neural Networks & Regular Methods

This will introduce a small discussion around the 'regular method' brough to us by the explicit scheme. This is followed by a discussion of the advantages and disadvantages of using a neural network to solve differential equations.

1. Explicit Scheme

First and foremost, when discussing the explicit scheme, one need to address the elephant in the room. The big bad wolf, or rather the most prominent thing that we need to address using the explicit scheme is one of scaling. Namely the stability requirement of the explicit scheme is an issue here. If we wish to increase the spatial resolution of our simulation, we would have to (...). Luckily, there are other methods, for instance the Crank-Nicolson scheme[5], which do not suffer from the same limitations of the explicit scheme does in terms of stability.

A final notion we can look at to affirm that our explicit scheme solution is working as intended is to look to our results in figure 6. In this figure, we have changed up the boundary conditions such that at $x = 1$, we have $u(x = 1, t) = 1$. We should as such expect a smooth transition from our half-sine wave to a linear function in the scope of our simulation. Sure enough, our function $u(x, t)$ seems to stabilize to a linear function $u(x, t) = x$ for $t >> 0$, which is the expected behaviour. As such, we are fairly certain that this scheme is working correctly, qualitatively speaking.

2. Advantages and Disadvantages of Neural Networks

In terms of applying a neural network to solving differential equations, we have mixed feelings from these results. Looking

One of the main advantages we can initially observe by using a neural network as opposed to using the explicit scheme is one of flexibility. Our explicit scheme will promptly calculate values for a set of points (x, t), but none of the points in between. This means that if we wish to extract the function value $u(x, t)$ between any of the points we have calculated, we would have to interpolate between these points to approximate our value. For this specific calculation, this will likely not induce a large error. The error could however be significant for another choice of initial condition, boundary condition, or PDE in general. For our neural network, we are free to feed it any value for x and t separately as we like, and the network will return a function value $u(x, t)$ without a hitch. Given that the values for x and t that we pick are within the ranges of the values that we have trained the network in, this can be done without a problem. Picking a point (x, t) outside the range that the neural network has actively trained on could however create some problems in terms of extrapolation, and should be taken into consideration when applying a neural network in solving a PDE.

One disadvantage one however must address is one of training time. Using our program, training our network for 250 iterations on 400 different points ends up at a grand total of 50 minutes of run time. Looking at figure 8, it is apparent that we would need more data points, especially near the largest gradients, to drive the error further down. This would in turn require more computations, and a slower program overall. One should also note that this specific program uses both a high-level programming language (Python), and has not been prone to particular optimizing that could drive down computational time. As such, this program might not be representative of the optimal approach. However, comparing this to the explicit scheme, it is several orders of magnitude slower in computations.

However, given that the program gets to run for a longer amount of time, optimally for even more data points, we should eventually achieve a neural network solution that is comparable in accuracy as compared to the explicit scheme, and even beyond this. When this has been achieved, the parameter values defining both the bias and weights for all of the neurons can be promptly saved for further use. Once the network has been trained to a satisfying result for a given PDE with boundary and initial conditions, you don't need to train it again given that these parameters are saved for future use.

One must also take into consideration that there are extremely many factors that goes into the design and parameter choice of a neural network. For a start, one could potentially determine to use another activation function on some (or all) of the neurons, instead of our approach of using the sigmoid function. In addition to this, just having the freedom of deciding both the depth and width, and eventually the structure of the neural network could come into play as to whether or not our network will converge to the correct value, eventually at what speed. Higher complexity of the network means a larger degree of freedom to adjust and fit the data, but necessarily at a computational cost.

In addition to this, we have the added complexity brought forth by the learning rate λ. This is the single most important parameter to adjust. Having a too small λ would make convergence of our network extremely slow, while a large value of λ could potentially make our model worse at predicting the given function $u(x,t)$, i.e divergent properties. In addition to this, one could also introduce other regularization parameters that potentially could further improve performance, however at an even larger computational cost. Scouting the parameter space of learning rates, regularization parameters, and finding the ideal dimensions of the hidden layers, with the large computational cost we already have testing for one set of parameters, is an extreme computational cost when compared to the simpler, more straightforward methods already discussed.

B. Non-Linear Differential Equations, Neural Networks & Eigenvalue Solvers

The discussion here is mostly based on the "what-if" our program worked approached, and only skims general questions, setbacks, and advantages to using our method in determining eigenvectors and eigenvalues of a symmetric matrix. Had our program worked properly, we could have potentially dug deeper into this discussion.

First and foremost, the most prominent setback that our method of solving for eigenvalues and eigenvectors has, is the fact that we can only find the maximum and minimum eigenvalue, and the corresponding eigenvector to these, when using this method. For our particular case of $\mathbf{A} \in \mathbb{R}^{6\times6}$, by finding the maximum and minimum eigenvalue and corresponding eigenvectors, we will still have 4 more eigenvalues and eigenvectors to compute. As such, our solution will be incomplete. However, for many purposes in linear algebra, it is the largest and / or smallest eigenvalue which are of importance, and we need not calculate the rest.

Another issue one could encounter is one where the initial vector $\mathbf{x}(0)$ is orthogonal to the maximum or minimum eigenvector of our matrix $\mathbf{A}$. If this occurs, the convergence of $\mathbf{x}(t)$ to the corresponding eigenvector can potentially not occur. This has not been tested, but it could prove interesting to see if the convergence still occurs when $\mathbf{x}(0)$ is orthogonal to the eigenvector corresponding to the maximum or minimum eigenvalue of $\mathbf{A}$.

As is discussed in the article of Yi et al.[1], it is unknown if attempting to perform this algorithm on a non-symmetric matrix will give the same results. The mathematical proof only applies for a symmetric and real matrix, and if we are to limit ourselves to just these types of matrices, we can only find eigenvalues and eigenvectors to a very limited subset of all matrices. However, as the article discussing this was published back in 2004, there is bound to have been progress in determining the limits to this algorithm. In addition, one could also very well find a similar non-linear differential equation which has the added benefit of finding eigenvalues of a more general matrix. This dive into the literature, or potential test of a non-symmetric matrix using our solver, has not been conducted. (The latter for the obvious reason that our solver was non-functioning).

VI. CONCLUSION

In terms of results, the neural networks haven't impressed thoroughly as compared to standard methods of linear algebra or differential equation solving. For the PDE case, it has somewhat worked out, giving us mediocre results at best, while for the matrix case, our program is not working at all.

In terms of our runs solving the PDE, there is no doubt that the explicit scheme blows the neural networks' out of the water, both in terms of accuracy, computational speed, and simplicity of implementation. However, this is only the case for our specific choice of parameters, and scouting for a more optimal fit of parameters in all of parameter space proved a task too much. If the time and computational power would have been available to search the parameter space more thoroughly for optimal parameters, one would potentially and probably find a comparable, if not better, fit of the solution for $u(x, t)$. If such a fit is found, the network parameters could be stored away, and the network could be used indefinitely and speedily without the need for further training.

In terms of the eigenvalue solver, we obviously have no results to directly point to, and as such, our only conclusion could really be to have none at all. It is sub-optimal for a method to be of such complexity that to create a proper solver is way more difficult than using many of the standard methods of linear algebra. However, on the offchance that this program had worked properly, it would have been interesting to see how this method would stack up both in terms of accuracy and efficiency as compared to standard methods one otherwise would use. Especially in the limit of larger matrices than 6×6.

Overall, for such relatively simple systems that we have looked closer at in this report, neural networks don't impress all that much. This is somewhat expected however, as neural networks tend to be more impressive at finding complex structures, and is not all that impressive at simple problems, when compared to simple algorithms. In a sense, neural networks are expected to scale well where other methods don't, but at this low of complexity, it is not expected to prosper especially well compared to other methods.

[1] Z. Yi, Y. Fu, and H. J. Tang, *Neural Networks Based Approach for Computing Eigenvectors and Eigenvalues of Symmetric Matrix*, Computers and Mathematic with Applications **47**, (2004), 1155-1164.

[2] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2013), p. 307.
Course URL: https://www.uio.no/studier/emner/matnat/fys/FYS4150/index-eng.html.

[3] M. M. Carstensen, G. S. Cowie, and J. E. Ødegård, *A Comparison of Neural Networks to Linear and Logistic Regression for Regression and Classification Problems*, University of Oslo (2020).
URL: https://github.com/Jan-Egil/FYS_STK4155/blob/master/proj2/PDF/Project2_Handin.pdf.

[4] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2013), p. 306.
Course URL: https://www.uio.no/studier/emner/matnat/fys/FYS4150/index-eng.html.

[5] The Crank Nicolson scheme is properly discussed elsewere. Wikipedia is a fine start: https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method (Dated 17/12/2020).

[6] MIT Lecture Notes on PDE'S and the solution of the Heat Equation. Fall 2006. Ch. 2 Author: Matthew J. Hancook. URL: https://ocw.mit.edu/courses/mathematics/18-303-linear-partial-differential-equations-fall-2006/lecture-notes/heateqni.pdf.

Appendix A: Analytic Solution to Partial Differential Equation

This solution follows to a large degree that of the lecture notes on PDEs given at MIT, and the derivation might look quite similar to this. [6].

We will look closer at the following PDE:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \text{ where } t > 0 \text{ and } x \in [0, L] \tag{A1}$$

We pick our initial condition to be:

$$u(x, 0) = \sin(\pi x) \ \ , \ \ 0 < x < L \tag{A2}$$

We also set our boundary conditions to be:

$$u(0,t) = u(L,t) = 0 \ \ , \ \ t \geq 0 \tag{A3}$$

We will for simplicity assume $L = 1$, and scale everything thereafter. To solve this PDE, we will assume that the solution is of separable form. That is, It can be written on the form $u(x,t) = X(x)T(t)$. Doing this, our equation will be on the form:

$$\frac{\partial^2}{\partial x^2}\big[X(x)T(t)\big] = \frac{\partial}{\partial t}\big[X(x)T(t)\big] \implies T(t)\frac{\partial^2 X(x)}{\partial x^2} = X(x)\frac{\partial T(t)}{\partial t} \tag{A4}$$

We rewrite this such that everything that is dependent on t is on one side of the equality and everything depending on x is on the opposite side:

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)}. \tag{A5}$$

Here, the prime denotes the derivative with respect to the corresponding variable. If we were to vary time or position separately, the derivative with respect to the other variable must be kept fixed. As such, we can assume that these both equal a constant, which we denote as such:

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)} = -\lambda \tag{A6}$$

Where we pick the minus sign for calculational convenience. This can as such be separated into two independent Ordinary Differential Equations for $X(x)$ and $T(t)$ respectively. These are:

$$X''(x) + \lambda X(x) = 0 \tag{A7}$$

$$T'(t) + \lambda T(t) = 0 \tag{A8}$$

First, we attempt to solve for $X(x)$. The solution to this depends on whether $\lambda > 0$, $\lambda < 0$ or $\lambda = 0$. For the two latter cases ($\lambda = 0$ & $\lambda < 0$), we only get trivial solutions for $X(x)$ (that is, $X(x) = 0$ for all t). That is not compatible with our initial condition, so we discard this. We then solve this for $\lambda > 0$, which gives us the solution on the form:

$$X''(x) + \lambda X(x) = 0 \implies X(x) = A\cos(\sqrt{\lambda}x) + B\sin(\sqrt{\lambda}x) \tag{A9}$$

Implementing the boundary conditions to this expression implies that $X(0) = X(1) = 0$. This is mainly because this needs to be valid for all t, so the boundary conditions only apply for the spatial equation. This gives:

$$X(0) = A\cos(\sqrt{\lambda}0) + B\sin(\sqrt{\lambda}0) \overset{!}{=} 0 \implies A = 0 \tag{A10}$$

$$X(1) = A\cos(\sqrt{\lambda}1) + B\sin(\sqrt{\lambda}1) \overset{!}{=} 0 \implies B\sin(\sqrt{\lambda}) = 0 \tag{A11}$$

For the final expression, we have implemented that $A = 0$ shown in the first expression. For this to hold true without getting trivial solutions (which is not compatible for our initial condition), we $\sin(\sqrt{\lambda}) = 0$. This is the case if $\lambda = n^2\pi^2$ for $n \in \mathbb{N}$. The constant $B \to b_n$ then gets a dependence on n, given that this now becomes a superposition of sine-functions. Our final solution for $X(x)$ thus becomes:

$$X(x) = b_n \sin(n\pi x) \tag{A12}$$

Here we have that an explicit sum over all n is implied in the notation.

Further, we need to solve for $T(t)$. This has the form:

$$T'(t) + \lambda T(t) = 0 \tag{A13}$$

Implementing our newfound knowledge on $\lambda \to \lambda_n = n^2\pi^2$, this becomes:

$$T'(t) + n^2\pi^2 T(t) = 0 \tag{A14}$$

This ODE has a simple solution with respect to the exponential function, which is:

$$T(t) = c_n e^{-n^2\pi^2 t} \tag{A15}$$

Where c_n is the integration constant for each choice of n. Gathering up our solutions for $T(t)$ and $X(x)$ back into $u(x,t)$, this becomes:

$$u(x,t) = X(x)T(t) = b_n \sin(n\pi x) c_n e^{-n^2\pi^2 t} \tag{A16}$$

We can combine the constants together to form a new constant $B_n = b_n c_n$:

$$u(x,t) = B_n \sin(n\pi x) e^{-n^2\pi^2 t} \tag{A17}$$

We now look to our initial condition $u(x,0)$ to lock in the constants B_n. Writing this down gives us:

$$u(x,0) = B_n \sin(n\pi x) \overset{!}{=} \sin(\pi x) \tag{A18}$$

This trivially only applies when $B_1 = 1$ and $B_{n\neq 1} = 0$. Having this determined, we have our full solution $u(x,t)$ to be:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t} \qquad \text{Where } \ t > 0 \ \text{ and } \ x \in [0,1] \tag{A19}$$

$\square$