Solving a diffusion equation using neural networks

Sakarias Frette, William Hirst, Mikkel Metzsch Jensen

(Dated: December 3, 2021)

CONTENTS

I. INTRODUCTION

Differential equations are a cornerstone in understanding the physical processes around us. From the Schrödinger equation, which explains low energy quantum mechanical systems, to the Navier-Stokes equations that governs flow of fluids, they explain how the system impacts itself through change with respect to time, space or both. Traditional numerical solutions such as finite difference[4] and finite element[5] have yielded good results, and they are computationally fast, but they lack in one aspect. They base themselves on discretization, and so the solution you get is also discretized. Thus, you do not get a function with which can be evaluated at any given point. with the Neural ordinary differential equation paper[1] from 2018 and other papers, it was shown that neural networks also can solve both ODE's and PDE's. With these networks one can get a actual function that can be evaluated, though at a potential cost of computation time.

It was also shown in 2003[6] that neural networks can be used to solve eigenvalue problems. Eigenvalue problems occur in many physical phenomenon, such as the Schrödinger equation, where the energy of the system is the eigenvalue of the Hamilton operator, or in vibration analysis, where the eigenvalue corresponds to the eigen-frequency.

This is the motivation for this report, where we will use both finite difference and machine learning to solve the diffusion equation and compare computation time and error, as well as compute eigenvalues for a given matrix using machine learning.

The report is structured in the sections theory, implementation results and discussion, and conclusion. The theory includes the mathematics behind the diffusion equation, the finite difference method, neural networks for solving PDE's and eigenvalue problems using neural networks. The implementation includes the pseudo code algorithms for the finite difference method, the neural network PDE solver, as well as some of the choices used to solve the eigenvalue problem. The results and discussion includes results from the finite difference method, the neural network for the PDE and the eigenvalue problem, as well as discussion comparing the methods. We finally summarize our findings in the conclusion.

II. THEORY

A. The diffusion equation

The generalized diffusion equation is given as

$$\frac{\partial}{\partial t}u(\mathbf{x},t) = \nabla[\alpha(u,\mathbf{x},t)(\nabla u(\mathbf{x},t))],$$

where u is the solution, **x** is a 3D vector containing spacial coordinates, t is temporal coordinate, and alpha is a function comprised of the temporal and spacial derivative constants and possibly the solution u. For this project we will look at a simpler example, namely the 1D diffusion equation with $\alpha = 1$. It is given as

$$\frac{\partial^2}{\partial x^2}u(x,t) = \frac{\partial}{\partial t}u(x,t), \tag{1}$$

and in our case is defined in the domain of

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

for $t > 0$, with the following initial condition

$$u(x,0) = \sin(\pi x)\ ;\ 0 < x < L,$$

and with Zero-Dirichlet boundary conditions

$$u(0,t) = u(L,t) = 0\ ;\ t \geq 0.$$

1. *Analytical solution*

This equation has an analytical solution, which we will use to test our network's solution. This is found by analysing the equation and "guessing" on a general form. From our equation we note that we need a function that is its own second derivative, namely a sine or cosine function. We also note that our initial condition is a sine function, which supports that guess. We also need a function that does not change when using a time derivative operator on it, and a exponential function is thus a good guess. we then have

$$u_e(x,t) = A\sin(kx)e^{-\omega t}.$$

Here we have three degrees of freedom, A, k and ω. A is the amplitude of the sinusoidal function, k is the angular frequency of the spacial domain and ω is the rate of decay. We first find ω by putting our guess function into equation 1:

$$\frac{\partial^2 u_e(x,t)}{\partial x^2} = -k^2 A\sin kx e^{-\omega t},$$

$$\frac{\partial u_e(x,t)}{\partial t} = -\omega A \sin{(kx)} e^{-\omega t}.$$

This relation gives us

$$\omega = k^2.$$

Next we implement the Zero-Dirichlet boundary condition:

$$A \sin{(k \cdot 0)} e^{-k^2 t} = A \sin{(kL)} e^{-k^2 t} = 0,$$

$$\sin{(kL)} = \sin{(k \cdot 0)},$$

$$kL = \arcsin 0 = N\pi,$$

$$k = \frac{N\pi}{L}.$$

Finally we implement the initial condition

$$A \sin{(\frac{\pi}{L}x)} e^{(\frac{\pi}{L})^2 \cdot 0} = \sin{(\pi x)},$$

where we choose the length of the rod L to be 1, so we get that

$$A \sin{(\pi x)} = \sin{(\pi x)}.$$

Thus $A = 1$. Our final, analytical solution is then

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

2. *Finite difference discretization*

To solve this equation numerically we need to use discretization. One such method is finite difference, where we use finite approximations to the derivatives. The simplest case here is to use the so-called explicit Forward Euler algorithm, with forward difference on first order derivatives, and center difference on second order derivatives.

By convention we will use subscript indexing for spacial coordinates and superscript indexing for temporal coordinates, where j is the index to the numerical solution for time, and i is the index for spacial coordinate. This convention is somewhat arbitrary, but helps if one extends the diffusion equation to more dimensions in space. We will also use the convention that $u(x_i, t_j) = u_i^j$, where x_i is the i'th element in space, and t_j is the j'th element in time.

Thus, the first order derivative with respect to time in equation 1 is given as

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u_i^{j+1} - u_i^j}{\Delta t}.$$

The second order derivative with respect to space uses center difference, and is given as

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

Assembling these two discretizations together we get that

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}.$$

$$u_i^{j+1} = u_i^j(1 - \frac{2\Delta t}{\Delta x^2}) + \frac{\Delta t}{\Delta x^2}(u_{i+1}^j + u_{i-1}^j). \tag{3}$$

This equation has a stability criteria which must be satisfied for the numerical solution. The general criteria, formalised by von Neuman[2], is given as

$$\frac{\alpha \Delta t}{\Delta x^2} \leq \frac{1}{2}, \tag{4}$$

where α is the constant multiplied with the spacial part of the diffusion equation. In our case $\alpha = 1$, which allows for reasonably small step sizes, but for large α this method requires too small step sizes for practical use.

B. Neural networks and PDE's

1. *Trial function*

To solve this problem we need to contruct a trial function, $g_t(x,t) \approx u(x,t)$, defined as

$$g_t(x,t) = h_1(x,t) + h_2(x,t) N(x,t,P), \tag{5}$$

where $N(x,t,P)$ is the data from the neural network, $h_1(x,t), h_2(x,t)$ are functions that ensures that the boundary and initial conditions are kept. We have zero-Dirichlet conditions on the boundaries, so a function like $x(1-x)$ ensures that the function is zero at the boundaries. We also have an initial condition, so the final trial function is thus

$$\begin{aligned} g_t(x,t) &= h_1(x,t) + h_2(x,t) N(x,t,P) \\ &= (1-t)\sin{(\pi x)} + x(1-x) t N(x,t,P). \end{aligned} \tag{6}$$

2. *Cost function*

When using supervised learning one usually has a well defined cost function testing the model against the exact data. But in the case of solving PDE's, it is not given that one has access to an exact solution to compare against. We thus introduce a different cost function. The cost function uses the residual of the differential equation to calculate the loss. The residual is defined as putting all the terms on one side of the equation, and in the case of the diffusion equation, we have that the residual R is

$$R = \frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2} = 0. \tag{7}$$

We then square the residual and take the mean, giving us the cost function

$$C(x,t) = mean(R^2) = mean\left(\left[\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2}\right]^2\right). \tag{8}$$

Thus, our problem has become a minimization problem with respect to the calculation it self.

C. Finding eigenvalues through a neural network

The work of Yi et al. [6] proposes a method of using a neural network to find eigenvectors of a symmetric matrix. The idea is to define the eigenvectors of A as solutions of a simple ODE. In this section we will in short terms describe some of the essential equation and calculations needed to implement sed method.
For a matrix, A and a constant, λ the eigenvectors of A, v are defined by the following equation

$$Av = \lambda v. \tag{9}$$

From 9 it follows that

$$X^T XAX - X^T AX^T X = 0, \tag{10}$$

if and only if X=v or X = 0. And again it follows from 9 and 10 that

$$\lambda = \frac{X^T AX}{X^T X}. \tag{11}$$

As described in [6], the goal of the method is to introduce 10 into our ODE, and consequently therefore into the cost function of the network, such that we find the eigenvectors of A. But, as is obvious from 10, the network will far to easily gravitate towards the trivial solution. Therefore we instead suggest a more dynamical suggestion as our ODE. [6] suggests the following ODE

$$\frac{dX(t)}{dt} = f(x) - X(t), \tag{12}$$

where

$$f(X) = [X^T XA - (1 - X^T AX)I]X. \tag{13}$$

A is a symmetric matrix, I is the identity matrix and t is just a variable.
From 12 we can study the dynamics of the length of X(t) as

$$\frac{d(X^T X)}{dt} = 2X(t)\frac{dX(t)}{dt}. \tag{14}$$

Given that we want X(t) to equal a eigenvector of A, we know from 10 and 12 that 14 = 0. In otherwords we have that $X(t)^T X(t)$ must be constant for all t, $X^T X = X(0)^T X(0)$. From this we can write our ODE as

$$\frac{dX(t)}{dt} = X(0)^T X(0)AX - X^T AX^T X, \tag{15}$$

At this point we state the following theorem

Theorem 1. *Given an initial state, X(0) every solution of 15 will converge towards an eigenvector of A corresponding to the largest eigenvalue, v_{max} when $t \to \infty$ given that the following is true*

- *X(0) is non-zero,*
- *X(0) is not orthogonal to the v_{max}.*

For the proof of this theorem we refer again to the text of Yi [6], specifically theorems 3 and 4 on pages 1159 and 1160.

III. IMPLEMENTATION

The code can be found on Github.

A. Solving a PDE with an explicit solver

Before we solve our PDE with a neural network, we first want to solve it using an explicit solver. The solver sets up a recursive algorithm which loops over each timestep, and for each time-step calculates 3 over all values of x_i. The explicit solution was calculated using algorithm 1.

Algorithm 1: Simulate system using vectorized solution.

```
1  Initialize complete solution array, $u_{complete}$ ;
2  Initialize domain;
3  Initialize array for solution at t, u ;
4  Initialize array for solution at $t - \Delta t$, $u_1$;
5  Insert initial condition ;
6  for $1 \rightarrow t_N$ do
7      Calculate the inner spacial points for u;
8      Insert boundary conditions;
9      Update u, $u_1$;
10     Insert solution to $u_{complete}$;
11 end
12 Return $u_{complete}$
```

Note here the use of three arrays, namely the final solution $u_{complete}$, u and u_1. $u_{complete}$ stores the solution for every combinations of x and t. u_1 stores the values of the previously loop and is set equal to the initial condition, II A. u stores the values calculated in the current iteration.

B. Solving a PDE with an neural network

1. The algorithm

When developing a class to solve the PDE in equation 1, we decided to use TensorFlow infrastructure. The main structure of the code is divided into three parts; creating the model, training of the model and tracking the process.
The model structure is shown in table I. The training is a executed with a custom training loop. The necessity of a custom loop rises from the fact that there is no traditional training data and the loss-function must be customized for our PDE. When training the model we used an ADAM optimizer III B 2, which used the derivative of the loss 8, where the same loss is used to track the process. The loss value is derived with respect to the trainable variables, i.e weights and biases which gives us the gradient for the model. Algorithm 2 displays the general algorithm of the class.

Algorithm 2: Training of the network.

```
1  Initialize model ;
2  for epoch in epochs do
3      Calculate the loss and the gradient of the loss
         with respect to hidden variables;
4      Apply gradient to model;
5      Store loss;
6  end
7  Return total loss
```

2. Optimizers

The code uses the ADAM[3] optimizer. This method uses an adaptive momentum. The ADAM optimizer is ideal in situations where the model gets stuck at a local minimum. Given such a situation the ADAM optimizer applies a small increase in the learning rate in an attempt to "jump" out of the minimum.

3. Data

The data set here has two features, namely position, x, and time, t. As we want the network to be valid for all values of both x and t, we need to train on every possible combination of the two features. To do this, we create a meshgrid, as shown in array 16.

$$\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix}, \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{bmatrix} \rightarrow \left.\begin{bmatrix} [t_1, x_1] \\ [t_1, x_2] \\ \vdots \\ [t_2, x_1] \\ [t_2, x_2] \\ \vdots \\ [t_N, x_M] \end{bmatrix}\right\} (M \cdot N). \tag{16}$$

Equation 16 displays how we expand and stack our data, creating the $2 \times (M \cdot N)$ matrix for t_i and x_j with $i \in \{1, 2, .., N\}$ and $j \in \{1, 2, .., M\}$.

4. Structure of the network

To solve the PDE and the eigenvalue problem, we structured the data as shown in table I
This number of layers and nodes where not arbitrarily chosen. We tested for multiple combinations but found that this combination gave the best final loss.

C. Eigenvalue and eigenvectors

In calculating the eigenvalues and eigenvectors of a matrix A, we must first design our matrix. In this report

Table I: Each layer has a given set of nodes, with number of tunable parameters and an activation function

Layer	Nodes	Params	Activation
1	20	60	Sigmoid
2	20	420	Sigmoid
3	20	420	Sigmoid
4	1	21	None

we will limit ourselves to a symmetric 6×6-matrix. The matrix was designed by randomly generating a 6×6-matrix, Q with the values sampled from a "standard normal" distribution. To insure that the matrix is symmetric, we define the following matrix

$$A = \frac{Q^T + Q}{2}. \tag{17}$$

From this point the main points of the method of solving the ODE, 12 are the same as those discussed in section III B.
The main distinction from III B are the sizes of the first and last layers and the cost function. For the PDE case the input was two-dimensional (for x and t) and the output was one. In our ODE case we only have one variable, t and our output is a 6-dimensional vector. Therefore the input is one-dimensional and the output is 6.
As for the cost function we will again use the residual of our equation. In the case of our ODE it is simply the unequivalent in the right side (RS) and the left side (LS) of 15, ie RS-LS.
Finally we must also address certain criteria in theorem 1. Firstly, as we have no pre-knowledge on v_{max}, there is no way of guarantying that our choice of X(0) will not be orthogonal to v_{max}. Therefore we instead randomly generated the values of X(0), again sampling from a "standard normal" distribution. Secondly, we must also only approximate the convergence for $t\infty$. We do so by defining our t as $t \in [0, 10^{15}]$.

IV. RESULTS AND DISCUSSION

A. Diffusion Equation

In this section we will by studying the diffusion equation discussed in section II A.

1. Finite Difference

In this section we have studied the explicit solution of equation 1, described in section II A 2. We are interested in testing our solver for two spatial step-sizes: 0.1 and 0.01. The temporal step-size is calculated using the von Neuman condition 4. The result from using finite difference discretization for $dx = 0.1$ are shown in figures 1.

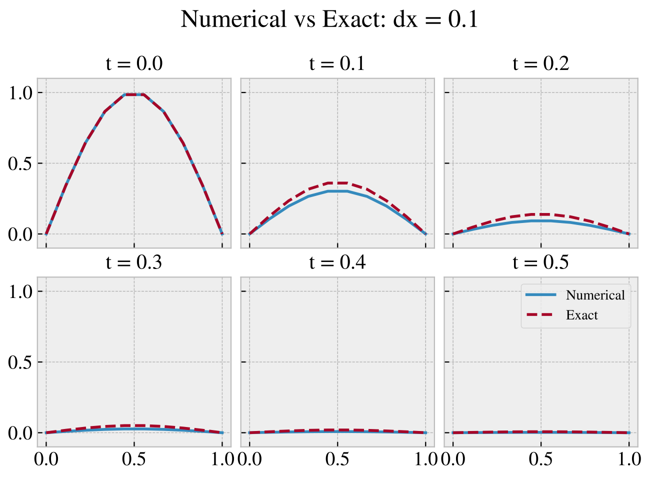

Figure 1: Explicit solution vs exact solution for diffusion equation with $dx = 0.1$, $dt = 5e^{-5}$, $L = 1$ and $T = 0.5$.

In figure 1 we display the time evolution of the numerical solution against the exact solution as function of space in 6 different time-steps. From the figure we observe that the numerical approximation is accurate for t=0 and for t=0.5, but struggles between the initial and final state. The numerical solution decays faster than the exact solution.
The result from using finite difference discretization for $dx = 0.01$ are shown in figures 1.

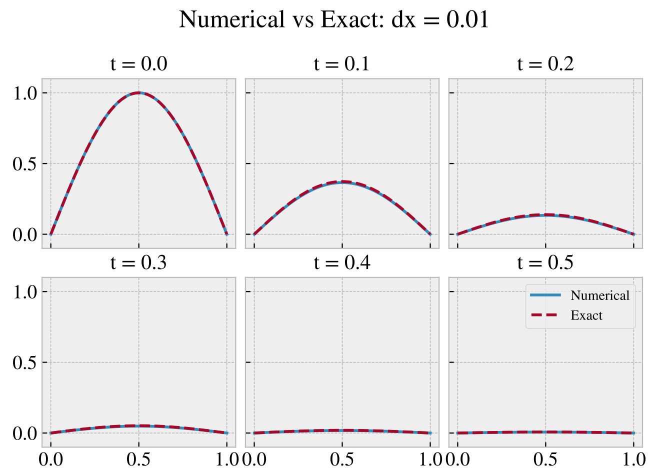

Figure 2: Explicit solution vs exact solution for diffusion equation with $dx = 0.01$.

In figure 2 we observe that the numerical solution preform far better with the new step-size. The numerical solution is almost indistinguishable from the exact solution.

2. *Neural Network*

The results from using the neural network to calculate the PDE solution is shown in figures (3) and (4).

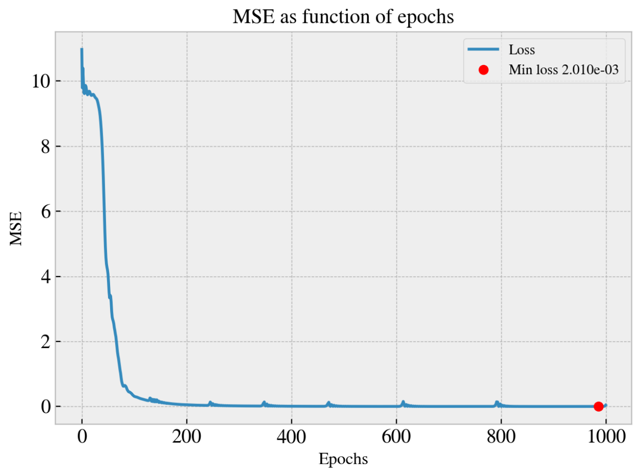

Figure 3: Tensorflow MSE as function of epochs, using $\eta = 0.05$, 1000 epochs, $dt = dx = 0.01$, $L = 1$ and $T = 1$.

In figure 3 we observe that the mean squared error falls down rapidly, and then stabilizes around 0.004. The sawtooth behavior is due to the ADAM optimizer, which tries to see if the solution is stuck in a local minimum, and not the global minimum. From the plot we can assume that it has found the global minimum.

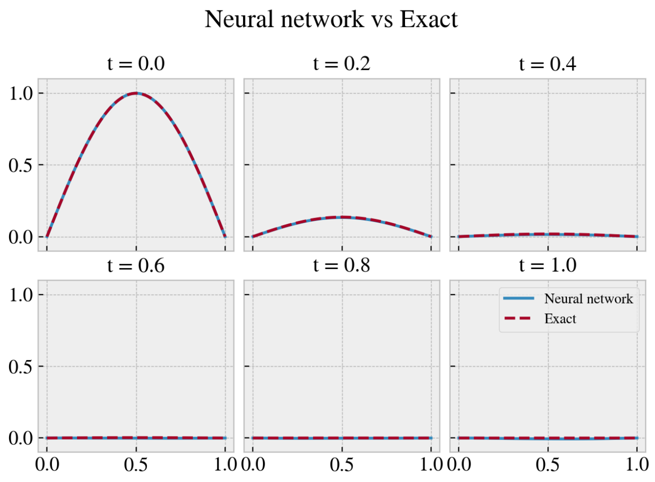

Figure 4: Tensorflow solution against exact solution for $t = 0, 0.2, 0.4, 0.6, 0.8, 1$, using $\eta = 0.05$, 1000 epochs, $dt = dx = 0.01$, $L = 1$ and $T = 1$.

In figure 4 we see the machine learning solution against the exact solution for $t = 0, 0.2, 0.4, 0.6, 0.8, 1$. Here we observe that the solution fits very well with the exact solution. A gif of the entire solution, and other solutions can be found here.

3. *Explicit vs neural network*

To compare the two methods, one can look at the mean absolute error. Here we have solution for several time steps, so we calculate the absolute error for one time step and then take the mean. This is shown in figure 5 and 6.

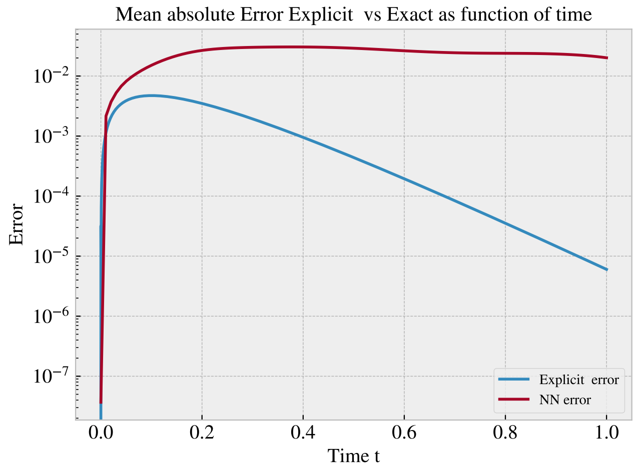

Figure 5: Log plot of Tensorflow and explicit solution as function of time with $\eta = 0.05$, 10^3 epochs, $dt = dx = 0.01$ for the network, $dx = 0.01$ and $dt = \frac{dx^2}{2}$, $L = 1$ and $T = 1$, using sigmoid as activation

In figure 5 we observe the mean absolute error as function of time for both the neural network and the explicit solution. We observe here that the neural network quickly gains larger absolute error, and stabilizes around $5 \cdot 10^{-2}$, whilst the explicit solution declines after one tenth of a second.

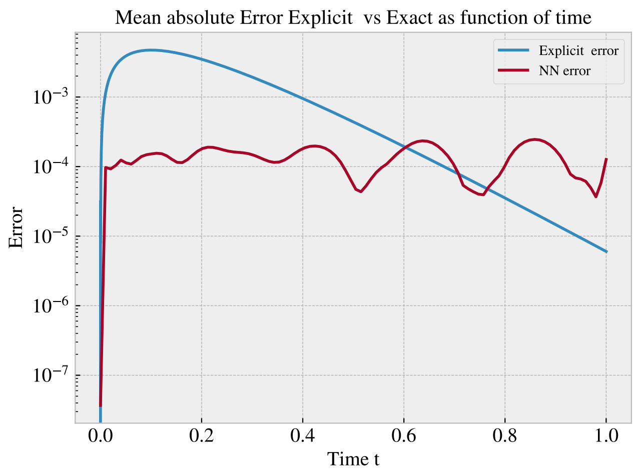

Figure 6: Log plot of Tensorflow and explicit solution as function of time with $\eta = 0.05$, 10^4 epochs, $dt = dx = 0.01$ for the network, $dx = 0.01$ and $dt = \frac{dx^2}{2}$, $L = 1$ and $T = 1$.

In figure 6 we observe that the absolute error for the neural network is lower than for the explicit solution up 6 tenths of a second, maintaining a oscillating error around 10^{-4}.

When evaluating the two methods we will judge them over 5 metrics. The metrics are

- Speed
- Absolute error
- Code difficulty
- Applicability (discrete/continuous)
- Stability

With regards to speed, the explicit solution is definitely faster. The calculation in figure 5 took 0.08 seconds for the explicit solution and 14 seconds for the neural network. The absolute error here is for higher epochs than 10^3 somewhat tied, but if speed is to be taken into account then the explicit is still better. It took Tensorflow 2 minutes and seconds to compute the error in figure 6.
With regards to code difficulty, we found the explicit solution to be easier to be implemented. We had little experience with Tensorflow, and to solve the PDE we needed to create a custom loss function, gradient function and train function. Given that we had prior experience with explicit solvers, it took less time to create that code. But, having learned the two methods, they are not that different in difficulty.
A consequence of using finite difference to solve PDE's is that we discretize the domains, and thus the solution as well. Thus we end up with a solution composed of points, which cannot be evaluated and is only valid inside the domain it was calculated over. The neural network works differently here, creating the trial function which can be evaluated continuously on the domain, and even outside. There are of course limitations to the trial function, given that the weights and biases are trained on the domain we define, and thus will be less accurate outside the domain.
The last comparison is with regards to stability, and this is where the neural network outperforms the explicit solution. The step size of the domains for the explicit has to follow equation 4 to remain stable. Thus for systems evolving at small spacial and temporal scales, such as molecular dynamics, the step size gets so small that the computation time for the explicit solution will drastically increase. The neural networks however are not subject to the stability criteria, and also scale better for higher order complexity. In the case of molecular dynamics, this could mean for example many particles.
For the examples one meets in Fys-Stk4155 or other physics courses where one calculates PDE's numerically, the explicit solution most likely will suffice, however the neural network can compete, at least with respect to a few of the metrics mentioned.

B. Eigen value problem

In this section we have studied the the eigenvalue problem discussed in section II C. The optimizer, number of hidden layers and nodes used in this section are the same as in the previous sections.
We generate a random symmetric 6×6-matrix and run the network over 200 epochs with an initial learning rate of 5×10^{-3}. The resulting maximum eigenvalue of the matrix over each epoch is plotted in figure 7.

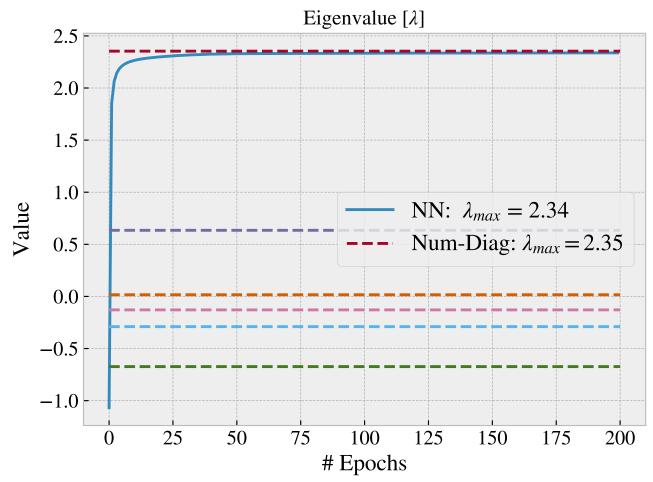

Figure 7: Dashed lines represent all 6 eigenvalues of the randomly generated matrix as calculated by numpy's eigen-solver. Solid line shows the neural networks calculated eigenvalue over each epoch.

Figure 7 displays that the neural network converges towards λ_{max}. After 200 epochs the network produces a $\lambda_{max} = 2.34$. Compared to the the standard eigenvalue solver, $\lambda_{max} = 2.35$ the network produces an error of 0.01, $\approx$ 4%.
As discussed in section II C, we calculated the eigenvector in order to find the eigenvalue. Figure 8 shows the evolution of the eigenvector to the corresponding eigenvalue presented above, over all epochs.

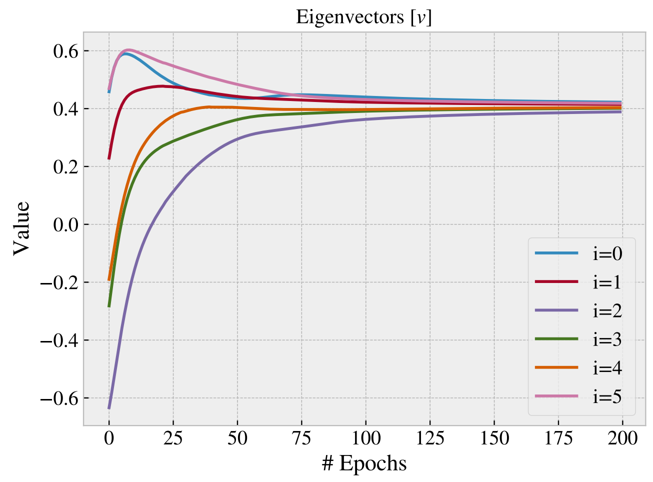

Figure 8: The lines in the figure show the evolution over 200 epochs of the values of each index in the eigenvector produced in the network.

In figure 8 we observe that the spread in the eigenvector decreases over each epoch. After 200 epochs all values in the vector stabilizes around 0.4. The final eigenvector produced after 200 epochs is

$$\lambda^{NN}_{max} = \begin{bmatrix} 0.42 \\ 0.41 \\ 0.39 \\ 0.40 \\ 0.40 \\ 0.49 \end{bmatrix} \tag{18}$$

, and the vector produced by numpy's eigenvalue solver is

$$\lambda^{ND}_{max} = \begin{bmatrix} 0.38 \\ 0.40 \\ 0.46 \\ 0.37 \\ 0.41 \\ 0.43 \end{bmatrix} \tag{19}$$

The comparison between the neural network and numerical diagonalization is displayed in figure 9.

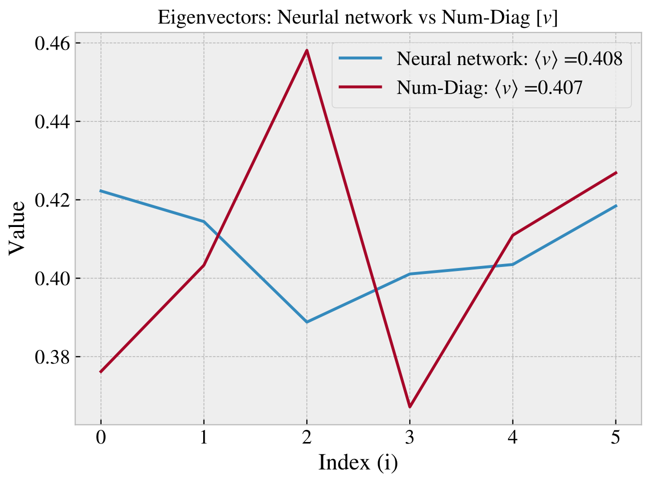

Figure 9: A comparison between the eigenvector produced by the network nad the eigenvector produced by numerical diagonalization.

From figure 9 it seems that the vector produced from the network is not very accurate and that it does not show the correct trend. At best, the network produces an accuracy of 10^{-2}. Instead the network has produced something more similar to a rough fit. This becomes obvious when comparing the mean of each of the vectors, which is also presented in figure 9. The error in the mean is of order 10^{-3}. By examination of the cost function (INSERT REF), we see that this is a consequence of the fact that $X^T X$ is heavily included in the calculation. Given that this term is proportional to the mean, it could explain why the network has focused on converging to the correct mean.

V. CONCLUSION

From the comparison of the finite difference method and the neural network we found that for simple problems such as equation 1 the finite difference method is faster and more accurate. With $dx = 0.01$ FD took 0.08 seconds whilst the network took 12 seconds at 1000 epochs and 2 minutes with 10000 epochs. We did however find that finite difference struggles at small scales due to the Neuman stability criteria, which the neural network isn't subject to. One problem with the FD method is that you get a discretized array of values, whilst the network calculates a function that can be evaluated everywhere. Thus for problems where this property is needed, one has to use neural networks.

REFERENCES

[1] David Deuvenad et al. “Neural Ordinary Differential Equations.” In: (2018). DOI: https://arxiv.org/pdf/1806.07366.pdf.

[2] J. G. Charney, R. FjÖrtoft, and J. Von Neumann. “Numerical Integration of the Barotropic Vorticity Equation”. In: *Tellus* 2.4 (1950), pp. 237–254. DOI: 10.3402/tellusa.v2i4.8607. eprint: https://doi.org/10.3402/tellusa.v2i4.8607. URL: https://doi.org/10.3402/tellusa.v2i4.8607.

[3] Diederik Kingma and Jimmy Lei Ba. “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.” In: (2015). DOI: https://arxiv.org/pdf/1412.6980.pdf.

[4] Hans Petter Langtangen. *Finite Difference Computing with PDEs*. Texts in Computational Science and Engineering. Springer, 2016. ISBN: 978-3-319-55455-6.

[5] Hans Petter Langtangen. *Introduction to Numerical Methods for Variational Problems*. Texts in Computational Science and Engineering. Springer, 2016. ISBN: 978-3-319-55455-6.

[6] Zhang Yi, Yan Fu, and Hua Jin Tang. “Neural Networks Based Approach Computing Eigenvectors and Eigenvalues of Symmetric Matrix .” In: (2003). DOI: https://www.sciencedirect.com/science/article/pii/S0898122104901101.