FYS-STK4155 MACHINE LEARNING AND APPLIED DATA ANALYSIS
Project 3: Using neural networks for solving the diffusion equation

Martin Krokan Hovden

December 19, 2019

Abstract

In this report we study the diffusion equation in one dimension with given boundary conditions and initial conditions. We compare the performance of the classic Forward Euler method with a more recent method using a neural network. Code for the Forward Euler method is developed from scratch and the neural networks is implemented using tensorflow. We study the theory behind the methods in detail. The optimal results for the neural network was found by doing a grid search. Analytical solution for the diffusion equation is derived for measuring the accuracy of the different methods.

We will see that the Forward Euler scheme outperforms the neural network on all test. The accuracy is better and the computational time is faster. For a grid with $\Delta t = 0.005$ and $\Delta x = 0.1$ the MSE from using Forward Euler is 3.13e-5 for t = 0.3 and 3.56e-5 for t=0.02. For the neural network we get 3.1e-4 and 9.1e-4 for the same t-values. In addition, the time it took to train the best performing network was 1337 seconds while the forward Euler used less time than the computer was able to register. The conclusion is that the Forward Euler method is still a better alternative for normal applications. The only problem is the stability condition. However, for this equation we could easily bypass that by choosing Δx and Δt within the stability limit without any problems. The best neural network was a network with one hidden layer with 50 neurons and sigmoid activation function. The standard gradient descent method was used with learning rate = 0.01 and $2 * 10^5$ iterations in training.

After studying the diffusion equation we use the neural network for finding eigenvalues of symmetric matrices. The same network architecture as for the diffusion equation was used with slight modifications to the loss function. The method is effective for smaller matrices up to size 10x10. After that the time start to increase and the results are not as stable. For symmetric matrices of size less than 10x10 the results are very accurate. For a 6x6 matrix the method finds the exact eigenvalues. The eigenvectors have an MSE of order 1e-18.

Contents

1 INTRODUCTION

Solving partial differential equations is of great interest in almost every part of science and engineering. They are used for modelling problems in fields ranging from fluid mechanics to finance. Some partial differential equations can be solved analytically. However, in general, this is not the case. Traditionally numerical methods have been used for solving them efficiently and accurately. Numerous numerical methods have been developed over the years, and better and more efficient methods is still being researched. Over the last years new approaches to solving them have been developed. One of the main problems using the traditional finite difference methods is that they use a discretized domain. This lead to limited differentiability [6], and methods that avoid this is of interest. One alternative is to use neural networks. This gives a function that can be used to calculate the solution for all points, and are not restricted to the discretized domain.

In this project we study and compare different methods for solving the diffusion equation in one dimension. In its most simple form, the diffusion eqaution is given by

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

It can for example be used for modelling the heat in a rod or the fluid flow between two moving plates. We will develop code for forward Euler (FE) and the neural network method, for solving the diffusion equation. The neural network method is a method that with slight modifications can find the solutions of a wide specter of differential equations [6]. The theory behind the methods will be studied in depth and various properties of the algorithms is discussed. We can easily derive analytical expression for the diffusion equation in its most simple form with the boundary conditions and initial conditions used in this report. These will be presented later. We will also see that we are able to cast an eigenvalue problem to a problem suitable for solving with a neural network. We will then be able to find the eigenvalues of symmetric matrices using a neural network.

The report is split into four main parts. We will start by looking at the theory behind the diffusion equation and the different numerical methods. By deriving the methods from scratch we will get an intuitive idea about how to the methods actually finds the solutions. This is beneficial when trying to debug code and analysing results. Then we will look at how the GitHub repository is structured and where to find the different methods we have developed. I have tried to write explaining comments and doc-string within the code so it should be possible to follow. Some of the code is also presented in the theory part. After that the results obtained are presented and the performance is discussed. After that we will wrap up the report with a short conclusion that summarizes the main findings from the report and ideas for future work.

2 THEORY AND METHODS

The theory and methods part is based on the lecture notes written by Morten Hjort-Jensen for the courses FYS4150 and the notes written by Kristine Hein about solving differential equations using neural networks. For more information and detailed derivations of the results, see [1] [5]. For the diffusion equation, we will focus on solving it using a neural network and then compare the performance to the more classical approach of the Forward Euler scheme. Since we know the analytical solution we are able to get accurate estimates of each algorithms performance. For the eigenvalue problem, we will use the same neural network with slight modifications to the loss function. The obtained solutions will be compared to numpy's implementation for finding eigenvectors and eigenvalues.

2.1 Description of the problem

In the first part of this project we study the diffusion equation. The diffusion equation can for example be used to model the temperature gradient in a rod or the fluid flow between two plates where the upper plate moves [1]. The diffusion equation in one spatial dimension in general form is given by

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

with initial condition at t=0 set to

$$u(x,0) = g(x) \quad 0 < x < L \tag{3}$$

where we will use L = 1 and $t \geq 0$. The initial condition describes the temperature at the beginning. We will use the initial condition

$$g(x) = \sin(\pi x) \tag{4}$$

so the temperature distribution is curved in the beginning. In addition we have boundary conditions

$$u(0,t) = a(t) \quad t \geq 0 \tag{5}$$

and

$$u(1,t) = b(t) \quad t \geq 0. \tag{6}$$

with

$$a(t) = b(t) = 0. \tag{7}$$

This can be interpreted as that the ends of the rod are held a a constant temperature, and that there is a sinusoidal temperature distribution in the interior of the rod. We assume that since no heat is added and that the ends are held constant, that after time the temperature in the rod will be constant and equal to a linear function between the endpoints. This will be the steady state of the system. In our case the steady state of the rod is with temperature equal to zero in the whole rod. This is the intuition behind it, and the steady state will be derived mathematically later.

The diffusion equation is a partial differential equation (PDE). A partial differential equation contains multivariate functions and their partial derivatives [3]. The goal is to find a function that fits into the PDE. In general we can not find analytical solution to PDE's by hand. However, for the diffusion equation with the given initial and boundary conditions we can derive expressions for the analytic solution. This is beneficial since we are able to measure how good our numerical methods works.

2.2 Analytical solution

An often used method for solving the diffusion equation analytically is by using separation of variables. This means that we guess that the solution is on a particular form, so that we can separate each variable on opposite sides of an equality [3]. This can be done by guessing that the solution can be written as a product of two functions. The first function is only dependent on time and the other function is only dependent on the spatial coordinates. u(x,t) can then be written as

$$u(x,t) = F(x)G(t) \tag{8}$$

where F is only dependent on the spatial coordinate x and G on time t. We insert this into the original equation 2 and get that

$$\frac{F''(x)}{F(x)} = \frac{G''(t)}{G(t)}. \tag{9}$$

Since this have to be true for all t and x we note that each side have to be equal to an constant, for example $-\lambda^2$. The constant could be anything but we will see that letting the constant be $-\lambda^2$ cleans up the later results significantly. Using this results in

$$\frac{F''(x)}{F(x)} = \frac{G'(t)}{G(t)} = -\lambda^2 \tag{10}$$

where the derivatives are taken with respect to the variable in the functions. We are now left with two ordinary differential equations to solve. By rearranging we get that

$$F''(x) + \lambda^2 F(x) = 0 \tag{11}$$

and

$$G'(t) + \lambda^2 G(t) = 0. \tag{12}$$

These equations can now be solved separately. Both equations can be solved using standard techniques for ordinary differential equations. See for example [1]. We will start by focusing on equation 11 which is the spatial part. The solutions are given by

$$F(x) = A\sin(\lambda x) + B\cos(\lambda x). \tag{13}$$

[1]. The constants can be found by using the initial conditions. We start by setting x = 0. This gives

$$F(0) = B = 0 \tag{14}$$

and the cosine term vanishes from F. We are then left with

$$F(x) = \sin(\lambda x). \tag{15}$$

By using the second boundary condition we see that

$$F(1) = \sin(\lambda) = 0 \tag{16}$$

leads to the requirement that

$$\lambda = \pi k. \tag{17}$$

This results in

$$F(x) = A\sin(\pi k x). \tag{18}$$

We can now move on to solving equation 12 which have the solution

$$G(t) = Ce^{-\lambda^2 t}. \tag{19}$$

[1]. Combining the two equations gives

$$u_n(x,t) = A_n \sin(n\pi x)e^{-(n\pi)^2 t}. \tag{20}$$

where I have changed k to n. Since the diffusion equation is linear [1] a linear combination of solutions is also a solution. The solution can then be written as a sum

$$\sum_{n=1}^{\infty} A_n \sin(n\pi x)e^{-(n\pi)^2 t}. \tag{21}$$

A_n is found using the initial condition and the Fourier coefficients [1] of the initial condition

$$u(x,0) = \sin(\pi x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) \tag{22}$$

where A_n is the Fourier coefficients of $sin(\pi x)$. Thus A_n can be found by

$$A_n = 2\int_0^1 \sin(\pi x)\sin(n\pi x)dx. \tag{23}$$

[1]. Since $\sin(m\pi x)$ and $\sin(n\pi x)$ are orthogonal, it means that

$$\int_0^1 \sin(m\pi x)\sin(n\pi x) = \delta_{mn} = \begin{cases} 1/2 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}. \tag{24}$$

[3]. This leads to

$$A_n = 2\int_0^1 \sin(\pi x)\sin(n\pi x)dx = \delta_{1n}, \tag{25}$$

and that $A_n = 0$ for $n > 1$. The only remaining term is A_1. We can calculate the integral and get that

$$A_1 = 2\left[\frac{1}{2}\right] = 1. \tag{26}$$

The full solution then becomes

$$u(x,t) = \sin(\pi x)e^{-(k\pi)^2 t}. \tag{27}$$

The analytical solution can easily be implemented in python with the following code snippet

```
analytic_solution = np.sin(np.pi*x_values)*np.exp((-np.pi**2)*t)
```

We can also find the steady state of the system. We know that when the system have reached the steady state, we have that

$$\frac{du(x,t)}{dt} = 0 \tag{28}$$

since the temperature no longer changes. Inserting this in the diffusion equation gives

$$\frac{d^2u(x,t)}{dx^2} = 0. \tag{29}$$

The equation is called the Laplace function in one dimension [1]. The solution is found by integrating twice, and we get that

$$u(x,t) = Ax + B. \tag{30}$$

Using the boundary conditions, we get

$$u(x,t) = 0 \tag{31}$$

as the steady state. This is what is expected as discussed in the beginning of the section. We will now look at the forward Euler method for solving the diffusion equation. We can actually use the same methods as is used for the full diffusion equation for finding the steady state also. Ideally we would have to simulate for $t \to \infty$. This is not possible, but by simulating for a long enough time, we would get approximately the solution we are looking for.

2.3 Numerical methods for solving the PDE

Now that the analytical solution have been derived we will study the numerical methods to be used. In this part we will focus on the Forward Euler method for solving the diffusion equation. Stability conditions will be derived for the algorithm. We will see that we need to be careful when choosing the parameters when using the Forward Euler scheme.

2.3.1 Forward Euler

The forward Euler scheme is a finite difference method. In finite difference methods we set up a discretized domain which acts as grid points. The idea is to make approximations of the functions derivatives at each grid point and try to find function values that satisfies the equation for each grid point [3]. We make grids in the spatial dimension and in the time dimension. Let x_i denote the i'th grid point and t_j denote the j'th time step. We will use the notation u_i^j where i is the spatial index and j is the time step. We can then write $u(x_i, t_j+\Delta t) = u_i^{j+1}$ and $u(x_i+\Delta x, t) = u_{i+1}^j$.

Forward Euler method is an explicit finite difference scheme [1]. This means that the values in the next time step is only dependent on values in the previous time step. The idea is to replace the terms in the equation with numerical approximations of the derivatives. This means that by using a finer grid, we typically will get a better approximation of the solution.

For the forward Euler method we approximate the derivatives by

$$u_t \approx \frac{u_i^{j+1} - u_i^j}{\Delta t} \tag{32}$$

where u_t denotes the derivative of u with respect to t. The truncation error for the first derivative with respect to time is $O(\Delta t)$. We can also find

$$u_{xx} \approx \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2} \tag{33}$$

with an truncation error of $O(\Delta x^2)$. u_{xx} is the second derivative of u with respect to x. Inserting the approximations of the derivatives into the diffusion equation gives that

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}. \tag{34}$$

We are interested in finding an expression for the solution for the next time step. By rearranging the equation we can find an recursive expression for finding the time development of u.

$$u_i^{j+1} = \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right) + u_i^j. \tag{35}$$

We note that the solution for the next timestep only is dependent on the previous time step. This is what's characteristic for explicit finite difference schemes. The forward step can easily be implemented in python with the following code snippet

```
u[t_index, x_index] = alpha*(u[t_index-1, x_index + 1] - 2*u[t_index-1, x_index] + \
              u[t_index-1, x_index - 1]) + u[t_index-1, x_index].
```

The code is vectorized to increase speed. By iterating over the grid with the step-function one could simulate the solution for any t-value (Might take some time for large t-values). An example on how to use this is in the file `diffusion_equation.py` and `5c.py`.

The Forward Euler method is easy to implement and provides an intuitive solution to the problem. However, one of the main problems is the bad stability conditions. By not fulfilling the conditions, we can end up with a solution that blows up. The stability condition is given by $\alpha \leq \frac{1}{2}$ [1]. This means that if we have fixed a value for Δx then we have to have $\Delta t \leq \frac{1}{2}\Delta x^2$. If we for example have $\Delta x = 0.01$ we would need $\Delta t \leq 0.00005$. Simulating the temperature for high values of t would then be problematic with this short step length. Other methods is not limited by this criteria. For a discussion on alternative methods, see [1].

2.3.2 Derivation of stability criterion of the forward Euler scheme

The stability criterion can be derived by writing the scheme on matrix form. By introducing the vector

$$U^j = \begin{bmatrix} u_1^j \\ u_2^j \\ ... \\ u_n^j \end{bmatrix} \tag{36}$$

and the matrix

$$A = \begin{bmatrix} 1-\alpha & \alpha & 0 & ... & ... & 0 \\ \alpha & 1-\alpha & \alpha & 0 & ... & 0 \\ 0 & \alpha & 1-\alpha & \alpha & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 0 & \alpha & 1-\alpha \end{bmatrix}, \tag{37}$$

we can write the scheme as

$$U^{j+1} = AU^j. \tag{38}$$

We can then find the u-values for the j'th timestep by using the matrix iteratively

$$U^j = A^j U^0. \tag{39}$$

By rewriting equation 35 as

$$\boldsymbol{u}^{j+1} = A\boldsymbol{u}^j \tag{40}$$

the stability criterion is that

$$\rho(A) < 1 \tag{41}$$

where

$$\rho(A) = \max\{|\lambda| : \det(A - \lambda I) = 0\}. \tag{42}$$

[1]. This means that the largest eigenvalue have to be smaller than zero. We can now find the eigenvalues of A. The first step is to introduce the new matrix B and write A as

$$A = I - \alpha B, \tag{43}$$

where B is

$$\begin{bmatrix} 2 & -1 & 0 & ... & ... & 0 \\ -1 & 2 & -1 & 0 & ... & 0 \\ 0 & -1 & 2 & -1 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 0 & -1 & 2 \end{bmatrix}. \tag{44}$$

We can then find the eigenvalues of A given the eigenvalues μ_i of B from

$$\lambda_i = 1 - \mu_i. \tag{45}$$

The elements of B can be written as

$$b_{i,j} = 2\delta_{i,j} - \delta_{i+1,j} - \delta_{i-1,j} \tag{46}$$

so the eigenvalue problem can be written as

$$(Bx)_i = 2x_i - x_{i+1} - x_{i-1} = \mu_i x_i \tag{47}$$

By saying that x can be rewritten as a linear combination of the basis $(\sin(\theta), \sin(2\theta)...)$ where $\theta = \frac{l\pi}{n} + 1$ we get by inserting into the equation that $\mu x_i = 2(1 - \cos(\theta))x_i$ which gives the eigenvalues $\mu_i = 2 - 2cos(\theta)$. Since we want $\rho(A) < 1$ we need that

$$-1 < 1 - \alpha(2 - 2cos(\theta)) < 1 \tag{48}$$

so we see that the condition is that $\alpha < 1 - \cos(\theta)^{-1}$. From this we get that the stability criterion is that

$$\alpha \leq 1/2. \tag{49}$$

This can be a problem when simulating a system over time. We often use $\Delta x = 0.01$ which means that we have to have $\Delta t < 0.5\Delta x^2 = 0.00005$. If we want to simulate over a long time period, this method becomes extremely slow. Other methods excist that is not restricted by this limit. We will not discuss those in this report. If interested, see for example [1] or [3].

2.4 Solving PDE's using neural networks

In this section we will discuss a method for solving partial differential equations using neural networks. For an introduction to neural networks and how they work, see for example [4]. The idea is to use a network of layers and neurons with weights connecting the neurons. By feeding inupt to the network, the network transforms the input to some output. We want to use a numerical optimization method for finding the weights that minimizes the error between the output of the network and the actual output on a training set.

The main idea we use when solving partial differential equations using a neural network is the universal approximation theorem. The theorem states that a neural network with only one hidden layer and a finite number of neurons can approximate almost any continuous function to any given accuracy [4]. The solution of a differential equation is a function, so we could then assume that it would be possible to use a neural network to find this function. This is the main idea that this method builds upon. However, we need to reformulate the problem into a form that the neural network can use.

Again, we will study the diffusion equation with the given initial conditions. The first step is to find a trial solution. The trial solution is a function that includes the output from the neural network and the goal of the trial function is to restrict the solution to fulfil the boundary conditions and initial conditions [5]. In general, we say that the trial function for a function of x and t is on the form

$$g(x,t,P) = h_1(x,t) + h_2(x,t,N(x,t,P)) \tag{50}$$

where N is the neural network, and P is its weights and biases [5]. The trial solution consist of two parts where the first part makes the solution fulfill the initial condition while the second one makes it fulfill the boundary conditions. We now have a function where we can tweak the weights and biases when minimizing a cost function.

By inserting the trial solution in the diffusion equation, the problem can be restated as

$$\frac{\partial g(x,t,P)}{\partial t} = \frac{\partial^2 g(x,t,P)}{\partial x^2}. \tag{51}$$

We can then introduce a cost function c,

$$c(x,t,P) = MSE(\frac{\partial g(x,t,P)}{\partial t} - \frac{\partial^2 g(x,t,P)}{\partial x^2}, 0) \tag{52}$$

where MSE is the mean squared error. We can now use the neural network to minimize this cost function by tweaking the weights and biases of the network. In the implementation, we will also test for different regularization parameters. More info on regularization parameters can be found in [4]. For our case we will add the square of the weights in the network to the cost function to restrict the size of the weights. For some cases this works well, and the results obtained gets better by adding regularization. For some examples this is not the case. We will study the effect of regularization parameters in the results part.

We will now focus on how to set up the trial function. The idea is to find a trial solution so that the boundary conditions and initial conditions are fulfilled. By introducing the trial function

$$g_t(x,t) = h_1(x,t) + x(1-x)tN(x,t,P). \tag{53}$$

where

$$h_1(x,t) = (1-t)\sin(\pi x). \tag{54}$$

we see that it fulfils the boundary conditions and initial conditions. The first conditions makes sure that it fulfils the given conditions, while the second parts make sure that the impact of the output from the neural network is zero where the boundary conditions are supposed to hold. We note that for t = 0, we have

$$g_t(x,0) = \sin(\pi x), \tag{55}$$

which fit the initial condition. We also see that for the other boundary conditions, we have that

$$g_t(0,x) = g_t(1,x) = 0 \tag{56}$$

since

$$\sin(0) = \sin(\pi) = 0. \tag{57}$$

The idea behind the trial function is to force the solution to produce a function that fulfills the boundary conditions and the initial conditions, while the network tries to minimize the loss function [6]. We hope that the trial function converges towards the analytical solution as we train the network. The training is done by iteratively updating the weights and biases in the network, as one would usually do in the training of a neural network.

We typically have the input in a matrix X. The matrix X is given by

$$\begin{bmatrix} \boldsymbol{x_1}^T \\ \cdots \\ \cdots \\ \boldsymbol{x_n}^T \end{bmatrix} \tag{58}$$

where each row is the solution for a different time-step, so that $\boldsymbol{x}_i(x_{11}, x_{12}, ..., x_{1m})$. The matrix X is thus fed into the neural network, and we get the solution for each time-step.

The trial function and the loss function can be implemented in python using the following code snippet

```python
with tf.name_scope('loss'):
    g_trial = (1-t)*u(x) + x*(1-x)*t*dnn_output

    g_trial_dt = tf.gradients(g_trial, t)
    g_trial_d2x = tf.gradients(tf.gradients(g_trial,x),x)

    reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
    loss = tf.losses.mean_squared_error(zeros, g_trial_dt[0] - g_trial_d2x[0]) +
        regularization_parameter*sum(reg_losses)
```

Here we have used tensorflow's ability to find the gradient of functions efficiently. Regularization is also added to the loss to restrict the weights from growing very large. This loss is then minimized using a neural network and a numerical minimization algorithm. How this can be done in tensorflow is shown in the jupyter notebook `nn_diffusion_equation.ipynb` and in the file `diffusion_solvers.py`. More detailed explanation on how the code is implemented is in the implementation part of this report.

2.5 Comparing the methods

The comparison will focus on computational time and accuracy. For time we will simply measure the time it takes for the algorithm to compute the solution for all time-steps up to a given time t. To compare the accuracy of the schemes there are different methods that can be used. Since we know the exact solution of the problem we can easily obtain accurate error measures. One could for example extract the maximum difference between the numerical solution and the analytical solution. However, I do not think this is the best way to compare the methods. It is more interesting get an idea about how the method performs over the whole spatial domain. For the analysis I will therefore study the mean squared error (MSE) for each time step. The MSE for a time t is given by

$$\frac{1}{N_x}\sum_{i=1}^{N_x}(u_{\text{numerical}} - u_{\text{analytical}})^2. \tag{59}$$

where N_x is the number of grid points in x-direction and u and $u_{\text{analytical}}$ is the numerical and analytical solution at time t. For a qualitative analysis of how the methods compare we will also plot the error for each scheme. By plotting

$$\text{error} = u_{\text{analytic}} - u_{\text{numerical}} \tag{60}$$

we can study from which side the numerical solution approaches the exact solution.

2.6 Finding eigenvalues of symmetric matrices using neural networks.

The eigenvalues and eigenvectors of a matrix A is defined by

$$A\boldsymbol{v} = \lambda \boldsymbol{v} \tag{61}$$

where λ is the eigenvalue and $\boldsymbol{v}$ is the eigenvector. The problem of finding eigenvalues is important and many physical problems can be transformed to an eigenvalue problem. For simple cases and small matrices we can calculate the eigenvalues and eigenvectors analytically. See for example [1]. However, when the matrices is large, this can become extremely tedious. Traditionally, the choice of method for larger matrices have been to use iterative methods. Those methods iterate towards the exact eigenvalues and eigenvectors and gives better and better estimates for each iteration. However, in the article [2] the authors present an alternative method. This method uses neural networks and the same idea as we used for solving the diffusion equation. The eigenvalue problem is transformed into a differential equation. The equation we will use to solve the problem is

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)) \tag{62}$$

where f is a function that contains the matrix A,

$$f(x) = [x^T x A + (1 - x^T A x)]x. \tag{63}$$

[2]. The matrix A is a symmetric nxn matrix.

This is a non linear differential equation. x is the the column vector $x = (x_1, x_2, ..., x_n)^T$ and it is the output from the neural network. It can be shown that any stationary point of the equation is an eigenvector of the matrix A [2]. We can therefore disregard the time-dependency in x. The stationary state is reached when $\frac{dx(t)}{dt} = 0$, so the problem is now simplified to solve

$$x(t) = f(x(t)). \tag{64}$$

The x that satisfies this equation is the steady state of the system. We can now use the same technique as for the diffusion equation. However, we are no longer resticted by the boundary conditions and the initial condition. The trial solution is thus just some function of the input which the network is supposed to find.

The cost-function can then be set up as follows,

$$C(x(t)) = MSE[x(t) - f(x(t)), 0]. \tag{65}$$

where x is the output of the network given some input. When the neural network trains, it will try to minimize the difference between $x(t)$ and $f(x(t))$ by tweaking the weights and biases. Starting from any non-zero input vector, the network will converge to the eigenvector corresponding to the largest eigenvalue [2].

We can also make the network converge to other eigenvectors not corresponding to the largest eigenvalue. The easiest one to obtain is the one corresponding to the smallest eigenvalue. By using -A instead of A in f, we will converge to the eigenvector corresponding to the smallest eigenvalue of A [2]. It is also possible to find the eigenvectors for the remaining eigenvalues. It can be shown that if the input vector, lets call it x_0, is orthogonal to the a set of k eigenvectors, then the solution converges to an eigenvector that is orthogonal to the k eigenvectors [2]. We will see examples of this later. However, one of the problems with doing this is numerical errors when computing the orthogonal vector. Typically, we will not end up with a vector that is exactly orthogonal, and we will converge to the largest eigenvalue instead.

For proofs and further information, see [2].

3 IMPLEMENTATION

The details on implementations and how to use the function are found at the GitHub repository: `https://github.com/MartinKHovden/FYS-STK_project3`. Some familiarity with tensorflow is required to follow the code using neural networks. For an introduction to tensorflow I would recommend the text [4] by Aurélien Géron. It is a good introduction for getting started with using neural networks in python.

In the file `diffusion_solvers.py` you will find a function for the forward Euler method and alternative numerical methods. In this project we will focus on the forward Euler method. The method is tailored for the diffusion equation in one dimension as stated in equation 2. It is implemented as described in the theory part. The code is commented so it should be easy to follow. The function lets the user fine-tune the problem. It is possible to choose the interval of x-values, Δx, Δt, max t-value and the boundary and initial conditions. For more info see the doc-string. In the jupyter notebook `part_b.ipynb` examples of how to use the code for different Δt and Δx are found. This is also used for making the different plots in the report. Comments in the notebook explains how to test for different parameters.

In the file `diffusion_solvers.py` you will also find a method for solving the diffusion equation using a neural network. The method uses the functionality of tensorflow 1. The method sets up a neural network, defines the loss function as described in the theory part, train the network and returns the solution. The network can be customized up by user. It is possible to define the learning rate, regularization parameter, number of iterations in training, and the number of hidden neurons in each layer. It is restricted to using the sigmoid activation function in all the layers. This can easily be changed by updating the solver directly in the function. In the jupyter notebook `nn_diffusion_equation.ipynb` we present a script on how to use the function. The notebook `GridSearchDiffusionEquation.ipynb` is used for tuning the parameters. It loops over different combinations of learning rates and regularzation parameteres and produces a heatmap of the results.

A script for finding the eigenvalues of symmetric matrices are found in the jupyter notebook `nn_eigenvalues.ipynb`. The script is imlemented as desribed in the theory part. We start by setting up the neural network where the input is a vector of length equal to the dimensions of the matrix A. Then the neural network minimize the loss and returns the desired eigenvector.

For testing the code, run

```
C:.../pytest diffusion_solvers.py
```

in the folder where the file is located. This will run some simple tests to check if the forward Euler scheme fulfills the boundary conditions at all steps and if it gives the correct solution to simple cases where we know the analytical solution. Since the neural network is implemented using tensorflow we should trust that Google have done extensive testing of the code already, and no tests will be conducted by us.

4 RESULTS AND DISCUSSION

4.1 Diffusion equation

In this section we look at the performance of the Forward Euler method and the neural network for solving the diffusion equation with the initial conditions and boundary conditions given in the theory part. In the title of the plots you will find information about which parameters are used for producing the plots.

4.1.1 Forward Euler

We start by studying the performance of the Forward Euler scheme for two different times, t=0.02 and t=0.3. All the results are studied for the diffusion equation as given in 2 on the domain $x \in [0, 1]$. The times are chosen so that for t=0.02 the solution is still significantly curved and for t=0.3 the solution is almost linear. We will study how the accuracy of the methods change as the shape of the function changes. It is also interesting to see how the solution behaves for varying Δt and Δx. The stability conditions of the scheme can lead to bad results if we are not careful when setting up the solver, which we will see later.

We start by looking at the results for t=0.02. The results are shown in figure 1 for $\Delta x = 0.1$ and $\Delta t = 0.005$ which is at the stability limit for Forward Euler. This is early in the process and we see that the solution is close to the initial conditions. In the right part of the figure the error is shown. We see that the forward euler method undershoots the actual solution for all values (plotting analytical minus numerical), except for at the boundaries. By increasing the resolution of the grid we get the results in figure 2. It is for the same time, but now we have $\Delta x = 0.01$ and $\Delta t = 0.00005$, which also is at the stability limit. Again, the numerical solution is close to the analytic solution. However, we see that the absolute values of the error is smaller than for the larger step-lengths. This is as expected. The maximum error with the finer grid size is just above 0.000025 while for the less detailed grid the maximum error is 100 times as high. This shows that by making a finer grid and reducing the step lengths we get a more accurate solution when using the forward Euler method.

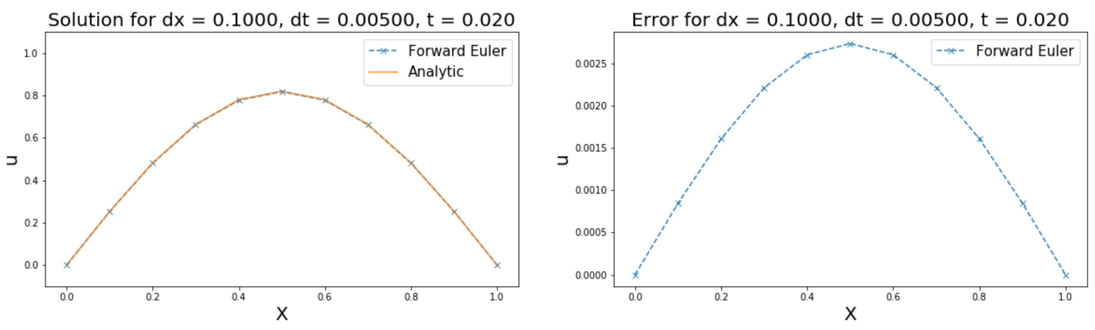

Figure 1: Left: Forward Euler and analytic solution. Right: Error for Forward Euler. t = 0.02, $\Delta x = 0.1$ and $\Delta t = 0.005$.

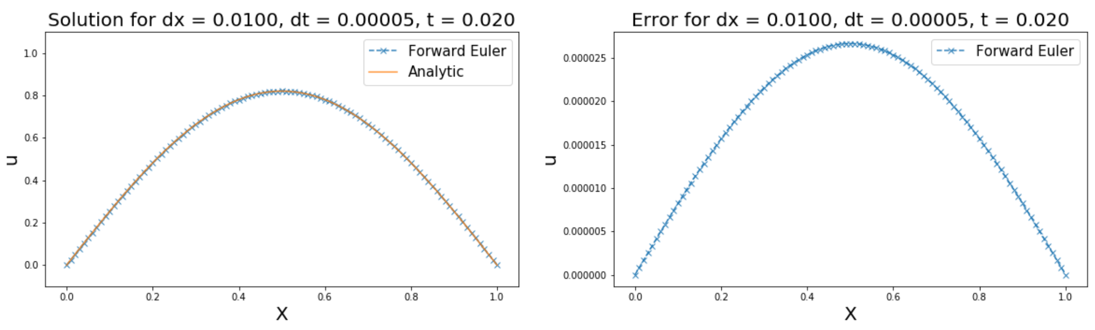

Figure 2: Left: Forward Euler and analytic solution. Right: Error for Forward Euler. t = 0.02, $\Delta x = 0.01$ and $\Delta t = 0.00005$.

We can now study the numerical solution when the temperature is almost linear between the end points. In figure 3 and 4 the solution is shown for t = 0.3. Again, we see that the numerical approximation is close to the analytic solution for both combinations of grids. However, as we saw for t=0.2, the error is smaller for the finer grid. Again the difference is by a factor of 100. We can conclude that as long as we are within the stability limit of the forward Euler scheme, the solution will improve by making Δx and Δt smaller. The problem is that if we are not careful that the

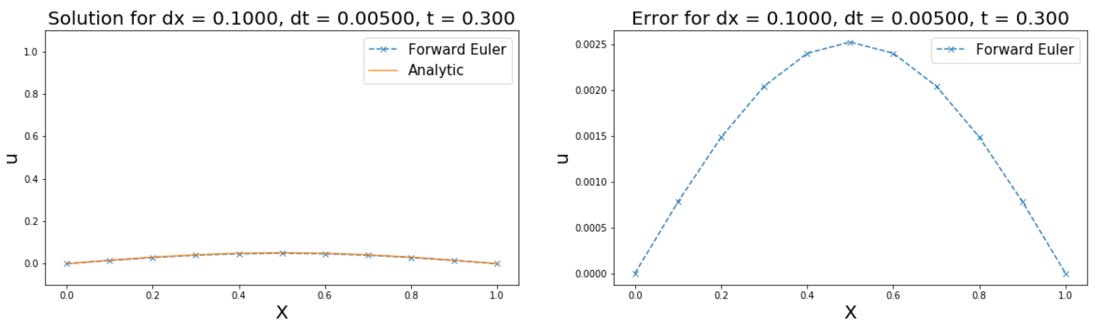

Figure 3: Left: Forward Euler and analytic solution. Right: Error for Forward Euler. t = 0.3, $\Delta x = 0.1$ and $\Delta t = 0.005$.

stability criterion is fulfilled, the solution can get very bad very fast. Only a slight breach of the criteria lead to extremely bad results. This can be seen in figure 5. For $\Delta x = 0.01$ the criteria for Δt is $\Delta t \leq 0.00005$. In figure 5 the numerical solution are plotted for $\Delta t = 0.00006$. The solution blows up and the error is of order 1e32. This is the main problem with the Forward Euler scheme. Other methods can be used instead to avoid this criteria. However, those often involves solving a matrix equation and are a lot slower than the forward Euler scheme [3]. All in all the results from using the Forward Euler method was good, and we saw that by increasing the number of grid points we get a better solution. The problem is that the computational time increases when making a finer grid by decreasing Δt and Δx. This can be seen in table 1 and 2. We see that for the smallest grids the time is less than machine precision. For the larger grid the time is 0.0074. However, we see that the MSE is much lower with more grid-points. We also note that the Forward Euler method is more precise when the solution is closer to linear.

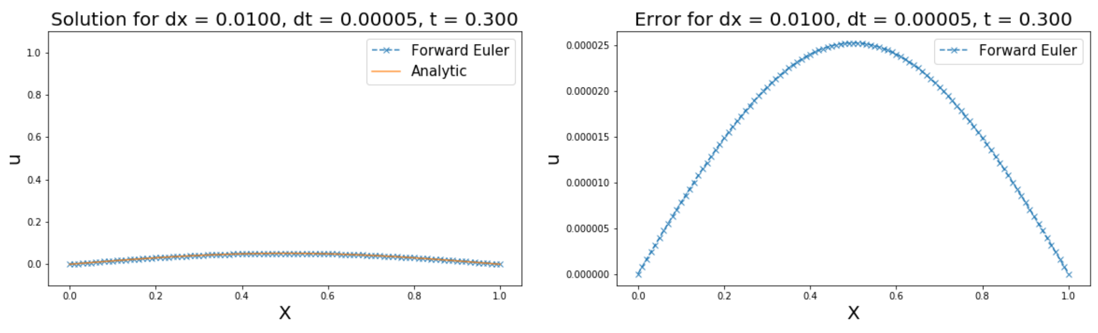

Figure 4: Left: Forward Euler and analytic solution. Right: Error for Forward Euler.t = 0.3, $\Delta x = 0.01$ and $\Delta t = 0.00005$.

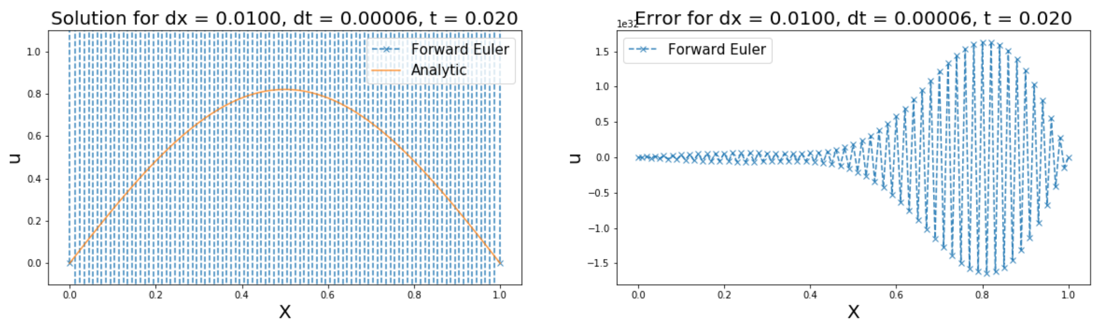

Figure 5: t = 0.02, $\Delta x = 0.01$ and $\Delta t = 0.00006$. Left: Forward Euler and analytic solution. Right: Error for Forward Euler.

Method	Time [s]	MSE t = 0.02	MSE t = 0.25
Forward Euler	0.0073792	1.265e-07	8.119e-11

Table 1: Results for $\Delta t = 0.00001$ and $\Delta x = 0.01$. Time is for calculating solution up to t = 1.

Method	Time [s]	MSE t = 0.02	MSE t = 0.25
Forward Euler	0.000	0.00013	1.65e-10

Table 2: Results for $\Delta t = 0.001$ and $\Delta x = 0.1$. Time is for calculating solution up to t = 1.

4.1.2 Neural network

We will now focus on the results from using the neural network for solving the diffusion equation. For the parameter exploration we will use $\Delta x = 0.1$ and $\Delta t = 0.1$. The problem with the neural network is that it is more time-consuming than the traditional numerical methods. We will therefore restrict the grid search to this grid-size to save time. When the best parameters are found we will test for smaller values of Δt and Δx.

For the neural network we have more parameters we need to choose compared to the traditional methods. In addition to Δx and Δt we have to choose how the network is set up. The first step will be to choose the architecture of the network. We will use a standard feed-forward network. In table 3 we see the results for different number of layers and neurons. The results are an average of four runs to get at better picture of how the networks perform. All test are ran using the standard gradient descent method and 100000 iterations in training. This was chosen after studying the development of the loss function as a function of number of iterations. For all the tests done, the loss flattened after around 100000 iterations. Running for higher number of iterations for larger networks was very time-consuming. However, when the most promising model is found we will try to run for longer and for finer grids to see how the results compares to the Forward Euler scheme.

Looking at table 3 we see the results from testing the network for different architectures. For only one hidden layer the network with 50 neurons got the best MSE. The time it takes to train the networks does not vary to much between 10,50 and 100 neurons, but we still see that the time increases when the number of neurons increases. This is as expected. When adding another hidden layer, the MSE's does not improve very much. However, the time it takes to train the network almost doubles. The best MSE for two hidden layers was obtained for 50 neurons in both layers. Comparing this to one layer with 50 neurons, the difference is approximately 0.25e-5. The results are a bit better, but since the time to train the network doubles, we will use the network with one layer with 50 neurons for the rest of the analysis.

Neurons	Max abs error	MSE	time [s]
10	0.0173	3.16e-5	42
50	0.0131	2.05e-5	52
100	0.0186	2.76e-5	59
10,10	0.18	0.0053	51
50,50	0.015	1.75e-5	97
100,100	0.019	2.5e-5	142

Table 3: Table of error for different neural network structures. All values are calculated by taking the average of 4 runs. Learning rate = 0.01. Number of iterations = 100000. All layers have the sigmoid activation function.

With the chosen architecture with one hidden layer with 50 neurons we can now try to find the best learning rate and regularization parameter. The regularization parameters are added to the loss function to restrict the size of the weights in the network. For more info on regularization parameters, see the report for project 2 or [4]. The results from testing for different combinations are shown in figure 6. The most important parameter seems to be the learning rate. For the regularization parameter, the MSE's does not change much when the learning rate are held constant. The results from using different learning rate are significant. We see that the best results are for learning-rate=0.01. One thing to keep in mind is that we might have ran to few iterations for the smaller learning rates to be able to give good results. However, the convergence rate is then extremely slow, so we will use the learning-rate = 0.01. We also see that the best results are obtained by not using any regularization parameter, and given by MSE = 1.25e-5.

We can now summarize the findings from the discussion. We will use a feedforward neural network with one hidden layer with 50 neurons. The hidden layer have the sigmoid activation function. For training the neural network we use the standard gradient descent method with a learning rate =0.01 and a loss function with no regularization. We can now start to compare the neural network to the forward Euler scheme.

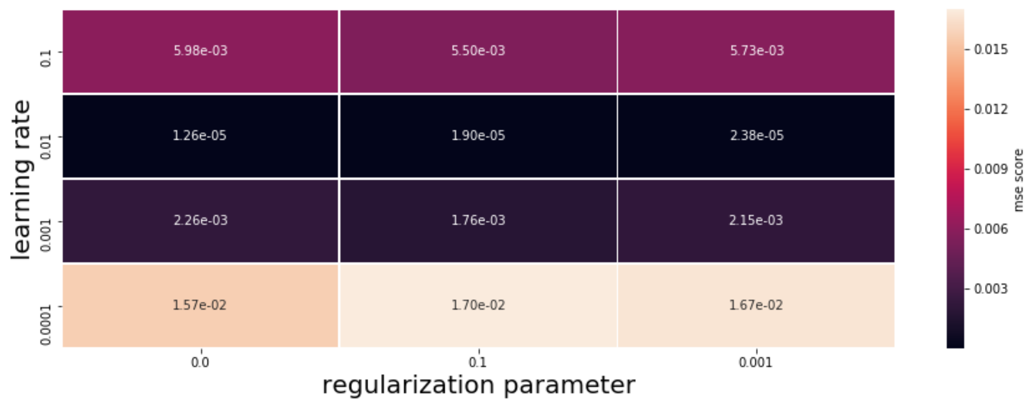

Figure 6: Heatmap of the MSE for solving the diffusion equation using a neural network with one layer with 50 neurons. The layer have the sigmoid activation function. Number of iterations = 100000.

4.1.3 Comparison between NN and Forward Euler

The benefit with the neural networks is that we get a continuous function as output, compared to a discretized domain from the finite difference method. This is a benefit if we want to use the output for further calculations. However, we will see that the performance of neural networks is not as good as for the Forward Euler method.

We will now test the optimal network found in the previous section and compare it to the forward Euler method with $\Delta x = 0.1$ and $\Delta t = 0.005$. This is at the stability limit for the forward Euler scheme, so we will be able to test the accuracy of both methods. The time used for calculating the solution is also compared for each method.

As discussed, the best network was the one with one layer 50 neurons, learning rate = 0.01 and no regularization. In addition, we use the standard gradient descent optimizer with both 100000 and

200000 iterations.

In figure 7 - 10 we see the results from running both the neural network and the forward Euler method. The neural networks is trained with both 100000 and 200000 iterations. We can start by looking at the case with t = 0.01. The solution is still curved. We see in the left part of figure 7 where the network is trained with 100000 iterations that both methods yields an acceptable solution. However, the error is approximately three times higher for the neural network compared to Forward Euler. In addition, the time it uses to fit the neural network to obtain this accuracy is way higher than for the Forward Euler method. The neural network used 774 seconds while the Forward Euler method uses zero seconds (lower than machine precision) to obtain the solution.

Method	MSE, t=0.3	MSE, t=0.02	time [s]
NN (10^5 iter)	3.2e-4	2.8e-3	774
NN ($2 * 10^5$ iter)	3.1e-4	9.1e-4	1337
FE	3.13e-5	3.56e-5	0

Table 4: MSE for the neural networks and Forward Euler for different times. $\Delta t = 0.005$ and $\Delta x = 0.1$.

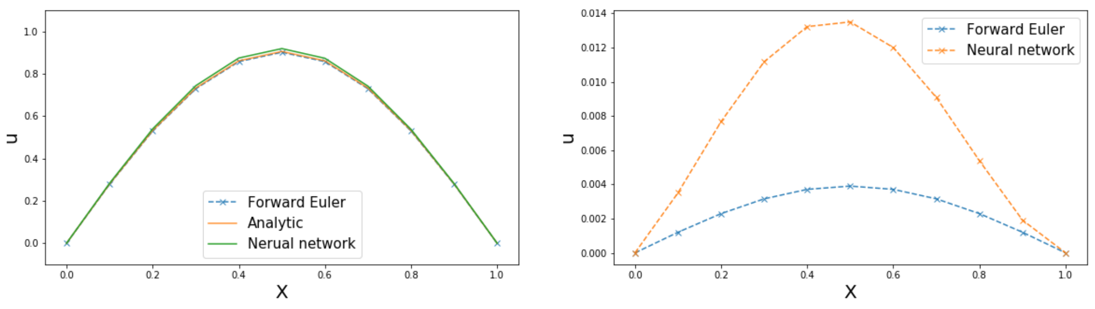

Figure 7: Results and error for forward euler and neural network. Left: Solution from the neural network and the forward Euler method. Right: Error of the neural network and the forward Euler method. t = 0.01, $\Delta x = 0.1$ and $\Delta t = 0.005$. Number of iterations = 10^5

In figure 8 the results from training the neural network for 200000 iterations is shown. The time it took to train the network was very high, 1337 seconds. Now we can see that the solution is better than before and closer to the results obtained using forward Euler. The error is approximately halved, However, they are still worse than for the Forward Euler method.

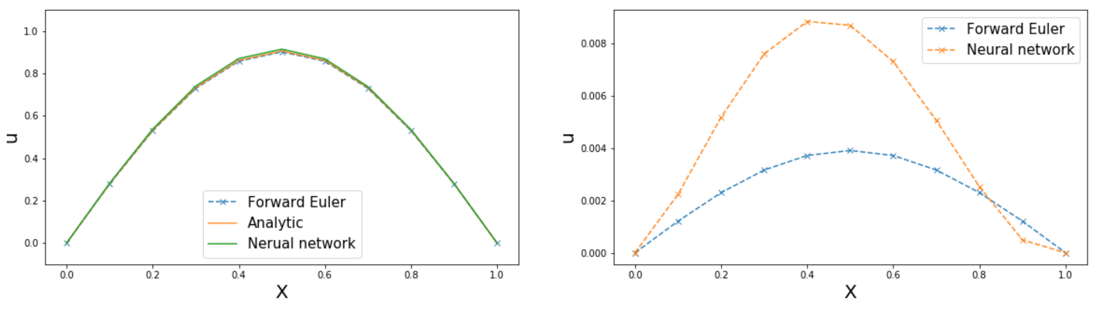

Figure 8: Results and error for forward euler and neural network. Left: Solution from the neural network and the forward Euler method. Right: Error of the neural network and the forward Euler method. t = 0.01, $\Delta x = 0.1$ and $\Delta t = 0.005$. Number of iterations = $2 * 10^5$

We can then look at a time where the temperature is almost linear and closer to the steady state. In figure 9 the same network is used, however, we now look at t = 0.2. In figure 9 the results are shown for 100000 iterations. Again the Forward Euler method is the best. Increasing the number of iterations to 200000 gives better results. In figure 10 the neural network solution is actually better for a part of the domain. However, in general the solution is worse than what we got with the Forward Euler method.

4.2 Eigenvalues

In this section we will look at the results from finding the eigenvectors of symmetric matrices using the neural network. We will focus on the results from solving the problem with a neural

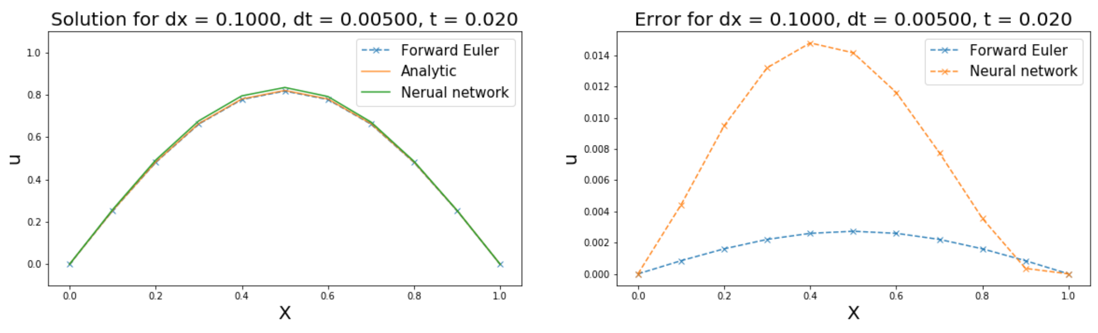

Figure 9: Results and error for forward euler and neural network. Left: Solution from the neural network and the forward Euler method. Right: Error of the neural network and the forward Euler method. t = 0.2, $\Delta x = 0.1$ and $\Delta t = 0.005$. Number of iterations = 10^5

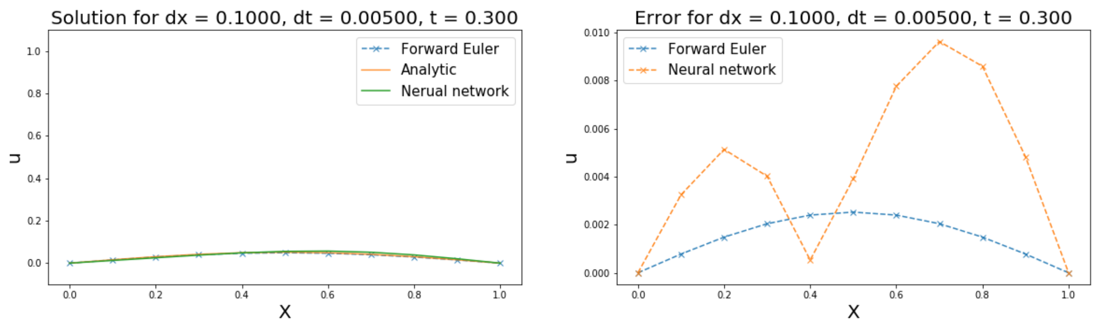

Figure 10: Results and error for forward euler and neural network. Left: Solution from the neural network and the forward Euler method. Right: Error of the neural network and the forward Euler method. t = 0.2, $\Delta x = 0.1$ and $\Delta t = 0.005$. Number of iterations = $2 * 10^5$

network with one hidden layer with 50 neurons. The learning rate will be set to 0.001. Using a higher learning rate often leads to the solution blowing up. All other combinations that actually converged gave the exact same eigenvectors as the one with the described architecture. For this part we will therefore focus on the results instead of comparing different parameter combinations. This is because we wont get any better results by using any other methods. The matrices can be constructed by generating a random matrix B and then let

$$A = \frac{B + B^T}{2}. \tag{66}$$

For this example we will use the matrix

$$A = \begin{bmatrix} 0.73226725 & 0.47787427 & 0.61724428 & 0.87025615 & 0.26483054 & 0.46805582 \\ 0.47787427 & 0.39464279 & 0.31475892 & 0.26076868 & 0.24813765 & 0.47866796 \\ 0.61724428 & 0.31475892 & 0.83864138 & 0.8378511 & 0.33978901 & 0.60903542 \\ 0.87025615 & 0.26076868 & 0.8378511 & 0.29715185 & 0.93241803 & 0.64813176 \\ 0.26483054 & 0.24813765 & 0.33978901 & 0.93241803 & 0.85834939 & 0.41329663 \\ 0.46805582 & 0.47866796 & 0.60903542 & 0.64813176 & 0.41329663 & 0.96960967 \end{bmatrix} \tag{67}$$

The precision is set to 0 for the following analysis. This means that the loss have to go to zero before the algorithm stops. The results from using the neural network is given by

$$v_{\min} = \begin{bmatrix} -0.34367058 \\ 0.13466038 \\ -0.21292647 \\ 0.81898913 \\ -0.37410143 \\ -0.08788837 \end{bmatrix}, \qquad v_{\max} \begin{bmatrix} -0.42200925 \\ -0.25821362 \\ -0.44667138 \\ -0.47008322 \\ -0.37705239 \\ -0.4388301 \end{bmatrix} \tag{68}$$

with corresponding calculated eigenvalues $\lambda_{\min} = -0.73845294$ and $\lambda_{\max} = 3.37070192$. The network converged after 5000 iterations. To compare the results from the neural network we will use numpy's functionality to calculate the eigenvalues and eigenvectors. The results from numpy is

$$v_{\min}^{\text{numpy}} = \begin{bmatrix} -0.34367059 \\ 0.13466039 \\ -0.21292647 \\ 0.81898911 \\ -0.37410143 \\ -0.08788837 \end{bmatrix}, \qquad v_{\max}^{\text{numpy}} \begin{bmatrix} -0.42200924 \\ -0.2582136 \\ -0.44667138 \\ -0.47008322 \\ -0.37705239 \\ -0.4388302 \end{bmatrix} \tag{69}$$

with corresponding eigenvalues $\lambda_{\text{min}}^{\text{numpy}} = -0.73845294$ and $\lambda_{\text{max}}^{\text{numpy}} = 3.37070192$. The results are very accurate and the eigenvalues are exactly the same. We can calculate the MSE of the eigenvectors and they are of order 1e-18, which extremely good.

We can now look at the convergence of the components of the eigenvector corresponding to the largest eigenvalue. In figure 11 we see how the components of the eigenvectors corresponding to the largest eigenvalue converges. The estimates of the eigenvalues seems to reach the values after

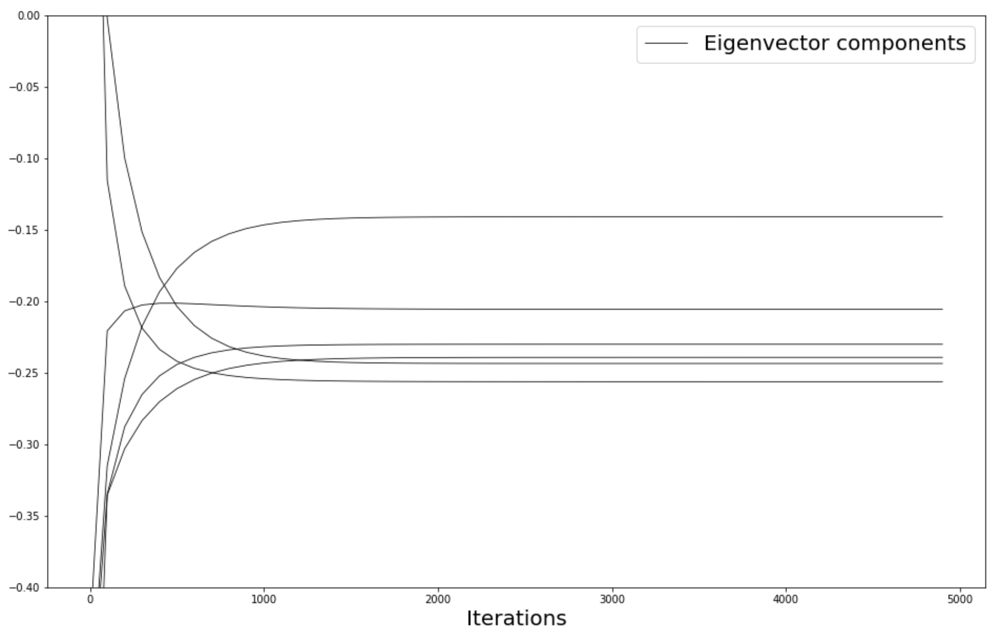

Figure 11: Convergence of the components of the eigenvector corresponding to the largest eigenvalue.

around 2000 iterations. It does not reach the desired loss value of 0 before 5000 iterations. This happens after 2 seconds.

We can also try to find the eigenvector corresponding to second largest eigenvalue by choosing an initial state that is orthogonal to the eigenvector corresponding to the largest eigenvalue as described in the theory. The initial vector is found with the following python code

```
k = eigen_vecs[:,0]
x -= x.dot(k) * k / np.linalg.norm(k)**2
```

where k is the eigenvector to the largest eigenvalue and x is a random vector with the same dimension as the eigenvectors. The problem is that due to numerical error, the dot product between k and x is not always exactly 0. However, when it is we can get results as below. It seems to be random but for some runs the wanted eigenvectors was obatained. An example is below where the eigenvectors corresponding to the second largest eigenvalue was found

$$v = \begin{bmatrix} 0.34367059 \\ 0.13466039 \\ 0.21292646 \\ -0.81898912 \\ 0.37410143 \\ 0.08788837 \end{bmatrix}, v^{\text{numpy}} = \begin{bmatrix} 0.34367058 \\ 0.13466039 \\ 0.21292643 \\ -0.81898912 \\ 0.37410144 \\ 0.08788837 \end{bmatrix} \tag{70}$$

Again, we see that the method is very accurate, and the eigenvalues obtained are exactly the same for numpy and the neural network method, $\lambda_2 = -0.73845294$.

5 CONCLUSION

In this report we have studied the diffusion equation. We have looked at two different solution methods, the classic forward Euler method and a more recent approach using neural networks. The forward Euler method was implemented in python from scratch using various libraries to optimize the speed. The neural network was implemented using tensorflow to make it easier to write efficient code.

The best results was obtained using the Forward Euler method. This was also the case after doing hyper-parameter tuning for finding the optimal neural network for this problem. The accuracy was better for all tests conducted within the stability limit. For a grid with $\Delta t = 0.005$ and $\Delta x=$ 0.1 the MSE from using Forward Euler was 3.13e-5 for t = 0.3 and 3.56e-5 for t=0.02. For the neural network we got 3.1e-4 and 9.1e-4 for the same t-values. However, the main benefit pointing towards Forward Euler being the best method is the computational time. For the best results using the neural network it trained for 1337 second. Forward Euler time was lower than machine precision so the difference was huge. However, the problem with Forward Euler is the stability limit. A slight breach of the limit leads to the solution blowing up. This was not the case for neural networks. However, for the problem studied in this report, the Forward Euler method was the best choice. There was no problem in choosing the grid so that Δt and Δx fulfilled the stability criteria. If one want to simulate over a longer time frame it might lead to problems if one want Δx small. Then Δt have to be small, and this can lead to having to take a lot of unnecessary steps to reach the final time. However, scaling the problem fixes this. All in all I would say that Forward Euler is the best method with todays methods and computational power.

After that we studied the possibilities of finding the eigenvalues of symmetric values using a similar neural network as we used for the diffusion equation. The results for matrices of size up to 10x10 was very good. The Eigenvalues found was exactly the same as the ones obtained by numpy. In addition, the MSE of the eigenvectors was of order 1e-18.

As it stand today, neural network does not compete against the more traditional methods. The accuracy is not as good and the computational time is useless compared to the Forward Euler method. It would be interesting to see if the computational time would decrease if the network was trained on a more powerful computer with a GPU. It would let us train the network for longer, and we could probably improve the results. However, as the Forward Euler method is so fast, we could just increase the number of grid points and probably get an even better solution. It will be interesting to see if the developments in computing power and new methods for solving PDE's will change that in the future.

For the eigenvalue problem we have only focused on symmetric matrices. It would be interesting to study Whether the methods works for non-symmetric matrices. Using more powerful hardware for trying to find the eigenvalues of larger matrices using a smaller step-length would also be an idea for future study.

References

[1] Jensen M. H., *Computational Physics, lecture notes 2015*, Oslo, 2015.

[2] Yi Z., Fu Y., *Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix*, Computers Mathematics with Applications **47**, `https://www.sciencedirect.com/science/article/pii/S0898122104901101` (2004).

[3] Tveito A., Winther R., *Introduction to Partial Differential Equations: A Computational Approach*, Springer, 1998.

[4] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn* & *TensorFlow.* O'REILLY, The United States of America, 2017.

[5] Hein B. H., *Using neural networks to solve ODEs and PDEs*, `https://compphysics.github.io/MachineLearning/doc/pub/odenn/html/odenn.html`, Oslo, 2018

[6] Lagaris I. E., Likas A., Fotiadis D. I. *Artificial neural networks for solving irdinary and partial differential equaions*, IEEE transactions on neural networks **95**, `https://www.semanticscholar.org/paper/Artificial-neural-networks-for-solving-ordinary-and-Lagaris-Likas/5ebbe0b1a3d7a2431bbb25d6dfeec7ed6954d633` (1998).