18.12.2019

Solving Differential Equations using Neural Networks

Martin Heltberg, Eirik Storrud Røsvik & Odin Midbrød Sanner

https://github.com/EirikSR/FYS-STK-4155-Project-3

Table of Contents

Abstract

In this project, two neural networks have been created to explore applicability of the models in solving partial differential equations (PDE), specially the diffusion equation, and finding the extreme eigenpairs of a real matrix. When solving for the diffusion equation, the neural network is compared against a finite differential equation, while solving for eigenpairs, NumPy's own linear algebra functions was the comparison. Our findings show that neural networks can solve differential equations, while having some problems finding the minimal eigenpair. The finding is that the neural networks are fairly applicable in both cases, being a flexible but computing heavy approach.

Introduction

Solving partial differential equations is an important task in many different sciences, amongst them physics. It is therefore important to have and use an efficient and precise algorithm to solve these equations. This is an old problem and many different algorithms has been created throughout the years. Some solve more complex DEs than others, but higher order methods usually are slower and require more computing power. Using Neural Networks to solve these PDEs are therefore interesting. Neural networks can approximate any functions and are in general a good candidate. Neural networks offer a good accuracy/efficiency trade-off for several different equations and often even outperforms the old way of doing it. Therefore, a comparison between the neural network and the forward Euler method will be done. In addition, the neural network will be tasked with finding extrema eigenpairs, and this will be compared with the NumPy's own linear algebra functions.

Theory

Forward Euler's method

Euler's method also called explicit forward Euler's algorithm, is a numerical mathematical method used to solve first order differential equations with given initial values (Bui, 2010).

If there is a function f(x) and the initial values of f(x) is known (x_0) as well as f `($x_0$), then by using Euler's method, it is possible to find any approximation of a point on the f(x) curve by using the tangent line to the graph as shown in figure 1. What Euler's method suggest is that f `(x_0) estimates the change in y for each change in x. x increase by the step size h so therefore y should increase approximately by the function h * f `($x_0$). So $x_1 = x_0 + h * f`(x_0)$ (Khan, 2019).

The size of h determines how accurate the approximated solution is and how many steps needs to be computed. Choosing smaller step sizes or h reduces the error between the actual and approximated. By halving the step size, it will roughly half the error however this doubles the number of computations.

These two figures show the accuracy of using different step sizes, figure (bottom) is more accurate as it uses more steps with smaller h values (Khan, 2019).

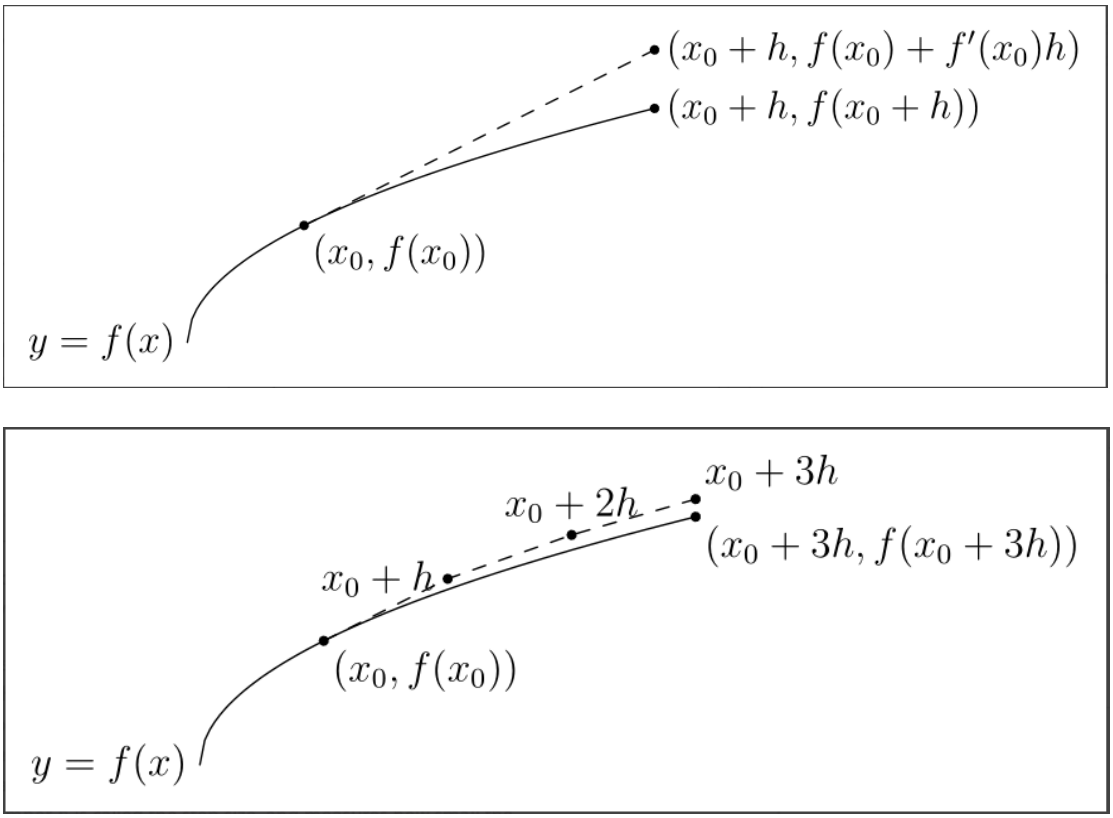

Figure 1: Theory behind Euler's method from: Khan et al. (2019)

Diffusion equation

A simple diffusion equation can be express as (Munster, Unknown):

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \quad t > 0 \;\; x\varepsilon[0,L] \qquad (1)$$

The initial conditions when t=0 and L=1 is expressed by:

$$u(x,0) = \sin(\pi x) \qquad (2)$$

Equation 1 is the same as the heat equation, which can describe the distribution of heat of a region in a given time.

In order to get a precise comparison when solving PDEs differently, first calculating the exact solution is needed in order to calculate the error the method. By using separation of variables, we get an equation:

$$u(x,t) = X(x)T(t) \qquad (3)$$

Then by substituting u back into equation 1 we get:

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)} \qquad (4)$$

Since both sides of the equation is dependent to some constant, for the right side it is depended on the constant x and for the left side it is depended on the constant t. We can choose a constant for them both to be equal to $(-\lambda^2)$. This will then give two equations:

$$X''(x) + \lambda^2 X(x) = 0 \qquad (5)$$

$$T''(t) + \lambda^2 T(t) = 0 \qquad (6)$$

When looking at the initial conditions given in equation 2, the only way to satisfy this for X it needs to be expressed as:

$$X(x) = B\sin(\lambda x) + C\cos(\lambda x) \qquad (7)$$

By substituting the boundary conditions leads to C=0 and $\lambda=\pi$

T(t) gives the equation:

$$T(t) = Ae^{-\lambda^2 t} \qquad (8)$$

As shown earlier $\lambda=\pi$ and from the initial condition AxB=1, this gives the exact solution to be:

$$u(x,t) = e^{-\pi^2 t}\sin(\pi x) \qquad (9)$$

Discretization of the diffusion equation

For the discretization of the diffusion equation, the Euler method will be used (Langtangen, 2015) , Eulers method is explained earlier in the report. The error of the time discretization is gives by Δt.

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x,t+\Delta t) - \mathrm{u(x,t)}}{\Delta t} \qquad (10)$$

The error of the spatial discretization is given by Δx^2.

$$\frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u(x+\Delta x,t) - 2\mathrm{u}(\mathrm{x},\mathrm{t}) + u(x-\Delta x,t)}{\Delta x^2} \qquad (11)$$

We can simplify this with notations $u_i^n = u(x_i, t_n)$, then the discrete form of the equation will be expressed as:

$$u_{xx} = u_t$$

$$\frac{u_{i+1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} = \frac{u_i^{n+1} - u_i^n}{\Delta t} \qquad (12)$$

From this it is possible to calculate each next time- step for i by solving for u_i^{n+1}:

$$u_i^{n+1} = \frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i+1}^n) + u_i^n \qquad (13)$$

This will have stability level for:

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2} \qquad (14)$$

PDEs with neural network

When using neural network to solve PDE, the goal is to get an approximated trial function $\psi(x,t)$ as similar as possible to the true function $u(x,t)$. When solving the PDE with neural network we substitute the trial function into equation 1:

$$\frac{\partial^2 \psi(x,t)}{\partial x^2} = \frac{\partial \psi(x,t)}{\partial t} \quad t > 0 \;\; x\varepsilon[0,L] \qquad (15)$$

The error of the trial function will be:

$$E = \frac{\partial^2 \psi(x,t)}{\partial x^2} - \frac{\partial \psi(x,t)}{\partial t} \qquad (16)$$

The neural network will update the trial function for each iteration based on previous calculations preformed, this is why it is important to choose a form that fits our trial function well. We chose the following equation because this correlates with the Dirichlet condition and initial conditions of equation 1.

$$\psi(x,t) = (1-t)I(x) + x(1-x)tN(x,t,p) \qquad (17)$$

I(x)= the initial condition, N(x,t,p)= the output of the NN and p= weight and bias.

For each iteration done the trial function $\psi(x,t)$ is calculated according to the updated neural network output N(x,t,p) and thereby it updates the cost. As the cost function gets smaller, the error E will converge towards zero and the trial function $\psi(x,t)$ will give a solution for PDE. How much it is possible to minimize the error is of the trial function is depended on the maximum number of iterations possible by the neural network.

Computing eigenpairs with Neural Networks

In Yi et a. (2004), a way to compute the eigenvector, v_{max}, that belongs to the largest eigenvalue, w_{max}, of the n × n matrix A. This is done by solving the ordinary differential equation:

$$\frac{\partial v(t)}{\partial t} = -v(t) + f\big(v(t)\big), t \geq 0 \qquad (19)$$

Here $v = [v_1, v_2, \dots, v_n]^T$, $f(v) = [v^T vA + (1 - v^T Av)I]v$ and I is the identity matrix for the n × n matrix. When t approaches infinity, any random non-zero vector initial v, will approach v_{max}, as long as v is not orthogonal to v_{max}. To compute the subsequent eigenvalue, w_{max}, one can use the following equation:

$$w = \frac{v^T Av}{v^T v} \quad y \qquad (20)$$

To find the eigenvector that belongs with the smallest eigenvalues, v_{min} and w_{min} respectively, simply replace A with – A in equation 20.

To solve equation 19 with a neural network, a trial function is needed. Since $v \in R^n$, a trial function that is dependent on both position and time was chosen. Approximated eigenvectors are thus given as $[v(1,t), v(2,t), \dots, v(n,t)]^T$. This leads to the rewriting of equation 19:

$$\frac{\partial Y(x,t)}{\partial t} = -Y(x,t) + f\big(Y(x,t)\big), t \geq 0, x = 1,2, \dots, n \qquad (21)$$

The trial function was then defined as:

$$Y(x,t) = v_0 + tN(x,t,p) \qquad (22)$$

Where v_0 is the initial v, that is randomly chosen.

Results

Diffusion Equation

Using the Euler method, the diffusion equation was calculated and plotted as a surface, with the x-axis representing points along the rod, and the z-axis the time. The height of the

surface in the y-axis represents the calculated value in this time and position; in our case it simulates temperature. Each time (t) will have a corresponding curve displaying the heat along the rod. As can be seen in figure 2 for (dx =0.1) the resolution is course in along the x-axis while smoother in along the z axis. This derives from the stability criterion (eq. 14) requiring a finer resolution for the time steps. As can be seen in figure 4 and 5, the difference is almost none for the initial step, which is not too surprising, as the Euler method incorporates the initial values from the analytic solution (as time = 0) For the finer time resolution the same patterns are seen, with the difference gradually growing from the initial conditions to a peak, then reclining.

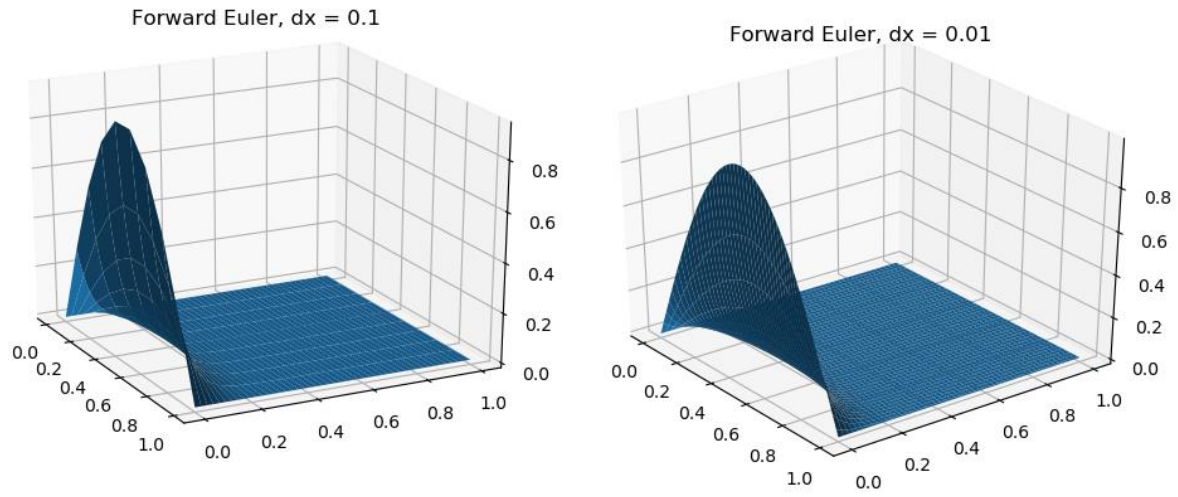

Figure 2 & 3: Forward Euler's solution of the diffusion equation, for dx=0.1 and dx=0.01

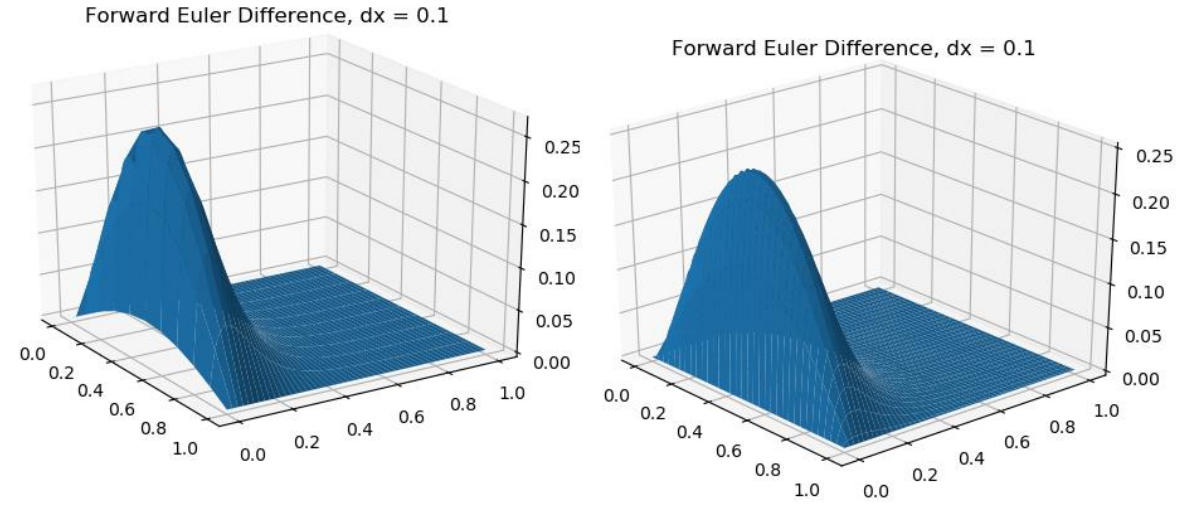

Figure 4 & 5: The difference between forward Euler and the analytical method for (4) dx=0.1 and (5) dx = 0.01

The neural network was computed using dx = 0.1. The final cost being 0.0042 computed as the MSE between the analytic and computed surfaces. Observing figure 7 the neural network follows a similar difference trend as the Euler method with initial conditions resulting in low difference for low t-values and growing to a peak in the areas of the surface corresponding to the downslope seen around t=0.1 in figure 8. The general surface produced by the neural network seen in figure 6 from the Euler method surface in that the neural network doesn't need more values for t than for x, so it appears courser.

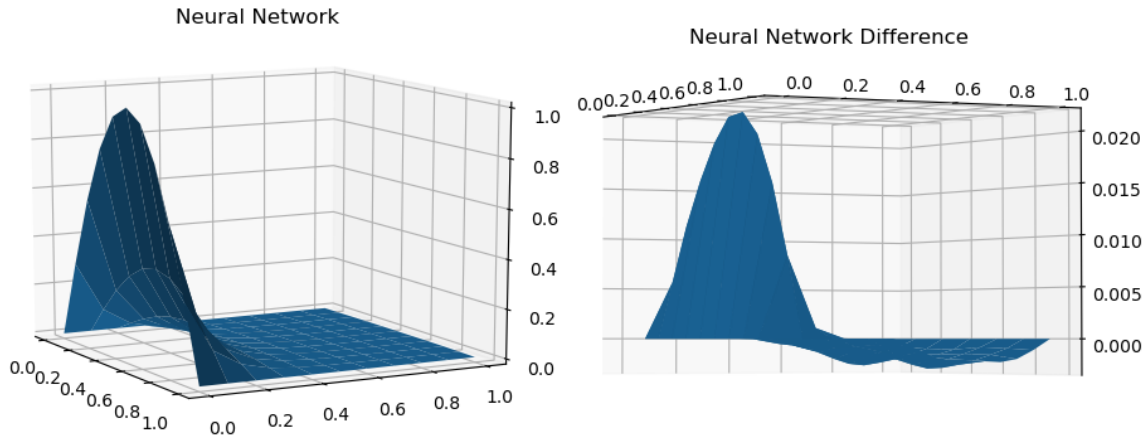

Figure 6 & 7: (6) The neural networks solution of the diffusion equation, with (7) the difference between the analytical solution and the neural networks solution.

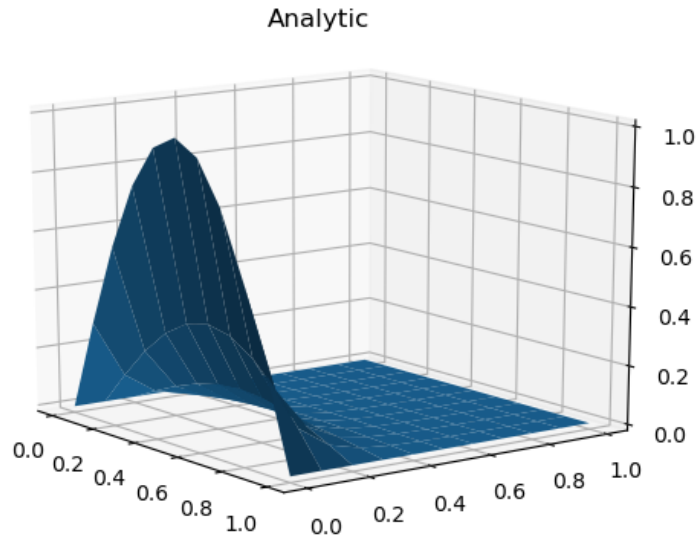

Figure 8: Analytical solution of the diffusion equation.

Eigenpairs

The eigenpairs are taken from matrix A, which can be seen in the appendix. The maximum eigenvalue calculated with numpy.linalg is given as w_{max}^{np}, with the subsequent eigenvector v_{max}^{np}. The same eigenpair, calculated with a neural network is v_{max}^{nn}, and w_{max}^{nn}. All the eigenvalues have been normalized to allow for better comparison between the eigenvalues. As seen the neural network is fairly good at finding the maximum eigenvalue and eigenvector. Generally, the neural network differs on the second or third decimal place in the eigenvector, while the eigenvalues differs with about 0.002.

$$v_{max}^{np} = \begin{matrix} 0.41172028 \\ 0.42491252 \\ 0.32718456 \\ 0.53446463 \\ 0.34427791 \\ 0.37335479 \end{matrix}, w_{max}^{np} = 3.381671$$

$$v_{max}^{nn} = \begin{matrix} 0.40635684 \\ 0.41582151 \\ 0.32614925 \\ 0.5250986 \\ 0.36157058 \\ 0.37367318 \end{matrix}, w_{max}^{nn} = 3.379778$$

Applying the neural network on matrix A, leads to figure 9. The figure shows how v_{max}^{nn} and $w_{max}^{nn}$is developing over time. w_{max}^{nn} nearly the NumPy linear algebra functions solution, and all the eigenvalue elements are converging on the solution from NumPy. It is also clear that the NumPy solution is much quicker than the neural network, with a computation time in the milliseconds as opposed to the neural network using around one minute.

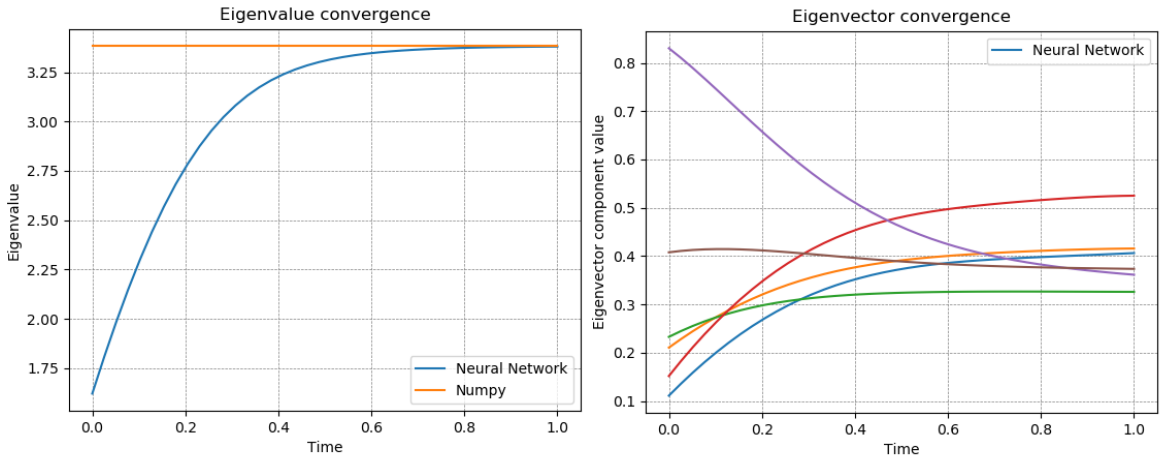

Figure 9: The changing in the value of the maximum eigenvalue and its corresponding eigenvector. For the eigenvector plot, each line is one of the elements of the eigenvector.

In addition to finding the largest eigenpair, the smallest eigenpair was also computed. The eigenpair computed with numpy.linalg is given as v_{min}^{np}, and w_{min}^{np}. While the eigenpair computed with a neural network is given as v_{min}^{nn}, and w_{min}^{nn}.
For the minimum eigenpairs, the neural network seems to be struggling. Both the eigenvalue and the eigenvector are missing the solution but forward by NumPy by quite a big margin. The eigenvalue is off already in the first decimal place, while the elements of the eigenvector are often off by as much as 1.

$$v_{min}^{np} = \begin{matrix} -0.00729076 \\ -0.12606344 \\ -0.40921278 \\ -0.3340569 \\ 0.18387711 \\ 0.81926489 \end{matrix}, w_{min}^{np} = -0.637130$$

$$v_{min}^{nn} = \begin{matrix} 0.30027464 \\ 0.0837312 \\ 0.0583052 \\ 0.35471336 \\ -0.81857839 \\ -0.29903486 \end{matrix}, w_{min}^{nn} = -0.467632$$

Figure 10 shows a plot of how v_{min}^{nn} and $w_{min}^{nn}$is developing over time. This was done by substituting A with -A in the cost function. As seen the eigenvalue is converging towards a value greater than the minimum eigenvalue. The neural network does not reach as low as the NumPy computed eigenvalue, not surprising as the eigenvector is completely off.

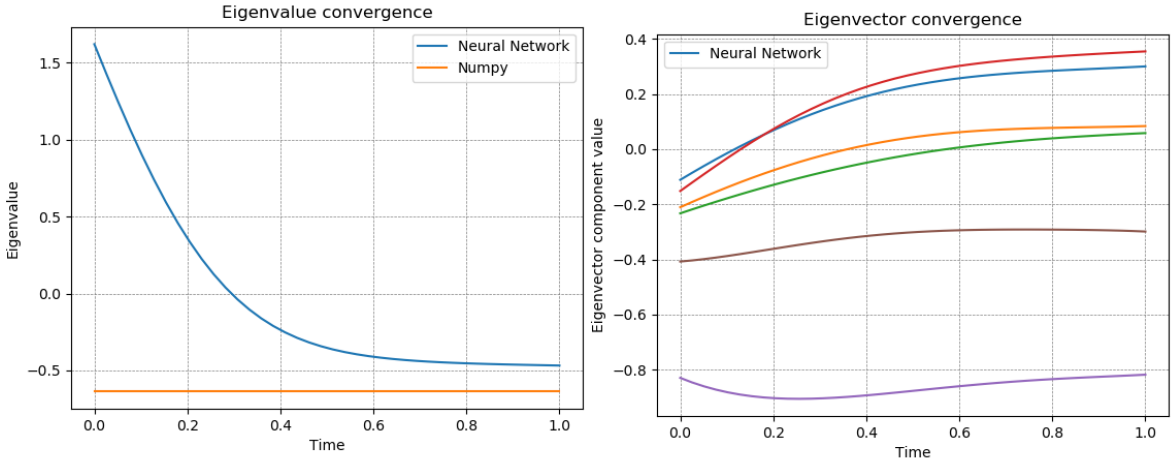

Figure 10: The changing in the value of the minimum eigenvalue and its corresponding eigenvector. For the eigenvector plot, each line is one of the elements of the eigenvector.

Discussion

The results from the partial differential equation show that a neural network approach is possible. The neural network produces results similar to those obtained by finite difference equations such as the Euler method.

The difference observed when comparing the Euler method to the analytic solution is larger at the start, but as these values are based on absolute difference, the relative difference is presumably lower, as the lower differences are where the computed values are lower. The neural network computes a lower difference for the peak values for low t, while the difference computed for the higher t-values are considerably noisier than the Euler method. This probably stems from the different methods of computation; the Euler method uses more time steps and gets more precise for each step, while the neural network estimates all parts of the surface simultaneously. The difference in computation needed makes the Euler method likely to be the preferred option for most applications. As can be seen in figure 4 and 5, the resolution along the x-axis does not appear to be corresponding with a higher accuracy when compared to the analytical.

Regarding the eigenpairs, the neural network struggles a bit more. Calculating the Eigenvalues using the neural network proved to be quite variable in time usage, depending on the initial vector and the matrix. Some matrices would be complete by 5000 iterations, while other took over 20000. Some random matrices and initial vectors also proved to find Eigenvalues with a corresponding vector differing from NumPy's corresponding vector for the same eigenvalue. In general, pairs of random matrices and vectors with lower initial cost had lower computation time. This was explored using the different random seeds.

Solving for the minimum eigenvalue, the neural network fails. The values seem to be converging towards a value about 0.2 higher than the actual value for all minimum eigenvalues. The approach used was to simply invert all the values in the real matrix. Changing the cost function with either subtracting or adding the initial vector produced the same value. This error is likely to be in the cost function, as the models calculated MSE is comparable for both min and max eigenvalues. All computations for the minimum eigenvalue ran until the maximum number of iterations were reached, not reaching the cutoff point for precision.

Further Work

The most important thing to fix for this neural network is the struggle with computing the smallest eigenvalue, and its eigenvector. This is a big problem and should be fixed before doing anything else on the network. The method for finding the eigenpairs only gives out the smallest and largest eigenpairs, with the subsequent eigenvectors. To find other eigenvalues, more work must be done with the neural network with a different cost function. This also leads to the neural network not being able to compute eigenpairs of real symmetric matrix. Further improvements could also include finding eigenpairs for not only symmetrical matrixes. A better implementation or better computer power might solve it.

Conclusion

When the neural network has solved the diffusion equation and this is compared with a finite definite solution, the results are quite good. The neural networks difference from the analytical method is smaller than Euler's method, and generally the neural network solves differential equations well. The time it takes for the neural network to train and later compute the equation is on the other hand slower than Euler's method, a definite solution.
A neural network was also tasked to find the extreme Eigen pairs of a real, symmetric matrix, by solving an ordinary differential equation. Comparing the results to NumPy's analytical calculation, it was found that the neural network solves well for the largest Eigenvalue, while underperforming when calculating the min Eigen value. While for the maximum values, the corresponding calculated Eigen vector is close to the analytic, this was not the case for the minimum.

References

Brownlee, Jason (2016) "Linear Regression for Machine Learning" From: https://machinelearningmastery.com/linear-regression-for-machine-learning/ Downloaded: 05.12.19,

Bui, Timothy, (2010) "Explicit and Implicit Methods in Solving Differential Equations" Honors Scholar Theses. 119. https://opencommons.uconn.edu/srhonors_theses/119 Downloaded: 09.12.19

Chapter 6 "The finite difference method" From: https://www.ljll.math.upmc.fr/frey/cours/UdC/ma691/ma691_ch6.pdf Downloaded: 10.12.19

Khan et al. (2019) "Euler's method" From: https://brilliant.org/wiki/eulers-method/ Downloaded: 14.12.19

Langtangen, Hans Petter (2015) «The 1D diffusion equation» From: http://hplgit.github.io/num-methods-for-PDEs/doc/pub/diffu/sphinx/._main_diffu001.html Downloaded: 10.12.19

Munster University (Unknown) “The Diffusion equation” From: https://www.uni-muenster.de/imperia/md/content/physik_tp/lectures/ws2016-2017/num_methods_i/heat.pdf Downloaded: 09.12.19

Zhang Yi, Yan Fu, and Hua Jin Tang. Neural networks-based approach for computing eigenvectors and eigenvalues of symmetric matrix. Computers & Mathematics with Applications, 47(8-9):1155–1164, 2004.

Appendix

```
[[0.5557906  0.38191063 0.32705763 0.89895635 0.7321493  0.43496982]
 [0.38191063 0.85760612 0.48611947 0.69786825 0.38229057 0.67496548]
 [0.32705763 0.48611947 0.24575605 0.58633785 0.34111468 0.6812225 ]
 [0.89895635 0.69786825 0.58633785 0.73632338 0.74704752 0.80061204]
 [0.7321493  0.38229057 0.34111468 0.74704752 0.14548493 0.36468141]
 [0.43496982 0.67496548 0.6812225  0.80061204 0.36468141 0.05546416]]
```

Appendix 1: Matrix A, that the minimum and maximum eigenpairs are taken from.