Solving partial differential equations using neural networks

Daniel Pinjusic and Gudmund Gunnarsen

December 23, 2020

Abstract

In this project we looked at the performance of a neural network versus a more traditional discreet method such as the explicit forward Euler method for solving partial differential equations, and using a neural network for solving eigenvalue problems. We find that a neural network is capable of solving a partial differential equitation to a higher accuracy than the forward Euler, with less data points required, however the network performs slower. We also find that a neural network can solve for the eigenvalues of a symmetric matrix with a high degree of accuracy, but that the speed of computation is much slower. In the future it would be of interest too study whether it is possible to get a faster implementation of our neural network, such that it did not lose in speed to more traditional methods.

1 Introduction

The study of partial differential equations(PDE) is one of great importance in many fields of science, including many fields of physics such as particle- and fluid mechanics as well as sciences such as biology and chemistry. This is because such equations are particularly good at describing real physical problems, however, they are often too complex to solve by hand with no easily defined solutions and computationally demanding to solve, even with efficient algorithms. Considering the growth of neural networks the last few decades and their continued scope of usage, it is of definitive interest to assert weather or not such a network could deliver performance on the same level of current methods, or even better.

Another mathematical concept that has usage in many fields of science is the concept of eigenvalues. Eigenvalues have many uses, but computing the eigenvalues of matrices quickly becomes computationally heavy when matrix sizes increase. We want to explore how a neural network performs when tasked with computing the maximum and minimum eigenvalues of a symmetric matrix.

We will in this project look at approximating the diffusion equation using one of the most common methods of approximating PDEs, namely the explicit forward Euler method and compare it to the result of our own neural network built using *tensorflow* [4]. Then we will adapt our network to solve for the eigenvalues of a symmetric matrix, showing the adaptability of a neural network. Finally, we will make comparisons of our computed eigenvalues with those computed using the *numpy* [1] Python library to check how our neural network compares to standard numerical eigenvalue solvers.

First, we will present some of the theory and methods used in this report, then we will present our results before discussing our results and finally end the report with some concluding remarks.

2 Method and theory

In this section we will cover the theory of the diffusion equation, explaining some numerical methods for solving it and doing a brief summary of the theory pertaining to finding the eigenvalues of a symmetric matrix using a neural network.

2.1 The diffusion equation

The diffusion equation in one dimension can be written on the form

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

with initial condition $u(x,0) = g(x)$ for $0 < x < L$ and with boundary conditions

$$u(0,t) = a(t) \quad t \geq 0 \tag{2}$$
$$u(L,t) = b(t) \quad t \geq 0. \tag{3}$$

The functions $a(t)$ and $b(t)$ are dependent on time only and in our case we are interested in the special case when $a(t) = b(t) = 0$. Function g will be set to $g(x) = \sin(\pi x)$.

The analytical solution to the diffusion equation using these boundary conditions is given by

$$u(x,t) = B_n \sin\Big(\frac{\pi n}{L}x\Big)e^{-\left(\frac{\pi n}{L}\right)^2 t}, \tag{4}$$

as derived in appendix B (6.2). Using $L = 1$ gives $B_1 = 1$ for $n = 1$. Then the solution reduces to

$$u(x,t) = \sin(\pi x)e^{-(\pi)^2 t}. \tag{5}$$

2.2 Explicit forward Euler

The explicit forward Euler algorithm is a finite difference method where one approximates the derivatives of a function with finite differences. This means that we will approximate a continuous function to a finite number of steps.

The diffusion equation be can be discretized as

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \tag{6}$$

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

which can be further simplified using $u_{i,j} = u(x_i, t_j)$ and combined such that

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

Solving for $u_{i,j+1}$ then gives

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

where $\alpha = \frac{\Delta t}{\Delta x^2}$ is the stability criterion which has to conform with $\alpha \leq \frac{1}{2}$ for the approximation to reach a stable solution.

Algorithm 1: Forward Euler scheme

Result: $u(x,t)$
$u_1(x) = g(x)$;
$t = 0$ **while** $t \leq T$ **do**
  $u(x) = \alpha u_1(x+1) + (1-2\alpha)u_1(x) + \alpha u_1(x-1)$;
  $u(0) = 0$;
  $u(x_n) = 0$;
  $u_1, u = u, u_1$;
  $t \mathrel{+}= \Delta t$;
end

2.3 Partial differential equations in neural networks

A general partial differential equation for a function $u(x_1, x_2, \ldots, x_N) = u(\vec{x})$ with N variables can be written as

$$f\left(\vec{x}, \frac{\partial u(\vec{x})}{\partial x_1}, \ldots, \frac{\partial u(\vec{x})}{\partial x_N}, \frac{\partial^2 u(\vec{x})}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n u(\vec{x})}{\partial x_N^n}\right) = 0 \tag{10}$$

To make this equation solvable by a neural network, we will use a trial solution on the form

$$u_t(\vec{x}) = h_1(\vec{x}) + h_2(\vec{x}, N(\vec{x}, P) \tag{11}$$

where $h_1(\vec{x})$ is a function that ensures $u_t(\vec{x})$ satisfies some given conditions and $h_2(\vec{x}, N(\vec{x}, P))$ is a function that uses the output of the neural network $N(\vec{x}, P)$ with weights and biases P to ensure that it is zero when u_t is evaluated at certain $\vec{x}$ where the given conditions must be satisfied.

A neural network will try to minimize f for a set of parameters P. We will her consider the mean squared function as the loss function, and apply it to f. We then get

$$C(\vec{x}, P) = f\left(\vec{x}, \frac{\partial u(\vec{x})}{\partial x_1}, \ldots, \frac{\partial u(\vec{x})}{\partial x_N}, \frac{\partial^2 u(\vec{x})}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n u(\vec{x})}{\partial x_N^n}\right)^2 \tag{12}$$

If we have M different sets of values for $\vec{x}$ such that $\vec{x}_i = (x_1^{(i)}, \ldots, x_n^{(i)})$ for $i = 1, \ldots, M$ are the rows in the matrix $\mathbf{X}$, then we can further generalise the cost function into

$$C(\mathbf{X}, P) = \sum_{i=1}^{M} f\left(\vec{x}_i, \frac{\partial u(\vec{x}_i)}{\partial x_1}, \ldots, \frac{\partial u(\vec{x}_i)}{\partial x_N}, \frac{\partial^2 u(\vec{x}_i)}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n u(\vec{x}_i)}{\partial x_N^n}\right)^2. \tag{13}$$

In the case of the diffusion equation, the cost function can be stated as

$$C = \frac{1}{N_x N_t} \sum_i \left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)^2. \tag{14}$$

where N_x and N_t are the number of discreet values for x and t respectively, with a possible trial solution

$$u_t(x,t) = (1-t)\sin(\pi x) + x(1-x)t \cdot N(x,t,P), \tag{15}$$

as described by K. Hein. [2]

2.4 Eigenvalues of symmetric matrices

We will here do a short introduction to the theory of finding the eigenvalues, specifically the minimum and maximum eigenvalues, of a symmetric matrix. Since most of the theory is covered in the study done by Yi et al [6], we will simply restate the most pertinent material.

First, let us consider a $n \times n$ matrix A that is defined as

$$A = \frac{Q^T + Q}{2}, \tag{16}$$

where Q is any randomly generated real matrix. Obviously A is a symmetric matrix as $(Q + Q^{\mathrm{T}})^{\mathrm{T}} = Q + Q^{\mathrm{T}}$. Given an eigenvector v of A, it is possible to calculate the corresponding eigenvalue λ by the Rayleigh Quotient

$$\lambda = \frac{v^{\mathrm{T}} A v}{v^{\mathrm{T}} v}. \tag{17}$$

Now all that remains is to establish a method of finding the eigenvector v of A, or more precisely finding v_{max} of A. In Yi et al. [6] the differential equation

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)), \tag{18}$$

for $t \geq 0$, where

$$f(x) = [x^{\mathrm{T}} x A + (1 - x^{\mathrm{T}} A x) I] x,$$

and $x = (x_1, \ldots, x_n)^{\mathrm{T}} \in R^n$ is proposed as a neural network model where x represents the state of the network and I is the $n \times n$ identity matrix. This is a dynamic non linear system, and is the system we will base our own model on. Furthermore, as proven by Yi et al. [6], the solution of this system will converge on the eigenvector which corresponds to the largest eigenvalue. Contrary, to find the eigenvector corresponding with the smallest eigenvalue, it is simply enough to train the network on the negative of the symmetric matrix.

2.5 Optimising the cost function

To optimise the loss of our cost function, we will test several optimisers. In particular, the stochastic gradient descent method ADAM [3] is of great interest.

Any other hyperparameters associated with a given optimiser will be handled by the *tensorflow* library.

3 Results

For all calculations pertaining to diffusion equation we use $L = 1$, and $n = 1$ so $B_1 = 1$ for the analytical solution. The network has just one hidden layer with 300 neurons and is trained with a learning rate of $\eta = 0.001$ using the ADAM optimiser. The activation on the hidden layer for the diffusion case is the tanh function, while the activation for the eigenvalue case is the sigmoid [5]. For the eigenvalue a problem, we use a hidden layer with 10 neurons.

3.1 Forward Euler

Figures (1) and (3) show approximation made using the explicit forward Euler method. Figure (2) shows the approximation using the forward Euler method over all $0 \leq t \leq 1$ and $0 \leq x \leq L = 1$, with a comparison to the analytical solution using the absolute error ϵ.

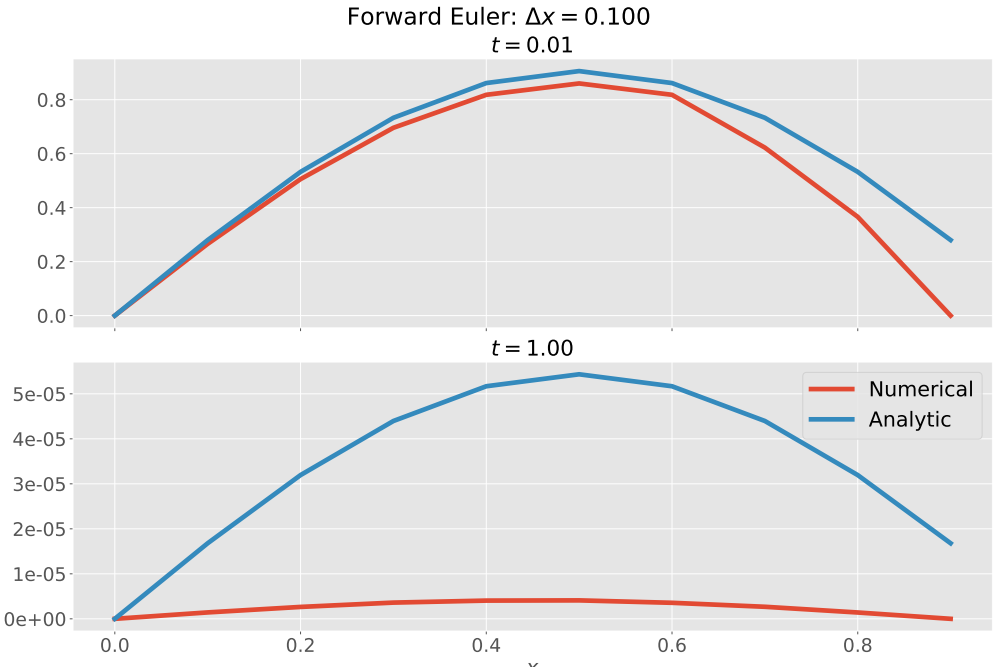

Figure 1: Plot showing an approximation of the diffusion equation and the associated analytical solution using the explicit forward Euler method for $\Delta x = 0.1$ for specific times $t \approx 0$ and $t \approx 1$

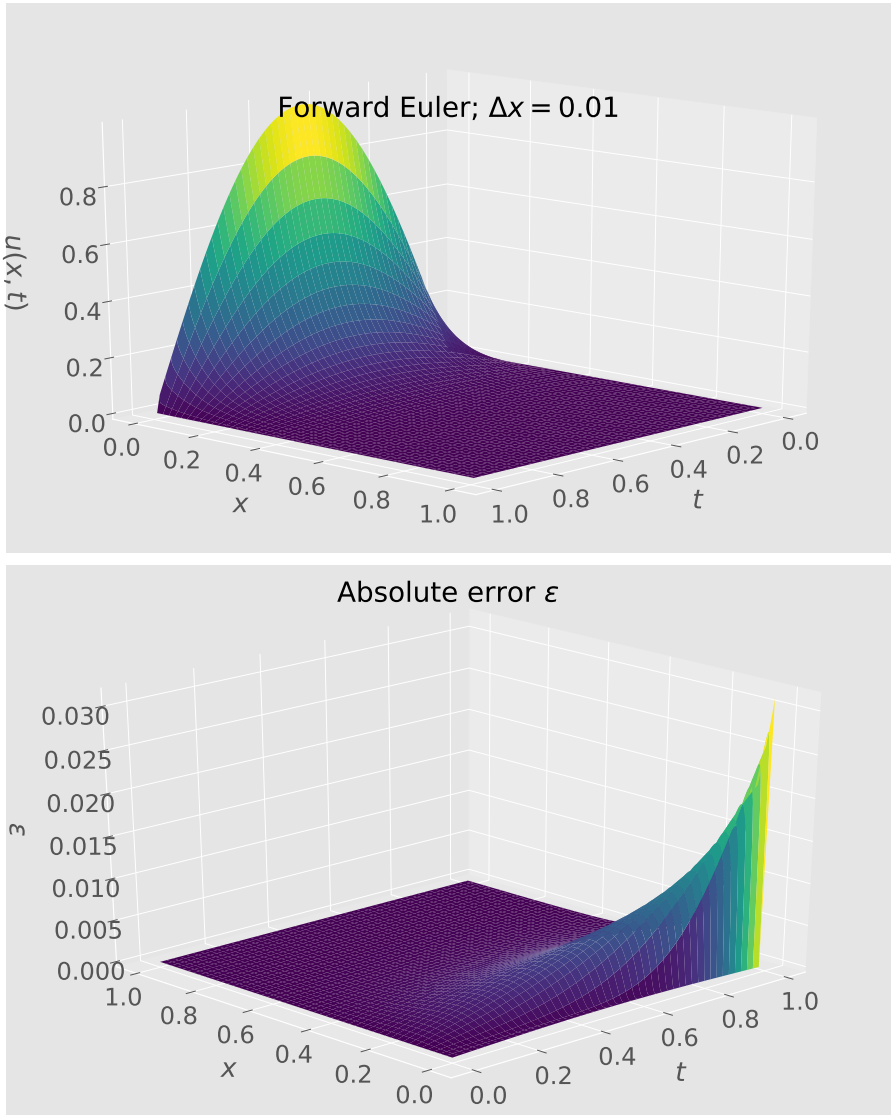

Figure 2: Plot showing an approximation of the diffusion equation and the associated error with respect to the analytical solution using the explicit forward Euler method

3.2 Neural network trained on the diffusion equation

In figure (4) we see a plot of different optimisers built into *tensorflow*, compared with respect to the mean squared error calculated during training.

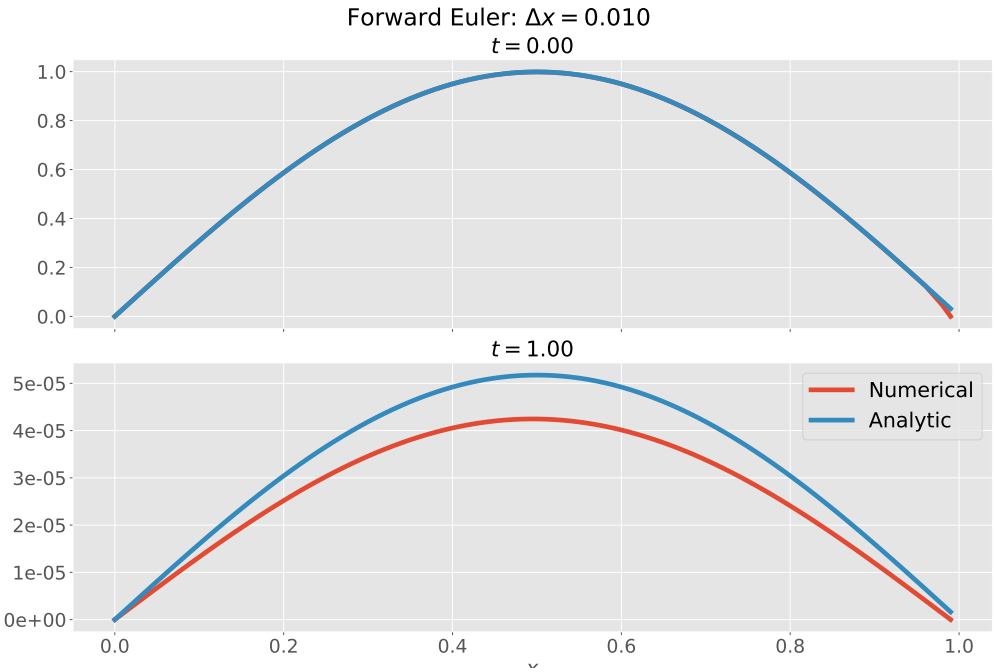

Figure 3: Plot showing an approximation of the diffusion equation and the associated analytical solution using the explicit forward Euler method for $\Delta x = 0.01$ for specific times $t \approx 0$ and $t \approx 1$

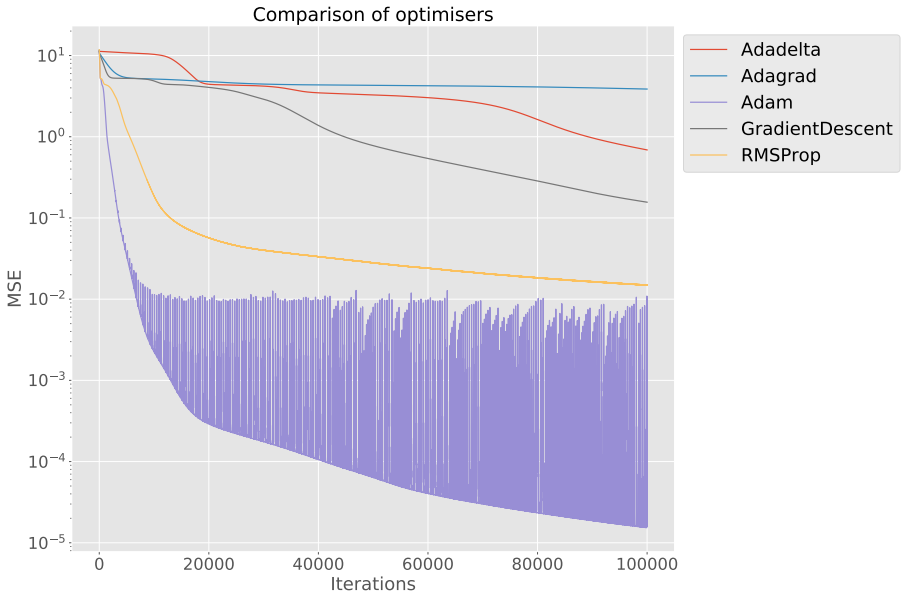

Figure 4: A comparison of the mean squared error(MSE) of several optimisers as a function of iterations. The network is trained on a grid consisting of $N = 100$ discreet values, where $N = N_x N_t$ with $N_x = N_t = 10$.

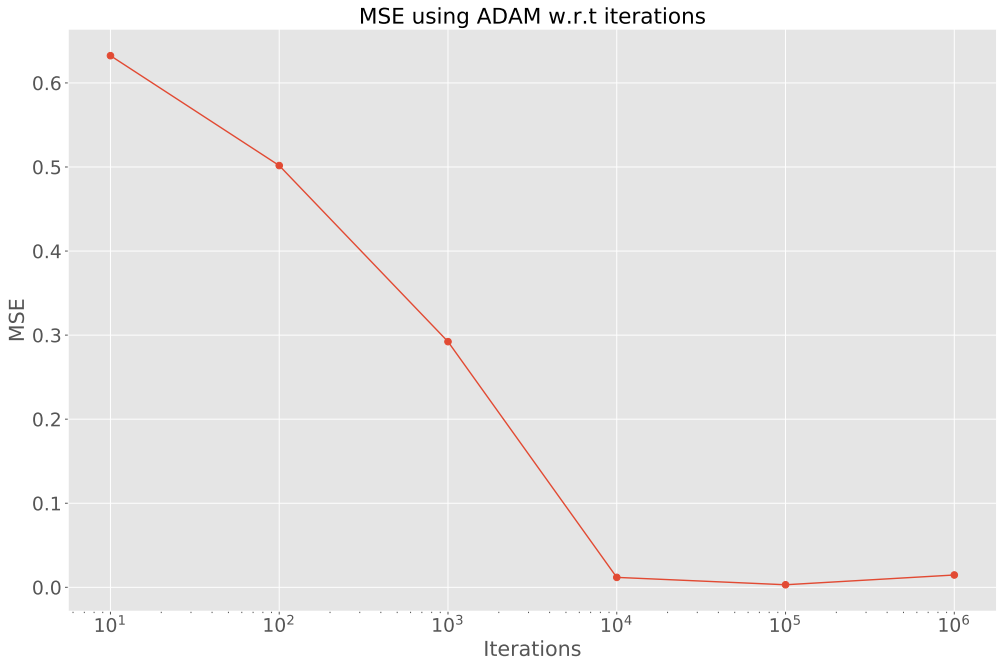

Figure 5: A comparison of the mean squared error(MSE) during training when minimising the loss with ADAM. The network is trained on a grid consisting of $N = 100$ discreet values, where $N = N_x N_t$ with $N_x = N_t = 10$.

3.3 Finding the eigenvalues of a symmetric matrix

Training our network on the randomly generated 6×6 matrix

$$\mathrm{A} = \begin{bmatrix} 0.0750 & 0.7889 & 0.6589 & 0.2967 & 0.4067 & 0.7699 \\ 0.7889 & 0.1085 & 0.4101 & 0.5166 & 0.5248 & 0.3668 \\ 0.6589 & 0.4101 & 0.2428 & 0.9234 & 0.8391 & 0.6635 \\ 0.2967 & 0.5166 & 0.9234 & 0.2311 & 0.1299 & 0.2346 \\ 0.4067 & 0.5248 & 0.8391 & 0.1299 & 0.8136 & 0.2954 \\ 0.7699 & 0.3668 & 0.6635 & 0.2346 & 0.295473 & 0.1800 \end{bmatrix}$$

we get the results as stated in table 3.3.

	λ_{max}	λ_{min}
numpy	2.94283628...	-1.04244393...
NN	2.94283628...	-1.04244393...

Table 1: Comparison of eigenvalues calculated using the *numpy* library standard eigenvalue calculator and our neural network implementation. λ_{max} value found when training the network with 10000 iterations, while λ_{min} value was found when training with 50000 iterations.

The absolute error of λ_{max} computed by the neural network and the *numpy* implementation is $\delta = 3.11 \cdot 10^{-15}$ and the error for λ_{min} is $\delta = 3.95 \cdot 10^{-11}$.

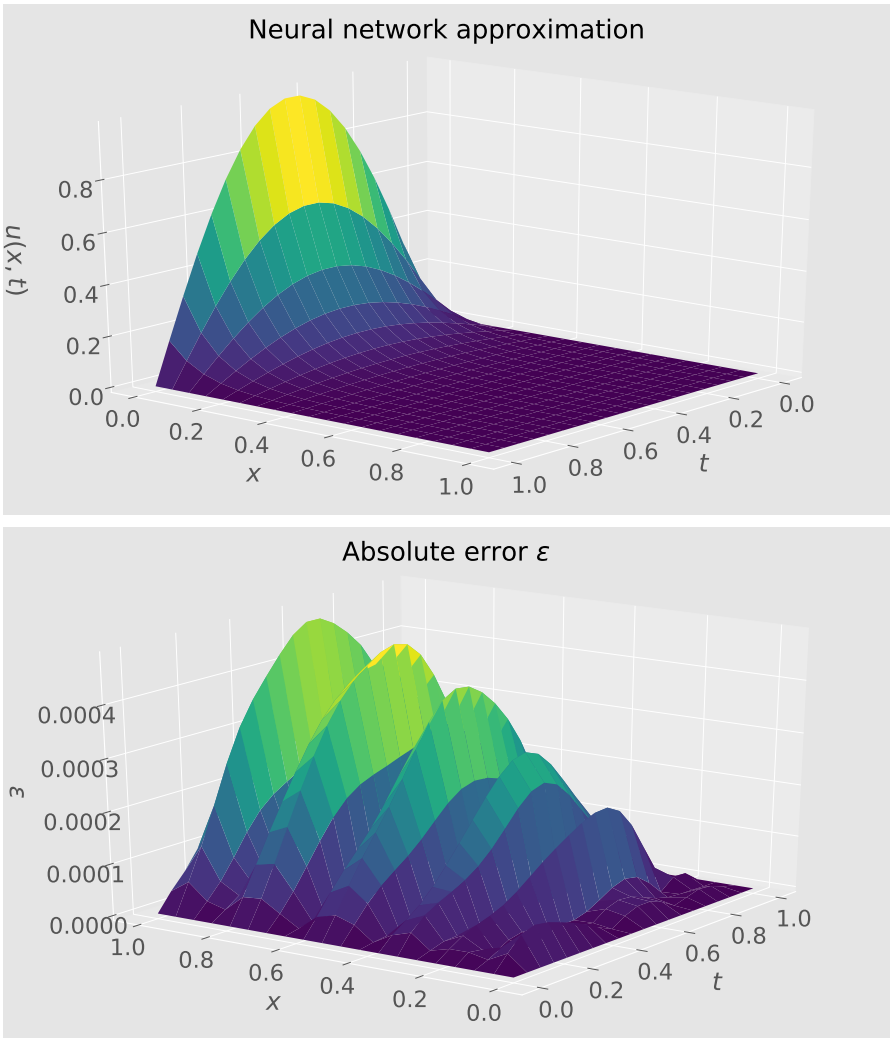

Figure 6: Plot showing an approximation of the diffusion equation and the associated error with respect to the analytical solution using our neural network with the optimal parameters derived from above results.

4 Discussion

4.1 Diffusion equation

When using the forward Euler explicit scheme method, we can see that when Δx gets smaller and Δt also gets smaller according to the relation $\Delta t = \frac{1}{2}\Delta x^2$ that the numerical solution approaches the analytical solution from below.

The neural network solution is considerable slower than the forward Euler for any Δx, but does not suffer from a weak stability condition like the explicit scheme. However, finding the appropriate number of layers and neurons for the network is not simple. There is also the aspect of finding what activation function works the best for the problem at hand, as well as what kind of optimization function to use.

4.2 Finding the optimal network setup

As can be seen in figure (4), the stochastic gradient descent method ADAM minimised the cost function the best. Not only does the ADAM method reach a satisfactory low error for less iterations than the other methods, but it also achieves the overall lowest error, although the error does almost jump by a factor $\sim 10^3$ for some iterations, but still staying lower then the second best method, the RMSprop method either way.

It is not strange that these two methods would perform relatively similarly, as they are both based on stochastic gradient descent with an adaptive learning rate. However, in this case, the ADAM method was the most accurate for all of our test. It is of note however, that the ADAM method has such a huge (10^{-2}- $\sim 10^{-5}$) variation in mean squared error when iterations increase to ≈ 10000.

When we then tested the training of our network over a different number of iterations, using the ADAM optimiser, as seen in (5) we found that iterations in the range of $10^4 - 10^6$ performed to a satisfactory degree, being capable of producing absolute max errors δ_{max} of $\delta_{max} \approx 0.005$. It is of note that when iterations go up to 10^6, that the mean squared error actually increases. This might be due to overfitting happening when the number of iterations become to many and the learning periods therefore become too long. Since we allow *tensorflow* to deal with the optimisation of any hyperparameters, we can not be sure that the correct choice of hyperparameters were chosen either. It might also be the case that this particular result is just a product of randomness, and that just one more iteration would again decrease the mean squared error. When considering figure (4) again, it becomes clear that the mean squared error is very unstable, and this does add weight to the argument that the mean squared error for 10^6 is simply due to the unstable nature of the error. Either way, we find that the iterations $\sim 10^5$ produce accurate results with low compute time.

4.3 Neural network performance

From figure (6) we can see the approximation of the diffusion equation made by our neural network and the corresponding absolute error ϵ compared with the analytical solution. It is clear that the network does a good job of approximation, having a maximum absolute error of just ~ 0.0005. This produces In comparison to the forward Euler

4.4 Finding the eigenvalues of a symmetric matrix

From table (3.3) we can see that computed solutions for the eigenvalues of the symmetric matrix are very similar, with very low errors. This tells us that using a neural network to calculate the eigenvalues of a symmetric matrix is definitively feasible, although our network implementation was significantly slower at calculating just the maximum and minimum eigenvalues, as opposed to the time it took for the *numpy* implementation to calculate all eigenvalues of A. This however, might be due to a lack of optimisation on our part, or simply training our model for longer than required.

4.5 Determinism

Using the *tensorflow* network, we found that there was a distinct lack of determinism in our results, even when using random seeds. We speculate that there is something within *tensorflow* behind the curtains which is causing this non-deterministic behavior. Due to this, some training session with the network could produce particularly bad results. This was particularly the case when training the network to solve for the maximum

eigenvalue of a symmetric matrix. These training session would produce very variable results. When training the network on the diffusion equation, on the other hand, the results were more predictable, and more importantly, more reproducible.

5 Conclusion

We find that using a neural network to solve partial differential equations can produce satisfactory results which when compared to traditional methods such as the forward Euler method are more accurate, with a difference in maximum error of $\Delta\delta \approx 0.025$. We also find that using the stochastic gradient descent method ADAM produces low mean squared errors during the training of the network, but that the these mean squared error values are very volatile.

We conclude that using a neural network to solve for the maximum and minimum eigenvalues of a symmetric matrix is a viable solution. Our findings gives the absolute difference between the computed solution using *numpy* and our neural network for the biggest and smallest eigenvalues as $\delta = 3.11 \cdot 10^{-15}$ and $\delta = 3.95 \cdot 10^{-11}$. respectively.

Finally, we find that using the tensorflow library as the backbone to our network may produce unintended issues, as lack of deterministic properties may make it difficult to train and reproduce results.

6 Appendix

6.1 A

Github repository with codes and figures can be found at `https://github.com/Gudmunhg/ML-projects`.

6.2 B

To find the analytical solution to the diffusion equation as defined in (1) we start by using separation of variables.

We assume that there are two functions $\varphi(x)$ and $G(t)$ and that the function $u(x,t)$ is equal to the product of these two. We can then write

$$u(x,t) = \varphi(x)G(t) \tag{19}$$

If we know insert (19) into the expression for the diffusion equation (1) we get

$$\frac{\partial^2(\varphi(x)G(t))}{\partial x^2} = \frac{\partial(\varphi(x)G(t))}{\partial t} \tag{20}$$

$$G\frac{d^2\varphi}{dx^2} = \varphi\frac{dG}{dt} \tag{21}$$

$$\frac{1}{\varphi}\frac{d^2\varphi}{dx^2} = \frac{1}{G}\frac{dG}{dt} \tag{22}$$

From (22) we can see that the the terms on both sides of the equal sign must be equal to the seperation constant λ. We can therefore write

$$\frac{1}{\varphi}\frac{d^2\varphi}{dx^2} = -\lambda \tag{23}$$

and

$$\frac{1}{G}\frac{dG}{dt} = -\lambda. \tag{24}$$

We have now simplified the diffusion equation, which is a second order partial differential equation into two ordinary differential equations. Note that the choice of using $-\lambda$ as the separation constant is arbitrary, and is chosen as such so that our calculations will be easier later on. Before we solve (23) and (24) we will look at the boundary conditions again. We know that $u(0,t) = u(L,t) = 0$. This implies that either $\varphi(0) = \varphi(L) = 0$ or $G(t) = 0$. $G(t) = 0$ would lead to the trivial solution $u(x,t) = 0$, so we will instead assume that $\varphi(0) = \varphi(L) = 0$.

We then solve equation (23) as

$$\varphi'' + \lambda\varphi = 0. \tag{25}$$

To determine λ we will look at what happens when $\lambda > 0$, $\lambda = 0$ and $\lambda < 0$.

When $\lambda > 0$ the equation (25) can be rewritten as the characteristic polynomial

$$r^2 + \lambda = 0, \tag{26}$$

with solutions $r_{1,2} = \pm\sqrt{-\lambda}$. The general solution of (25) then becomes

$$\varphi(x) = c_1 \cos\Big(\sqrt{\lambda}x\Big) + c_2 \sin\Big(\sqrt{\lambda}x\Big). \tag{27}$$

Applying the boundary conditions on (27) then gives us

$$\varphi(0) = 0 \Rightarrow c_1 = 0 \tag{28}$$

$$\varphi(L) = 0 \Rightarrow c_2 = 0 \vee \sin\Big(\sqrt{\lambda}x\Big) = 0. \tag{29}$$

If $c_2 = 0$ then we get the trivial solution $\varphi(x) = 0$, so we assume that $c_2 \neq 0$ and that $\sin\Big(\sqrt{\lambda}L\Big) = 0$. This implies that $\sqrt{\lambda} = \frac{\pi n}{L}$ for $n = 1, 2, 3, \ldots$ and gives the solution in the case of $\lambda > 0$ as

$$\varphi(x) = c_2 \sin\Big(\frac{\pi n}{L}x\Big). \tag{30}$$

Next we will look at the case $\lambda = 0$. Equation (25) reduces to $\varphi'' = 0$ which gives the general solution $\varphi = c_1 + c_2 x$. Applying boundary conditions show that $c_1 = c_2 = 0$, and so $\lambda = 0$ can not be an eigenvalue for this problem.

Finally, we look at the case when $\lambda < 0$. Similarly as for the case of $\lambda > 0$, we can find a general solution on the form

$$\varphi(x) = c_1 \cosh\Big(\sqrt{-\lambda}x\Big) + c_2 \sinh\Big(\sqrt{-\lambda}x\Big). \tag{31}$$

Applying the boundary conditions gives us $c_1 = 0$ and $c_2 \sinh\big(\sqrt{-\lambda}L\big) = 0$. We are looking for solutions where $c_2 \neq 0$ so then $\sqrt{-\lambda}L = 0$. This could only be the case if $\lambda = 0$, however we know that $\lambda < 0$ so we can conclude that there are no negative eigenvalues. The eigenvalues for this problem must therefore be $\lambda_n = \big(\frac{\pi n}{L}\big)^2$, and the solution to the x-dependent function φ is $\varphi = c_2 \sin\big(\frac{\pi n}{L}x\big)$.

We can now solve (24), replacing λ with λ_n to represent the fact that there are infinitely many solutions. Then we get

$$\frac{1}{G}\frac{dG}{dt} = -\lambda_n \tag{32}$$

$$G = Ce^{-\lambda_n t} \tag{33}$$

$$G = Ce^{-\left(\frac{\pi n}{L}\right)^2 t}. \tag{34}$$

Combining the results from (30) and (32) we get the final analytical solution for the diffusion equation with boundary conditions $u(0,t) = u(L,t) = 0$ as

$$u(x,t) = B_n \sin\Big(\frac{\pi n}{L}x\Big) e^{-\left(\frac{\pi n}{L}\right)^2 t}, \tag{35}$$

where

$$B_n = \frac{2}{L}\int_0^L g(x) \sin\Big(\frac{n\pi x}{L}\Big) dx, \tag{36}$$

$n = 1, 2, 3, \ldots$ and $g(x)$ is the initial condition. The constants c_2 and C are absorbed into B_n, where the suffix n represents the fact that the constant B might be different for any n.

References

[1] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] Kristine Baluka Hein. *Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs*. URL: https://github.com/CompPhysics/MachineLearning/blob/master/doc/pub/odenn/pdf/odenn-minted.pdf. (accessed: 16.20.2020).

[3] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[4] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[5] David H. Von Seggern. *CRC Standard curves and Surfaces with Mathematica*. Apple Academic Press Inc, 2016.

[6] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix". In: *Computers Mathematics with Applications* 47.8 (2004), pp. 1155–1164. ISSN: 0898-1221. DOI: https://doi.org/10.1016/S0898-1221(04)90110-1. URL: http://www.sciencedirect.com/science/article/pii/S0898122104901101.