FYS-STK4155 Project 3: Solving differential equations using neural networks

Jan Ole Åkerholm
(Dated: December 19, 2019)

Abstract

Two methods are compared for solving differential equations. The more traditional explicit finite difference (FD) method using forward euler and centered difference approximations to the derivatives is compared with a deep neural network approach (DNN). The neural network approach is applied to a practical purpose when I attempt to compute the largest and smallest eigenvectors and eigenvalues of a symmetric, real and square random matrix, using the ideas outlined by Yi et al. [5]. The eigenpairs are compared to results found using the standard linear algebra functionality of Numpy. In general I find that the neural network usually requires more computation time, but has some advantages over the other methods. For example there is no stability criterion on the resolution of the mesh used for the spatial and temporal coordinates when using the NN approach, unlike for the FD approach, where the computed solution may become unstable if the user is careless with the mesh resolution. The results provided by the neural network matched the results generated by the traditional methods to some degree, and with sufficient computations appear to be able to match the traditional methods to an arbitrary precision.

Link to github repository: Click here

I. INTRODUCTION

Partial differential equations are a central topic in a wide range of sciences, including mathematics, physics, biology and many more. Finding analytical solutions of these equations are not always practical, so fast and accurate numerical methods are of great interest. Traditionally the methods of choice have often been finite difference or finite element methods. In particular finite difference methods use discrete numerical differentials to replace the exact mathematical ones from the equations, and calculate the solution from this. The advantage is that it is relatively easy to understand how this works, but the disadvantage is that the method may be inaccurate or even completely wrong if the differentials are not calculated appropriately.

A different approach is to take advantage of the fact that a neural network is able to approximate any function, and use this property to calculate the solution of a differential equation. There may be several advantages to this approach, first and foremost in the ability of the method to provide an accurate result. This has been attempted before, notably by Lagaris et al. [4] and Chiaramonte and Kiener [1], with promising results.

In addition, this method may even be applied to finding the eigenvalues and eigenvectors of a real, symmetric, square matrix. This is also something of great interest in many fields, and has previously been studied by Yi et al. with promising results [5].

II. THEORY AND METHOD

A. First PDE

The first partial differential equation (PDE) I will solve in this project is a specific case of the diffusion equation:

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

with $L = 1$, the initial condition at $t = 0$ is given by:

$$u(x, 0) = \sin(\pi x) \tag{2}$$

And I use Dirichlet boundary conditions:

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

The typical interpretation of the equation in this form is that it describes the temperature of a very thin (1D) rod of length L, which starts with a higher temperature in the middle while the temperature on the edges are kept at a constant of 0.

1. *Analytical solution*

An analytical solution of the PDE can be found without much problem, and is useful in order to compare the numerical results with what we know to be correct. First assume the solution can be separated into two factors, one dependant only on x and the other only on t, i.e.

$$u(x, t) = X(x)T(t), \quad X(0) = X(L) = 0 \tag{4}$$

Inserting in the differential equation (1):

$$T(t)\frac{dX(x)}{dx} = X(x)\frac{dT(t)}{dt} \tag{5}$$

$$\implies \frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)} \tag{6}$$

Because each side now only depends on the independent variables x and t respectively, each side must equal the same constant, $-\lambda$. This gives the two separate equations:

$$X''(x) + \lambda X(x) = 0 \tag{7}$$

$$T'(x) + \lambda T(t) = 0 \tag{8}$$

Looking at equation 7 first, note that this has different results depending on whether λ is larger than 0, smaller than 0 or equal to 0. First, if $\lambda < 0$, the characteristic equation gives the solution:

$$X(x) = c_1 \cosh(\sqrt{-\lambda}x) + c_2 \sinh(\sqrt{-\lambda}x)$$

From the boundary conditions, $X(0) = 0 = c_1$, and

$$X(L) = c_2 \sinh(\sqrt{-\lambda}L) = 0$$

$$\implies c_2 = 0$$

Because this is a trivial solution it is not very interesting, so all solutions for $\lambda < 0$ can be discarded.

Next, for $\lambda = 0$ the solution is simply $X''(x) = 0 \implies X(x) = c_1 + c_2 x$

From the boundary conditions, $X(0) = 0 = c_1$ and $X(L) = c_2 L = 0 \implies c_2 = 0$.

Again this makes the solution trivial, so solutions with $\lambda = 0$ can be discarded.

Finally, with $\lambda > 0$ the characteristic equation gives the solution

$$X(x) = c_1 \cos(\sqrt{\lambda}x) + c_2 \sin(\sqrt{\lambda}x) \tag{9}$$

Now, the boundary conditions gives:

$$X(0) = 0 = c_1$$

and

$$X(L) = 0 = c_2 \sin(\sqrt{\lambda}L) \tag{10}$$

For (10) to be true, the argument in the sine function must take the value $n\pi$ for $n = 1, 2, 3, ...$ ($n = 0$ gives a trivial solution). I.e.

$$\sqrt{\lambda_n}L = n\pi \implies \lambda_n = \left(\frac{n\pi}{L}\right)^2 \tag{11}$$

With $L = 1$ the solution for $X(x)$ then is:

$$X_n(x) = c_2 \sin(n\pi x) \tag{12}$$

With λ known, the solution for $T(t)$ (eq 8) can be found:

$$T_n(t) = C_n e^{-(n\pi)^2 t} \tag{13}$$

Combining (12) and (13), the general solution is obtained:

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

where c_2 has been combined into C_n. Next, the initial condition lets us find the final solution:

$$u(x,0) = C_n \sin(n\pi x) = \sin \pi x$$

It's clear that the solution is correct if $n = C_n = 1$, leaving the final analytical solution as:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t} \tag{14}$$

2. Explicit discretization scheme

For the finite difference scheme, the differential equation will be approximated using, as the name implies, finite differences for the differentials. First, the time differential is discretized using the Forward Euler (FE) discretization:

$$\frac{\partial u(x,t)}{\partial t} \simeq \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \tag{15}$$

The spatial differential is a second order derivative, so instead of using FE, this is discretized using a centered difference:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \simeq \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2} \tag{16}$$

where the simulation domain, $x \in [0,1]$ and $t \in (0,1]$ is divided into a discrete grid, such that $t_n = n\Delta t$ and $x_i = i\Delta x$ (note that $t = 0$ is not included in the simulation, as the initial condition is known, but t_0 still exists on the grid).

Using the notation $u_i^n = u(x_i, t_n)$, and the above discretizations (15 and 16), the differential equation can then be approximated as:

$$\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} = \frac{u_i^{n+1} - u_i^n}{\Delta t} \tag{17}$$

The solution can then be computed by calculating each next time step from the previous one, since the initial condition is known. Solving for u_i^{n+1}:

$$u_i^{n+1} = \frac{\Delta t}{(\Delta x)^2}\left[u_{i+1}^n - 2u_i^n + u_{i-1}^n\right] + u_i^n \tag{18}$$

Note that only the state of the previous time step is required to calculate the next one, and since the boundaries are always equal to 0, the solution can be calculated only for the internal points of x_i on the simulation grid.

This method has the stability criterion of

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

for which it is stable. For other values the results may be unpredictable and chaotic.

3. Solving the PDE using a deep neural network (DNN)

For an in-depth discussion of neural networks and how to solve differential equations with neural networks, see the notes by Morten Hjorth-Jensen [3] and Kristine Baluka Hein [2].

The general idea behind solving the PDE using a neural network is to create a trial function $u_t(x,t)$ which in some way depends on the output of the neural network, and also makes sure the initial and boundary conditions are met. The trial function is then substituted in the original differential equation (1) in order to find an approximation:

$$\frac{\partial^2 u_t(x,t)}{\partial x^2} = \frac{\partial u_t(x,t)}{\partial t}, \quad t > 0, \quad x \in [0,1]$$

The error of the approximation is then found by moving one term to the other side:

$$E = \frac{\partial^2 u_t(x,t)}{\partial x^2} - \frac{\partial u_t(x,t)}{\partial t} \tag{19}$$

A grid of t and x values are created, and the error function is calculated on each point on the grid. The error is squared, and the sum of this is the cost function which is to be minimized using a number of training iterations.

The trial function must be chosen in such a way that it fulfills the criteria given by the initial condition and boundary conditions, and it must also depend on the output from the neural network. In this project I used the following trial function:

$$u_t(x,t) = (1-t)I(x) + x(1-x)tN(x,t,P) \tag{20}$$

Where $I(x)$ is the initial condition (2) and $N(x,t,P)$ is the output from the neural network at position x, time t and with P representing the current set of weights and biases of the network.

Note that when $t = 0$, the trial function reduces to $u_t(x, 0) = I(x)$, and when $x = 0$ or $x = 1$, it reduces to 0 (since $I(0) = I(1) = 0$).

For each training iteration, the state of the network with the current weights and biases are calculated. The trial function and its derivatives are then calculated and the cost function evaluated. As the error E approaches 0, the trial function should then approach the solution of the PDE.

B. Finding eigenvectors and eigenvalues with neural networks

As explained by Yi et al. [5], the largest eigenvector of an $n \times n$, real and symmetric matrix A can be found by solving the ordinary differential equation

$$\frac{dv(t)}{dt} = -v(t) + f(v(t)), \quad t \geq 0 \tag{21}$$

where

$$v = [v_1, v_2, ..., v_n]^T \tag{22}$$

and

$$f(v) = [v^T vA + (1 - v^T Av)I]x$$

where I is the $n \times n$ identity matrix.

When $t \to +\infty$, $v(t)$ will then approach the largest eigenvector $v_{\max}$ of the matrix A, provided that $v(t)$ is initialized as some initial vector which is non-zero and not orthogonal to $v_{\max}$. The smallest eigenvector can be computed by substituting A with $-A$ for the computation.

Once an eigenvector is found, its corresponding eigenvalue w can be found from:

$$w = \frac{v^T Av}{v^T v} \tag{23}$$

1. Trial function

In order to use the same method as in the section for the PDE to solve the differential equation, a trial function is required. Because $v(t)$ is a vector containing n values dependant on time, this can be described as a two-dimensional equation in x and t, where $x = [1, 2, ..., n]$. The trial function used in this project is then:

$$v_t(x, t) = v_0 + tN(x, t, P) \tag{24}$$

where as before, $N(x, t, P)$ is the output from the neural network, and P describes the weights and biases at any iteration. v_0 is the initial v.

This can then be inserted in the differential equation (21):

$$\frac{\partial v_t(x, t)}{\partial t} = -v_t(x, t) + f(v_t(x, t)) \tag{25}$$

The error is then given by:

$$E = -v_t(x, t) + f(v_t(x, t)) - \frac{\partial v_t(x, t)}{\partial t} \tag{26}$$

This error squared and summed over all the time steps is then used as the cost function.

III. IMPLEMENTATION

The implementation for both the PDE and the eigenvalue problem is done with TensorFlow, by using the methods outlined by Kristine B. Hein [2]. The finite difference method is implemented as a simple Python function with a loop over the time steps. To avoid a (slow) Python loop for the spatial direction, Numpy array slicing is used instead. The integration loop is as follows:

```
for n in range(N_t - 1):
    # Vectorized method for the
    # spatial calculation skips one Python loop
    u_array[n + 1, 1:-1] = (C*(u_array[n, 2:]
                            - 2*u_array[n, 1:-1]
                            + u_array[n, 0:-2])
                            + u_array[n, 1:-1])

    t_array[n + 1] = t_array[n] + dt
```

Because Numpy slicing is *much* faster than a regular Python loop, the speedup over a more traditional method is significant. The same method using two loops would look as follows:

```
for n in range(N_t - 1):
    for i in range(1, N_x - 1):
        # only loop through the internal
        # points since the edges are always 0.
        # Outermost edge is at index N_x - 1
        u_array[n + 1, i] = (C*(u_array[n, i + 1]
                             - 2*u_array[n, i]
                             + u_array[n, i - 1])
                             + u_array[n, i])

    t_array[n + 1] = t_array[n] + dt
```

IV. RESULTS AND DISCUSSION

A. Finite difference solution of the diffusion equation

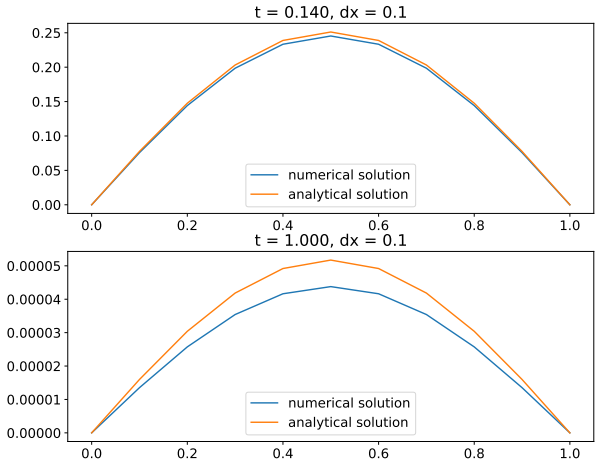

FIG. 1: Comparison between the analytical solution and the finite difference scheme for $dx = 0.1$ at two different time points. See the "figures" folder in the GitHub respository for the full resolution plot.

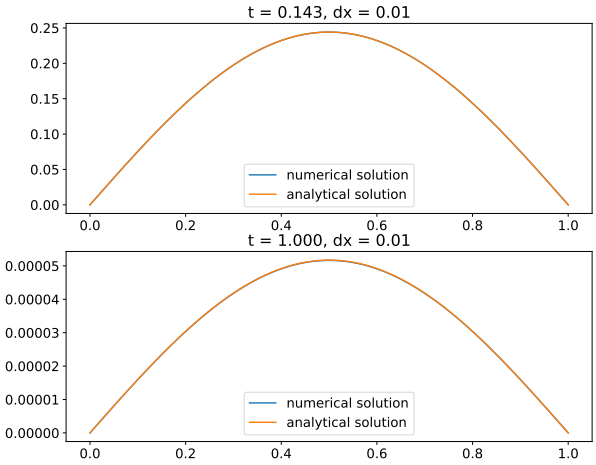

FIG. 2: Comparison between the analytical solution and the finite difference scheme for $dx = 0.01$ at two different time points. See the "figures" folder in the GitHub respository for the full resolution plot.

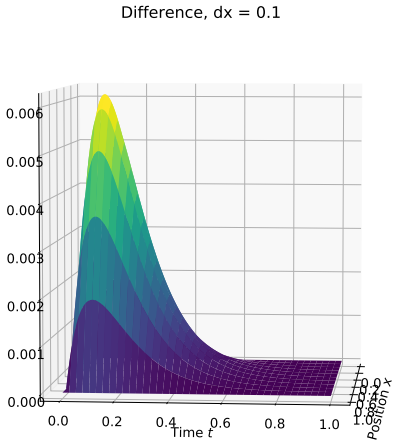

FIG. 3: Absolute difference between the analytical solution and the finite difference solution, showing how the absolute error drops significantly as the solution gets smaller.

FIG. 1 and FIG. 2 compares the solutions from the finite difference scheme with the analytical solution at two points in time, one point where the solution is still somewhat curved, and one where it is very close to the stationary state (note the short y-axis on the lower subplot in each figure). The comparison is done for two sets of spatial resolutions, $\Delta x = 1/10$ and $\Delta x = 1/100$. Δt is chosen from the stability criterion in each case. The error in FIG 1 is clearly visible, especially at $t = 1$, where the relative error is clearly quite large.

The mean squared errors and R2 scores are:

dx	MSE (t = 0.14)	MSE (t = 1)	R2 (t = 0.14)	R2 (t = 1)
0.1	1.5265e-5	2.8687e-11	0.998	0.880
0.01	1.5879e-9	3.4869e-15	0.9999997	0.9999866

TABLE I: The error results from the finite difference approximation

In TABLE I it's clear that the MSE drops as the time increases, but the R2 score decreases. This makes sense, since the solution itself decreases the MSE can still decrease while the relative error increases, which is exactly what the R2 score shows. This can also be seen in FIG. 1.

FIG. 3 also shows that the absolute error is largest around the time of 0.1. The absolute error obviously starts out as zero, as both the analytical solution and the FD solution are exactly equal to the initial condition at this point. The error then grows for a while before it again becomes very small when the solution itself also becomes very small.

The complete evolution of the solution over time can be seen in the following figures:

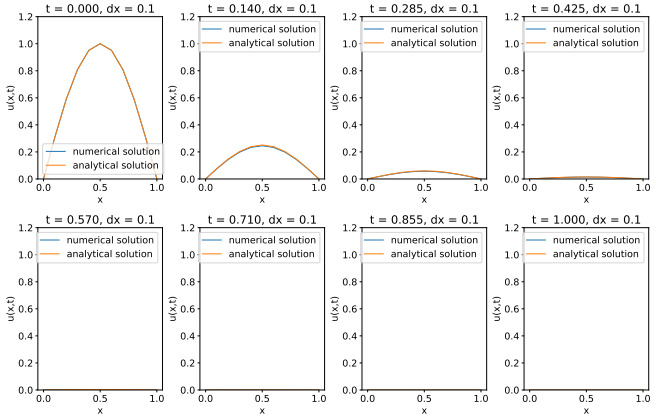

FIG. 4: Time evolution of the finite difference and analytical solution from $t = 0$ to $t = 1$ with $dx = 0.1$. Full resolution can be found in the github repository. Filename: "finite_diff_0010.pdf".

Note that the solution appears to decrease fairly rapidly, then taper off towards the stable state more slowly. At $t = 0.14$ the maximum point on the arch is already less than half of what it started as, but there is still a visible arch even at $t = 0.570$ (this might be hard to see in the report, but it is quite clear from the PDF file included in the GitHub repository ("fi-

nite_diff_0010.pdf"). For $t \geq 0.710$ the solution (both analytic and numerical) looks entirely flat.

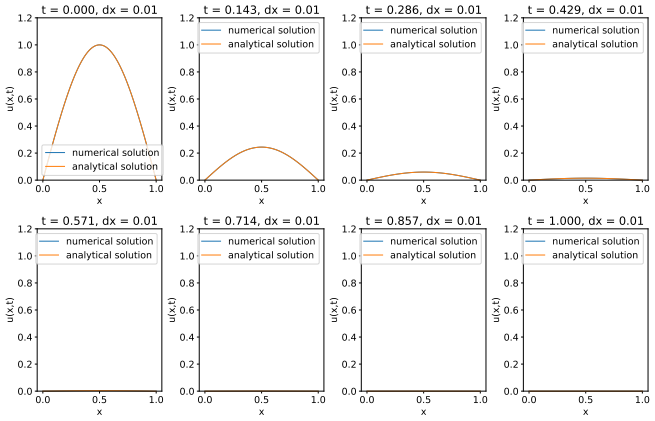

FIG. 5: Time evolution of the finite difference and analytical solution from $t = 0$ to $t = 1$ with $dx = 0.01$. Full resolution can be found in the github repository. Filename: "finite_diff_0001.pdf".

B. Neural network solution of the diffusion equation

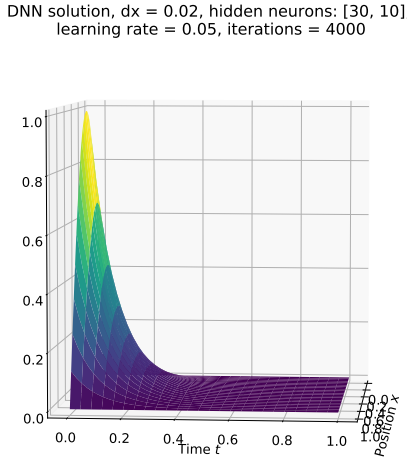

FIG. 6: Neural network solution of the diffusion equation with $dx = dt = 0.02$, a learning rate of 0.05 and 4000 iterations. The 3D figure can be studied by running "nn_solver.py".

FIG. 6 shows the solution from the neural network method. The parameters for the learning rate and number of iterations was chosen by testing various values and seeing where the results were best, more on this later.

The solution appears just from this plot to have the correct shape, with the values being at 0 for all times at the edges, starting high in the middle and first rapidly decreasing before slowly trending towards the stable state, similar as in FIG. 4 and FIG. 5. Comparing with the analytical solution as seen in FIG. 7 it's clear that the solution at least *looks* fairly correct.

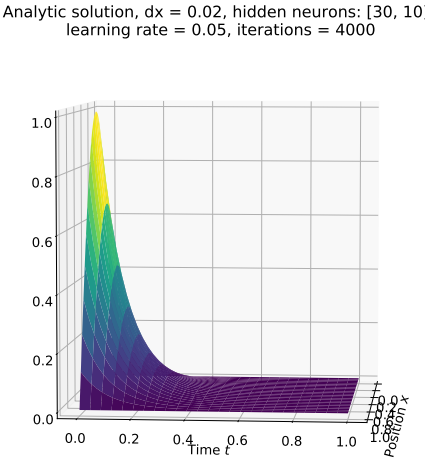

FIG. 7: Analytic solution of the diffusion equation with $dx = dt = 0.02$

it is not possible to see any significant difference between the solutions with the naked eye. FIG. 8 shows the absolute difference between the two solutions plotted in the same way.

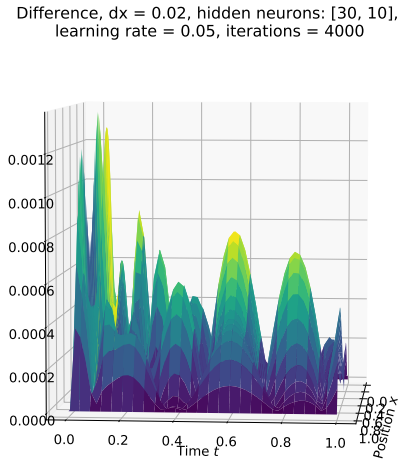

FIG. 8: Difference between the NN solution and the analytical solution with $dx = dt = 0.02$, a learning rate of 0.05 and 4000 iterations.

FIG. 8 shows that the absolute error is much less predictable for the neural network solution, compared to the finite difference solution (FIG. 3). Instead of becoming

very small as the solutions become small, the absolute error appears to be almost entirely random, other than at the edges where the trial solution forces the solution to be exact.

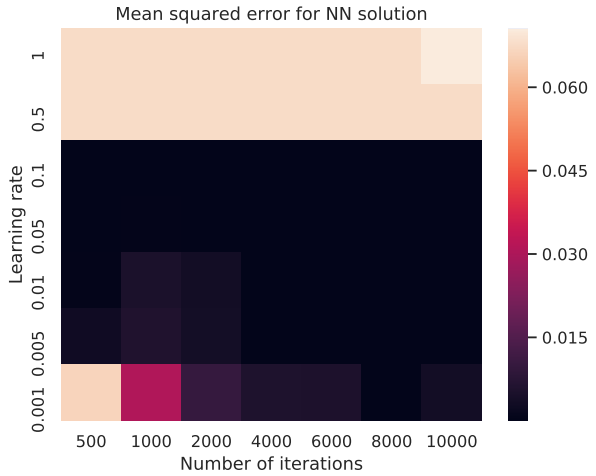

FIG. 9: Mean squared error, for a selection of learning rates at different numbers of iterations. $dx = dt = 0.02$.

FIG. 9 shows the mean squared error (MSE) of the neural network solution for a selection of different learning rates and number of iterations. In general the solution will become more accurate as the number of iterations increases, and in this case it's clear that for all of the learning rates between 0.005 and 0.1 the solution has already become somewhat accurate within 500 iterations.

For learning rates of 0.5 and 1 the MSE does not appear to drop at all, at least not within 10000 iterations. This could possibly be because the learning rate is simply too large for the gradient descent method of the neural network to converge to any minimum.

For the learning rate of 0.001 the solution slowly gets better and by $\sim$ 8000 iterations it is starting to be comparable to the rest of the converged solutions.

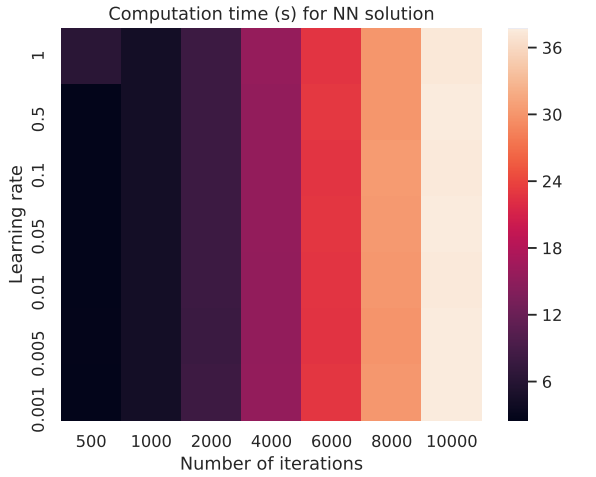

FIG. 10: Time spent (in seconds) computing, for a selection of learning rates at different numbers of iterations

FIG. 10 shows the time the neural network needed to train in order to reach a certain number of iterations. The computation time does not depend at all on the learning rate, but is directly influenced by the number of iterations required. However an accurate solution can be reached with fewer iterations if the learning rate is large. This means that choosing the optimal learning rate for reaching a good solution is very important for efficient computation, because the number of iterations can then be reduced.

C. Comparison of methods

In general, the finite difference method appeared to produce more accurate solutions in a shorter amount of time, however both methods produced good results in reasonable amounts of time. With $dx = 0.02$ and $dt = 0.02$, 4000 iterations, a learning rate of 0.05 and two hidden layers with 30 and 10 neurons for the first and second hidden layer respectively, the neural network took 23.6 seconds to compute the solution on my computer. The mean squared error after this time was 2.25863595e-7.

In comparison, the finite difference method, with $dx = 0.02$ and dt determined by the stability criterion, took 0.023 seconds and the mean squared error was 5.27556864e-9.

The full results:

Method	Time spent	MSE	R2 score
Neural network	23.6 seconds	2.25863595e-07	0.9999909
Finite difference	0.023 seconds	5.27556864e-09	0.9999998

TABLE II: Comparison of performance by the two different methods

From this it seems like the neural network method is inferior to the finite difference method when it comes to efficiency, however for cases with more spatial or time coordinates, and different sets of parameters for the neural network, there may be other results. This is something that should be looked further into in the future.

An advantage of the neural network is that there is no stability criterion between dx and dt, and it is also possible to compute an accurate solution for arbitrarily small values of both. This is in contrast to the finite difference method, where the solution may become unstable or inaccurate if these values are chosen without being careful.

An additional advantage of this is that the computation mesh does not have to be uniform for the NN solution. For example it is possible to have a more detailed grid in regions where the solution varies more rapidly, and a smaller grid in regions where the solution is almost constant. The finite difference method use here requires a uniform grid, which could lead to some regions having a more detailed mesh than actually required, increasing the computation time.

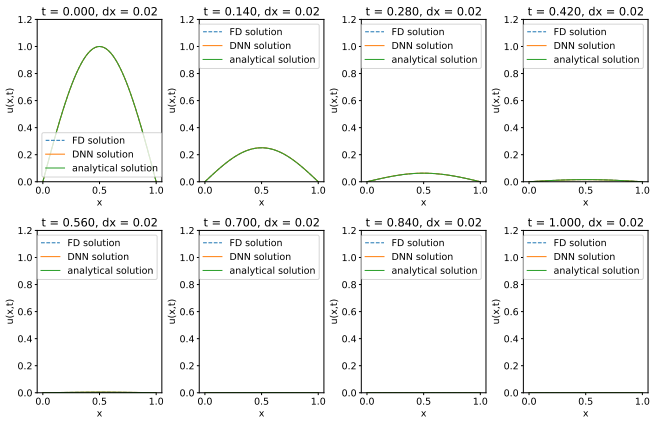

FIG. 11: All of the solutions plotted together for 8 different time points. It is not possible to see any significant difference between any of the solutions using only the naked eye. Full resolution can be found in the github repository. Filename: "multiple_dnn.pdf".

D. Eigenvectors and eigenvalues

The matrix A used for the computation was generated randomly by Numpy and can be found in appendix A (eq. 27).

The two different methods used (Numpy and neural network (NN)) gave the following results for the largest eigenvector:

$$v_{\max}^{\text{numpy}} = \begin{bmatrix} 0.50459793 \\ 0.41564063 \\ 0.34250246 \\ 0.4009538 \\ 0.39118732 \\ 0.37619727 \end{bmatrix} \quad v_{\max}^{\text{NN}} = \begin{bmatrix} 0.50432206 \\ 0.41608217 \\ 0.34274453 \\ 0.40074023 \\ 0.38959418 \\ 0.37773629 \end{bmatrix}$$

and the two corresponding eigenvalues were calculated as:

$$w_{\max}^{\text{numpy}} = 3.5309384 \quad w_{\max}^{\text{NN}} = 3.5309213$$

For the smallest eigenvector and eigenvalue the results were:

$$v_{\min}^{\text{numpy}} = \begin{bmatrix} 0.40662659 \\ -0.08778346 \\ 0.07917342 \\ -0.84657843 \\ 0.31846945 \\ 0.05062231 \end{bmatrix} \quad v_{\min}^{\text{NN}} = \begin{bmatrix} 0.38506859 \\ -0.07301816 \\ 0.07067102 \\ -0.85299467 \\ 0.33277961 \\ 0.05526265 \end{bmatrix}$$

$$w_{\min}^{\text{numpy}} = -0.5309233 \quad w_{\min}^{\text{NN}} = -0.53042474$$

The solutions for the maximum eigenvector and eigenvalue are almost identical, but they are not exactly the same. Looking at FIG. 12 the neural network solution (colored lines) clearly approaches the Numpy solution (dashed lines) as t increases.

The results for the minimum eigenvector and eigenvalue are not as good as for the maximum. From looking at FIG. 13, the solution does not appear to have completed converging yet, even at $t = 8$.

With further calculation and perhaps increasing time the solution is calculated for, the solutions might approach eachother further.

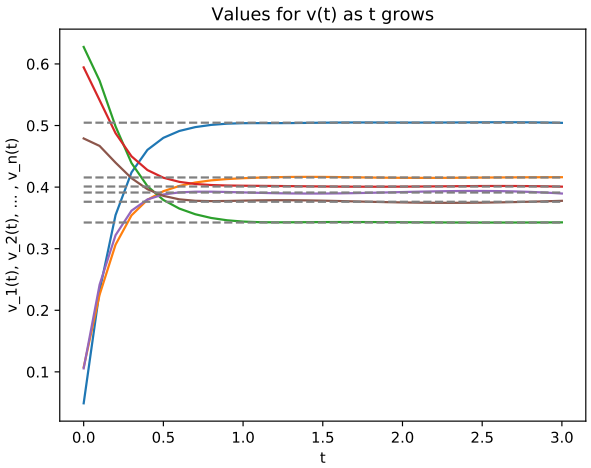

FIG. 12: The largest eigenvector. The colored lines represent the elements of the function $v(t)$ as described in equation 22. The dashed lines represent the values of the eigenvector as calculated by Numpy. As $t \to +\infty$ the colored lines should each match up with one of the dashed ones.

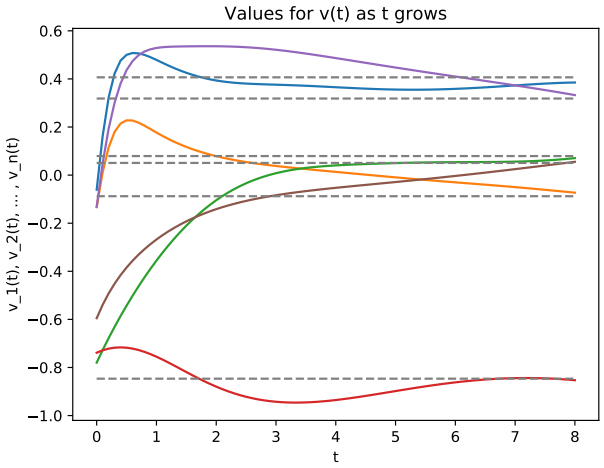

FIG. 13: The smallest eigenvector. The colored lines represent the elements of the function $v(t)$ as described in equation 22. The dashed lines represent the values of the eigenvector as calculated by Numpy. As $t \to +\infty$ the colored lines should each match up with one of the dashed ones.

The neural network used in this computation had three hidden layers, each with 10 neurons. Some experimentation was done, but this appeared to give the best result out of the configurations I tried. In each case, the neural network method was much slower than Numpy. For the maximum eigenvector, the computation was continued until the cost function reached a value smaller than 10^{-4}. For the minimum eigenvector this took a very long time, and I had to increase the threshold for the cost function to $5 \cdot 10^{-3}$. This is likely a large contributing factor to the results being somewhat incorrect for the minimum vectors. With more computation time, a better result

should be achievable.

Part of the reason for the slow computation probably lies in the construction of the cost function. I struggled to find a decent way to implement this, even after attempting to look up a few different ways to do it. With an improved cost function the time for the NN solution could probably be much improved, so that is definitely something to look into in the future.

The total computation for Numpy (for *all* of the eigenvectors and eigenvalues) was just 0.00018 seconds. For the neural network, finding just the largest eigenvector took almost 74 seconds on my computer.

V. CONCLUSION SUGGESTIONS FOR FUTURE WORKS

The general takeaway from this project is that neural networks indeed can be applied to the problem of solving both partial and ordinary differential equations. With sufficient computation the results match those from more traditional methods such as finite difference methods. The neural network approach tends to require more computation time than the more traditional methods, but the neural network approach is not without advantages. For example, the accuracy of the solution is not directly dependant on the resolution of the computation grid, and there is no stability criterion, meaning there is no chance of the solution becoming unstable if the resolution computation mesh is chosen carelessly, unlike for the finite difference method.

For the future I would suggest exploring the computation time for meshes with larger temporal and spatial resolution. The neural network might outperform the finite difference method for much larger meshes than the ones explored in this project.

There is also the case of the construction of the neural network which should be further studied. In this project two hidden layers were used, with 30 and 10 neurons in the first and second respectively. Other combinations are possible which might give better results.

Also of interest would be a comparison with other finite difference methods, or even finite element methods. There are many improvements that can be made to the finite difference method employed in this project, and these should also be compared to the neural network method before any conclusion can be made on which methods are superior (if any).

Applying a neural network to the problem of finding the largest and smallest eigenpairs of a symmetric, real and square matrix also showed some promise, although the computation time was significantly larger than optimal, and much greater than the more traditional linear algebra method used by Numpy. The results from the neural network solution match the results from Numpy to the second decimal place for the elements of the largest eigenvector, and the fourth decimal place for the largest eigenvalue. For the smallest eigenvector the computation had to be cut short because of time constraints, but all the elements of the eigenvector still match on the first decimal place, and the eigenvalues match to the second decimal place.

With an improved method for the cost function, I suspect the neural network method for the eigenpairs can be significantly improved, so that is definitely something which should be of interest for future studies.

However it is worth noting that this method can only provide the largest and smallest eigenpairs. The traditional linear algebra method however is capable of providing all of the eigenpairs for a given matrix.

Another thing which should be explored in the future is the relation between the size of the matrix and the computation time for both the methods. In particular it would be interesting to see whether the computation time for the neural network improves with a larger matrix, as compared to the traditional linear algebra methods.

REFERENCES

[1] M. M. Chiaramonte and M. Kiener. "Solving differential equations using neural networks." In: (2013). URL: http://cs229.stanford.edu/proj2013/ChiaramonteKiener-SolvingDifferentialEquationsUsingNeuralNetworks.pdf.

[2] Kristine Baluka Hein. *Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs.* https://compphysics.github.io/MachineLearning/doc/pub/odenn/html/odenn-bs.html. Accessed: 2019-12-18.

[3] Morten Hjorth-Jensen. *Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning.* https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html. Accessed: 2019-12-18.

[4] I E Lagaris, A Likas, and D I Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

[5] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix." In: (2004). URL: https://www.sciencedirect.com/science/article/pii/S0898122104901101.

VI. APPENDIX

A. A matrix

The matrix used for computing the eigenvectors and eigenvalues was the following:

$$A = \begin{pmatrix} 0.81341245 & 0.79244685 & 0.52089062 & 0.92459289 & 0.74526376 & 0.53488355 \\ 0.79244685 & 0.95929677 & 0.41409547 & 0.39186706 & 0.21724698 & 0.75778893 \\ 0.52089062 & 0.41409547 & 0.67064946 & 0.54232174 & 0.5435487 & 0.30469502 \\ 0.92459289 & 0.39186706 & 0.54232174 & 0.25945033 & 0.81929952 & 0.46796169 \\ 0.74526376 & 0.21724698 & 0.5435487 & 0.81929952 & 0.54039631 & 0.50196774 \\ 0.53488355 & 0.75778893 & 0.30469502 & 0.46796169 & 0.50196774 & 0.67812068 \end{pmatrix} \tag{27}$$