Solving differential equations using neural networks

Eina B. Jørgensen, Anna Lina P. Sjur and Jan-Adrian H. Kallmyr

December 14, 2019

Abstract

In this paper we use a neural network approach for solving the diffusion equation and compare this with a traditional finite difference approach using Forward Euler and Centered Difference methods. We also find the eigenpairs of a real symmetric matrix using a similar neural network approach. Our findings show that neural networks can be used to solve partial differential equations. Compared with a traditional approach, we find that the neural network can achieve a lower MSE of about 2-3 orders of magnitude, but is in general 3 orders of magnitude slower. We also looked at the effect of learning rate on MSE.

1 Introduction

In physics, as well as many other fields, differential equations (DE's) play a central role in analysis of a wide variety of problems. As such, it is important to use an efficient and accurate algorithm to produce reliable results. There exists many such algorithms of different orders, and each have their uses for different DE's. More complex DE's usually require higher order algorithms such as fourth order Runge-Kutta, while simple DE's can do with a first or second order algorithm such as Forward Euler (FE), or Centered Difference (CD), respectively. Higher order methods usually perform slower, and for complex coupled models, it is important to choose just the right algorithm for solving each sub-problem, as a slow method might significantly slow down model runs. It is therefore of interest if a general method for solving DE's exist which is efficient. A possible candidate are Neural Networks (NN's), which can approximate any function. In general, NN's for solving DE's could be useful if they offer a good accuracy/efficiency trade-off for many different types of equations, or if they outperform traditional methods. We will focus on the latter, and compare a traditional CD/FE-approach with a NN-approach. Research provided by Lagaris et al. (1998) and Chiaramonte and Kiener (2013) suggest that one can indeed solve PDEs and ODEs with Neural Networks with high precision and several advantages,but that it comes at a computational cost. We will see if we when solving the Diffusion Equation, come to the same conclusion. Furthermore, we will also look at an immediate application of the NN-approach in finding extrema eigenpairs (Yi et al., 2004) and compare this with the standard approach.

The Partial Differential Equation (PDE) we will solve (eq. 1), as well as all background theory and methods, can be found in Section 2. Our most important results are showcased in Section 3, and mainly consists of comparisons between different methods. In Section 4 we discuss the pros and cons of each method, as well as taking a deeper look at interesting results. Finally, in section 5 we summarise the article, presenting the most important takeaways, as well as possible future uses.

2 Theory and implementation

2.1 The general problem

The first problem to be solved in this project is a simple diffusion equation

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

Alternatively, this can be written as $u_{xx} = u_t$. Using $L = 1$, the initial condition at $t = 0$ are given by

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

Dirichlet boundary conditions are used, given by

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

This problem can for instance model the temperature of a rod that has been heated in the middle, and as time progresses the heat is transported through the rod and the temperature falls. We will first look at how an explicit scheme can be used, followed by a method using neural networks.

In the second part, we will look at how we can compute the eigenvalues and eigenvectors of a real, symmetric 6×6 matrix A. In particular, we will see how methods used to solve Eq. (1) can be applied to this problem.

To construct such a matrix, we let

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

where Q is a random, real matrix.

2.2 Exact solution of the diffusion equation

As we will be comparing the precision of the different ways of solving the partial differential equation, we need to calculate the exact solution in order to calculate the error. Through separation of variables, the equation can be expressed as

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

Differentiating this according to (1) and moving some terms, we get

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

As the to sides of this equation are not dependant on the same variables, they must both be equal to a constant. (We can choose this constant to be $-\lambda^2$). This gives the two equations.

$$X''(x) = -\lambda^2 X(x)$$
$$T'(t) = -\lambda^2 T(t)$$

For X can have three possible forms given by the characteristic equation. In order to satisfy the initial condition (2), $X(x)$ must be on the form

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

The initial condition then rules $C = 0, \lambda = \pi$. For $T(t)$ the solution is on the form

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

As we know $\lambda = \pi$ the solution is then:

$$u(x, t) = X(x)T(t) = Ae^{-\pi^2 t} B\sin(\pi x)$$

And finally from the initial condition, we know that $A \cdot B = 1$, and the exact solution is

$$u(x, t) = e^{-\pi^2 t}\sin(\pi x) \quad (4)$$

2.3 Discretization of the diffusion equation

For time discretization, as time is only used in first order derivative, we will use the explicit Forward Euler Scheme, which gives an error proportional to Δt. This is given as

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (5)$$

For the spatial discretization we use centred difference, which has an error proportional to Δx^2, given by

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} \tag{6}$$

On a discrete time and space grid, $u(x,t) = u(x_i,t_n)$, $t + \Delta t_n = t_{n+1}$ and so on. For simplicity we use the notation $u_i^n = u(x_i,t_n)$. The equation in it's discrete form is then

$$\begin{aligned} u_{xx} &= u_t \\ [u_{xx}]_i^n &= [u_t]_i^n \\ \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} &= \frac{u_i^{n+1} - u_i^n}{\Delta t} \end{aligned} \tag{7}$$

Solving this for u_i^{n+1} we can caluclate the next time step for each spatial point i:

$$u_i^{n+1} = \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + u_i^n \tag{8}$$

Which has a stability level for the grid resolution given by

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

2.4 Solving PDEs with Neural Networks

A different approach to solve a PDE is with the aid of a neural network. For an more in-depth description of how a neural network functions, see Jørgensen et al. (2019). Much of the same logic will here be applied to solve a partial differential equation.

In order to solve PDEs with a neural network, a trial function $\Psi(x,t)$ must be approximated, and our aim is to get this Ψ as close to the true function u as possible Lagaris et al. (1998). When aiming to solve Eq. (1), the corresponding equation, substituting with the trial function, is

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

The error (or residual) in the approximation is then

$$E = \frac{\partial^2 \Psi(x,t)}{\partial x^2} - \frac{\partial \Psi(x,t)}{\partial t} \tag{9}$$

The cost function to be minimized by the Neural Network is the sum of this E, evaluated at each point in the space and time grid.

For each iteration in the calculations, we will update our trial function based on the Neural Network's previous calculations. Therefore we must choose a fitting form for our trial function. This is based on the order of the PDE, and its initial condition. To satisfy the initial condition and Dirichlet conditions of Eq. (1), we choose:

$$\Psi(x,t) = (1-t)I(x) + x(1-x)tN(x,t,p), \tag{10}$$

where $I(x)$ is the initial condition, $N(x,t,p)$ is the output from the neural network, and p is the weights and biases.

Then, for each iteration in the network, the partial derivatives of Ψ is calculated according to the new state of the network $N(x,t,p)$, updating the cost. As the cost is minimized, the error term E gets closer to zero, and out trial function $\Psi(x,t)$ will approach the solution of the PDE.

How small we are able to get the cost is dependant of the maximum number of iterations we allow the Network to execute, the learning rate, and the structure of the Neural Network in terms of the number of hidden layers, and the number of nodes in each layer.

2.5 Computing eigenpairs with Neural Networks

Following the discussion by Yi et al. (2004), it is possible to compute the eigenvector v_{max} corresponding to the largest eigenvalue w_{max} of the $n \times n$ matrix A by solving the ordinary differential equation

$$\frac{dv(t)}{dt} = -v(t) + f(v(t)), \quad t \geq 0 \tag{11}$$

where $v = [v_1, v_2, \ldots, v_n]^T$ and $f(v)$ is given as

$$f(v) = \left[v^T v A + \left(1 - v^T A v\right) I\right] v \qquad (12)$$

Here, I is the $n \times n$ identity matrix.

As shown by Yi et al. (2004), when $t \to \infty$, any random non-zero initial v, as long as it's not orthogonal to v_{max}, will approach v_{max}. To compute the corresponding eigenvalue w_{max}, the following equation can be used.

$$w = \frac{v^T A v}{v^T v}$$

To compute the eigenvector v_{min} corresponding to the smallest eigenvalue w_{min} of A, one can simply substitute A with $-A$ in Eq. (11).

As described earlier, a trial function is needed to solve Eq. (11) with a Neural Network. Since $v \in \mathbb{R}^n$, we choose a trial function $\Psi(x,t)$ dependent both on position x and time t, so that for each time step, the approximated eigenvector is given as $[v(1,t), v(2,t), \ldots, v(n,t)]^T$. Eq. (11) can then be rewritten as

$$\frac{\partial \Psi(x,t)}{\partial t} = -\Psi(x,t) + f(\Psi(x,t)) \qquad (13)$$

with $t \geq 0$ and $x = 1, 2, \ldots, n$. We defined the trial function as

$$\Psi(x,t) = v_0 + tN(x,t,p)$$

where v_0 is the initial v, chosen at random. The error is then the difference between the two hand sides of Eq. (13). For this project, the mean squared error (MSE) was used as cost function.

2.6 Implementing the Neural Network

There are many ways of implementing the Neural Network method for solving differential equations. In this project, we have chosen to use *TensorFlow* for python3, as it is fast, stable and relatively simple to use. For implementation, see the git repository of janadr.

3 Results

3.1 Diffusion equation

In Figure 1 two different instants in the time evolution of the diffusion equation (eq. 1), as obtained from finite difference methods, are shown. For both, a comparison is made with the exact solution and a run with higher spatial resolution ($\Delta x = 1/100$) and lower spatial resolution ($\Delta x = 1/10$). In both cases, the graphs of the exact solution and the one with higher spatial resolution are difficult to distinguish.

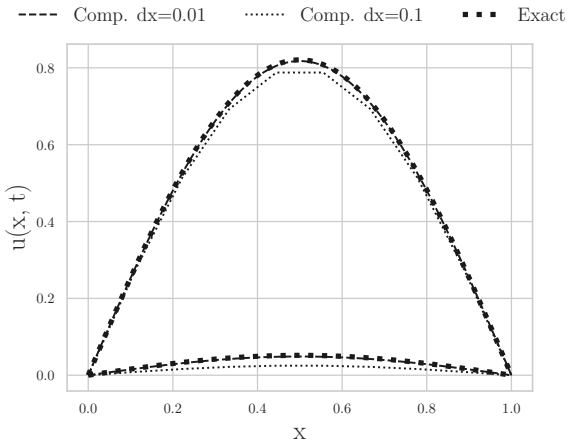

Figure 1: Time evolution of eq. 1 solved using finite difference methods: CD and FE for time's $t = 0.02$ s and $t = 0.3$ s. For each point in time a visual comparison is made with the exact solution.

Likewise, in Figure 2 we show a parallell to Figure 1 with solutions from a neural network. In this case, we also see agreement between the computed and exact solutions for both points in time.

Shown in Figure 3 are the MSE's for two instants in time for both finite difference methods and a neural network. For the finite difference methods we observe a decrease in MSE for increasing temporal solution, slowing down after around 500 time points for both cases. For

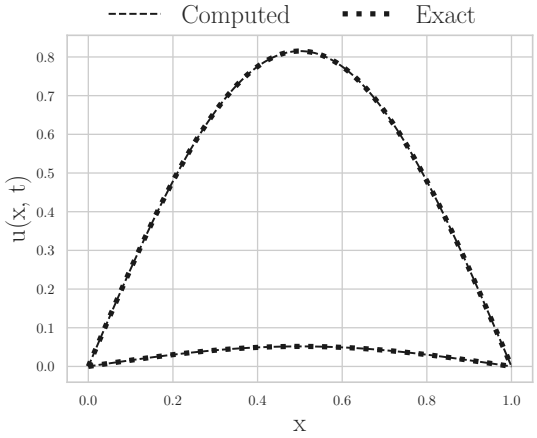

Figure 2: Time evolution of eq. 1 solved using a neural network. For both instants in time, $t = 0.02$ s and $t = 0.3$ s, a visual comparison is made with the exact solution, represented by dotted lines. The computations are done with $N_t = 30$ and $N_x = 100$.

$t = 0.02$ s (a.), the MSE is minimal at around 10^{-6} for 1000 time points, while for $t = 0.3$ s (c.) it only becomes around 10^{-5} for 1000 time points.

For the neural network, we observe a decrease in MSE for an increasing number of iterations. In the case of $t = 0.02$ s (b.), we see that the MSE decreases below 10^{-8} after around 6000 iterations, with a further, but slight, decrease at 10000 iterations. As for $t = 0.3$ s (d.), the MSE is below 10^{-8} after around 4000 iterations and is mostly unchanged afterwards.

Comparison between the finite difference methods and the neural network, we see that the neural network can obtain an MSE 2-3 orders of magnitude lower than the finite difference methods.

Accompanying Figure 3, we show the CPU times as a function of temporal resolution and number of iterations, respectively, for the different scenarios in Figure 4. We see that the general curve is very similar for all scenarios.

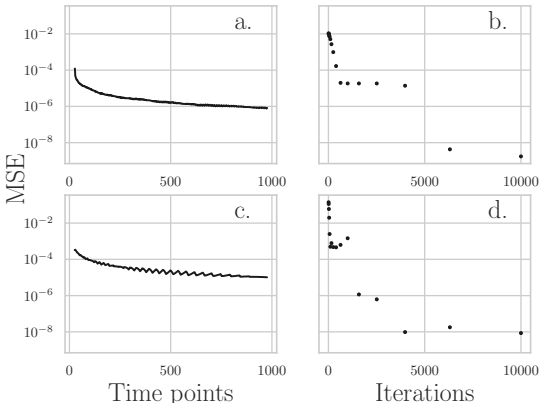

Figure 3: Shown on the left-hand side, top-to-bottom (a., c.), are the MSE's as a function of temporal resolution, obtained from using finite difference methods for $t = 0.02$ and $t = 0.3$, respectively. Likewise, on the right-hand side (b., d.) the MSE's yielded from the neural network are shown as a function of iterations with network parameters $N_t = 10$, $N_x = 100$, and $\gamma = 0.004$.

For the CD/FE method, we see that the CPU time becomes slightly longer for shorter t at 1000 time points. The neural network looks to have the same curve in both b. and d.

Comparing the two methods, we see that the neural network is around 3 orders of magnitude, if not more, slower than the finite difference methods.

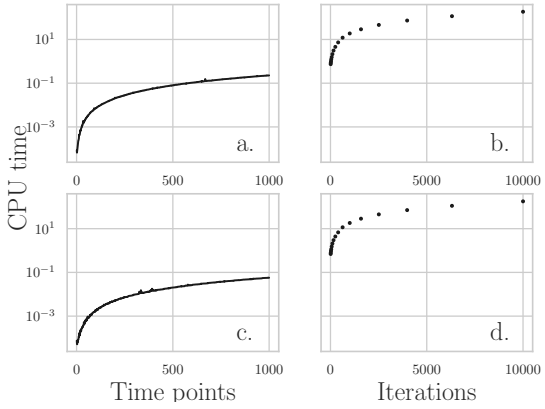

Figure 4: Shown on the left-hand side, top-to-bottom (a., c.), are the CPU time's as a function of temporal resolution, obtained from using finite difference methods for $t = 0.02$ and $t = 0.3$, respectively. Likewise, on the right-hand side (b., d.) the CPU time's yielded from the neural network are shown as a function of iterations with network parameters $N_t = 10$, $N_x = 100$, and $\gamma = 0.004$.

3.2 Eigenpairs

Looking at the eigenpairs of matrix A (A.1), the maximum eigenvalue computed with `numpy.linalg` is given under as w^{np}_{max}, with corresponding eigenvector v^{np}_{max}. The same eigenpair, computed with a neural network, is given as w^{nn}_{max} and v^{nn}_{max}. Note that the eigenvectors have been normalized, to allow for comparison. We see that the two eigenvalues only differ at the 5th decimal place, while the eigenvector elements first differ at the 3th decimal place. Comparing the CPU times of the two methods for one run, the neural network is slower by a factor of about 1000.

$$v^{np}_{max} = \begin{bmatrix} 0.4025950 \\ 0.4790814 \\ 0.3357358 \\ 0.3835011 \\ 0.3542414 \\ 0.4723555 \end{bmatrix} \qquad w^{np}_{max} = 2.83515$$

$$v^{nn}_{max} = \begin{bmatrix} 0.40165637 \\ 0.47992723 \\ 0.33576365 \\ 0.38384145 \\ 0.35298366 \\ 0.47294087 \end{bmatrix} \qquad w^{nn}_{max} = 2.83514$$

How v^{nn}_{max} evolves over time is shown in Figure 5. The network used for this computation had three hidden layers, all with ten neurons. The learning rate was 0.001, while the time step was 0.1. The precision, which was the MSE at which the training was stopped, was set to 10^{-4}, resulting in 10820 iterations.

Analogously to what was shown above for the maximum eigenvalue, the minimum eigenvalue computed with `numpy.linalg` is shown below as w^{np}_{min} together with the corresponding eigenvector v^{np}_{min}. Similarly, the same pair computed with neural network is denoted as w^{nn}_{min} and v^{nn}_{min}. Again, the eigenvalues differ first at the 5th decimal place, and the egenvectors differ at the 3th decimal place, not taking the sign into account.

$$v^{np}_{min} = \begin{bmatrix} 0.6928048 \\ -0.0074343 \\ -0.7048516 \\ -0.1487365 \\ 0.0122096 \\ 0.0296413 \end{bmatrix} \qquad w^{np}_{min} = -0.515594$$

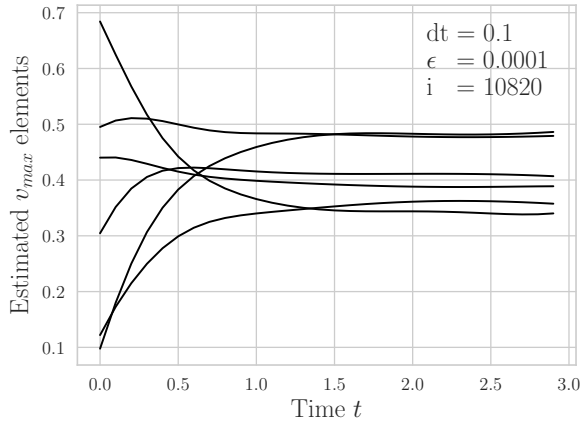

Figure 5: Figure showing how the value of the elements of the estimated eigenvector v_{max} evolves over time, when computed by a neural network. Each line represents one of the elements of the vector. Time step dt, precision ϵ and iterations i needed to achieve said precision is also shown.

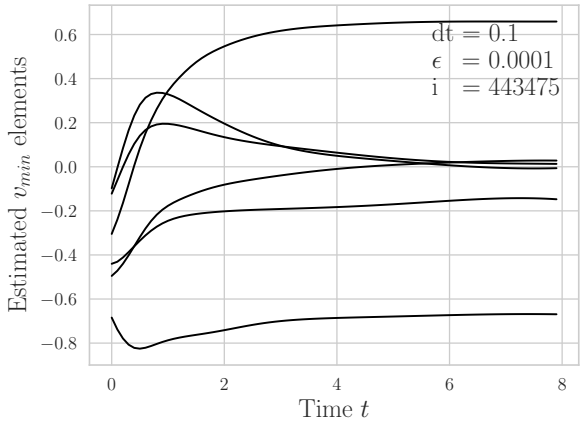

Figure 6: Figure showing how the value of the elements of the estimated eigenvector v_{min} evolves over time, when computed by a neural network. Each line represents one of the elements of the vector. Time step dt, precision ϵ and iterations i needed to achieve said precision is also shown.

$$v_{min}^{nn} = \begin{bmatrix} 0.69273272 \\ -0.00676815 \\ -0.70356177 \\ -0.15496268 \\ 0.0137166 \\ 0.02957751 \end{bmatrix} \qquad w_{min}^{nn} = -0.515579$$

Similarly to Figure 5, the graph in Figure 6 shows the evolution of the eigenvector v_{min}^{nn}. Note the different time values in the two figures. The same network, with the same configurations as described above, is used, except that A was replaced with $-A$ in the cost function. To reach a precision of 10^{-4}, 443475 iterations was needed.

4 Discussion

When it comes to calculating eigenpairs with the neural network, it seems to yield good results that agrees with the `numpy.linalg` method, when it comes to precision. In this report, the network was set to run until the cost function went below a threshold of 10^{-4}. When experimenting with the network, it was clear that there was a direct link between the threshold of the cost function, and the level of agreement between the network results and the numpy results. In other words, it seems to be possible to tune the network corresponding to the level of precision needed. However, the higher the precision, the more iterations is needed.

The maybe biggest drawback of the neural network is it's time use, with a CPU time about 1000 times that of `numpy.linalg` for a precision of 10^{-4} for the largest eigenvalue. The smallest eigenvalue demanded even more iterations to reach that precision. This leads us to another feature of the neural network. As can be seen in Figure 5 and 6, the time t at which the estimated eigenvectors stabilises varies from matrix to matrix, and the number

of iterations needed varies as well. So although it is possible to compute eigenpairs with a high level of precision, the computation can possibly become very time consuming.

Lastly, this methods only gives us the largest and the smallest eigenvalues, together with the corresponding eigenvectors. In other words, this method can not be used to find all the eigenpairs of a real symmetric matrix.

5 Conclusion

References

MM Chiaramonte and M Kiener. Solving differential equations using neural networks. *Machine Learning Project*, page 1, 2013.

Eina B. Jørgensen, Anna Lina P. Sjur, and Jan-Adrian H. Kallmyr. Using Neural Networks and Logistic Regression for Classification and Regression Problems. 2019.

Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

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

$$A = \begin{bmatrix} 0.37454012 & 0.50439896 & 0.78221829 & 0.51530175 & 0.30604431 & 0.38176969 \\ 0.50439896 & 0.86617615 & 0.40672706 & 0.49965086 & 0.40288023 & 0.57021699 \\ 0.78221829 & 0.40672706 & 0.18182497 & 0.3976287 & 0.25195801 & 0.29490401 \\ 0.51530175 & 0.49965086 & 0.3976287 & 0.13949386 & 0.40318954 & 0.65762369 \\ 0.30604431 & 0.40288023 & 0.25195801 & 0.40318954 & 0.59241457 & 0.50604122 \\ 0.38176969 & 0.57021699 & 0.29490401 & 0.65762369 & 0.50604122 & 0.80839735 \end{bmatrix} \tag{A.1}$$