Solving differential equations using neural networks

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

December 13, 2019

Abstract

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. 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

$$u(x,t) = X(x)T(t) \tag{3}$$

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.

$$\begin{aligned} X''(x) &= -\lambda^2 X(x) \\ T'(t) &= -\lambda^2 T(t) \end{aligned}$$

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

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

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 (SOURCE). This is given as

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

For the spatial discretization we use centred difference, which has an error proportional to Δx^2 (SOURCE), 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:

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

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 PDAs 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

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

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:

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

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

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

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

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

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

$$v_{max}^{np} = \begin{bmatrix} 0.4025950 \\ 0.4790814 \\ 0.3357358 \\ 0.3835011 \\ 0.3542414 \\ 0.4723555 \end{bmatrix} \qquad w_{max}^{np} = 2.83515$$

$$v_{max}^{nn} = \begin{bmatrix} 0.4023249 \\ 0.4791663 \\ 0.3364637 \\ 0.3822769 \\ 0.3541386 \\ 0.4730503 \end{bmatrix} \qquad w_{max}^{nn} = 2.83514$$

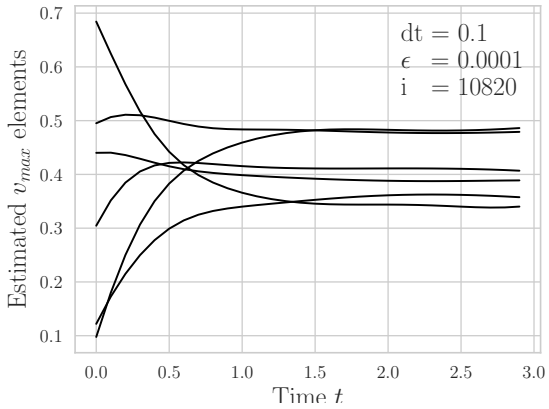

Figure 1: Figure showing how the value of the elements of the estimated eigenvector v_{max} evolves over time, when computed by a neural network. Each line represents one of the elements of the vector. Time step dt, precision ϵ and iterations i needed to achieve that precision is also shown.

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

$$v_{min}^{nn} = \begin{bmatrix} -0.69116113 \\ 0.00492352 \\ 0.70646873 \\ 0.14887754 \\ -0.01982836 \\ -0.02482513 \end{bmatrix} \qquad w_{min}^{nn} = -0.515528$$

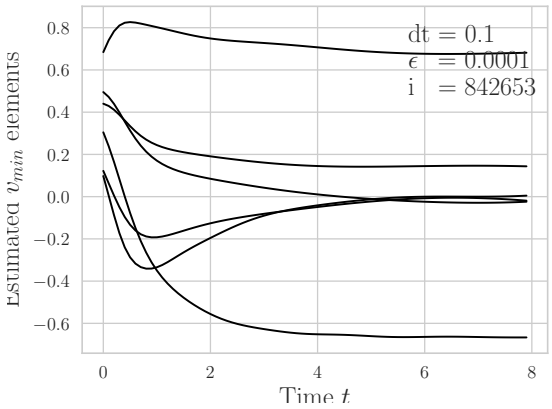

Figure 2: Figure showing how the value of the elements of the estimated eigenvector v_{min} evolves over time, when computed by a neural network. Each line represents one of the elements of the vector. Time step dt, precision ϵ and iterations i needed to achieve that precision is also shown.

4 Discussion

5 Conclusion

References

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.

A

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