Differential Equations, Eigenvalues and Neural Networks

Halvor Melkild*
Department of Physics, University of Oslo

Martin Tømterud†
Department of Physics and Technology, University of Bergen and
Centre for Materials Science and Nanotechnology, University of Oslo

Code available at git rep https://github.com/martintomterud/DA-ML

(Dated: June 2, 2023)

We compare and study applications of neural networks to two fundamental problems in mathematics; solving a partial differential equation and finding the eigenvalues of a symmetric matrix. We find that, for our implementation, the neural network does not yet reach the same accuracy as the common forward Euler iteration scheme applied to solving the partial differential equation, but that it does offer a great flexibility in choosing the discretisation parameters.

Keywords: machine learning, differential equations, neural networks, eigenvalues

I. INTRODUCTION

Many physical models amount to solving differential equations. This is the case for physicists looking at the world on the atomic scale, and meteorologists predicting the weather. Vast amounts of the world's computing resources are permanently occupied by solving Navier-Stokes to provide as accurate weather forecasts as possible. When solving differential equations, or other complicated equations, computers often spend a lot of time on diagonalizing matrices. This involves computing the eigenvectors and eigenvalues of the matrix one wants to diagonalize. Therefore scientists and engineers are hard at work developing algorithms that can handle these problems more efficiently. Neural networks have been suggested as a possible aid in pursuing more efficient algorithms. In this project, we explore how neural networks can be applied to solve a partial differential equation, namely the 1D diffusion equation (commonly referred to as the heat equation), and, we look at how a neural network can be used to solve a differential equation that can be used to identify the largest and smallest eigenvalues of a real, symmetric matrix. To say something about the efficiency and accuracy of these methods, we compare the solution of the heat equation to a solution obtained by a common forward Euler iteration scheme and the eigenvalues to library routines available in Numpy.

* Correspondence email address: halvor.melkild@fys.uio.no
† Correspondence email address: martin.tomterud@uib.no

II. THEORY

A. Partial Differential Equations

1. The Diffusion Equation

The diffusion equation is an example of a partial differential equation. In one dimension it reads

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

where D is the diffusion constant, or, in short-hand notation,

$$u_t = D\, u_{xx}. \tag{2}$$

The equation is used in many descriptive models in physics, for instance, in particles undergoing Brownian motion. It is also common to use diffusion to describe the temperature flow in materials, and in this particular example, will use it to model the temperature gradient of some rod of length L.

The one-dimensional diffusion equation can be solved analytically. The solution consists of a three-step process, which is generally used to solve PDEs:

1. Separate the PDE into two known ordinary differential equations (ODEs).
2. Solve the ODEs individually.
3. Compose the solution to the PDE of the two solutions of the ODEs.

We, therefore, start by assuming that we can write the solution, $u(x,t)$ as a product of two functions depending on only x and t separately:

$$u(x,t) = F(x)G(t). \tag{3}$$

We can therefore write the derivatives of u as

$$\frac{\partial u(x,t)}{\partial t} = F(x)\frac{dG(t)}{dt}, \tag{4}$$

$$\frac{\partial^2 u(x,t)}{\partial x^2} = G(t)\frac{d^2F(x)}{dx}. \tag{5}$$

Inserting these derivatives into the diffusion equation, we obtain

$$F(x)\frac{dG(t)}{dt} = DG(t)\frac{d^2F(x)}{dx^2}. \tag{6}$$

We now rearrange the equation above such that the left-hand side depends only on x and the right-hand side depends only on t. Since these two expressions must be equal, neither can depend on either x or t and they must therefore be constant:

$$\frac{1}{F(x)}\frac{d^2F(x)}{dx^2} = \frac{1}{D}\frac{1}{G(t)}\frac{dG(t)}{dt} = -k^2. \tag{7}$$

Anticipating the solution, we have chosen the constant to be a negative square number, $-k^2$. The x-dependent and t-dependent parts of our equation can now be equated to our constant $-k^2$ to obtain the two known ODEs:

$$\frac{d^2F(x)}{dx^2} = -k^2F(x), \tag{8}$$

$$\frac{dG(t)}{dt} = -k^2DG(t). \tag{9}$$

The solutions to this pair of ODEs can be found in any number of introductory textbooks in mathematics and physics, and are

$$F(x) = A\sin(kx) + B\cos(kx), \tag{10}$$

$$G(t) = Ce^{-k^2Dt}. \tag{11}$$

Redefining A and B to include C, we have our solution for u:

$$u(x,t) = e^{-k^2Dt}\left[A\sin(kx) + B\cos(kx)\right], \tag{12}$$

where A, B, and k are constants to be determined by the initial and boundary conditions.

B. Eigenvalues

A vector $\mathbf{v}$ of size n is said to be an eigenvector of some linear transformation A if it changes only by some scalar factor λ when the linear transformation is applied to it. in mathematical terms, this statement reads,

$$A\mathbf{v} = \lambda\mathbf{v}. \tag{13}$$

When this is the case, $\mathbf{v}$ is said to be an eigenvector of the matrix A, and λ is said to be the corresponding eigenvalue.

In 2002, Yi *et al.* proposed a neural network model for computing the eigenvalues and -vectors for the smallest and largest eigenvalue of a matrix [1]. The outline of the theory applied in their paper is repeated here.

For a column vector $\mathbf{x} = (x_1, ..., x_n)^T$ and an $n \times n$ real symmetric matrix A, the dynamics of the proposed neural network is given by the differential equation

$$\frac{d\mathbf{x}(t)}{dt} = -\mathbf{x}(t) + f\left(\mathbf{x}(t)\right), \quad t \geq 0, \tag{14}$$

with

$$f(\mathbf{x}) = [\mathbf{x}^T\mathbf{x}A + (1 - \mathbf{x}^TA\mathbf{x})I]\mathbf{x}. \tag{15}$$

By the theorems presented in the paper [1], it follows that an equilibrium point of Eq. (14) is an eigenvector of A, *i.e.* $\mathbf{x}$ is an eigenvector of A if it satisfies

$$\mathbf{x} = f(\mathbf{x}). \tag{16}$$

Once an eigenvector has been computed, its corresponding eigenvalue can be computed by the above relation. Inserting back in for f and writing out the product, Eq. (16) reads

$$\mathbf{x}^T\mathbf{x}A\mathbf{x} = \mathbf{x}^TA\mathbf{x}\mathbf{x} \tag{17}$$

Multiplying both sides with the eigenvector product $[\mathbf{x}^T\mathbf{x}]^{-1}$, yields

$$A\mathbf{x} = \frac{\mathbf{x}^TA\mathbf{x}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}. \tag{18}$$

Therefore, if $\mathbf{x}$ is an eigenvector of A, the corresponding eigenvalue is given by

$$\lambda_{\mathbf{x}} = \frac{\mathbf{x}^TA\mathbf{x}}{\mathbf{x}^T\mathbf{x}}. \tag{19}$$

III. METHODS

A. The Diffusion Equation

In order to solve the diffusion equation numerically we must establish the boundary and initial conditions, as well as a numerical scheme that can compute the solution.

1. *Boundary and Initial Conditions*

As stated above, we will assume the diffusion equation to be a model of the temperature gradient of a rod. We will assume that the temperature distribution at $t = 0$ is a trigonometric distribution on the form

$$u(x, t = 0) = \sin\left(\frac{\pi x}{L}\right), \quad 0 < x < L. \tag{20}$$

Furthermore, we will assume the fixed boundary conditions

$$u(x=0,t) = u(x=L,t) = 0 \qquad \forall\ t. \tag{21}$$

Furthermore, we will assume that the diffusion constant $D = 1$ and that the rod has length $L = 1$. From the general analytical solution of the diffusion equation, we can now obtain the specific solution for our case using the conditions above. Inserting the initial and boundary conditions in Eq. (12), we obtain:

$$u(x,0) = A\sin(kx) + B\cos(kx) = \sin(\pi x), \tag{22}$$

using $\sin(0) = 0$ and $\cos(0) = 1$:

$$u(0,t) = Be^{-k^2 t} = 0, \tag{23}$$

and using $\sin(\pi) = 0$ and $\cos(\pi) = -1$

$$u(1,t) = -Be^{-k^2 t} = 0. \tag{24}$$

This is enough to determine that

$$\begin{aligned} B &= 0, \\ A &= 1. \\ k &= \pi, \end{aligned}$$

The solution to the diffusion equation, in this particular case, is therefore

$$u(x,t) = \sin(\pi x)e^{-\pi^2 t}. \tag{25}$$

2. Forward Euler Method

Solving the PDE numerically implies writing down an approximation for the derivatives in the PDE. We will apply the centre-forward Euler method. For a small timestep Δt and spatial discretisation Δx, the approximate derivatives reads

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t}, \tag{26}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2}. \tag{27}$$

Inserting the approximations into the diffusion equation, $u_t = u_{xx}$, we obtain a method for propagating the initial conditions forward in time:

$$\begin{aligned} u(x,t+\Delta t) = &\left(1 - \frac{2\Delta t}{(\Delta x)^2}\right) u(x,t) \\ &+ \frac{\Delta t}{(\Delta x)^2}\left(u(x+\Delta x,t) + u(x-\Delta x,t)\right) \end{aligned}$$

The timestep Δt will depend on the spatial discretisation Δx through the stability criterion,

$$\Delta t \leq \frac{1}{2}(\Delta x)^2. \tag{28}$$

We will use the values in Table I when evaluating the diffusion equation.

Table I. Discretisation values in time and space used for evaluating the diffusion equation.

$1/\Delta x$	$1/\Delta t$
10	200
100	20000

3. Neural Network and the Diffusion Equation

We present a general methodology for solving an ODE with a neural network. It is straightforward to generalize the procedure to a PDE, it simply requires adding an additional variable. A simple ODE can be written in the following fashion

$$f(x, g(x), g'(x), g''(x), ..., g^{(n)}(x)) = 0, \tag{29}$$

where g is the function we seek. To introduce the neural network, we require a trial solution, g_t. We write this trial as

$$g_t(x) = h_1(x) + h_2(x, N(x,P)).$$

The function h_1 should make g_t satisfy the initial conditions, and $N(x,P)$ is the neural network with weights and biases described by P. The function h_2 is a function involving the neural network. Its role is to ensure that $h_2 = 0$ when g_t should satisfy the initial conditions. The neural network itself is optimised through back propagation with the chosen optimiser and cost function. For this we have used Keras[2]. Having constructed the network model, there are only two user-defined inputs needed to solve the problem; the trial solution g_t and the cost function $\mathcal{C}$.

For the trial solution, we choose $h_1(x,t) = \sin(\pi x)$, which is the initial condition of our network. $h_2(x,t)$ must furthermore satisfy $h_2(x, t=0) = 0$, which can be done by including a factor of t. It should also be equal to zero for $x = 0$ and $x = L$. To satisfy this, we choose to include the factor $\sin(\pi x)$. The full trial solution, therefore, reads

$$g_t(x,t) = \sin(\pi x)(1 + t\,N(x,t,P)).$$

To define the cost function, we rewrite the PDE as

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0, \tag{30}$$

which allows us to define our cost function $\mathcal{C}$ as

$$\mathcal{C}[u(x,t)] = \left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)^2. \tag{31}$$

B. Eigenvalues

We are looking for the eigenvalues of a real, symmetric square matrix of size $n = 6$. We generate a random matrix R and ensure it is symmetrized by using the matrix

A, given by

$$A = \frac{R + R^T}{2}. \tag{32}$$

Our trial input is a randomised state vector $\mathbf{x}$ that is the input of the neural network. The cost function $\mathcal{C}$ is in this case given by

$$\mathcal{C} = \mathbf{x} - f(\mathbf{x}), \tag{33}$$

where the function f is given by Eq. (15).

C. Error Metrics

We will use the mean square error (MSE) as the main metric for estimating the accuracy of our model. This will take a few forms depending on the model we are testing.

For the PDE we evaluate a two-dimensional parameter space and the MSE will therefore average over either the time domain or the spatial domain. Since the PDE generally is propagated forward in time, we choose to take the average over space. Thus, for a spatial grid with length N, the MSE reads,

$$\text{MSE}^{\text{PDE}}(t) = \frac{1}{N}\sum_{i=1}^{N}(u(x_i,t) - \tilde{u}(x_i,t))^2, \tag{34}$$

where u is the analytical solution of the diffusion equation given by Eq. (25), and $\tilde{u}$ is the solution found by either the explicit forward Euler scheme or the neural network. We will also present simply the square error, which is given by

$$\text{SE}^{\text{PDE}}(t) = (u(x_i,t) - \tilde{u}(x_i,t))^2, \tag{35}$$

For the eigenvalue problem, we use the defining equation of the eigenvalues and vectors given in Eq. (13). For an $N \times N$ matrix A, with an $N \times 1$ eigenvector $\mathbf{v}$ with correspondingly estimated eigenvalue $\tilde{\lambda}$ and estimated eigenvector $\tilde{\mathbf{v}}$, the MSE is given by

$$\text{MSE}^{\text{eig.}} = \frac{1}{N}\sum_{i=1}^{N}([A\mathbf{v}]_i - [\tilde{\lambda}\tilde{\mathbf{v}}]_i)^2, \tag{36}$$

where i is the i-th component of the vector.

IV. RESULTS AND DISCUSSION

A. Heat Equation

The solution of the heat equation using the forward Euler iteration scheme is shown in Figures 1 and 3 for the discretisation parameters shown in Table I. The plots are two-dimensional, with the colorbar indicating the value of the solution $u(x,t)$. The final time $T = 0.5$ (unitless units) was chosen as the result converged towards zero. The square error and MSE of the results are shown in Figures 2 and 4, respectively. We observe that the results converge well for both discretisations, but the finer discretisation gives an MSE of 3 orders of magnitudes smaller.

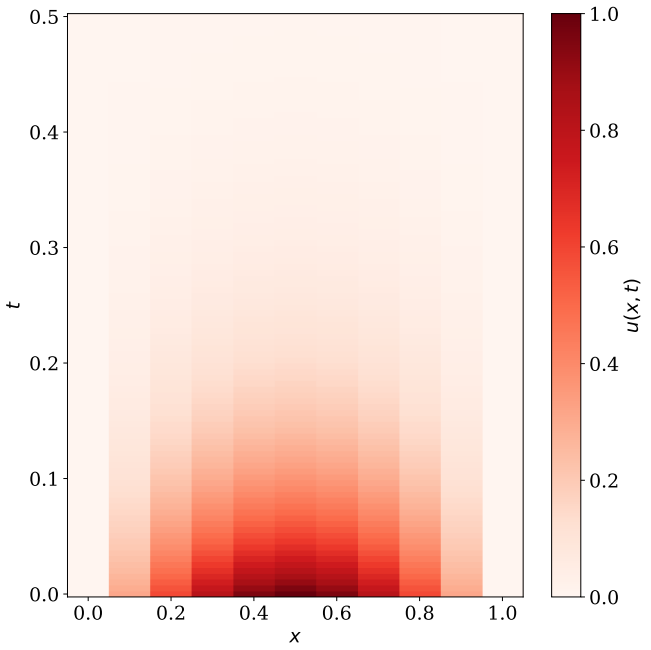

Figure 1. The diffusion equation $u(x,t)$ plotted for both its variables until the temperature has decreased to the boundary temperature. The spatial discretisation is $\Delta x = 1/10$.

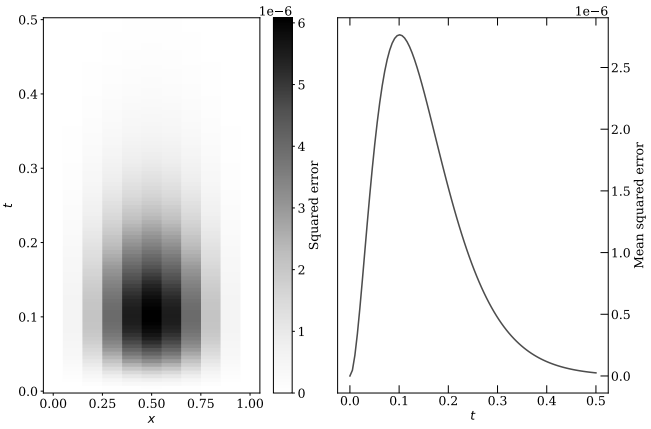

Figure 2. The square error (left) and mean square error at each timestep (right) of the diffusion equation, with $\Delta x = 1/10$.

The corresponding results found using the neural network are shown in Figures 5 and 7. In the former, the neural network was trained on only 10 random points in the domain, and in the latter, on 100 random points (per x, t dimension). The square errors of the two results are presented in Figures 6 and 8, respectively. Comparing

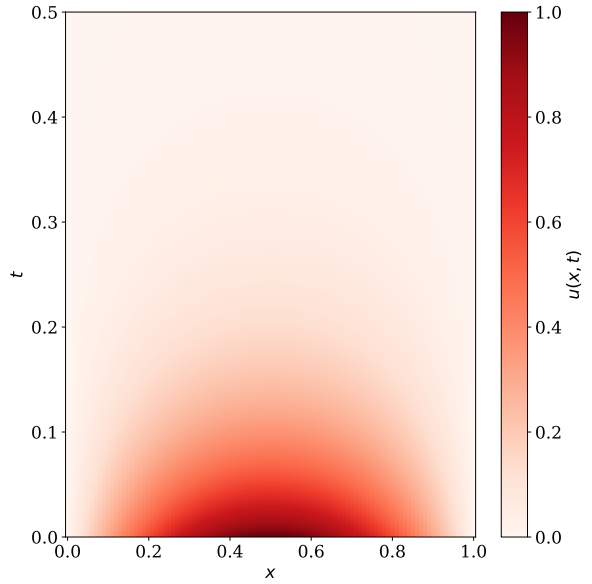

Figure 3. The diffusion equation $u(x,t)$ is plotted for both its variables until the temperature has decreased to the boundary temperature. The spatial discretisation is $\Delta x = 1/100$.

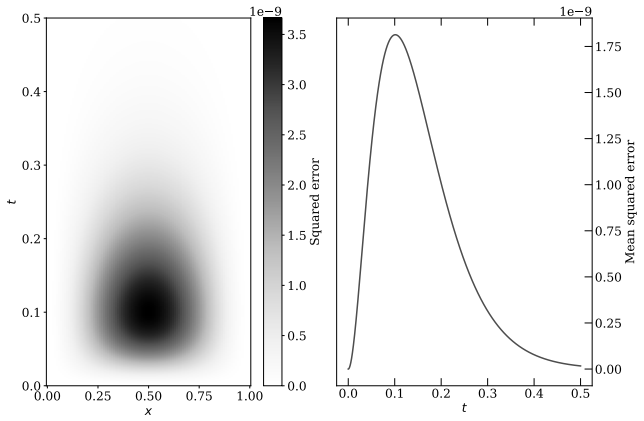

Figure 4. The square error (left) and mean square error at each timestep (right) of the diffusion equation, with $\Delta x = 1/100$.

the two, it is evident that the network trained on a larger number of points achieves better accuracy. Furthermore, we observe that for the large grid-trained network, there is only one region (in (x,t)-space) with large square errors (compared with the rest of the grid), whereas for the small grid-trained network, there are two regions with comparatively large square error. This is only true for the neural network and was not the case for the Euler iteration scheme. From the MSE, it is evident that we obtain better results with larger training grid sizes, but the smaller grid size we use to train the network also yields a well-converged solution with an MSE not too far from the one we obtain by using the larger grid. Neither of these MSEs compares well to the one we obtain with

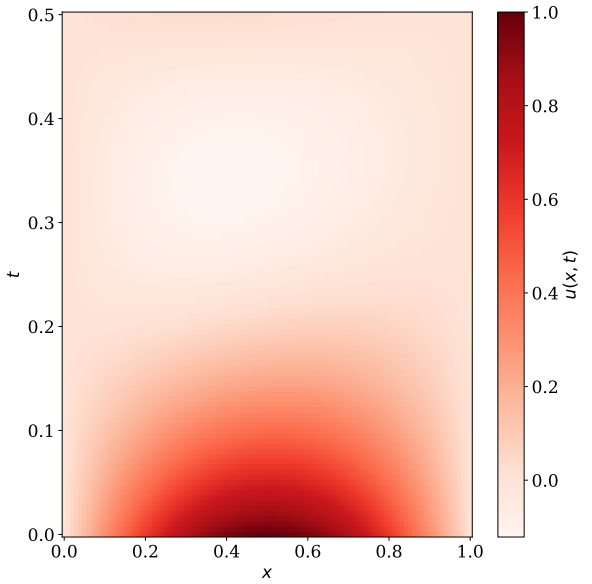

Figure 5. Predicted solution of the heat equation, $u(x,t)$ from a neural network trained on 10 random points in the domain.

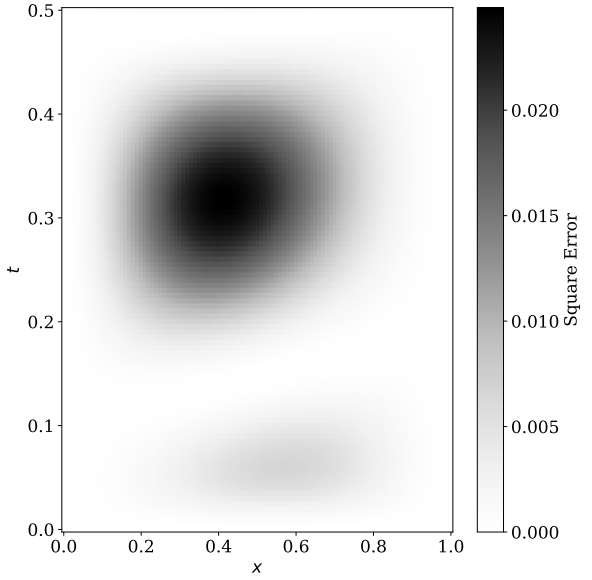

Figure 6. Square error between the neural network solution to the heat equation trained on 10 random points and the analytical solution.

the iterative method, which is several orders of magnitude smaller. The iterative method is also considerably faster. The main advantage of the neural network is that we can use the trained network to evaluate any point in the $[0,L] \times [0,T]$-domain. Both of the solutions are evaluated on 100×100 grids. If we wanted to do the same with the Euler finite difference scheme, we would have to do some sort of interpolation or redo the calculation at the correct discretisation. Furthermore, there is no need for the neural network grid to obey the same

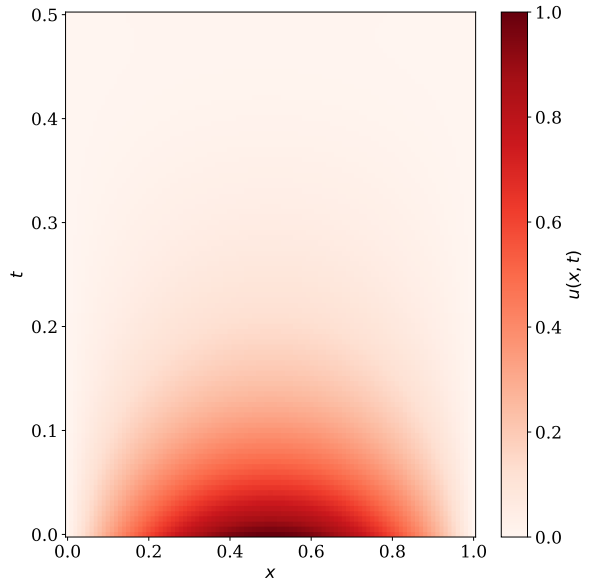

Figure 7. Predicted solution of the heat equation, $u(x,t)$ from a neural network trained on 100 random points.

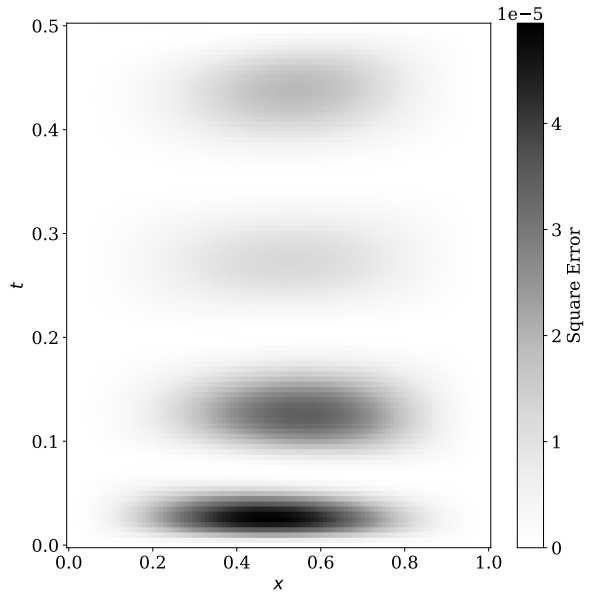

Figure 8. Square error between the neural network solution to the heat equation trained on 100 random points and the analytical solution.

stability criterion as the Euler method. This relation is quadratic, meaning that if we want 10-times more points in the time domain, we would need a factor of 100 more spatial points. This is a significant advantage for the neural network which can be trained on an arbitrary number of points in both space and time dimensions, independent of each other.

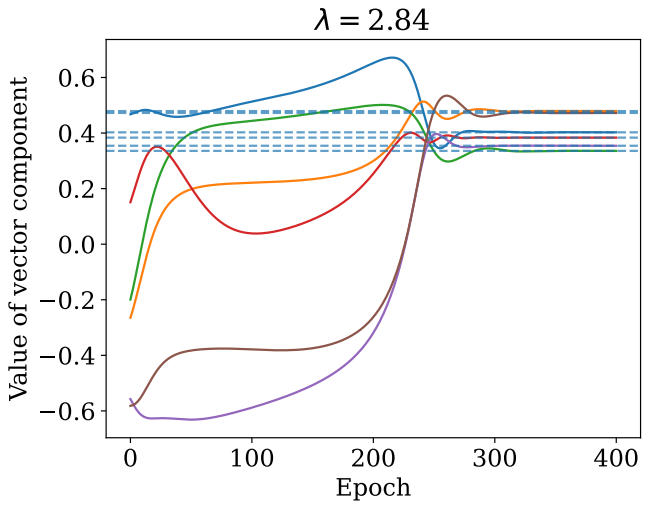

Figure 9. Components of the eigenvector corresponding to the largest eigenvalue of the matrix as a function of the number of epochs completed by the neural network. The dashed horizontal lines show the true values found by library routines.

B. Neural Network and Eigenvalues

We have implemented two solutions to this problem. One is a gradient-descent version following the algorithm outlined in Ref. 1, and the second is a neural network implementation to solve the ODE outlined in the same reference. Generally, we observe that both methods converge towards the desired eigenvector and eigenvalue. The neural network can occasionally ”skip” the largest eigenvalue and return one of the smaller eigenvalues and its corresponding eigenvector.

Figures 9 and 11 show the components of the eigenvectors of the largest and smallest eigenvalues, respectively, converging towards the solution found by Numpy’s eigenvector solver, which are indicated by dashed lines. We have also plotted the MSE as defined in Eq. (36) for both instances in Figures 10 and 12, and show that they converge to zero when the eigenvector components converge to the those found by library routines.

There is one further large challenge related to computing the eigen-parameters with neural networks, namely the convergence time. It is not straightforward to determine that the results from the neural network are steady-state solutions. For the results to be usable, it is therefore important to choose a number of epochs such that what is computed indeed is the steady-state solution of Eq. (16). Without prior knowledge of the system, picking the correct number of iterations is a problem, especially considering that we want efficient calculations. Picking a too-large number of iterations wastes computing power, while picking a number too small implies useless results. This fact, together with what is mentioned above, makes the use-case for neural networks for this application limited.

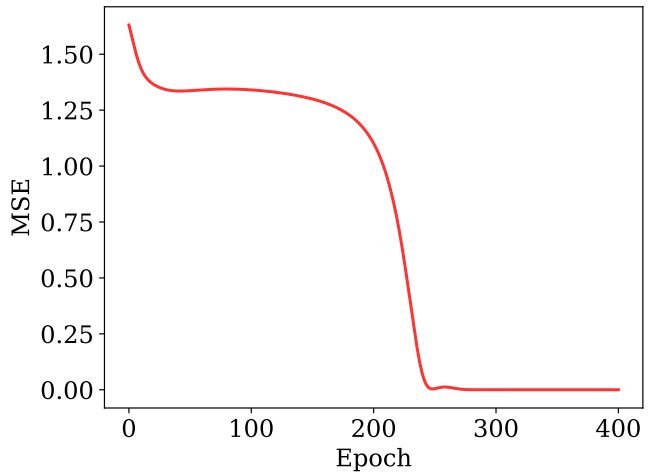

Figure 10. The mean square error defined in Eq. (36) for the largest eigenvalue.

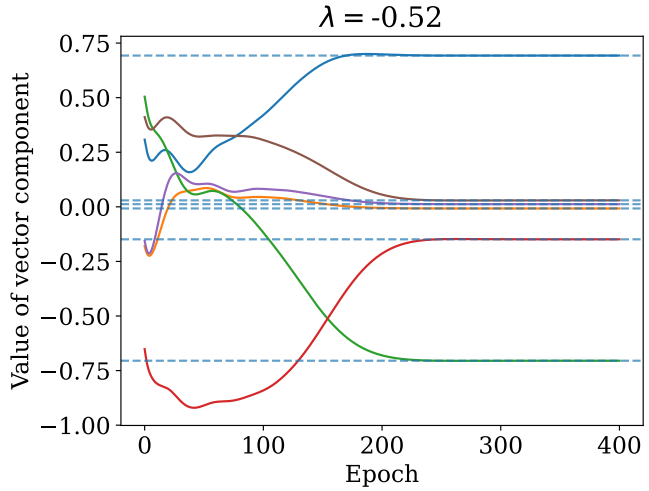

Figure 11. Components of the eigenvector corresponding to the smallest eigenvalue of the matrix as a function of the number of epochs completed by the neural network. The dashed horizontal lines show the true values found by library routines.

V. CONCLUSION AND OUTLOOK

We have performed an analysis on the application of neural networks to solving differential equations. First, we applied a neural network to solve a PDE, namely the 1D diffusion equation, and compared our results to those obtained using a standard algorithm for solving the problem, namely the forward Euler iteration scheme. Second, we applied a neural network to an ODE designed to compute the smallest and largest eigenvalues of a symmetric, real matrix. Both applications successfully reproduce the results of the techniques they are compared to. They are, however, slower in execution time and not as accurate as the iterative solution. The neural network solution of the differential equation offers a great advantage in not being bound by the stability criterion of the finite difference scheme. The eigenvalue solver finds the correct eigenvalues and -vector, and once it has converged, there is no error to speak of. It is, however, limited to only finding the largest and smallest eigenvalues; it is considerably slower than its library counterpart and limited in applications by choosing the correct number of iterations. Summarised, our findings indicate that the neural networks can reproduce the correct results, but our implementation cannot compete with established numerical approaches regarding computational time.

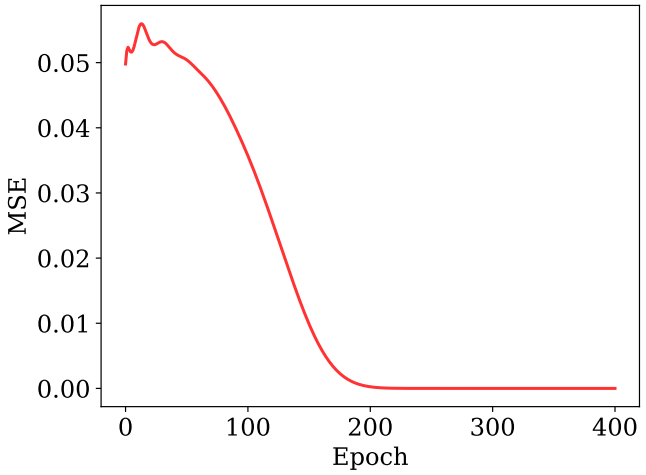

Figure 12. The mean square error defined in Eq. (36) for the smallest eigenvalue.

[1] Z. Yi, Y. Fu, and H. J. Tang, Computers & Mathematics with Applications **47**, 1155 (2004).
[2] F. Chollet *et al.*, "Keras," https://keras.io (2015).