The Hitchhiker's Guide to Solving Differential Equations and Eigenvalue Problems Using Feed Forward Neural Networks

René Ask, Kaspara Skovli Gåsvær & Maria Linea Horgen

(Dated: December 15, 2020)

Methods using neural networks to solve differential equations are presented. The method applied to the one-dimensional diffusion equation is studied and compared with the forward Euler scheme in time for reference. The neural network based method is found to be outperformed by the forward Euler scheme given the limited access to hardware acceleration and memory resources. We conclude that the method is not suited for the one the diffusion equation studied here, but that the method may be suitable for other partial differential equations and/or higher-dimensional problems where conventional methods are intractable and neural networks combined with hardware acceleration can thrive. The method is also applied to a differential equation whose equilibrium states yield eigenvectors of a symmetric matrix. We find that the method converges quickly for small size matrices. Although the solutions consistently converge to eigenvectors, our results disagrees with the claim that solutions cannot converge to eigenvectors orthogonal to the eigenspace the initial state reside in [8].

I. INTRODUCTION

Numerous problems in physics, biology and other branches of science can be modelled by partial differential equations. They are an essential part of our quest to explain the world around us, but the vast majority of interesting and realistically modelled natural processes do, however, not allow themselves to be solved analytically, prompting a development of reliable numerical methods. Well established methods such as Euler schemes, Runge-Kutta- or finite element methods can often provide good approximations, but are limited to only produce discrete output or solutions of limited differentiability.

Earlier work [6] describes a procedure in which a neural network can be used to solve ordinary and partial differential equations (ODEs and PDEs) which results in a differentiable solution in closed analytic form. We shall apply this procedure to the one-dimensional diffusion equation with Dirchlet boundary conditions. Comparison with the so-called explicit Euler scheme will be provided. Both methods will be analyzed against the analytical solution derived in V A.

In quantum mechanics (QM), any observable is represented by a Hermitian operator which in many cases will be an ordinary symmetric matrix. One of the fundamental postulates of QM is that any measurement outcome will necessarily be an eigenvalue of the operator corresponding to the observable. Thus to test the theoretical predictions of QM, eigenvalue spectra of operators are necessary. An ODE can be formulated such that the steady state solution necessarily converges to one of the eigenvalues of a defined symmetric matrix [8]. We will test this theory on a symmetric (6×6)-matrix and compare the results with the standard eigenvalue solver provided by Numpy [5].

All the necessary theory is developed, and presented in section II. In section III we provide a critical analysis of the methods implemented and the results obtained with focus on the validity of using neural networks as a mean to solve differential equations and eigenvalue problems.

The neural networks will be implemented using the widely popular Keras API of TensorFlow [2][4]. TensorFlow is an end-to-end platform designed for easy building of machine learning models while Keras is a deep-learning API built on top of TensorFlow. For codes and documentation see Github repository [1].

II. FORMALISM

A. Differential Equations

Differential equations (DEs) can generally be formulated as

$$G(x, t, u, \partial_t u, \ldots, \partial_t^m u, \nabla u, \ldots, \nabla^n u) = 0, \tag{1}$$

where G is some differential operator acting on a function $u : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^k$ and $\partial_t^n \equiv \partial^n/\partial t^n$ and $x \in \mathbb{R}^d$. Here $u \equiv u(x,t)$ and d is the number of spatial coordinates. To completely specify a solution of (1), we must provide n boundary conditions and m initial conditions.

1. *The Diffusion Equation*

In the one-dimensional case we'll solve the diffusion equation

$$G = \partial_x^2 u - \partial_t u = 0, \qquad t > 0, \qquad x \in [0,1], \tag{2}$$

with the following constraints:

$$\begin{aligned} u(0,t) &= 0, \\ u(1,t) &= 0, \\ u(x,0) &= \sin(\pi x). \end{aligned} \tag{3}$$

The closed-form solution, derived in section V A, of this equation is given by

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

2. *The network DE*

This section relies heavily on the mathematical framework derived in 'Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix' [8], meaning that we restrict ourselves to just render the most important mathematical results and omitting their proofs. The DE proposed to compute eigenvalues is

$$G = \frac{\mathrm{d}x(t)}{\mathrm{d}t} + x(t) - g(x(t)) = 0 \quad \text{for} \quad t \geq 0, \tag{5}$$

with $g(x)$ being

$$g(x) = \left(x^T x A + (1 - x^T A x)I\right) x, \tag{6}$$

where $A \in \mathbb{R}^{n\times n}$ and $x : \mathbb{R} \to \mathbb{R}^n$ and $I \in \mathbb{R}^{n\times n}$ is the identity matrix. It can be shown that any equilibrium state of eq. (5) is an eigenvector of A, as long as it is not the zero vector. Denote $v \in \mathbb{R}^n$ as said equilibrium state, i.e $v = \lim_{t\to\infty} x(t)$. It satisfies

$$g(v) - v = 0, \tag{7}$$

from which we can deduce that

$$Av = \frac{v^T A v}{v^T v} v, \tag{8}$$

demonstrating that $\lambda = v^T Av/v^T v$ is an eigenvalue of A corresponding to the eigenvector v. The solution $x(t)$ starting from an arbitrary nonzero $x(0) \in \mathbb{R}^n$ will converge to an eigenvector of A as $t \to \infty$, which is rigorously proven in [8].

Convergence Properties: Some further details regarding the convergence properties of eq. (5) are proved in [8]. Which eigenvector the solution converges to is dependent on $x(0)$, which is connected to the fact that symmetric matrices form mutually orthogonal eigenspaces, such that any two eigenvectors from two distinct eigenspaces are orthogonal to each other, as stated by the *spectral theorem for symmetric matrices.* In general, any $x(0)$ can be written as a linear combination of the eigenvectors

$$x(0) = \sum_j c_j(0) v_j, \tag{9}$$

which is possible since the eigenvectors form a complete set. The general solution can be shown to be of the form

$$x(t) = \sum_j c_j(t) v_j, \tag{10}$$

for some differentiable function $c_j(t) : \mathbb{R} \to \mathbb{R}$. This implies that $x(t)$ cannot escape the eigenspace(s) $x(0)$ lies in.

The following two cases are of special importance

- **Case 1**: If $x(0)$ is not orthogonal to the eigenspace of the largest eigenvalue, the solution of eq. (5) starting from $x(0)$ converges to an eigenvector corresponding to the *largest* eigenvalue of A.
- **Case 2**: If $x(0)$ is not orthogonal to the eigenspace of the smallest eigenvalue, and replacing $A \to -A$ in eq. (5), the solution starting from $x(0)$ converges to an eigenvector corresponding to the *smallest* eigenvalue of A.

B. Explicit Scheme: Forward Euler

We'll operate with the following notational conventions: $u(x_j, t_m) \equiv u_j^m$ where $x_j = j\Delta x$ and $t_m = m\Delta t$.

We assume that we have $n+1$ coordinate gridpoints such that $\Delta x = 1/(n+1)$ and M distinct timesteps such that $\Delta t = T/M$, where T represents the total time of simulation. The initial- and boundary conditions in this discretized format is then

$$u_0^m = u_{n+1}^m = 0, \qquad u_j^0 = f(x_j) \equiv f_j. \tag{11}$$

An explicit scheme is a prescription from which the state of a system at a time t_{m+1} can be computed explicitly from a complete description of the system at a time t_m. To obtain such a scheme for the diffusion equation we Taylor expand forwards in time about the point (x,t):

$$u_j^{m+1} = u_j^m + \Delta t \partial_t u_j^m + \mathcal{O}(\Delta t^2), \tag{12}$$

which naturally leads to

$$\partial_t u = \frac{u_j^{m+1} - u_j^m}{\Delta t} + \mathcal{O}(\Delta t). \tag{13}$$

Similarly, we'll Taylor expand about (x,t) forwards and backwards in coordinate space. Expanding forwards yields:

$$\begin{aligned} u_{j\pm1}^m = u_j^m \pm \Delta x \partial_x u_j^m + \frac{\Delta x^2}{2!}\partial_x^2 u_j^m \\ \pm \frac{\Delta x^3}{3!}\partial_x^3 u + \mathcal{O}(\Delta x^4). \end{aligned} \tag{14}$$

Adding these two equations ($\pm$) gives us

$$u_{j+1}^m + u_{j-1}^m = 2u_j^m + \partial_x^2 u_j^m + \mathcal{O}(\Delta x^4) \tag{15}$$

Solving this with respect to the second derivative leads to

$$\partial_x^2 u_j^m = \frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2} + \mathcal{O}(\Delta x^2). \tag{16}$$

Inserting the expression for the time derivative from eq. (13) and our newly derived second derivative of x into the diffusion equation (2) gives us

$$\frac{u_j^{m+1} - u_j^m}{\Delta t} = \frac{u_{j+1}^m - 2u_j^m + u_{j-1}^m}{\Delta x^2} + \mathcal{O}(\Delta x^2, \Delta t) \tag{17}$$

which clearly carries a truncation error $\mathcal{O}(\Delta x^2, \Delta t)$. Neglecting the higher order terms, we rearrange the equation and arrive at the explicit scheme

$$u_j^{m+1} = r\left(u_{j+1}^m - 2u_j^m + u_{j-1}^m\right) + u_j^m, \tag{18}$$

with $r \equiv \Delta t/\Delta x^2$. To achieve a stable solution, one must choose $r \leq 1/2$ which we show in the appendix, see section V B. When an upper-bound on r exists for a finite difference scheme, we refer to the scheme as *conditionally* stable, a classification which the explicit scheme is subordinated.

C. Neural Nets

In this section we give an overview of the mathematical formalism necessary for solving differential equations and eigenvalue problems with neural networks. The basic mathematical foundation behind neural networks, and topics like activation functions, optimizers, the backpropagation algorithm and performance metrics, are thoroughly explored in our previous work with neural nets in [3], and is thus not included in the report.

1. *General theory of construction of trial functions*

To solve the general DE in eq. (1), we define a trial solution $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^k$ which is partitioned onto two parts:

$$f(x, t, p) = B(x, t) + F(x, t, N(x, t, p)), \tag{19}$$

where $N(x, t, p)$ is a feed forward neural network with parameters p. The term $B(x, t)$ satisfies the initial- and boundary conditions and contains no adjustable parameters. The second term is constructed such that is does not affect these conditions, meaning that the problem has been reduced from a constrained optimization problem to an unconstrained one. The term $F(x, N(x, t, p))$ depends on the input data $(x, t) \in \mathbb{R}^d \times \mathbb{R}$ and the features p of the network, i.e weights and biases, as adjustable parameters. The combination of the two terms will thus satisfy the initial- and boundary conditions, while the network is trained such that the trial function satisfies the DE at hand.

2. *Trial function: diffusion equation*

Construction of the trial function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ to solve (2) must obey the constraints in eq. (3), which is achieved with

$$f(x, t, p) = \sin(\pi x) + tx(1 - x)N(x, t, p), \tag{20}$$

where N denotes the neural network and p are the adjustable parameters of the network, namely the weights and biases.

The basic procedure is to define the cost-function for minimization as

$$E[p] = \min_p \sum_{\substack{x_i \in D \\ t_i \in T}} \left(\partial_x^2 f(x_i, t_i, p) - \partial_t f(x_i, t_i, p)\right)^2 \tag{21}$$

which is equivalent to minimizing G^2 as in eq. (2).

3. *Trial function: the network DE*

The trial function $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^n$ constructed to solve eq. (5) needs to obey

$$\begin{aligned} f(0, p) &= x(0), \\ \lim_{t\to\infty} f(t, p) &= \lim_{t\to\infty} N(t, p), \end{aligned} \tag{22}$$

where $N(t, p)$ is the neural network model. Here d represents the number of parameters p. The following trial function possess these properties

$$f(t, p) = e^{-t}x(0) + (1 - e^{-t})N(t, p), \tag{23}$$

where $x(0)$ is the initial condition.

The appropriate cost-function to minimize is

$$E[p] = \min_p \sum_{t_i \in T} \left(\partial_t f(t_i, p) + f(t_i, p) - g(f_i)\right)^2, \tag{24}$$

where $g(f_i) \equiv g(f(t_i, p))$.

D. TensorFlow

TensorFlow is an end-to-end platform designed for easy building of machine learning models. It provided simple APIs to construct feed-forward neural networks with ease such as the ones we've used in this article.

We built our models core architecture using functionalities such as the model type *Sequential* and *layers* from the Keras framework. The *Dense* layers added to our model consists of one input layer with no activation function, variable numbers of hidden layers with the sigmoid activation function and one output layer using linear activation. We employed the optimizer Adam and the mean squared error loss function, both functionalities from Keras. The code produced to build our model is presented below.

```
class NeuralBase(tf.keras.Sequential):
    def __init__(self, layers, input_sz):
        super(NeuralBase, self).__init__()
        # First hidden layer connected to the
            input
        self.add(tf.keras.layers.Dense(
                layers[0],
                input_shape=(input_sz,),
                activation=None
            )
        )
        # Hidden layers
        for layer in layers[1:-1]:
            self.add(tf.keras.layers.Dense(
                    layer,
                    activation="sigmoid"
                )
            )
        # Output layer
        self.add(tf.keras.layers.Dense(
                layers[-1],
                activation="linear"
            )
        )
        self.optimizer = tf.keras.optimizers.
            Adam()
        self.loss_fn = tf.keras.losses.
            MeanSquaredError()
```

We use functionalities from TensorFlow to calculate and update gradients. This is quite technical and the usage depends highly on the shape of your data and the network model you wish to produce. We therefore redirect the eager reader to TensorFlow's own guide for building training loops[1]. Our codes [1] will naturally give the concrete implementation of the custom training loops which is open-source.

III. RESULTS & DISCUSSION

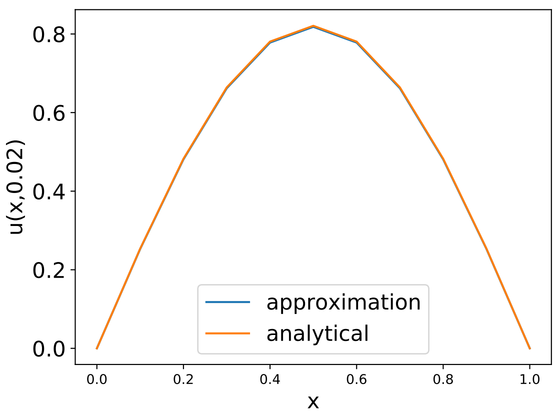
A. Forward Euler

FIG. 1 and 2 displays a comparison between the numerical solution of the one-dimensional diffusion equation using the forward Euler scheme and the analytical solution in eq. (4) at two different times for two different stepsizes, respectively $t \in \{0.02, 1.0\}$ and $\Delta x \in \{0.1, 0.01\}$.

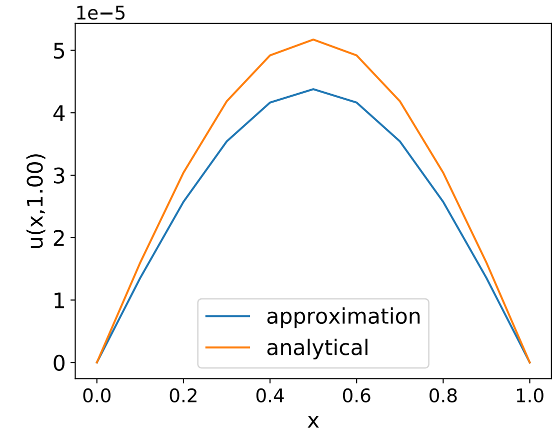
As seen from figures 1 and 2, and expected from the theory, the performance of the forward Euler scheme is highly dependent on the step size Δt, given that the explicit scheme carries a truncation error $\mathcal{O}(\Delta x^2, \Delta t)$. The numerical solution is in accordance with the analytical expression, and we observe that the solution goes to zero as $t \to \infty$. To obtain a more quantitative measurement of the performance of forward Euler, we looked at the relative error in the numerical solution in all timesteps when the total time was $t = 1.0$. The chosen stepsize was $\Delta x = 0.01$, based on the behaviour observed in figures 1 and 2. The result is displayed in figure 3.

In figure 3 we observe that the relative error accumulate with time, and that there are minuscule fluctuations in the relative error in the space domain for a given time, giving the impression that the error is independent of x.

[1] Custom training with tf.distribute.Strategy https://www.tensorflow.org/tutorials/distribute/custom_training



(a)



(b)

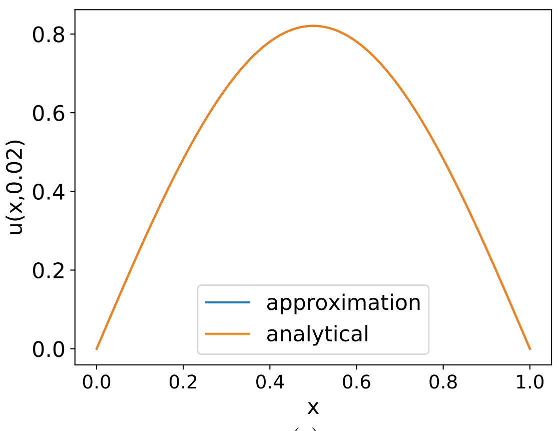
FIG. 1. The figures show the numerical solution for the forward Euler algorithm versus the analytical solution for two different values of t. In figure **(a)** the time is $t = 0.02$, and in figure **(b)** the time is $t = 1.0$. The parameters were set to $r = 0.5$, $\Delta x = 0.1$ and $\Delta t = r(\Delta x)^2$.

B. Diffusion Equation with Neural Network

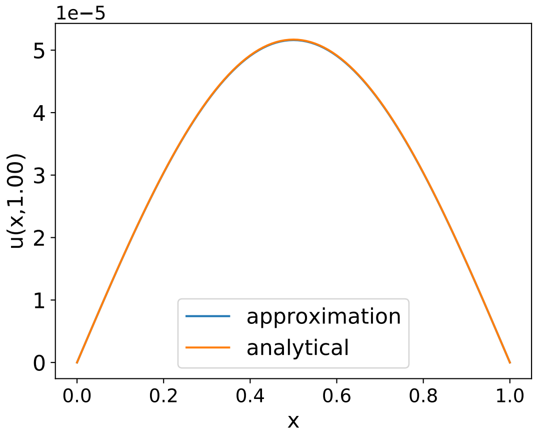
1. *Parameter tuning*

To obtain optimal values for the parameters in our neural network we performed a gridsearch on the parameters nodes in hidden layers and number of hidden layers. We looked at the R_2 score achieved by our model when predicting on the same domain as the training domain $x \in [0, 1]$ and $t \in [0, 1]$. The resulting heatmap is shown in figure 4.

In figure 4 one can identify a region of good performance between $2 - 8$ hidden layers with $500 - 1000$ neurons in each layer. With this result in mind we chose number of hidden layers to 4 and number of neurons in each layers to be 1000 when producing the rest of our results.



(a)



(b)

FIG. 2. The figures show the numerical solution for the forward Euler algorithm versus the analytical solution for two different values of t. In figure **(a)** the time is $t = 0.02$, and $t = 1.0$ in figure **(b)**. The parameters were set to $r = 0.5$, $\Delta x = 0.01$ and $\Delta t = r(\Delta x)^2$.

2. *Analytical solution vs. neural network*

The neural net was trained using 4 hidden layers with 1000 neurons in each on $n = 10^5$ evenly distributed datapoints $(x, t) \in [0, 1]^2$. We opted for the built-in Adam optimizer with a learning rate $\eta = 0.001$ and trained the net for 10000 epochs. Every pixel of the images shown in this section consist of an observed value.

In figure 5 we display the trial function evaluated at evenly distibuted points $(x, t) \in [0.01, 0.99] \times [0, 1.2]$. We include this only to show that the qualitative features of the true solution is indeed captured by the network, even though it fails at the finer details, which is discussed shortly.

In figure 6, we show the relative error for $(x, t) \in [0.01, 0.99] \times [0, 0.5]$ uniformly distributed. This result indicates that the neural net is approximating the solution fairly well for small t, at a level which it can compete with the forward Euler scheme as shown in figure 3.

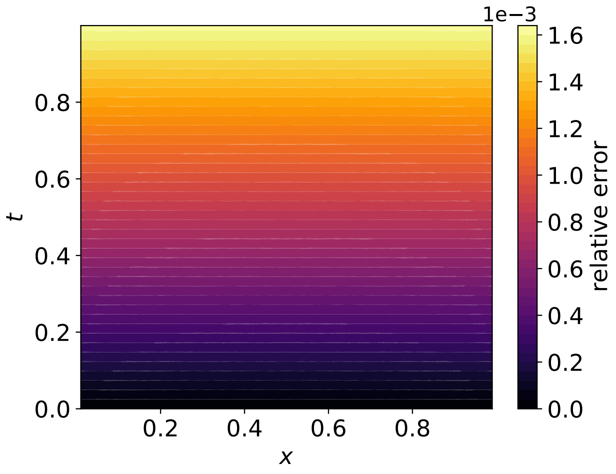
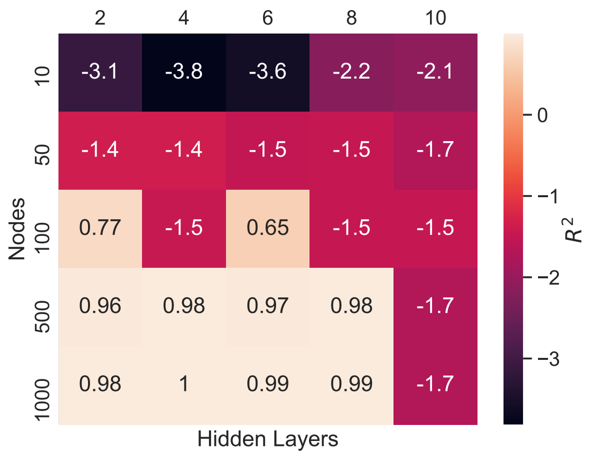


FIG. 3. The figure displays the relative error for the forward Euler scheme in every timestep when the total time is $t = 1.0$, with remaining parameters set to $r = 0.5$, $\Delta x = 0.01$ and $\Delta t = r(\Delta x)^2$.



Nodes \ Hidden Layers	2	4	6	8	10
10	-3.1	-3.8	-3.6	-2.2	-2.1
50	-1.4	-1.4	-1.5	-1.5	-1.7
100	0.77	-1.5	0.65	-1.5	-1.5
500	0.96	0.98	0.97	0.98	-1.7
1000	0.98	1	0.99	0.99	-1.7

FIG. 4. The heatmap shows the achieved R_2 score for the solution of the diffusion equation of our neural network when training and predicting using various amounts of hidden layers and nodes in hidden layers on the domain $x \in [0, 1]$ and $t \in [0, 1]$.

In figure 7, the relative error for evenly distributed $(x, t) \in [0.01, 0.99] \times [0.5, 0.8]$ is displayed and in figure 8, the relative error for evenly distributed $(x, t) \in [0.01, 0.99] \times [0.8, 1.2]$ is shown. From these results, it's clear that as t grows, so does the relative error. In the area outside the training domain, i.e $t > 1$, we observe a significant increase in the relative error which suggests the net generalizes poorly outside the domain its fitted on.

In figure 9, the absolute error for evenly distributed

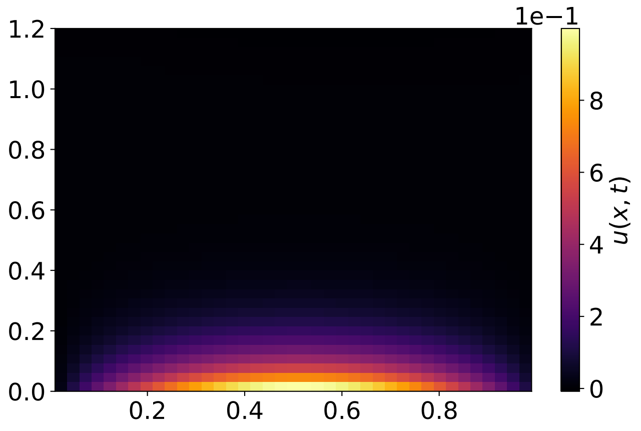


FIG. 5. The figure shows the trial function evaluated at $(x,t) \in [0.01, 0.99] \times [0, 1]$. The network was trained on $n = 10^5$ evenly spaced points on $(x,t) \in [0,1]^2$ with 4 hidden layers consisting of 1000 neurons.

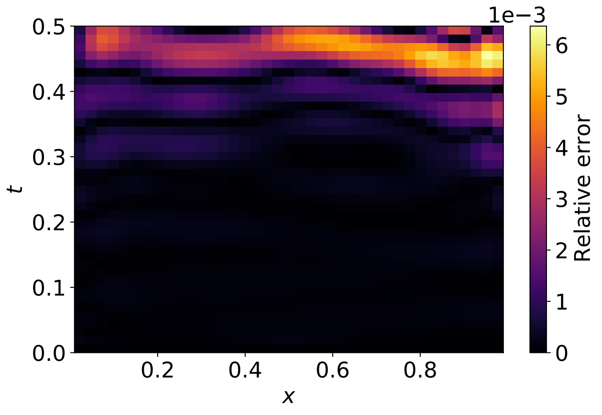


FIG. 6. The figure shows the relative error for $(x,t) \in [0.01, 0.99] \times [0, 0.5]$. The network was trained on $n = 10^5$ evenly spaced points on $(x,t) \in [0,1]^2$ with 4 hidden layers consisting of 1000 neurons.

points $(x,t) \in [0.01, 0.99] \times [0, 1]$ is shown. The main observation to note from this result is that the absolute error remains the same order of magnitude throughout large areas of the grid, which at the very least explains why the relative error increases as t grows (since the true solution converges to zero as $t \to \infty$). Why this is the case, is difficult to assess without further testing. Perhaps the minimization procedure chosen here is not fit for functions that span several orders of magnitude. However, we make no specific claims as to why this happens, but rather suggest that this should be investigated further. We trained the network using the freely available access to a single GPU at Google Colab, but the achieved hardware acceleration was limited due to only having access to a single GPU and limited memory access.

From the results here, it's clear that solving the diffu-

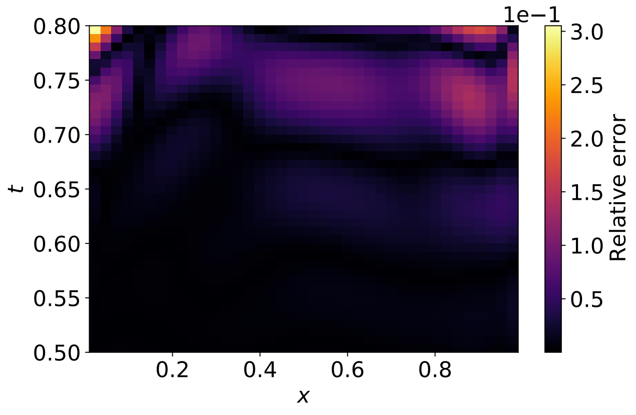


FIG. 7. The figure shows the relative error for $(x,t) \in [0.01, 0.99] \times [0.5, 0.8]$. The network was trained on $n = 10^5$ evenly spaced points on $(x,t) \in [0,1]^2$ with 4 hidden layers consisting of 1000 neurons.

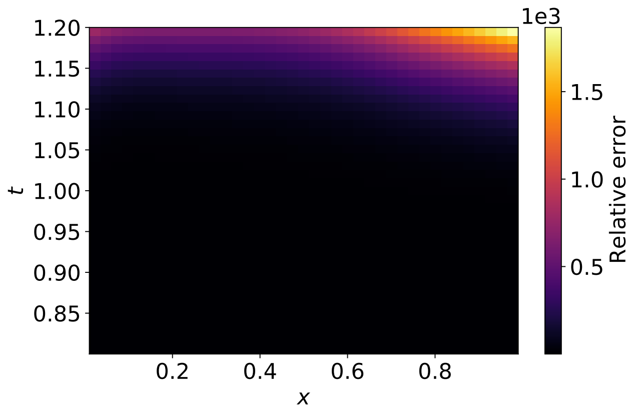


FIG. 8. The figure shows the relative error for $(x,t) \in [0.01, 0.99] \times [0.8, 1]$. The network was trained on $n = 10^5$ evenly spaced points on $(x,t) \in [0,1]^2$ with 4 hidden layers consisting of 1000 neurons.

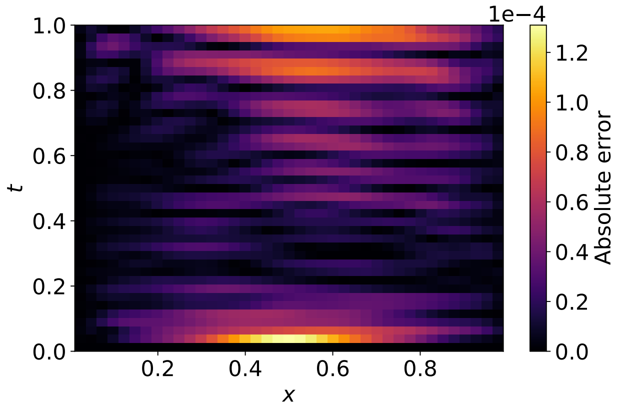


FIG. 9. The figure shows the absolute error error for $(x,t) \in [0.01, 0.99] \times [0, 1]$. The network was trained on $n = 10^5$ evenly spaced points on $(x,t) \in [0,1]^2$ with 4 hidden layers consisting of 500 neurons.

sion equation using a neural network isn't a better choice than using the forward Euler scheme if accuracy is desirable, at the very least not on standard laptops. If server grade GPU(s) are available and the dimensionality is larger, the methods presented here might be suitable because training the network will be significantly faster and will yield a solution function in closed-form (at least in principle) that may generalize well to regions outside the training domain. But this is speculative and testing this is necessary to determine whether use of neural networks to solve PDEs is a tractable method. We conclude that with limited access to hardware acceleration, conventional methods like finite difference schemes are more suitable for solving the one-dimensional diffusion equation than the neural net methods discussed here, especially if accuracy and time-consumption is important. We do not know if this generalizes to other PDEs and suggest this to be tested in the future.

3. R^2-score as a predictor of performance

The R^2-scores used to determine number of layers and nodes seem to indicate a model which performs nearly perfectly on the training domain, however the relative errors presented clearly contradict this notion. This leads us to conclude that R^2-scores on its own is is as a predictor of good performance is of limited value when studying the neural network methods for DEs.

C. Eigenvalues and Eigenvectors

In figure 10 and 11 we show the estimate of the maximum eigenvalue and its corresponding eigenvector as a function of training epochs. The initial vector $x(0)$ was generated randomly where each component was sampled from the normal distribution $\mathcal{N}(0,1)$. The neural net was fitted on the interval $t \in [0, 1000]$ with $N = 50$ points on a uniform mesh. The estimates were computed at $t = 1000$. The neural net had a single hidden layer with 5000 nodes.

From the figures, we can note that the eigenvalue estimate converges to the maximum eigenvalue quickly (at about 500 epochs), but the eigenvector components converges a bit later (at around 1000 epochs).

In figure 12 and 13 we show the estimate of the smallest eigenvalue and its corresponding eigenvector as a function of training epochs. All parameters used were the same.

We observe that the convergence is a bit slower in this case, but that the eigenvalue estimate converges faster (in about 1000 epochs) while the eigenvector converges a bit after (roughly after 1500 epochs).

In figure 14, we show the eigenvalue estimate of the model where $x(0)$ started from the eigenvector with the smallest eigenvalue. The estimate was performed with the same model as in the two former cases with $A \to -A$ such that convergence to the eigenvector with the smallest eigenvector should be a certainty.

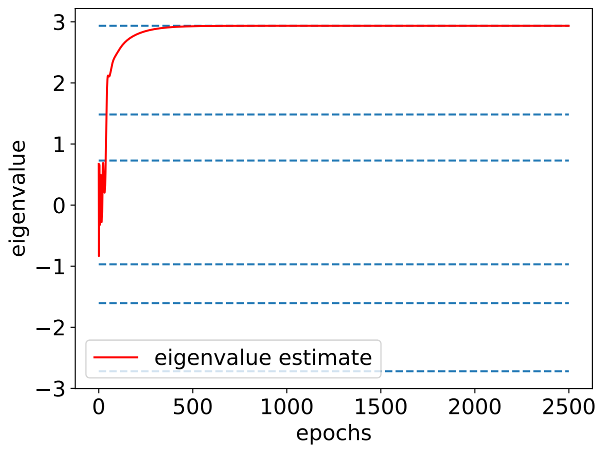


FIG. 10. The figure shows the estimated eigenvalue as a function of training epochs using A. The neural network used had a single hidden layer with 5000 nodes. The correct eigenvalues of A are shown as blue dashed lines. These were computed with Numpy's eigenvalue solver *eig*.

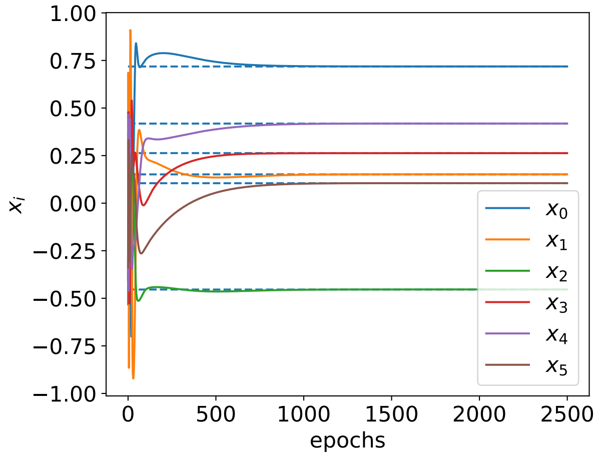


FIG. 11. The figure shows the components of the estimated eigenvector corresponding to the maximum eigenvalue estimate in 10 as a function of training epochs using A. The neural network used had a single hidden layer with 5000 nodes. The components of the correct eigenvector of A with the maximum eigenvalue is shown as blue dashed lines. These were computed with Numpy's eigenvalue solver *eig*.

This result is important because it contradicts the assertion that if $x(0)$ lies in the eigenspace of the smallest eigenvalue and we replace $A \to -A$, it necessarily converges to the eigenvector with the smallest eigenvalue. What we have here appears to be a smoking gun, because this result shows that this is not necessarily the case. We do not claim that the theory necessarily is wrong, however, because there are several factors that could play an important role here. The authors of the theory claim $x(t)$ is to represent the *state of the network*

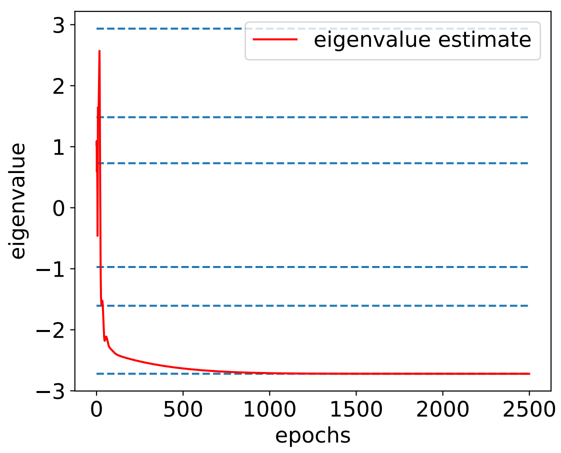


FIG. 12. The figure shows the estimated minimum eigenvalue as a function of training epochs. The computation was performed by replacing $A \to -A$. The correct eigenvalues of A are shown as blue dashed lines. These were computed with Numpy's eigenvalue solver *eig*.

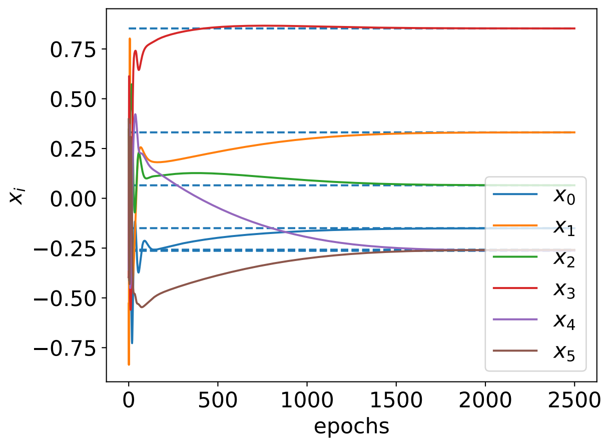


FIG. 13. The figure shows the components of the estimated eigenvector corresponding to the maximum eigenvalue estimate in 10 as a function of training epochs by replacing $A \to -A$. The neural network used had a single hidden layer with 5000 nodes. The components of the correct eigenvector of A with the smallest eigenvalue is shown as blue dashed lines. These were computed with Numpy's eigenvalue solver *eig*.

[8]. We have in this article, however, created a trial function in which only a part of the function corresponds to the neural net, which may not be what the authors had in mind. They derive the actual general solution of $x(t)$ which has properties that are strictly mathematical and independent of the notion of neural networks. The validity of their results must be disputed on grounds that the mathematical logic carried out is flawed, and our numerical results here does not in any way do this. This does not give us grounds to doubt that our implementation is wrong, however, because the theory ultimately states

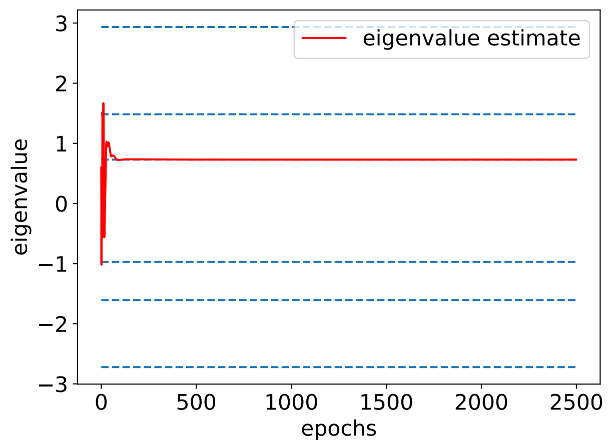


FIG. 14. The figure shows the estimated eigenvalue as a function of training epochs with $x(0)$ being the eigenvector with the smallest aigenvalue. The computation was performed by replacing $A \to -A$. The correct eigenvalues of A are shown as blue dashed lines. These were computed with Numpy's eigenvalue solver *eig*. Most importantly, note that the eigenvalue estimate converges to an eigenvalue distinct from the smallest eigenvalue, effectively contradicting the theoretical grounds described in the source material [8].

that the solution will converge to *some* eigenvector for an arbitrary $x(0)$ which all our results provide evidence for. Further testing using the neural nets with different architectures should be performed to assess whether this result is a general feature or if perhaps the mathematical formalism applied by the authors doesn't directly translate into the methods used here.

IV. CONCLUSION

We implemented neural network methods to solve general DEs. We applied the method to the one-dimensional diffusion equation for which we compared it to the performance of the forward Euler scheme with an analytic solution for reference. We found that the conventional method outperformed the neural network based methods given the limited access to hardware acceleration. We conclude based on our results that these methods are not to be preferred over conventional finite difference methods if accuracy and time-consumption is important. We don't know how these results generalize to other PDEs and suggest further work on both the PDE studied here and other PDEs are studied further with access to hardware acceleration such as GPUs or neural processors specifically designed to acceleration typical neural network operations.

We applied the neural network method to an ODE designed to yield eigenvectors and eigenvalues of symmetric matrices as equilibrium states. We found that our implementation converge to correct eigenvectors and eigenvalues with a fairly small number of epochs, but found discrepancies between our results and the theorems on which the theory rests. Firstly, the solution should not be able to converge to eigenvectors that lies orthogonal to the eigenspace the initial condition reside, but our results indicates that this is not the case. We conclude there can be several reasons for this, but we think that it's most likely due to the methods we've implemented does not directly translate into what the authors of [8] had in mind. They vaguely coin $x(t)$ as the *state* of the network and it's unclear whether the methods we've based the solution of the DE on is consistent with this. The authors are furthermore not transparent with respect to precisely how their method is implemented.

V. APPENDIX

A. Analytical solutions of the diffusion equation

First step in solving the partial differential equation

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{25}$$

with the boundary- and initial conditions

$$\begin{aligned} u(0,t) &= 0, \\ u(1,t) &= 0, \\ u(x,0) &= \sin(\pi x), \end{aligned} \tag{26}$$

is to make an educated guess about the solution. We simply guess that the solution is separable with respect to x and t, and make the ansatz

$$u_n(x,t) = X_n(x)T_n(t). \tag{27}$$

Inserting this into eq. (25) yields

$$X_n T_n' = X_n'' T_n. \tag{28}$$

Next we divide both sides in eq. (28) with $X_n T_n$ to obtain the following relation,

$$\frac{T_n'}{T_n} = \frac{X_n''}{X_n} = -k_n^2, \tag{29}$$

where k_n^2 is a constant. The left- and right side of the expression above is only dependent of their respective variables, hence, both expressions must be equal to a shared constant. Eq. (29) can now be recast into two ordinary differential equations:

$$T_n' = -k_n^2 T_n, \tag{30}$$

$$X_n'' = -k_n^2 X_n. \tag{31}$$

Looking at eq. (30), and requiring on physical grounds that the solution $u(x,t) \to 0$ when $t \to \infty$, it is evident that the time-dependent part of the solutions reside in the subspace,

$$T_n \propto e^{-k_n^2 t}. \tag{32}$$

Eq. (31) is easily recognised as a harmonic oscillator, which has the general solution

$$X_n = A_n \cos(k_n x) + B_n \sin(k_n x), \tag{33}$$

for some constants A_n and B_n. The boundary conditions gives $A_n = 0$, and the constant k_n^2.

$$X_k(1) = B_n \sin(k_n \cdot 1) = 0 \qquad \Rightarrow \qquad k_n = n\pi, \tag{34}$$

for integer values of $n \geq 1$. From eq. (33) and (34) it follows that the solution is of the form

$$u_n(x,t) = e^{-k_n^2 t} B_n \sin(k_n x), \tag{35}$$

$$u(x,t) = \sum_{n=1}^{\infty} B_n \sin(n\pi x) e^{-(n\pi)^2 t}. \tag{36}$$

In the last step we applied the principle of superposition. It states that any linear combination of particular solutions creates a new solution. To decide the Fourier constants $\{B_n\}_{n=1}^{\infty}$, we make use of the initial condition

$$\sum_{n=1}^{\infty} B_n \sin(n\pi x) = \sin(\pi x). \tag{37}$$

Which gives $A_1 = 1$ and $A_n = 0$ for $n \neq 1$. Hence, the solution of the one dimensional diffusion equation is given by

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t}. \tag{38}$$

B. Stability analysis

In this section we will follow the treatment presented here [7]. To assess the stability of the methods, we'll apply von Neumann analysis which is to require that the time-dependent part of the discrete solution is bounded by the maximum value of the time-dependent part of the solution for the continuous case. The discrete solution of eq. (2) can be written as

$$u_j^m = (a_k)^m e^{ik\pi x_j}, \tag{39}$$

where $x_j \equiv j\Delta x$ and $k = 0, \pm 1, \pm 2, ...$; Mathematically, then, the criterion to impose is

$$|(a_k)^m| \leq \max_{t \in [0,\infty)} \left| e^{-k^2 t} \right| = 1. \tag{40}$$

Inserting the discrete solution into eq. (18) for the explicit scheme produces the following equation

$$\frac{(a_k)^{m+1} - (a_k)^m}{\Delta t} e^{ik\pi j\Delta x} = \\ \frac{e^{ik\pi(j-1)\Delta x} - 2e^{ik\pi j\Delta x} + e^{ik\pi(j+1)\Delta x}}{\Delta x^2} (a_k)^m, \tag{41}$$

which can be rewritten into

$$\frac{a_k - 1}{\Delta t} = \frac{e^{ik\pi\Delta x} - 2 + e^{-ik\pi\Delta x}}{\Delta x^2} = 2\left[\frac{\cos(k\pi\Delta x) - 1}{\Delta x^2}\right]. \tag{42}$$

Using the trigonometric identity

$$\sin\left(\frac{x}{2}\right) = \pm\sqrt{\frac{1-\cos x}{2}},$$

we obtain the relation

$$a_k = 1 - 4r\sin^2\left(\frac{k\pi\Delta x}{2}\right). \tag{43}$$

From eq. (40), the following criterion for r arises

$$\begin{aligned}\left|1 - 4r\sin^2\left(\frac{k\pi\Delta x}{2}\right)\right| &= \\ \left|4r\sin^2\left(\frac{k\pi\Delta x}{2}\right) - 1\right| & \\ &\leq |4r - 1| \leq 1,\end{aligned} \tag{44}$$

from which it follows that the upper-bound on r to attain a stable simulation is

$$r \leq \frac{1}{2}. \tag{45}$$

What this is means is that r must be chosen according to eq. (45) to obtain a stable numerical simulation.

[1] Link to our Github repository with code documentation. https://github.com/reneaas/fys-stk4155/tree/master/project3/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] René Ask, Kaspara Gåsvær, and Maria Horgen. A comparative study of neural nets and vanilla methods in regression and classification tasks. https://github.com/reneaas/fys-stk4155/blob/master/project2/report, 2020.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[6] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[7] Aslak Tveito, Ragner Winther. *Introduction to Partial Differential equations: A Computational Approach*, chapter 4.2-4.3. Springer, 2005.

[8] 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):1155 – 1164, 2004.