Solving Differential Equations with Neural Networks

Nicholas Karlsen and Thore Espedal Moe
University of Oslo
(Dated: December 17, 2020)

We investigate the application of neural networks to the solution of both partial differential equations, exemplified by the 1-D heat equation; as well as systems of nonlinear ordinary differential equations, exemplified by a system of equations describing the dynamics of a class of Recurrent Neural Networks that can be used to find eigenvalues and eigenvectors of a real, symmetric matrix. For the heat equation we compare the solution from a neural network with both the analytic solution and a numerical solution from a simple Finite-Difference scheme for different resolutions of the computational grid and different amounts of training points for the network. We perform a similar analysis for the nonlinear system; this time comparing the networks solution with a numerical integration using the Forward-Euler method. However, as the nonlinear system does not readily admit an analytical solution, only the equilibrium state of the system can be independently computed (through standard numerical eigendecomposition of the mentioned matrix) as a "ground truth" of the solution. In general we find that the neural network-based solution can reach a reasonable level of accuracy with fewer grid-points/training points than the standard numerical integration schemes, and with far less stringent requirements on the distance between grid-points. Notably, the neural network-solution does not need its training grid-points to satisfy the famously restrictive stability criterion for the Finite-Difference method applied to the heat equation. We do find, however, that it is more difficult to scale up the accuracy of the network-based solutions, as compared to more traditional schemes.

I. INTRODUCTION

Per the preponderating proliferation of differential equations in the sciences, both natural and otherwise, there is intense and inherent interest in novel ways of obtaining their solutions. Thus, one of the more exciting developments in machine learning, and the topic of this paper, is the application of neural networks to the task of solving differential equations.

We will be looking at two distinct problem types; partial differential equations (PDE) and systems of nonlinear ordinary differential equations (ODE). Concretely, we will be using the 1-D heat equation as an example of a PDE, while a method from [1] for finding eigenvalues and eigenvectors of real and symmetric matrices will serve as the example of nonlinear systems of ODEs.

In both cases we will compare solutions obtained from neural network-based approaches with those from more traditional numerical methods, namely the simplest available Finite-Difference schemes. For the heat equation an analytic solution is easily obtained as the ground truth of the methods. For the eigenproblem-related system an analytic solution is not easily acquired; rather, as the steady-state equilibrium points of the system correspond to the eigenvectors of a given matrix, the asymptotic solution can be computed by standard numerical eigendecomposition of the matrix.

Structurally, we begin with a brief discussion of how neural networks can be applied to solve differential equations in general. We then move on to a more detailed description of the two particular use cases, starting with a short introduction to the context in which they arise, as well as some aspects of their analytically known properties. Continuing on we describe the traditional methods we use for solving the equations, before presenting the specific setup of the neural network-based methods.

Subsequently we compare the performance of the network-based methods with the traditional numerical integration-methods, paying special attention to the effects and requirements of the computational grids' sizes on the convergence of the methods. We find that, in general, the neural networks place far laxer demands on the computational grids for achieving reasonable convergence. Especially noteworthy, we find that the distance between training grid-points for the neural network in the PDE-case do not have to satisfy the same strict stability criterion as the Finite-Difference scheme. However, we do note that it is more difficult to increase the accuracy of the network-methods, as compared to the traditional schemes. In most regards, we conclude that the network-based methods are inferior to the standard methods for the toy problems at hand. However, for different/larger problem dimensions we recognize the possibility of the network-based methods to achieve superior performance in terms of computational cost for some levels of accuracy.

II. THEORY & METHODS

A. Solving Differential Equations with Neural Networks

In general, a ordinary differential equation (ODE) may be written as a function in the form

$$F\Big[x, u(x), u'(x), u''(x), \dots, g^{(n)}(x)\Big] = 0 \qquad (1)$$

where $u(x)$ denotes the solution of the ODE. One may then propose a trial solution $\tilde{u}(x, P)$ in the form

$$\tilde{u}(x, P) = u_{bc}(x) + u_{nn}(x)N(x, P) \tag{2}$$

where $u_{bc}(x), u_{nn}(x)$ are functions which enforce the boundary conditions of the ODE and $N(x, P)$ is a neural network with independent variables denoted by $P = \{W, \boldsymbol{b}\}$, the set of weights and biases in the network respectively. In order to optimize $N(x, P)$ such that $\tilde{u}(x)$ yields a solution of the ODE, we define the (scalar) cost function $\mathcal{C}$ as

$$\mathcal{C}[u(x)] = \left(F\left[x, u(x), u'(x), \ldots, u^{(n)}(x)\right]\right)^2 \tag{3}$$

for each training point x. In essence, the problem of solving the differential equation through neural networks then becomes the convex optimization problem of making the equation 1 self-consistent for the chosen trial solution. For vector-valued functions (systems of differential equation) the above cost-function is straightforwardly modified to become instead the mean of the scalar cost-functions for the individual vector-components. The extension to partial differential equations (PDE) and functions of several variables should be obvious. We may then construct a Feed-Forward Neural Network (FFNN) where, for each iteration, we update the weights and biases of the network according to some optimization scheme containing the derivatives of the cost-function with respect to the network parameters.

In our previous paper [2] we gave an overview of FFNNs and how they can be trained using the back-propagation algorithm with stochastic gradient descent and mini-batching. For a more in-depth exposition, along with a derivation of the backpropagation algorithm, the reader is referred to the excellent textbook by Nielsen [3]. We will thus not repeat much of our previous discussion regarding FFNNs in the present report. Instead, we only note the considerations unique to the application of FFNNs for solving differential equations in general; and some aspects of our chosen optimization scheme in particular.

The key difference between the regression and classification problems considered in [2] and the currently considered differential equations lies in the form of the cost-function. Whereas we previously worked with cost-functions only having a functional dependence on the network output, we are now faced with a cost-function depending on both the network output and the derivatives of the network output with respect to the network input. This renders the backpropagation algorithm, at least without modifications, rather inapplicable. In order to start the backpropagation one needs to evaluate the error in the outermost layer of the network (which is given by the partial derivatives of the cost-function with respect to the output activations), before one propagates it towards the inner layers according to repeated invocations of the chain rule. For our previously pondered problems this was very simple, as the error in the outermost layer could be computed in a single step yielding a number. Now, however, the computation of the network derivatives with respect to the inputs in the cost-function requires the whole chain of chain rule applications to already have been performed before even starting the backpropagation. The input-derivatives must be set up in advance of the backpropagation, defeating the purpose of using the algorithm for simplifying the chain-rule computations. Furthermore, once one has obtained by hand an expression for the derivatives of the network with respect to the inputs, it becomes a nontrivial, or at least an extremely tedious, task to ensure that the functional dependencies of the input-derivatives are carried correctly through the backpropagation, should one insist on maintaining that scheme. It is more straightforward, though still very tedious, to obtain expressions for all the desired derivatives by hand and evaluating these expressions directly. The unique challenge presented by differential equations is thus the computation of the network derivatives.

For some network architectures it is possible to obtain fairly simple representations of the derivatives of these types of cost-functions with respect to the network parameters, see e.g. [4]. However, it is in general quite burdensome to set up the network-derivatives by hand. Thus a major boost for the use of neural networks in solving differential equations comes from the advent of Automatic Differentiation, which by tracking the elementary operations performed in the network computations is capable of accurately and automatically producing the sought-after gradients. In the present work we rely on the automatic differentiation functionality of PyTorch, to produce all necessary derivatives.

Further, we also utilize the Adam optimization algorithm [5] for updating the weights and biases, an adaptive variant of stochastic gradient decent which we previously discussed in [2] that combines the benefits of AdaGrad and RMSProp. The model also benefits from being rather simple, only containing four free parameters. The learning rate α, exponential decay rates β_1, β_2 and a small term ϵ, usually set to 10^{-8}, which role is to prevent division by zero. The authors also a propose a set of default parameters $\alpha = 0.001$ $\beta_1 = 0.9$ and $\beta_2 = 0.999$, which serves as a good starting point when using Adam. For the full context of these parameters, we refer the reader to the original paper by Kingma and Ba [5] which lists and describes the algorithm in full detail.

B. The Heat Equation

As our first example we investigate the application of neural networks for solving the so-called heat equation. In physics, the heat equation models how the temperature of some material $u(\boldsymbol{r}, t)$ evolves over time by the

partial differential equation (PDE)

$$\frac{\partial u}{\partial t} = c^2 \nabla^2 u \tag{4}$$

where c^2, the thermal diffusivity is a material dependent constant. This may be reduced to a one dimensional problem, modeling i.e a thin, insulating wire of length L as

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{5}$$

where we have also for simplicity set $c^2 = 1$. We may then solve this analytically with boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = f(x)$.

1. *Solving the Heat Equation analytically*

We start by making the assumption that $u(x,t)$ is separable, giving

$$u(x,t) = F(x) \cdot G(t) \tag{6}$$

which lets us rewrite Eqn. 5 to the form

$$F \cdot \frac{\partial G}{\partial t} = \frac{\partial^2 F}{\partial x^2} \cdot G \tag{7}$$

which may be expressed as

$$\frac{\dot{G}}{G} = \frac{F''}{F} = k \tag{8}$$

where k is some constant. We further have that the boundary conditions may only be satisfied for $k < 0$, so we set $k = -\rho^2$ and write

$$F'' + \rho F = 0 \qquad \dot{G} + \rho^2 G = 0 \tag{9}$$

starting with the spatial equation, we have a general solution in the form

$$F(x) = c_1 \cos(\rho x) + c_2 \sin(\rho x) \tag{10}$$

where $c_1 = 0$ is fixed by $u(0,t) = 0$. Similarly, $u(L,t) = 0$ imposes the requirement $\sin(\rho x) = 0$ which is satisfied by setting $\rho = n\pi/L$, yielding a discrete spectrum of solutions

$$F_n(x) = \sin\left(\frac{n\pi x}{L}\right) \tag{11}$$

where we omit the constant c_1, which we will include later in $G(t)$.

We then move on to the temporal part, $G(t)$, which we write as

$$\dot{G} = -\rho^2 G \tag{12}$$

which we immediately recognize as having a general solution

$$G(t) = c_n e^{-n^2\pi^2 t/L^2} \tag{13}$$

Thus, $u(x,t)$ is solved by a superposition of the discrete set of eigenfunctions

$$u_n(x,t) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) e^{-n^2\pi^2 t/L^2} \tag{14}$$

with initial condition

$$u(x,0) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) = f(x) \tag{15}$$

a Fourier series. Thus, the coefficients c_n are determined by the integral

$$c_n = \frac{2}{L} \int_0^L \mathrm{d}x\, f(x) \sin\left(\frac{n\pi x}{L}\right) \tag{16}$$

if we set the initial condition $f(x) = \sin(\pi x)$ and fix $L = 1$, we get coefficients

$$c_n = 2 \int_0^1 \mathrm{d}x\, \sin(\pi x) \sin(n\pi x) = 1 \tag{17}$$

we thus have an analytical solution for the heat equation with boundary conditions $u(0,t) = u(L,t) = 0$, $L = 1$, and initial condition $u(x,0) = \sin(\pi x)$ as

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t} \tag{18}$$

a plot of which for $t \in [0,1]$ is shown in Fig. 1.

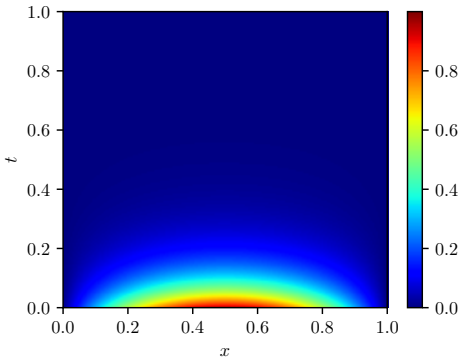


FIG. 1: Analytic solution to the heat equation for $(x,t) \in [0,1] \times [0,1]$ with initial condition $u(x,t) = \sin(\pi x)$ and boundary conditions $u(0,t) = u(1,t) = 0$

2. *Solving the Heat Equation through a simple Finite-Difference scheme*

While we in this case were able to obtain a simple analytic solution to the differential equation, that is generally not possible for most PDEs so numerical methods must be

used instead. The simplest, though neither the most accurate nor the most stable, class of numerical methods for solving PDEs are the so-called explicit finite-difference (FD) schemes. The main idea consists of dividing the computational domain into a set of grid points, and discretizing the differential equations to obtain relations between the function values at those grid-points.

The distinction between explicit and implicit schemes is, of course, that with an explicit scheme the values at the next grid points can be explicitly calculated from the previous grid points so that given the boundary conditions one obtains a rule for calculating the function values of the whole domain one step at the time. Oppositely, implicit schemes give the value at a grid-point as a function of both that grid-point's value and the values of the previous grid-points thus requiring the (usually numerical) solution of a linear system of equations for each grid-point. The benefit of implicit methods is that they usually have more lax restrictions on how the grid-points must be spaced in order to maintain stability of the solution.

For the case at hand we utilize the following FD scheme:

$$u(x_i,t_i) = (u(x_i-\Delta x, t_i-\Delta t) - 2u(x_i,t_i-\Delta t) + u(x_i+\Delta x, t_i-\Delta t))\cdot\frac{\Delta t}{\Delta x^2} \tag{19}$$

The scheme is obtained by first discretizing the computational domain into a set of grid points x_i and t_i with equidistant spacing Δx and Δt, and subsequently approximating the derivatives by the function u evaluated at neighboring grid points in the following manner:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x_i-\Delta x) - 2u(x_i) + u(x_i+\Delta x)}{\Delta x^2} \tag{20}$$

and

$$\frac{\partial u}{\partial t} \approx \frac{u(t_i+\Delta t) - u(t_i)}{\Delta t} \tag{21}$$

Simple algebraic manipulations of the two equations above yields the scheme 19, which step by step yields the solution starting from the boundary conditions on $t_0 = 0$ and $x_0 = 0$. The scheme can be compactly rewritten as a matrix equation in the form

$$\boldsymbol{u}(t_i+\Delta t) = \mathbf{B}\boldsymbol{u}(t_i)\frac{\Delta t}{\Delta x^2} \tag{22}$$

where $\boldsymbol{u}(t_i) = (u(x_0,t_i), u(x_1,t_i),\dots)^T$ and $\mathbf{B}$ is a tridiagonal matrix in the form

$$\mathbf{B} = \begin{bmatrix} 0 & & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & & 0 \end{bmatrix} \tag{23}$$

where the zero-padding on top and bottom rows takes care of the boundary conditions. It is thus extremely easy to setup the computation of this FD-scheme, and should computational speed be critical there are robust, standard numerical linear-algebra-techniques for sparse matrices that outperform naive matrix-vector-multiplication. The really crushing caveat for this approach to differential equations, is however as mentioned stability concerns. It is a standard exercise in numerical analysis courses to show that this method is only stable so long as the stability criterion $\Delta t/\Delta x 2 \leq 1/2$ is fulfilled. Quite separately from the local approximation error in each step due to the discretization of the computational grid and the derivatives, failure to obey the stability criterion will lead to rounding errors eventually hijacking the solution's trajectory, pushing it off-course and blowing it up. For a discussion of stability vs approximation error the reader is referred to e.g. chapter 4 in [6]. The main point, for our purposes, is to note that an e.g. tenfold decrease in the separation between grid-points in the x-dimension (Δx) necessitates a hundredfold decrease in the separation between grid-points in the t-dimension (Δt). That is, a linear increase in resolution (number of grid-points) along one dimension comes at the cost of a quadratic increase in grid-points for the other dimension in order to maintain stability. Thus the cost of increasing the accuracy/approximation error of the solution (determined by the combined effects of both grid-spacings) as well as the resolution (synonymous with the number of grid points the solution is computed at) scales rather poorly.

3. *Solving the Heat Equation with Neural Networks*

We now turn to the use of a neural network as an alternative method for obtaining the solution of the heat equation. We start by re-writing equation. 5 to the form

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{24}$$

which gives rise to the cost function

$$\mathcal{C}[u(x,y)] = \left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)^2 \tag{25}$$

for a single point (x,t). For a mini-batch, one would sum and average the cost over the entire batch. setting the boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = \sin(\pi x)$, we then have a trial solution in the form

$$u_t(x,t,P) = (1-t)\sin(\pi x) + x(1-x)tN(x,t,P) \quad (26)$$

where $N(x,t,P)$ denotes the output of the neural network.

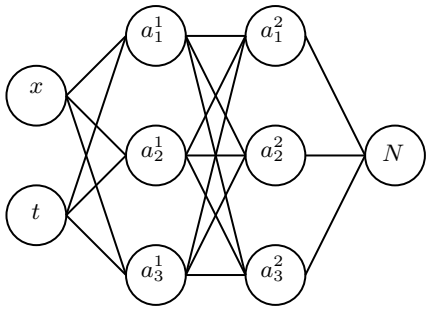


FIG. 2: Possible network structure of a Feed-Forward network with input (x,t) and 2 hidden layers each with 3 neurons with activations denoted as a_j^l and a single output N.

C. A Nonlinear System of Ordinary Differential Equations for Finding Eigenvalues and Eigenvectors of Real Symmetric Matrices

For our second example, we investigate the use of neural networks for the solution of a nonlinear system of ordinary differential equations. The particular use case is the system of differential equations presented by [1] for finding the largest eigenvalue λ_1 and corresponding eigenvector $\boldsymbol{v}_1$ of an $N \times N$ real and symmetric matrix $\mathbf{A}$:

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = -\boldsymbol{x}(t) + [\boldsymbol{x}(t)^T\boldsymbol{x}\mathbf{A} - (1-\boldsymbol{x}(t)^T\boldsymbol{x}(t))\mathbb{I}]\boldsymbol{x}(t) \quad (27)$$

where $\boldsymbol{x}(t)$ is a $1 \times N$ column vector and $\mathbf{I}$ is an $N \times N$ identity matrix. In [1] it is shown that for any initial vector $\boldsymbol{x}(0)$ which is not orthogonal to eigenspace of the eigenvalue λ_1, the solution $\boldsymbol{x}(t)$ of the system 27 as t goes to infinity converges to an eigenvector $\boldsymbol{v}_1$ corresponding to the largest eigenvalue of $\mathbf{A}$. It is further shown in [1] that replacing $\mathbf{A}$ with $-\mathbf{A}$ in 27 makes the solution converge instead to an eigenvector of the smallest eigenvalue λ_N belonging to $\mathbf{A}$, so long as the initial vector $\boldsymbol{x}(0)$ is not orthogonal to the eigenspace of that eigenvalue λ_N. Finally, they give the following recipe for obtaining the eigenvalue from the converged eigenvector:

$$\lambda = \frac{\boldsymbol{v}^T\mathbf{A}\boldsymbol{v}}{\boldsymbol{v}^T\boldsymbol{v}} \quad (28)$$

which follows immediately from the definition of eigenvectors and eigenvalues.

Thus, given a real and symmetric matrix, the largest and smallest eigenvalues along with their corresponding eigenvectors can easily obtained from solving 27 with an appropriate initial vector $\boldsymbol{x}(0)$. Now, in order to obtain the remaining eigenvalues and eigenvectors one could in principle apply a deflation technique, like Hotelling's or Wielandt's deflation [7] [8], to the matrix $\mathbf{A}$ and repeat the process for the deflated matrix $\mathbf{A}_D$. That is, however, beyond the scope of our present work; and furthermore, non-trivial numerical issues might be expected to arise in such repeated solutions of 27.

1. *Solution through the Forward-Euler Method*

The authors of [1] presented the equation 27 in the context of Recurrent Neural Networks (RNN), where it describes the dynamics of a certain class of such networks. RNNs are outside our current areas of expertise; however, to the best of our knowledge, the simplest RNN that obeys the dynamics prescribed by 27 should be completely equivalent to the following explicit integration method (Forward-Euler):

$$\boldsymbol{x}(t_i + \Delta t) = \boldsymbol{x}(t_i) + \frac{\mathrm{d}\boldsymbol{x}(t_i)}{\mathrm{d}t}\Delta t \quad (29)$$

where $\Delta t = t_{i+1} - t_i$ is a suitably small time-step, $\frac{d\boldsymbol{x}(t_i)}{dt}$ is equation 27 evaluated at $t = t_i$, and the iterations are started from some initial vector $\boldsymbol{x}(0)$.

This is, in any case, the simplest numerical integration scheme for solving equation 27, and is what we will be using to compare our neural network-based solutions with.

2. *Solution through Neural Networks*

The procedure for solving equation 27 with a neural network is very much the same as for the heat equation. First a trial solution satisfying the initial conditions is constructed, in this case we use:

$$\tilde{\boldsymbol{f}}(t) = \boldsymbol{x}_0 e^{-t} + (1-e^{-t})\boldsymbol{N}(t,P) \quad (30)$$

Then the cost-function to be optimized is defined, in this case as:

$$\mathcal{C}\left[\tilde{\boldsymbol{f}}(t)\right] = \frac{1}{M}\sum_{i}^{M}\left[\frac{\mathrm{d}\tilde{\boldsymbol{f}}_i(t)}{\mathrm{d}t} - \left(-\tilde{\boldsymbol{f}}_i(t) + \left(\tilde{\boldsymbol{f}}_i^T(t)\tilde{\boldsymbol{f}}_i(t)\mathbf{A} + \left(1 - \tilde{\boldsymbol{f}}_i^T(t)\mathbf{A}\tilde{\boldsymbol{f}}_i(t)\right)\mathbb{I}\right)\tilde{\boldsymbol{f}}_i(t)\right)\right]^2 \tag{31}$$

Where again, this cost function is valid for a single sample t, and for mini-batches one would sum the cost over all the sampled t and take the mean.

Finally the derivatives of the cost-function with respect to the network parameters are computed, and the network parameters are updated using Adam.

The only real change from the case of the heat equation is that the network now must produce M outputs for each input t, as opposed to one output for each pair x, t of inputs.

III. RESULTS & DISCUSSION

A. The Heat Equation

We solved the heat equation (Eqn. 1) in the domain $[x, y] \in [0, 1] \times [0, 1]$ using a Feed-Forward Neural network and an explicit Forward-Euler scheme. Both methods were then benchmarked against the analytical solution to the PDE which we derived in Section II B. All the results contained in this report may also be observed, and reproduced by re-running the Jupyter notebooks located in our Github repository [9], which also contains some additional supplementary results, like the evolution of the cost functions during training, which have not been included in this report.

The Neural Network was implemented using the PyTorch [10] machine learning library, chosen for its flexible interface enabling us to easily tailor our network to fit the problem at hand. When designing our model, we experimented with various network depths and layer sizes as well as activation functions, and found a reasonable balance between convergence rate and computational cost by using deep neural network with 4 hidden layers each consisting of 100 neurons. We also found good convergence with a learning rate of 0.002 and utilizing the standard set of parameters in the ADAM optimizer as proposed by its authors Kingma and Ba [5]. Lastly, we used an identity activation in the outer layer, as we may not in general infer about the range of a solution a priori.

The main results of our investigations are presented in Fig. 3, where we compared the output of two identical Neural networks using the ReLU activation in the hidden layers trained on $[11 \times 11]$[1] and $[100 \times 100]$ equispaced grids for trained for 10^4 and 10^2 epochs respectively, with mini-batches of size 16. The differing training time being justified by the fact that both networks will undergo approximately[2] the same number of total iterations. We then computed the absolute error as well as the mean absolute error (MAE) with respect to the analytic solution on a $[200 \times 200]$ equispaced grid to gauge the performance of both models. We further present identical networks also trained on $[11 \times 11]$ equispaced grids, but instead using the Sigmoid and tanh activation functions where the evaluation and error is computed in the same way as before. The results from this investigation are found in one of the Jupyter notebooks in our repository [9].

Lastly, we solved the Heat equation using the Finite-Difference scheme discussed in section II B 2 in the same domain for step-sizes $\Delta x = 1/10$ and $\Delta x = 1/100$ with Δt being chosen as to satisfy the stability criterion $\Delta t/\Delta x \leq 1/2$, both the solutions and the associated errors are presented in Fig 4.

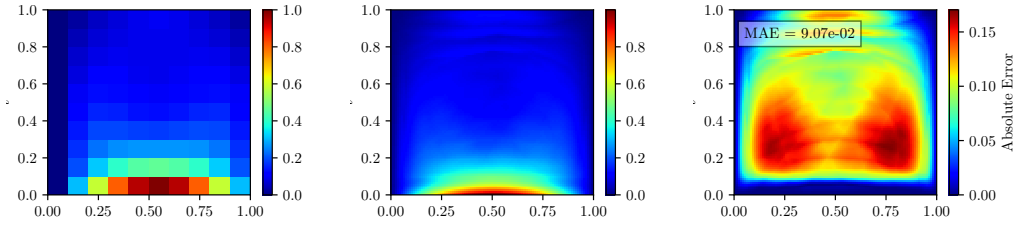
In our initial experimentation with different activation functions, we found that ReLU seemed the most consistent at capturing the qualitative results of the heat equation, as observed by monitoring the evolution of the cost-functions as the networks were being trained. The full results of these models are omitted from the report, but can be found in the fully rendered Jupyter notebooks in our repo. We thus used the ReLU activation as a basis for our subsequent analysis.

We observe in Fig. 3 that both models seems to yield essentially the same performance, where in particular the smaller data set is able to produce similar accuracy to the much larger dataset, whilst being two orders of magnitude smaller in size. However, when compared to the solutions by Finite-Difference, the Neural network performs rather poorly in terms of accuracy. Looking back to Fig. 3 we observe that the Neural network seems to struggle the most in capturing the transition between the "curved" part and the steady state in the region $t \approx 0.2$.

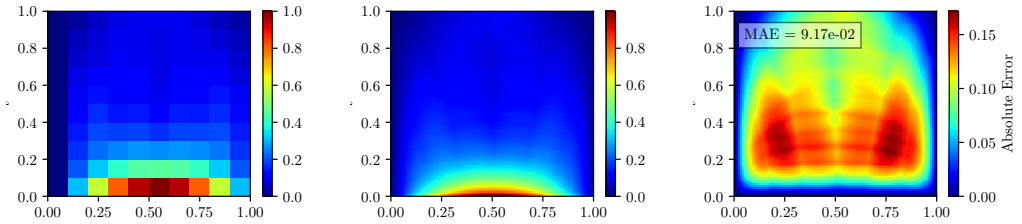
However, one thing to note in favor of the Neural network is that we are able to evaluate over the entirety of the domain $[0, 1] \times [0, 1]$, not just on the grid points we trained it on, but any other points in the domain. In contrast, for the solution provided by the finite difference method, we would have to utilize interpolation methods or redo the calculation with the desired points included in the computational grid. Significantly, we do not need the training points to obey the same constraints as the finite-difference method, that is; the stability criterion $\Delta t/\Delta x \leq 1/2$. That is, if we were to increase the resolution in x, the resolution in t must be increased quadratically, blowing up if one wishes high resolution in x for the finite-difference scheme. This is in stark

[1] 11 being chosen instead of 10 for more symmetric plots!

[2] Approximately because $11^2 \approx 100$

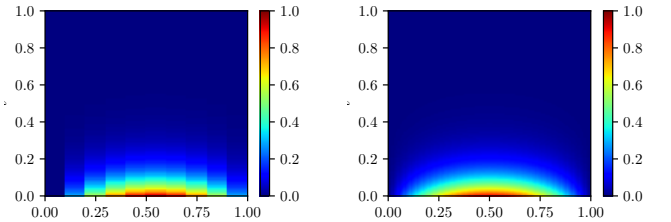


(a) Model I evaluated on 11x11 grid (b) Model I evaluated on 100x100 grid (c) Error of model I

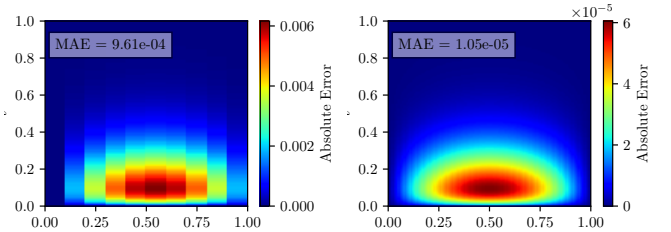


(d) Model II evaluated on 11x11 grid (e) Model II evaluated on 100x100 grid (f) Error of model II

FIG. 3: Feed-Forward Neural Networks with 4 hidden layers each consisting of 100 Neurons using the ReLU activation function trained on 11×11 (Model I) and 100×100 (Model II) equispaced points in the intervals $[x, t] \in [0, 1] \times [0, 1]$, where absolute error, and mean absolute error (MAE) are computed with respect to the analytic solution of the PDE on a grid of 200×200 equispaced points in the same interval.



(a) $\Delta x = 1/10, \ \Delta t = 1/200$ (b) $\Delta x = 1/100, \ \Delta t = 1/20000$



(c) $\Delta x = 1/10, \ \Delta t = 1/200$ (d) $\Delta x = 1/100, \ \Delta t = 1/20000$

FIG. 4: Solution of the heat equation solved by finite difference (Top row) as well as the Absolute error (Bottom row) with respect to the analytic solution for $\Delta x = 1/10$ and $\Delta x = 1/100$

contrast to the neural network, which can be trained on similar resolutions in both dimensions.

It should further be noted that we are not entirely confident that our network is optimized, and one may possibly with further time and tweaking obtain a neural network yielding a higher accuracy. However, given that our network has successfully reproduced the general features of the true solution, the method seems to work for solving PDEs.

B. The Eigen-problem

We wish to find the extrema eigenpairs of a real, symmetric matrix by solving a set of coupled, nonlinear differential equations presented by Yi *et al.* [1], Eqn. 27. We prepared a random matrix by sampling a $\mathbf{Q} = [6 \times 6]$ array of random numbers from the standard normal distribution $\mathcal{N}(0, 1)$ using a fixed seed ensuring that our results are reproducible. The matrix $\mathbf{Q}$ was then transformed to a real, symmetric matrix $\mathbf{A}$ by

$$\mathbf{A} = \mathbf{Q}^T + \mathbf{Q} \tag{32}$$

Following our work on the heat equation, we used a similar approach by solving the differential equations using a Neural Network as well as an explicit Forward-Euler scheme, where the latter was used to gauge how well the network is able to reproduce the dynamics of the system. The eigenpairs produced by the network were benchmarked against the eigh method in Numpy, which calls on numerically robust methods in LAPACK. We

TABLE I: Maximal eigenpairs predicted with the 30pt (N_{30}) and 300pt (N_{300}) models

	λ_{max}	x_1	x_2	x_3	x_4	x_5	x_6
N_{30}	3.689	0.535	-0.007	-0.281	0.696	0.223	-0.318
N_{300}	3.707	0.551	-0.019	-0.306	0.676	0.224	-0.308
Numpy	3.722	0.578	-0.021	-0.337	0.627	0.239	-0.318

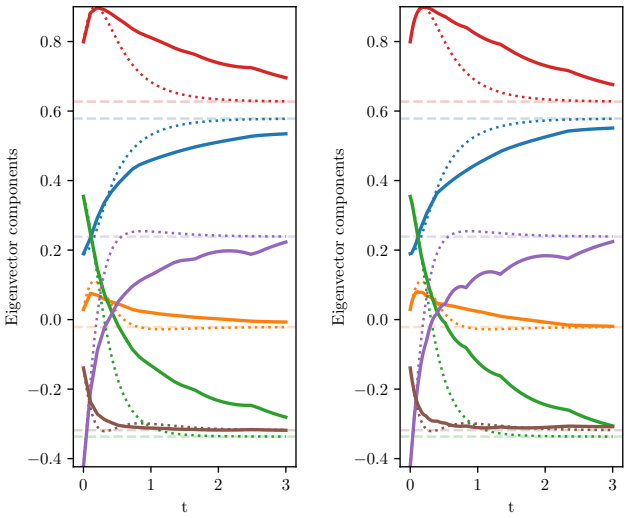
again wished to infer about the way in which the NN solution is affected by sample sizes, and thus trained two identical neural networks on the interval $t = [0, 3]$ using 30 and 300 data points. Where the appropriate interval was determined by observing whether the explicit scheme had converged. In both cases, we constructed neural networks with two hidden layers with 50 neurons per layer using the ReLU activation function. We again trained the network using the Adam optimization scheme with standard parameters and a learning rate of 0.002 as well as mini-batches of sizes 6 and 32 for the models trained on 30 and 300 data-points respectively.

The predicted eigenpairs are shown in Table I for both the models, along with the eigenpairs produced by Numpy. We further present the full dynamics of the differential equations in Fig. 5. From this, we also computed the inner products between the eigenvector produced by Numpy and the eigenvectors produced by our neural networks, which yielded $\langle \boldsymbol{x}_{np}, \boldsymbol{x}_{30} \rangle = 0.995$ and $\langle \boldsymbol{x}_{np}, \boldsymbol{x}_{300} \rangle = 0.998$ for the 30 and 300 point neural networks respectively, where we note that the eigenvectors are normalized.

As our first and most striking observation, we see in Fig. 5 that the network based solution to a large degree, but not fully, mimics the solution from the Forward-Euler scheme. Interestingly, for the present computational domains, the results seems to be largely independent of the number of points used in the training set given a comparable amount of training in both cases. Whilst we have not been able to fully reproduce the dynamics suggested by the Forward-Euler, the qualitative reproduction of their character by the models is such as to suggest that given more training and tweaking of the networks one may obtain a more accurate reproduction.

Secondly, we remark that the eigenpairs computed by both networks provide reasonable estimates of the ground truth. There is however a problem with relying solely on the solutions to the differential equations produced by the neural network to infer about the eigenpairs, namely due to the difficulty in determining whether the end of the time domain corresponds to a steady state solution. If one wishes to compute the eigenpairs, one needs to pick a time domain such that the true solutions have reached a steady state and converged to the components of the eigenvector, however, we do not in general have a method for picking a suitable time domain without prior knowledge of the dynamics. To illustrate this point; if we did not know the dynamics of the solutions a priori, the choice between time domains of $[0, 3]$ and for example $[0, 0.5]$ is completely arbitrary, where in the latter case one could never hope to produce the eigenvectors, as seen in Fig. 5. At the other extreme, experiments with overly large time-domains seemed to significantly degrade the convergence of the network-based solution to the Forward-Euler solution. Thus, we are not convinced that using neural networks to solve the differential equations (Eqn. 27) is a very useful way of computing eigenpairs.

We also computed the minimal eigenpairs in a completely identical manner by substituting $\mathbf{A}$ with $-\mathbf{A}$, given that the results are rather similar to the ones we have already looked at, we will not discuss there further and simply refer the reader to the Jupyter notebook for the results.



(a) 30 Points (b) 300 Points

FIG. 5: Dynamics of the eigenvector components of the largest eigenvalue of the matrix A modeled by Neural networks (Solid lines) trained on 30 and 300 data points as well as the same dynamics solved using a Forward Euler method (dotted lines) where Numpy eigenvector components is indicated by the semi-transparent dashed lines.

IV. CONCLUSION

We have successfully applied neural networks to the solution of both partial differential equations and systems of nonlinear ordinary differential equations. We find that our neural network-based models reasonably reproduce the true solutions; getting the right qualitative behavior, but achieving mediocre accuracy as compared to more standard methods. The main issue is that the training

in our current implementation appears to stall/plateau once a certain degree of accuracy has been reached.

For most domains we tried, the network-based solutions seemed largely independent of number of training points; so long as the total number of training iterations was similar, the results were similar. This indicates that a further increase in the number of training points will not help the accuracy, conversely fairly few training points are required to obtain a decent approximation, at least for the cases we have considered. This could be described as a blessed curse, one of the more obvious ways for improving the accuracy looks dishearteningly ineffective. On the other hand, as mentioned previously, a somewhat reasonable solution can be computed on a quite limited set of data-points, and, crucially, be directly evaluated on any point in the trained domain. That represents one of the clear advantages of neural network-based solutions as contrasted with traditional finite-difference schemes, which require either interpolation or recomputation on a finer or shifted grid.

An observation we made, but have not pursued in any level of detail, is that the extent of the domain we solve for seemed to have a sizable impact on the convergence of the solutions. Extending the training domain far into the steady-state region for the eigenpair-example lead to qualitatively worse reproduction of the dynamics. One important limitation of our investigation is that the points were either randomly sampled from uniform distribution, or taken as an equidistant grid. Conceivably it might have lead to better accuracy if more training points were clustered around rapidly varying parts of the solutions.

Other possible approaches for improving the accuracy of the network solutions include even further training, tweaking of the optimization-procedure/learning rate, testing different network architectures or even experimenting with different forms of the trial solutions (as they are not unique). Particularly, the fact that we see very small changes to the solution with further training indicate a need for more aggressive learning rates; or possibly some error in our implementation. Though the qualitative reproduction of the true solutions gives us confidence in the correctness of our results, it is not definite proof that the implementation is free of bugs.

For the considered example cases, the solutions based on neural networks were not at all competitive with those from standard methods. Especially for the eigenpair-problem it seems like using neural networks to solve the differential equation is a bad approach if the goal is the eigenpairs themselves. However, we want to reiterate that [1] do not suggest using neural networks to solve the differential equation 27, they propose using a RNN whose dynamics obey that equation for finding eigenpairs. That might very well be a promising approach for matrices whose dimensions make standard numerical linear-algebra-methods appear unappetizing.

While the cases we have considered do not look too favorable for the network-based solutions, we would like to point out that our experiments have been with fairly low-dimensional problems. The benefit of neural networks might first kick in at higher dimensions. To wit, it is quite easy to setup trial functions for arbitrarily large dimensions, while the corresponding finite-difference schemes become more and more burdensome to both implement and compute. Other numerical integration methods like finite-element methods (FEM) also face serious difficulties when the dimensionality of the problem increases; if the neural networks-based approaches can be made more accurate, we believe there to be a large niche of multi-dimensional problems where they may find themselves superior to traditional methods.

In summary, we have shown some potential for neural networks to solve both partial differential equations and systems of nonlinear differential equations. We find that accuracy is an issue, but recognize that there are many avenues to investigate in search of improving that issue. As the potential benefits of neural networks applied to high-dimensional problems are significant, we deem it a fruitful area for further study.

[1] Z. Yi, Y. Fu, and H. J. Tang, Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix, Computers & Mathematics with Applications **47**, 1155 (2004).

[2] N. Karlsen and T. Espedal Moe, Classification and regression, from linear and logistic regression to neural networks (2020).

[3] M. A. Nielsen, Neural networks and deep learning (2015).

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

[5] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization (2017), arXiv:1412.6980 [cs.LG].

[6] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations* (Cambridge University Press, 2009).

[7] Y. Saad, *Numerical Methods for Large Eigenvalue Problems - 2nd Edition* (2011).

[8] S. Roberts and M. Term, Computation of matrix eigenvalues and eigenvectors.

[9] N. Karlsen and T. Espedal Moe, Fys-stk4155 (2020).

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 8024–8035.