Solution of the diffusion equation and eigenvalue problems using Neural Networks

Gianmarco Puleo, David Svejda, Henrik Breitenstein
University of Oslo
(Dated: December 18, 2022)

We have trained neural networks (NN) to solve the diffusion equation in 1+1 dimensions, and compared the results to finite difference (FD) methods. The NNs were inferior in quality and computational cost. We also trained NNs to solve the differential equation set forth by Yi et al.[1] which finds the eigenvectors of a symmetric matrix. In this way, we were able to find four of the eigenvalues of a 6×6 symmetric matrix, with relative errors of order 10^{-5} or smaller. However, the method was ineffective at finding all of the eigenvectors. In both of the cases traditional FD methods for solving differential equations outperformed NN-based methods.

I. INTRODUCTION

Differential equations are used to describe a huge plethora of physics phenomena: some notable examples are Newton's second law of dynamics, Schrödinger equation in quantum mechanics and Einstein's field equations in general relativity. We could make a longer list, but as it is we deem it sufficient to render the importance of differential equations in both classical and modern physics. It also well-known that analytical, closed-form expressions of the solutions to differential equations are in general not easy to find, apart from few text-book cases which are presented in the majority of calculus courses. Therefore, it is important to develop algorithms that can find numerical solutions. To this end, finite differences and finite elements methods are the most-commonly employed ones. In this work, we present how deep learning is an alternative to such standard techniques. This approach consists in letting the solution of the differential equation be represented by an artificial neural network (NN). It was first proposed by Lagaris et al. in 1998[2], but variations on its theme were proposed, for example, in recent research related to many-body nuclear physics[3–5]. It is in fact known that the deep learning method proves to be comparable if not better than the FD in high dimension[6].

We are going to study two problems: the first is Fourier's heat equation, which serves as a toy-case where we can compare the NN's performance against a standard finite-difference algorithm. The second one is an ordinary differential equation (ODE), which allows to diagonalize a symmetric matrix and was presented by Yi et al. in 2004[1]. This paper is structured as follows: in section II we go through the details of these two equations and present specifically tailored neural network ansatzes and cost functions for their solution. In section III we compare the performance of the deep learning method against standard numerical solvers. In section IV we draw our conclusions. Our code makes use of the TensorFlow and NumPy libraries [7, 8] and is availiable at https://github.com/giammy00/FYS-STK4155-Project-3.git.

II. METHODS

II.1. The diffusion equation

We are going to study the diffusion equation:

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t} \tag{1}$$

where u is a function of time t and position x:

$$\begin{aligned} u : [0, L] \times \mathbb{R}^+ &\to \mathbb{R} \\ (x, t) &\mapsto u(x, t), \end{aligned} \tag{2}$$

where L is a real positive number. This equation could describe the time evolution of the temperature at every point of a rod of length L, which we choose to be equal to 1 for simplicity. For this reason this problem is often referred to as "the heat equation". We consider the following initial condition:

$$u(x, t = 0) = u_0(x) = \sin(\pi x) \tag{3}$$

and the boundary conditions:

$$\begin{aligned} u(0, t) &= 0 \quad \forall t, \\ u(1, t) &= 0 \quad \forall t. \end{aligned} \tag{4}$$

The latter can be interpeted thinking of our rod being in thermal contact with an environment at fixed zero temperature. The analytical solution of the equation is:

$$u(x, t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) e^{-n^2 \pi^2 t / L^2}, \tag{5}$$

where A_n are the Fourier coefficients of the function $u_0(x)$. In this case, we already expressed $u_0(x)$ as a trivial Fourier series, so we have simply

$$A_n = \begin{cases} 1 & \text{if } n = 1, \\ 0 & \text{else.} \end{cases}$$

So, the analytical solution in our case reduces to

$$u(x, t) = \sin(\pi x) e^{-\pi^2 t}, \tag{6}$$

which has a meaningful physical interpretation: as time goes by, the temperature u tends to become uniform all

along the rod, and in this case it tends to 0, which is the fixed temperature of the environment. Equation (6) is used in section III to assess the accuracy of the various numerical solutions we compute. Namely, we are going to compare a standard numerical finite difference method against a Neural Network which is trained to obey the same differential equation.

II.2. Finite difference solution

The simplest way of solving the heat equation numerically is the *forward difference* scheme. First we choose small space and time step sizes Δx and Δt. Denoting

$$\begin{aligned} x_n &= n\Delta x, \quad n \in \mathbb{N}, \\ t_m &= m\Delta t, \quad m \in \mathbb{N}, \end{aligned} \tag{7}$$

and $u(x_i, t_j) = u_{i,j}$, we can replace the derivatives with their corresponding discretized approximate versions, and get

$$\frac{u_{i+1,j}, -2u_{i,j} + u_{i-1,j}}{\Delta x^2} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{8}$$

Incidentally, we notice that the left hand side of the equation can be written as

$$\frac{(u_{i+1,j} - u_{i,j}) + (u_{i-1,j} - u_{i,j})}{\Delta x^2}, \tag{9}$$

which offers and interesting physical intuition: the second spatial derivative measures how much, on average, the temperature in the neighbourhood of x_i differs from the temperature at x_i. This quantity is proportional to the rate of change in time of the function u. Consequently, for example, if both the temperatures at x_{i+1} and x_{i-1} are higher than at x_i at a given instant t_i, then $u(x_i)$ will increase at the next time step. This is what we reasonably expect for the behaviour of a temperature function. Equation (8) yields an explicit scheme for computing the solution at every point in space x_i, at every time step, namely:

$$u_{i,j+1} = u_{i,j} + \frac{\Delta t}{\Delta x^2}\left(u_{i+1,j}, -2u_{i,j} + u_{i-1,j}\right). \tag{10}$$

It is known that, in order for the algorithm to converge, the stability condition

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2} \tag{11}$$

must be satisfied[9]. One of our goals is to compare this solution with the Neural Network trained as described in the next section.

II.3. Neural Network solution

We define a neural network with one hidden layer as follows:

$$\begin{aligned} N(x, t, \beta) &= \sum_i v_i \sigma(z_i), \\ z_j &= \sum_{k=0}^{1} w_{jk} x_k + b_j. \end{aligned}$$

Here, v_i are the weights of the output neuron, w_{jk} is the weight of input k in the j-th neuron of the hidden layer, b_j is the corresponding bias. The activation function σ is chosen to be the sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{12}$$

Also, we have re-labelled the inputs as $x_0 = t$ and $x_1 = x$, and we let β be a collective variable representing the whole set of parameters $\{v_i, b_j, w_{k\ell}\}$. We want to train the network so that

$$\tilde{u}(x, t) = (1 - t)\sin(\pi x) + x(1 - x)tN(x, t, \beta) \tag{13}$$

is a solution of the diffusion equation. Note that the initial and boundary conditions are always automatically satisfied by $\tilde{u}$, independently of the output of the neural network. With the same notation of the previous section, we have a grid of points (x_i, t_j) where we want to evaluate $\tilde{u}_{i,j}$. In order to rephrase this into a deep learning problem, it is natural to define a cost function in terms of the squared difference between the right and the left hand side of the equation (1). More precisely:

$$C(\beta) = \sum_{j=1}^{M}\sum_{i=1}^{N}\left(\frac{\partial^2 \tilde{u}(x_i, t_j)}{\partial x^2} - \frac{\partial \tilde{u}(x_i, t_j)}{\partial t}\right)^2. \tag{14}$$

Clearly, this cost function is non negative, and it is zero if and only if $\tilde{u}$ is equal to a solution to the diffusion equation when evaluated at all grid points (x_i, t_j). To implement backpropagation, it is necessary to compute its gradient with respect to β. To do so, we first write explicitly the derivatives both with respect to the position:

$$\begin{aligned} \frac{\partial \tilde{u}}{\partial x} &= \pi(1 - t)\cos(\pi x) + t\left[(1 - 2x)N + x(1 - x)\frac{\partial N}{\partial x}\right], \\ \frac{\partial^2 \tilde{u}}{\partial x^2} &= -\pi^2(1 - t)\sin(\pi x) \\ &\quad + t\left[2(1 - 2x)\frac{\partial N}{\partial x} - 2N + x(1 - x)\frac{\partial^2 N}{\partial x^2}\right], \end{aligned} \tag{15}$$

and with respect to time:

$$\frac{\partial \tilde{u}(x, t)}{\partial t} - \sin(\pi x) + x(1 - x)\left[N + t\frac{\partial N}{\partial t}\right]. \tag{16}$$

Notice that the derivatives of the network with respect to its inputs appear in these expressions. Their graident can be computed using automatic differentiation which is built into the TensorFlow library[1]. In appendix A we express the gradient of the cost function using repeated application of the chain and product rules for differentiation. Remarkably, this procedure is general, and can be applied to any differential equation. In the next sections, we outline how this is useful to solve a matrix diagonalization problem.

II.4. Eigenvalue equations as a system of ODEs

We summarize here the results obtained by Yi et al.[1]. The goal is to diagonalize a real symmetric matrix $A \in \mathbb{R}^{n\times n}$, i. e. to find its orthonormal eigenbasis and the eigenvalues. We consider the system of ordinary differential equations:

$$\frac{d\mathbf{x}}{dt} = -\mathbf{x}(t) + \mathbf{f}(\mathbf{x}(t)), \tag{17}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is defined as follows:

$$\mathbf{f}(\mathbf{x}) = \left[\mathbf{x}^\intercal \mathbf{x} A + (1 - \mathbf{x}^\intercal A \mathbf{x}) I\right] \mathbf{x}. \tag{18}$$

Despite the equation does not describe any physical process, in the following we often refer to the variable t as "time", for mere habit. The set of equilibrium points of the dynamical system defined by (17) is proven to be the union of all eigenspaces of the matrix A. Remember that in this case $\mathbf{x}_0$ is said to be an equilibrium point when it satisfies $-\mathbf{x}_0 + \mathbf{f}(\mathbf{x}_0) = 0$. In other words, if the initial condition of our problem happens to be an eigenvector $\mathbf{v}_\lambda$ of A, then we have at $t = 0$:

$$\frac{d\mathbf{x}}{dt} = 0,$$

so the time evolution of the system will be stationary: $\mathbf{x}(t) = \mathbf{v}_\lambda \ \forall t$. More mathematical details about stability theory can be found in [10].

Interestingly, it is possible to prove that for any non-zero initial condition $\mathbf{x}(0)$, the solution $\mathbf{x}(t)$ converges to one of such equilibrium points. This suggests that the eigenvectors of A can be computed by letting the system evolve for sufficiently long time. Before getting to the gritty-nitty details of how this can be done, we introduce the following notation: the eigenvalues of A are denoted as λ_j, and are ordered so that $\lambda_i \geq \lambda_j \iff i \leq j$. We also label V_{λ_i} the eigenspace corresponding to the eigenvalue λ_i. Now we summarize the most important theorems proved by the article:

- **Theorem A.** If $\mathbf{x}(0)$ is not orthogonal to V_{λ_1} then the solution converges to an eigenvector $\mathbf{v} \in V_{\lambda_1}$:

$$\lim_{t\to\infty} \mathbf{x}(t) = \mathbf{v} \in V_{\lambda_1}. \tag{19}$$

Note λ_1 is the largest eigenvalue of A. Thus, replacing A with $-A$ allows to find also the lowest eigenvalue.

- **Theorem B.** If $\mathbf{x}(0)$ is orthogonal to every eigenspace V_{λ_i}, for $i \in \{1, \dots, c\}$, $c < n$, then the solution $\mathbf{x}(t)$ converges, as $t \to \infty$, to an eigenvector which is also orthogonal to all these eigenspaces.

A recipe for systematically finding all the eigenvectors of A is therefore available:

1. We choose a random initial condition $\mathbf{x}(0)$. It is very unlikely that this is orthogonal to V_{λ_1}, and even if it were such, we could add a small perturbation to make it non-orthogonal to V_{λ_1}. We then let the system evolve for sufficiently long time and find the first eigenvector, which we call $\mathbf{v}_1$.
2. We choose a new initial condition $\mathbf{x}(0)$ orthogonal to the last eigenvector we found. This can be done, for example, by the Gram-Schmidt orthogonalization procedure, which we sketch in appendix B. The corresponding solution converges to a new eigenvector. We repeat this step $n - 1$ times, until we find a whole eigenbasis $\{\mathbf{v}_i\}$ of A.
3. The spectrum of A is readily computed as follows:

$$\lambda_i = \frac{\mathbf{v}_i^\intercal A \mathbf{v}_i}{\mathbf{v}_i^\intercal \mathbf{v}_i}, \text{ for all. } i \in \{1, 2, \dots, n\} \tag{20}$$

So, we have discussed how the convergence properties of a peculiar kind of differential equation allow to diagonalize a symmetric matrix. The simplest way of solving it is probably the forward Euler (FE) scheme, obtained by replacing the time derivative in the same manner as on the right hand side of equation (8). In this case, we obtain the update rule

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta t(-\mathbf{x}_i + \mathbf{f}(\mathbf{x}_i)). \tag{21}$$

An alternative approach is the use of a neural network to approximate the solution, similarly to what was explained in section II.3. We define our cost function as follows:

$$C(\beta) = \sum_{i=1}^{N} \left\| \frac{d\tilde{\mathbf{x}}}{dt}(t_i) + \tilde{\mathbf{x}}(t_i) - \mathbf{f}(\tilde{\mathbf{x}}(t_i)) \right\|^2, \tag{22}$$

and we let our solution $\tilde{\mathbf{x}}$ be represented by a network $\mathbf{N}(t)$ with one input t, n ouputs, and one hidden layer, in the following way:

$$\tilde{\mathbf{x}}(t) = \tilde{\mathbf{x}}_0(1 - t) + t\mathbf{N}(t). \tag{23}$$

The initial condition $\tilde{\mathbf{x}}(t = 0) = \tilde{\mathbf{x}}_0$ is automatically satisfied, and must be chosen depending of which eigenvector we want to converge to. The time derivative appearing in the cost function can be expressed as:

$$\frac{d\tilde{\mathbf{x}}}{dt} = -\tilde{\mathbf{x}}_0 + \mathbf{N}(t) + t\frac{d\mathbf{N}}{dt}. \tag{24}$$

[1] precisely, the function `tf.GradientTape()` does specifically what we want.

The computation of this derivative and of $\nabla C(\beta)$ is again achieved by automatic differentiation. Once this is done, gradient descent can be implemented, again using the functionalities of the TensorFlow library.

III. RESULTS AND DISCUSSION

III.1. Solution of the diffusion equation

We start by comparing the two methods for solving differential equations in the simple case of the diffusion equation before shifting our focus to the equation described in section II.4.

Finite-difference solution

In this section we go over our approximations of the diffusion equation solution using the finite-difference scheme II.2. We use time steps $\Delta t = 2 \cdot 10^{-4}$ and we study what happens at $t_1 = 0.1$, when the solution is close to the initial curve, and $t_2 = 0.3$ when it approaches the stationary linear state. We do this both for spatial resolution $\Delta x = 1/10$ and $\Delta x = 1/100$, and we get the plots depicted in figures 1 and 2. They can be reproduced with the Notebook 0 at our GitHub repository.

In both cases, we compute the relative error as:

$$\epsilon = \left| \frac{u_{\text{true}} - u_{\text{fd}}}{u_{\text{true}}} \right| \tag{25}$$

and we see that it stays constant with respect to x. We observe that the relative error increases with time. This makes sense since we know the numerical error propagate through each time-step taken by the algorithm. We can also see that the resolution along the position axis has a large impact on the accuracy of the approximate solutions. With about ten times the resolution we see that at both time points the error decreases by about a factor $\alpha \approx 10$ as well. Also, we observe that the ratio $\epsilon(t_2)/\epsilon(t_1)$ (which we could think as an "error gain" as we move through time) only gets slightly larger when moving to a finer spatial resolution. This suggests that the error gain from one time point to another is mostly affected by Δt, which was kept fixed. However, further investigations are needed to check whether this is the case.

Approximation using Neural Networks

We have implemented the methods outlined in section II.3, and applied them to a neural network with two dense hidden layers, the first with 50 neurons and the second with 20 neurons. All hidden layers use a sigmoid activation function, and the output layer uses no activation function. The network is trained using the Adam optimizer[7] with a constant learning rate of 0.005, on the domain $x \in [0, 1]$ and $t \in [0, 1]$ with a fixed regularly spaced set of 20 by 20 points. The training was done in 100 epochs consisting of 100 steps each consisting of 100 updates, so in total 1 million updates were done. The model is saved in our GitHub repository and the plots we are showing now can be reproduced in Notebook 4. Figure 3 shows the solution found by the network at two times $t_1 = 0.1$ and $t_2 = 0.3$, as well as the relative error, the full solution is animated in the 1M_animation gif in the GitHub repository. We see that the solution that the network has found is a bit asymmetrical, especially at later times, and that the relative error becomes quite big. The asymmetry in the solution is due to the random initialisation of the network, and apparently the loss associated with it is not that big. This could maybe be prevented by somehow ensuring that the network is initialised symmetrically, but it would probably be easier to average the output with the horizontally flipped output, and to do this in training as well. This would guarantee that the solution is symmetric, but these tricks would only work on symmetric problems.

The real cause of this problem is probably that the loss of an error is roughly proportional to the size of the function at that point[2], at least for the diffusion equation and other linear equations. And while the relative error becomes quite big in this case, the absolute error decreases as the model does get quite close to the final state in the end.

Training the neural network took approximately 75 minutes[3], which is a lot for such a small problem, although for this model the solution doesn't improve much beyond one epoch, which takes about a minute to train. Whatever changes cause the loss to drop by a factor of 100 in the next 99 epochs is not clear, but is doesn't seem to affect the solution produced by the network much.

Comparing finite-difference and neural net approximation

The finite-difference method approximates the one-dimensional diffusion equation more accurately and in a much shorter time than the neural net. The neural net has a lower relative error at $t = 0.1$, as seen in 3, and is low close to the endpoint $x = 1$ in general, but loses to FD towards $x = 0$. With the diffusion equation in one dimension, which results in a two dimensional problem, the neural net approximation falls short to the finite-difference method in general. This is to be expected because with so few dimensions it is easy to calculate the derivatives, while in higher dimension it is shown that neural networks catches up to finite-difference method.[6]

---

[2] technically the loss is more proportional to the size of the derivatives in the differential equation, but those are generally strongly correlated with the size of the function in linear problems

[3] using a HP Pavilion Laptop 15, running on the cpu

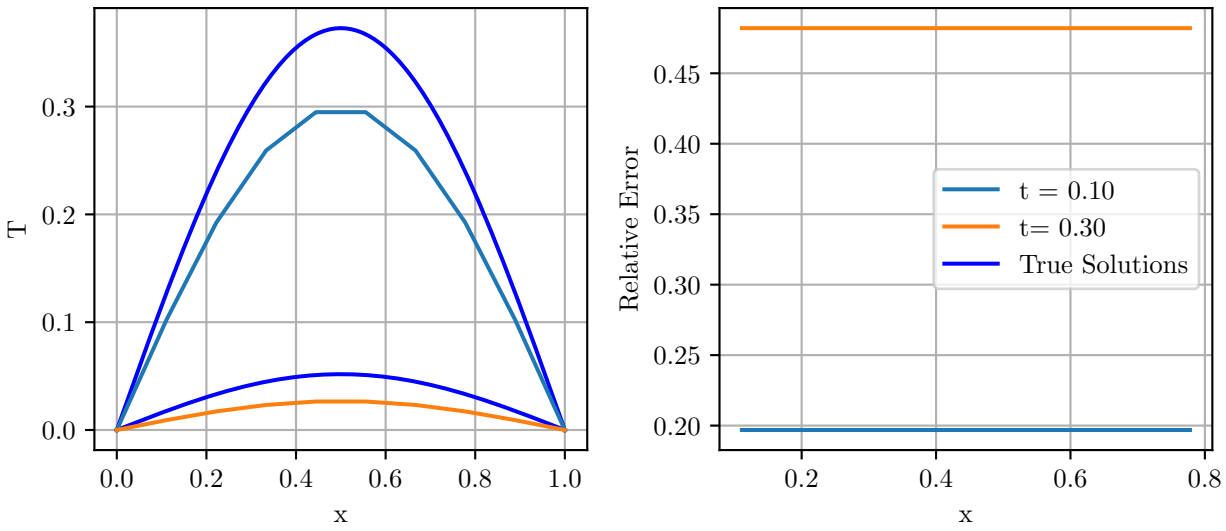

Figure 1: Study of the numerical solution of the diffusion equation with space resolution $\Delta x = 1/10$. On the left, we show the exact solution (blue) together with the finite difference approximation, at both time points $t_1 = 0.1$ and $t_2 = 0.3$. On the right we show the relative error as function of x. It was not evaluated at the endpoints, due to the fact that the true solution is zero there, therefore the relative error being meaningless.

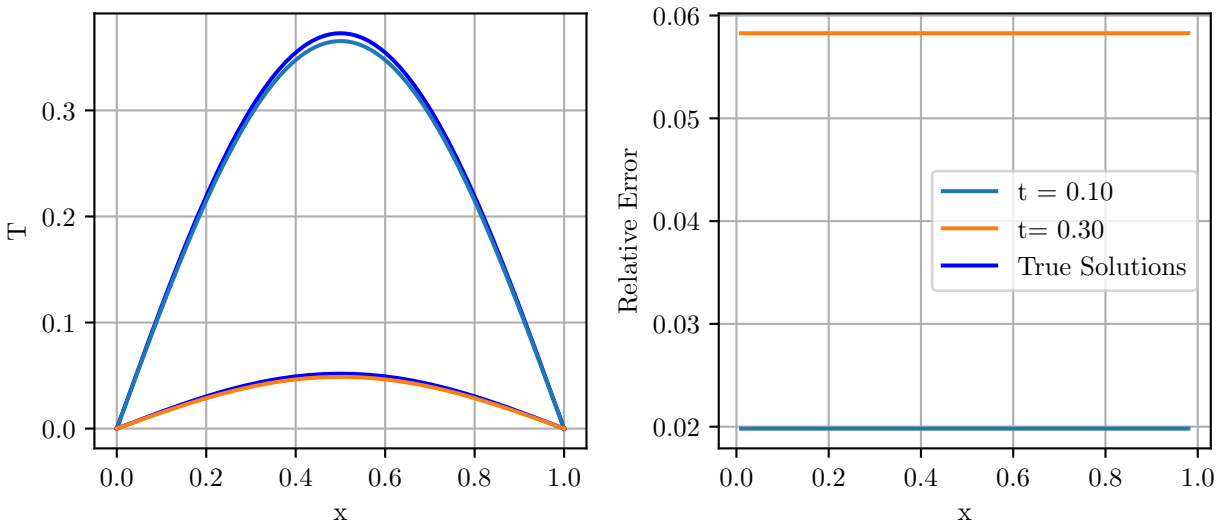

Figure 2: Analogue of figure 1, now with space resolution $\Delta x = 1/100$. On the left the exact solution (blue) is compared to the finite difference approximation at both the time points $t_1 = 0.1$ and $t_2 = 0.3$. On the right, we portray the relative error.

III.2. Diagonalizing a symmetric matrix

We present here a critical discussion of the diagonalization method presented in the article [1]. First of all it is worth mentioning that, despite such paper is about a neural network method to solve an eigenvalue problem via a differential equation, none of the references in its bibliography mentions the approach by Lagaris et al.[2], which we described here and that was presented during the lectures. Moreover, we note that regarding the ordinary differential equation (17), Yi et al. write

> "Clearly, this is a class of recurrent neural networks".

We feel that how an ordinary differential equation can be represented by a RNN is unclear and non-obvious to us. It's also worth pointing out that the article is very vague about how their numerical results are obtained: neither the training procedure nor any sort of cost function are specified, no detail is even provided about the architecture of the network. For all the aforementioned reasons, understanding and implementing what they have done is challenging, and it seems very unlikely

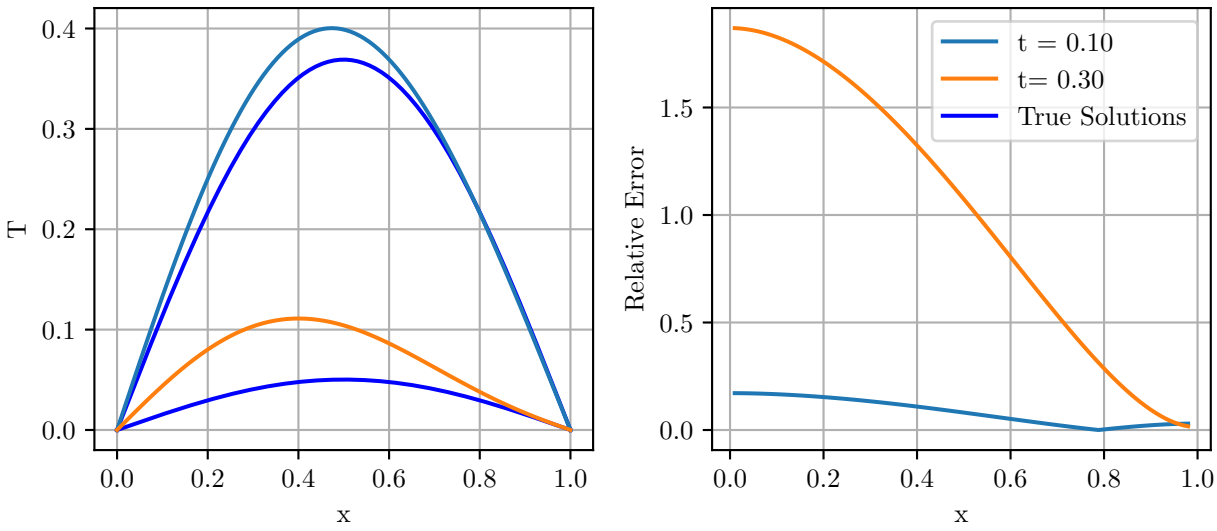

Figure 3: Neural network solution of the diffusion equation. One the left hand plot it is compared to the exact solution (blue) at both the time points $t_1 = 0.1$ and $t_2 = 0.3$. On the right we portray the relative error as function of x, where we have removed both endpoints for stability reasons.

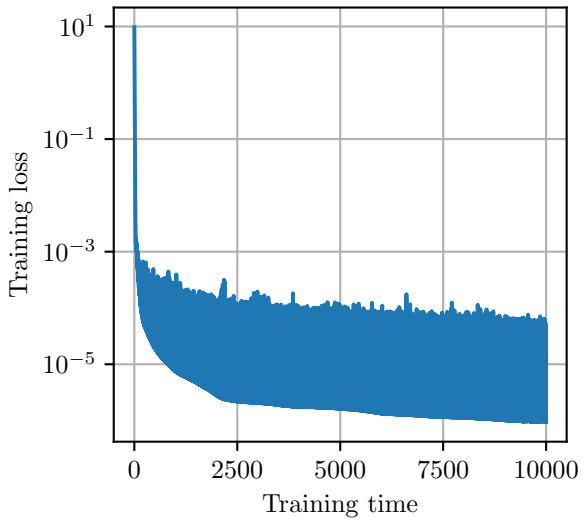

Figure 4: Training of a neural network to solve the diffusion equation, by minimization of the cost function defined in eq. (14), using the Adam optimizer. The plot depicts the training loss of the neural network, averaged per 100 updates of the weights, as function of the number of steps taken in the descent (100 updates are considered to be one step).

that what we are doing here should exactly reproduce what is shown in the article. In light of these observations, we now present our analysis, which is organized as follows: first we solve the differential equation (17) using the forward Euler method to find eigenvectors of a symmetric matrix. Afterwards, we try to solve the same problem using a neural network, trained as described in section II.4. The matrix which we aim to diagonalize is the following:

$$A = \begin{pmatrix} -0.112 & -0.399 & -0.259 & 0.107 & -0.473 & -0.479 \\ -0.399 & 1.297 & 0.359 & 0.428 & 0.678 & 0.719 \\ -0.259 & 0.359 & -0.855 & 0.526 & -0.632 & 0.945 \\ 0.107 & 0.428 & 0.526 & 0.144 & -1.579 & 0.175 \\ -0.473 & 0.678 & -0.632 & -1.579 & -0.769 & -0.515 \\ -0.479 & 0.719 & 0.945 & 0.175 & -0.515 & -0.015 \end{pmatrix}, \tag{26}$$

which was generated from a random normally distributed matrix Q as

$$A = \frac{1}{2}(Q + Q^{\intercal}).$$

Forward Euler diagonalization

What follows can be reproduced using Notebook 1 in our GitHub repository. We use the FE algorithm to compute 6 numerical solutions of equation (17), each of them has a different initial condition $\mathbf{x}_0^{(i)}$ and yields a different eigenvector $\mathbf{v}_i$. We denote by $\bar{\mathbf{x}}^{(i)}(t)$ the i-th numerical solution and let $\mathbf{x}^{(i)}(t)$ be the corresponding exact solution. We choose a number of $N = 2 \times 10^5$ evenly spaced time steps in the interval $t \in [0, t_f]$, where we set $t_f = 8$. As first initial condition, we let

$$\mathbf{x}_0^{(1)} = \begin{pmatrix} -0.996 \\ -0.442 \\ -0.726 \\ 0.422 \\ -0.488 \\ 0.322 \end{pmatrix}, \tag{27}$$

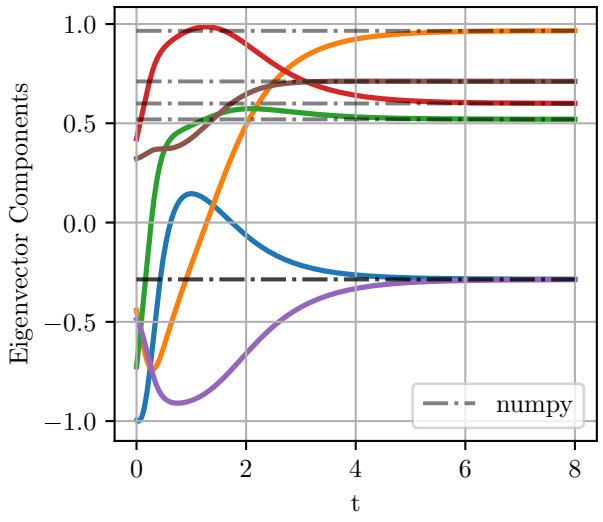

Figure 5: The components of the forward Euler solution of the ODE (17), with initial condition specified by eq. (27), are shown in coloured solid lines. As t grows, they get closer and closer to the eigenvector of A computed by NumPy, which is shown in dashed-dotted lines and corresponds to the largest eigevalue of A, namely $\lambda_1 \approx 2.2$.

where every component was sampled from a standard normal distribution (i. e. with mean 0 and variance 1). The time evolution of the components of $\bar{\mathbf{x}}^{(1)}(t)$ is shown in figure 5. We then proceed according to the algorithm described in section II.4, and find all eigenvectors and eigenvalues. If we had:

$$\lim_{t\to\infty} \mathbf{x}^{(i)}(t) = \bar{\mathbf{x}}^{(i)}(t_f), \tag{28}$$

it should be sufficient to compute the i-th eigenvector as:

$$\mathbf{v}_i = \mathbf{x}^{(i)}(t_f). \tag{29}$$

However, equation (28) is only true within some approximation, due to the numerical error introduced by the finite difference scheme and to the fact that we are not able to evaluate the solution at $t = \infty$, but only at finite times. For this reason the vector $\mathbf{x}^{(i)}(t_f)$, *as is*, is not exactly an eigenvector of A and therefore is not guaranteed to be orthogonal to the others that were previously found. The following slight correction seems therefore necessary when $i > 1$:

$$\mathbf{v}_i = \mathbf{x}^{(i)}(t_f) - \sum_{k=1}^{i-1} \frac{\mathbf{v}_k^\intercal \mathbf{x}^{(i)}(t_f)}{\|\mathbf{v}_k\|^2} \mathbf{v}_k. \tag{30}$$

Each term in the summation removes from $\mathbf{x}^{(i)}(t_f)$ its component along the direction specified by $\mathbf{v}_k$. This ensures that the set $V = \{\mathbf{v}_k\}_{k=1}^{i}$ is orthogonal for any i, and it can be used to construct an initial condition $\mathbf{x}^{(i+1)}(t)$ which is orthogonal to all elements of V. In this way we can use theorem B to obtain the full spectrum and eigenbasis of A. Our results can be compared with the output of the library function `numpy.linalg.eigh` in the following way. We compute the relative error on the eigenvalues as:

$$\epsilon_i = \left| \frac{\lambda_{i,\text{numerical}} - \lambda_{i,\text{numpy}}}{\lambda_{i,\text{numpy}}} \right|. \tag{31}$$

Also, we check whether the eigenvectors $\mathbf{v}_{i,\text{numerical}}$ and $\mathbf{v}_{i,\text{numpy}}$ are parallel, remembering that this is the case if and only if

$$\Lambda_i \equiv \frac{\langle \mathbf{v}_{i,\text{numerical}} | \mathbf{v}_{i,\text{numpy}} \rangle^2}{\|\mathbf{v}_{i,\text{numpy}}\|^2 \|\mathbf{v}_{i,\text{numerical}}\|^2} - 1 = 0. \tag{32}$$

The results of this analysis are summarized in table I. Both Λ and ϵ are smaller than 3×10^{-6} in all cases. We

λ_{numpy}	$\lambda_{\text{numerical}}$	ϵ	Λ
-2.307	-2.307	6.99×10^{-7}	2.11×10^{-6}
-1.542	-1.542	1.04×10^{-6}	2.11×10^{-6}
-0.540	-0.540	1.01×10^{-6}	7.58×10^{-7}
0.190	0.190	2.89×10^{-6}	7.54×10^{-7}
1.686	1.686	8.12×10^{-8}	2.66×10^{-7}
2.202	2.202	6.24×10^{-8}	2.66×10^{-7}

Table I: Comparison between NumPy and forward Euler diagonalization of the matrix A. The eigenvalues λ gotten in both ways are shown, together with the corresponding relative error and Λ. The latter is defined in equation (32) and it being close to zero means that the eigenvectors which we found are almost perfectly parallel to the ones computed by NumPy.

have therefore succeeded in diagonalizing the matrix A within great accuracy. We finally observe that the time it took to run this simulation on a standard laptop is around 30 seconds. This concludes our investigation about the finite difference solution of equation (17), and offers a sound foundation for the following. Namely, we now aim to train a neural network to reproduce this matrix diagonalization procedure.

Can a neural network diagonalize the matrix?

We let our neural network have 100 neurons in its hidden layer. We train it on a grid of 100 evenly spaced time steps in the range $[0, t_f]$, now with $t_f = 2$. We optimize the cost function (22), by non-stochastic gradient descent, with 10^5 epochs, using the Adam optimizer. For the sake of comparison with forward Euler, we set the same initial condition as specified by equation (27). The results we are now presenting are reported in Notebook 2. The behaviour of the network solution (23) after training is shown in figure 6. We clearly see that $\tilde{\mathbf{x}}$ does *not* reproduce the numerical solution (figure 5). Two main differences are apparent:

1. The network solution $\tilde{\mathbf{x}}(t)$ stabilizes around some stationary values immediately, i. e. a lot earlier than FE.

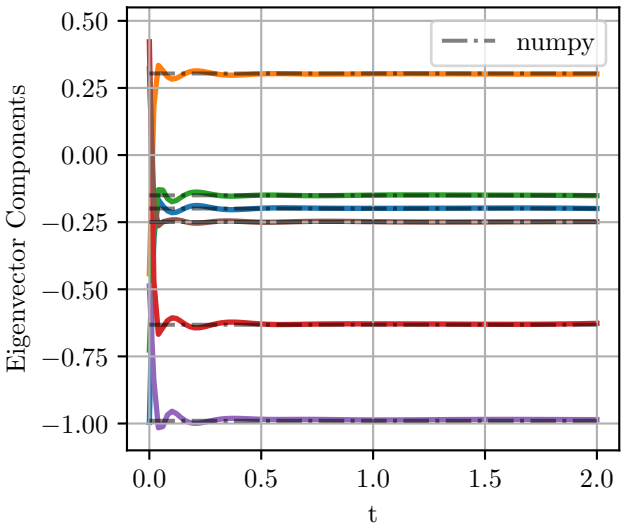

Figure 6: Neural network solution of the ordinary differential equation (17), evaluated as prescribed in eq. (23). It converges to the eigenvector of A with eigenvalue $\lambda \approx -2.3$. The corresponding components computed by the NumPy library are shown in dashed-dotted lines. Despite the initial conditions being the same as in the FE example shown in figure 5, convergence is reached at earlier values of t and we find a different eigenvector.

.

2. Such values do not match the asymptotic behaviour of the FE solution.

We investigate further about $\tilde{\mathbf{x}}(t_f)$ and find that it is an eigenvector of A, but it does not correspond to the expected eigenvalue. In fact, plugging $\mathbf{v}_i = \tilde{\mathbf{x}}(t_f)$ into equation (20) leads to the eigenvalue $\lambda = -2.307$ (truncated here to three decimal digits), which matches the lowest one reported in table I. We can compute the relative error ϵ and the corresponding value of Λ (note this quantity was defined in eq. 32):

$$\epsilon = 4.5 \times 10^{-6} \ , \quad \Lambda = 6.4 \times 10^{-6},$$

which are pleasingly close to zero, meaning that what we have is indeed an eigenvector. This suggests that the proposed network model is able to find eigenvectors, but it seems hard to train it to reproduce an accurate solution of the differential equation 17 with specified initial conditions. Consequently it is not possible to exploit theorems A and B of section II.4 to have control over which eigenvector will be represented by $\tilde{\mathbf{x}}(t_f)$ after training. It is therefore challenging to perform a systematic and efficient search of the eigenvectors using the algorithm offered by those theorems. Not being able to find the true solution suggests that something goes wrong during the training, but if this is the case, why do we find an (unexpected) eigenvector with such a good accuracy? We can check whether the overall cost function (equation 22) decreases as we train the model, by plotting it as function of the number of epochs elapsed. Also, we can plot the individual losses (i. e. the single terms appearing inside the summation in equation 22) at all time steps at the end of the training. These two plots are shown in figure 7. Clearly, the cost becomes smaller and smaller during the gradient descent, but we observe that after training the losses remain significantly high at the early stages of the evolution (small values of t). This can be interpreted as follows. First, recall that with initial condition $\mathbf{x}_0 = \mathbf{v}_i$, where $\mathbf{v}_i$ is *any* eigenvector of A, the solution of the ODE is stationary. For this reason, if the learned solution were $\tilde{\mathbf{x}}(t) = \mathbf{v}_i$ for all t, then it would have clearly cost zero. From figure 6, we can see that this the case for *almost all* but not all values of t, because the definition of $\tilde{\mathbf{x}}(t)$ has been built so as to satisfy the initial condition (which, in our case is not an eigenvector), regardless of the network output. So, the gradient descent has lead to something which resembles a solution of (17), except for the earliest time points. For this reason, we see relatively high losses at the beginning of the evolution depicted in figure 7, on the left hand side. We therefore conclude that we have fallen into a region which is close to a local minimum of the cost function, but it is not the global minimum which we aimed at.

We repeat that, because our network is not able to reproduce accurately the dynamics of the differential equation, we cannot take advantage of the convergence criteria described in section II.4. Therefore, we implement the following brute force approach. We train $\mathcal{N} = 30$ neural networks in the same way as we did before, but every time we set a new, random initial condition, drawn from a standard normal distribution. Also, we train each model for 5×10^5 epochs. This analysis can be also visualized and reproduced in Notebook 3. We see that the network does not always converge to the same eigenvector, and we cannot predict *a priori* which one will be found. Also we see sometimes a convergence to vectors with very small components, which can be explained observing that also the zero vector is a stationary solution of the differential equation, therefore having a low cost. In table II we report a summary of the (partial) spectrum and of the eigenvectors that we found by this deep learning approach. Note that running our search for about one hour has lead to only 4 of the orthogonal eigenvectors of the matrix A. Remind that it only took 30 seconds to find all of them using forward Euler. Also, the numpy library function `np.linalg.eigh` is always able to compute eigenvectors of such a small matrix in fractions of a second. It is then apparent how slow and ineffective a neural network is for diagonalizing symmetric matrices.

IV. CONCLUSION

In this project, we trained artificial neural networks to find the solution of two differential equations. In both cases, we found that the simplest finite difference algorithms outperform the deep learning approach, which on the other hand is found to be quite time consuming and unstable to compute. This is mainly due

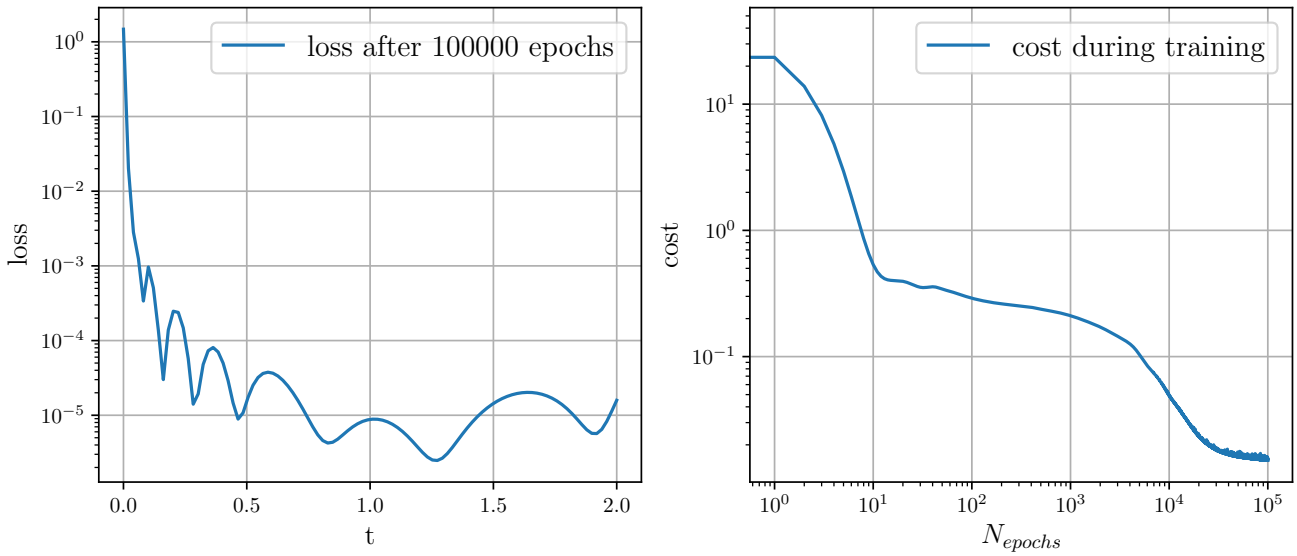

Figure 7: Study of the cost function (22), during and after the training of the neural network to find an eigenvector of a symmetric matrix. The individual losses at the end of training are shown on the left, and the overall cost is plotted against the number of epochs on the right. In other words, the y coordinate of the right-most point on the right-hand plot equals the average of the y coordinates of all points of the plot on the left-hand side. The corresponding learned solution is shown in figure 6, and we clearly see that it has a small cost (of order 10^{-2}), which is dominated by the losses at the first few time steps.

λ_{numpy}	λ_{NN}	ϵ	Λ
-2.307	-2.307	1.274×10^{-5}	1.904×10^{-5}
2.202	2.202	5.757×10^{-7}	9.693×10^{-7}
-0.540	-0.540	3.231×10^{-6}	2.274×10^{-6}
0.190	0.190	2.901×10^{-7}	3.070×10^{-8}

Table II: Comparison between Neural Network and Numpy's diagonalization methods of symmetric matrix A. This table shows the eigenvalues of the matrix A found with the NN, compared to their correspondents found by `numpy.linalg.eigh`, with ϵ being the relative error and Λ defined by equation (32), where $\mathbf{v}_{i,\text{numerical}}$ is now considered to be the one found by the NN. The fact that these quantities are very close to zero in all cases is a confirmation that what we found were indeed eigenvectors and eigenvalues of A.

to the high cost of the gradient computation, which is huge if compared to the one of the standard solvers. While it is possible that the use of more optimal network architectures, training methods and hyper-parameters could bring the neural networks in line with the finite difference methods, it seems very likely that the former will still have much higher computational cost. On top of that, improving the accuracy of finite difference methods is straightforward while for neural networks it is much less so.
We were not able to test how neural networks perform at solving differential equations in problems with higher dimensionality, so we are not able to compare our results to those of [6]. Also, we conclude that it is really inconvenient to try to solve the eigenvalue problem as described by Yi et al.[1], using the method proposed by Lagaris et al.[2] to train the network. All these things considered, we feel that we gained experience with training neural networks and understood the powerfulness of automatic differentiation tools provided by standard machine learning libraries. Moreover, we have learned a very interesting method of solving differential equations, which turned out to be inefficient in the problems we studied, but which could hopefully become useful in our future research.

A. Computing the gradient of the cost function

We approach here the computation of the gradient of the cost function which was defined in equation (14). The main reference for this section is [2]: we report here their results for the sake of completeness. Note that in our codes this procedure is taken care of automatically by the TensorFlow library, which employs automatic differentiation. We begin by calculating the derivative of the network with respect to one of its inputs x_h:

$$\frac{\partial N}{\partial x_h} = \sum_j v_j \sigma'(z_j) w_{jh}. \tag{A1}$$

Doing this twice leads to:

$$\frac{\partial^2 N}{\partial x_\ell \partial x_h} \sum_j v_j \sigma''(z_j) w_{jh} w_{j\ell}, \tag{A2}$$

We recognize a pattern, which allows us to express any derivative of the network:

$$\frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_1}} \frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}} \cdots \frac{\partial^{\lambda_n}}{\partial x_n^{\lambda_n}} = \sum_j v_j \sigma^\Lambda(z_j) P_j \equiv N_g \tag{A3}$$

where we have introduced the quantities

$$\Lambda = \sum_{i=1}^n \lambda_i, \quad P_j = \prod_{k=1}^n (w_{jk})^{\lambda_k}, \quad \frac{d^\Lambda \sigma(z)}{dz^\Lambda} = \sigma^\Lambda(z). \tag{A4}$$

Now we look back at equation (14), and let $F(x_i, t_i, \beta)$ be the quantity between round parentheses. Taking the gradient leads to:

$$\nabla_\beta C(\beta) = \sum_{j=1}^M \sum_{i=1}^N 2F(x_i, t_i, \beta) \nabla_\beta F(x_i, t_i, \beta). \tag{A5}$$

Plugging the expression (A3) into equations (15) and (16) allows to compute F, but we also need its gradient with respect to the network parameters β. To this end, we differentiate (A3) in the following way:

$$\begin{aligned} \frac{\partial N_g}{\partial v_i} &= \sigma^\Lambda(z_i) P_i, \\ \frac{\partial N_g}{\partial b_i} &= v_i \sigma^{\Lambda+1}(z_i) P_i. \\ \frac{\partial N_g}{\partial w_{ik}} &= v_i \sigma^{\Lambda+1}(z_i) x_i P_i + v_i \sigma^\Lambda(z_i) \lambda_k w_{ik}^{-1} P_i. \end{aligned} \tag{A6}$$

These expression provide a general way of computing the gradient of a cost function defined in terms of the neural network and any of its derivatives with respect to the inputs.

B. Gram-Schmidt orthogonalization

Here we include a brief summary of the Gram-Schmidt orthogonalization procedure, which we use to compute the new initial conditions for the diagonalization algorithm described in section II.4. Suppose we have available a set of orthogonal vectors

$$\{\mathbf{u}_i\}_{i=1}^n.$$

We want to find a new vector, which is orthogonal to all of them. To do so, we generate a random vector $\mathbf{v}_{n+1}$ and we construct a vector $\mathbf{u}_{n+1}$ with the desired properties in the following way:

$$\mathbf{u}_{n+1} = \mathbf{v}_{n+1} - \sum_{i=1}^n \frac{\mathbf{u}_i^\intercal \mathbf{v}_{n+1}}{\|\mathbf{u}_i\|^2} \mathbf{u}_i \tag{B1}$$

[1] Z. Yi, Y. Fu, and H. Tang, "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix," *Computers & Mathematics with Applications*, vol. 47, pp. 1155–1164, 04 2004.

[2] I. Lagaris, A. Likas, and D. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998. [Online]. Available: https://doi.org/10.1109%2F72.712178

[3] A. Gnech, C. Adams, N. Brawand, G. Carleo, A. Lovato, and N. Rocco, "Nuclei with up to $a = 6$ nucleons with artificial neural network wave functions," *Few-Body Systems*, vol. 63, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1007%2Fs00601-021-01706-0

[4] A. Lovato, C. Adams, G. Carleo, and N. Rocco, "Hidden-nucleons neural-network quantum states for the nuclear many-body problem," 2022. [Online]. Available: https://arxiv.org/abs/2206.10021

[5] M. Rigo, B. Hall, M. Hjorth-Jensen, A. Lovato, and F. Pederiva, "Solving the nuclear pairing model with neural network quantum states," 2022. [Online]. Available: https://arxiv.org/abs/2211.04614

[6] V. I. Avrutskiy, "Neural networks catching up with finite differences in solving partial differential equations in higher dimensions," *Neural Computing and Applications*, vol. 32, no. 17, pp. 13 425–13 440, Sep 2020. [Online]. Available: https://doi.org/10.1007/s00521-020-04743-8

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane,

J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[9] M. Hjorth-Jensen, *Computational Physics, lecture notes*, 2015. [Online]. Available: https://github.com/CompPhysics/ComputationalPhysics/raw/master/doc/Lectures/lectures2015.pdf

[10] V. Moretti, *Meccanica analitica. Meccanica classica, meccanica lagrangiana e hamiltoniana e teoria della stabilità*, ser. La matematica per il 3+2. Springer Verlag, 2020. [Online]. Available: https://books.google.no/books?id=B5XyDwAAQBAJ