CompSci Project 2: Solving Differential Equations with Neural Networks

Kosio Beshkov[*]
Department of Bioscience, University of Oslo, Norway

Mohamed Safy[†]
Department of Chemistry, University of Oslo, Norway

Tim Zimmermann[‡]
Institute of Theoretical Astrophysics, University of Oslo, Norway

(Dated: September 1, 2024)

This project showcases the applicability of neural networks as a solution method for both ordinary (ODE) and partial differential equations (PDE). We study their performance in comparison to established algorithms in two different settings.

One Dimensional Diffusion Equation: Based on the one dimensional diffusion equation, a parabolic, linear PDE, a detailed study between FTCS — a simplistic finite difference scheme — and physics-informed neural nets (PINN) is conducted. Special attention is given to the question of hyperparameter tuning and its impact on the predictive power of the network approach.

Eigenvalue Problems: We attempt to extract the largest eigenvalue of a real, symmetric matrix by adopting an ODE-based surrogate model for the analysis of a time-discrete recurrent neural network and feed it into a non-recurrent PINN. Implications of (i) the choice of hyperparameters (ii) the structure of the cost function and (iii) the form of the ODE and associated sampling of the time-domain are investigated.

I. INTRODUCTION

Point of departure is a theoretical glimpse into two archetypal fields of computational science: numerical linear algebra in the context of *finding the eigenspectrum*, more precisely the largest eigenvalue, of symmetric matrices and the *integration of partial differential equations* showcased on the one dimensional diffusion equation. After summarizing some elementary facts about both problems, we proceed by outlining a set of conventional, simplistic approaches to solving these tasks. More precisely, section I A 1 a introduces the *power iteration* algorithm to extract the dominant, i.e. largest in magnitude, eigenvalue and its associated eigenvector of a diagonalizable matrix. Section I B 1 discusses the forward-time-central-space (FTCS) finite difference scheme as viable discretization of the diffusion equation. Technicalities of both methods, their applicability, limitations as well as comments on extensions to production-grade algorithms will be made along the way. We then proceed to rephrase both problems as minimization tasks of a cost function that encodes the physical properties of a potentially adjoint dynamical system, [1]. This will lead to the idea of *physics informed neural networks* (PINN), [2], introduced in section I C 1.

A. Eigendecomposition of Symmetric Matrices

Let $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ be a real matrix with *a priori* no special properties besides being square. We then call $\boldsymbol{A}$ *diagonalizable* iff an invertible matrix $\boldsymbol{V} \in \mathbb{R}^{N\times N}$ exists such that:

$$\begin{aligned} &\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1} \\ \Leftrightarrow\quad &\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}\boldsymbol{\Lambda} = \left(\boldsymbol{v}_1 \dots \boldsymbol{v}_N\right) \mathbf{Diag}[\lambda_1 \dots \lambda_N]\,, \end{aligned} \tag{1}$$

where $\lambda_i \in \mathbb{C}\backslash\{0\}$ denotes the *eigenvalue* associated with the *eigenvector* $\boldsymbol{v}_i \in \mathbb{R}^N$. The above factorization is then called the *eigendecomposition* of $\boldsymbol{A}$ and its importance in any quantitative scientific field cannot be understated. For instance, the eigendecomposition of the Hamiltonian matrix of a quantum mechanical system gives access to the its fundamental states and their respective energies.

We note in passing that multiple eigenvectors can yield the same eigenvalue λ_i — a property called degeneracy. Moreover, while the set of eigenvectors may not be orthogonal in the general setting above, it is still assured that they form a complete basis of $\mathbb{R}^N$.

Now, various sufficient conditions exist that guarantee a square, real matrix $\boldsymbol{A}$ to admit an eigendecomposition, alongside additional implications for the properties of both $\boldsymbol{\Lambda}$ and $\boldsymbol{V}$. We confine ourselves to the case of *symmetric* matrices $\boldsymbol{A} = \boldsymbol{A}^\intercal$. In this case it is not only assured that eq. (1) exists, but also that (i) the set of all eigenvalues, called the *spectrum* of $\boldsymbol{A}$, is real and (ii) $V^{-1} = V^\intercal$, i.e. the eigenvectors constitute an *orthonormal* basis.

To fix our notation, we will order eigenvalues of a real, symmetric matrix in descending order, i.e. $\lambda_1 \geq \lambda_2 \geq$

[*] constb@ibv.uio.no
[†] m.e.a.safy@kjemi.uio.no
[‡] tim.zimmermann@astro.uio.no

$\ldots \geq \lambda_N$. The objective of our discussion may then be stated as follows: *Given an arbitrary, real symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$, extract its largest eigenvalue and eigenvector $\{\lambda_1, \boldsymbol{v}_1\}$.* In what follows, two different approaches, for solving this task will be explored in more detail.

1. *Numerical Linear Algebra*

Numerical Linear Algebra takes the problem at face value and offers a multitude of both direct and iterative methods to extract the spectrum of matrix $\boldsymbol{A}$. The range of possible methods is vast and beyond of the scope of this work. Here, we confine ourselves to a particularly simple method, known as *power iteration*, see section I A 1 a, and only point out more sophisticated methods in section I A 1 b. Note that we will not use the power iteration in practice as it imposes additional assumptions on matrix $\boldsymbol{A}$ beyond our problem specification. We will merely use it as a valuable tool in the stability analysis of the FTCS scheme for the diffusion equation in section I B 1.

a. Power Iteration In addition to the assumptions stated in section I A, let us assume that $|\lambda_1| > |\lambda_N|$, i.e. the largest eigenvalue also dominates the spectrum in magnitude.

Since $\{\boldsymbol{v}_i\}_{i=1\ldots N}$ forms an orthonormal basis, any vector $\boldsymbol{w} \in \mathbb{R}^N$ may be written as $\boldsymbol{w} = \sum_i \alpha_i \boldsymbol{v}_i$. k-fold multiplication of $\boldsymbol{w}$ with $\boldsymbol{A}$ then yields:

$$\begin{aligned} \boldsymbol{A}^k\boldsymbol{w} &= \sum_i \alpha_i \boldsymbol{A}^k \boldsymbol{v}_i = \sum_i \alpha_i (\boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^\intercal)^k \boldsymbol{v}_i = \sum_i \alpha_i \lambda_i^k \boldsymbol{v}_i \\ &= \alpha_1\lambda_1^k \left(\boldsymbol{v}_1 + \frac{\alpha_2}{\alpha_1}\left(\frac{\lambda_2}{\lambda_1}\right)^k \boldsymbol{v}_2 + \ldots + \frac{\alpha_N}{\alpha_1}\left(\frac{\lambda_N}{\lambda_1}\right)^k \boldsymbol{v}_N\right) \\ &\rightarrow \alpha_1 \lambda_1^k \boldsymbol{v}_1 \ , \end{aligned} \tag{2}$$

which follows from exploiting $\boldsymbol{V}\boldsymbol{V}^\intercal = \mathbb{1}$ and carrying out the remaining matrix-vector products.

Thus, successive, in theory infinite, multiplications of an arbitrary vector $\boldsymbol{w}$ with $\boldsymbol{A}$ result in a vector that is parallel to the eigenvector $\boldsymbol{v}_1$ associated with the dominant eigenvalue λ_1 of matrix $\boldsymbol{A}$.

Two additional tweaks are necessary to turn this observation into an operational algorithm: Firstly, depending on the magnitude of λ_i, an exponential shrinkage or blow-up of the iterated vector occurs. This is compensated by renormalizing it to unity after each iteration. Secondly, $\boldsymbol{w}$ must be chosen randomly so that a non-zero overlap $\boldsymbol{w} \cdot \boldsymbol{v}_1 \neq 0$ can be guaranteed.

We reiterate that the dominance condition, $|\lambda_1| > |\lambda_N|$, imposes an additional assumption on $\boldsymbol{A}$ beyond what we specified in section I A. Section I B 1 will exploit the power method as a theoretical argument to establish the stability criterion of the FTCS scheme.

b. Beyond Power Iteration Off-the-shelf *exact* methods for computing the eigendecomposition of a matrix typically proceed in two steps: (i) apply a series of similarity transformations to reduce the initial matrix to a simpler form and (ii) diagonalize the latter.

In case of real, symmetric matrices (i) is typically carried out by a *Housholder transformation*, during which a series of orthogonal projections turns matrix $\boldsymbol{A}$ into a *tridiagonal* matrix.

For step (ii), one typically invokes a QL-factorization which is guaranteed to exist for tridiagonal matrices. Other options are a recursive "divide-and-conquer" procedure with QL-decomposition at the end of the recursive hierarchy. We refer to [3] and references therein for more information.

It is worth noting that *there is no need* to reimplement these algorithms today. Reference implementation for all of these exact methods exist in form of the famous `LAPACK`-suite and hardware vendor/community driven projects provide sophisticated, highly optimized variants of these as well as convenient wrappers for `python` such as `numpy`. Thus, a simple call to `numpy.linalg.eigh` solves our problem entirely.

2. *ODE-based Techniques*

In the quest of studying the asymptotic system state of a time-discrete, recurrent neural network, [1] analyzed a time-continuous surrogate model that takes the form of the following nonlinear, first order, ordinary differential equation:

$$\dot{\boldsymbol{x}} = \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{f}_{\boldsymbol{A}}(\boldsymbol{x}) = \left[\boldsymbol{x}^\intercal\boldsymbol{x}\boldsymbol{A} - \boldsymbol{x}^\intercal\boldsymbol{A}\boldsymbol{x}\right]\boldsymbol{x} \quad \text{and} \quad \boldsymbol{A} = \boldsymbol{A}^\intercal \ . \tag{3}$$

We refer to [1] for the full-fledged analysis of eq. (3) including the rigorous proofs for the theorems below and instead summarize the key findings.

Theorem 1. *The set of equilibrium points $E = \{\boldsymbol{x} \in \mathbb{R}^N \mid \boldsymbol{f}_{\boldsymbol{A}}(\boldsymbol{x}) = 0\}$ are the union of all eigenspaces $V_{\lambda_i} = \{\boldsymbol{x} \in \mathbb{R}^N \mid \boldsymbol{A}\boldsymbol{x} = \lambda_i\boldsymbol{x}\}$, i.e.:*

$$E = \cup_{i=1}^N V_{\lambda_i} \ .$$

Put differently, an equilibrated state for eq. (3) is also an eigenvector of $\boldsymbol{A}$.

So far it is unclear if (i) solutions to eq. (3) attain an equilibrium naturally, (ii) equilibration is sensitive to the initial condition and (iii) which eigenvector gets selected as dynamical attractor. The following theorem clarifies all these points.

Theorem 2. *Given any nonzero initial condition vector $\boldsymbol{x}(0) \in \mathbb{R}^N$, if $\boldsymbol{x}(0)$ is nonorthogonal to V_{λ_1}, then the solution to the initial value problem set by eq. (3) and $\boldsymbol{x}(0)$ converges to a vector in V_{λ_1}.*

In other words, random initial conditions are expected to converge to $\boldsymbol{v}_1$, the eigenvector corresponding to the largest eigenvalue λ_1

To exemplify both statements, the reader is referred to Figure 1. In it, we depict the vector field $\boldsymbol{f_A}(\boldsymbol{x})$ a real, symmetric matrix $\boldsymbol{A}$. For visualization purposes, the dimensionality was set to $N = 2$.

Notice how the vector field vanishes where the orthogonal eigenspaces $V_{1,2} = \mathrm{span}(\mathrm{v}_{1,2})$ are located. This is precisely the statement of Theorem 1.

Theorem 1 becomes plausible once we numerically integrate eq. (3) for a small number of random initial conditions. Clearly, as long as the initial conditions are not exactly located on the orange ray, i.e. the orthogonal subspace V_{λ_2}, trajectories will converge to a point in V_{λ_1}.

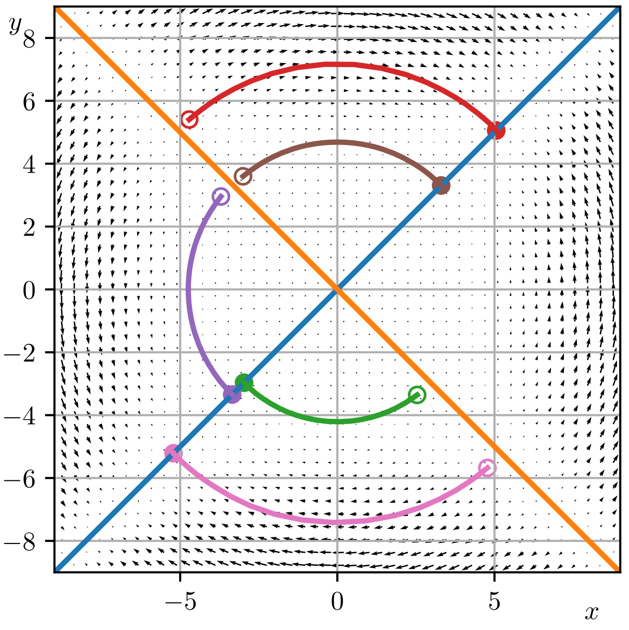

FIG. 1. Vector field $\boldsymbol{f_A}(\boldsymbol{x})$ for a real, symmetric matrix $\boldsymbol{A}$ alongside the two orthogonal eigenvector rays (orange and blue) and various integral curves of eq. (3) starting from random initial conditions (open circles). Note that all curves converge to a point on the eigenvector ray that corresponds to the largest eigenvalue (blue) The plot was generated by integrating eq. (3) with `scipy.integrate.solve_ivp`'s Runge-Kutta integrator.

Although the ODE of eq. (3) was initially meant as a analytically tractable tool to study the asymptotic system state of a time-discrete, recurrent neural network, there is *a priori* nothing wrong with using eq. (3) as a starting point instead and then focus on efficient methods to integrate it. In this sense, Figure 1 can be understood as a proof-of-concept employing classical integration techniques. In the light of section I C 1, we ask whether a physics-informed neural network may be an alternative to these canonical integration techniques and refer to section II B for a thorough investigation of this.

B. The 1D Diffusion Equation

Conservation laws of extensive properties, such as mass or energy, are of paramount importance in physics and encoding them into the dynamics of their intrinsic counterparts, like mass or energy density, sets the equation of motion for the latter.

Consider the extensive property U enclosed by a fixed control volume V and assume it is conserved as a function of time. Put differently, the only way the amount of U stored inside $V = [0, 1]$ is allowed to change is through an influx j_U of U through the volume boundaries $\partial V = \{0, 1\}$. Other then that no sinks or sources exist. Assuming we settle for $d = 1$ spatial dimensions, one may encode this argument mathematically in the following form:

$$\frac{\mathrm{d}U}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\int_V \mathrm{d}x u(x,t) = \int_V \mathrm{d}x\partial_t u(x,t)$$
$$\overset{!}{=} -j_U(x,t)\Big|_0^1 = -\int_V \mathrm{d}x\partial_x j_U(x,t) \;.$$

We thus conclude:

$$\partial_t u + \partial_x j_U = 0 \;. \tag{4}$$

The exact form of the flux associated with quantity U depends on the properties of the underlying transport mechanism. For instance, if there is a concentration gradient of the intrinsic (per volume) quantity $\partial_x u$, then the system tries to redistribute quantity U such that the gradient vanishes. This implies $j_U(x,t) = -\partial_x u$. We thus arrive at the one dimensional diffusion equation:

$$\partial_t u - \partial_x^2 u = 0 \;, \quad x \in (0,1) \;. \tag{5}$$

Eq. (5) is parabolic and therefore needs to by augmented with both initial conditions (IC), $u(x,0) = u_0(x)$, and boundary conditions (BC) $f\left(u, \partial_x u\right)\big|_{\partial V} = 0$. In what follows, we assume Dirichlet boundary conditions, i.e. $u(0,t) = u(1,t) = 0$.

Again, a multitude of methods exist in the literature to solve this initial boundary value problem either analytically or numerically. We postpone the derivation of an explicit, analytical solution of eq. (5) to section II A when we specify concrete initial conditions and focus in the following on the *forward-time-central-space* method to integrate eq. (5) in an IC-agnostic way.

1. Forward-Time-Central-Space Discretization

FTCS is a simplistic finite difference discretization method in which differential operators get replaced by their finite difference counterparts at the expense of picking up a truncation error. FTCS earns its name from the specific choice for these finite difference approximations.

For notational simplicity, let us introduce an equidistant spatio-temporal grid defined by the space- and time increments $\{\Delta x, \Delta t\}$ so that we may write

$u_n^j \equiv u(x_n, t_j) = u(n\Delta x, j\Delta t)$. FTCS's difference operators then take the form:

$$D_{\Delta t} u_n^j \equiv \frac{1}{\Delta t}\left(u_n^{j+1} - u_n^j\right) \ , \tag{6}$$

and:

$$D^2_{\Delta x} u_n^j \equiv \frac{1}{\Delta x^2}\left(u_{n-1}^j - 2u_n^j + u_{n+1}^j\right) \ . \tag{7}$$

Application of multiple Taylor expansions then reveal a difference equation that is second order accurate in space and first order accurate in time:

$$0 = D_{\Delta t} u_n^j - D^2_{\Delta x} u_n^j = \partial_t u - \partial_x^2 u + \mathcal{O}(\Delta x^2, \Delta t) \ . \tag{8}$$

It will proove instructive to recast the finite difference part of eq. (8) into vector-matrix form. For this, we define $\boldsymbol{u}^j = \left(u_1^j \dots u_{N-2}^j\right)^\intercal$ as *interior* solution vector at time $t_j = j\Delta t$. Note that $\boldsymbol{u}^n$ *excludes* the trivial Dirichlet boundary points $u_0^j = u_{N-1} = 0$.

Solving eq. (8) for $\boldsymbol{u}^{j+1}$ then yields:

$$\begin{aligned} \boldsymbol{u}^{j+1} &= \boldsymbol{M}_{\text{FTCS}}(N-2)\boldsymbol{u}^j \\ &= \begin{pmatrix} 1-2\gamma & \gamma & 0 & \dots & 0 \\ \gamma & 1-2\gamma & \gamma & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \gamma & 1-2\gamma & \gamma \\ 0 & \dots & 0 & \gamma & 1-2\gamma \end{pmatrix} \boldsymbol{u}^j \ , \end{aligned} \tag{9}$$

where $\gamma = \frac{\Delta t}{2\Delta x^2}$. This is the *one-step* update rule of the FTCS scheme. To propagate the initial state vector $\boldsymbol{u}_0$ to time $t_j = j\Delta t$, we simply apply eq. (9) iteratively:

$$\boldsymbol{u}^j = \boldsymbol{M}^j_{\text{FTCS}}(N-2)\boldsymbol{u}^0 \ . \tag{10}$$

To forgoing discussion might suggest that the spatial increments of both space and time are two independent parameters of the discretization. This is not true. We can understand this by analyzing eq. (10) more closely. Realize that the iteration matrix $\boldsymbol{M}_{\text{FTCS}}(N-2)$ is real and symmetric. Hence, eq. (10) is structurally equivalent to the iteration equation of the power method in eq. (9). We must therefore conclude that after a sufficiently large number of time steps j the numerical solution proposed by the FTCS scheme transforms into a scaled version of the eigenvector $\boldsymbol{v}_1$ corresponding to the dominant (largest in magnitude) eigenvalue of $\boldsymbol{M}_{\text{FTCS}}(N-2)$, again denoted with λ_1:

$$\boldsymbol{u}^j \xrightarrow{j\to\infty} (\boldsymbol{u}_0 \cdot \boldsymbol{v}_1)\lambda_1^j \boldsymbol{v}_1 \ . \tag{11}$$

Clearly, a physical constraint we deduce from this is $\lambda_1 \leq 1$, as otherwise an unphysical blow-up of the solution occurs. Afterall, it is a diffusive problem we are modelling.

To check if this stability constraint is satisfied, we seek for the dominant eigenvalue λ_1 of the iteration matrix $\boldsymbol{M}_{\text{FTCS}}(N-2)$ and a direct application of the power iteration, c.f. section I A 1 a, would gives us access to it. However, it does not provide us with any insights on how this eigenvalue is set by our discretization parameters.

Fortunately, $\boldsymbol{M}_{\text{FTCS}}(N-2)$ is not just real and symmetric, but also *tridiagonal*, i.e. only the first sub-diagonal, main diagonal, and first super-diagonal are non-vanishing, and *toeplitz*, i.e. elements on the same diagonal are identical. For this highly specialized subclass of matrices an analytical expression for the eigenspectrum exists:

$$\begin{aligned} &\text{Sp}\left(\boldsymbol{M}_{\text{FTCS}}(N)\right) \\ &\quad = \left\{1 - 2\gamma + 2\gamma\cos\left(\frac{n}{N+1}\right) \mid n = 1\dots N\right\} \end{aligned} \tag{12}$$

Two remarks are in order:

Firstly, note that the spectrum in eq. (12) is strictly positive ($\boldsymbol{M}_{\text{FTCS}}(N)$ is thus a positive definite matrix) and it is for this reason that the largest eigenvalue λ_1 is also the dominant one.

Secondly, for the largest eigenvalue to satisfy $\lambda_1 \leq 1$, we must have:

$$2\gamma \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{1}{2}\Delta x^2 \ . \tag{13}$$

This is FTCS' stability criterion, often referred to as Courant–Friedrichs–Lewy (CFL) condition. With this, a fully functioning, stable, implementation of FTCS may look like this:

```python
def ftcs_diffusion_dirichlet(u0, T, dx):
    dt = 0.5 * dx**2
    # Number of temporal points
    N = int(1/(dt)) + 1
    t = np.linspace(0, 1, N)

    # Number of spatial points
    M = int(1./dx) + 1
    # Full spatial domain
    x = np.linspace(0,1,M)

    #Setup iteration matrix
    A = np.zeros((M - 2,M - 2))
    i,j = np.indices(A.shape)
    A[i == j] = 1 - 2 * dt/dx**2
    A[i == j-1] = dt/dx**2
    A[i == j+1] = dt/dx**2

    u = np.empty((N,M))
    # Initial condition
    u[0,1:M-1] = u0(x[1:M-1])
    # Dirichlet boundary
    u[:,0] = 0
    u[:,M-1] = 0

    for n in range(1, N):
        u[n,1:M-1] = A @ u[n-1,1:M-1]

    return t, x, u
```

We close by mentioning that the quadratic scaling of eq. (13) makes the application of FTCS challenging for highly resolved domains. Implicit methods can relax the stability requirements on $\{\Delta x, \Delta t\}$ at the cost of increased computational complexity per time step, usually in the form of a matrix inversion of the iteration matrix. For the sake of brevity, these methods will not be discussed.

C. Neural Network Based Methods for Solving PDEs

Recently, the application of neural networks, see Appendix for a brief review, for finding approximate solutions to differential equations has received significant attention. This use case is particularly interesting for physics applications where in many cases the differential equation is the only information we have on the problem and generating training data may be prohibitively expensive so that canonical supervised learning approaches for regression are unfeasible. Multiple techniques exist for how to approximate the solutions of dynamical systems by means of a neural net approach. Here, we focus on *physics-informed neural networks* (PINNs), [2].

1. Physics-informed Neural Networks

Consider the abstract, potentially nonlinear, initial boundary value problem (IBVP):

$$\begin{aligned} 0 &= \partial_t u(x,t) + \mathcal{N}[u](x,t) \ , && x \in \Omega \ , t > 0 && (14)\\ u(x,0) &= u_0(x) \ , && x \in \Omega && (15)\\ 0 &= b(u, \partial_x u, t) \ , && x \in \partial\Omega \ , && (16) \end{aligned}$$

with $\mathcal{N}[u]$ as a potentially nonlinear operator possibly involving higher order derivatives of u in both space and time. For instance, the diffusion equation (5) employs $\mathcal{N}[u] = -\partial_x^2 u$.

Backed up by the *universal approximation theorem*, see Theorem 3, PINNs propose the output of a neural network as approximation to the true solution of an ODE/PDE and space-time points $(x,t) \in \Omega \times \mathbb{R}^+$ as network input. The IBVP is then rephrased as a minimization task by encoding the equation of motion, the initial conditions and potential boundary constraints into the cost function.

Realize that if we accept a certain neural network "discretization" $u(x,t) = N(x,t;\boldsymbol{\Theta})$ with $\boldsymbol{\Theta}$ as hyperparameters, to be a sufficiently good approximation of the true solution, $u^*(x,t)$, then it is possible to compute the space-time derivatives within this discretization *exactly.* In practice, this involves taking the derivatives of the network *with respect to the network inputs* (x,t) to form the cost function:

$$\begin{aligned} \mathcal{C} = &\frac{1}{2N_{\mathcal{T},\Omega}} \sum_{i=1}^{N_{\mathcal{T},\Omega}} \{\partial_t N(x_n,t_n) + \mathcal{N}\left[N(x_n,t_n)\right]\}^2 \\ &+ \frac{1}{2N_{\mathcal{T},\mathrm{IC}}} \sum_{i=1}^{N_{\mathcal{T},\mathrm{IC}}} \{N(x_n,0) - u_0(x_n)\}^2 \\ &+ \frac{1}{2N_{\mathcal{T},\mathrm{BC}}} \sum_{i=1}^{N_{\mathcal{T},\mathrm{BC}}} \{b\left(N(x_n,t_n), \partial_x N(x_n,t_n), t_n\right)\}^2 \ , \end{aligned} \qquad (17)$$

with training data:

$$\begin{aligned} X_{\mathcal{T},\Omega} &= \{(x,t) \in \Omega \times [0,T] \mid (x,t) \sim U(\Omega)U([0,T])\} \ , \\ X_{\mathcal{T},\mathrm{IC}} &= \{(x,0) \in \Omega \times [0,T] \mid (x,t) \sim U(\Omega)\} \ , \\ X_{\mathcal{T},\mathrm{BC}} &= \{(x,t) \in \Omega \times [0,T] \mid (x,t) \sim U(\partial\Omega)U([0,T])\} \ , \end{aligned} \qquad (18)$$

typically uniformly sampled from the respective restriction the domain $\Omega \times \mathbb{R}^+$.

From here on, standard procedure applies, i.e. one intends to minimize the cost function by means of a (stochastic) gradient descent, or higher order method, in the network parameters $\boldsymbol{\Theta}$.

In section II, we apply a feedforward net-based PINN to both the ODE-formulation of eq. (3) for finding the largest eigenvalue of a symmetric matrix and the diffusion equation in (5).

We close by drawing attention to an important intricacy. Although PINNs do not intend to constrain the architecture of the underlying neural net, the differential equation in question does. More precisely, it is the order of the highest derivative in either space or time that fixes the set of admissible activation functions: *A n^{th}-order equation requires activation functions that have a non-vanishing derivatives at least up to order n.* [1] If this is not satisfied, the PDE/ODE contribution in eq. (17) will not model the PDE. A direct consequence of this observation is that the popular `ReLU` activation of eq. (A.8) is not applicable for the diffusion problem.

D. Implementational Details

The code developed for this work can be found at the following -repo. Its backbone forms the `PiNN` class, a wrapper around a `PyTorch`-based feedforward neural net, [4], parametrized on (i) the number of hidden layers, (ii) the nodes per layer and (iii) the activation function. We tweak these model parameters, as well as additional hyperparameters such as (i) the optimizer, (ii) learning rate, (iii) number of minibatches, via `RayTune`, an industry-grade,

[1] The authors wasted two days of their lives figuring this out

distributed `python` framework for hyperparameter-tuning, see [5]. Although generic in nature, the code should be rather self-explanatory, and the authors don't see any added value in elaborating on its internal workings. For its usage, we refer to the `jupyter` notebooks for the diffusion and eigenvalue problem.

One aspect worth mentioning is that to compute the derivatives of the network with respect to its input rather than its parameters, we can employ `PyTorch`'s autodifferentiation capabilities. The differential operator ∂_x acting on a function $y\colon \mathbb{R} \to \mathbb{R}^N, N \geq 1$ then takes the form:

```
def derivative(y, x):
    dim = y.shape[-1]
    dydx = torch.zeros_like(y, dtype=y.dtype)
    for i in range(dim):
        yi = y[:, i].sum()
        dydx[:, i] = torch.autograd.grad(
            yi, x, create_graph=True,
                allow_unused=True
        )[0].squeeze()
    return dydx
```

II. RESULTS

A. The 1D Diffusion Equation

We intend to study the following IBVP:

$$0 = \partial_t u - \partial_x^2 u \ , \qquad x \in (0,1) \ , t \in (0,1) \tag{19}$$

$$u(x,0) = \sin(\pi x) \ , \qquad x \in (0,1) \tag{20}$$

$$0 = u(0,t) = u(1,t) \tag{21}$$

The simplicity of the problem at hand, allows us to find an analytical solution. Guessing, or more formally separation of variables, yields:

$$u^*(x,t) = e^{-\pi^2 t} \sin(\pi x) \ . \tag{22}$$

Thus, the initial profile decays exponentially with characteristic timescale $\tau = \pi^{-2} \approx 0.1$ which will serve us as dividing line between high-curvature, short-term dynamics and low-curvature, asymptotic behavior.

In what follows, we analyze how well FTCS, section I B 1, or PINNs, section I C 1, recover the true solution and what parameters influence the numerical result most significantly.

1. *Forward-Time-Central-Space*

Figure 2 compares the FTCS solution for $\Delta x = 10^{-1}$ and $\Delta x = 10^{-2}$ against the analytical solution in eq. (22). The time increment has been set according to the CFL criterion (13). On balance, one finds satisfactory agreement between both FTCS solutions and the ground truth at evolutionary stages of high ($t = 0.05$) or low curvature ($t = 0.25$). Closer inspection suggests that the coarse grain solution underestimates $u^*(x,t)$ more than the higher resolution approximation does.

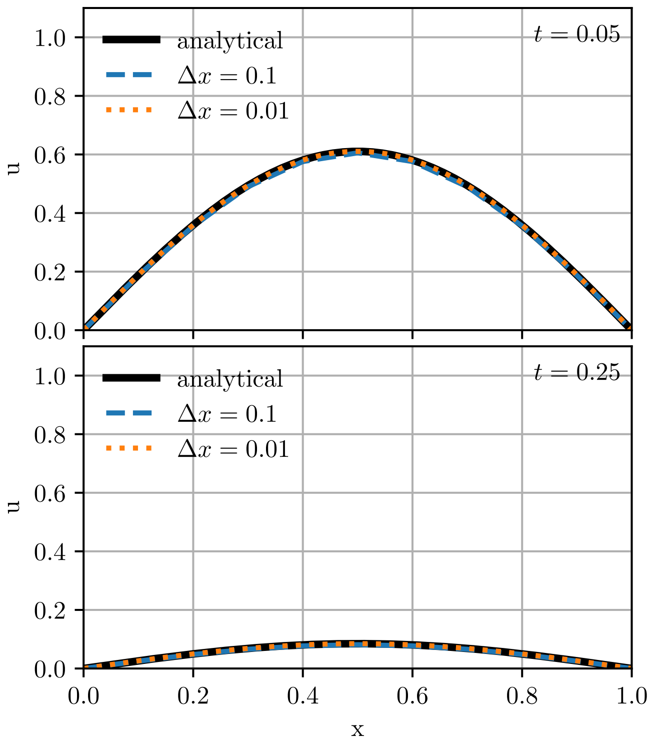

FIG. 2. Comparison of two FTCS solutions with the analytical solution of eq. (22). Solutions differ in the resolution of the spatio-temporal grid with Δx set manually and Δt then deduced via eq. (13). **Top**: Situation at high solution curvature. **Bottom**: Situation at low solution curvature.

To make this observation manifest, we study the absolute difference at the domain center $x = 0.5$ at the same instances in time, but vary the spatial increment, and via eq. (13) also Δt, over two orders of magnitude. Figure 3 illustrates the result. As expected, one recovers a quadratic error dependency in Δx, implying that FTCS is second order accurate in space.

If we focus on $x \in \{0.5, 0.75\}$ in the high curvature snapshot at $t = 0.05$ and observe the absolute error compared to eq. (22) as a function of temporal increment, one arrives at Figure 4. Evidently, the truncation error decays linearly in Δt and FTCS is therefore, in alignment with our expectation, first order accurate in time.

2. *DiffusionPINN*

Moving on to the PINN approach of section I C 1. As explained therein, we attempt to minimize the cost function (17) evaluated on a training set $X_{\mathcal{T}} = X_{\mathcal{T},\Omega} \cup X_{\mathcal{T},\text{IC}} \cup X_{\mathcal{T},\text{BC}}$ of space-time samples (x,t) drawn from a uniform distribution that is adjusted to either the domain interior, its boundary or the initial condition, cf. eq. (18). In what follows, we fix $N_{\mathcal{T},\Omega} = 2^{10}$,

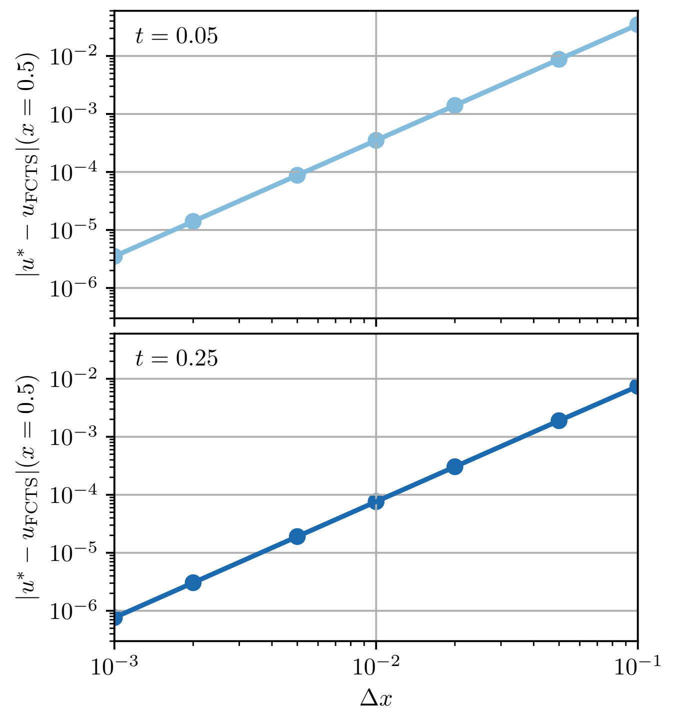

FIG. 3. Truncation error as a function of spatial increment Δx evaluated at $x = 0.5$ and $t = 0.05$ (**Top**), $t = 0.25$ (**Bottom**). Second order error decay is recovered.

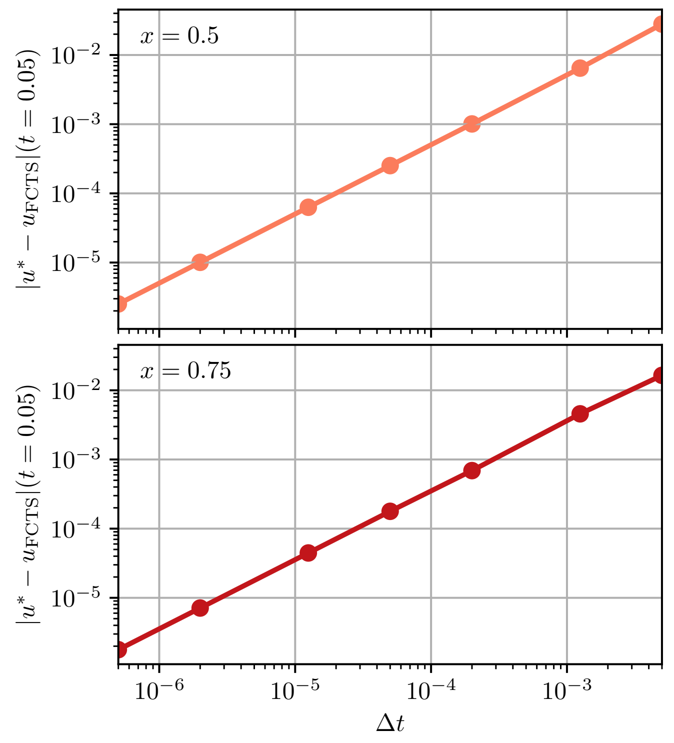

FIG. 4. Truncation error as a function of temporal increment Δt evaluated for the high curvature scenario at $t = 0.05$ and $x = 0.5$ (**Top**), $x = 0.75$ (**Bottom**). First order error decay is recovered.

$N_{\mathcal{T},\mathrm{BC}} = 2^9$, $N_{\mathcal{T},\mathrm{IC}} = 2^7$, which amounts to $N_{\mathcal{T}} = 1664$ unstructured grid points compared to the $N_{\mathrm{FTCS}} = 2000$ points of the uniform FTCS discretization at $\Delta x = 0.1$.

Given the large number of hyperparameters involved in the problem, i.e. the network topology, the activation function and standard hyperparameters that go into the minimization process, we opt for assessing the quality of different parameter combinations by computing a *validation loss* after each minimization epoch and use `RayTune` to (i) sample the hyperparameter space and (ii) select the best parameter combination. For this, a second random but fixed validation set $X_{\mathcal{V}}$ is constructed exactly in the same way as $X_{\mathcal{T}}$. Figure 5 visualizes the former for $N_{\mathcal{V},\Omega} = 2^{10}$, $N_{\mathcal{V},\mathrm{BC}} = 2^9$, $N_{\mathcal{V},\mathrm{IC}} = 2^7$.

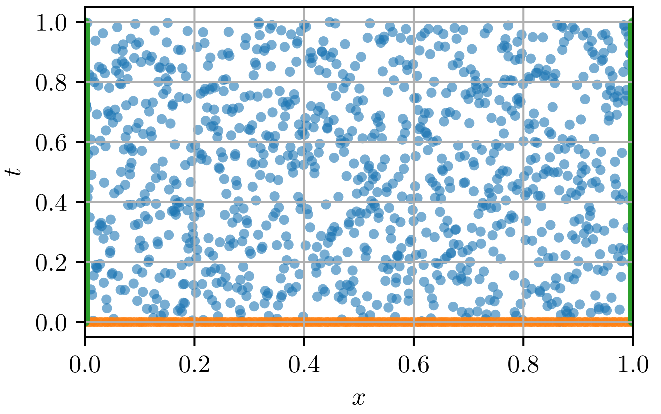

FIG. 5. Validation set $X_{\mathcal{V}}$ entering the hyperparameter tuning. **Blue**: $X_{\mathcal{V},\Omega}$ **Green**: $X_{\mathcal{V},\mathrm{BC}}$ **Orange**: $X_{\mathcal{V},\mathrm{IC}}$

With the validation set in place, we proceed to deduce an approximately ideal hyperparameter combination taken from the search space in Table I. Preliminary tests showed that a single minibatch outperformed a multibatch setup, which is why we fix the number of minibatches to one.

Hyperparameter	Sampling Space/Strategy
`# layers`	`randint(1,4)`
`# neurons per layer`	`randint(5,40)`
`learning rate`	`loguniform(1e-4, 1e-1)`
`optimizer`	`choice(LBFGS, Adam)`
`activation function`	`Tanh`

TABLE I. Hyperparameter search space and sampling strategy for the diffusion PINN.

We sample from Table I 200 times, for each parameter combination initialize a `PINN` and optimize eq. (17) on $X_{\mathcal{T}}$ while monitoring the validation loss, i.e. eq. (17) evaluated on $X_{\mathcal{V}}$, after each epoch. Depending on the loss behavior (increasing, decreasing, stagnating) `RayTune` then stops unpromising trials before we arrive at our artificially imposed epoch maximum of $E = 1000$.

The result of this process is depicted in Fig. 6. Here we show the validation cost $\mathrm{MSE}_{\mathcal{V}}$ of all 200 trial runs. Our findings indicate, not surprisingly, that the choice of the minimization algorithm (`LBFGS` vs. `Adam`) has the biggest influence on our results. `LBFGS` — a second order method — consistently outperforms standard `Adam`, which never reaches a convergence plateau but gets stopped early.

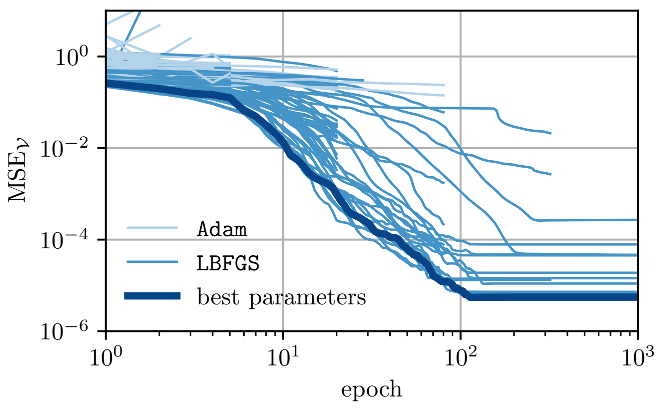

FIG. 6. Evolution of the validation cost $\mathrm{MSE}_\mathcal{V}$ of all explored hyperparameter combinations for the DiffusionPINN.

Table II summarizes the best hyperparameter combination found. Interestingly, the problem seems to have a modest inclination to favour network width over depth.

`# layers`	3
`# neurons per layer`	39
`learning rate`	0.053
`optimizer`	LBFGS
`activation function`	Tanh

TABLE II. Best hyperparameter combination for the diffusion PINN

3. *FTCS vs. PINN*

With the "ideal" parameters and trained network at hand, we proceed to compare the FTCS results with the PINN-suggested solution. This is done in multiple ways. First, we refer to Fig. 7 which illustrates the residuals between the FTCS solution (Panel B) and the PINN result (Panel C) vs. the analytical solution. Here, the PINN is fed with a newly generated test set X_{Test} that coincides with the uniform, stable FTCS grid for $\Delta x = 10^{-1}$. Evidently, FTCS performs worst when the true solution is most feature-rich, i.e. in the high curvature regime, while the PINN result produces an oscillating, bounded residual surface that is insensitive to the exact structure of the analytical solution. Note that no errors accumulate over time as the PINN minimizes the PDE residual in a global sense and is not to be confused with the incremental, time-local approach of a finite difference scheme. At $\Delta x = 10^{-1}$, the PINN solution slightly outperforms the FTCS result with an absolute error of $\epsilon_{\mathrm{PINN}} \approx 10^{-3}$ vs. $\epsilon_{\mathrm{FTCS}} \approx 0.003$.

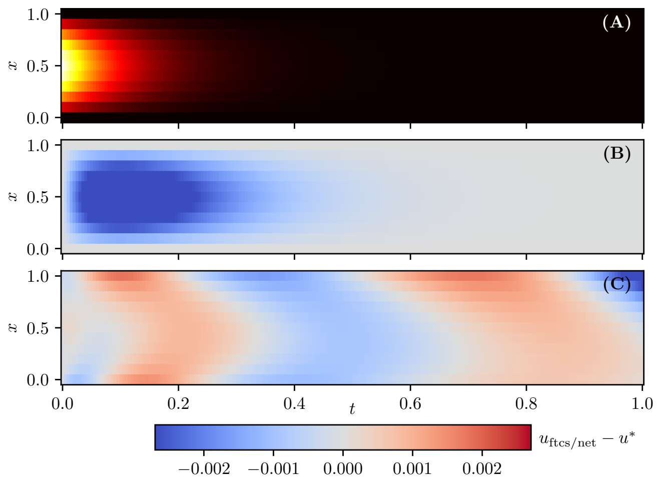

FIG. 7. Spatio-temporal evolution of the analytical function **(A)**, the FTCS residual **(B)** and the PINN error **(C)**, at an computational/inference grid with $\Delta x = 10^{-1}$ and $\Delta t = 5 \cdot 10^{-3}$. The PINN coincides with the trained network associated with the hyperparameters of Table II. No retraining, just forward inference is done to arrive at Panel **(C)**.

Ten-folding the spatial resolution of the uniform and stable FTCS/inference grid yields the situation depicted in Fig. 8.

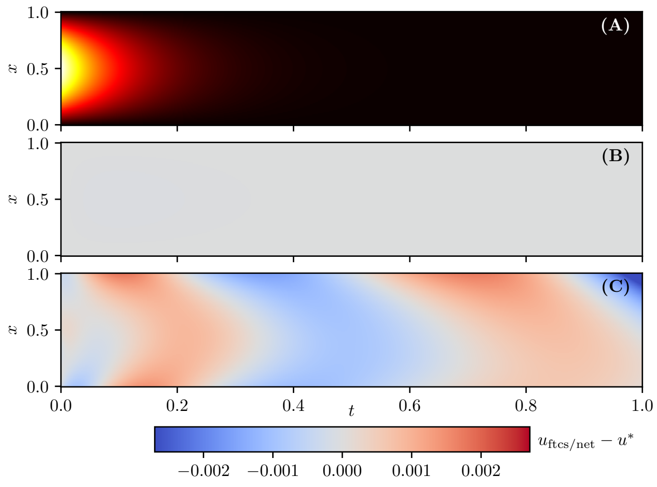

FIG. 8. Spatio-temporal evolution of the analytical function (**(A)**), the FTCS residual **(B)** and the PINN error **(C)**, at an computational/inference grid with $\Delta x = 10^{-2}$ and $\Delta t = 5 \cdot 10^{-5}$. The PINN coincides with the trained network associated with the hyperparameters of Table II. No retraining, just forward inference is done to arrive at Panel **(C)**.

We observe virtually no error in the finite difference solution and exactly the same residual surface for the PINN-suggested solution. This is expected: The PINN approach is essentially equivalent to a mesh-free discretization which induces no additional interpolation errors.

Put differently, once the network is trained its error residual is an invariant of the network. Changing the resolution intrinsic to X_{Test} only dictates how finely this invariant error surface gets evaluated. It is "only" the resolution of the training set $X_{\mathcal{T}}$ and the multitude of hyperparameters which determine the quality of the fit.

To make this point more explicit, we compute the L_2-error:

$$||u_{\text{ftcs/net}} - u^*||_2 = \left(\int \mathrm{d}x\mathrm{d}t |u_{\text{ftcs/net}} - u^*|^2\right)^{\frac{1}{2}} , \quad (23)$$

on various-resolution FTCS inference grids. Again, this only implies re-computation of the FTCS solution, not the PINN result. Approximating eq. (23) via Simpson's quadrature rule, yields Figure 9 and as expected, while the FTCS improves in quality on finer grids, our PINN solution quality is insensitive to the inference grid resolution.

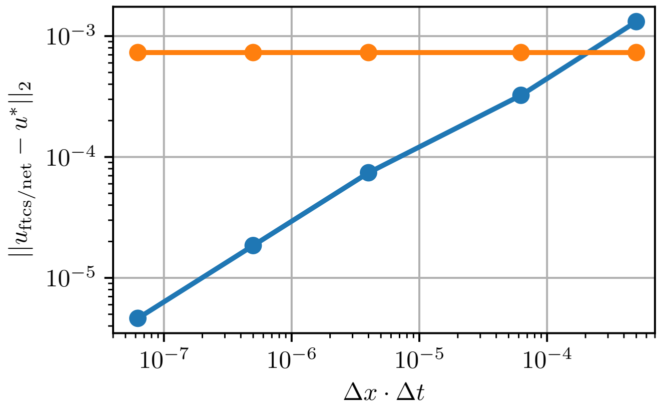

FIG. 9. Behavior of the L_2 error, eq. (23), as a function of grid cell volume. Inference for a trained PINN does not induce an additional interpolation error since the method is mesh-free.

An interesting experiment to conduct is to vary the point density in $X_{\mathcal{T}}$ across multiple training runs. However, preliminary tests showed that the optimal hyperparameters are sensitive to the structure of the training set (not a surprise). It is therefore hard to disentangle both effects in a clean way, at least for the scope of this work. We thus omit a more in depth analysis.

B. Eigendecomposition of Symmetric Matrix

Let us shift our focus to finding the largest eigenvalue of a real, symmetric matrix via solutions to the ODE of eq. (3). In order to make the problem well-defined, a specific matrix has to be fixed. As explained in section I B 1 a particularly instructive choice is a toeplitz-type, symmetric tridiagonal matrix since a simple analytical expression for its spectrum exists, cf. eq. 12. We choose:

$$\boldsymbol{A} = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix} \quad \Rightarrow \quad \lambda_1 = 2 + 2\cos\left(\frac{\pi}{7}\right) , \quad (24)$$

implying the analytical spectrum of Fig. 10.

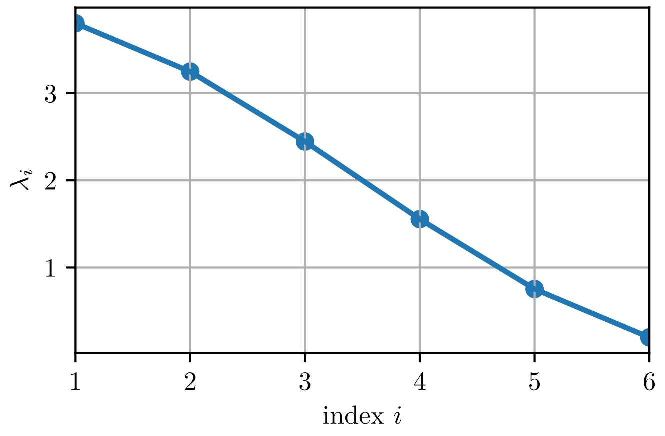

FIG. 10. Analytical eigenspectrum of the symmetric, tridiagonal Toeplitz matrix $\boldsymbol{A}$ as defined in eq. (24).

Before we continue our discussion, very much in the same vein as in section II A 2, and find solution trajectories of eq. (3) by means of a suitably tuned PINN, some preparatory remarks are due.

a. Setting up the Initial Value Problem Realize that Theorem 2 states that equilibrium points are only attained in the limit $t \to \infty$. Clearly, this is unfeasible in a practical setting. Thus, we are required to define a finite, final time T of the temporal domain on which we train the network. As done for Fig. 1, we find a reasonable value for T by integrating eq. (3) via a Runge-Kutta type integrator.

Fig. 11 and 12 then illustrate the time evolution of the vector components x_i alongside the absolute deviation of the true eigenvalue λ_1 and the Rayleigh quotient:

$$\lambda_{\text{ODE}}(t) = \frac{\boldsymbol{x}^\intercal(t)\boldsymbol{A}\boldsymbol{x}(t)}{\boldsymbol{x}(t)^\intercal\boldsymbol{x}(t)} \quad (25)$$

Both results suggest rather fast convergence into the asymptotic equilibrium state around $T \approx 10$. On closer inspection, Fig. 12 suggests an exponential convergence to the true maximal eigenvalue.

Thus, a tractable initial value problem (IVP1) may then be specified as:

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \left[\boldsymbol{x}^\intercal\boldsymbol{x}\boldsymbol{A} - \boldsymbol{x}^\intercal\boldsymbol{A}\boldsymbol{x}\right]\boldsymbol{x} , \quad t \in [0, 10] \quad (26)$$

$$\boldsymbol{x}(0) = \frac{1}{\boldsymbol{x}_0^\intercal\boldsymbol{x}_0}\boldsymbol{x}_0 , \quad \boldsymbol{x}_0 \sim U([0,1])^N . \quad (27)$$

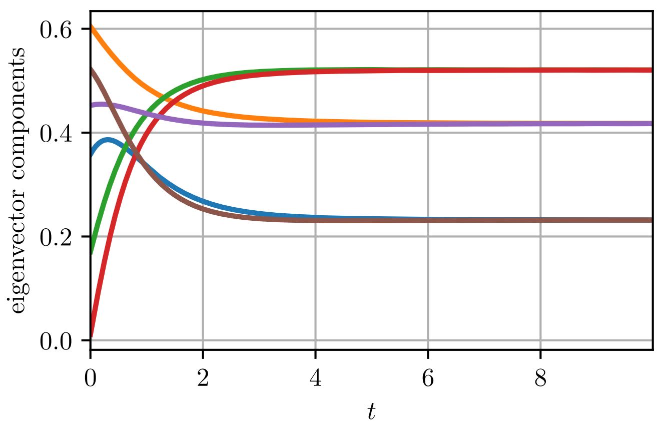

FIG. 11. Time evolution for the vector components of the solution to eq. (3). Initial conditions are chosen randomly. The asymptotic regime is reached at $T \approx 10$.

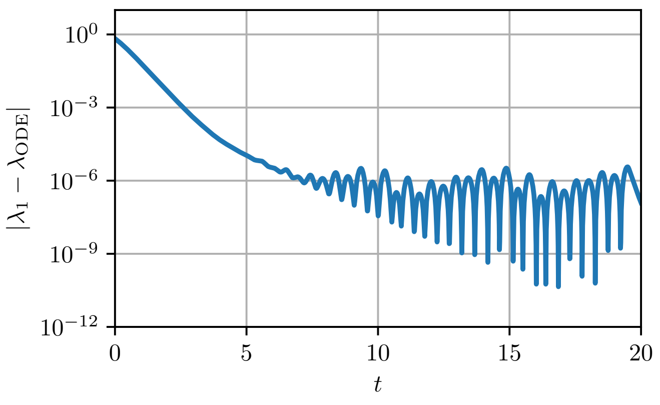

FIG. 12. Time evolution of the absolute deviation between the Rayleigh quotient, cf. (25), associated with the solution of Fig. 11 and the true, largest eigenvalue λ_1. Exponential convergence is observed.

Another approach to solve the half-open time domain problem is to apply a smooth time transformation of the form $\tau = \operatorname{arctanh}(t)$. The resulting initial value problem (IVP2) takes the form:

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\tau} = \frac{1}{1-\tau^2}\left[\boldsymbol{y}^\intercal \boldsymbol{y} \boldsymbol{A} - \boldsymbol{y}^\intercal \boldsymbol{A} \boldsymbol{y}\right] \boldsymbol{y} \ , \quad \tau \in [0,1] \tag{28}$$

$$\boldsymbol{y}(0) = \frac{1}{\boldsymbol{y}_0^\intercal \boldsymbol{y}_0} \boldsymbol{y}_0 \ , \qquad \boldsymbol{y}_0 \sim U([0,1])^N \ . \tag{29}$$

A priori, it is not clear which IVP performs better in practice and we will therefore analyze both.

b. Sampling Strategies for Training and Validation Sets Each IVP implies a different time domain to sample from in order to construct $X_\mathcal{T}$ and $X_\mathcal{V}$. The sampling strategy, however, is still unspecified.

Without any prior knowledge about the solution or its derivatives, a uniform sampling is a consistent choice: No particular patch in the domain should be dominant in the training procedure. This reasoning applies to IVP1.

The situation may be different for IVP2 as the leading factor of the right-hand-side of eq. (28) diverges as $\tau \to 1$. If we want to fix the trajectory as accurate as possible, it may (or may not) be advisable to increase the number of τ-samples close to this divergence. Thus, besides uniform sampling we also consider the possibility of sampling from $\tau \sim (1-\tau)^2$ in the interior of the domain, i.e. for $\tau \in (0,1)$. In practice this is achieved via *rejection sampling*. We tacitly assume the reader is familiar with this technique.

In summary, we are left with three different scenarios: (i) IVP1 trained on uniformly sampled data (IVP1($t \sim U$)), (ii) IVP2 trained on uniformly sampled data (IVP2($\tau \sim U$)) and (iii) IVP2 optimized for a dataset following from rejection sampling (IVP2($t \sim (1-\tau^2)^{-1}$)).

Fig. 13 illustrates the three types of training/validation sets for $N_{\mathcal{T},\Omega} = N_{\mathcal{V},\Omega} = 2^7$.

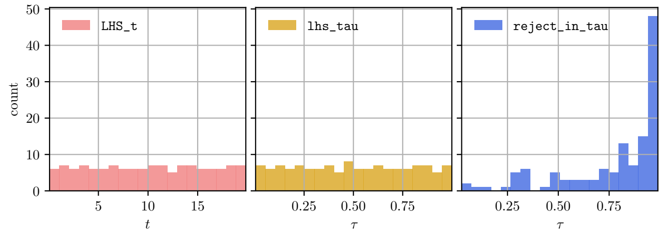

FIG. 13. Comparison of different data set sampling strategies depending on whether a prior time transformation was imposed on the IVP. **Left**: Training set for IVP1 constructed via sampling from $U([0,10])$, **Middle**: Training set for IVP2 constructed via sampling from $U([0,1])$ **Right**: Training set for IVP2 constructed via rejection sampling from $(1-\tau^2)^{-1}$.

c. Cost Function Variations Since we are dealing with an IVP, the concept of standard boundary conditions is absent. However, we can incorporate any dynamical insight into the cost function in order to make the optimization sensitive to additional constraints of the time continuous problem at hand.

In the present situation, *norm conservation* might be such an additional constraint: It is easy to see that eq. (26) and (28) both satisfy

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\tau} = 0 \ . \tag{30}$$

As this holds true for all values of t and τ respectively, we may interpret eq. (30) as a type of "boundary condition" that can be turned on or off at will. We analyze the influence of this in section II C

Furthermore, it is possible to make the initial conditions manifest in the structure of the network-based solution. So far we only considered the case where the network output $\boldsymbol{N}(t)$ is the solution approximation $\boldsymbol{x}(t)$. However, we may also propose a solution of the form:

$$\boldsymbol{x}(t) = \boldsymbol{x}(0) + t\boldsymbol{N}(t) \ . \tag{31}$$

Clearly, this would render the initial condition term in eq. (17) superfluous. Understanding the influence of such an initial condition ”de-constraining” is also analyzed in section II C.

On balance, three cost function types will be investigated: (i) fully constrained (C = ODE + norm + IC), (ii) manifest initial conditions (C = ODE + norm) and (iii) minimally constrained (C = ODE).

C. EigenPINN

Following the discussion in section II A 2, we proceed by selecting the most performant hyperparameter combination out of the search space of Table III.

Hyperparameter	Sampling Space/Strategy
`# layers`	`randint(1,4)`
`# neurons per layer`	`randint(5,40)`
`learning rate`	`loguniform(1e-4, 1e-1)`
`optimizer`	`choice(LBFGS, Adam)`
`activation function`	`choice(Tanh, ReLU)`

TABLE III. Hyperparameter search space and sampling strategy for EigenPINN.

In contrast to Table I, we now also allow the activation function to vary. The number of minibatches remains fixed at unity, in accordance with preliminary testing.

In total nine training scenarios need to be investigated, namely all elements of the cartesian product:

$$\{\text{IVP1}(t\sim U), \text{IVP2}(\tau\sim U), \text{IVP2}(\tau\sim \text{arctanh})\} \times \{\text{ODE+norm+IC}, \text{ODE+norm}, \text{ODE}\}$$

Figure 14 depicts the hyperparameter tuning results for $\{\text{IVP1}(t\sim U)\} \times \{\text{ODE+norm+IC}, \text{ODE+norm}, \text{ODE}\}$. Interestingly, while the discussion of section II A 2 suggested that the hyperparameter tuning is most sensitive to the choice of the optimizer, the situation is more indifferent in the present case. That said, `LBFGS` still performs better, although its significantly longer runtime per step may become prohibitive for larger datasets.

A superficial inspection suggests that the fully constrained cost function setting should yield the best results for IVP1 since its validation cost is lowest although the full cost function contains the largest number of positive terms and should therefore be largest. Similarly, we anticipate the IC-manifest but norm sensitive scenario (ODE+norm) to outperform the minimally constrained setting (ODE).

Extracting the best hyperparameter run for each of the nine IVP-dataset-cost function combinations yields Fig. 15, where unlabeled colors coincide with Fig. 14.

Realize that the magnitude of $\text{MSE}_\mathcal{V}$ may not be a reliable proxy for the final inferrence quality across different IVPs: Just because the validation loss is less for IVP1 (blue, green, orange) then for the IVP2 scenarios does not necessarily mean that IVP1 will yield better estimates for λ_1. After all, the divergence in IVP2 at $\tau = 1$ makes it a much more challenging ODE to begin with.

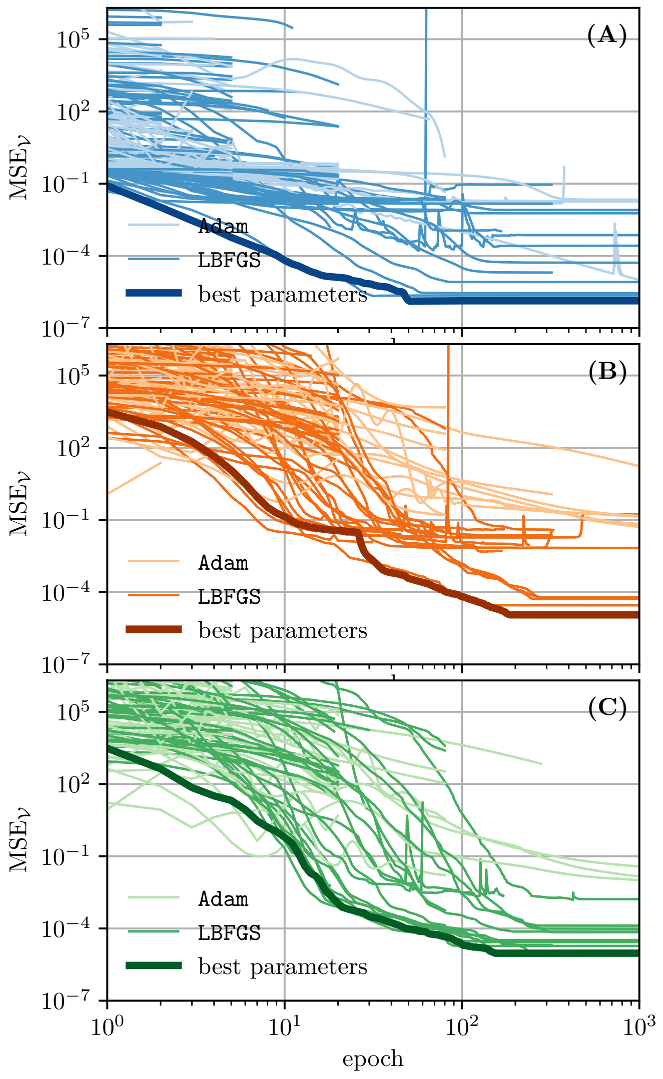

FIG. 14. Illustration of the hyperparameter tuning for IVP1($t \sim U$) and various cost function variants: **(A)**: ODE+norm+IC, **(B)**: ODE+norm, **(C)**: ODE. For each panel 200 hyperparameter combinations were tested. Solid lines indicate the best parameters found.

Ultimately, one has to extract the equilibrated solution vector at $t = T$ or $\tau = 1$ respectively. Doing this for the 9 best PINNs and inserting the result into the Rayleigh quotient, cf. eq. (25) yields Table IV.

The main result may then be summarized as follows: (i) Switching from an artificially bounded $t-$domain (IVP1) to transformed $\tau-$domain (IVP2) has almost no effect. (ii) Non-uniform sampling in IVP2 worsens the predictive capabilities of the PINN marginally. (iii) The most constrained minimization problem, ODE+norm+IC, yields the most accurate results, while dropping either the initial condition constrained or both the IC and norm term yield equally degraded results.

For completeness, Table V reports the optimal hyperparameters of the best PINN. We find that the problem at hand seems to favour network breadth over network

	IVP1($t \sim \mathcal{U}(0, 20)$)	IVP2($\tau \sim \mathcal{U}(0, 1)$)	IVP2($\tau \sim$ arctanh)
ODE + norm + IC	$\lambda_{\text{ODE}} = 3.8019, \epsilon_\lambda = 0.00\%$	$\lambda_{\text{ODE}} = 3.8019, \epsilon_\lambda = 0.00\%$	$\lambda_{\text{ODE}} = 3.7993, \epsilon_\lambda = 0.07\%$
ODE + norm	$\lambda_{\text{ODE}} = 3.0539, \epsilon_\lambda = 19.68\%$	$\lambda_{\text{ODE}} = 3.0534, \epsilon_\lambda = 19.69\%$	$\lambda_{\text{ODE}} = 3.0474, \epsilon_\lambda = 19.85\%$
ODE	$\lambda_{\text{ODE}} = 3.0459, \epsilon_\lambda = 19.89\%$	$\lambda_{\text{ODE}} = 3.0495, \epsilon_\lambda = 19.79\%$	$\lambda_{\text{ODE}} = 3.0701, \epsilon_\lambda = 19.25\%$

TABLE IV. Final inference of the eigenvalue λ_{ODE} for the 9 best trained PINNs. Shown are both the value of λ_{ODE} as well as the relative deviation $\epsilon_\lambda = \lambda_1^{-1} |\lambda_{\text{ODE}} - \lambda_1|$

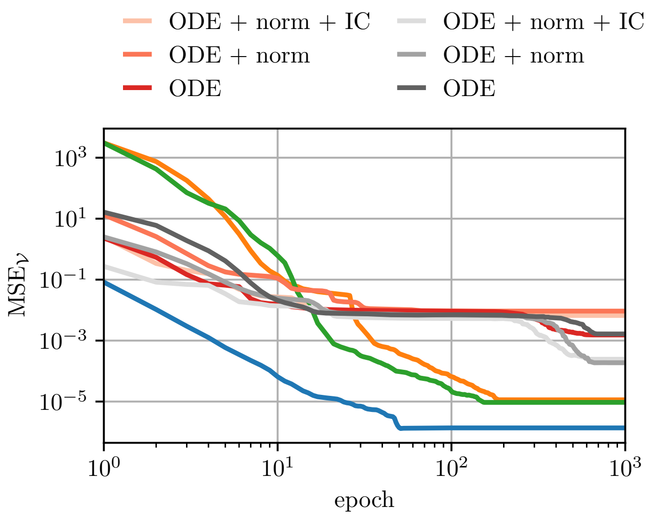

FIG. 15. Direct comparison of the validation loss for the 9 best PINNs obtained through the hyperparameter tuning procedure. **Red**: IVP2($\tau \sim U$) **Gray**: IVP2($\tau \sim$ arctanh) **Other colors**: as in Fig. 14 Evidently, the uniformly sampled IVP1 scenario yields the most accurate results, irrespective if the cost function in use.

depth.

`layers`	1
`nodes`	29
`activation_function`	`Tanh()`
`optimizer_name`	`lbfgs`
`lr`	0.08450
`number_of_minibatches`	1

TABLE V. Optimal hyperparameters of the fully constrained IVP1($t \sim U$) PINN.

We close this section by asking whether the ODE-based PINN approach for extracting the largest eigenvalue of a symmetric matrix is actually competitive with other established methods. The short answer is *no*.

Our exhaustive tests showed that convergence of the network to the largest eigenvalue is highly sensitive to the ODE at hand. In many cases less ideal hyperparameters either let the PINN convergence to a different, non-maximal eigenvalue, or no eigenvalue at all. The hyperparameter tuning presented above was therefore critical in achieving the results of Table IV.

Moreover, PINNs are trained on a per-ODE-basis and do not generalize to classes of ODE/PDEs, thereby limiting the predictive power. Consequently, for each new matrix $\boldsymbol{A}$, a completely new training cycle is required and this is without taking the potential necessity of tweaking hyperparameters into account. The time-to-solution for EigenPINNs is therefore prohibitive for small problems.

`numpy` encapsulated linear algebra methods, on the other hand, are extremely accurate up to machine precision and, compared to the computational effort of EigenPINNs, instantaneous — at least for $N = 6$ dimensions.

It is unclear if our approach becomes more competitive in higher dimensions as exact diagonalization requires $\mathcal{O}(N^3)$ steps. Without further tests it is hard to gauge what implications an increased dimensionality might have besides a trivial increase of nodes in the output layer.

We also want to stress that the EigenPINN approach is *not* what the original work [1] envisioned. Following their argumentation, eq. (3) is merely an analytically tractable surrogate model for a time discrete recurrent network. It is unclear to us how a transformation from the time-continuous domain back to the time-discrete domain might proceed. However, we anticipate the underlying recurrent network to somehow encode matrix $\boldsymbol{A}$ as its connection weights and vector $\boldsymbol{x}$ be the hidden state of the system. Without any external influences, the network should then converge into an equilibrium state consistent with the ODE analysis sketched in section II C.

III. CONCLUSION

Purpose of this work was to showcase in a proof-of-concept style the applicability of neural networks for the solution of partial differential equations — either of ordinary or partial type.

To this end, we studied the one dimensional diffusion equation and an ODE-formulation of the symmetric matrix eigenvalue problem and compared against canonical techniques and analytical results to gauge the quality of the network-based solution and its dependence on various hyperparameters.

For the diffusion problem, we conducted an elaborate hyperparameter study for our network and compared its performance against both an analytical solution and a simplistic, explicit finite difference scheme (FTCS). Not surprisingly, our findings indicate that the network solution quality depends on the hyperparameters of the model but most sensitively on the choice of optimizer. `LBFGS` — a second order optimization scheme — yielded superior accuracy at a moderate runtime. Settling for roughly the same number of points in the training set and the spatio-temporal domain of FTCS resulted in comparable

L_2-errors of both methods compared to the analytical, ground truth. Ten-folding the number of spatio-temporal points for FTCS improved its accuracy by roughly two orders of magnitude at the expense of an increase of runtime by a factor of 1000 — a consequence of the restrictive, quadratic stability criterion. PINNs on the other hand may be interpreted as a mesh-free discretization bypassing the need of explicit stability criterions. Put differently, any sampling of the spatio-temporal domain is in principle admissible for the training of the network and numerical errors do not accumulate naively as the optimization procedure minimizes the PDE residual in a global sense across the entire space-time domain. PINNs also don't incur any interpolation errors by performing inference of the solution at a different set of points compared to the training set — the residual surface is intrinsic to the trained network.

Although not tested in this work, we hypothesize PINNs could outperform standard finite difference methods in higher dimensions when the *curse of dimensionality* in conjunction with stability criterions for explicit schemes or matrix inversions for implicit schemes render their application impossible.

For the eigenvalue analysis, we set out to extract the largest eigenvalue of a real, symmetric, Toeplitz Matrix via an ODE-based PINN and compared it against the analytically known spectrum. To this end, various formulations of the initial value problem, sampling strategies for the training/validation set and cost functions were analyzed. As outlined in section II C, we find the most constrained setting, taking ODE, initial condition and norm conservation into account, to perform best. With hyperparameters optimally tuned, our EigenPINN achieved relative errors of less than 10^{-5}. However, as we discussed in our concluding remarks in section II C, the fine-tuning that is required to achieve such precisions is not competitive with established, exact methods of numerical linear algebra, extension to higher dimensional problems may change this observation.

ACKNOWLEDGMENTS

As last time, the authors would like to express their gratitude to Morten Hjorth-Jensen for his organizational flexibility. 👍

Appendix: Neural Network Basics

1. The Universal Approximation Theorem

The mathematical basis of neural networks as universally applicable approximation to potentially highly nonlinear functions is founded on the *universal approximation theorem.*

Theorem 3. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a locally bounded, piecewise continuous activation function and $\boldsymbol{f} : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m \in C(\Omega, \mathbb{R}^m)$. Iff σ is not polynomial, then for every $\epsilon > 0$ there exist a $k \in \mathbb{N}$, weight matrices $\boldsymbol{W}^1 \in \mathbb{R}^{n\times k}$, $\boldsymbol{W}^2 \in \mathbb{R}^{k\times m}$ and bias vectors $\boldsymbol{b}^1 \in \mathbb{R}^k$, $\boldsymbol{b}^2 \in \mathbb{R}^m$ such that*

$$\sup_{\boldsymbol{x}\in\Omega} ||f(\boldsymbol{x}) - N(\boldsymbol{x})|| < \epsilon \,, \tag{A.1}$$

with:

$$\begin{aligned} \boldsymbol{N} &: \Omega \to \mathbb{R}^m \\ &\boldsymbol{x} \mapsto \boldsymbol{N}(\boldsymbol{x}) = \boldsymbol{W}^{2\intercal} \sigma\left(\boldsymbol{W}^{1\intercal}\boldsymbol{x} + \boldsymbol{b}^1\right) + \boldsymbol{b}^2 \,, \end{aligned} \tag{A.2}$$

and element-wise application of σ.

In other words, under mild assumptions on the nonlinear activation function σ, a composition of linear and nonlinear transformations is capable of reproducing any potentially highly complex, but continuous, function $\boldsymbol{f}$ to arbitrary accuracy.

We call the compostion based approximation $\boldsymbol{N}$ in (A.2) a *feedforward neural network* and mention that other architectures are admissible as well, see [6]. A simple extension of the above three layer network, i.e. one input layer, one hidden layer and one output layer, is to add additional hidden layers with their respective weights, biases and activation functions. We then coin such a network configuration a *deep feedforward neural network* (DFNN).

Note that Theorem 3 only guarantees the existence of such a network representation. It does not specify the number of so called hidden neurons k residing in each layer nor the exact form of the activation function. And although sufficiency of a single hidden layer *is* guaranteed, it might be more accurate to extend the network to an DFNN. Thus, the exact structure of our network approximation is left unspecified and become tunable *hyperparameters* of our model.

We also emphasize that Theorem 3 does not specify the optimal weights or biases or how to compute these *model parameters.*

2. Network Training & Backpropagation

Given (noisy) realization of the training set $\{\boldsymbol{x}_i, \boldsymbol{y}_i \approx f(\boldsymbol{x}_i)\}_{i=1\ldots N_{\mathcal{T}}}$, we can follow the general discussion outlined in [7] and find approximate values for all weights and biases by *minimizing a suitable cost function* $\mathcal{C}(\boldsymbol{N}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ via a (stochastic) gradient descent, or one of its many variants such as `Adam` or `RMSProp`.

This requires specification of the gradients of $\mathcal{C}$ with respect to the *model parameters.* Luckily, the structure of the network representation allows for a computationally efficient procedure, known as *backpropagation*, to compute such.

Let $o_j^k = \sigma(a_j^k)$ be the output value of node j in layer k that comes from piping the linear activation value $a_j^k =$

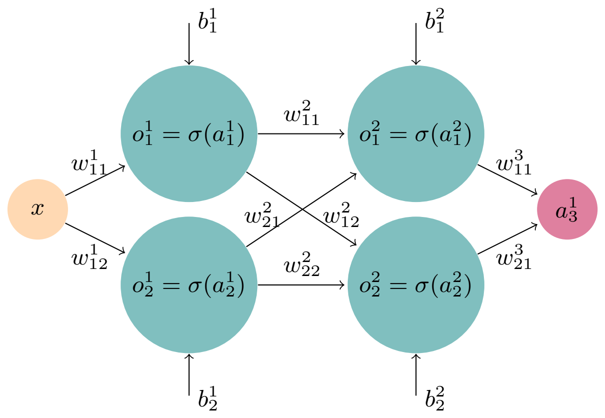

FIG. 16. Graph representation of a FFNN consisting of one input layer with x as input, $K = 2$ hidden layers and one output layer. Each nontrivial neuron is characterized by a weight w_{ij}^k modelling the interaction strength between neuron i of layer $k-1$ and node j in layer k, as well as a bias b_j^k. In this example all hidden layers use the same activation function σ mapping activations a_j^k to outputs o_j^k. Since the graph is fully connected and has no self-connections, one coins this graph structure a *fully connected, directed, acyclic graph.*

$\sum_{i=1}^N w_{ij}^k o_i^{k-1} + b_j^k$ into the activation function σ which, for the sake of simplicity, is assumed to be identical across all $k = 1 \dots K$ hidden layers of constant neuron number $i = 1 \dots N$. We refer to Fig. 16 for a graphical illustration of the case $N = 2, K = 2$.

The sought after gradient elements are then $\frac{\partial \mathcal{C}}{\partial b_j^k}$ and $\frac{\partial \mathcal{C}}{\partial w_{ij}^k}$ and application of the chain rule implies:

$$\begin{aligned} \frac{\partial \mathcal{C}}{\partial b_j^k} &= \frac{\partial \mathcal{C}}{\partial a_j^k}\frac{\partial a_j^k}{\partial b_j^k} = \frac{\partial \mathcal{C}}{\partial a_j^k} \equiv \delta_j^k \,, \\ \frac{\partial \mathcal{C}}{\partial w_{ij}^k} &= \frac{\partial \mathcal{C}}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} \,. \end{aligned} \tag{A.3}$$

At this stage, we merely used the chain rule as a bookkeeping tool to separate the architecture- and cost function independent network bits, i.e. the invariant structure of the activation a_j^k, from the activation and cost function sensitive *error term* δ_j^k. Its exact form will also dependent on the cost function in use. Given the discussion of section I C 1, let us fix $\mathcal{C}_n = \frac{1}{2}(\boldsymbol{y}_n - N(\boldsymbol{x}_n))^2$ as our per-sample-cost.

We then have:

$$\begin{aligned} \delta_j^k = \frac{\partial \mathcal{C}}{\partial a_j^k} &= \sum_{l=1}^N \frac{\partial \mathcal{C}}{\partial a_l^{k+1}}\frac{\partial a_l^{k+1}}{\partial a_j^k} \\ &= \sum_{l=1}^N \delta_l^{k+1} \frac{\partial}{\partial a_j^k}\left(\sum_{m=1}^N w_{ml}^{k+1}\sigma(a_m^k) + b_l^{k+1}\right) \\ &= \sigma'(a_j^k)\sum_{l=1}^N w_{jl}^{k+1}\delta_l^{k+1} \,. \end{aligned} \tag{A.4}$$

A couple of remarks are in order:

Firstly, we note that the computation of the j^{th}-error term of layer k is dependent on error terms of layer $k+1$. Thus, contrary to the information flow in the forward computation, i.e. the execution of the network composition from left to right, gradient computation requires information to flow backwards (right to left). Hence the name backpropagation. In essence, the backflow property originates from the observation that, in contrast to composition, which defines a direction of information flow, multiplication can be carried out from left to right or right to left. The chain rule establishes the link between these operations by mapping unidirectional composition of functions to bidirectional multiplication of their respective gradients.

Secondly, the dependence on our choice of cost function enters through the output layer error term which closes the recurrence hierarchy. For MSE-like cost functions as above, one finds:

$$\delta_i^K = (\boldsymbol{y}_n - \boldsymbol{N}(\boldsymbol{x}_n))\,\sigma'(a_i^K)\,. \tag{A.5}$$

Thirdly, eq. (A.4) is still activation function agnostic, but (A.4) suggests functions with easily computable first derivative. Common choices are:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \Rightarrow \sigma'(x) = \sigma(x)\,(1-\sigma(x)) \tag{A.6}$$

$$\sigma(x) = 1 - \frac{2}{e^{2x}+1} \qquad \Rightarrow \sigma'(x) = 1 - \sigma^2(x) \tag{A.7}$$

$$\sigma(x) = \max(0,x) \qquad \Rightarrow \sigma'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{else} \end{cases} \,. \tag{A.8}$$

At last, we emphasize that in order to evaluate eq. (A.4), the results for the forward pass activations a_j^k, or outputs o_j^k if the activation function derivative allows for it, are required. It is for this reason that a backward pass always follows a forward pass cost function evaluation.

State-of-the-art deep learning frameworks such as `PyTorch` or `TensorFlow` implement gradient computations in a similar, albeit more generalized way, through *reverse-mode autodifferentiation.* Backpropagation, as outlined above, can be understood as reverse-mode autodifferentiation applied to scalar functions, such as the cost function in our case.

It is important to note that the formalism is not limited to computing gradients with respect to network parameters. In fact, autodifferentiation is applicable to *any* computational graph or composition of elementary functions. We use this fact in section I C 1 when we compute the derivative of the network *with respect to its input x* to construct the PDE residual of the PINN cost function.

[1] Z. Yi, Y. Fu, and H. J. Tang, Computers & Mathematics with Applications **47**, 1155 (2004).

[2] M. Raissi, P. Perdikaris, and G. E. Karniadakis, (2017), arXiv:1711.10561 [cs.AI].

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. (Cambridge University Press, USA, 2007).

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS-W* (2017).

[5] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, arXiv preprint arXiv:1807.05118 (2018).

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[7] K. Beshkov, M. Safy, and T. Zimmermann, "CompSci Project 1:Regression Analysis and Resampling Methods," (2022), availiable on request.