Journal of Computational Physics 508 (2024) 112986

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Positional embeddings for solving PDEs with evolutional deep neural networks

Mariella Kast *, Jan S. Hesthaven

Chair of Computational Mathematics and Simulation Science, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland

ARTICLE INFO

Keywords:
Scientific machine learning
Reduced order models
Many query applications
Partial differential equations

ABSTRACT

This work extends the paradigm of evolutional deep neural networks (EDNNs) to solving parametric time-dependent partial differential equations (PDEs) on domains with geometric structure. By introducing positional embeddings based on eigenfunctions of the Laplace-Beltrami operator, geometric properties are encoded intrinsically and Dirichlet, Neumann and periodic boundary conditions of the PDE solution are enforced directly through the neural network architecture. The proposed embeddings lead to improved error convergence for static PDEs and extend EDNNs towards computational domains of realistic complexity.

Several steps are taken to improve performance of EDNNs: Solving the EDNN update equation with a Krylov solver avoids the explicit assembly of Jacobians and enables scaling to larger neural networks. Computational efficiency is further improved by an ad-hoc active sampling scheme that uses the PDE dynamics to effectively sample collocation points. A modified linearly implicit Rosenbrock method is proposed to alleviate the time step requirements of stiff PDEs. Lastly, a completely training-free approach, which automatically enforces initial conditions and only requires time integration, is compared against EDNNs that are trained on the initial conditions. We report results for the Korteweg-de Vries equation, a nonlinear heat equation and (nonlinear) advection-diffusion problems on domains with and without holes and various boundary conditions, to demonstrate the effectiveness of the method. The numerical results highlight EDNNs as a promising surrogate model for parametrized PDEs with slow decaying Kolmogorov n-width.

1. Introduction

In many science and engineering applications, partial differential equations (PDEs) are used to model the dynamic behavior of the problem. The efficient numerical solution of such PDEs remains an active research topic, and is especially relevant for many-query applications that require repeated evaluation for different parameter settings. Reduced basis (RB) methods [1,2] seek to approximate PDE solutions in a low-dimensional linear subspace of the solution manifold, leading to computational models of much smaller cost. For dynamic problems with slowly decaying Kolmogorov n-width, e.g., PDEs with strong advective terms, RB methods typically require a basis of large size and they may lose their efficiency.

Using deep neural networks (DNNs) for surrogate modeling of PDEs has produced exciting results during the past few years. When experimental data or high fidelity (HF) solution data from a traditional solver is available, neural networks can be trained to

* Corresponding author.
E-mail address: mariella.kast@epfl.ch (M. Kast).

https://doi.org/10.1016/j.jcp.2024.112986
Received 7 August 2023; Received in revised form 15 February 2024; Accepted 1 April 2024

Available online 8 April 2024

approximate the PDE operator, e.g. the DeepONet [3], neural ODEs [4], (Fourier) neural operators [5], or to recover the parameter-to-solution map, e.g. POD-NN [6]. When HF data is scarce or absent, automatic differentiation tools can be used to train directly on the fit of the PDE residual without requiring access to a classical numerical solver, as utilized in physics-informed neural networks (PINNs) [7,8] or the Deep Ritz method [9]. Such neural networks typically learn a mapping from the spatio-temporal input coordinate to the solution output.

While DNNs offer universal function approximation, it is well known that training PINNs can be challenging and sensitive to hyperparameter tuning [10], especially when balancing different terms in the residual loss, e.g. the PDE dynamics, boundary conditions, and initial conditions. Recent insights through the lense of neural tangent kernel (NTK) theory [11,12] confirm that the optimization problem in PINNs suffers from stiffness and may not converge to the desired solution. In [12,13], Fourier Features are used to embed the spatial coordinates in a more favorable solution space with a rotation- and translation invariant NTK, which can alleviate the stiffness problem and enable faster training and better accuracy.

Building upon these insights, we propose a positional embedding via variational harmonic features. Previously used in Gaussian process regression [14,15], harmonic features encode geometric information about the computational domain and take into account the gaps and holes that most engineering computational domains possess. The advantages of the proposed embeddings are two-fold, 1) the solution space of the neural network is informed by the geometry in an intrinsic way, 2) it is possible to enforce boundary conditions directly on the neural network, hence eliminating the loss term for the boundary conditions. In [16,17], the authors enforce boundary conditions exactly through the multiplication of the output with specially constructed (polynomial) functions. Similarly, in [18] a linear combination of neural network outputs is constructed to enforce homogeneous Dirichlet boundary conditions on convex domains. To our knowledge, this work is the first example of a unified framework to enforce both Dirichlet and Neumann boundary conditions via embedding functions on a neural network.

When training DNNs with time as an input coordinate, ensuring causality is another major challenge in solving time-dependent problems with PINNs. Recently, exciting progress has been made for directly evolving the parameters of a neural network in time [18–20], via an update equation that is derived from the PDE dynamics. The "training" is thus done via time-integration over the PDE dynamics, similar to classical ODE solvers. Building on these ideas, we propose a numerically efficient computation of the update step by employing Krylov solvers. We further describe how linearly-implicit solvers based on Rosenbrock methods [21,22] can be used to deal with stiff problems at a small additional computational cost. Constructing neural networks that automatically satisfy the initial condition allows one to completely remove the need for gradient-descent based training. Combining these advances with the feature embeddings, allows for an to efficient solution of parametrized transport-dominated problems on complicated domains.

In Section 2, we briefly discuss the problem setup for parametrized time-dependent PDEs and the different model paradigms that can be used to approximate the solution. In Section 3, we discuss the construction and properties of the positional embedding and show how boundary conditions are enforced. In Section 4, the time-stepping scheme and "training-free" approach is explained and we introduce the use of linearly-implicit solvers, as well as an active sampling scheme. In Section 5, we demonstrate the methods on numerical examples to highlight their capabilities and limits. We summarize our results and offer concluding remarks in Section 6.

2. Problem setting

2.1. PDE formulation

We consider time dependent PDEs, which may depend on a parameter vector $\boldsymbol{\alpha} \in \Omega_\alpha$, defined on a spatial domain $\Omega_x \subset \mathbb{R}^d$ with boundary $\partial\Omega_x$:

$$\begin{aligned} &\frac{\partial u(t,\mathbf{x},\boldsymbol{\alpha})}{\partial t} = f(u,t,\mathbf{x},\boldsymbol{\alpha}) \quad \text{for } (t,\mathbf{x},\boldsymbol{\alpha}) \in [0,\infty)\times\Omega_x\times\Omega_\alpha, \\ &\mathcal{B}u(t,\mathbf{x},\boldsymbol{\alpha}) = g(t,\mathbf{x},\boldsymbol{\alpha}) \quad \text{for } (t,\mathbf{x},\boldsymbol{\alpha}) \in [0,\infty)\times\partial\Omega_x\times\Omega_\alpha, \\ &u(0,\mathbf{x},\boldsymbol{\alpha}) = u_0(\mathbf{x},\boldsymbol{\alpha}) \quad \text{for } (\mathbf{x},\boldsymbol{\alpha}) \in \Omega_x\times\Omega_\alpha, \end{aligned} \tag{1}$$

where $\mathbf{x}$ is the spatial coordinate, $u(t,\mathbf{x},\boldsymbol{\alpha})$ is the time-dependent solution field, $f(u,t,\mathbf{x},\boldsymbol{\alpha})$ is a nonlinear differential operator and $u_0(\mathbf{x},\boldsymbol{\alpha})$ is a compatible initial condition. We assume that the PDE problem (1) is equipped with an appropriate boundary condition operator $\mathcal{B}$, such that the solution u is well posed for all times t and parameter instances $\boldsymbol{\alpha}$. We refer to the collection of all solution instances as the solution manifold $\mathcal{M} = \{u(t,\mathbf{x},\boldsymbol{\alpha}) : (t,\mathbf{x},\boldsymbol{\alpha}) \in [0,\infty)\times\Omega_x\times\Omega_\alpha\}$.

We seek to build a computational model which accurately approximates all solution instances u in $\mathcal{M}$ by their numerical counterparts $\hat{u}$, which are part of the numerical solution manifold $\hat{\mathcal{M}} = \{\hat{u}(t,\mathbf{x},\boldsymbol{\alpha}) : (t,\mathbf{x},\boldsymbol{\alpha}) \in [0,T]\times\Omega_x\times\Omega_\alpha\}$, where T is the final computation time. The choice of numerical method for computing $\hat{u}$ influences the properties of $\hat{\mathcal{M}}$, in particular how well the true solution manifold can be approximated and at what computational cost.

2.2. FE solution and reduced basis methods

Classic numerical techniques such as the Finite Element (FE) method, discretize the physical domain to recover a coefficient vector $\mathbf{c}$, reflecting a finite number of degrees of freedom, which allows to write the solution as a linear combination of N basis functions $\varphi(\mathbf{x})$:

$$\hat{u}_{\mathrm{FE}}(t,\mathbf{x},\boldsymbol{\alpha}) = \sum_{i=1}^{N} c_i(t,\boldsymbol{\alpha})\varphi_i(\mathbf{x}) \tag{2}$$

This coefficient vector is then evolved in time for each parameter instance separately. The computational cost of approximating the whole solution manifold scales with the dimension of the physical and parameter space and suffers from the curse of dimensionality. Reduced basis methods are an efficient way of decreasing the computational burden, when the problem has a fast-decaying Kolmogorov n-width [23–25] – in such cases it is possible to approximate the true solution manifold by a linear combination of n basis functions $\Psi(\mathbf{x})$, where $n << N$:

$$\hat{\mathcal{M}}_{\mathrm{RB}} = \mathrm{span}\{\Psi_1, \Psi_2, ..., \Psi_n\} \approx \hat{\mathcal{M}}_{\mathrm{FE}}$$

and the solutions can be expressed with a reduced coefficient vector $\boldsymbol{\beta}$

$$\hat{u}_{\mathrm{RB}}(t,\mathbf{x},\boldsymbol{\alpha}) = \sum_{i=1}^{n} \beta_i(t,\boldsymbol{\alpha})\Psi_i(\mathbf{x}). \tag{3}$$

2.3. Neural network solvers

Neural networks have appeared as an appealing alternative of nonlinear models as they promise universal function approximation without suffering from the curse of dimensionality. In this work, we consider fully connected feed forward neural networks, recursively defined as

$$\mathbf{v_{l+1}}(\mathbf{v_l}) = \sigma(\mathbf{W_l}\mathbf{v_l} + \mathbf{b_l}), \tag{4}$$

where $\sigma(\cdot)$ is the activation function, $\mathbf{W_l}$ and $\mathbf{b_l}$ are the weight matrix and bias of the lth layer, and inputs to the NN are specified via the 0th layer $\mathbf{v_0}$. The activation function for the output is taken to be the identity function leading to a linear last layer $v_L = \mathbf{W_{L-1}}\mathbf{v_{L-1}} + \mathbf{b_{L-1}}$.

Physics-informed neural networks (PINNs) provide a possible solution approach, which trains the NN to learn the map from inputs $\mathbf{v_0} = (t,\mathbf{x},\boldsymbol{\alpha})$ to obtain the NN approximation $\hat{u}_{\mathrm{PINN}}(t,\mathbf{x},\boldsymbol{\alpha};\boldsymbol{\theta}) = v_L$. The challenge lies in finding the NN parameter vector $\boldsymbol{\theta}$, which is the collection of all weights and biases, such that the PINN captures the full solution manifold for all time and parameter instances.

This is usually achieved via the optimization of a loss function, which incorporates the PDE dynamics, the fit of the initial condition and the boundary conditions as defined in (1), and is evaluated at (randomly sampled) $n_c = n_{\mathrm{PDE}} + n_{\mathrm{init}} + n_{\mathrm{BC}}$ collocation points:

$$\begin{aligned}\mathcal{L} = &\sum_{i=1}^{n_{\mathrm{PDE}}}\left(\frac{\partial\hat{u}(t_i,\mathbf{x}_i,\boldsymbol{\alpha}_i;\boldsymbol{\theta})}{\partial t} - f(\hat{u},t,\mathbf{x}_i,\boldsymbol{\alpha}_i)\right)^2 + \sum_{i=1}^{n_{\mathrm{init}}}\left(\hat{u}(0,\mathbf{x}_i,\boldsymbol{\alpha}_i;\boldsymbol{\theta}) - u_0(\mathbf{x}_i,\boldsymbol{\alpha}_i)\right)^2\\ &+ \sum_{i=1}^{n_{\mathrm{BC}}}\left(\mathcal{B}\hat{u}(t_i,\mathbf{x}_i,\boldsymbol{\alpha}_i;\boldsymbol{\theta}) - g(t_i,\mathbf{x}_i,\boldsymbol{\alpha}_i)\right)^2\end{aligned} \tag{5}$$

The loss in (5) is usually minimized via gradient-based methods such as ADAM, where automatic differentiation (AD) is used to evaluate (5) and the loss gradient $\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}}$. It is well known that this training process requires careful balancing of the loss terms and may fail to converge to acceptable results [10,12] unless all hyperparameters are tuned carefully.

We propose two major improvements to facilitate the learning of parameters $\boldsymbol{\theta}$: Firstly, we introduce positional embeddings $\Phi(\mathbf{x}) = \left[\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), ..., \phi_{n_\phi}(\mathbf{x})\right]$ for the spatial coordinate, which adapt the NN to the targeted solution manifolds and automatically enforce the boundary conditions. Secondly, we use evolutional deep neural networks (EDNNs), where only the spatial and parametric coordinates form the input vector of the NN, $\mathbf{v_0} = (\Phi(\mathbf{x}),\boldsymbol{\alpha})$, and the neural network parameters are updated in time: $\hat{u}_{\mathrm{EDNN}}(\Phi(\mathbf{x}),\boldsymbol{\alpha};\boldsymbol{\theta}(t))$. This turns the learning problem into finding the time trajectory of NN parameters $\boldsymbol{\theta}(t)$. Fig. 1 provides a visual comparison of the RB, PINN and EDNN approach.

Remark. We note an interesting connection between the EDNN approach and the reduced basis formulation in (3). Let $(\mathbf{v_{L-1}})_i = \Gamma_i(\Phi(\mathbf{x}),\boldsymbol{\alpha};\boldsymbol{\theta}(t))$ be the intermediate values at the second to last layer in (4), and write the NN output as

$$\hat{u}_{\mathrm{EDNN}}(\Phi(\mathbf{x}),\boldsymbol{\alpha};\boldsymbol{\theta}(t)) = \sum_{i=1}^{n_{\mathrm{hid}}} \beta_i(t)\Gamma_i(\Phi(\mathbf{x}),\boldsymbol{\alpha};\boldsymbol{\theta}(t)). \tag{6}$$

One can then interpret the positional embedding Φ of the EDNN approach as a reduced basis, which is evolved in time with the transformation Γ. This time adaptation of the basis allows to handle problems with slow-decaying Kolmogorov n-width.

3. Positional embeddings

In this section, we will show how a positional embedding layer for a NN can be used to directly enforce the boundary conditions. We extend the work in [18] as our framework can be used for Dirichlet and Neumann boundary conditions on spatial domains

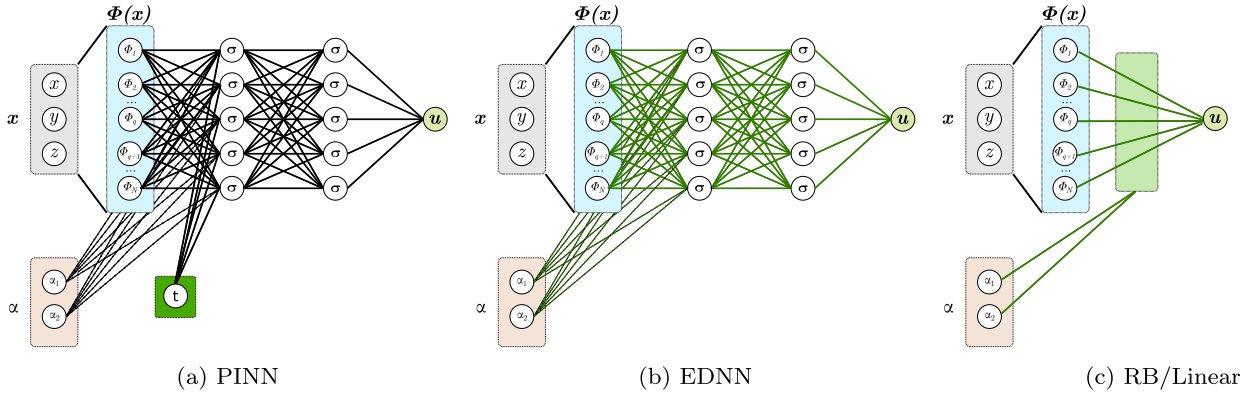


Fig. 1. Overview of different numerical models. Green color indicates that a NN parameter or RB coefficient evolves in time. The green box in the RB model (c) symbolizes the computation of the RB coefficient vector in (3). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

with arbitrary geometry. The positional embeddings are constant in time, consequently their computation can be treated as a pre-processing step and all necessary numerical operations can happen in an "offline" phase.

3.1. Variational harmonic features

Variational harmonic features have previously been used for Gaussian process regression [14,15] and are computed via the eigenfunctions of the Laplace (Beltrami) operator:

$$\begin{aligned} -\nabla^2\phi(\mathbf{x}) &= \lambda\phi(\mathbf{x}), \quad \mathbf{x}\in\Omega_x, \\ \mathcal{B}\phi(\mathbf{x}) &= 0, \quad \mathbf{x}\in\partial\Omega_x. \end{aligned} \tag{7}$$

When solving PDEs, the geometric shape and boundary conditions often encode key features of the solution, and variational harmonic features allow us to adapt the neural network to these characteristics. For certain geometries, the solutions to (7) are known analytically, while for domains with arbitrary shapes, no closed form solutions to (7) are known and the eigenfunctions need to be computed numerically.
The embedding with (Gaussian) random Fourier features [12,13] takes the form

$$\phi_i(\mathbf{x}) = [\cos(\mathbf{b_i}^\top\mathbf{x}), \sin(\mathbf{b_i}^\top\mathbf{x})], \quad \text{with } (\mathbf{b}_i)_l \sim \mathcal{N}(0,\sigma^2), \tag{8}$$

where the vectors $\mathbf{b_i}$ are sampled from a normal distribution according to a lengthscale parameter σ. This also fits within this more general framework: Fourier features correspond to the solutions of (7) for the infinite domain ("no boundary conditions"). Indeed, for computer vision tasks, Fourier features and the induced spatially invariant NTK, are a natural choice as images and videos are usually taken of a continuous real world scene. In contrast to this, variational harmonic features impose a spatially-varying NTK, accounting for general features, e.g. holes, in the geometry.

Remark. By restricting to a finite domain, the solution spectrum of (7) becomes discrete, and the features are not sampled from a continuous probability measure in contrast to (8). We chose to select the n_Φ eigenfunctions associated with the n_Φ lowest eigenvalues. In certain applications e.g. multi-scale phenomena, other choices may be more appropriate [12].

3.2. Approximation space and boundary conditions

In this section, we seek to qualitatively understand which functions we can approximate for a NN with embedding $\Phi(\mathbf{x})$, i.e. we wish to characterize the solution manifold $\hat{\mathcal{M}}$ of $\hat{u}(\Phi(x);\boldsymbol{\theta})$. It has previously been demonstrated in [18,19], that $\hat{u}(\Phi(x))$ inherits the periodicity when the embedding functions $\Phi(x)$ are periodic. We now extend this to embeddings with Neumann and Dirichlet boundary conditions.

3.2.1. Imposing boundary conditions

Let $\mathcal{B}_{\mathrm{NM}}u(\mathbf{x}) = \nabla_{\mathbf{x}}u\cdot\mathbf{n}(\mathbf{x})$, be the operator acting on the Neumann boundary $\partial\Omega_{\mathrm{NM}}\subset\partial\Omega_x$, where $\mathbf{n}(\mathbf{x})$ is the unit vector normal to the boundary. Let $\mathcal{B}_{\mathrm{DC}}u(\mathbf{x}) = u(\mathbf{x})$ be the operator acting on the Dirichlet boundary $\partial\Omega_{\mathrm{DC}}\subset\partial\Omega_x$. Let $\Phi(x)$ be an embedding, where each $\phi_i(\mathbf{x})$ is a solution to (7) such that

$$\begin{aligned} \mathcal{B}_{\mathrm{NM}}\phi_i(\mathbf{x}) &= 0 \quad \forall\mathbf{x}\in\partial\Omega_{\mathrm{NM}}, \\ \mathcal{B}_{\mathrm{DC}}\phi_i(\mathbf{x}) &= 0 \quad \forall\mathbf{x}\in\partial\Omega_{\mathrm{DC}}. \end{aligned} \tag{9}$$

For the NN surrogate $\hat{u}(\Phi(\mathbf{x}))$, it then holds that

- $\mathcal{B}_{\mathrm{NM}}\hat{u}(\Phi(\mathbf{x})) = 0 \; \forall \mathbf{x} \in \partial\Omega_{\mathrm{NM}}$, i.e. the neural network will inherit the Neumann boundary conditions of the Laplace eigenfunctions. The proof follows directly from applying the chain rule:

$$\mathcal{B}_{\mathrm{NM}}\hat{u}(\Phi(\mathbf{x})) = \nabla_{\mathbf{x}}\hat{u} \cdot \mathbf{n}(\mathbf{x}) = \sum_{i=1}^{n_\Phi} \frac{\partial \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x})} \underbrace{\nabla_{\mathbf{x}}\phi_i(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})}_{=0} = 0 \quad \text{for } \mathbf{x} \in \partial\Omega_{\mathrm{NM}}. \tag{10}$$

- $\mathcal{B}_{\mathrm{DC}}\hat{u}(\Phi(\mathbf{x})) = \mathcal{B}_{\mathrm{DC}}\hat{u}(\mathbf{0}) = c$, for $\mathbf{x} \in \partial\Omega_{\mathrm{DC}}$, i.e. all points on the Dirichlet boundary are mapped to the same constant c.

We can use this to our advantage and construct an auxiliary network $\tilde{u}$, that automatically enforces homogeneous Dirichlet (and Neumann) boundary conditions:

$$\tilde{u}(\Phi(\mathbf{x})) = \hat{u}(\Phi(\mathbf{x})) - \hat{u}(\mathbf{0}), \tag{11}$$

so that

$$\mathcal{B}_{\mathrm{DC}}\tilde{u}(\Phi(\mathbf{x})) = c - c = 0 \quad \forall \mathbf{x} \in \partial\Omega_{\mathrm{DC}}. \tag{12}$$

When inhomogeneous boundary conditions need to be prescribed, one can add a lifting term to (11). An appropriate embedding can thus eliminate the boundary condition loss term in (5) and simplify the optimization problem.

It is essential to acknowledge that the proposed embedding restricts the solution space of the neural network. Using an embedding that automatically enforces Neumann boundary conditions would be a poor choice to represent a PDE problem with homogeneous Dirichlet boundary conditions as the gradients along the boundary will be forced to 0. Fourier features (8) form a larger solution space and can be used to approximate any boundary condition. We will further illustrate this generality vs. specificity trade-off in an example in Section 3.4.

Remark. One can compute Laplace eigenfunctions with Robin boundary conditions, but this does not eliminate a loss term as the NN does not inherit the Robin boundary condition of the embedding.

3.2.2. Selecting basis functions

We provide some insights into how the size of the embedding affects the approximation space/solution manifold of the NN $\hat{u}(\Phi(\mathbf{x}))$. It is widely accepted, that NNs are universal function approximators, this section discusses how an embedding layer can limit the approximation capability of the NN and how to avoid this issue.

For a bounded domain Ω_x, it is well established that the eigenfunctions of the Laplace operator form an orthogonal and complete basis of the Hilbert space $L_2(\Omega_x)$ [14]. In practice, we will need to truncate this basis to a finite number of embedding functions n_Φ, and $\Phi(\mathbf{x})$ will not be a complete basis for $L_2(\Omega)$. For reduced basis methods, this truncation error is well understood [2], but the inclusion of nonlinear layers in the neural network enhances the expressivity of its solution manifold and demands a shift in perspective:

To enforce boundary conditions, it is sufficient to include a single embedding function $\phi_i(\mathbf{x})$. This may, however, restrict the solution manifold of the NN due to symmetries in $\phi_i(\mathbf{x})$. A trivial example is given by the first Laplace eigenfunctions on the unit line with Dirichlet boundary conditions: $\phi_1(x) = \sin(\pi x), \phi_2(x) = \sin(2\pi x)$. It is impossible to find a mapping $\hat{f}$, such that

$$\hat{f}\left(\phi_1(x)\right) = \phi_2(x) \tag{13}$$

due to the reflection symmetry around $x = 0.5$, e.g. $\phi_1\left(\frac{1}{4}\right) = \phi_1\left(\frac{3}{4}\right)$, but $\phi_2(\frac{1}{4}) = -\phi_2(\frac{3}{4})$. This implies that a NN $\hat{u}(\phi_1(x))$ can only approximate functions with the same reflection symmetry. Conversely, a NN with only two embedding functions $\Phi_2(\mathbf{x}) = [\phi_1(x), \phi_2(x)]$ uniquely identifies each point on the unit line, and consequently the approximation of $L_2(\Omega)$ via the solution manifold of the NN is only limited by the size of the parameter space of the NN.

The Neumann case on the unit line does not suffer from symmetries, in fact we know from the definition of the Chebyshev polynomials that there exists a transformation T_n, such that

$$T_n(\cos(\pi x)) = \cos(n\pi x), \tag{14}$$

i.e. an embedding of size one is sufficient to recover the solution space of interest.

In higher dimensions, it is more difficult to identify symmetries, but similar considerations apply: as long as the embedding is rich enough to uniquely identify each point in the spatial domain Ω_x, the nonlinear approximation capabilities of the neural network allow for general function approximation. This also implies that any collection of functions ϕ_i can be used as a positional embedding $\Phi(x)$ as long as they encode the boundary conditions and form a rich enough space. It is thus possible to use functions obtained from a reduced basis approximation, the eigenfunctions of another operator, or directly learn the embeddings from data. We focus on the eigenfunctions of the Laplace operator due to the close connection with Fourier features and their favorable influence on training [12].

Equipped with these insights, we propose a simple algorithm to test whether a given choice of embedding functions is sufficient to uniquely identify each point for a given spatial discretization. In Euclidian space, we consider two points $\mathbf{x}_i$, $\mathbf{x}_j$ to be close if their Euclidian distance is lower than a threshold value ε_E:

$$\text{Euclidian criterion: } ||\mathbf{x}_i - \mathbf{x}_j||_2 \leq \varepsilon_E.$$

We can define a similar criterion on the embedding vectors $\boldsymbol{\phi}(\mathbf{x}_i)$ $\boldsymbol{\phi}(\mathbf{x}_j)$ associated with the spatial points:

$$\text{Embedding criterion: } ||\boldsymbol{\phi}(\mathbf{x}_i) - \boldsymbol{\phi}(\mathbf{x}_j)||_2 \leq \varepsilon_\phi,$$

where ε_ϕ is a user chosen tolerance for the distance in the embedded space, which should also take numerical error in the computed embeddings into account. We then define a "collision" of two points if they are close in embedding space, but far apart in Euclidian space:

$$\text{Collision: } ||\boldsymbol{\phi}(\mathbf{x}_i) - \boldsymbol{\phi}(\mathbf{x}_j)||_2 \leq \varepsilon_\phi, \text{ but } ||\mathbf{x}_i - \mathbf{x}_j||_2 \geq \varepsilon_E$$

With these definitions, we can write a simple algorithm (see Algorithm 1) that scales quadratically with the size of the spatial discretization ($O(n_x^2)$ runtime), which checks that each pair of points ($\mathbf{x}_i$, $\mathbf{x}_j$) is collision free in the embedded space. Once a set of embedding functions has passed this test, the L_2 approximation capability of the neural network should scale with the size of the architecture in the usual way. In practice, let L be the typical length of the spatial domain, setting ε_E to be the average edge length of the mesh/distance between points and $\varepsilon_\phi = 0.1 L \varepsilon_E$, appears a reasonable choice and detected collisions for all analytic examples we considered.

Remark. Independent of this test, it can often be beneficial to include a higher number of embedding functions, as it appears to facilitate the learning of solutions with varying lengthscales, similar to Fourier Features [12].

Algorithm 1 Unique identification test.

Input: Vector of embedding functions $\Phi(\mathbf{x})$, spatial discretization $S = \{x_k\}_{k=1}^n$ of the spatial domain Ω_x (e.g. from FE mesh), tolerances ε_E, ε_Φ.
Output: `True` if no collisions are detected, otherwise `False` and a list of collisions.

```
Initialize: C = [ ], Indicator b = True.
for i = 1 to n do
    for j = i + 1 to n do
        d_E = ||x_i − x_j||_2
        d_Φ = ||φ(x_i) − φ(x_j)||_2
        if d_E > ε_E and d_Φ < ε_Φ then
            C.append((x_i, x_j))
            b = False
        end if
    end for
end for
return b, C
```

3.3. Numerical approximation

For geometric domains in two or three spatial dimensions, we compute solutions to (7) using a FE method using the Ansatz introduced in (2). Let

$$\hat{\phi}(\mathbf{x}) = \sum_{i=1}^{N} c_i \varphi_i(\mathbf{x}), \tag{15}$$

be the trial functions and v be test functions of the same FE space with C^0 continuous elements for the weak formulation of (7):

$$\int_{\Omega_x} \nabla\hat{\phi} \cdot \nabla v \mathrm{d}\mathbf{x} = \lambda \int_{\Omega_x} \hat{\phi} v \mathrm{d}\mathbf{x}, \forall v. \tag{16}$$

To utilize (15) as a positional embedding for a PINN or EDNN, we need to compute its gradients with respect to the spatial coordinate $\mathbf{x}$. As the gradient of the FE function $\hat{\phi}(\mathbf{x})$

$$\nabla_{\mathbf{x}}\hat{\phi}(\mathbf{x}) = \sum_{i=1}^{N} c_i \nabla_{\mathbf{x}} \varphi_i(\mathbf{x}), \tag{17}$$

is no longer C^0 continuous, we project the gradient (17) back to the original FE space component-wise:

$$\int_{\Omega_x} g_l v \mathrm{d}\mathbf{x} = \int_{\Omega_x} (\nabla_x \hat{\phi})_l v \mathrm{d}\mathbf{x}, \tag{18}$$

so that

$$\nabla_{\mathbf{x}} \hat{\phi}_i(\mathbf{x}) \approx \mathbf{g_i}(\mathbf{x}). \tag{19}$$

Higher order derivatives can then be evaluated recursively. To control the projection error, we solve (16) on a fine mesh and use P3 Lagrange elements. Assembling the system leads to the discrete linear eigenvalue problem

$$K\mathbf{c} = \lambda M\mathbf{c}, \tag{20}$$

which can be solved with standard linear algebra routines.

To compute spatial derivatives of the neural network, we use the chain rule to combine the spatial FE gradient of the embedding function and the automatic differentiation gradient (AD) of the neural network with respect to its inputs $\phi_i(\mathbf{x})$:

$$\frac{\partial \hat{u}\left(\Phi(\mathbf{x})\right)}{\partial x_l} = \sum_{i=1}^{n_\Phi} \underbrace{\frac{\partial \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x})}}_{\text{AD}} \underbrace{\frac{\partial \phi_i(\mathbf{x})}{\partial x_l}}_{\text{FE}} \approx \sum_{i=1}^{n_\Phi} \underbrace{\frac{\partial \hat{u}(\hat{\Phi}(\mathbf{x}))}{\partial \hat{\phi}_i(\mathbf{x})}}_{\text{AD}} \left(\mathbf{g_i}(\mathbf{x})\right)_l. \tag{21}$$

Computing second order spatial derivatives requires slightly more effort – applying the chain rule twice leads to

$$\frac{\partial^2 \hat{u}}{\partial x_l \partial x_k} = \sum_{i=1}^{n_\Phi} \left(\underbrace{\frac{\partial \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x})}}_{\text{AD}} \underbrace{\frac{\partial^2 \phi_i(\mathbf{x})}{\partial x_l \partial x_k}}_{\text{FE}} + \sum_{j=1}^{n_\Phi} \left(\underbrace{\frac{\partial^2 \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x}) \partial \phi_j(\mathbf{x})} \frac{\partial \phi_i(\mathbf{x})}{\partial x_l}}_{\text{AD of (21)}} \frac{\partial \phi_j(\mathbf{x})}{\partial x_k} \right) \right). \tag{22}$$

The first term of (22) can be computed similarly to (21), where the spatial Hessian $\left((H_i(\mathbf{x}))\right)_{lk} \approx \frac{\partial^2 \phi_i(\mathbf{x})}{\partial x_l \partial x_k}$ is computed with a FE method offline. To avoid an explicit online assembly of the Hessian $\frac{\partial^2 \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x}) \phi_j(\mathbf{x})}$ in the second term and realizing that AD treats the FE terms $\left(\mathbf{g_i}(\mathbf{x})\right)_l$ as constant, we rewrite (22) as:

$$\frac{\partial^2 \hat{u}}{\partial x_l \partial x_k} = \sum_{i=1}^{n_\Phi} \left(\underbrace{\frac{\partial \hat{u}(\Phi(\mathbf{x}))}{\partial \phi_i(\mathbf{x})}}_{\text{AD}} \left((H_i(\mathbf{x}))\right)_{lk} + \sum_{j=1}^{n_\Phi} \left(\underbrace{\frac{\partial \left(\frac{\partial \hat{u}(\Phi(\mathbf{x}))}{\partial x_l} \right)}{\partial \phi_j(\mathbf{x})}}_{\text{AD}} \left(\mathbf{g_i}(\mathbf{x})\right)_k \right) \right), \tag{23}$$

which reduces the number of AD calls from $O(n_\Phi^2)$ to $O(n_\Phi)$ (albeit nested) calls. Similar nesting strategies can be exploited for higher order derivatives. We stress that using a mesh-based approach does not restrict our function evaluation to the nodes of the mesh: FE methods allow for solution evaluation at any point in the spatial domain. In practice, we compute the eigenfunctions and their derivatives at a pre-selected number of collocation points during the offline phase and then subsample this set at the desired resolution in the online phase.

3.4. An illustrative example

We shortly illustrate the benefits of the harmonic feature embeddings on a 2D example. Consider the static PDE

$$\begin{aligned} &\nabla \cdot (a(\mathbf{x}) \nabla u) = 1, \\ &u(\mathbf{x}) = 0 \quad \forall \mathbf{x} \in \partial\Omega_{DC}, \\ &a(\mathbf{x}) = \exp(-(x-0.25)^2 - (y-0.25)^2), \end{aligned} \tag{24}$$

on the 2D unit square with a hole. We solve the PDE with a NN without an embedding layer, an embedding layer with 10 Fourier features ("FF") and an embedding layer with 10 harmonic features ("HF"). The NN has 4 layers with 10 hidden units each ($n_\theta = 3760$), uses a tanh activation function and is trained on the PDE residual (and boundary condition loss) with ADAM.

In Fig. 2, we visualize the mesh and the computed embeddings. In Fig. 3, we visualize the solution and convergence of the relative L_2 error

$$\varepsilon = \frac{||\hat{u}_{\text{NN}} - \hat{u}_{\text{FE}}||_2}{||\hat{u}_{\text{FE}}||_2}. \tag{25}$$

We also provide the projection error of the embedding as a reference, i.e. we compute the error

$$\varepsilon_{\text{proj}} = ||\mathbf{\Phi}\mathbf{\Phi}^\top \hat{u}_{\text{FE}} - \hat{u}_{\text{FE}}||_2 \tag{26}$$

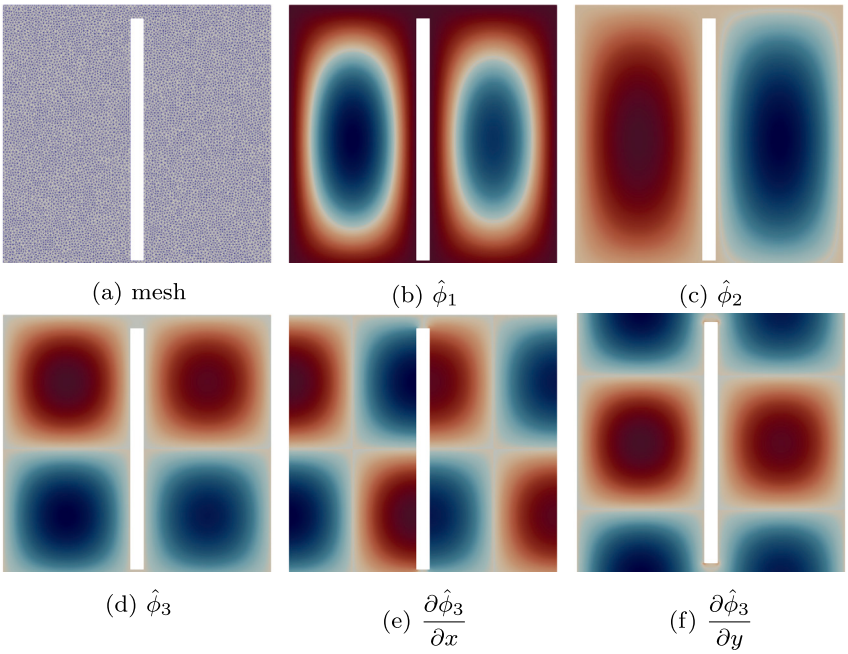


Fig. 2. Variational harmonic features computed with a FE method for the static PDE example.

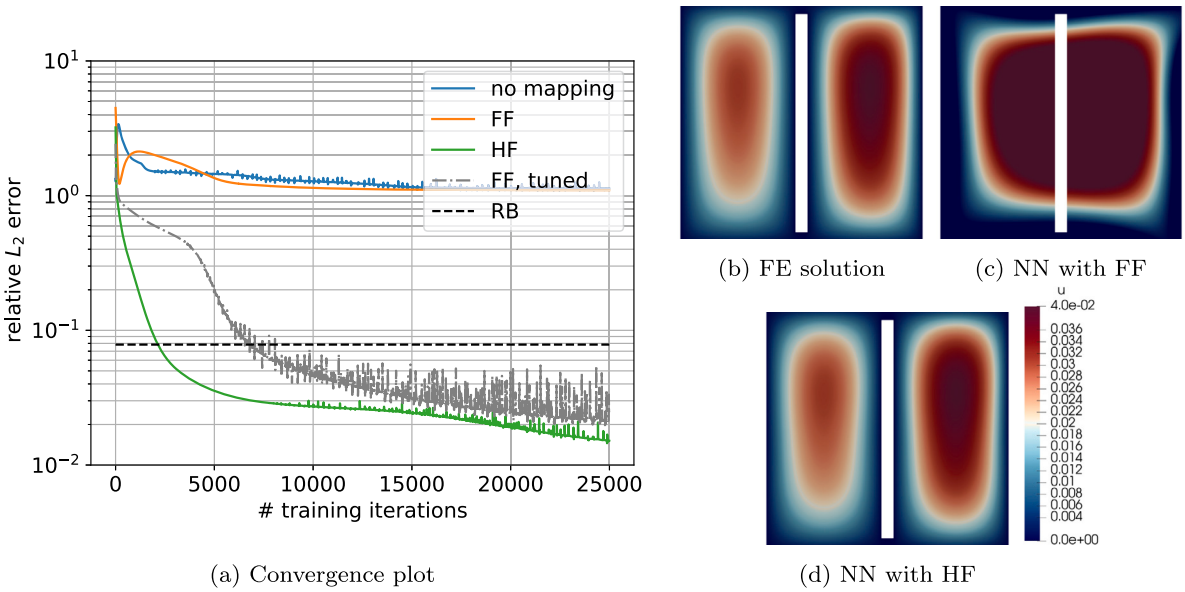


Fig. 3. Solution plots and error for the static PDE example.

to confirm that the use of a neural network enriches the solution manifold compared to the linear subspace. We observe that variational harmonic features have the fastest convergence and achieve the lowest errors. As expected, the NN without the input mapping fails to converge to the true solution. Notably, the same happens for the NN with Fourier features, unless one carefully balances the weights in the loss term ("tuned" case). These results demonstrate that the training process is a main challenge for obtaining good solutions.

4. Time-stepping with neural networks

In this section, we derive the time stepping scheme. We drop the explicit parameter dependence of the problem for conciseness: without loss of generality we can write $\tilde{\mathbf{x}} = [\mathbf{x}, \boldsymbol{\alpha}]$, $\Omega = \Omega_x \times \Omega_\alpha$, and continue the derivation with respect to the combined vector $\tilde{\mathbf{x}}$.

4.1. Deriving an explicit update equation

To extract the temporal dependence of the neural network on the NN parameter vector $\boldsymbol{\theta}$, we apply the chain rule

$$\frac{\partial \hat{u}(\tilde{\mathbf{x}};\boldsymbol{\theta}(t))}{\partial t} = \underbrace{\frac{\partial \hat{u}(\tilde{\mathbf{x}};\boldsymbol{\theta}(t))}{\partial \boldsymbol{\theta}}}_{J(\tilde{\mathbf{x}})} \cdot \frac{\partial \boldsymbol{\theta}}{\partial t} = J(\tilde{\mathbf{x}})\dot{\boldsymbol{\theta}}, \quad \forall \tilde{\mathbf{x}} \in \Omega. \tag{27}$$

Simply speaking, we seek the time derivative $\dot{\boldsymbol{\theta}}$, such that the PDE dynamics (1) are observed at every collocation point $\tilde{\mathbf{x}}$ of the computational domain. More formally, at each time t we find $\dot{\boldsymbol{\theta}}$ as the solution to the minimization problem

$$\dot{\boldsymbol{\theta}} = \text{argmin}_{\gamma} \underbrace{\frac{1}{2}\int_{\Omega} \left(J(\tilde{\mathbf{x}})\boldsymbol{\gamma} - f\left(\hat{u}(\tilde{\mathbf{x}};\boldsymbol{\theta}(t)), t, \tilde{\mathbf{x}}\right)\right)^2 \mathrm{d}\tilde{\mathbf{x}}}_{g(\gamma)}. \tag{28}$$

Applying the first order optimality condition $\nabla_{\gamma} g(\boldsymbol{\gamma}) = 0$, yields an equation of linear operators to be solved for the parameter update:

$$M\boldsymbol{\gamma}^* = F, \tag{29}$$

where the matrix operator M and vector operator F are given by the inner products:

$$\begin{aligned} M &= \int_{\Omega} J(\tilde{\mathbf{x}})^{\top} \cdot J(\tilde{\mathbf{x}}) \mathrm{d}\tilde{\mathbf{x}}, \\ F &= \int_{\Omega} J(\tilde{\mathbf{x}})^{\top} \cdot f\left(\hat{u}(\tilde{\mathbf{x}};\boldsymbol{\theta}(t)), t, \tilde{\mathbf{x}}\right) \mathrm{d}\tilde{\mathbf{x}}. \end{aligned} \tag{30}$$

We refer the reader to [19] for a more thorough discussion of the theoretical background of this optimization problem and its relation to the Dirac-Frenkel variational principle. In practice, M and F are approximated by randomly sampling n_x collocation points, to obtain a Monte Carlo approximation of the integrals and a finite dimensional linear system

$$M \approx \frac{1}{n_x}\mathbf{J}^{\top}\mathbf{J}, \quad F \approx \frac{1}{n_x}\mathbf{J}^{\top}\mathbf{f}, \tag{31}$$

$$\mathbf{J}^{\top}\mathbf{J}\gamma^* = \mathbf{J}^{T}\mathbf{f}, \tag{32}$$

where $\mathbf{J}$ is the discrete neural network Jacobian $(\mathbf{J})_{ik} = \dfrac{\hat{u}(\tilde{\mathbf{x}}_i;\boldsymbol{\theta}(t))}{\partial \theta_k}$ and $\mathbf{f}$ is vector of right hand side evaluations of the differential operator at the collocation points $(\mathbf{f})_i = f\left(\hat{u}(\tilde{\mathbf{x}}_i;\boldsymbol{\theta}(t)), t, \tilde{\mathbf{x}}_i\right)$.

We remark that there is no a priori guarantee that the linear system (32) has a unique solution as $\mathbf{J}^{\top}\mathbf{J}$ may not be invertible. In such a case, we seek the solution of minimum norm, i.e. the smallest possible update to $\boldsymbol{\theta}$. We can thus express the time derivative of the parameters via the Moore-Penrose pseudo-inverse

$$\dot{\boldsymbol{\theta}} \approx \boldsymbol{\gamma}^* = (\mathbf{J}^{\top}\mathbf{J})^{\dagger}\mathbf{J}^{T}\mathbf{f}. \tag{33}$$

Equation (33) can also be interpreted geometrically - the evolution update equation is a projection of the PDE right-hand side operator onto the tangent space of the neural network. A successful EDNN model thus needs to

- accurately approximate the true solution trajectory via the output of the neural network,
- have a rich enough tangent space along this solution trajectory, such that the PDE dynamics can be accurately represented in the update equation.

4.2. Efficient solution of the parameter update equation

In practice, direct inversion of the linear system (32) with the generally dense matrix $\mathbf{J}^{\top}\mathbf{J}$ is costly, especially if the Moore-Penrose pseudo-inverse is computed. We avoid direct assembly of the neural network Jacobian $\mathbf{J}$, as it requires n_{θ} derivative evaluations of size n_x. Instead, we propose the use of a Krylov solver, which only needs Jacobian-vector products:
To compute the action $\mathbf{w} = \mathbf{J}^{\top}\mathbf{J}\boldsymbol{\gamma}$, we first compute $\mathbf{v} = \mathbf{J}\boldsymbol{\gamma}$ and then evaluate $\mathbf{w} = \mathbf{J}^{\top}\mathbf{v}$ using automatic differentiation.

Noting that (32) is symmetric, MINRES [26] would be an appropriate solver choice. It is algebraically equivalent and more numerically stable, especially for ill-conditioned $\mathbf{J}$, to directly apply LSMR [27] to the following least squares problem:

$$\min_{\gamma} ||\mathbf{J}\boldsymbol{\gamma} - \mathbf{f}||_2^2, \tag{34}$$

which can be seen as a discrete version of the minimization problem in (28).

When the least-squares problem in (34) has multiple solutions, LSMR actually solves the problem

$$\begin{aligned} &\dot{\boldsymbol{\theta}} = \text{argmin}_{\boldsymbol{\gamma}} ||\boldsymbol{\gamma}||_2^2, \\ &\text{subject to } \mathbf{J}\boldsymbol{\gamma} = \mathbf{f}. \end{aligned} \tag{35}$$

Therefore, LSMR is guaranteed to return the solution of minimum norm. Additionally the norm $||\boldsymbol{\gamma}||$ of the current solution iterate $\boldsymbol{\gamma}$ has numerically been shown to increase monotonically. LSMR is also preferable over the more common LSQR solver as it directly minimizes the residual of (32), $||\mathbf{J}^\top\mathbf{J}\boldsymbol{\gamma} - \mathbf{J}^\top\mathbf{f}||_2^2$, in every step, allowing for faster termination of the procedure. We can then treat underdetermined problems, where the number of NN parameters n_θ exceeds the number of spatial collocation points n_x, identically to the overdetermined case, hence decoupling the spatial resolution from the modeling capabilities of the neural network. With this problem setup, the computational cost of a single update step is dominated by the repeated calls of forward ($\mathbf{v} = \mathbf{J}\boldsymbol{\gamma}$) and backwards ($\mathbf{w} = \mathbf{J}^\top\mathbf{v}$) automatic differentiation, we thus expect approximately linear scaling of computation times in both the number of parameters and collocation points ($O(n_x n_\theta)$).

Remark. Numerically, we terminate the LSMR solve at a finite precision tolerance, such that (32) is not solved exactly even when a compatible solution exists. This can be considered as a form of regularization as LSMR will produce parameter updates with smaller norm for higher error tolerances. It is also possible to explicitly add a regularization term to the least squares problem, as explored in [20], which can lead to further speed-ups.

4.3. Overview time stepping procedure

In this section, we summarize the time stepping procedure for the neural network parameters. As a first step, we need to ensure that at time $t = 0$, the neural network output $\hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}(0))$ reproduces the initial condition $u_0(\tilde{\mathbf{x}})$. In [18,19], this is achieved by training on the residual of the initial condition at a finite number of collocation points $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_\mathbf{1}, \tilde{\mathbf{x}}_\mathbf{2}, ..., \tilde{\mathbf{x}}_{\mathbf{n}_{\text{train}}}]$:

$$\boldsymbol{\theta}_0 = \text{argmin}_{\boldsymbol{\gamma}} ||\hat{u}(\tilde{\mathbf{X}}; \boldsymbol{\gamma}) - u_0(\tilde{\mathbf{X}})||_2^2, \tag{36}$$

which may require a large number of epochs until a sufficient accuracy is reached. We propose an additional "trainingfree" approach, where the neural network output is modified to satisfy the initial condition by construction

$$\tilde{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}(t)) = u_0(\tilde{\mathbf{x}}) + \hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}(t)) - \hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}(0)), \tag{37}$$

which allows skipping the training step, when $u_0(\tilde{\mathbf{x}})$ is available analytically. This comes at the cost of doubling the number of neural network evaluations, which consequently also increases the cost of evaluating the differential operator and the Jacobian.
For the time integration itself, it is possible to rely on established ODE solvers. We consider Runge-Kutta methods which allow flexible adaptation of the time step, specifically the forward Euler scheme and the 5(4) stage method by Tsitouras (Tsit5) [28]. The procedure is summarized in Algorithm 2.

Algorithm 2 Time integration of PDEs with neural networks.

Input: Neural network $\hat{u}$, PDE rhs f, initial condition u_0, collocation points $\tilde{\mathbf{X}}_\mathbf{c}$, RK method with stage weights a, b, c, stopping tolerance `tol` for `LSMR`.
Output: PDE solution $\tilde{u}$ at test locations $\tilde{\mathbf{X}}_{\text{test}}$ at the final computation time T.

Randomly initialize $\boldsymbol{\theta}_0$.
$\boldsymbol{\theta}_n \leftarrow \boldsymbol{\theta}_0$
if trainingfree **then**
  Assign NN: $\tilde{u} \leftarrow u_0(\tilde{\mathbf{x}}) + \hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}_n) - \hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}_0)$
else
  Train $\boldsymbol{\theta}_n$ to fit the initial condition according to (36).
  Assign NN: $\tilde{u} \leftarrow \hat{u}(\tilde{\mathbf{x}}; \boldsymbol{\theta}_n)$
end if
$t \leftarrow 0$
while $t < T$ **do**
  for each RK stage i **do**
    $t_{\text{loc}} \leftarrow t + c_i h$
    $\boldsymbol{\theta}_{\text{loc}} \leftarrow \boldsymbol{\theta}_n + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{\gamma}_j$
    $\mathbf{f} \leftarrow f\left(\tilde{u}(\tilde{\mathbf{X}}_\mathbf{c}; \boldsymbol{\theta}_{\text{loc}}), t_{\text{loc}}, \tilde{\mathbf{X}}_\mathbf{c}\right)$
    $\mathbf{J} \leftarrow \dfrac{\tilde{u}(\tilde{\mathbf{X}}_\mathbf{c}; \boldsymbol{\theta}_{\text{loc}})}{\partial \boldsymbol{\theta}_{\text{loc}}}$ ▷ Initialize Jacobian action.
    $\boldsymbol{\gamma}_i \leftarrow \texttt{LSMR}(\mathbf{J}, \mathbf{f}, \texttt{tol})$
  end for
  $\Delta\boldsymbol{\theta} \leftarrow h \sum_{i=1}^{s} b_i \boldsymbol{\gamma}_i$ ▷ Error control and time step adaptation may happen here.
  $t \leftarrow t + h$
  $\boldsymbol{\theta}_n \leftarrow \boldsymbol{\theta}_n + \Delta\boldsymbol{\theta}$
end while
Evaluate solution $\tilde{u}(\tilde{\mathbf{X}}_{\text{test}}, \boldsymbol{\theta}_n)$.

4.4. Dealing with stiffness

The parameter update equation (33) that we have derived is explicit in nature. This severely limits the time-step choice as some physical systems have natural stiffness, resulting in the time-stepping of the EDNN being expensive. In [19], an implicit Euler scheme is proposed. This comes at significant computational cost as second order derivatives of the neural network with respect to its parameters $\boldsymbol{\theta}$ need to be computed repeatedly in an iterative training procedure. Here, we propose a "linearly implicit" scheme based on Rosenbrock methods. Similarly to the previous application of the chain rule, we note that at first order we can write

$$\begin{aligned} u(\theta_{n+1}) &= u(\theta_n + \Delta\theta) \approx u(\theta_n) + J_u \Delta\theta, \\ f(\theta_{n+1}) &= f(\theta_n + \Delta\theta) \approx f(\theta_n) + J_f \Delta\theta \end{aligned} \tag{38}$$

where J_u is the previously derived neural network Jacobian $\mathbf{J}$ and

$$J_f = \frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial u} J_u \tag{39}$$

is the Jacobian of the right hand side. We consider the classic Rosenbrock triple with time step h and parameters $\gamma = \frac{1}{\sqrt{2}+2}$, $e_{32} = 6 + \sqrt{2}$. To integrate from u_0 to u_1, we compute

$$\begin{aligned} (I - h\gamma \frac{\partial f}{\partial u})k_1 &= f(u_0), \\ (I - h\gamma \frac{\partial f}{\partial u})k_2 &= f(u_0 + \frac{1}{2}\gamma k_1) - h\gamma \frac{\partial f}{\partial u} k_1, \\ u_1 &= u_0 + hk_2, \end{aligned} \tag{40}$$

and evaluate the third stage for error control with error ε as

$$\begin{aligned} (I - h\gamma \frac{\partial f}{\partial u})k_3 &= f(u_1) - e_{32}\left(k_2 - f(u_0 + \frac{1}{2}\gamma k_1)\right) - 2\left(k_1 - f(u_0)\right), \\ \varepsilon &= \frac{h}{6}(k_1 - 2k_2 + k_3). \end{aligned} \tag{41}$$

We insert $k_i \approx J_u^{(i)} \kappa_i$, to obtain

$$\begin{aligned} (J_u^{(1)} - h\gamma J_f)\kappa_1 &= f(u(\theta_0)), \\ (J_u^{(2)} - h\gamma J_f)\kappa_2 &= f(u(\theta_0 + \frac{1}{2}\gamma\kappa_1)) - h\gamma J_f \kappa_1, \\ \theta_1 &= \theta_0 + h\kappa_2. \end{aligned} \tag{42}$$

We solve the first two lines of (42) in a collocation sense with LSMR, i.e. we first compute

$$\min_{\kappa_1} ||(J_u^{(1)} - h\gamma J_f)\kappa_1 - f(u(\theta_0))||_2, \tag{43}$$

and then

$$\min_{\kappa_2} ||(J_u^{(2)} - h\gamma J_f)\kappa_2 - f(u(\theta_0 + \frac{1}{2}\gamma\kappa_1) + h\gamma J_f \kappa_1||_2 \tag{44}$$

We then find the final update in (42). For error control, we proceed similarly for the third stage of the Rosenbrock triple and error metric:

$$\begin{aligned} (J_u^{(3)} - h\gamma J_f)\kappa_3 &= f(\theta_1) - e_{32}\left(J_u^{(2)}\kappa_2 - f(\theta_0 + \frac{1}{2}\gamma\kappa_1)\right) - 2\left(J_u^{(1)}\kappa_1 - f(u_0)\right), \\ \varepsilon &= \frac{h}{6}(\kappa_1 - 2\kappa_2 + \kappa_3), \end{aligned} \tag{45}$$

which requires one additional LSMR solve as in (44). All remaining computations follow the usual ODE-based time-stepping described in Algorithm 2.

Remark. Rosenbrock methods usually come at severely increased computational cost as they need to form the Jacobian, which is the price to pay for using a larger time step. In this setting, we need to form the costly Jacobian action of the neural network anyway, so the cost increase of forming the Jacobian of the right hand side is limited.

4.5. Active sampling of collocation points

We already indicated that the full resolution of the FE mesh may not be required for the time-stepping procedure. Subsampling is especially beneficial when only part of the computational domain contains relevant information about the PDE dynamics, and we

can concentrate collocation points in relevant areas. In [19], the authors propose to dynamically adjust the collocation points based on structural properties of the PDE, an alternative approach is also discussed in [29].

Inspired by this, we propose an ad-hoc sampling strategy, which can be applied to any PDE: we choose an indicator criterion, e.g. the magnitude of the right-hand-side

$$\omega(\tilde{\mathbf{x}}_i) = \omega_i = |f(\tilde{\mathbf{x}}_i; \boldsymbol{\theta})|$$

which informs us whether a collocation point $\tilde{\mathbf{x}}_i$ contributes to the PDE dynamics residual. As evaluating the neural network is cheap compared to solving the update equation, we can do so on a large set of n candidate points. The vertices of the full resolution FE mesh, which discretizes Ω_x, are an obvious choice for a candidate set of spatial points S_{FE}. Uniformly sampling a point $x_i \in S_{\text{FE}}$, then ensures that we are sampling the computational domain consistently. In the parametric case, we uniformly sample a parameter value $\boldsymbol{\alpha}_i \in \Omega_\alpha$ for each sampled spatial point to form $\tilde{\mathbf{x}}_i = [\mathbf{x}_i, \boldsymbol{\alpha}_i]$ and then evaluate the indicator criterion $\omega(\tilde{\mathbf{x}}_i)$. The actual collocation points are sampled according to a discrete distribution on the candidate set with normalized probabilities

$$p(\tilde{\mathbf{x}}_i) = \frac{\omega_i}{\sum_{i=1}^{n_s} \omega_i} \tag{46}$$

ω_i for each point. The procedure for sampling from a spatial-parametric domain $\Omega_x \times \Omega_\alpha$ is summarized in Algorithm 3 and can lead to a significant speedup for problems with localized features.

Algorithm 3 Active sampling strategy.

Input: Set of spatial points $S_{\text{FE}} = \{\mathbf{x}_k\}_{k=1}^{n_s}$, parameter domain Ω_α, sampling criterion $\tilde{f}(\tilde{\mathbf{x}})$, size of candidate set n.
Output: At each time step: Set of collocation points S_c of size n_c.

```
Compute candidate point set S = {x̃_i}_{i=1}^n:
for i = 1 to n do
    Uniformly sample x_i ∈ S_FE and independently α_i ∈ Ω_α to form x̃_i = [x_i, α_i].
end for
for each time step t_j do
    for i = 1 to n do
        Evaluate ω_i = f̃(x̃_i, θ(t_j))
    end for
    Compute probability weights p_i = ω_i / Σ_{i=1}^{n_s} ω_i to form discrete probability distribution P(θ(t_j)) : P(X = x̃_i) = p_i.
    Sample n_c points X ~ P(θ(t_j)) to form S_c.
end for
```

5. Numerical examples

We first highlight key components of the developed methods in two 1D cases. On the two-soliton solution of the Korteweg-de-Vries (kdV) equation, we demonstrate that a positional encoding with two basis functions can be sufficient to solve a transport dominated problem with high accuracy. A nonlinear heat equation with a parameterized initial condition then show-cases how time-stepping can be used for many-query applications. We then focus on advection-diffusion problems on a square domain with and without holes, to show the flexibility over a variety of geometries.

5.1. Implementation and experiment settings

All FE calculations are performed with FEnics [30]. The time stepping scheme is implemented in Julia and builds on several packages: We use `Flux.jl` [31] for the NN presentation, `Zygote.jl` [32], `TaylorSeries.jl` [33] and `ForwardDiff.jl` [34] for automatic forward and backward differentiation and `DifferentialEquations.jl` [35] for the explicit Runge Kutte ODE solvers. We implement the Rosenbrock method and its adaptive time stepping scheme, but use the same PI controller settings as `DifferentialEquations.jl`. To compute the Jacobian of the right hand-side J_u, required for the Rosenbrock method, we use finite differences as the available AD packages do not succeed in this case. We use `TaylorSeries.jl` to ascertain that the approximation error of this step is below 1e-5. We further use the LSMR implementation of `IterativeSolvers.jl`, with termination tolerances set to "atol/btol = 5e-5" unless specified otherwise. All numerical experiments are conducted on an iMac desktop (3.1 GHz 6-Core Intel i5, 16 GB RAM) without GPU acceleration.

We use the same prototype of fully-connected neural network architectures for all problems, where we only vary the input embedding and size and number of hidden layers. We omit the bias on the last layer as it cancels out when Dirichlet BCs are enforced and we did not observe degraded performance for the other cases. We report the relative error as defined in (25) at each time instance. For the KdV equation, an analytic solution is used. In the other examples we treat the FE solution as the ground truth. In the parametrized setting, we compute the average L_2 -error over a grid of the parametric domain with N points as:

$$\varepsilon = \frac{1}{N} \sum_{i=1}^{N} \frac{||\hat{u}_{\text{NN}}(\boldsymbol{\alpha}_i) - \hat{u}_{\text{FE}}(\boldsymbol{\alpha}_i)||_2}{||\hat{u}_{\text{FE}}(\boldsymbol{\alpha}_i)||_2}. \tag{47}$$

Another useful metric is the mean solution deviation δ as it measures how much the solution varies across the parameter space. It is computed on the ground truth as

$$\begin{aligned} \mu &= \frac{1}{N}\sum_{i=1}^{N} \hat{u}_{\mathrm{FE}}(\boldsymbol{\alpha}_i) \\ \delta &= \frac{1}{N}\sum_{i=1}^{N} \frac{||\hat{u}_{\mathrm{FE}}(\boldsymbol{\alpha}_i) - \mu||_2}{||\mu||_2} \end{aligned} \tag{48}$$

5.2. Korteweg-de Vries equation

As a first test case, we consider the Korteweg-de Vries equation:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\frac{\partial^3 u}{\partial x^3} - 6u\frac{\partial u}{\partial x}, \quad x \in [-20, 20] \\ u(-20, t) &= u(20, t). \end{aligned} \tag{49}$$

We choose an initial condition that leads to a traveling two-soliton solution as described in [36], see Appendix A for the exact construction of the initial condition and reference solution. We use a NN with a simple periodic embedding

$$\Phi = [\cos(\frac{2\pi}{L}), \sin(\frac{2\pi}{L})],$$

where $L = 40$ is the size of the domain, and vary the number of layers n_l and number of hidden units n_n. The computational domain is discretized by randomly sampling $n_x = 1000$ points. We train each architecture to the initial condition on 5000 randomly sampled points for 100000 iterations. The same trained neural network is then used for time-stepping with a fixed time step Euler scheme, Tsit5 with automatic error control and the proposed Rosenbrock 2-step method.

The results in Fig. 4 show that we visually approximate the solution well. At the end of the simulation time, we observe small oscillation errors for the explicit time stepping methods. This is also reflected by the error plots in Fig. 5. The explicit Euler scheme with a time step of t = 0.001 fails to capture the solution trajectory accurately, while all other time stepping schemes produce results with 1-3% error. Even more strikingly, the Rosenbrock method produces the lowest error with the largest time step. Increasing the depth of the neural network as compared to the width seems to produce better results, but we observe no clear trends, as long as the neural network is large enough. In conclusion, we observe that the neural network indeed can reproduce transport behavior when using static positional embeddings and the modified Rosenbrock method alleviates the stiffness problem of the explicit solvers on this example.

5.3. A parametrized nonlinear heat equation

In this example, we consider a heat equation with a cubic nonlinearity and a parametrized initial condition with Dirichlet boundary conditions.

$$\begin{aligned} &\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - 16u^3, \\ &u(0, t) = u(1, t) = 1, \\ &u(x, 0; \alpha_1, \alpha_2) = 1 + \alpha_1 \sin(\pi x) + \alpha_2 \sin(3\pi x), \\ &\alpha_1, \alpha_2 \in [-0.5, 0.5]. \end{aligned} \tag{50}$$

We consider two different positional embeddings

$$\Phi_2 = [\sin(\pi x), \sin(2\pi x)]$$

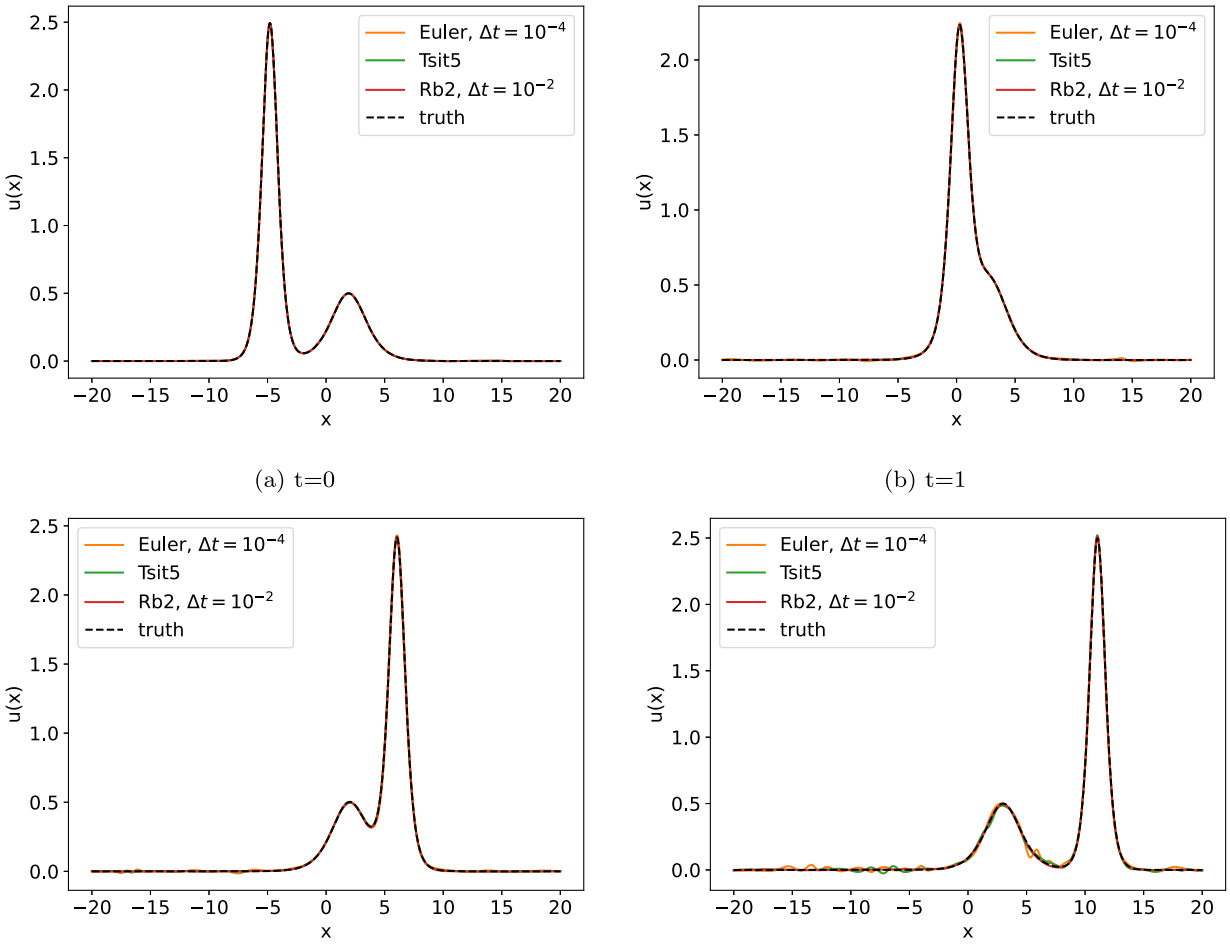
$$\Phi_4 = [\sin(\pi x), \sin(2\pi x), \sin(3\pi x), \sin(4\pi x)],$$

corresponding to the first eigenfunctions of the Laplace problem with Dirichlet boundary conditions. Choosing a constant lifting function $u_{\mathrm{l}}(x, t) = 1$, we model the PDE solution as

$$u(x, t) = u_{\mathrm{NN}}(\Phi_j(x), \boldsymbol{\alpha}; \theta(t)) - u_{\mathrm{NN}}(\mathbf{0}, \boldsymbol{\alpha}; \theta(t)) + u_{\mathrm{l}}(x, t) \tag{51}$$

We point out that the initial condition does not lie in the span of Φ_2 and that the richness of solutions across the parameter space decreases over time as all initial conditions converge to the same steady state solution. We use neural networks with 4 layers and 10 or 20 hidden units each and compare training to the initial condition (10000 points for 40000 iterations with ADAM) and the trainingfree scheme. The reference solutions are computed with a FE method on an equidistant grid of 1000 elements with Lagrange P2 elements, using implicit Euler for the time stepping with a step size of $\Delta t = 10^{-4}$.

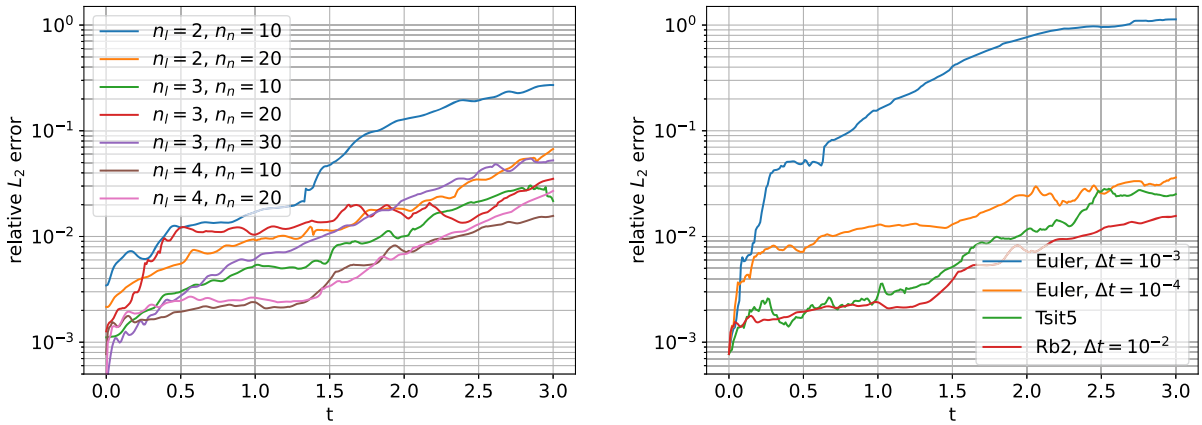
The plots in Fig. 6 show that we capture the solution well across a range of different parameter values and there is no visual difference to the FE reference solution. We compute the mean relative error across a grid of parameters with 121 points, shown



(a) t=0 (b) t=1

(c) t=2 (d) t=3

Fig. 4. Time evolution of the Korteweg-de Vries equation.



(a) Performance of the Rosenbrock (Rb2) method with $\Delta t = 0.01$ for different architectures.

(b) Performance of time steppers for a NN with $n_l = 4$, $n_n = 10$.

Fig. 5. Evolution of error for the Korteweg-de Vries equation.

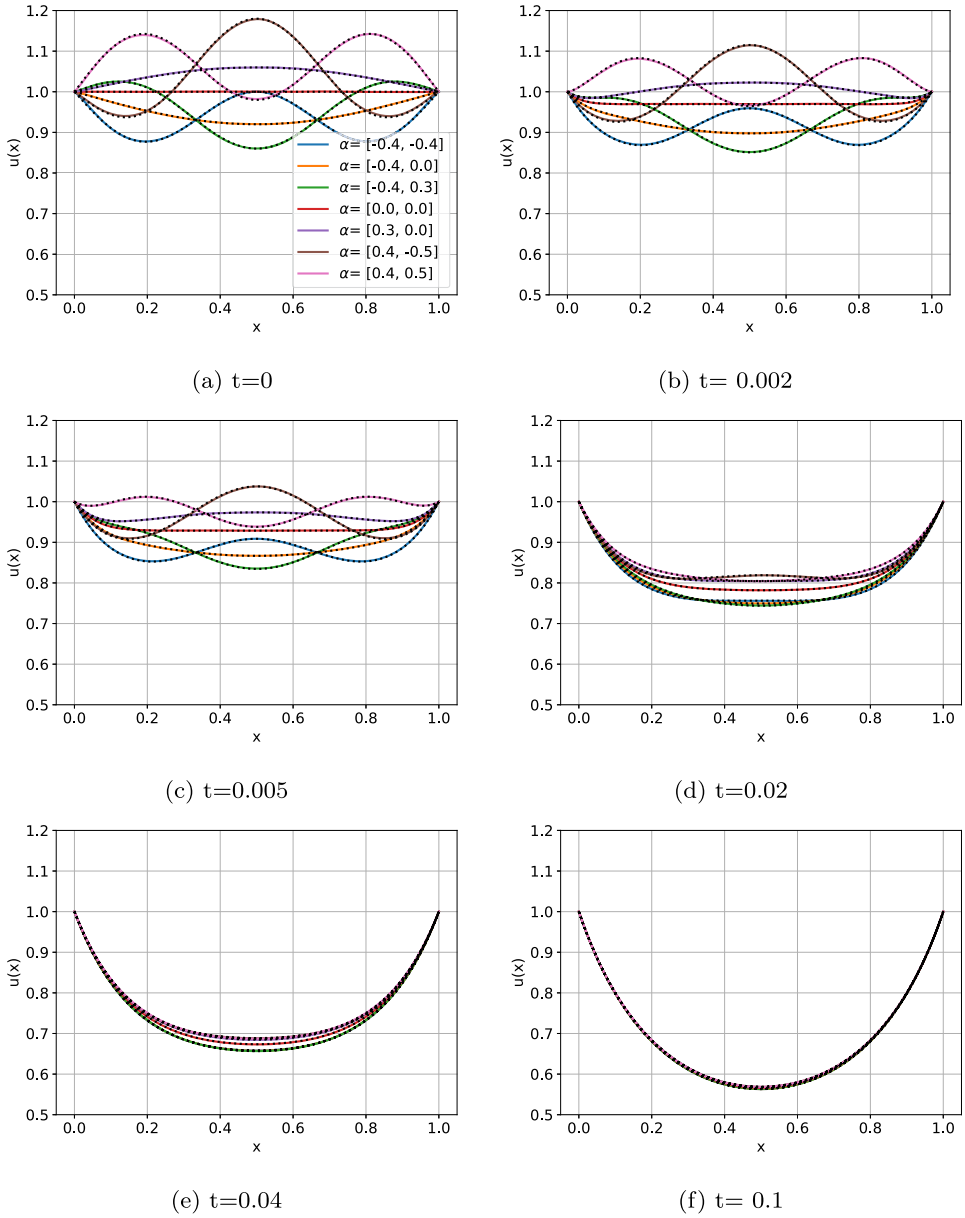


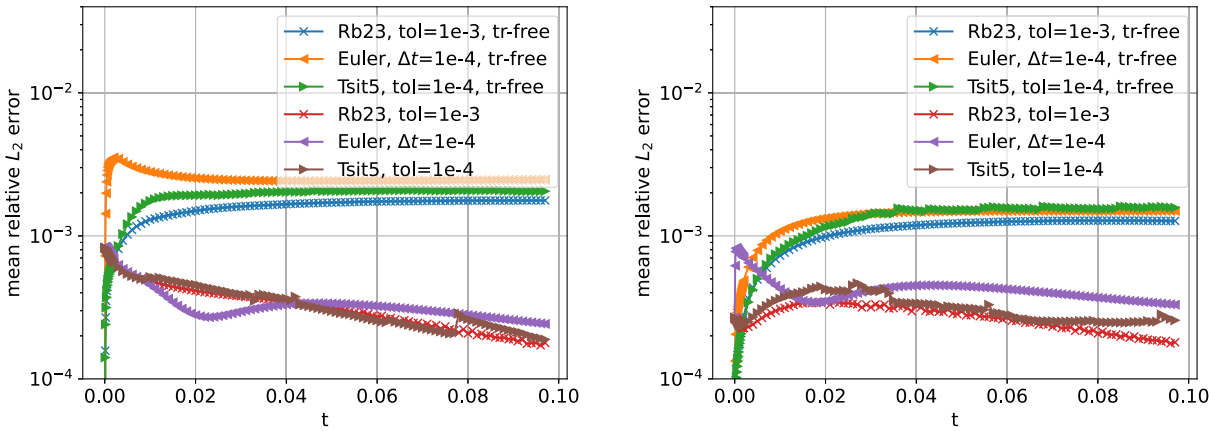
Fig. 6. Time evolution of the parametrized heat equation for different parameter values. The black-dotted points indicate the reference FE solution.

in Fig. 7. Comparing the two embeddings in (a) and (b) of Fig. 7, we observe negligible differences which demonstrates that we can leverage the nonlinearity of the neural network to fit functions that are not contained in the linear span of the embedding. We also confirm that for this simple heat equation example, the "trainingfree" approach produces acceptable errors, which are however higher compared to the neural networks which were trained on the initial condition.

We also note that the Rosenbrock method with adaptive time-stepping produces the lowest error. The tolerance of the time step adaptation was chosen one order of magnitude lower for Tsit5 to produce comparable results. We conjecture that this is due to the additional error sources in the neural network update equation: while error control is still possible qualitatively, not all quantitative rules carry over directly from traditional Runge-Kutta methods and the error estimate in the parameter update may not always be directly indicative of the error in the physical solution.

5.4. Parametrized nonlinear advection-diffusion problem

We consider a nonlinear advection-diffusion equation



(a) NN: Φ_2, $n_{\text{hid}} = 10$. (b) NN: Φ_4, $n_{\text{hid}} = 20$.

Fig. 7. Evolution of error for the parametrized heat equation in 1D. "tr-free" refers to the trainingfree approach (37).

$$\begin{aligned}
&\frac{\partial u}{\partial t} = 0.1\nabla \cdot \left(\left(1+\alpha_1 \sin(2\pi x_1)\right)\nabla u\right) + 4\begin{bmatrix}\cos(\pi\alpha_2)u\\ \sin(\pi\alpha_2)u\end{bmatrix}\cdot \nabla u\\
&\nabla u(\mathbf{x},t)\cdot \mathbf{n} = 0 \quad \text{for } x \in \partial\Omega\\
&u(\mathbf{x},0) = \sin^2(\pi x_1)\sin^2(\pi x_2)\\
&\alpha_1,\alpha_2 \in [-0.5, 0.5],
\end{aligned} \tag{52}$$

with the physical domain being the unit square. As this problem uses Neumann boundary conditions, we consider the first three eigenfunctions of the Neumann case for the Laplace operator for the positional embedding:

$$\Phi = [\cos(\pi x_1), \cos(\pi x_2), \cos(\pi x_1)\cos(\pi x_2)],$$

where we have dropped the constant eigenfunction $\phi_0 = 1$, as it does not enrich the function space - the bias term of the first layer should be able to model any necessary constant terms.

In this example, all solutions start from the same initial condition and then evolve according to the parametric dependence, i.e. we expect the solution space to become richer/more diverse as time increases. Here α_1 controls the variation in the diffusive term, while α_2 controls the direction of the nonlinear advective term. We train for the initial condition on 10000 randomly sampled points in the domain for 40000 iterations and compare again with the trainingfree approach, which appears especially appealing here: as the initial condition does not depend on the parameters, we train the neural network to produce identical values across the whole parameter space.

In Fig. 8, we compare the error of the two approaches for neural networks with 4 layers with different number of hidden units n_n and sampling points n_x. We plot the mean solution deviation introduced in (48) to demonstrate how the solution space becomes richer over time. We note that for both the trained and training-free approach the error increases until t= 0.02. From that point on, the error for the trained neural network increases substantially less than for the trainingfree approach. We conclude that for the best possible performance, training to the initial condition is preferable, while the trainingfree approach may be an acceptable time saving measure for short time spans. We also note that there is only minimal decrease in error when using $n_x = 10000$ vs. $n_x = 5000$ integration points, which indicates that we have sufficiently sampled the solution space.

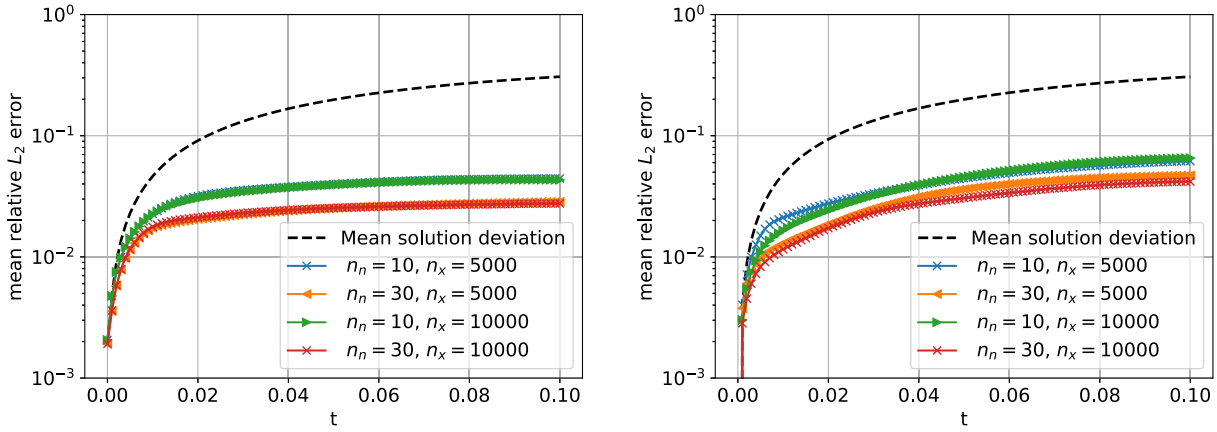
It is noteworthy that when the error stabilizes around t= 0.02, there is hardly any visual difference between the different parameter values (Fig. 9), whereas the difference is clear at the final integration time (Fig. 10). This shows that the neural network and its tangent space are rich enough to handle increasingly different parametric solutions without a blow-up of the error.

5.5. Advection-diffusion on a domain with holes

5.5.1. General setup

We consider an advection-dominated equation on the unit squares with two circular holes:

$$\begin{aligned}
&\frac{\partial u}{\partial t} = 0.001\nabla^2 u + \boldsymbol{w}(\mathbf{x})\cdot\nabla u,\\
&u(\mathbf{x},0) = \operatorname{sech}(-100((x_1 - 0.35. - 0.1\alpha_1)^2 + (x_2 - 0.7 - 0.1\alpha_2)^2))^2,\\
&\alpha_1,\alpha_2 \in [-0.5, 0.5].
\end{aligned} \tag{53}$$



(a) Trained to initial condition. (b) Training free.

Fig. 8. Evolution of error for the parametrized nonlinear advection diffusion equation.

The initial condition represents a blob of concentration that spreads according to the advective field $\boldsymbol{w}$ and the small diffusive term. We impose homogeneous Dirichlet boundary conditions on top and Neumann boundary conditions for the remaining domain boundary. The velocity field $\boldsymbol{w}(\mathbf{x})$ is shown in Fig. 11 and is precomputed numerically by solving the steady-state Navier Stokes equations, see Appendix B for details. We compute the reference solutions to (53) on the same mesh with a FE method using P2 elements and an implicit Euler method with time step $\Delta t = 1\text{e-}4$.

We first consider a single forward run and then compute the case for the parametrized initial condition. We consider positional embeddings of size $n_\Phi = 4$, $n_\Phi = 10$ and $n_\Phi = 15$, corresponding to the first n_Φ eigenfunctions of the Laplace operator which are computed numerically on the same FE mesh using P3 elements, see Fig. C.18 and Appendix C for more details. As the solution converges to 0 towards the end of the simulation, we normalize the L_2 -error with the norm of the initial condition for this experiment.

5.5.2. Single parameter case

All neural networks used in this experiment have 4 layers with 10 or 20 hidden units. We train for the initial condition for $\boldsymbol{\alpha} = [0.0, 0.0]$ on the $n_x = 6205$ nodes of the mesh for 40000 iterations. We use an Euler scheme with a time step size of $\Delta t = 10^{-3}$ to run the simulation until time t = 1, but we only visualize the solution until t = 0.6, see Fig. 14, when most of the concentration has left the domain.

In the error plots in Fig. 12 (a), we observe that using only $n_\Phi = 4$ features leads to errors larger than 10%, while using $n_\Phi = 10$ or $n_\Phi = 15$ features leads to much better approximations. Increasing the number of hidden layers only slightly reduces the error compared to increasing the number of features in the positional embedding. This is a striking difference to the cases on simpler domains, where the eigenfunctions were simple trigonometric functions.

Taking a closer look at the solution snapshots in Fig. 14, we observe that there is hardly any visual difference to the reference solution from t = 0.1 to t = 0.4, while we lose both the shape of the thin tail and the exact shape of the solution on the Neumann boundary in the last two snapshots.

The figures also show that large parts of the computational domain contain no information about the problem. We therefore aim to increase computational efficiency by sub-sampling the points on the mesh according to the size of the advective term as described in Algorithm 3. As can be seen from both the error plot in Fig. 12 and the solution (Fig. 14), the quality of the solution does not significantly degrade when only 1000 points are sampled. On the other hand, 500 points are not enough to recover the same quality of the solution, as errors are considerably higher. We remark that for the larger neural networks, the update equation is underdetermined as the number of network parameters exceeds the number of sampling points. The Krylov solver implementation can deal with this scenario without further modification. For 2000 and 1000 samples, errors are in the same range as for the full mesh. Fig. 13 shows the effect of the time step and sample size on both the error and computational times. For reference, the FE solution takes about 75 seconds to compute at a time step of $\Delta t =$ 1e-3, so is faster by a factor 6 to 12 and has a maximum L_2 error of 4.3% compared to the reference, which is similar to what we achieve with the EDNN solver. We remark that our code is not optimized for computational efficiency, but the values still show the theoretical trends: the computation time scales approximately linearly with both the number of time steps and the number of samples. In terms of error, we observe little benefit from reducing the time step size, which indicates that other factors (e.g. errors in the initial condition, approximation error in the FE embedding functions or the approximation capability of the neural network) are the dominant sources of error. With these encouraging results we turn to the parametrized problem, where we cannot afford to consider the full mesh for every parameter value.

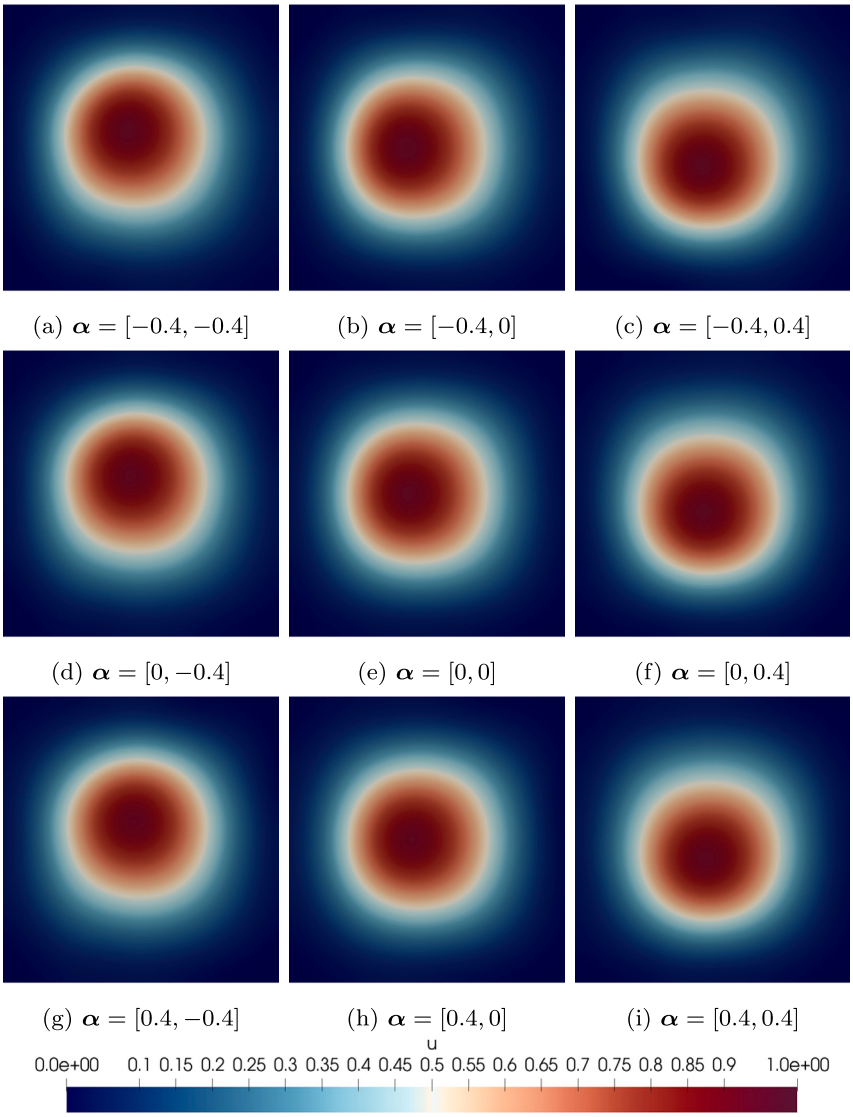


Fig. 9. Solution snapshots of the nonlinear advection diffusion problem at time t= 0.02 for different parameter values.

5.5.3. Parametrized initial condition

We use 4 layer neural networks with 20 hidden units and 10, 15 and 20 features in the positional embedding to account for the increased complexity of the solution manifold in the parametrized case. We consider the parametrized initial condition and train the neural network on 40000 points across the combined physical and parameter domain with a batch size of 1024 for 8000 epochs. We again use forward Euler with a time step size of $\Delta t = 10^{-3}$ and compute the parametrized solutions until t = 0.6.

We use the sampling algorithm for a discrete set of 40000 candidate points, for which the sampling criterion (absolute value of advective term) is evaluated. We then subsample 2000, 5000, 8000 and 10000 points respectively which are used to evaluate the update equation. In Fig. 15, one sees the solution for different parameter values and how they evolve differently according to their starting position. Considering the error plots in Fig. 16, we observe a larger error in this challenging case, however, the neural network solutions still provide valuable information. Intuitively, it appears natural that more sampling points are required for the parametrized case, which is also reflected in the increased error for e.g. 2000 sampling points when compared to the single parameter case. Increasing the sampling size to 8000 points leads to errors of 6.5% and the solutions are visually identical to the FE reference. Further increasing the sampling size did not lead to significant improvement, indicating that other sources of error dominate. In Fig. 17 we further notice the benefit of increasing the embedding size, which reduces the error and decreases the computation time. The latter might seem counterintuitive at first, as the NN with larger embedding has more parameters, however we did observe that the Krylov solver tends to converge in fewer iterations for the larger embedding sizes, which explains the small speed-up.
To put these results into perspective, we also evaluate the test set of 121 snapshots in the (optimized) FE solver with the same time step (Δt =1e-3), which takes about 8050 seconds and leads to a mean relative L_2 error of 4.2%. Our EDNN solver is thus competitive

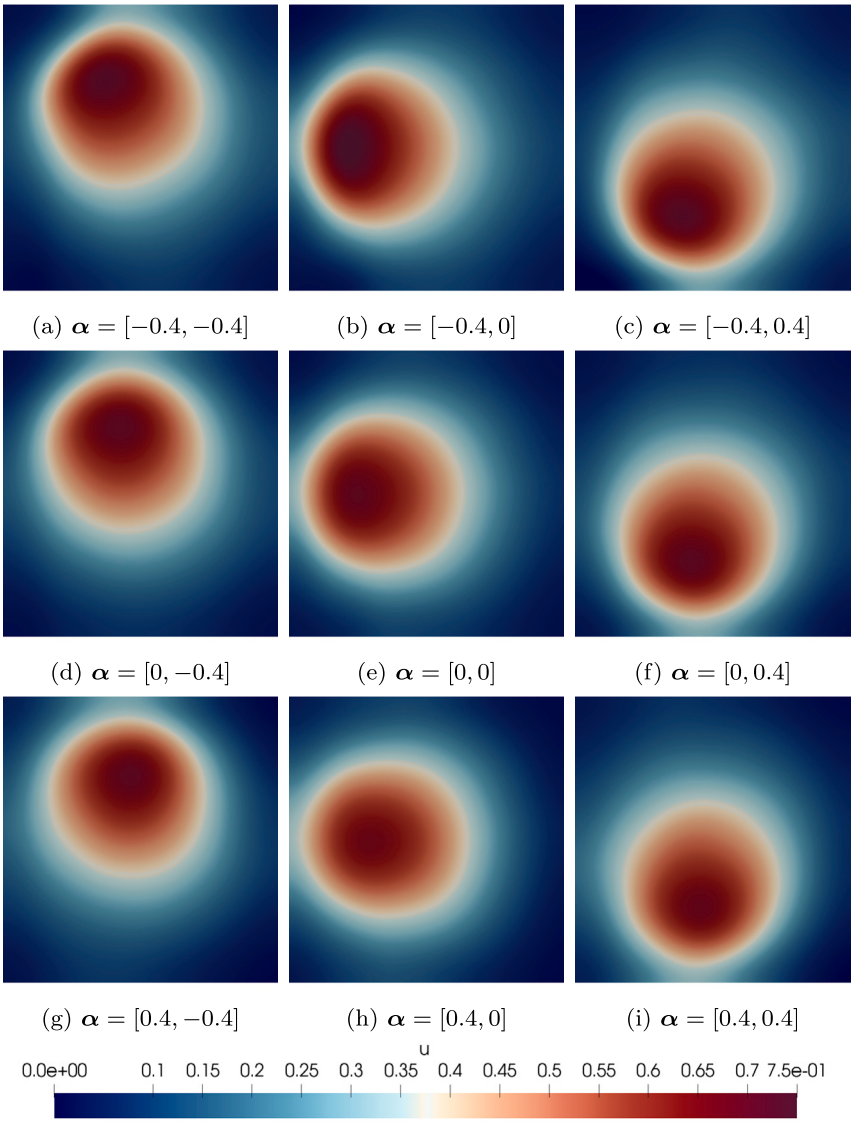
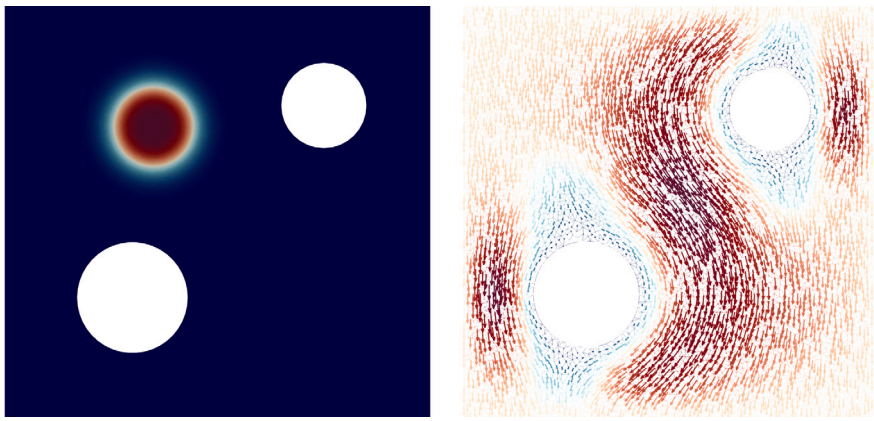
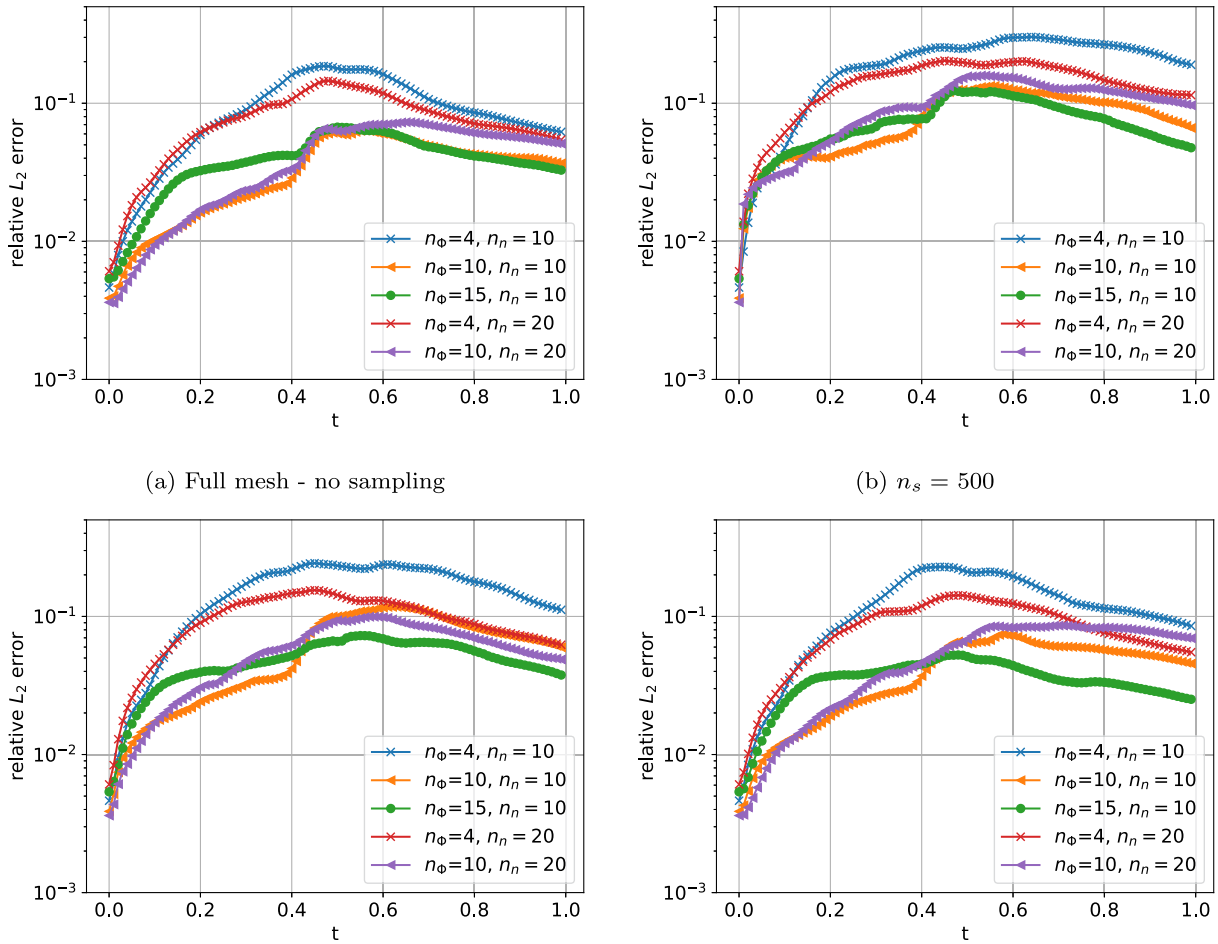


Fig. 10. Solution snapshots of the nonlinear advection diffusion problem at the final simulation time t = 0.1 for different parameter values.



(a) Initial condition for $\boldsymbol{\alpha}$=[0.0,0.0]. (b) Velocity field $\mathbf{w}(x)$.

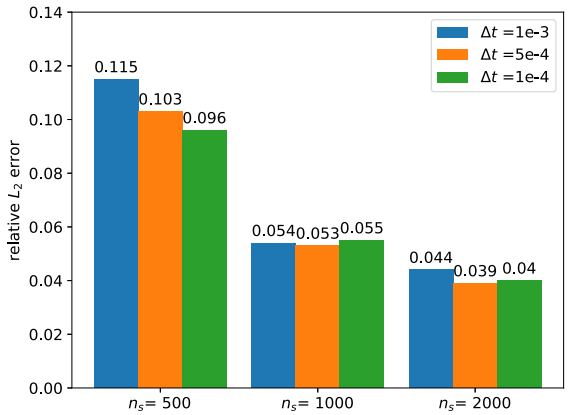
Fig. 11. Setting for advection-diffusion problem on a domain with holes.



(a) Full mesh - no sampling (b) $n_s = 500$

(c) $n_s = 1000$ (d) $n_s = 2000$

Fig. 12. Error plots for the advection-diffusion problem on the domain with holes. We compare the evaluation on the full-mesh, versus subsampling n_s points in every integration step. Each error plot shows varying architectures with n_Φ harmonic features and n_n number of hidden neurons per layer.

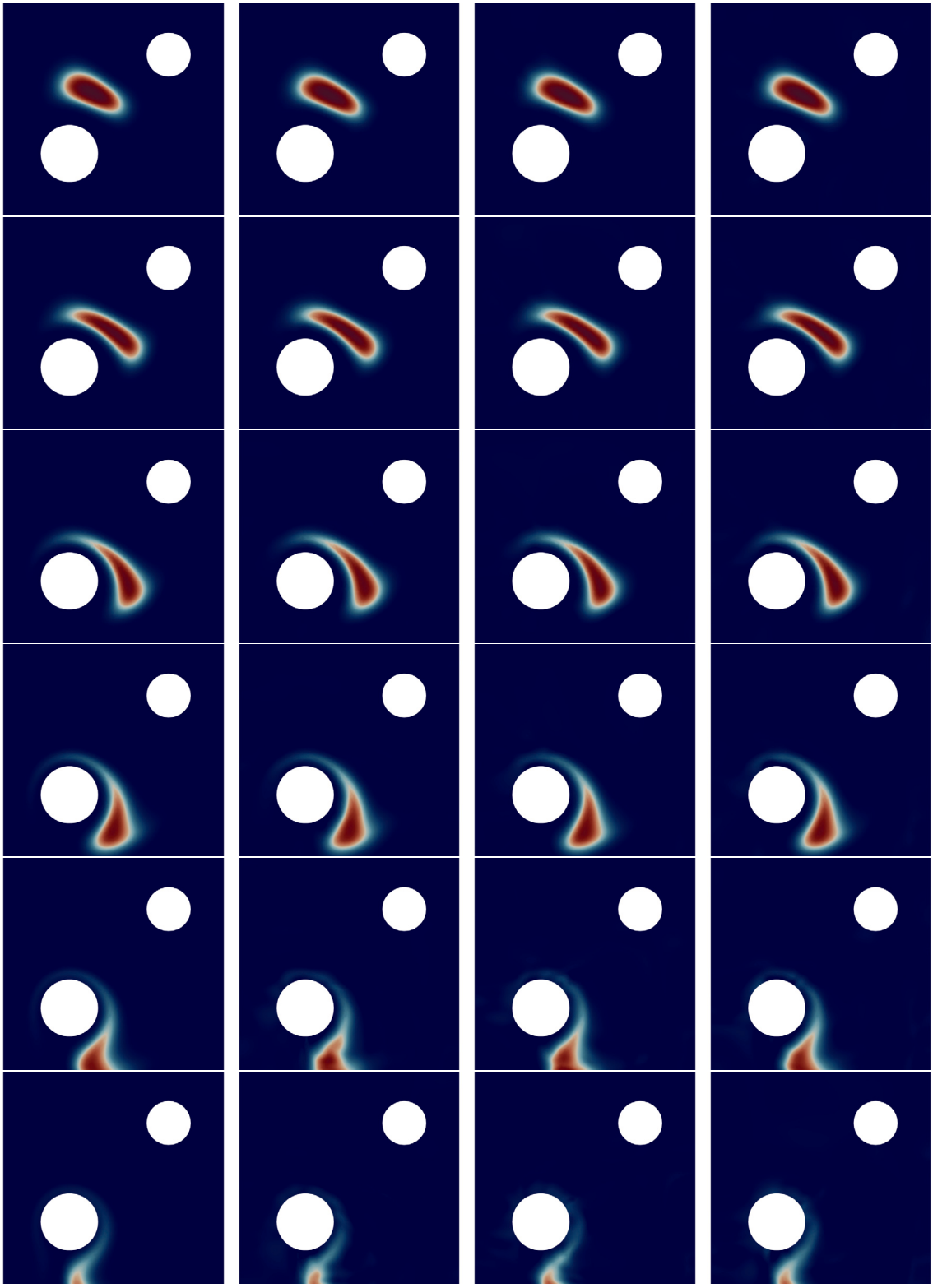


(a) Maximum relative L_2 error over time interval.

	Δt= 1e-3	Δt= 5e-4	Δt= 1e-4
n_s= 500	274	550	2750
n_s= 1000	445	880	4364
n_s= 2000	836	1636	8304

(b) Computation time in seconds.

Fig. 13. Advection-diffusion problem on the domain with holes: Effect of active sampling and time step size for the NN with embedding size $n_\Phi = 15$ and $n_n = 10$ neurons per layer.



(a) FE reference (b) $n_\Phi = 10$, full mesh (c) $n_\Phi = 15$, full mesh (d) $n_\Phi = 15$, n_s=1000

Fig. 14. Solution at t= [0.1, 0.2, 0.3, 0.4, 0.5, 0.6] (from top to bottom) for the reference solution and two different architectures, as well as the subsampled case, for the advection-diffusion problem on a domain with holes. The neural networks have four layers with 10 hidden units.

in terms of both computational time and error and also builds a surrogate model with extremely fast evaluation at previously unseen parameter values compared to a classic FE approach. We therefore see EDNNs as a promising surrogate model for parametrized, time dependent problems which are transport dominated.

6. Conclusion

We have successfully demonstrated the effectiveness of evolutional deep neural networks (EDNNs) for solving parametric time-dependent PDEs on domains with geometric structure. Our work proposes significant steps towards employing EDNNs to solve PDEs in real-world scenarios.

By introducing positional embeddings based on eigenfunctions of the Laplace-Beltrami operator, we achieve intrinsic encoding of geometric properties and automatic enforcement of Dirichlet, Neumann and periodic boundary conditions, resulting in a simplified and better-conditioned learning problem for both static and dynamic PDEs. We have shown that even a small number of feature functions and neural networks of moderate size can effectively handle transport-dominated problems.

Utilizing Krylov solvers instead of direct assembly has allowed us to scale the EDNN approach to larger neural networks, while our active sampling scheme has further improved computational efficiency. Moreover, the use of linearly implicit Rosenbrock methods allows us to effectively handle stiff PDE problems. We have further highlighted that training can be completely eliminated from the time-stepping scheme, albeit at the cost of larger approximation errors.

For single-query scenarios, traditional numerical methods will likely continue to outperform EDNNs, but in the many-query case, our approach demonstrates competitiveness as the computation of the solution across the parameter domain can happen in one single time integration. Additionally, the solution is available as a functional model with respect to the problem parameters, so that sampling or computing derivatives, for e.g. sensitivity studies, becomes inexpensive once time integration has been performed.

To fully understand the theoretical properties of EDNNs, additional investigations are necessary. Particularly, we need a further understanding of error control and how the approximation capabilities of neural networks affect their tangent spaces along the solution trajectory. Preconditioning, although unexplored in this study, holds potential as a beneficial technique for the Krylov solver. While our experiments focus on moderate-sized NNs, in future work the same methodology could be applied to more complex neural network structures. Furthermore, the idea of harmonic features can potentially be integrated into other neural network structures that currently rely on Fourier embeddings, e.g. the Fourier Neural Operator. In conclusion, our work contributes towards utilizing EDNNs for real-world PDE applications and we believe that some of the ideas introduced in this work can be applied to improve training in other neural-network based methods in PDE surrogate modeling.

CRediT authorship contribution statement

Mariella Kast: Conceptualization, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Jan S. Hesthaven:** Conceptualization, Funding acquisition, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix A. Initial condition for the Korteweg-de-Vries equation

The exact solution for the two soliton collision problem can be described by the following construction:

$$u(x,t) = 2\frac{\partial^2 \log(f(x,t))}{\partial x^2}, \tag{A.1}$$

where

$$\begin{aligned} f &= 1 + e^{\eta_1} + e^{\eta_2} + Ae^{\eta_1+\eta_2}, \\ \eta_i &= k_i x - k_i^3 t + \eta_i^{(0)}, \\ A &= \left(\frac{k_1 - k_2}{k_1 + k_2}\right)^2, \end{aligned} \tag{A.2}$$

with parameters $k_1 = 1$, $k_2 = \sqrt{5}$, $\eta_1^{(0)} = 0$, $\eta_2^{(0)} = 10.73$. The initial condition is given by evaluating the exact solution at $t = 0$: $u_0 = u(x,0)$. We point out that this solution is defined on the infinite domain, while the computational domain used here is the interval (-20,20). More details can be found in [36].

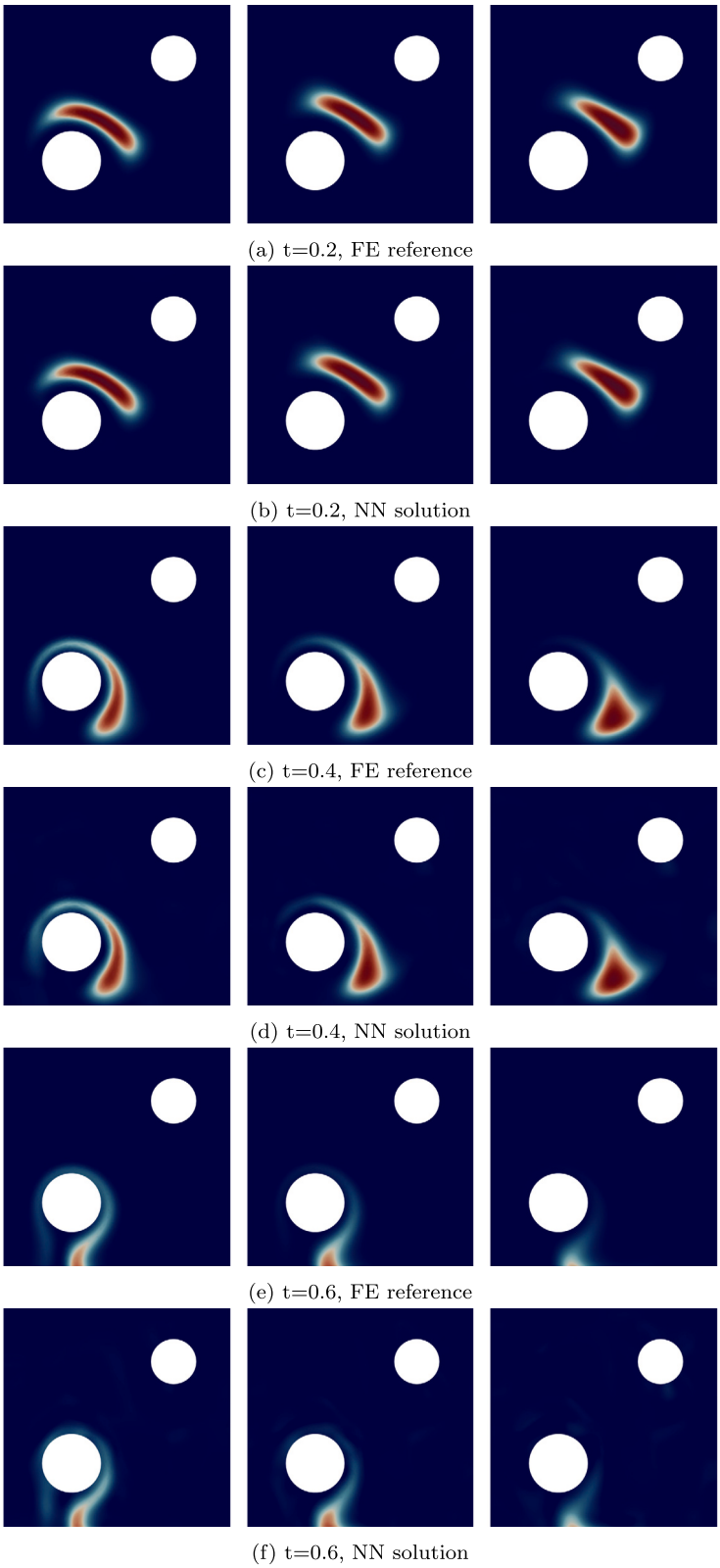
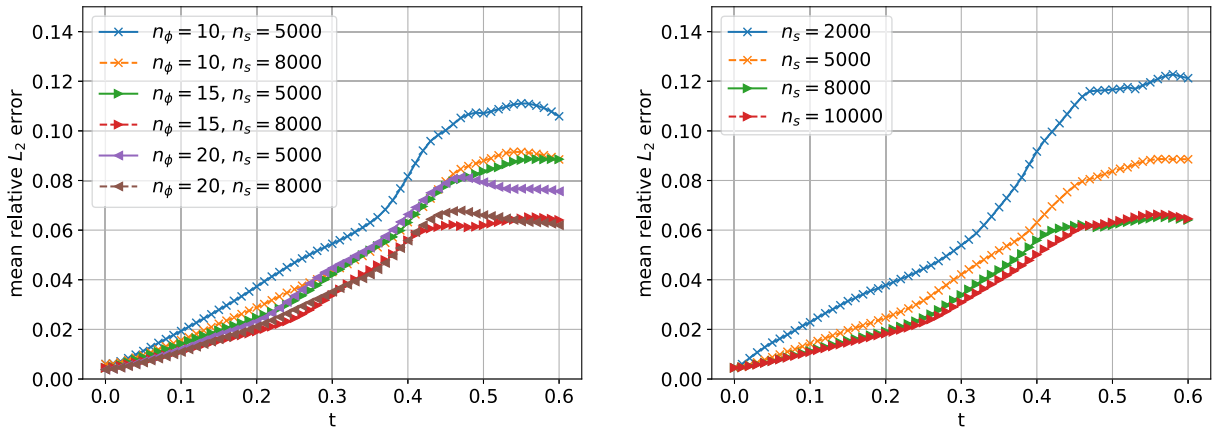


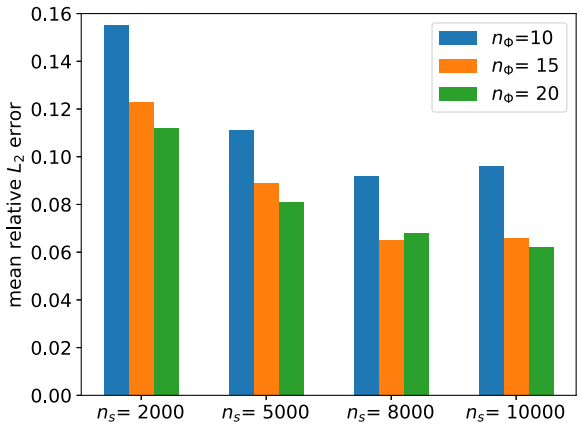
Fig. 15. Solution of the advection diffusion problem on a domain with holes for three different parameter values (left to right) $\boldsymbol{\alpha_1} = [-0.4, -0.4]$, $\boldsymbol{\alpha_2} = [0, 0.4]$ and $\boldsymbol{\alpha_3} = [0.4, 0.4]$ of the initial condition. The neural network uses $n_\Phi = 20$ input features and has four layers with 20 hidden units. 8000 points are sampled for the update equation in each step.



(a) Evolution of error for different embedding and sampling size.

(b) Effect of sampling size, $n_\Phi = 15$.

Fig. 16. Error plots for the advection-diffusion problem on the domain with holes and the parametrized initial condition.



(a) Maximum mean relative L_2 error over time interval.

	n_Φ=10	n_Φ= 15	n_Φ= 20
n_s= 2000	2509	2506	2818
n_s= 5000	5759	6148	5931
n_s= 8000	9884	9208	8552
n_s= 10000	12047	11602	11022

(b) Computation time in seconds.

Fig. 17. Parametrized advection-diffusion problem on the domain with holes: Effect of active sampling and embedding size for time step size Δt = 1e-3.

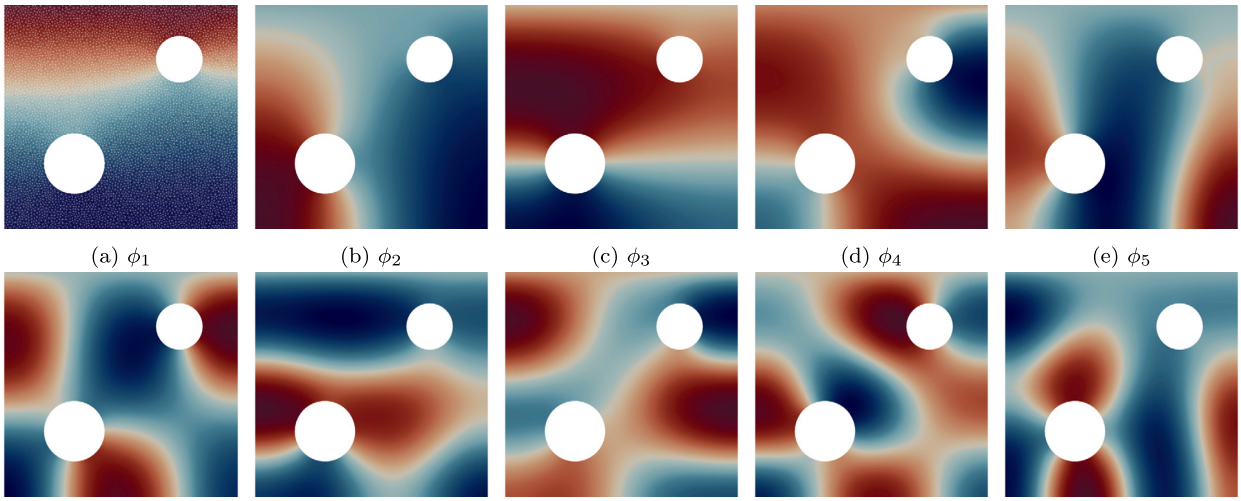
Appendix B. Velocity field computation for the domain with holes

To obtain a physically valid velocity field, we solve the steady states Navier-Stokes equation:

$$\begin{aligned} &\frac{-1}{\mathrm{Re}}\nabla^2 \boldsymbol{w} + \nabla q + \boldsymbol{w}\cdot\nabla\boldsymbol{w} = 0, \\ &\nabla\cdot\boldsymbol{w} = 0, \\ &\boldsymbol{w} = \mathbf{g_1} \quad x \in \partial\Omega_{\mathrm{top}}, \\ &\boldsymbol{w} = \mathbf{g_2} \quad x \in \partial\Omega\backslash\partial\Omega_{\mathrm{top}} \end{aligned} \tag{B.1}$$

By setting the Reynolds number $\mathrm{Re} = 100$, the solution will be in the laminar flow regime. The flow is driven by an inflow Dirichlet boundary condition $\mathbf{g_1} = (0, -1)$ on the top. A no-slip boundary condition $\mathbf{g_2} = (0, 0)$ is imposed on all remaining parts of the domain boundary. The pressure is set to 1 in one corner of the domain, to obtain a well-posed problem.

Equations (B.1) above are solved numerically in FEnics. The equations are discretized by a mixed element formulation, using P2-Lagrange elements for the velocity field and P1-Lagrange elements for the pressure. The nonlinear system is then solved with a Newton solver.



(a) ϕ_1 (b) ϕ_2 (c) ϕ_3 (d) ϕ_4 (e) ϕ_5

(f) ϕ_6 (g) ϕ_7 (h) ϕ_8 (i) ϕ_9 (j) ϕ_{10}

Fig. C.18. The first ten eigenfunctions that are used for the positional embedding on the domain with a hole. We plot the FE mesh on the first mode to visualize the resolution.

Appendix C. Eigenfunctions for the domain with holes

The test for unique identification (Algorithm 1) for the positional embedding registered collisions for embedding sizes smaller than 6. Starting from $n_\Phi = 6$, each point could be uniquely identified. This appears consistent with the eigenfunctions in Fig. C.18, which show very little variation close to the top boundary for the first 5 modes. For $n_\Phi = 4$, we detect 4 collisions on vertices close to the top boundary, which offers a potential explanation why the EDNN with only 4 embedding functions struggles to approximate the problem, even when the size of the neural network is increased.

References

[1] P. Benner, S. Gugercin, K. Willcox, A survey of projection-based model reduction methods for parametric dynamical systems, SIAM Rev. 57 (4) (2015) 483–531.
[2] J.S. Hesthaven, C. Pagliantini, G. Rozza, Reduced basis methods for time-dependent problems, Acta Numer. 31 (2022) 265–345.
[3] L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators, Nat. Mach. Intell. 3 (2) (2021) 218–229.
[4] R.T. Chen, Y. Rubanova, J. Bettencourt, D.K. Duvenaud, Neural ordinary differential equations, Adv. Neural Inf. Process. Syst. 31 (2018).
[5] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, arXiv preprint arXiv:2010.08895, 2020.
[6] J.S. Hesthaven, S. Ubbiali, Non-intrusive reduced order modeling of nonlinear problems using neural networks, J. Comput. Phys. (ISSN 0021-9991) 363 (2018) 55–78.
[7] M. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations, Commun. Numer. Methods Eng. 10 (3) (1994) 195–201.
[8] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.
[9] B. Yu, The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (1) (2018) 1–12.
[10] S. Wang, Y. Teng, P. Perdikaris, Understanding and mitigating gradient flow pathologies in physics-informed neural networks, SIAM J. Sci. Comput. 43 (5) (2021) A3055–A3081.
[11] A. Jacot, F. Gabriel, C. Hongler, Neural tangent kernel: convergence and generalization in neural networks, Adv. Neural Inf. Process. Syst. 31 (2018).
[12] S. Wang, H. Wang, P. Perdikaris, On the eigenvector bias of Fourier feature networks: from regression to solving multi-scale PDEs with physics-informed neural networks, Comput. Methods Appl. Mech. Eng. 384 (2021) 113938.
[13] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, R. Ng, Fourier features let networks learn high frequency functions in low dimensional domains, Adv. Neural Inf. Process. Syst. 33 (2020) 7537–7547.
[14] A. Solin, S. Särkkä, Hilbert space methods for reduced-rank Gaussian process regression, Stat. Comput. (ISSN 0960-3174) 30 (2) (2020) 419–446.
[15] A. Solin, M. Kok, Know your boundaries: constraining Gaussian processes by variational harmonic features, in: The 22nd International Conference on Artificial Intelligence and Statistics, in: PMLR, 2019, pp. 2193–2202.
[16] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Netw. 9 (5) (1998) 987–1000.
[17] J. Berg, K. Nyström, A unified deep artificial neural network approach to partial differential equations in complex geometries, Neurocomputing 317 (2018) 28–41.
[18] Y. Du, T.A. Zaki, Evolutional deep neural network, Phys. Rev. E 104 (4) (2021) 045303.
[19] J. Bruna, B. Peherstorfer, E. Vanden-Eijnden, Neural Galerkin schemes with active learning for high-dimensional evolution equations, J. Comput. Phys. 496 (2024) 112588.
[20] W. Anderson, M. Farazmand, Evolution of nonlinear reduced-order solutions for PDEs with conserved quantities, SIAM J. Sci. Comput. 44 (1) (2022) A176–A197.
[21] E. Hairer, G. Wanner, O. Solving, II: Stiff and Differential-Algebraic Problems, Springer Series in Computational Mathematics, Springer, Berlin [etc.], ISBN 3540537759, 1991.
[22] L.F. Shampine, M.W. Reichelt, The matlab ode suite, SIAM J. Sci. Comput. (ISSN 1064-8275) 18 (1) (1997) 1–22.

[23] A. Pinkus, N-widths in Approximation Theory, vol. 7, Springer Science & Business Media, 2012.
[24] J.S. Hesthaven, G. Rozza, B. Stamm, Certified Reduced Basis Methods for Parametrized Partial Differential Equations, vol. 590, Springer, 2016.
[25] T. Lassila, A. Manzoni, A. Quarteroni, G. Rozza, Generalized reduced basis methods and n-width estimates for the approximation of the solution manifold of parametric PDEs, in: Analysis and numerics of partial differential equations, Springer, ISBN 8847025915, 2013, pp. 307–329.
[26] C.C. Paige, M.A. Saunders, Solution of sparse indefinite systems of linear equations, SIAM J. Numer. Anal. (ISSN 0036-1429) 12 (4) (1975) 617–629.
[27] D.C.-L. Fong, M. Saunders, LSMR: an iterative algorithm for sparse least-squares problems, SIAM J. Sci. Comput. (ISSN 1064-8275) 33 (5) (2011) 2950–2971.
[28] C. Tsitouras, Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption, Comput. Math. Appl. (ISSN 0898-1221) 62 (2) (2011) 770–775.
[29] Y. Wen, E. Vanden-Eijnden, B. Peherstorfer, Coupling parameter and particle dynamics for adaptive sampling in neural Galerkin schemes, arXiv preprint arXiv:2306.15630, 2023.
[30] A. Logg, K.-A. Mardal, G. Wells, Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book, vol. 84, Springer Science & Business Media, 2012.
[31] M. Innes, Flux: elegant machine learning with Julia, J. Open Sour. Softw. (ISSN 2475-9066) 3 (25) (2018) 602.
[32] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V.B. Shah, W. Tebbutt, A differentiable programming system to bridge machine learning and scientific computing, arXiv preprint arXiv:1907.07587, 2019.
[33] L. Benet, D.P. Sanders, TaylorSeries. jl: Taylor expansions in one and several variables in Julia, J. Open Sour. Softw. (ISSN 2475-9066) 4 (36) (2019) 1043.
[34] J. Revels, M. Lubin, T. Papamarkou, Forward-mode automatic differentiation in Julia, arXiv preprint arXiv:1607.07892, 2016.
[35] C. Rackauckas, Q. Nie, Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in Julia, J. Open Res. Softw. (ISSN 2049-9647) 5 (1) (2017) 2049–9647.
[36] T.R. Taha, M.I. Ablowitz, Analytical and numerical aspects of certain nonlinear evolution equations. II. Numerical, nonlinear Schrödinger equation, J. Comput. Phys. (ISSN 0021-9991) 55 (2) (1984) 203–230.