Self-supervised neural operator for solving partial differential equations

Wen You[a,1], Shaoqian Zhou[a,1], Xuhui Meng[a,2]

[a] *Institute of Interdisciplinary Research for Mathematics and Applied Science, School of Mathematics and Statistics, Huazhong University of Science and Technology, Wuhan 430074, China*

Abstract

Neural operators (NOs) have emerged as a new paradigm for efficiently solving partial differential equations (PDEs) in various scientific and engineering disciplines. However, the training of NOs relies on large numbers of high-fidelity data generated by conventional numerical methods, which restricts the applications of NOs in complex physical systems due to computationally prohibitive costs of data generation. In this study, we propose a novel self-supervised neural operator (SNO), which is able to generate diverse highly accurate training data on the fly at low cost without using any numerical solvers. The SNO consists of three submodels: the first submodel is the physics-informed sampler (PI-sampler), which is based on the Bayesian physics-informed neural networks (B-PINNs), for efficient data generation, the second one is the function encoder (FE) which is used to learn compact representations for the inputs as well as outputs in learning operators, and the last submodel is an encoder-only Transformer for operator learning, which learns the mapping from different boundary/initial conditions, source term, and/or geometries to the solution of a specific PDE. We demonstrate the effectiveness of SNO using examples of one-dimensional steady/unsteady nonlinear reaction-diffusion equations, and a two-dimensional nonlinear PDE with different geometries. We also apply the SNO to the vortex-induced vibration of a flexible cylinder, which is widely investigated in fluid dynamics and ocean engineering. We are able to achieve relatively high accuracy in all the test cases. Furthermore, we showcase that with a light-weight finetuning

[1]The first two authors contributed equally to this work.
[2]Corresponding author: xuhui_meng@hust.edu.cn (Xuhui Meng).

of SNO (the number of trainable variables is at the order $O(100)$), we are capable of obtaining better accuracy with a few hundred finetuning steps on top of the zero-shot predictions. We envision that the present model opens a new tunnel for addressing the bottleneck for building pretrained foundation models as surrogate models for fast solving different PDEs.

Keywords: neural operator, self-supervised neural operator, physics-informed sampler, on-the-fly data generation
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

Partial differential equations (PDEs) are essential for understanding complex physical processes across diverse scientific disciplines, e.g., the transport equation in geophysics [1, 2], the Navier-Stokes equations in fluid dynamics, and the phonon Boltzmann equation for heat conduction in solid materials [3, 4]. Various numerical methods, e.g., finite difference method, finite volume method, finite element method, etc., have been developed in the past few decades to accurately solve PDEs. Although effective, conventional numerical methods are computationally infeasible in scenarios that require to solve equations repeatedly corresponding to different conditions, i.e., boundary/initial conditions, geometries, and/or source terms. For instance, in the design of airfoil, one may need to solve the Navier-Stokes equations thousands of times with different boundary conditions and/or geometries [5]. Fast numerical solvers are in desire for solving such problems.

Recently, data-driven methods based on machine learning have emerged as a new paradigm for solving PDEs [6, 7, 8, 9]. Neural operators (NOs), which are designed to learn the mappings between two function spaces via deep neural networks, are one of the most popular data-driven approaches for fast solving PDEs [10, 5, 11, 12, 13, 14, 15]. In particular, the inputs of NOs can be the representations of boundary/initial conditions, different geometries, and/or souring term, and the output is in general the solution to a specific PDE [16]. Once trained, the NOs are able to obtain the solution given a new instance of the input in one forward pass, which takes generally a fractional second and thus can be magnitudes faster than the conventional numerical methods [16, 5, 17, 18]. Among all the neural operators, deep operator networks (DeepONets) [10, 16] and Fourier neural operator (FNO)

[5, 17] are the most widely used models. Variants of DeepONets and FNO that are adapted to different applications have been proposed, e.g., POD-DeepONets [16], multi-inputs DeepONet [19], dFNO+ for problems with different input and output domains [16], and so on. Successful applications of NOs can be found in various scientific and engineering applications, e.g., shape optimization of airfoils [20], real-time prediction of pressure on different car surfaces [21], just to name a few. As aforementioned, the NOs learn the solution operator corresponding to a specific PDE [22]. Inspired by the success of large language models (LLM) in natural language processing (NLP), the community of scientific computing is also in favor to build pretrained models for solving a wide range of PDEs rather than one specific PDE [23, 22, 24, 25, 26]. Early attempts on building pretrained foundation models for solving multiple PDEs are mostly on top of neural operators, typical examples are PDEformer-2, Poseidon [25], etc.

Although remarkable progress has been made on fast solving PDEs using NOs, there are still limitations to be addressed for NOs or foundation models based on NOs. Generally, the training of these models relies on a large amount of high-fidelity data generated by conventional numerical methods, which requires to solve PDEs with diverse conditions, i.e., boundary/initial conditions, geometries, and source terms. For instance, in PDEformer-2, one of the latest foundation models for solving 2D PDEs, the datasize of the training data is about 4TB [22]. Assuming that we have 100 meshes in the additional dimension if we would like to build a foundation model for 3D PDEs, the datasize of the training data would increase to at least 4PB, which is computationally prohibited due to the expensive cost of conventional numerical methods. The expensive cost of generating high-fidelity training data becomes the bottleneck that restricts the NOs in border applications or building foundation models in scientific computing.

In this study, we propose a neural operator referred to as self-supervised neural operator (SNO) to alleviate the aforementioned issues in existing NOs. The main contributions of this work are listed below:

- We propose a novel data generation approach inspired by the Bayesian physics-informed neural networks (B-PINNs), which is capable of generating highly accurate training data for diverse differential equations on-the-fly at low cost.

- We develop a unified operator learning model based on the Transformer to learn the mapping from boundary/initial conditions, geometries, and/or source term to the solution of a specific PDE.

- We put forth a lightweight finetuning approach on top of SNO, which is embarrassingly parallel and computationally efficient, to refine the predictions of the SNO for better accuracy,

The rest of the paper is organized as follows: In Sec. 2, we introduce the problem formulation as well the proposed self-supervised neural operator, we further present a series of numerical experiments to validate the effectiveness of the SNO in Sec. 3, and then a summary of this study is present in Sec. 4.

2. Methodology

2.1. Problem formulation

Consider a PDE for the dynamics of a physical system as follows:

$$\mathcal{N}_{\boldsymbol{\beta}}[u(t,\boldsymbol{x})] = f(t,\boldsymbol{x}),\ \boldsymbol{x} \in \Omega_{\boldsymbol{x}},\ t \in \Omega_t,\ \boldsymbol{\beta} \in B, \tag{1a}$$

$$\mathcal{B}_{\beta}[u(t,\boldsymbol{x}_{bc})] = b(t,\boldsymbol{x}_{bc}),\ \boldsymbol{x}_{bc} \in \Gamma, \tag{1b}$$

$$u(t=0,\boldsymbol{x}) = u_0(t,\boldsymbol{x}), \tag{1c}$$

where $\boldsymbol{x}$ represents the $D_{\boldsymbol{x}}$-dimensional spatial coordinate, $\boldsymbol{x}_{bc}$ denotes the spatial coordinate at the boundary, t is the temporal coordinate, $\boldsymbol{\beta}$ is a $D_{\boldsymbol{\beta}}$-dimensional parameter in the equation with a probability space B, u is the solution to Eq. (1a), $\mathcal{N}$ denotes any operator, e.g., linear or nonlinear differential operator parametrized by β, f is the source term, which can be a function of time and space, $\mathcal{B}$ is the operator imposed on the boundaries, b is the boundary condition, u_0 denotes the initial condition, $\Omega_{\boldsymbol{x}}$ is a bounded domain with the boundary Γ, and Ω_t is the time domain.

Similar as the tasks in NOs, we would like to train a deep learning model, which is the self-supervised neural operator (SNO) here, to learn the mapping between the boundary/initial conditions, and/or the source term and the solution to a differential equation given data. Furthermore, we also consider the scenario where the temporal and/or spatial domains are different in different tasks.

2.2. Self-supervised neural operator

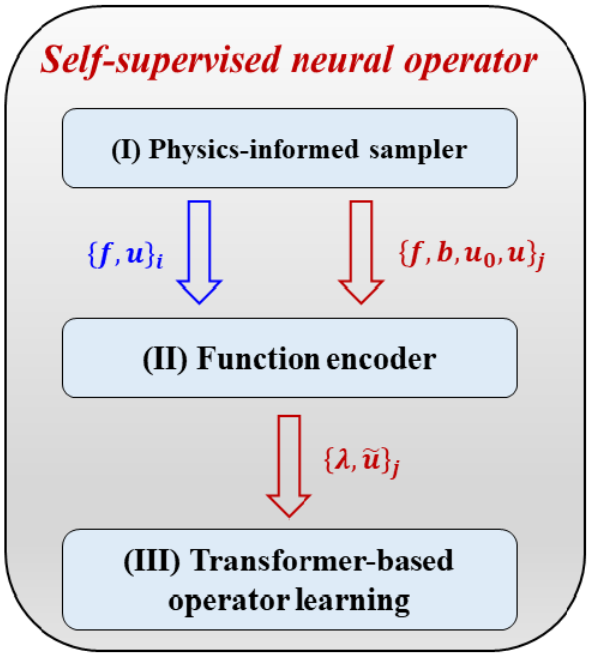

Self-supervised neural operator
(I) Physics-informed sampler
$\{f, u\}_i$
$\{f, b, u_0, u\}_j$
(II) Function encoder
$\{\lambda, \tilde{u}\}_j$
(III) Transformer-based operator learning

Figure 1: Schematic of self-supervised neural operator, which consists of three parts: (I) Physics-informed sampler (PI-sampler) for generating training data, (II) Function encoder (FE) for learning compact representations of u and f, and (III) operator learning using an encoder-only Transformer. Blue arrow and $\{f, u\}_i, i = 1, ..., B_{FE}$ (B_{FE} denotes the batch size): training data for FE generated from PI-sampler, Red arrow and $\{f, b, u_0, u\}_j, j = 1, ..., B_{OL}$ (B_{OL} denotes the batch size): training data for operator learning generated from PI-sampler, $\boldsymbol{\lambda}$ denotes the representations of $f/b/u_0$, and possibly the geometries.

As illustrated in Fig. 1, the SNO is composed of three parts: (I) Physics-informed sampler (PI-Sampler), which is to generate high-fidelity training data on-the-fly at low cost inspired by the Bayesian physics-informed neural networks (B-PINNs), (II) Function encoder, in which we apply the dimension reduction technique similar as the PCA to the solution, source term, and possibly the geometry, to enhance the computational efficiency in operator learning, and (III) Operator learning: in which we employ the Transformer to learn the mapping from the boundary/initial conditions, geometries, and source term to the solution of a specific PDE. Details on each part of SNO are present in the following subsections.

Remark that we refer to the present model as SNO since the model is trained based on the data generated by a specific part of the model itself. We do not aim to learn features by masking out parts of the unlabeled data that are widely used in the community of computer vision [27, 28].

2.2.1. On-the-fly data generation: physics-informed sampler

To train the NOs, a common way to generate training data is to assign a certain prior for the source term (i.e. f) and/or the boundary/initial condition (i.e. b/u_0), and then employ a numerical solver to solve the corresponding governing equation given samples drawn from the prior for $f/b/u_0$. Generally, the prior for $f/b/u_0$ is assumed to be a certain Gaussian process (GP). For instance, the NOs are utilized to learn the mapping from the initial condition to the solution at $t = 1$ for the Burgers equation in [16, 5], and the prior for the initial condition is a GP. As aforementioned, generating diverse training data using numerical solvers in scenarios involving complex physical processes can be computationally prohibitive. In this study, we propose the physics-informed sampler (PI-sampler) to generate highly accurate training data at low cost, in which we do not utilize any numerical approaches for data generation.

The PI-sampler is inspired by the Bayesian physics-informed neural networks (B-PINNs) [29]. We therefore will briefly review the workflow for B-PINNs prior to the introduction of PI-sampler. The schematic of B-PINNs adapted from [29] is illustrated in Fig. 2, in which we use Bayesian neural networks (BNNs) to approximate the solution to a certain PDE, and then the PDE is encoded in BNNs using the automatic differentiation (AD) similar as in PINNs. For a specific task, a prior distribution is assigned to each weight/bias ($\boldsymbol{\theta}$) in BNNs, and thus we can obtain the priors for u as well as f/b. Subsequently, we can derive the posterior distribution for $\boldsymbol{\theta}$ based on the Bayes rule given data on f (i.e., the source term) as well as b/u_0 (i.e., boundary/initial conditions) in forward problems. Finally, we are able to obtain samples from the posterior for $\boldsymbol{\theta}$ using the Hamiltonian Monte Carlo (HMC) or variational inference. Given the posterior samples of $\boldsymbol{\theta}$, the posterior samples for u and f can be computed from B-PINNs.

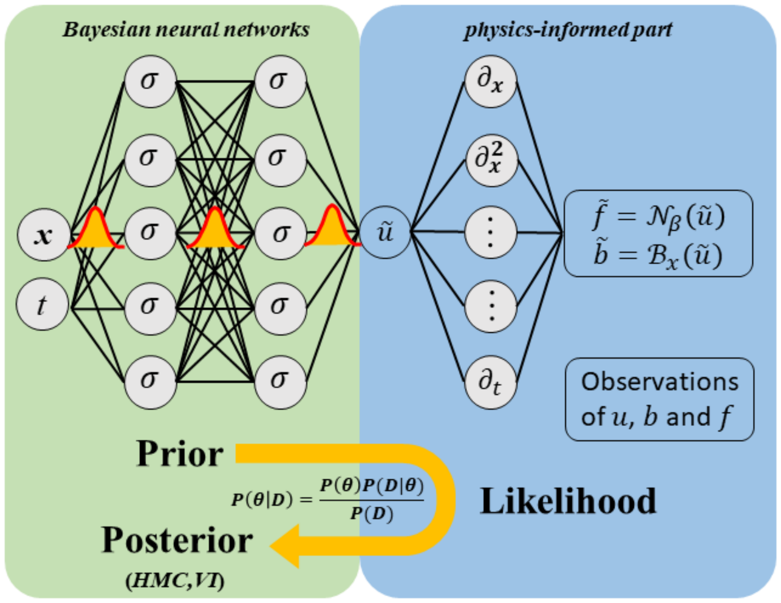

Bayesian neural networks
physics-informed part
x
t
σ
$\tilde{u}$
∂_x
∂_x^2
∂_t
$\tilde{f} = \mathcal{N}_\beta(\tilde{u})$
$\tilde{b} = \mathcal{B}_x(\tilde{u})$
Observations of u, b and f
Prior
$P(\theta|D) = \frac{P(\theta)P(D|\theta)}{P(D)}$
Likelihood
Posterior
(HMC,VI)

Figure 2: Schematic of Bayesian physics-informed neural networks (B-PINNs), in which the Bayesian neural networks (BNNs) are used to approximate the solution to a PDE, and the automatic differentiation (AD) is employed to encode the PDE in the BNNs (physics-informed part). The BNNs take as input the spatial and temporal coordinates i.e. $(\boldsymbol{x}, t)$, and outputs $\tilde{u}$, $\tilde{f}$ and b are obtained via the AD. D: observations of u, b, and f, $\boldsymbol{\theta}$: weights/biases in BNNs, $P(\boldsymbol{\theta})$: prior distribution for $\boldsymbol{\theta}$, $P(D|\boldsymbol{\theta})$: likelihood distribution for observations, $P(\boldsymbol{\theta}|D)$: posterior distribution of $\boldsymbol{\theta}$, $P(D)$: marginal likelihood, and σ: activation function in BNNs.

We proceed to introduce the PI-sampler as depicted in Fig . 3. We first construct BNNs with several single hidden layers and a specific activation function. We then impose a prior distribution for $\boldsymbol{\theta}$ i.e. the weights/biases, it is straightforward to get samples for u as we draw samples for $\boldsymbol{\theta}$. With the samples of u, we can utilize the AD to obtain samples of f based on Eq. (1a), similar as in B-PINNs. In addition, we can also obtain samples of u_0 (i.e. the initial condition) if the temporal coordinates in the inputs of BNNs are set as zeros. As for the boundary condition, if we substitute u into Eq. (1b), the boundary conditions can then be easily implemented. Note that here we do not focus on the posterior of $\boldsymbol{\theta}$ which is the quantity of interest in B-PINNs, we leverage the framework of B-PINNs to generate paired data $\{f, b, u_0, u\}$ to train the NOs. In particular, the prior of BNNs in the PI-sampler is of substantial significance to generate diverse data for u and subsequently $f/b/u_0$. As known, the BNNs with infinite width converge to a Gaussian process [30, 31, 32], and the kernel function in the corresponding GP is determined by the activation function employed as well as the prior distribution for $\boldsymbol{\theta}$ in BNNs. A detailed discussion on the relationship between BNNs and GPs is presented in Appendix A.1. More discussion on creating expressive priors of BNNs are directed to [30, 31, 32, 33].

As pointed out in Appendix A.1, we are able to generate diverse samples for u if we use different activation functions as well as prior distributions for $\boldsymbol{\theta}$ in BNNs of the PI-sampler. Further, the samples for u are agnostic to any PDEs, and they are diverse to cover the solutions to a wide range of PDEs. For instance, the same prior for u is utilized in all the six test cases described by different equations in [29]. It is also worth mentioning that the data are generated on the fly whenever we need training data, and it is of substantially low cost since we can obtain a batch of samples in one forward pass of the PI-sampler.

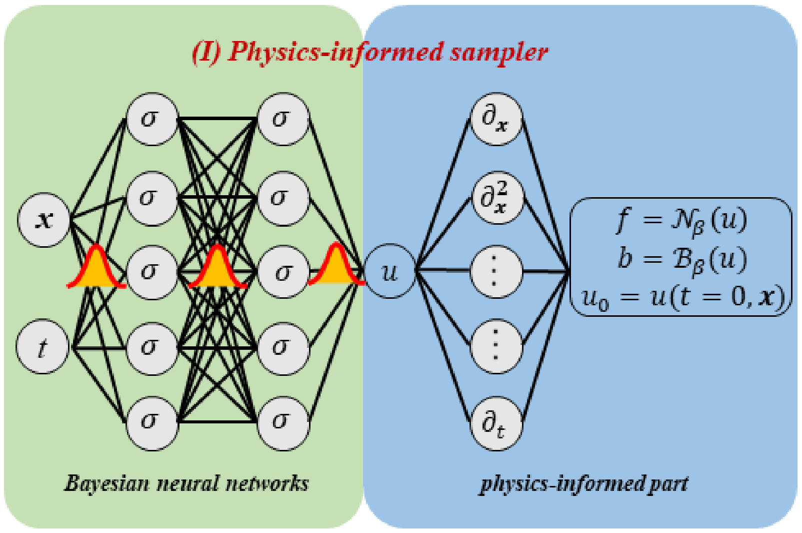

(I) Physics-informed sampler
$\boldsymbol{x}$
t
σ
u
$\partial_{\boldsymbol{x}}$
$\partial^2_{\boldsymbol{x}}$
∂_t
$f = \mathcal{N}_\beta(u)$
$b = \mathcal{B}_\beta(u)$
$u_0 = u(t=0, \boldsymbol{x})$
Bayesian neural networks
physics-informed part

Figure 3: Schematic of physics-informed sampler (PI-sampler), where the Bayesian neural networks (BNNs) are used to approximate the solution to a PDE, and the automatic differentiation (AD) is employed to encode the PDE in the BNNs (physics-informed part). The BNNs take as input the spatial and temporal coordinates i.e. $(\boldsymbol{x}, t)$, and output u, f and b are obtained via the AD. σ: activation function in BNNs.

We remark that the data generation using numerical solvers that are widely employed in the existing NOs and the PI-sampler can both be understood in the context of Bayesian framework. Without loss of generality, we assume that we need to generate paired data (f, u) for learning the mapping from f to u. In the former method, we first assign a prior for f, and the prior for u is obtained by solving the corresponding equation via conventional numerical methods. While in the latter approach, we first assign a prior for u, and then the prior for f is obtained using the AD, which is computationally much less costly.

2.2.2. Function encoder

In vanilla DeepONet and FNO, the input functions, e.g., source term, are represented by a certain number of function values on fixed points. It needs further improvements to DeepONets or FNO to handle problems with different computational spatial domains or geometries, since we cannot employ the same set of points to represent the input functions in different geometries.

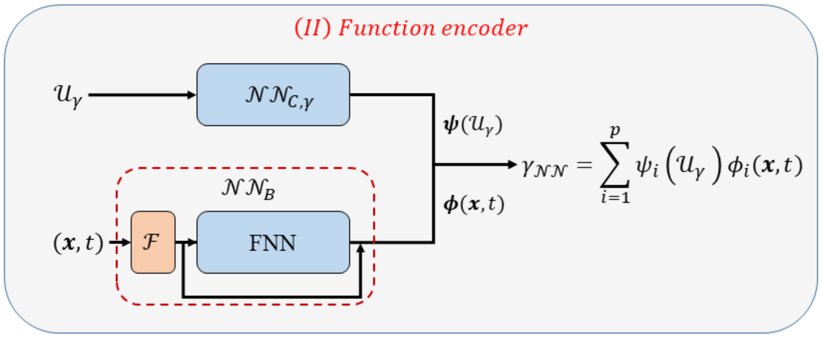

(II) Function encoder
$\mathcal{U}_\gamma$
$\mathcal{NN}_{C,\gamma}$
$\boldsymbol{\psi}(\mathcal{U}_\gamma)$
$\gamma_{\mathcal{NN}} = \sum_{i=1}^{p} \psi_i\left(\mathcal{U}_\gamma\right)\phi_i(\boldsymbol{x},t)$
$\boldsymbol{\phi}(\boldsymbol{x},t)$
$\mathcal{NN}_B$
$(\boldsymbol{x},t)$
$\mathcal{F}$
FNN

Figure 4: Schematic of function encoder (FE), in which (1) the $\mathcal{NN}_{C,\gamma}$ takes the representation of γ ($\gamma = u, f$), and outputs $\boldsymbol{\psi}(\mathcal{U}_\gamma)$, and (2) $\mathcal{NN}_B$ take as input the spatial and temporal coordinates, and outputs $\boldsymbol{\phi}(\boldsymbol{x}, t)$. The output of the FE is then expressed as $\gamma_{\mathcal{NN}} = \sum_{i=1}^{p} \psi_i(\mathcal{U}_\gamma)\phi_i(\boldsymbol{x}, t)$. σ: activation function, $\mathcal{F}$: feature expansion in $\mathcal{NN}_B$.

We propose the function encoder (FE) to address this issue in the current study. The FE has two subnetworks, i.e., $\mathcal{NN}_{C,\gamma}(\mathcal{U}_\gamma; \boldsymbol{\theta}_{C,\gamma})$ and $\mathcal{NN}_B(\boldsymbol{x}, t; \boldsymbol{\theta}_B)$, where $\gamma = u,\ f$. The first subnetwork $\mathcal{NN}_{C,\gamma}$ is parameterized by $\boldsymbol{\theta}_{C,\gamma}$. It takes as input $\mathcal{U}_\gamma$ and outputs $\boldsymbol{\psi}$ where $\boldsymbol{\psi} = (\psi_1, ..., \psi_p)^T$ with p denoting the number of dimensions for $\boldsymbol{\psi}$. In addition, the $\mathcal{NN}_B$ consists of a fully-connected neural networks (FNN) parameterized by $\boldsymbol{\theta}_B$ and a residual connection, which takes as input the spatial and/or temporal coordinate, and the output of $\mathcal{NN}_B$ is $\boldsymbol{\phi}$ where $\boldsymbol{\phi} = (\phi_1, ..., \phi_p)^T$. Also, $\mathcal{F}$ in $\mathcal{NN}_B$ denotes the feature expansions imposed on $(\boldsymbol{x}, t)$. The output of FE is obtained using the inner product of $\boldsymbol{\psi}(\mathcal{U}_\gamma)$ and $\boldsymbol{\phi}(\boldsymbol{x}, t)$ as follows:

$$\gamma_{\mathcal{N}\mathcal{N}} = \sum_{i=1}^{p} \psi_i(\mathcal{U}_\gamma; \boldsymbol{\theta}_{C,\gamma})\phi_i(\boldsymbol{x}, t; \boldsymbol{\theta}_B), \gamma = u, f. \tag{2}$$

where $\boldsymbol{\theta}_{C,u}$ and $\boldsymbol{\theta}_{C,f}$ are the parameters in $\mathcal{N}\mathcal{N}_C$ that is used to provide the coefficients for $u_{\mathcal{N}\mathcal{N}}$ and $f_{\mathcal{N}\mathcal{N}}$, respectively. In addition, $u_{\mathcal{N}\mathcal{N}}$ and $f_{\mathcal{N}\mathcal{N}}$ share the same basis function $\boldsymbol{\phi}$ here. All the parameters in $\mathcal{N}\mathcal{N}_{C,\gamma}$ and $\mathcal{N}\mathcal{N}_B$ are optimized based on the mean squared errors between the $\gamma_{\mathcal{N}\mathcal{N}}$ and the corresponding reference generated from the PI-sampler.

Finally, we discuss the treatment of the input for $\mathcal{N}\mathcal{N}_{C,\gamma}$ in the FE, especially for problems with different geometries. Without loss of generality, we assume that $\mathcal{U}$ is the representation of f, i.e., $\mathcal{U}_f$. At the training stage, we can evaluate the function values of f at any location based on the PI-sampler. It is then straightforward to use the function values at a fixed number of discrete points to represent f, similar as the input of Branch net in DeepONets. Also, it is computationally efficient if f is discretized on structured meshes since we can use convolutional neural networks (CNN) for $\mathcal{N}\mathcal{N}_{C,f}$. Otherwise, the FNNs are utilized for $\mathcal{N}\mathcal{N}_{C,f}$. At the testing stage, we can directly feed the representation of f to $\mathcal{N}\mathcal{N}_{C,f}$ to obtain the corresponding coefficients based on the pretrained FE, if f is discretized in the same way as in the training stage. While for cases in which we are not able to discretize f in the same way as in the training stage, e.g., for cases with different computational domains or geometries, we then express f_{test} as follows:

$$f_{test} = \sum_{i=1}^{p} \psi_{i,test}\phi_i(\boldsymbol{x}, t), \tag{3}$$

where ϕ_i are the pretrained basis functions in FE, and $\psi_{i,test}$ are the trainable variables, which will be obtained by minimizing the mean squared errors between f_{test} and the given data on f using Adam or any other optimizer. A more general way to deal with the input of $\mathcal{N}\mathcal{N}_{C,\gamma}$ is discussed in Appendix A.

2.2.3. Operator learning: Transformer-based operator learning

In the current study, we would like to develop a unified deep learning model that is capable of learning the mapping from the boundary/initial conditions, geometries, and the source term to the solution of a specific differential equation. Here we propose to employ the encoder-only Transformer [34] to approximate the solution operator with the training data generated from the PI-sampler. As shown in Fig. 5, the inputs for the SNO are the representations for the source term, the boundary/initial conditions, and possibly the geometries, while the output of the SNO is the representation for the solution of Eq. (1). Specifically, (1) $\boldsymbol{\psi}(\mathcal{U}_f)$ and $\boldsymbol{\psi}(\boldsymbol{\lambda})$ are the coefficients for f and u computed based on the pretrained basis functions in FE, respectively, (2) the boundary/initial conditions are represented by a certain number of discrete function values similar as the input of Branch net in DeepONets [10], and (3) the geometry can also be represented by numbers of coordinates. In addition, a linear mapping is utilized to each representation to ensure that all the inputs have the same dimensions. For cases in which the representations of a certain inputs are too long, we can employ the same approach that is widely used in Vision Transformer to reduce the number of dimensions, i.e., decompose a long sequence into smaller ones equally without overlapping [35]. More details regarding the Transformer-based operator learning, e.g. representations for$f/b/u_0$, etc., can be found in Appendix A. All the parameters in the Transformer are optimized by minimizing the following loss function:

$$\mathcal{L}_{OL} = \frac{1}{B_{OL}} \sum_{k=1}^{B_{OL}} [\boldsymbol{\psi}(\boldsymbol{\mathcal{U}_{u,k}}) - \boldsymbol{\psi}(\boldsymbol{\lambda_k})]^2, \tag{4}$$

where $\boldsymbol{\psi}(\boldsymbol{\mathcal{U}_u})$ and $\boldsymbol{\psi}(\boldsymbol{\lambda})$ are the coefficients for u corresponding to $\boldsymbol{\lambda}$, respectively. In addition, the former is obtained using the pretrained FE, and the latter is the prediction of the Transformer.

As mentioned, the output of the Transformer are the coefficients for u given a new instance of $\boldsymbol{\lambda}$ ($\boldsymbol{\lambda}$ denotes representations of $f/b/u_0$ and possibly the geometry), and the predicted u is expressed as

$$u = \sum_{i=1}^{p} \psi_i(\boldsymbol{\lambda}) \phi_i(\boldsymbol{x}, t), \tag{5}$$

where ϕ_i is the pretrained basis in the FE. Similar as in DeepONets, we can also obtain the function values of u at arbitrary locations.

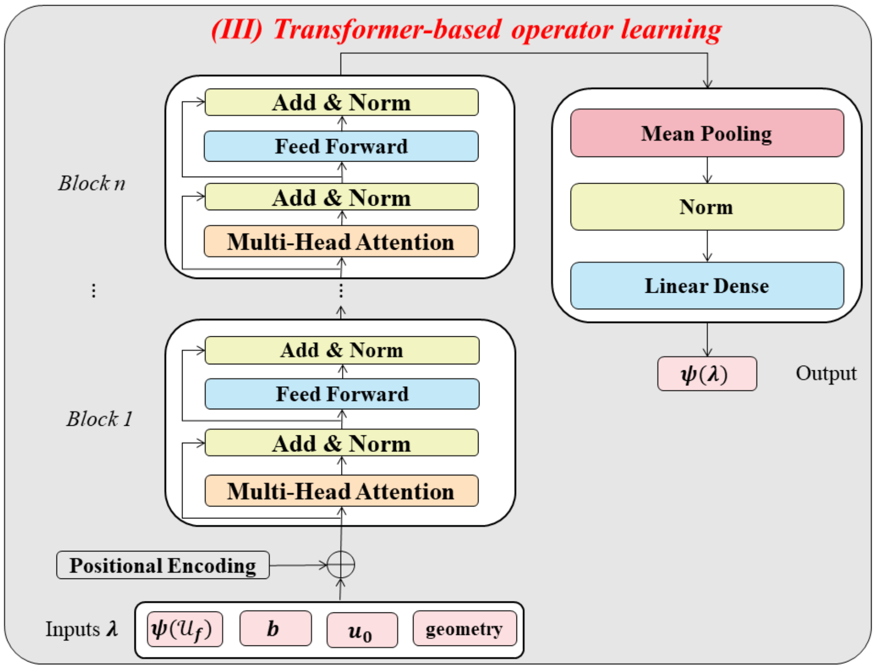


Figure 5: Schematic of the encoder-only Transformer for operator learning, in which $\boldsymbol{\lambda}$ denotes the representations of $f/b/u_0$ and possibly the geometry, and (2) $\boldsymbol{\psi}(\mathcal{U}_f)$ and $\boldsymbol{\psi}(\boldsymbol{\lambda})$ are the coefficients for f and u computed based on the pretrained basis functions in FE, respectively.

To achieve better accuracy on top of the predictions of the Transformer, we can employ the following finetuning approach, i.e., we adjust the predictions of $\boldsymbol{\psi}(\boldsymbol{\lambda})$ using the same idea of PINNs, in which the corresponding loss function for optimizing u is expressed as

$$\mathcal{L}(\boldsymbol{\psi}(\boldsymbol{\lambda})) = \mathcal{L}_{data} + \mathcal{L}_{eq}, \tag{6}$$

where $\mathcal{L}_{data}$ and $\mathcal{L}_{eq}$ denote the mean squared errors for the mismatch between predicted and given boundary/initial conditions and the residual of Eq. (1a), respectively. Generally, the number of basis functions for u and f are in the order of 100, i.e. $O(100)$. Therefore, the finetuning in the present work is lightweight and is expected to be computationally efficient.

3. Results and discussion

In this section, we employ the examples of one-dimensional steady/unsteady nonlinear reaction-diffusion equations, two-dimensional nonlinear PDE with different geometries, and a vortex-induced vibration of a flexible cylinder problem, to demonstrate the accuracy of SNO. Furthermore, we showcase the capability of SNO for extrapolation in time based on two time-dependent PDE problems. Details for our computations are present in Appendix B.

3.1. Nonlinear reaction-diffusion equation

We first consider the following one-dimensional nonlinear reaction-diffusion equation:

$$D\partial_x^2 u - u^3 = f, \ x \in [-1, 1], \tag{7}$$

where u represents the solution to the above equation, $D = 0.1$ is the diffusion coefficient, and f denotes the source term. The objective is to learn the mapping between the source term as well as the boundary conditions to the solution u via SNO.

For this specific case, the PI-sampler for data generation is set as follows: (1) the BNNs have a single hidden layer with 100 neurons each, (2) the random Fourier feature is employed as the activation function, and (3) the prior distributions for $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ are the same i.e., $\mathcal{N}(0, 5^2)$, and $\boldsymbol{W}_2 \sim \mathcal{N}(0, 1/H), H = 100$. The BNNs utilized here converge to a GP with the squared exponential kernel, and the corresponding correlation is $l = 1/5$.

Upon the training of SNO, we employ two different types of data to evaluate the accuracy of SNO, i.e., (a) we randomly draw 1000 samples of u from the BNNs used to generate the training data and refer to them as in-distribution testing data, (b) we randomly draw 1000 samples from a GP with the squared exponential kernel for f, and employ the Runge-Kutta method to solve Eq. (7) with randomly generated boundary conditions. In particular, we test three different correlation lengths for f, i.e., $l = 0.5$, 0.2, and 0.1. Representatives on the predictions from SNO in different test cases are illustrated in Fig. 6. It is observed the predictions from SNO are in good agreement with the reference solutions. Also, the relative L_2 errors for all the test cases are displayed in Table 1. As shown, (1) the mean relative L_2 error for the in-distribution testing data is less than 2%, (2) the computational errors for the cases in which the samples of f are drawn from GPs generally increase with the decreasing l, and they are smaller than 6% in all the test cases. The above results indicate the good accuracy of SNO.

We now conduct a comparison on the computational cost for different approaches for generating data for training the SNO. Assuming that we would like to learn the mapping from f to the solution u of Eq. (1a), it takes about

45.44 seconds if we first draw 1000 samples for f and then utilize the Runge-Kutta method in Matlab to solve Eq. (1) to obtain u on the Intel 13th Gen Intel(R) Core(TM) i5-13400 CPU. However, it takes about 0.07 second for the PI-sampler to generate the same number of training data using the same hardware, which is about three orders faster than the widely used data generation method in this specific case.

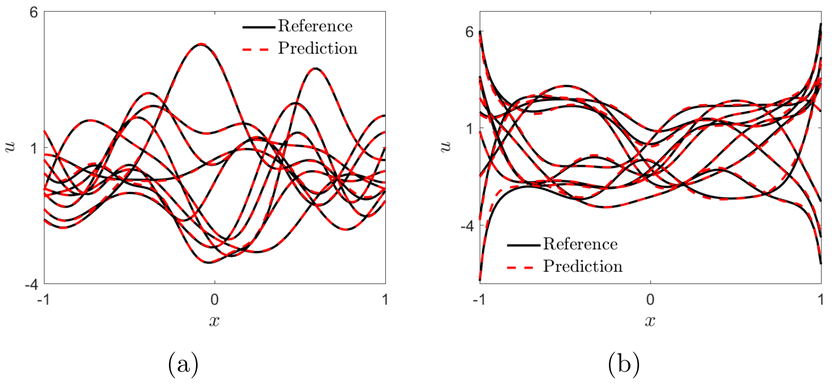


Figure 6: SNO for steady nonlinear reaction-diffusion equation: predictions of u from SNO. (a) In-distribution, (b) $f \sim \mathcal{GP}_{l=0.2}$. Black solid: Reference solution, Red dashed: Predictions from SNO.

Table 1: SNO for nonlinear reaction-diffusion equation: Relative L_2 errors for u via zero-shot predictions from SNO.

	In-distribution	$f \sim \mathcal{GP}_{l=0.5}$	$f \sim \mathcal{GP}_{l=0.2}$	$f \sim \mathcal{GP}_{l=0.1}$
E	1.28%	3.42%	3.87%	5.29%

To further improve the accuracy of SNO for the above test cases, we can then fine tune the zero-shot predictions from SNO. In particular, we perform an additional 100 steps using the LBFGS method to finetune the predicted coefficients from the Transformer using the idea of PINNs. As shown in Table 2, the mean relative L_2 errors for all the testing data in Table 1 are now smaller than 1%. Also, we note that it takes about 1 second for the finetuning in each test case if we use a batched training on one NVIDIA RTX 4090.

Table 2: SNO for nonlinear reaction-diffusion equation: Relative L_2 errors for u with 100 additional finetuning steps using LBFGS, on top of the zero-shot predictions from SNO.

	In-distribution	$f \sim \mathcal{GP}_{l=0.5}$	$f \sim \mathcal{GP}_{l=0.2}$	$f \sim \mathcal{GP}_{l=0.1}$
E	0.90%	0.33%	0.69%	0.48%

3.2. Time-dependent reaction-diffusion equation

We now test a time-dependent reaction-diffusion equation, which is expressed as:

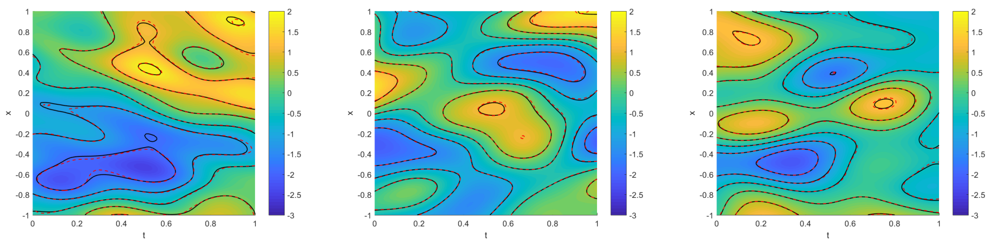
$$\partial_t u - D\partial_x^2 u + u^3 - u = f, \; x \in [-1, 1], \; t \in [0, 1], \tag{8}$$

where u denotes the solution to the equation above, $D = 0.01$ is the diffusion coefficients, and f denotes the source term. Here we learn the mapping between the source term, the boundary conditions and the initial conditions to the solution u via SNO.

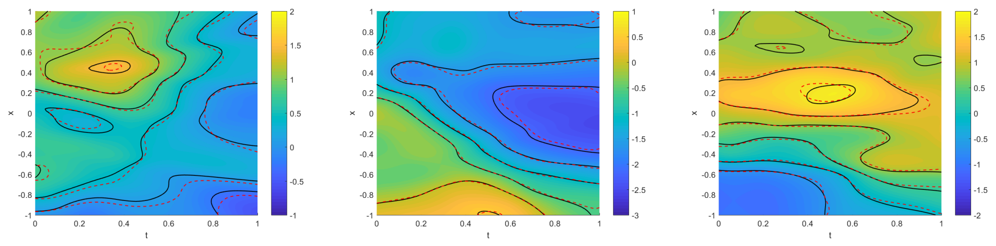
For the data generation in this specific case, the PI-sampler is set as follows: (1) the BNNs have a single hidden layer with 100 neurons each, (2) the random Fourier feature is employed as the activation function, and (3) the prior distributions for $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ are the same i.e., $\mathcal{N}(0, 5^2)$, and $\boldsymbol{W}_2 \sim \mathcal{N}(0, 1/H), H = 100$.

Similar as in the first test case, we employ two different types of data to evaluate the accuracy upon the training of SNO, i.e., (a) we randomly draw 1000 samples of u from the BNNs used to generate the training data, and (b) we randomly draw 1000 samples for u from BNNs that have the same architecture with the one used to generate the training data, but with the hyperbolic tangent function as the activation function here. In addition, the prior distributions for the weights/biases are the same as the BNNs in (a). We refer to the two types of data as in-distribution and out-of-distribution (OOD) testing data, respectively. Note that here we do not generate the second type of testing data using the same way as in Sec. 3.1, i.e., we first generate f and then use conventional numerical methods to solve Eq. (8) to obtain u, since generating data from numerical methods here is more expensive than in Sec. 3.1.

We present representatives on the predictions from SNO in different test cases in Fig. 7. As shown, the predictions from SNO for the in-distribution testing data are in good agreement with the reference solutions. However, the predictions for OOD testing data show larger computational errors compared to the in-distribution testing data. In addition, the mean relative L_2 errors for u in all the test cases are illustrated in Table 3. As shown, the errors are 2.61% and 8.74% for the in-distribution and OOD testing data, respectively, which are consistent with the results in Fig. 7. Similar as in the previous test case, the mean relative L_2 errors for u can be reduced to around 1% as we perform an additional 100 steps using the LBFGS method to finetune the predicted coefficients from the Transformer in the context of PINNs.



(a)



(b)

Figure 7: SNO for time-dependent reaction-diffusion equation: (a) In-distribution testing data, (b) OOD testing data. Background color and solid line: reference solutions, Red dashed: predictions from SNO.

Table 3: SNO for time-dependent reaction-diffusion equation: Relative L_2 errors for u between the zero-shot SNO predictions and reference solutions.

E	**In-distribution**	**OOD**
Zero-shot prediction	2.61%	8.74%
Finetuning	0.88%	1.04%

In this particular case, the time domain at the pretraining stage is $t \in [0, 1]$. However, the time domain for downstream tasks may be larger than that at the pretraining stage. To address this issue, we propose the following solution: we break down the entire time domain equally into subdomains with the length of each time domain as 1, and then use the SNO to obtain the predictions in each subdomain recurrently. Specifically, the prediction from SNO at the last time of one subdomain will serve as the initial condition for its adjacent subdomain. To test the accuracy of the proposed approach, we assume that we would like to obtain the predictions of u for $t \in [0, 3]$ in a new task. Further, the solution for u in this task is assumed to be as follows:

$$u = \exp(-0.01t)\sin(4xt), \;\; x \in [-1, 1], \;\; t \in [0, 3], \tag{9}$$

and $f/b/u_0$ can be obtained as we substitute u in Eq. (1) or (8). For this specific case, we divide the entire time domain equally into three subdomains. The computational errors between the predictions for u from SNO and the reference solution in each subdomain are illustrated in Fig. 8 and Table 4. Generally, the computational errors increase in deep learning models as we extrapolate in time. However, the SNO achieves remarkable accuracy for the time domains $t \in [1, 3]$, which are similar as in the first subdomain i.e. $t \in [0, 1]$.

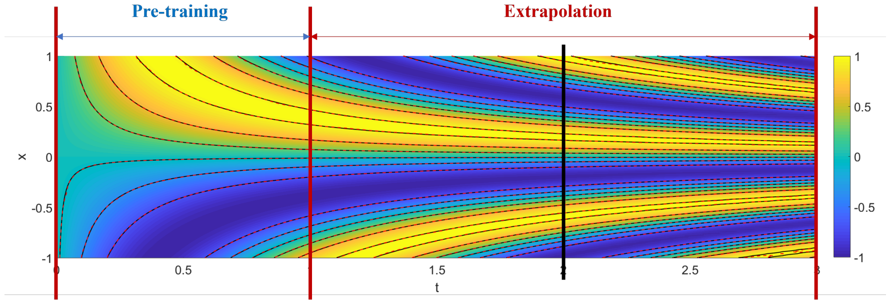


Figure 8: SNO for time-dependent reaction-diffusion equation: Predicted u in the representative case for extrapolation in time. The time domain for this specific case is $t \in [0, 3]$, which is twice larger than the time domain in the pretrained SNO. The red vertical line segments indicate the subdomains we partitioned. Background color and solid line: reference solutions, Red dashed: predictions from SNO.

To achieve better accuracy, we can also employ the finetuning technique for the extrapolation case. Specifically, we can finetune the coefficients predicted from the Transformer for u by performing 100 training steps using

LBFGS in each subdomain. In this way, we are able to improve the accuracy for the initial condition in the extrapolation. The computational errors using the finetuning are present in the third column of Table 4, which are about 1% in all subdomains. All the above results demonstrate the good accuracy of SNO for time-dependent PDE problems.

Table 4: SNO for time-dependent reaction-diffusion equation: Relative L_2 errors for u between the predictions from SNO and the reference solutions. Finetuning: 100 steps using LBFGS on top of the zero-shot predictions from SNO.

E	**zero-shot prediction**	**Finetuning**
$t \in [0, 1]$	4.77%	0.52%
$t \in [1, 2]$	6.14%	0.72%
$t \in [2, 3]$	6.76%	1.09%

3.3. Two-dimensional nonlinear problems with variable geometry

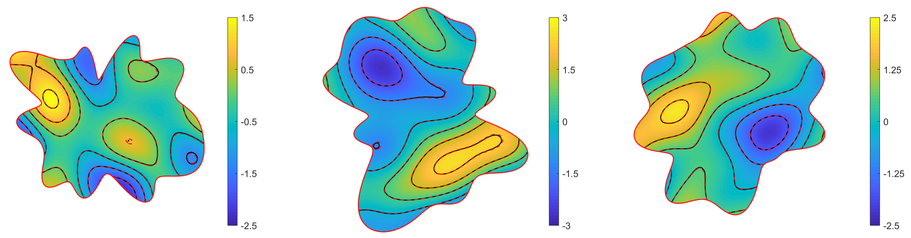
The Sine-Gordon equation is widely used in various physical fields such as nonlinear field theory and steady-state transport phenomena. We now consider the following nonlinear elliptic PDE which can be regarded as the steady Sine-Gordon equation:

$$\begin{aligned} -(\partial_x^2 u + \partial_y^2 u) + \sin(u) = f, \quad & \boldsymbol{x} \in \Omega_{\boldsymbol{x}}, \\ \mathcal{B}[u(\boldsymbol{x}_{bc})] = b, \ & \boldsymbol{x}_{bc} \in \Gamma. \end{aligned} \tag{10}$$

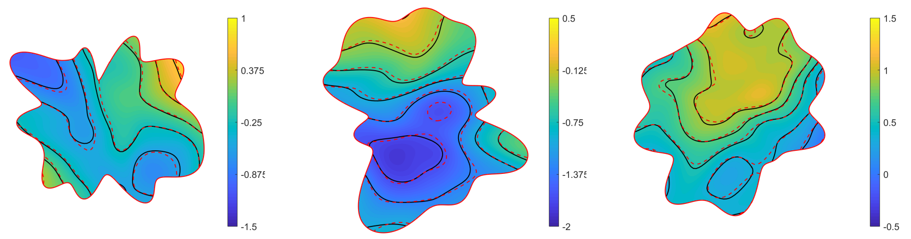
All the variables here are the same as in Eq. (1). Also, the boundary conditions imposed on all boundaries in this specific case are the Dirichlet boundary condition. Further, we also consider the computational domain or the geometry $\Omega_{\boldsymbol{x}}$, and hence the boundary Γ may vary in different cases. Here we learn the mapping between the source term, the boundary conditions, and the geometries to the solution u using SNO.

As for the data generation in the current case, the PI-sampler is set as follows: (1) the BNNs have a single hidden layer with 100 neurons each, (2) the random Fourier feature is employed as the activation function, and (3) the prior distributions for $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ are the same, i.e., $\mathcal{N}(0, 5^2)$, and $\boldsymbol{W}_2 \sim \mathcal{N}(0, 1/H), H = 100$.

To generate boundaries Γ for the computational domains in different cases, we describe Γ in the polar coordinate system and construct another BNNs to relate the angle $\alpha \in [0, 2\pi]$ and the corresponding radius r. Specifically, the input and output for the BNNs are α and r, respectively. Further, we apply the following warping to the input α, i.e., $\alpha \rightarrow [\cos(\alpha), \sin(\alpha)]$. In this way, the kernel of BNNs is periodic with a period 2π, and we can then obtain closed curves. We also utilize the following constraint to the output of BNNs to control the range of the radius, i.e., $r = \text{sigmoid}[0.6(r_{BNN}(\alpha) + 1)]$, where r_{BNN} denotes the output of the BNNs. Furthermore, the BNNs, e.g. architecture, prior distribution for $\boldsymbol{\theta}$, etc., employed here are the same as those for generating training data.



(a)



(b)

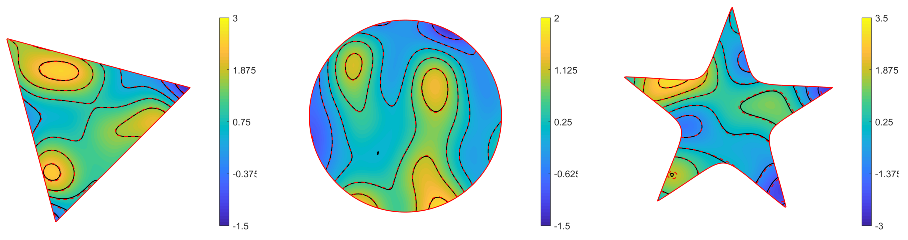
Figure 9: SNO for nonlinear elliptic partial differential equation: (a) In-distribution testing data, (b) OOD testing data. Background color and solid line: reference solutions, Red dashed: predictions from SNO.

Upon the training of the SNO, we generate the in-distribution and OOD testing data in the same way as in Sec. 8. We then randomly draw a geometry from the BNNs used to generate the boundaries in each test case. The representatives on the predictions from SNO in different test cases are illustrated in Fig. 9. As shown, the predictions from SNO for the in-distribution testing data agree quite well with the reference solutions. In addition, the predictions for OOD testing data show slightly larger computational errors compared to the in-distribution testing data. In addition, the mean relative

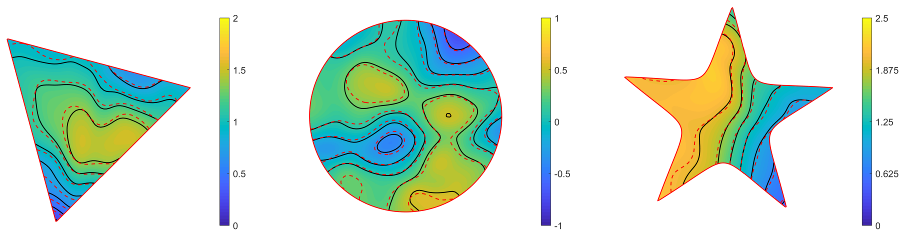
L_2 errors for u for the two types of testing data are illustrated in Table 5. As shown, the errors are 1.13% and 5.66% for the in-distribution and OOD testing data, respectively, which are consistent with the results in Fig. 9. Similar as in the previous test case, the mean relative L_2 errors for u can be reduced to about 1% as we perform an additional 100 steps using the LBFGS method to finetune the predicted coefficients from the Transformer in the context of PINNs.

Table 5: SNO for nonlinear elliptic partial differential equation: Relative L_2 errors for u between the zero-shot SNO predictions and reference solutions.

E	**In-distribution**	**OOD**
Zero-shot prediction	1.13%	5.66%
Finetuning	0.12%	1.12%



(a)



(b)

Figure 10: SNO for nonlinear elliptic partial differential equation: geometries unseen in the training data: (a) In-distribution testing data, (b) OOD testing data. Colored background and solid line: reference solutions, Red dashed: predictions from SNO.

We further evaluate the generalization of SNO on different geometries that are unseen in the training data. Specifically, we assume the geometries in downstream tasks are a triangle, a circle, and a pentagram as shown in Fig. 10. Here we employ the same testing data for u as in the previous test cases in this section. The representatives on the predictions from SNO in different test cases are illustrated in Fig. 10. As shown, the predictions from SNO for the in-distribution testing data agree quite well with the reference solutions in these unseen geometries. While the predictions for OOD testing data of u show larger computational errors similar as in previous cases. In addition, the mean relative L_2 errors for u for the two types of testing data on unseen geometries are illustrated in Table 6. As shown, the errors are less than 3% and 6% for the in-distribution and OOD testing data, respectively, which are consistent with the results in Fig. 10. Similar as in the previous test case, we can reduce the mean relative L_2 errors for u to about 1% as we perform an additional 100 steps using the LBFGS method to finetune the predicted coefficients from the Transformer in the context of PINNs.

Table 6: SNO for nonlinear elliptic partial differential equation: Relative L_2 errors for u between the zero-shot SNO predictions and reference solutions in cases where the geometries are unseen in the training data.

	E	**In-distribution**	**OOD**
Triangle	Zero-shot prediction	1.54%	5.24%
	Finetuning	0.22%	1.26%
Circle	Zero-shot prediction	1.02%	5.86%
	Finetuning	0.12%	1.14%
Pentagram	Zero-shot prediction	2.35%	5.78%
	Finetuning	0.12%	0.92%

We now conduct a comparison between the computational cost for generating data using different approaches in this specific case. Assume that the objective is to learn the mapping from f to u in Eq. (1a), (1) it takes about 5.67 seconds for the PI-sampler to generate 1000 paired data (f, u) on 1000 different geometries using the Intel 13th Gen Intel(R) Core(TM) i5-13400 CPU, and (2) the computational time is about 2425 seconds if we use the same f as well as geometries and utilize the *pdetoolbox* in Matlab to solve Eq. (10) to obtain u on the same CPU. As shown, the PI-sampler is about two orders faster than the conventional numerical methods for data generation in this particular case. Note that if we run the PI-sampler on the NVIDIA RTX 3090, it takes only 0.002 seconds which is three orders faster than on the CPU.

3.4. Application to vortex-induced vibration of flexible cylinder

Vortex-induced vibration of a flexible cylinder is widely investigated in the fluid dynamics and ocean engineering. We now consider to employ the SNO to solve the governing equation in the cross-flow (CF) direction (Fig. 11) which is expressed as:

$$\frac{\partial^2 \eta}{\partial t^2} + 2\zeta\omega_n \frac{\partial \eta}{\partial t} + \frac{\mathrm{EI}}{\mu}\frac{\partial^4 \eta}{\partial z^4} - \frac{\mathrm{T}}{\mu}\frac{\partial^2 \eta}{\partial z^2} = \frac{F_l}{\mu},\ z \in [-1, 1],\ t \in [0, 1], \tag{11}$$

where η is the displacement in the CF (i.e., y) direction, μ is the cylinder mass per unit length, ζ is the damping ratio, T is the tension, and EI is the bending stiffness. The details on all the parameters used in the computations are summarized in Table 7 following [36], where β denotes the damping

Table 7: VIV problem: Summary of the parameters used in the computations.

$U = 1$	$\rho = 1$	$d = 1$	$U_r = 12.66$	
$f_n = \frac{U}{U_r d}$	$\omega_n = 2\pi f_n$	$\zeta = 0.087$	$\mu = m^* \rho \frac{\pi d^2}{4}$	$\beta = 2\zeta\omega_n\mu$
$m^* = 4$	$L = 240$	$EI = 0.02T$	$T = (2 f_n L)^2 \left(m + \frac{\pi}{4}\right)$	

coefficients, and f_n is the natural frequency of the riser first modal vibration assuming uniform added mass coefficient $C_m = 1.0$ along the span. In addition, U_r is the corresponding reduced velocity, F_l is the y-component of the hydrodynamic force, exerted on the cylinder surface, d is the cylinder diameter, m^* is the mass ratio, i.e. ratio between structural mass and displaced fluid mass. In this study, we consider a uniform flexible cylinder with an aspect ratio $L/d = 240$, and the U is the inflow velocity. The above equation is constrained by pinned boundary condition($\eta = 0, \frac{\partial^2 \eta}{\partial z^2}$=0) at both ends, i.e., $z = -1$ and 1.

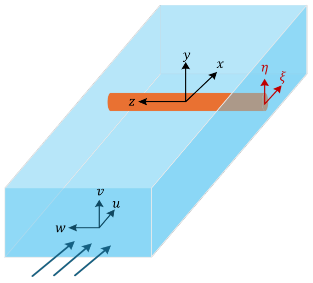


Figure 11: Schematic of the vortex-induced-vibration of a flexible cylinder. η and ξ are cross- and in-flow displacements of the cylinder, respectively. Adapted from [36].

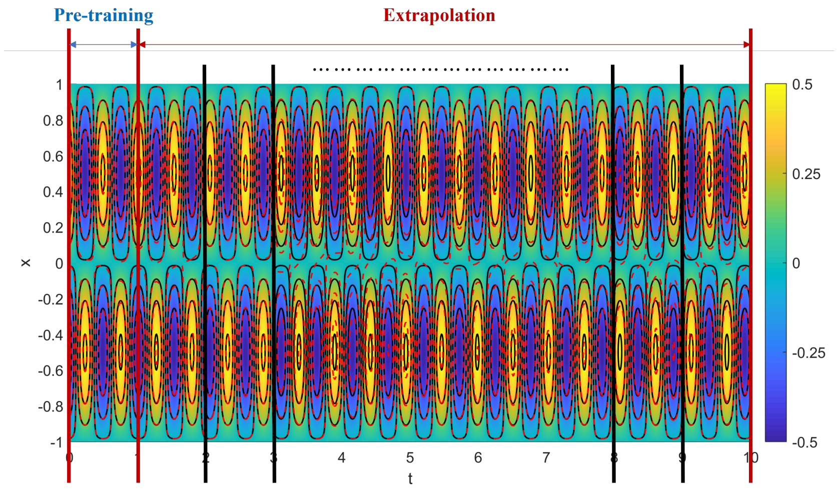


Figure 12: SNO for VIV problem: Predicted η from SNO. The time domains at the pre-training and testing stages are $t \in [0, 1]$ and $t \in [0, 10]$, respectively. Colored background and solid line: reference solutions, Red dashed: predictions from SNO.

In this specific case, the training data are generated using the same BNNs as in Sec. 3.2. After the pretraining of SNO, we assume that the solution for η in the new test case is expressed as follows:

$$\eta = 0.5 \sin(\pi z) \sin(12t + 2),\ \ t \in [0, 10]. \tag{12}$$

The source term as well as the initial condition can be obtained as we substitute η in Eq. (11). Note that the time domain in this case is 9 times larger than in the pretrained SNO. Similarly, we equally divide the entire time domain into ten subdomains, and we can obtain the predictions for each subdomain using the same way as in Sec. 3.2. The mean relative L_2 error for all the subdomains between the predictions from SNO and the reference solutions is about 16.48%. Moreover, to improve accuracy, we can sequentially finetune each the predictions in each subdomain. Similar as in Sec. 3.2, the mean relative L_2 error can be reduced to 7.48% for all subdomains as we further update the coefficients predicted from SNO using 100 additional training steps in LBFGS.

4. Summary

In this study, we introduced the self-supervised neural operator (SNO) to fast solve PDE problems by learning the solution operator given data. The attractive features of the SNO are summarized as follows:

- The physics-informed sampler (PI-sampler) is capable of generating highly accurate training data for diverse differential equations on-the-fly at low cost without using any numerical solvers.
- With the function encoder (FE) and encoder-only Transformer, the SNO is a unified framework which is capable of learning the mapping from boundary/initial conditions, geometries, and/or source term to the solution of a specific PDE.
- The lightweight finetuning approach of SNO can refine the predictions of the SNO for better accuracy with only a few hundred additional training steps, and is also embarrassingly parallel and computationally efficient.

Numerical experiments on various PDE problems, including steady and unsteady nonlinear reaction-diffusion equations, two-dimensional nonlinear PDEs with different geometries, and a vortex-induced vibrations of a flexible cylinder, confirm the good accuracy and versatility of the SNO in handling diverse boundary/initial conditions, source terms, and geometries. Further, it is observed that the SNO performed remarkably well for extrapolations in time for unsteady PDE problems, especially for cases with finetuning. Overall, the SNO provides a promising step forward in reducing the computational cost of training neural operators and could serve as a foundation for developing large-scale surrogate models for a wide range of PDE problems involving in diverse scientific and engineering disciplines.

Acknowledgements

W. Y., S. Z., and X. M. acknowledge the support of the National Natural Science Foundation of China (No. 12201229) and the Interdisciplinary Research Program of HUST (No. 2024JCYJ003). X. M. also acknowledges the support of the Xiaomi Young Talents Program.

Appendix A. Additional details on SNO

Appendix A.1. On relations between BNNs and GPs

It is well known that the BNNs with infinite width converge to a certain Gaussian process [30, 31, 32]. Consider BNNs with a single hidden layer, $u_{\mathcal{NN}}(\boldsymbol{x}) : R^{D_x} \to R$, the output for each layer can then be expressed as:

$$h(\boldsymbol{x}) = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1), \tag{A.1}$$

$$u_{\mathcal{NN}}(\boldsymbol{x}) = \boldsymbol{W}_2 h(\boldsymbol{x}) + \boldsymbol{b}_2, \tag{A.2}$$

where σ denotes the activation function, $\boldsymbol{W}_i/\boldsymbol{b}_i, ,\ i = 1, 2$ are the weights/biases in the ith layer, and $\boldsymbol{x}$ here represent both the spatial and/or temporal coordinates for simplicity. In addition, (1) we assume that the priors for all parameters have independent zero-mean Gaussian distributions in the BNNs, and (2) the prior distributions for the parameters in the same layer are independently and identically distributed. As we denote all weights/biases by $\boldsymbol{W}$, the mean of $u_{\mathcal{NN}}(\boldsymbol{x})$ and covariance or kernel function for BNNs read as:

$$\mathbb{E}_{\boldsymbol{W}}[u_{\mathcal{NN}}(\boldsymbol{x})] = 0, \tag{A.3}$$

$$\begin{aligned} K(\boldsymbol{x}, \boldsymbol{x}') &= \mathbb{E}_{\boldsymbol{W}}\left[u_{\mathcal{NN}}(\boldsymbol{x})u_{\mathcal{NN}}(\boldsymbol{x}')\right] \\ &= \mathbb{E}_{\boldsymbol{W}}\left[(\Sigma_{i=1}^{H}\omega_{2i}h_i(\boldsymbol{x})(\Sigma_{j=1}^{H}\omega_{2j}h_j(\boldsymbol{x}')\right] + \sigma_{\boldsymbol{b}_2}^2 \\ &= \mathbb{E}_{\boldsymbol{W}}\left[\omega_{2,1}h_1(\boldsymbol{x})\omega_{2,1}h_1(\boldsymbol{x}') + \omega_{2,1}h_1(\boldsymbol{x})\omega_{2,2}h_2(\boldsymbol{x}') + \cdots \right. \\ &\qquad \omega_{2,2}h_2(\boldsymbol{x})\omega_{2,1}h_1(\boldsymbol{x}') + \omega_{2,2}h_2(\boldsymbol{x})\omega_{2,1}\Phi_1(\boldsymbol{x}') + \cdots \\ &\qquad \left. \cdots + \omega_{2,H}h_H(\boldsymbol{x})\omega_{2,H}h_H(\boldsymbol{x}')\right] + \sigma_{\boldsymbol{b}_2}^2, \end{aligned} \tag{A.4}$$

where H denotes the width of the hidden layer, $\omega_{2,i}$ is the entry of matrix $\boldsymbol{W}_2$, h_i is the entry of matrix h, and $\sigma_{\boldsymbol{b}_2}$ represents the standard deviation of the prior distribution for $\boldsymbol{b}_2$, i.e. $\boldsymbol{b}_2 \sim \mathcal{N}(0, \sigma^2_{\boldsymbol{b}_2}\boldsymbol{I})$. Since the prior for each parameter is assumed to be independent, the last term in Eq. (A.4) becomes:

$$\begin{aligned} K(\boldsymbol{x}, \boldsymbol{x}') =& \mathbb{E}_{\boldsymbol{W}}\big[\sum_{k=1}^{H} \omega_{2,k} h_k(\boldsymbol{x}) \omega_{2,k} h_k(\boldsymbol{x}')\big] + \sigma^2_{\boldsymbol{b}_2} \\ &= H\mathbb{E}_{\boldsymbol{W}}\big[w_{2,k} h_k(\boldsymbol{x}) w_{2,k} h_k(\boldsymbol{x}')\big] + \sigma^2_{\boldsymbol{b}_2} \\ &= \sigma^2_{\boldsymbol{W}_2}\mathbb{E}_{\boldsymbol{W}}\big[h_k(\boldsymbol{x}) h_k(\boldsymbol{x}')\big] + \sigma^2_{\boldsymbol{b}_2} \end{aligned} \tag{A.5}$$

in which the variance of $\boldsymbol{W}_2$ is scaled by the width $1/H$, i.e., $\boldsymbol{W}_2 \sim \mathcal{N}(\boldsymbol{0}, \sigma^2_{\boldsymbol{W}_2}\boldsymbol{I}/H)$, and $\boldsymbol{W}_1 \sim \mathcal{N}(\boldsymbol{0}, \sigma^2_{\boldsymbol{W}_1}\boldsymbol{I})$, $\boldsymbol{b}_1 \sim \mathcal{N}(\boldsymbol{0}, \sigma^2_{\boldsymbol{b}_1}\boldsymbol{I})$, $\sigma_{\boldsymbol{W_1}} = \sigma_{\boldsymbol{b_1}}$, with $\boldsymbol{I}$ denoting the identity matrix. The output of BNNs in Eq. (A.1) is a summation of identically and independently distributed random variables, which is normally distributed as H goes to infinity as the central limit theorem is applied assuming that σ is bounded. Further, we can see that the kernel function for BNNs is determined by the activation function as well as the prior distributions for the parameters in BNNs.

In theory, we need the BNNs with infinite width to converge to a Gaussian process. However, it is empirically observed that BNNs with more than 50 nodes in the hidden layer are wide enough to converge to GPs [32, 29]. In addition, the kernel function can be obtained analytically for BNNs with several specific activation functions, e.g. ReLU, radial basis function (RBF), random Fourier feature activation function, and so on. For BNNs with activation functions, e.g. *tanh*, that are challenging to derive the kernel function analytically, we can visualize it numerically.

We now take the BNNs with random Fourier feature activation function, i.e. $\sigma = [\cos, \sin]^T$, as an example to derive the corresponding kernel function. We can rewrite h in Eq. (A.1) as $h(\boldsymbol{x}) = \sqrt{2}\sin(\boldsymbol{W_1 x} + \boldsymbol{b_1} + \frac{\pi}{4}\boldsymbol{I})$, With the aid of Eq. (A.5), we can obtain that:

$$K(\boldsymbol{x}, \boldsymbol{x}') \tag{A.6}$$

$$= \sigma^2_{\boldsymbol{W}_2}\mathbb{E}_{\boldsymbol{W}}\left[\sqrt{2}\sin\left(\boldsymbol{W_1 x} + \boldsymbol{b_1} + \frac{\pi}{4}\boldsymbol{I}\right)\sqrt{2}\sin\left(\boldsymbol{W_1 x}' + \boldsymbol{b_1} + \frac{\pi}{4}\boldsymbol{I}\right)\right] + \sigma_{\boldsymbol{b}_2} \tag{A.7}$$

$$= \sigma^2_{\boldsymbol{W}_2}\mathbb{E}_{\boldsymbol{W}}\left[\cos\left(\boldsymbol{W_1}(\boldsymbol{x} - \boldsymbol{x}')\right) - \sin\left(\boldsymbol{W_1}(\boldsymbol{x} + \boldsymbol{x}') + 2\boldsymbol{b_1}\right)\right] + \sigma_{\boldsymbol{b}_2} \tag{A.8}$$

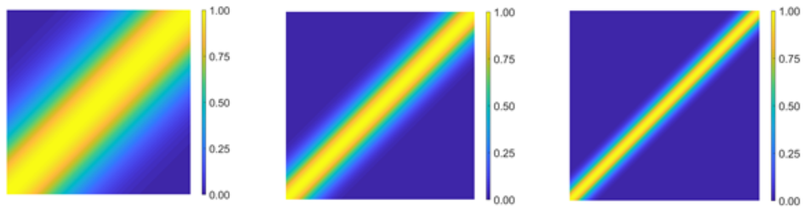
$$\begin{aligned} &= \sigma^2_{\boldsymbol{W}_2}\mathbb{E}_{\boldsymbol{W}}\Big[\cos\left(\boldsymbol{W_1}(\boldsymbol{x} - \boldsymbol{x}')\right) - \sin\left(\boldsymbol{W_1}(\boldsymbol{x} + \boldsymbol{x}')\right)\cos\left(2\boldsymbol{b_1}\right) \\ &\qquad - \cos\left(\boldsymbol{W_1}(\boldsymbol{x} + \boldsymbol{x}')\right)\sin\left(2\boldsymbol{b_1}\right)\Big] + \sigma_{\boldsymbol{b}_2} \end{aligned} \tag{A.9}$$

Since $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ are independently and identically distributed zero-mean Gaussian distributions, we can then obtain the following:

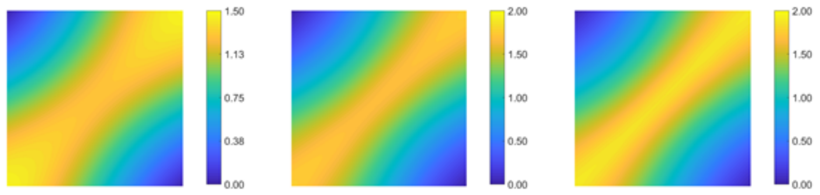
$$K(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2_{\boldsymbol{W}_2} \mathbb{E}_{\boldsymbol{W}} \left[\cos\left(\boldsymbol{W_1}(\boldsymbol{x} - \boldsymbol{x}')\right)\right] + \sigma_{\boldsymbol{b_2}} \tag{A.10}$$

$$= \sigma^2_{\boldsymbol{W}_2} \exp[-\frac{||\boldsymbol{x} - \boldsymbol{x}||^2}{2l^2}] + \sigma_{\boldsymbol{b_2}} \tag{A.11}$$

where $l = 1/\sigma_{\boldsymbol{W}_1}$. Illustration examples for BNNs with different priors are displayed in Fig. A.13, in which (1) all BNNs have one single hidden layer with the width 100, (2) the priors for all parameters are assumed to be independent zero-mean Gaussian distribution, (3) we drop the bias in the final linear layer of BNNs since it only changes the range of the the kernel function for simplicity, and (4) each kernel function is computed using Monte Carlo method with 10,000 samples.



(a)



(b)

Figure A.13: Kernel functions for BNNs with different prior distributions. (a) BNNs with random Fourier features: from left to right $\sigma_{\boldsymbol{W}_1} = 2,\ 5,\ 10$, (b) Ns with hyperbolic activation function: from left to right $\sigma_{\boldsymbol{W}_1} = 2,\ 5,\ 10$. All BNNs have one single hidden layer with the width 100. The kernel functions are computed using the Monte Carlo method with 10,000 samples.

It is worth noting that we can obtain diverse kernel functions to enrich the training data in SNO, e.g. periodic kernels, etc., if we apply more advanced techniques, e.g., feature expansion or wrapping, addition, multiplication, and so on, to BNNs. Interested readers are directed to [32, 33].

Appendix A.2. Additional details for FE

As mentioned, we have to compute the coefficients for a new f in cases with variable geometries since we may not be able to obtain the measurements on the same points for each f, which is not quite straightforward for practical use. Here we discuss a more general way to deal with the input for $\mathcal{NN}_{C,f}$ for cases with variable geometries.

Without loss of generosity, let us consider a two-dimensional case as an example. In Sec. 3.2, we assume that we have measurements on $N_{f,x}$ and $N_{f,y}$ discrete points in $x-$ and $y-$direction for each f. Subsequently, $\mathcal{U}_f$ has a dimension $B_{FE} \times N_{f,x} \times N_{f,y}$, where B_{FE} is the batch size for training the FE. Alternatively, we express $\mathcal{U}_f$ using the coordinates as well as the corresponding function values, i.e., $\{\boldsymbol{x}, f\}_i, i = 1, ..., N_{f,x} \times N_{f,y}$, and each $\mathcal{U}_f$ now has a dimension of $B_{FE} \times (N_{f,x} \times N_{f,y}) \times (D_{\boldsymbol{x}} + 1)$. We then feed $\mathcal{U}_f$ to $\mathcal{NN}_{C,f}$, which is in general a feed-forward neural network, to obtain an output that has a dimension $B_{FE} \times (N_{f,x} \times N_{f,y}) \times p$. The definition of p is the same as in Sec. 2. Finally, a dimension reduction technique, i.e., max or mean pooling, is imposed on the second dimension of the output above, to obtain a final output which has the dimension $B_{FE} \times p$. Note that in the approach discussed here: (1) we use $\boldsymbol{x}$ to denote both the spatial and/or temporal coordinates for simplicity, and (2) the number as well as the locations of the measurements for different f can be different since the input dimension is independent on the number of measurements for f. With the pretrained FE, we can directly obtain the coefficients for a new f as we feed the representation $\mathcal{U}_f$ to $\mathcal{NN}_{C,f}$ even we cannot use the same way to discrete f in cases with variable geometries.

Appendix A.3. Details on Transformer-based operator learning

Consider the objective is to learn the solution operator using the Transformer given the source term and the boundary condition. The source term f is represented by the coefficients $\Psi_{\mathcal{U}_f} \in R^{B_{OL} \times p}$ based on the pretrained basis functions Φ, where B_{OL} and p denote the batch size and the number of basis functions in the FE. The boundary conditions are represented by numbers of discrete points on the boundaries, which are denoted as $\{\boldsymbol{x}_{bc,i}, b_i\}_k$, $i = 1, ..., N_{bc,k}$ and $k = 1, ..., K$, where (1) K is the number of boundary conditions, for instance, $K = 2$ in the first test case since we have two boundary conditions in this particular case, and (2) $N_{bc,k}$ is the number of points for the k_{th} boundary condition. For a batched inputs, the kth boundary condition has a dimension $B_{OL} \times N_{bc,k} \times (D_{\boldsymbol{x}} + 1)$. Generally, we can utilize the same number of points for different boundary conditions, and hence we will omit k in $N_{bc,k}$ without confusion in what follows. Also, p or $N_{bc} \times (D_{\boldsymbol{x}} + 1)$ is referred to as the length of the sequence in Transformer. It will be computationally expensive if the length is too long. A commonly employed way to address this issue is to decompose the long sequence into smaller ones without overlapping. For instance, the representation for f becomes $\Psi'_{\mathcal{U}_f} \in R^{B_{OL} \times N_p \times p'}$ with $p = N_p \times p'$. The similar approach can also be applied to the boundary conditions, which will not be discussed in detail for simplicity. Also, we can deal with the initial conditions as well as the geometry in the same way for the treatment of boundary conditions since they are all represented by discrete values. Furthermore, we employ a linear mapping to the representations for f and b to guarantee that the number of the last dimension for each representation is the same, i.e. D, where the parameters in this mapping are trainable. We then concatenate all the inputs at the second dimension, which plays the same role as the tokens in the Transformer for NLP. Finally, the positional encoding used in [37] is employed here for all tokens.

Note that (1) the Mean Pooling in Fig. 5 is applied to the second dimension for the corresponding input, and (2) the remaining details in the Transformer shown in Fig. 5 are kept the same the original one developed in [34], e.g., Multi-Head Attention, Add & Norm, etc., and will not be introduced in detail here.

Appendix B. Details on the computations

In all the cases discussed in Sec. 3, the Adam optimizer and a cosine annealing schedule for the learning rate are utilized for training all neural networks. Additional details regarding the architectures and training steps for FE and Transformer-based operator learning are provided in Tables B.8 - B.11.

Table B.8: Architecture and training steps of FE in each case.

	$\mathcal{NN}_C$		$\mathcal{NN}_B$		Training steps
	width × depth	Activation	width × depth	Activation	
Sec. 3.1	512×4	ReLU	512×4	tanh	1200,000
Sec. 3.2	CNN	ReLU	512×4	tanh	50,000
Sec. 3.3	CNN	ReLU	[512,512,1024,1024]	tanh	500,000
Sec. 3.4	CNN	ReLU	512×4	tanh	30,000

Table B.9: Batch size and training points for each case in FE.

	Batch size	**Training points for each snapshot**
Sec. 3.1	4096	101 (equidistantly distributed in $[-1, 1]$)
Sec. 3.2	512	101 × 101 (uniform grid in $[0, 1] \times [-1, 1]$)
Sec. 3.3	400	1024 (uniform distributed in $[-1, 1] \times [-1, 1]$)
Sec. 3.4	512	99 × 101 (uniform grid in $[0, 1] \times [-1, 1]$)

Table B.10: Architecture of Transformer-based operator learning in each case.

	Model-dim	Block Num.	Heads	FFN hidden dim	FFN Activation
Sec. 3.1	256	2	4	256	ReLU
Sec. 3.2	256	2	4	256	ReLU
Sec. 3.3	256	4	8	1024	GeLU
Sec. 3.4	512	4	4	512	ReLU

Table B.11: Batch size and training steps for each case in Transformer.

	Batch size	**Training steps**
Sec. 3.1	4096	500,000
Sec. 3.2	512	58,680
Sec. 3.3	400	100,000
Sec. 3.4	2048	73,350

References

[1] R. Schunk, Mathematical structure of transport equations for multi-species flows, Reviews of Geophysics 15 (4) (1977) 429–445.

[2] D. Zhang, L. Lu, L. Guo, G. E. Karniadakis, Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems, Journal of Computational Physics 397 (2019) 108850.

[3] S. Mazumder, Boltzmann transport equation based modeling of phonon heat conduction: progress and challenges, Annual Review of Heat Transfer 24 (2021).

[4] Q. Lin, C. Zhang, X. Meng, Z. Guo, Monte carlo physics-informed neural networks for multiscale heat conduction via phonon boltzmann transport equation, arXiv preprint arXiv:2408.10965 (2024).

[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] 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, Journal of Computational physics 378 (2019) 686–707.

[7] Z. Long, Y. Lu, X. Ma, B. Dong, Pde-net: Learning pdes from data, in: International conference on machine learning, PMLR, 2018, pp. 3208–3216.

[8] J. Sirignano, K. Spiliopoulos, Dgm: A deep learning algorithm for solving partial differential equations, Journal of computational physics 375 (2018) 1339–1364.

[9] B. Yu, et al., The deep ritz method: a deep learning-based numerical algorithm for solving variational problems, Communications in Mathematics and Statistics 6 (1) (2018) 1–12.

[10] L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, Nature machine intelligence 3 (3) (2021) 218–229.

[11] Q. Cao, S. Goswami, G. E. Karniadakis, Laplace neural operator for solving differential equations, Nature Machine Intelligence 6 (6) (2024) 631–640.

[12] O. Ovadia, A. Kahana, P. Stinis, E. Turkel, D. Givoli, G. E. Karniadakis, Vito: Vision transformer-operator, Computer Methods in Applied Mechanics and Engineering 428 (2024) 117109.

[13] C.-W. Cheng, J. Huang, Y. Zhang, G. Yang, C.-B. Schönlieb, A. I. Aviles-Rivero, Mamba neural operator: Who wins? transformers vs. state-space models for pdes, arXiv preprint arXiv:2410.02113 (2024).

[14] B. Shih, A. Peyvan, Z. Zhang, G. E. Karniadakis, Transformers as neural operators for solutions of differential equations with finite regularity, Computer Methods in Applied Mechanics and Engineering 434 (2025) 117560.

[15] S. Cao, Choose a transformer: Fourier or galerkin, Advances in neural information processing systems 34 (2021) 24924–24940.

[16] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, G. E. Karniadakis, A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data, Computer Methods in Applied Mechanics and Engineering 393 (2022) 114778.

[17] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, A. Anandkumar, Neural operator: Learning maps between function spaces with applications to pdes, Journal of Machine Learning Research 24 (89) (2023) 1–97.

[18] K. Azizzadenesheli, N. Kovachki, Z. Li, M. Liu-Schiaffini, J. Kossaifi, A. Anandkumar, Neural operators for accelerating scientific simulations and design, Nature Reviews Physics 6 (5) (2024) 320–328.

[19] P. Jin, S. Meng, L. Lu, Mionet: Learning multiple-input operators via tensor product, SIAM Journal on Scientific Computing 44 (6) (2022) A3490–A3514.

[20] K. Shukla, V. Oommen, A. Peyvan, M. Penwarden, N. Plewacki, L. Bravo, A. Ghoshal, R. M. Kirby, G. E. Karniadakis, Deep neural operators as accurate surrogates for shape optimization, Engineering Applications of Artificial Intelligence 129 (2024) 107615.

[21] Z. Li, N. Kovachki, C. Choy, B. Li, J. Kossaifi, S. Otta, M. A. Nabian, M. Stadler, C. Hundt, K. Azizzadenesheli, et al., Geometry-informed

neural operator for large-scale 3d pdes, Advances in Neural Information Processing Systems 36 (2023) 35836–35854.

[22] Z. Ye, Z. Liu, B. Wu, H. Jiang, L. Chen, M. Zhang, X. Huang, Q. M. Zou, H. Liu, B. Dong, et al., Pdeformer-2: A versatile foundation model for two-dimensional partial differential equations, arXiv preprint arXiv:2507.15409 (2025).

[23] S. Subramanian, P. Harrington, K. Keutzer, W. Bhimji, D. Morozov, M. W. Mahoney, A. Gholami, Towards foundation models for scientific machine learning: Characterizing scaling and transfer behavior, Advances in Neural Information Processing Systems 36 (2023) 71242–71262.

[24] J. Sun, Y. Liu, Z. Zhang, H. Schaeffer, Towards a foundation model for partial differential equations: Multioperator learning and extrapolation, Physical Review E 111 (3) (2025) 035304.

[25] M. Herde, B. Raonic, T. Rohner, R. Käppeli, R. Molinaro, E. de Bézenac, S. Mishra, Poseidon: Efficient foundation models for pdes, Advances in Neural Information Processing Systems 37 (2024) 72525–72624.

[26] Y. Liu, J. Sun, X. He, G. Pinney, Z. Zhang, H. Schaeffer, Prose-fd: A multimodal pde foundation model for learning multiple operators for forecasting fluid dynamics, arXiv preprint arXiv:2409.09811 (2024).

[27] R. Balestriero, M. Ibrahim, V. Sobal, A. Morcos, S. Shekhar, T. Goldstein, F. Bordes, A. Bardes, G. Mialon, Y. Tian, et al., A cookbook of self-supervised learning, arXiv preprint arXiv:2304.12210 (2023).

[28] X. Yang, Z. Song, I. King, Z. Xu, A survey on deep semi-supervised learning, IEEE transactions on knowledge and data engineering 35 (9) (2022) 8934–8954.

[29] L. Yang, D. Zhang, G. E. Karniadakis, Physics-informed generative adversarial networks for stochastic differential equations, SIAM Journal on Scientific Computing 42 (1) (2020) A292–A317.

[30] R. M. Neal, Bayesian learning for neural networks, Vol. 118, Springer Science & Business Media, 2012.

[31] G. Pang, L. Yang, G. E. Karniadakis, Neural-net-induced gaussian process regression for function approximation and pde solution, Journal of Computational Physics 384 (2019) 270–288.

[32] T. Pearce, R. Tsuchida, M. Zaki, A. Brintrup, A. Neely, Expressive priors in bayesian neural networks: Kernel combinations and periodic functions, in: Uncertainty in artificial intelligence, PMLR, 2020, pp. 134–144.

[33] C. K. Williams, C. E. Rasmussen, Gaussian processes for machine learning, Vol. 2, MIT press Cambridge, MA, 2006.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[35] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, et al., A survey on vision transformer, IEEE transactions on pattern analysis and machine intelligence 45 (1) (2022) 87–110.

[36] E. Kharazmi, D. Fan, Z. Wang, M. S. Triantafyllou, Inferring vortex induced vibrations of flexible cylinders using physics-informed neural networks, Journal of Fluids and Structures 107 (2021) 103367.

[37] J. Gehring, M. Auli, D. Grangier, D. Yarats, Y. N. Dauphin, Convolutional sequence to sequence learning, in: International conference on machine learning, PMLR, 2017, pp. 1243–1252.