From Theory to Application: A Practical Introduction to Neural Operators in Scientific Computing

Prashant K. Jha[1]

Abstract

This focused review explores a range of neural operator architectures for approximating solutions to parametric partial differential equations (PDEs), emphasizing high-level concepts and practical implementation strategies. The study covers foundational models such as Deep Operator Networks (DeepONet), Principal Component Analysis-based Neural Networks (PCANet), and Fourier Neural Operators (FNO), providing comparative insights into their core methodologies and performance. These architectures are demonstrated on two classical linear parametric PDEs—the Poisson equation and linear elastic deformation. Beyond forward problem-solving, the review delves into applying neural operators as surrogates in Bayesian inference problems, showcasing their effectiveness in accelerating posterior inference while maintaining accuracy. The paper concludes by discussing current challenges, particularly in controlling prediction accuracy and generalization. It outlines emerging strategies to address these issues, such as residual-based error correction and multi-level training. This review can be seen as a comprehensive guide to implementing neural operators and integrating them into scientific computing workflows.

Keywords: neural operators; neural networks; operator learning; surrogate modeling; Bayesian inference

Contents

1 Introduction **2**
1.1 Organization of the article 3

2 Preliminaries **4**
2.1 Notations 4
2.2 Series representation of functions and finite-dimensional approximation 6
2.2.1 Finite element approximation 7
2.3 Dimensional reduction and singular-value decomposition (SVD) 8
2.3.1 Projectors via SVD 8
2.4 Probability sampling of functions aka infinite-dimensional random variables 10
2.4.1 Gaussian measures based on Laplacian-like operators 10

[1]Department of Mechanical Engineering, South Dakota School of Mines and Technology, Rapid City, SD 57701, USA.
Email address: prashant.jha@sdsmt.edu

3 Model problems **13**
3.1 Poisson problem . . . 13
3.1.1 Setup details and data generation . . . 14
3.2 Linear elasticity problem . . . 17
3.2.1 Setup details and data generation . . . 18

4 Neural networks as surrogate of the forward problem **19**
4.1 Deep Operator Network (DeepONet) . . . 19
4.1.1 Implementation of DeepONet . . . 21
4.1.2 Architecture and preliminary results . . . 25
4.2 Principal Component Analysis-based Neural Operator (PCANet) . . . 27
4.2.1 Implementation of PCANet . . . 29
4.2.2 Architecture and preliminary results . . . 31
4.3 Fourier Neural Operator (FNO) . . . 31
4.3.1 Implementation of FNO . . . 34
4.3.2 Architecture and preliminary results . . . 37

5 Neural Operators applied to Bayesian inference problems **37**
5.1 Abstract Bayesian inference problem in infinite dimensions . . . 37
5.1.1 Markov chain Monte Carlo (MCMC) method to sample from the posterior measure . . . 39
5.2 Inference of the diffusivity in Poisson problem . . . 41
5.2.1 Setup of the forward problem, prior measure, and synthetic data . . . 42
5.2.2 Inference results . . . 42
5.3 Inference of Young's modulus in linear elasticity problem . . . 45
5.3.1 Setup of the forward problem, prior measure, and synthetic data . . . 45
5.3.2 Inference results . . . 45

6 Conclusion **48**
6.1 Growing field of neural operators . . . 48
6.2 Controlling the neural operator prediction accuracy . . . 49
6.3 Final thoughts . . . 51

References **51**

1. Introduction

Neural operators have emerged as powerful tools for approximating solution operators of parametric partial differential equations (PDEs). Their key advantages include learning highly nonlinear mappings between function spaces and effectively reducing modeling errors when observational data is available, even if the underlying models are poorly formed. Additionally, their capacity for fast evaluations makes them particularly valuable in applications like real-time optimization and control.

This article offers a practical and hands-on introduction to several key neural operator architectures that have gained prominence in scientific computing. The following neural operators are explored:

1. Deep Operator Network (DeepONet) Wang et al. (2021a); Lu et al. (2021a,b); Goswami et al. (2020);

2. Principle Component Analysis/Proper Orthogonal Decomposition-based Neural Operator (PCANet/PODNet) Bhattacharya et al. (2021); Fresca and Manzoni (2022); and

3. Fourier Neural Operator (FNO) Li et al. (2021); Kovachki et al. (2021).

This introduction is designed to be self-contained, hands-on, and transparent regarding algorithmic details. Rather than providing an exhaustive review of generalizations or diverse applications, the focus remains on the foundational ideas behind these core neural operators and the practical aspects of their implementation. The aim is to equip readers with a solid understanding that can serve as a stepping stone for exploring broader extensions in the literature.

Several crucial topics are addressed to ensure a deeper understanding of neural operators:

- Sampling random functions using Gaussian measures on function spaces;
- Defining data structures specific to each neural operator;
- Algorithms and Python implementations for all critical computations, including random function sampling and Markov Chain Monte Carlo (MCMC) for Bayesian inference; and
- A detailed exploration of neural operators' application to Bayesian inverse problems.

To illustrate the use of neural operators, two classical linear parametric PDEs are used as model problems. The first model is based on the Poisson equation for a temperature distribution on a rectangular domain, where the input parameter field is the diffusivity. The second model corresponds to the in-plane deformation of a thin elastic plate, with Young's modulus being the input parameter field. For these model problems, neural operators approximating the solution operator are constructed, and their accuracy for random samples of the input function is assessed. For the demonstration, neural operators are used as a surrogate in Bayesian inference problems, where the parameter fields in the above two model problems are inferred from the observational data. The performance of neural operators as surrogates is comparable to the "true" model. The model problems considered in this work are linear and have a fast decay of singular values of input and output data. This indicates a low dimensional structure of the solution operator, making it easier to approximate using neural operators. For highly nonlinear problems and challenging inference and optimization problems, neural operators may show significant errors. This issue is discussed in the conclusion section, where some existing works on controlling neural operator prediction errors are surveyed.

1.1. Organization of the article

- Section 2 introduces notations and key mathematical preliminaries, covering finite-dimensional function approximations, singular value decomposition, and sampling random functions using Gaussian measures.
- Section 3 presents the two parametric PDEs used as model problems for developing neural operators.

- Section 4 discusses the architectures and implementations of DeepONet, PCANet, and FNO, focusing on their core principles, critical implementation details in Python, and an evaluation of their predictive performance.
- Section 5 explores the application of neural operators to Bayesian inverse problems, using them as surrogates for forward models in MCMC simulations.
- Section 6 concludes the article, summarizing key insights, referencing additional neural operator architectures not covered here, and discussing strategies for controlling prediction errors. The subsection on prediction accuracy (Section 6.2) highlights related works and outlines potential future research directions.

Codes and Jupyter notebooks for neural operator training and Bayesian inference are available at: https://github.com/CEADpx/neural_operators/ (check out tag `survey25_v1`). The data is shared separately in the Dropbox folder `NeuralOperator_Survey_Shared_Data_March2025`.

2. Preliminaries

This section collects crucial information that will provide a solid foundation for the topics covered in the rest of the sections. It begins by fixing the notations.

2.1. Notations

Let $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ denote the space of natural numbers, integers, and real numbers, respectively, and $\mathbb{R}^+$ denotes the space of all nonnegative real numbers. $\mathbb{R}^n$ denotes the n-dimensional Euclidean space. Space of L^2-integrable functions $f : D \subset \mathbb{R}^q \to \mathbb{R}^d$ is denoted by $L^2(D;\mathbb{R}^d)$; space $H^s(D;\mathbb{R}^d)$ for functions in $L^2(D;\mathbb{R}^d)$ with generalized derivatives up to order s in $L^2(D;\mathbb{R}^{q\times\sum_{i=1}^{s-1}q\times d})$. $L(M;U)$ denotes the space of continuous linear maps from M to U and $C^1(A;U)$ space of continuous and differentiable maps from $A \subset M$ to U. Given a generic complete normed (function) Banach space A, $\|\cdot\|$ denotes the norm, and if it is a Hilbert space $\langle u, v\rangle$ denotes the inner production for $u, v \in A$. A^* denotes the dual of the Banach space A and $\langle a, b\rangle$, where $a \in U^*$ and $b \in U$.

Throughout the text, $m \in M$ will denote the input or parameter field in the parametric boundary value problem, M the appropriate Banach function space for input fields, $u \in U$ the solution field, where U is the Banach space for solutions of the PDE. The so-called solution operator or forward operator that maps the parameter field $m \in M$ to the solution $u \in U$ of the PDE is denoted by $F(m)$. The neural network approximation of the forward operator $F(m)$ is denoted using $F_{NN}(m)$. The finite-dimensional approximation of functions $m \in M$ and $u \in U$ are denoted by the bold Roman letter $\mathbf{m}$ and $\mathbf{u}$, and more generally for any $a \in A$, $\mathbf{a}$. The corresponding map between discrete versions of input and output functions is $\mathbf{u} = \mathbf{F}(\mathbf{m})$. The projections of discretizations of functions, $\mathbf{m}$ and $\mathbf{u}$, onto lower-dimensional subspaces will be denoted by $\tilde{\mathbf{m}}$ and $\tilde{\mathbf{u}}$, respectively. The mapping between lower-dimensional subspaces will be denoted using $\tilde{}$ on top (e.g., $\tilde{\mathbf{F}}$).

The key notations used throughout the text are collected in Table 1.

Symbol	**Description**

m	Typical parameter field in the parametric PDEs, which is also the input function to neural operators
u	Typical solution of the PDE given m, which is also the output of the neural operator
D_a	Domain of some abstract function a
$\partial D, \Gamma_b$	Boundary of the domain D and subset of ∂D tagged by b, respectively
q_a	Dimension of the domain of function a
d_a	Dimension of the pointwise values of the function a
$M = \{m : D_m \subseteq \mathbb{R}^{q_m} \to \mathbb{R}^{d_m}\}$	Function space of the parameters in the parametric PDEs
$U = \{u : D_u \subseteq \mathbb{R}^{q_u} \to \mathbb{R}^{d_u}\}$	Function space of the solution of the PDE
$F : m \in M \mapsto F(m) = u \in U$	Forward solution operator, which is also the target operator of neural operators
$\phi_a = \{\phi_{a_i}\}_{i=1}^{\infty}$	Basis functions for the function space associated with the function a, e.g., ϕ_m for M so $m \in M$ has representation $m = \sum_i m_i \phi_{a_i}$
$x \in D_m$	Spatial coordinates in the domain of input/parameter functions
$y \in D_u$	Spatial coordinates in the domain of output/solutions
$\mathbf{a} \in \mathbb{R}^{p_a}$	Bold Roman symbol indicates the finite-dimensional representation/approximation of the function a, e.g., nodal values $(\mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_{p_m})$ in finite element discretization of $m \in M$
p_a	Dimension of the finite-dimensional representation of function a, i.e., $\mathbf{a}$ lives in $\mathbb{R}^{p_a}$
$\mathbf{a}_i$	i^{th} component of $\mathbf{a}$
$\mathbf{F} : \mathbf{m} \in \mathbb{R}^{p_m} \mapsto \mathbf{F}(\mathbf{m}) = \mathbf{u} \in \mathbb{R}^{p_u}$	Finite-dimensional approximation of the operator $F : M \to U$
r_a	Dimension of the reduced dimensional representation of $\mathbf{a} \in \mathbb{R}^{p_a}$, $r_a << p_a$
$\tilde{\mathbf{a}} \in \mathbb{R}^{r_a}$	The reduced-dimensional representation of $\mathbf{a} \in \mathbb{R}^{p_a}$
$\tilde{\mathbf{P}}_a : \mathbb{R}^{p_a} \to \mathbb{R}^{r_a}$	Projection operator that takes finite-dimensional representation of function $\mathbf{a} \in \mathbb{R}^{p_a}$ to the reduced space $\tilde{\mathbf{a}} = \tilde{\mathbf{P}}_a(\mathbf{a}) \in \mathbb{R}^{r_a}$

$\tilde{\mathbf{F}}_r : \tilde{\mathbf{m}} \in \mathbb{R}^{r_m} \mapsto \tilde{\mathbf{F}}_r(\tilde{\mathbf{m}}) = \tilde{\mathbf{u}} \in \mathbb{R}^{r_u}$	Reduced-order approximation of $\mathbf{F}$ mapping between reduced (latent) spaces of input and output functions
$N(m, C)$	Gaussian random field with mean $m \in M$ and covariance operator $C : M \times M \to M$
$\mathbf{N}(\mathbf{m}, \mathbf{C})$	Gaussian random field in finite-dimensional space $\mathbb{R}^{p_m}$, where $\mathbf{m} \in \mathbb{R}^{p_m}$ and $\mathbf{C} : \mathbb{R}^{p_m} \times \mathbb{R}^{p_m}$
$F_{NN} : M \times \Theta \to U$	Typical neural operator approximation of $F : M \to U$ with trainable neural network parameters $\Theta \in \mathbb{R}^{p_{nn}}$
$\mathbf{F}_{NN} : \mathbb{R}^{p_m} \times \Theta \to \mathbb{R}^{p_u}$	Neural operator approximation of finite-dimensional representation of the map $\mathbf{F} : \mathbb{R}^{p_m} \to \mathbb{R}^{p_u}$
N	Number of data for training and testing neural operators
$m^I, u^I, \mathbf{m}^I, \mathbf{u}^I$	I^{th} sample of input and output functions in functional and finite-dimensional settings
$\mathbf{X} = (\mathbf{m}^1, \mathbf{m}^2, ..., \mathbf{m}^N)^T \in \mathbb{R}^{N \times p_m}$	Input data to the neural operator, where $\mathbf{m}^I \in \mathbb{R}^{p_m}$ is seen as a column vector
$\mathbf{X}_{tr} \in \mathbb{R}^{N \times N_{tr} \times q_m}$	Input data to the trunk network of the DeepONet neural operator, where for each I, $1 \leq I \leq N$, $\mathbf{X}_{tr}^I$ is $N_{tr} \times q_m$ and it consists of N_{tr} number of spatial coordinates in the domain $D_u \subseteq \mathbb{R}^{q_u}$
$\mathbf{Y} = (\mathbf{u}^1, \mathbf{u}^2, ..., \mathbf{u}^N)^T \in \mathbb{R}^{N \times p_u}$	Output data to the neural operator
$\mathbf{X}^I, \mathbf{Y}^I$	I^{th} sample, $1 \leq I \leq N$, of the data from $\mathbf{X}, \mathbf{Y}$, respectively

Table 1: Key notations used in this text.

2.2. Series representation of functions and finite-dimensional approximation

One of the central ideas that various neural operators leverage is the finite-dimensional representation of functions consisting of coefficients and basis functions in their respective spaces. Following the notations in the previous section and Table 1, suppose that M and U are Hilbert spaces, and, therefore, have orthonormal sequences $\phi_m = \{\phi_{m_i}\}_{i=1}^{\infty}$ and $\phi_u = \{\phi_{u_i}\}_{i=1}^{\infty}$, respectively, so that

$$m(x) = \sum_{i=1}^{\infty} \mathbf{m}_i \phi_{m_i}(x)\,, \qquad \forall x \in D_m\,, \tag{1}$$

where $\mathbf{m}_i = \langle m, \phi_{m_i} \rangle$ are the coefficients or degrees of freedom associated with the i^{th} mode. It is useful to consider the example with $D_m = (0, 1)$ and $m \in L^2(D_m; \mathbb{R})$. In this case, one can write

$m = \sum_{i=1}^{\infty} \mathbf{m}_i \phi_{m_i}(x)$, where basis functions take the form

$$\phi_{m_1} = 1\,, \Phi_{m_2} = \frac{\cos(2\pi x)}{\sqrt{2}}\,, \phi_{m_3} = \frac{\sin(2\pi x)}{\sqrt{2}}\,, \cdots, \phi_{m_{2j}} = \frac{\cos(2j\pi x)}{\sqrt{2}}\,, \phi_{m_{2j+1}} = \frac{\sin(2j\pi x)}{\sqrt{2}}\,, \cdots \quad (2)$$

and the coefficients are given by

$$\mathbf{m}_i = \langle m, \phi_{m_i} \rangle = \int_0^1 m(x)\phi_{m_i}(x)\,\mathrm{d}x\,, \qquad \forall i\,. \quad (3)$$

Focusing on the abstract setting, let $\{\phi_{m_i}\}_{i=1}^{p_m}$, where p_m a finite integer, are the finite collection of basis functions, and $\{\mathbf{m}_i\}_{i=1}^{p_m}$ are the corresponding coefficients. Then, the finite-dimensional approximation is $\sum_{i=1}^{p_m} \mathbf{m}_i \phi_{m_i}(x) \approx m(x)$ with the error given by $\|m - (\sum_{i=1}^{p_m} \mathbf{m}_i \phi_{m_i})\|$. The same can be done for the function $u \in U$ to have $u(y) \approx \sum_{i=1}^{p_u} \mathbf{u}_i \phi_{u_i}(y)$, $y \in D_u$. Very often, the neural operator will try to imitate this finite-dimensional approximation technique, where the goal will be to find (learn) the bases $\{\phi_{u_i}\}_{i=1}^{p_u}$ (or its pointwise values $\{\phi_{u_i}(y)\}_{i=1}^{p_u}$ for $y \in D_u$ [e.g., in DeepONet]) and the coefficients $\{\mathbf{u}_i\}_{i=1}^{p_u}$ such that $\sum_{i=1}^{p_u} \mathbf{u}_i \phi_{u_i}$ provides the "best" approximation of $u = F(m)$, $m \in M$ being the input function to the operator.

2.2.1. Finite element approximation

For a general class of function space $M \subseteq \{m : D_m \subseteq \mathbb{R}^{q_m} \to \mathbb{R}^{d_m}\}$ and spatial domain D_m, the theory above to develop a finite-dimensional approximation of functions can be restrictive. The more straightforward and numerical way to obtain the finite-dimensional approximation is using numerical techniques such as finite difference and finite element approximation. In this work, the finite element method is used (e.g., to generate samples of input functions using Gaussian priors, solve PDE-based problems, and solve the Bayesian inference). To be more precise, consider a finite element discretization D_{m_h} of the domain D_m consisting of simplex elements $\{T_e\}_{e=1}^{N_e}$ so that $D_{m_h} = \cup_e \bar{T}_e \approx D_m$. Suppose ϕ_{m_i} is the linear Lagrange basis of the i^{th} vertex. Let $V_{m_h} = \text{span}\{\phi_{m_i}\}_{i=1}^{p_m}$, p_m being the number of vertices. Then, the function $m \in M$ can be approximated by a function $m_h \in V_{m_h}$ given by

$$m(x) \qquad \approx \qquad m_h(x) = \sum_{i=1}^{p_m} \mathbf{m}_i \phi_{m_i}(x)\,, \qquad \forall x \in D_{m_h}\,, \quad (4)$$

provided the coefficients $\mathbf{m} \in \mathbb{R}^{p_m}$ is selected appropriately. For example, $\mathbf{m}$ is selected such that it minimizes the L^2 error $e = ||m - m_h||_{L^2}$).

Given finite dimensional approximations m_h and u_h of m and u, respectively, the map $F(m) = u$ is also approximated by $F_h(m_h) = u_h$ in the sense that F_h takes m_h and returns the output h_h such that the error

$$||F(m) - F_h(m_h)|| \quad (5)$$

is small for the appropriate collection of m.

Before concluding this section, note that, for the fixed mesh and the basis functions ϕ_m, it is easy to see that if $\mathbf{m} \in \mathbb{R}^{p_m}$ is fixed, then the function m_h is completely characterized. If the m_h is fixed, using the unique representation of m_h, the coefficients $\mathbf{m}$ are completely characterized. So, V_{m_h} can be identified using $\mathbb{R}^{p_m}$ (and vice versa). This makes it possible to represent the finite-dimensional function space V_{m_h} by the Euclidean space $\mathbb{R}^{p_m}$ of the coefficients. Throughout the paper, functions m and u will be represented by the finite-dimensional approximations $\mathbf{m}$ and

$\mathbf{u}$, where the functional representation of coefficients $\mathbf{m}$ and $\mathbf{u}$, m and u, respectively, is assumed implicitly. In the same spirit, since $u_h = F_h(m_h)$ (for a given m_h) can be identified by $\mathbf{u}$, a map $\mathbf{F} : \mathbb{R}^{p_m} \to \mathbb{R}^{p_u}$ is defined as follows:

$$\mathbf{F}(\mathbf{m}) = \mathbf{u} \qquad \Rightarrow \qquad F_h(m_h) = u_h = \sum_{i=1}^{p_u} \mathbf{u}_i \phi_{u_i} \quad \text{with} \quad m_h = \sum_{i=1}^{p_m} \mathbf{m}_i \phi_{m_i} \,. \tag{6}$$

Here, $\mathbf{F}$ maps the coefficient vector $\mathbf{m}$ to $\mathbf{u}$ and is induced by the map F.

2.3. Dimensional reduction and singular-value decomposition (SVD)

While the theory is based on functions defined on a continuum domain, computer implementations introduce discretization of the domain and, consequently, discrete approximation of functions. For example, the training data for neural operators is typically a collection of pairs $(\mathbf{m}^I, \mathbf{u}^I)$ where $\mathbf{m}^I \in \mathbb{R}^{p_m}$ and $\mathbf{u}^I = \mathbf{F}(\mathbf{m}^I) \in \mathbb{R}^{p_u}$ are discrete approximations of functions in M and U, $\mathbf{F}$ being the finite-dimensional mapping between $\mathbb{R}^{p_m}$ and $\mathbb{R}^{p_u}$ approximating the operator of interest $F(m)$. Generally speaking, the dimensions of input and target functions, p_m and p_u, are large, and the problem of approximating the map $\mathbf{F}$ between high dimensional spaces becomes challenging and quite possibly ill-posed.

The second key idea, the first being the linear basis representation discussed earlier, used in various neural operators is reducing the dimensions of discretized input and output functions; see Bhattacharya et al. (2021). If $\mathbf{m} \in \mathbb{R}^{p_m}$ and $\mathbf{u} \in \mathbb{R}^{p_u}$, and the goal is to determine a map $\mathbf{m} \mapsto \mathbf{u} = \mathbf{F}(\mathbf{m})$ from the data $\{(\mathbf{m}^I, \mathbf{u}^I)\}_{I=1}^N$, then, alternative to learning/approximating the map $\mathbf{F}$, one could attempt to characterize the map $\tilde{\mathbf{F}}$, where

$$\mathbf{m} \quad \mapsto \quad \mathbf{u} = \tilde{\mathbf{P}}_u^T \left(\tilde{\mathbf{F}} \left(\tilde{\mathbf{P}}_m(\mathbf{m}) \right) \right) . \tag{7}$$

Here, $\tilde{\mathbf{P}}_m \in \mathbb{R}^{r_m \times p_m}$ is the projection operator that projects $\mathbf{m} \in \mathbb{R}^{p_m}$ into a lower dimensional subspace, $\tilde{\mathbf{m}} := \tilde{\mathbf{P}}_m(\mathbf{m}) \in \mathbb{R}^{r_m}$ (with $r_m << p_m$). $\tilde{\mathbf{P}}_u \in \mathbb{R}^{r_u \times p_u}$ has the same role as $\tilde{\mathbf{P}}_m$ but for target functions $u \in \mathbb{R}^{p_u}$. The transpose of $\tilde{\mathbf{P}}_u$, $\tilde{\mathbf{P}}_u^T$, projects the element in $\mathbb{R}^{r_u}$ into $\mathbb{R}^{p_u}$. Note that $\tilde{\mathbf{F}} : \mathbb{R}^{r_m} \to \mathbb{R}^{r_u}$ that needs to be learned is the mapping between two smaller dimensional spaces, and, hence, identifying $\tilde{\mathbf{F}}$ is less daunting compared to $\mathbf{F}$. In summary, using $\tilde{\mathbf{P}}_m$ and $\tilde{\mathbf{P}}_u$, the dimensions of the operator inference problem are significantly reduced, and, by controlling r_m and r_u, one can balance the trade-off between accuracy and computational cost.

2.3.1. Projectors via SVD

The projectors $\tilde{\mathbf{P}}_m$ and $\tilde{\mathbf{P}}_u$ for dimensional reduction can be obtained via singular-value decomposition (SVD). Focusing on the input space $\mathbb{R}^{p_m}$, let $\mathbf{R}$ denote a $p_m \times N$ matrix such that, for $1 \leq I \leq N$, $\mathbf{m}^I$ makes up the I^{th} column of the matrix $\mathbf{R}$. Let $r_R = \text{rank}(\mathbf{R}) \leq \min\{p_m, N\}$ be the rank of the matrix, and $\mathbf{R} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ its singular-value decomposition, where $\mathbf{U}$ and $\mathbf{V}$ are column-orthonormal matrices of sizes $p_m \times p_m$ and $N \times N$, respectively, and $\mathbf{D}$ is a $p_m \times N$ diagonal matrix. Focusing on the matrix $\mathbf{U}$, the i^{th} column is denoted by a vector $\mathbf{w}_i \in \mathbb{R}^{p_m}$. The set of vectors (columns of $\mathbf{U}$) $\{\mathbf{w}_i\}_{i=1}^{p_m}$ form the orthonormal basis for $\mathbb{R}^{p_m}$.

Let $r_m > 0$ such that $r_m \leq \text{rank}(\mathbf{R})$ is the dimension of the reduced space $\mathbb{R}^{r_m}$ for which a projector $\mathbf{P}_m : \mathbb{R}^{p_m} \to \mathbb{R}^{r_m}$ is sought. Given r_m, a matrix $\mathbf{U}_{r_m}$ of size $p_m \times r_m$ is constructed by

removing the $(p_m - r_m)$ columns of $\mathbf{U}$ from the end:

$$\mathbf{U}_{r_m} = \begin{bmatrix} | & | & & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_{r_m} \\ | & | & & | \end{bmatrix}. \tag{8}$$

Noting the properties of $\mathbf{U}_{r_m}$ (e.g., see (Jha, 2024, Section 3.2.1)), $\mathbf{U}_{r_m}^T$ is taken as the projector, i.e., $\tilde{\mathbf{P}}_m := \mathbf{U}_{r_m}^T$.

For $\mathbf{u}$, the projector $\tilde{\mathbf{P}}_u$ is obtained following the same procedure as above using a matrix $\mathbf{R}$ of size $p_u \times N$ with $\mathbf{u}^I$ making its I^{th} column. SVD of $\mathbf{R}$, say $\mathbf{U}$, is truncated by retaining the first r_u columns of $\mathbf{U}$. If the truncated matrix is $\mathbf{U}_{r_u}$ then $\tilde{\mathbf{P}}_u := \mathbf{U}_{r_u}^T$.

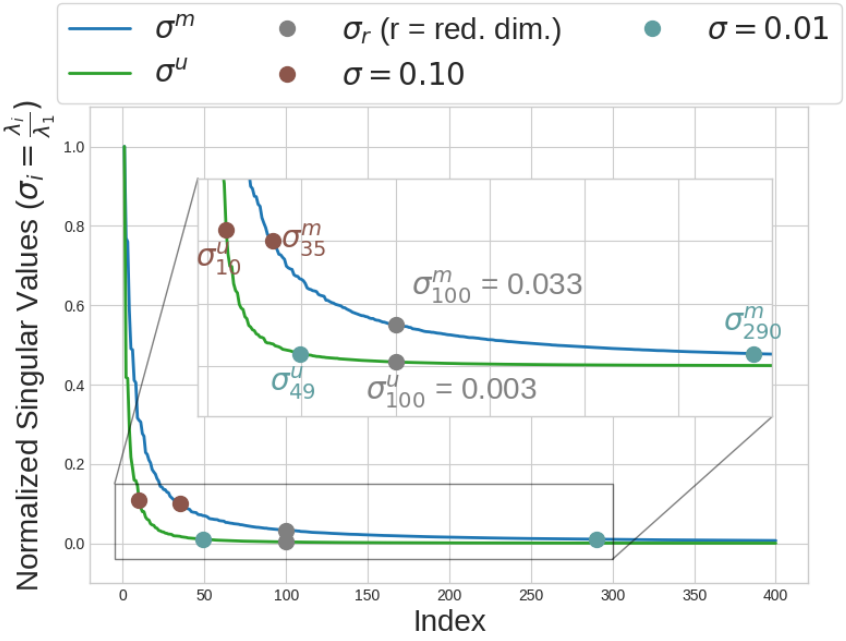


Figure 1: Singular values of input and output data (centered and normalized) $\{\mathbf{m}_i = \exp(\mathbf{w}_i) | \mathbf{w}_i \sim \mathbf{N}(\mathbf{0}, \mathbf{C})\}$, and $\{\mathbf{u}_i = \mathbf{F}(\mathbf{m}_i)\}$, where $\mathbf{N}(\mathbf{0}, \mathbf{C})$ is the p_m-dimensional Gaussian density obtained via the finite element approximation of the random Gaussian field in function space $M := L^2(D_m; \mathbb{R})$ (see Section 2.4 for details), $\mathbf{m}_i$ discretized input to the parametric PDE, and $\mathbf{F}$ a discretized solution operator associated with the PDE. σ^a, $a \in \{m, u\}$, represents the normalized singular values. Small dots show corresponding modes when the normalized singular value is 0.01 or 0.1. The dimension of the reduced space is 100, and the grey dots show the corresponding singular value in the plot.

In Figure 7, the normalized singular values of representative centered and normalized input and output data are shown. The grey, brown, and cadet blue dots on each curve represent the singular value at mode 100, the mode with a singular value of 0.1, and the mode with a singular value of 0.01. Based on the plot, if u is projected using SVD to 100, 10, and 49, the average projection error will be around 0.3%, 10%, and 1%, respectively, relative to the most significant singular value. Similarly, for m, projection into reduced dimension $r = 100$, 35, and 290 will result in the average projection error (approx.) 3.3%, 10%, and 1%, respectively.

2.4. Probability sampling of functions aka infinite-dimensional random variables

The final topic to conclude this section is sampling random parameters, which are functions. Consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is a sample space and $\mathcal{F}$ is a σ-algebra on which probability measure $\mathbb{P}$ is defined with $\mathbb{P}(\Omega) = 1$. The goal is to draw W-valued random fields, where W is assumed to be a separable Hilbert space (i.e., design a random field $\mathcal{Z} : \Omega \to W$ such that given $z \in \Omega$, $w = \mathcal{Z}(z) \in W$). Suppose such an $\mathcal{Z}$ is designed; then, the probability that values of $\mathcal{Z}$ are in some subset $A \subseteq W$ is the pushforward measure $\mu_{\mathcal{Z}}$ of A given by

$$\mu_{\mathcal{Z}}(A) = \text{probability of } \mathcal{Z} \in A = \mathbb{P}\left(\{z \in \Omega : \mathcal{Z}(z) \in A\}\right) = \mathbb{P}\left(\mathcal{Z}^{-1}(A)\right) , \tag{9}$$

where $\mathcal{Z}^{-1}(A) \in \mathcal{F}$ is assumed to be measurable in probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Thus, $\mu_{\mathcal{Z}}$ is a measure on W induced by the random field $\mathcal{Z}$. Now, suppose that a random field $\mathcal{Z}$ is such that the measure $\mu_{\mathcal{Z}}$ is Gaussian in the sense of Dashti and Stuart (2017); Mandel (2023), (e.g., see (Dashti and Stuart, 2017, Definition 6 and Lemma 23)). In this case, $\mu_{\mathcal{Z}}$ is written as $\mu_{\mathcal{Z}} = N(\bar{w}, C)$. Here, $\bar{w} \in W$ is the mean function, and $C : W \to W$ is called the covariance operator. At the outset, C is assumed to be a trace-class operator; see (Dashti and Stuart, 2017, Lemma 23) and (Mandel, 2023, Theorem 7).

Our next goal is to consider specific examples of $\mathcal{Z}$ such that $\mu_{\mathcal{Z}}$ is Gaussian, as mentioned above, and see how the random samples of functions are generated. In this direction, it is useful to highlight the role of C in generating samples $w = \mathcal{Z}(z) \in W$; this will help to understand why the widely-used forms of C make sense and work. Consider another random field $\mathcal{S} : \Omega \to W$ such that $\mu_{\mathcal{S}} = N(0, 1)$, where $0 \in W$ is the mean function and $1 : C \to C$ is the identity covariance operator. Given $\bar{w} \in W$ and $C : W \to W$, a sample $w = \mathcal{Z}(z)$ ($\mathcal{Z}$ such that $\mu_{\mathcal{Z}} = N(\bar{w}, C)$) is computed by transforming the sample $s = \mathcal{S}(z)$ as follows:

$$\mathcal{Z}(z) = w := \bar{w} + C^{1/2} s \, . \tag{10}$$

Thus, $C^{1/2}$, the square root of the covariance operator, plays a key role in generating random functions. As such, C should be designed so that $C^{1/2}$ is more straightforward to apply while satisfying the properties such that the W-valued samples are well-defined, have desired regularity, and have bounded correlation between pointwise values.

2.4.1. Gaussian measures based on Laplacian-like operators

Following Bui-Thanh et al. (2013), let $C = L_{\Delta}^{-2}$, where $L_{\Delta} : W_{L_{\Delta}} \subset W \to W$ is a Laplacian-like operator given by

$$L_{\Delta} := \begin{cases} -\mathsf{a}_c \nabla \cdot \mathsf{b}_c \nabla + \mathsf{c}_c \, , & \text{in } D_w \, , \\ \gamma n \cdot \mathsf{b}_c \nabla \, , & \text{on } \partial D_w \, . \end{cases} \tag{11}$$

Here, D_w is the domain of functions w, and $\mathsf{a}_c, \mathsf{b}_c, \mathsf{c}_c$ are parameters in the operator (they could vary over the domain or be taken as constants). In this work, a_c and c_c will be constant, and in some situations, b_c will be considered to be a spatially varying scalar-valued $L^2(D_w)$ function. In the literature, it is also common to take b_c as $\mathbb{R}^{q_w \times d_w} \times \mathbb{R}^{q_w \times d_w}$-valued function allowing one to encode anisotropy and inhomogeneous behavior in the prior. In the above, $W_{L_{\Delta}} \subseteq W$ is the domain of operator L_{Δ} such that $L_{\Delta}(w)$ is well-defined for all $w \in L_{\Delta}$. The natural choice is $W_{L_{\Delta}} = \{w \in W : ||w||_{H^2} < \infty\}$. If L_{Δ} is defined in a weak form, i.e.,

$$\langle v, L_{\Delta}(w) \rangle = \int_{D_w} \left[\mathsf{a}_c \mathsf{b}_c \nabla w \cdot \nabla v + \mathsf{c}_c w v \right] \mathrm{d}x \, , \qquad \forall w, v \in W \cap H^1(D_w; \mathbb{R}^{d_w}) \, , \tag{12}$$

then W_{L_Δ} can be taken as $W_{L_\Delta} = W \cap H^1(D_w; \mathbb{R}^{d_w})$.

With the above specific form of C and what was discussed earlier, the sampling of functions $w \in W$ can be summarized as follows:

1. Fix $\bar{w} \in W$, $C = L_\Delta^{-2}$, and $\mathcal{S} : \Omega \to W$ a random field such that $\mu_{\mathcal{S}} = N(0,1)$;
2. Draw a sample $s = \mathcal{S}(z)$, $z \in \Omega$; and
3. Compute

$$\begin{aligned} & w = \bar{w} + C^{1/2}(s) \\ \Rightarrow\ & w = \bar{w} + L_\Delta^{-1}(s) \\ \Rightarrow\ & w = \bar{w} + v\,, \text{ where } v \text{ satisfies: } s = L_\Delta(v) \\ \Rightarrow\ & w = \bar{w} + v\,, \text{ where } v \text{ satisfies: } \langle g, s\rangle = \langle g, L_\Delta(v)\rangle\,, \quad \forall g \in W\,, \\ \Rightarrow\ & w = \bar{w} + v\,, \text{ where } v \text{ satisfies: } \int_{D_w} sg\,\mathrm{d}x = \int_{D_w} [\mathsf{a}_c\mathsf{b}_c\nabla v \cdot \nabla g + \mathsf{c}_c v g]\,\mathrm{d}x\,,\ \forall g \in W\,, \end{aligned} \tag{13}$$

   where the last two equations spell out the weak form definition of L_Δ that can be solved using the finite element method.

In terms of the numerical implementation, consider a finite element mesh D_{w_h} and finite element function space $V_h \subset W$ with $\dim(V_h) = p_w$, and proceed as follows:

1. Draw $s = \mathcal{S}(z) \in V_h$:
   (a) For each i, $1 \le i \le p_w$, draw a number $\mathbf{s}_i \sim N(0,1)$, where $N(0,1)$ is a standard normal probability density on $\mathbb{R}$; and
   (b) Take $s = \sum_i \mathbf{s}_i \phi_i$, where $\{\phi_i\}$ are the finite element basis functions.
2. Solve for $v \in V_h$ such that $L_\Delta(w) = s$ in V_h:
   (a) Assemble matrix and load vector from the variational form

$$\int_{D_w} sg\,\mathrm{d}x = \int_{D_w} [\mathsf{a}_c\mathsf{b}_c\nabla v \cdot \nabla g + \mathsf{c}_c v g]\,\mathrm{d}x\,, \tag{14}$$

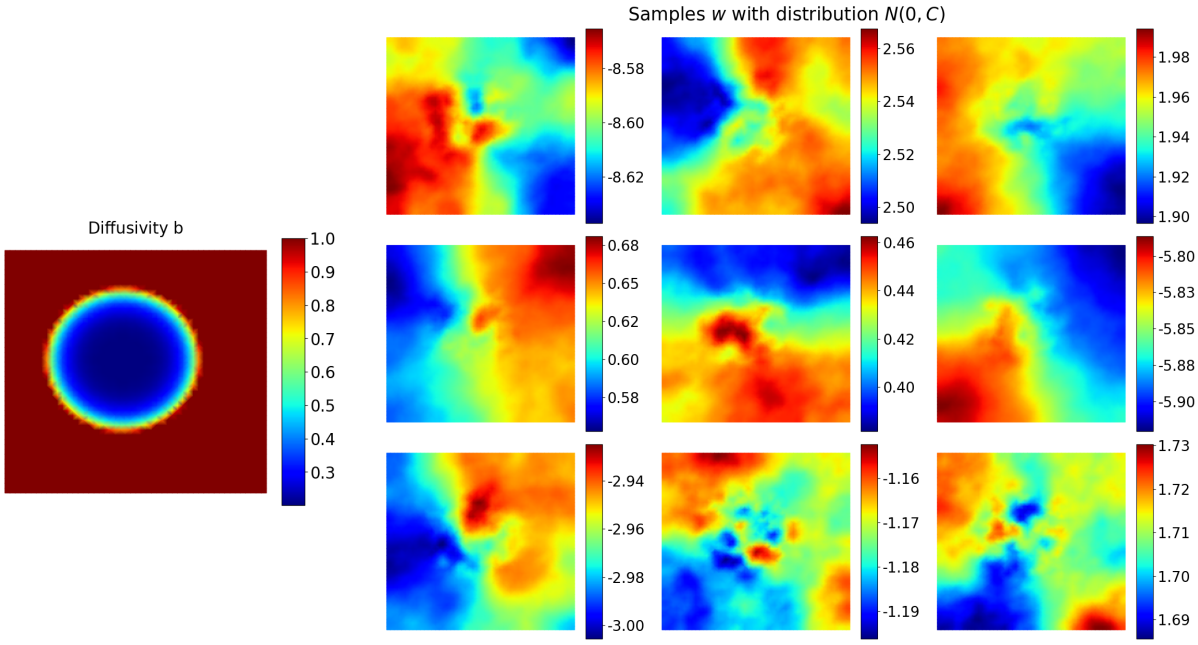
   where $v \in V_h$ and $g \in V_h$ are trial and test functions, respectively; and
   (b) Solve the underlying matrix problem to get $\mathbf{v} \in \mathbb{R}^{p_w}$ and $v = \sum_i \mathbf{v}_i \phi_i$.
3. A new sample w is then $w = \bar{w} + v$.
4. Optionally, since the state space V_h is finite-dimensional, compute the negative log of probability density $\pi(w)$ using

$$\begin{aligned} -\log \pi(w) &= ||w - \bar{w}||_C + \text{const.} \\ &= \langle C^{-1/2}(w - \bar{w}), C^{-1/2}(w - \bar{w})\rangle_W + \text{const.} \\ &= \langle C^{-1/2}C^{1/2}s, C^{-1/2}C^{1/2}s\rangle_W + \text{const.} \\ &= \langle s, s\rangle_W + \text{const.} \\ &= \mathbf{s} \cdot (\mathbf{M}\mathbf{s}) + \text{const.} \end{aligned} \tag{15}$$

where $\mathbf{M}$ is the mass matrix such that $\mathbf{M}_{ij} = \int_{D_w} \phi_i \phi_j \, \mathrm{d}x$. In Listing 1, the above algorithm is implemented using a python library `FEniCS` Alnæs et al. (2015). In Figure 2, results from Gaussian sampler with parameters $\mathsf{a}_c = 0.01$, $\mathsf{c}_c = 0.2$, inhomogeneous diffusivity function b_c, and zero mean $\bar{w} = 0 \in W$ are shown.

```
...
import numpy as np
import dolfin as dl
...

class PriorSampler:

    def __init__(self, V, gamma, delta, seed = 0):
        ...
        # function space
        self.V = V

        # vertex to dof vector and dof vector to vertex maps
        self.V_vec2vv, self.V_vv2vec = build_vector_vertex_maps(self.V)
        ...
        self.a_form = self.a*self.b_fn\
                        *dl.inner(dl.nabla_grad(self.u_trial), \
                                  dl.nabla_grad(self.u_test))*dl.dx \
                    + self.c*self.u_trial*self.u_test*dl.dx
        self.L_form = self.s_fn*self.u_test*dl.dx
        ...

    def assemble(self):
        self.lhs = dl.assemble(self.a_form)
        self.rhs = dl.assemble(self.L_form)

    def __call__(self, m = None):
        # forcing term
        self.s_fn.vector().zero()
        self.s_fn.vector().set_local(np.random.normal(0.,1.,self.s_dim))
        ...
        # assemble (no need to reassemble A) --> if diffusion is changed, then A would
     have been assembled at that time
        self.rhs = dl.assemble(self.L_form)

        # solve
        self.u_fn.vector().zero()
        dl.solve(self.lhs, self.u_fn.vector(), self.rhs)

        # add mean
        self.u_fn.vector().axpy(1., self.mean_fn.vector())

        # vertex_dof ordered
        self.u = self.u_fn.vector().get_local()[self.V_vec2vv]
        ...
        if m is not None:
            m = self.u.copy()
            return m
        else:
            return self.u.copy()
```

Listing 1: Generating random functions with Gaussian measure.



(a) Diffusivity b_c in L_Δ (b) Random samples

Figure 2: Random samples w using Gaussian measure based on a Laplacian-like operator L_Δ defined in (11).

3. Model problems

Neural operators will be discussed in the context of solving two PDE-based problems. These are presented in the following two subsections.

3.1. Poisson problem

Consider a two dimensional domain $D_u = (0, L_1) \times (0, L_2) \subset \mathbb{R}^2$ and suppose $u : D_u \to \mathbb{R}$ denotes the temperature field. The balance of energy governs it via the differential equation:

$$\begin{aligned} -\nabla \cdot (m(x)\nabla u(x)) &= f(x)\,, \qquad && \forall x \in D_u\,, \\ u(x) &= 0\,, && \forall x \in \Gamma_{u_d}\,, \\ m(x)\nabla u(x) \cdot n(x) &= q(x)\,, && \forall x \in \Gamma_{u_n}\,, \end{aligned} \tag{16}$$

where $\Gamma_{u_d} := \{x \in \partial D_u : x_1 < L_1\}$ (i.e., all sides except the right side of the rectangular domain) and $\Gamma_{u_n} := \partial D_u - \Gamma_{u_d}$. Here, $f(x)$ is the specified heat source (J/m^3/s), $m(x)$ diffusivity (J/K/m/s), $n(x)$ unit outward normal, and $q(x)$ specified heat flux (J/m^2/s). To be consistent with the notations in Section 2.1, here $D_m = D_w = D_u$ (domains of functions m, w (w to be introduced shortly), and u), $q_w = q_m = q_u = 2$ (dimensions of the domains of functions), and $d_m = d_w = d_u = 1$ (dimensions of the pointwise values of functions). The appropriate function spaces for diffusivity and temperature are:

$$M := \left\{m : D_m \to \mathbb{R}^+ : \|m\|_{L^2} < \infty\right\} \quad \text{and} \quad U := \left\{u \in H^1(D;\mathbb{R}) : u(x) = 0\ \forall x \in \Gamma_{u_d}\right\}. \tag{17}$$

The diffusivity field m is assumed to be the unknown and uncertain parameter field. To ensure that m is a positive function (diffusivity can not be zero or negative), consider a random field

$\mathcal{Z} : \Omega \to W := L^2(D_w; \mathbb{R})$ (with $D_w = D_m = D_u$) and suppose the associated measure $\mu_{\mathcal{Z}}$ on W is Gaussian $N(0, C)$ (0 mean and C covariance operator defined in Section 2.4.1 with parameters $\mathsf{a}_c, \mathsf{b}_c, \mathsf{c}_c$). Now, given a sample $w = \mathcal{Z}(z) \in W$, sample $m = \mathcal{Q}(z)$ is computed as follows:

$$\mathcal{Q}(z) = m = \alpha_m \exp(w) + \beta_m \,, \qquad \text{where} \quad z \in \Omega \,, w = \mathcal{Z}(z) \,, \text{ and } \mu_{\mathcal{Z}} = N(0, C) \,. \tag{18}$$

Here, α_m and β_m are constants. The generation of samples w is discussed in Section 2.4, and given w, computing m using the formula above is straightforward; see `transform_gaussian_pointwise()` in Listing 3.

Finally, given $m \in M$, let $F(m) = u \in U$ be the solution of boundary value problem (BVP) (16), i.e., $F : M \to U$ is the solution/forward operator.

3.1.1. Setup details and data generation

Let $L_1 = L_2 = 1$, and consider following for f and q

$$f(x) = 1000(1 - x_2)x_2(1 - x_1)^2 \qquad \text{and} \qquad q(x) = 50 \sin(5\pi x_2) \,. \tag{19}$$

For the covariance operator C in $\mu_{\mathcal{Z}} = N(0, C)$, it is taken to be $C = L_\Delta^{-2}$, where L_Δ is defined in (11). The parameters in L_Δ and transformation of w into m take the following values:

$$\mathsf{a}_c = 0.005 \,, \quad \mathsf{b}_c = 1 \,, \quad \mathsf{c}_c = 0.2 \,, \quad \alpha_m = 1 \,, \quad \beta_m = 0 \,. \tag{20}$$

Data for neural operator training is generated using finite element discretization with triangle elements and linear interpolation for both input and output functions. The number of elements, the nodes in the mesh, and the number of degrees of freedom for u and m are 5000, $N_{nodes} = 2601$, and $p_m = p_u = 2601$, respectively. Listing 3 details the setup and solution of the Poisson model problem (16) building on the abstract class shown in Listing 2. The sampling of uncertain parameter $m = \alpha_m \exp(w) + \beta_m \sim \mu_{\mathcal{Q}}$, where $w \sim \mu_{\mathcal{Z}} = N(0, C)$, is straight-forward using the method and implementation of sampling $w \sim \mu_{\mathcal{Z}}$ discussed in Section 2.4.1 and Listing 1. In Figure 3**a**, samples of w and corresponding (m, u) pairs are shown. The notebook `Poisson.ipynb`[2] implements methods to generate and post-process data for neural operator training.

```
import numpy as np
import dolfin as dl
from fenicsUtilities import build_vector_vertex_maps

class PDEModel:

    def __init__(self, Vm, Vu, \
                 prior_sampler, seed = 0):
        ...
        # prior and transform parameters
        self.prior_sampler = prior_sampler

        # FE setup
        self.Vm = Vm
        self.Vu = Vu
        self.mesh = self.Vm.mesh()
```

[2] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/poisson/Poisson.ipynb

```
        ...
        # vertex to dof vector and dof vector to vertex maps
        self.Vm_vec2vv, self.Vm_vv2vec = build_vector_vertex_maps(self.Vm)
        self.Vu_vec2vv, self.Vu_vv2vec = build_vector_vertex_maps(self.Vu)
        ...
        # variational form
        self.u_trial = None
        self.u_test = None
        self.a_form = None
        self.L_form = None
        self.bc = None
        self.lhs = None
        self.rhs = None

    def empty_u(self):
        return np.zeros(self.u_dim)

    def empty_m(self):
        return np.zeros(self.m_dim)

    # following functions must be defined in the inherited class
    # boundaryU(x, on_boundary)
    # is_point_on_dirichlet_boundary(x)
    # assemble(self, assemble_lhs = True, assemble_rhs = True)
    # empty_u(self)
    # compute_mean(self, m)
    # solveFwd(self, u = None, m = None, transform_m = False)
    # samplePrior(self, m = None, transform_m = False)
```

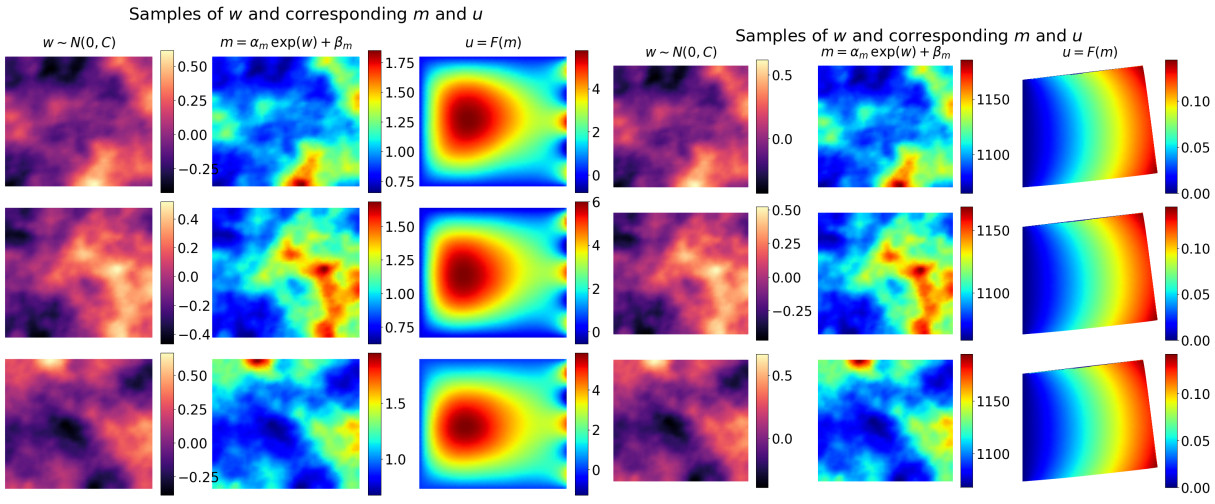
Listing 2: Abstract PDE class

```
...
import dolfin as dl
...
from pdeModel import PDEModel
...

class PoissonModel(PDEModel):
    def __init__(self, Vm, Vu, \
                 prior_sampler, \
                 logn_scale = 1., \
                 logn_translate = 0., \
                 seed = 0):
        super().__init__(Vm, Vu, prior_sampler, seed)
        # prior transform parameters
        self.logn_scale = logn_scale
        self.logn_translate = logn_translate

        # Boundary conditions
        self.f = dl.Expression("1000*(1-x[1])*x[1]*(1-x[0])*(1-x[0])", degree=2)
        self.q = dl.Expression("50*sin(5*pi*x[1])", degree=2)

        # store transformed m where input is from Gaussian prior
        self.m_mean = self.compute_mean(self.m_mean)

        # input and output functions (will be updated in solveFwd)
        self.m_fn = dl.Function(self.Vm)
```

```
27          self.m_fn = self.vertex_to_function(self.m_mean, self.m_fn, is_m = True)
28          self.u_fn = dl.Function(self.Vu)
29
30          # variational form
31          ...
32          self.a_form = self.m_fn*dl.inner(dl.nabla_grad(self.u_trial), dl.nabla_grad(
        self.u_test))*dl.dx
33          self.L_form = self.f*self.u_test*dl.dx \
34                    + self.q*self.u_test*dl.ds # boundary term
35
36          self.bc = [dl.DirichletBC(self.Vu, dl.Constant(0), self.boundaryU)]
37
38          # assemble matrix and vector
39          self.assemble()
40
41      @staticmethod
42      def boundaryU(x, on_boundary):
43          return on_boundary and x[0] < 1. - 1e-10
44
45      @staticmethod
46      def is_point_on_dirichlet_boundary(x):
47          # locate boundary nodes
48          tol = 1.e-10
49          if np.abs(x[0]) < tol \
50              or np.abs(x[1]) < tol \
51              or np.abs(x[0] - 1.) < tol \
52              or np.abs(x[1] - 1.) < tol:
53              # select all boundary nodes except the right boundary
54              if x[0] < 1. - tol:
55                  return True
56          return False
57
58      def assemble(self, assemble_lhs = True, assemble_rhs = True):
59          if assemble_lhs or self.lhs is None:
60              self.lhs = dl.assemble(self.a_form)
61          if assemble_rhs or self.rhs is None:
62              self.rhs = dl.assemble(self.L_form)
63
64          for bc in self.bc:
65              if assemble_lhs and assemble_rhs:
66                  bc.apply(self.lhs, self.rhs)
67              elif assemble_rhs:
68                  bc.apply(self.rhs)
69              elif assemble_lhs:
70                  bc.apply(self.lhs)
71
72      def transform_gaussian_pointwise(self, w, m_local = None):
73          if m_local is None:
74              self.m_transformed = self.logn_scale*np.exp(w) + self.logn_translate
75              return self.m_transformed.copy()
76          else:
77              m_local = self.logn_scale*np.exp(w) + self.logn_translate
78              return m_local
79
80      def compute_mean(self, m):
81          return self.transform_gaussian_pointwise(self.prior_sampler.mean, m)
```

```
    def solveFwd(self, u = None, m = None, transform_m = False):
        ...
        # reassamble (don't need to reassemble L)
        self.assemble(assemble_lhs = True, assemble_rhs = False)

        # solve
        dl.solve(self.lhs, self.u_fn.vector(), self.rhs)

        return self.function_to_vertex(self.u_fn, u, is_m = False)

    def samplePrior(self, m = None, transform_m = False):
        if transform_m:
            self.m_transformed = self.transform_gaussian_pointwise(self.prior_sampler
    ()[0], self.m_transformed)
        else:
            self.m_transformed = self.prior_sampler()[0]

        if m is None:
            return self.m_transformed.copy()
        else:
            m = self.m_transformed.copy()
            return m
```

Listing 3: Class for the Poisson problem. It takes random parameter field sampler and function spaces as input, defines the variational forms, and provides a method to solve for the state variable given the input parameter field.



(a) Poisson problem

(b) Linear elasticity problem

Figure 3: Some representative data samples for Poisson (a) and linear elasticity (b) problems.

3.2. Linear elasticity problem

The second problem concerns the in-plane deformation of the thin plate with the center plane given by $D_u = (0, L_1) \times (0, L_2) \subset \mathbb{R}^2$. Suppose $E(x)$ is the Young's modulus at a point $x \in D_u$ and ν Poisson ratio, $u = (u_1, u_2) : D_u \to \mathbb{R}^2$ displacement field, $e(x) = \text{sym}(\nabla u) = (\nabla u + \nabla u^T)/2$ linearized strain, $\sigma(x)$ Cauchy stress, and $b(x)$ body force per unit volume. The equation for u is

based on the balance of linear momentum and reads:

$$\begin{aligned} -\nabla \cdot \sigma(x) &= b(x)\,, && \forall x \in D_u\,, \\ \sigma(x) &= \lambda(x) e_{ii}\mathbf{I} + 2\mu(x) e\,, && \forall x \in D_u\,, \\ u(x) &= 0\,, && \forall x \in \Gamma_{u_d}\,, \\ \sigma(x)n(x) &= t(x)\,, && \forall x \in \Gamma_{u_q}\,, \\ \sigma(x)n(x) &= 0\,, && \forall x \in \partial D_u - \Gamma_{u_q} - \Gamma_{u_d}\,, \end{aligned} \tag{21}$$

where $\mathbf{I}$ is the identity second order tensor, and λ and μ are Lamé parameters and are related to E and ν as follows:

$$\lambda(x) = \frac{E(x)\nu}{(1+\nu)(1-2\nu)} \quad \text{and} \quad \mu(x) = \frac{E(x)}{2(1+\nu)}\,. \tag{22}$$

In (21), $\Gamma_{u_d} := \{x \in \partial D_u : x_1 = 0\}$ and $\Gamma_{u_q} := \{x \in \partial D_u : x_1 = L_1\}$, n unit outward normal, and t is the specified traction on the right edge of the domain. The scalar field $E \in M$ is considered to be uncertain, and the forward map $F : M \to U$ is defined such that given $E \in M$, $F(m) = u \in U$ solves the BVP (21). To be consistent with the notations, here $D_w = D_m = D_u$, $q_w = q_m = q_u = 2$, $d_w = d_m = 1$, and $d_u = 2$. Appropriate function spaces for the parameter field and solution are as follows:

$$M := \left\{m : D_m \to \mathbb{R}^+ : \|m\|_{L^2} < \infty\right\} \quad \text{and} \quad U := \left\{u \in H^1(D_u; \mathbb{R}^2) : u(x) = 0\ \forall x \in \Gamma_{u_d}\right\}\,. \tag{23}$$

As in the case of the Poisson problem, the samples of E are generated by transforming the Gaussian samples, i.e.,

$$\mathcal{Q}(z) = E = \alpha_m \exp(w) + \beta_m\,, \qquad \text{where} \quad z \in \Omega\,, w = \mathcal{Z}(z)\,, \text{ and } \mu_{\mathcal{Z}} = N(0, C)\,. \tag{24}$$

3.2.1. Setup details and data generation

Let $L_1 = L_2 = 1$, and consider following for b and t

$$b(x) = 0\hat{e}_1 + 0\hat{e}_2 \quad \text{and} \quad t(x) = 0\hat{e}_1 + 10\hat{e}_2\,. \tag{25}$$

The covariance operator in $N(0, C)$ is taken to be L_Δ^{-2}, where L_Δ is defined in (11), and parameters in L_Δ and parameters associated with the transformation of w into m are given by

$$\mathsf{a}_c = 0.005\,, \quad \mathsf{b}_c = 1\,, \quad \mathsf{c}_c = 0.2\,, \quad \alpha_m = 100\,, \quad \beta_m = 1000\,. \tag{26}$$

As in the case of the Poisson problem, finite element approximation with triangle elements and linear interpolation is utilized for both the input and output functions. The number of elements, the nodes in the mesh, and the number of degrees of freedom for u and m are 5000, $N_{nodes} = 2601$, $p_u = 5202$, and $p_m = 2601$, respectively. Listing 4 outlines crucial steps in solving the problem. The sampling of E is similar to m in problem 1. Figure 3**b** shows representative samples of w and corresponding (m, u) pair. The notebook `LinearElasticity.ipynb`[3] implements methods to generate and post-process data for neural operator training.

[3] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/linear_elasticity/LinearElasticity.ipynb

```
...
from pdeModel import PDEModel

class LinearElasticityModel(PDEModel):

    def __init__(self, Vm, Vu, \
                 prior_sampler, \
                 logn_scale = 1., \
                 logn_translate = 0., \
                 seed = 0):
        ...
        self.bc = [dl.DirichletBC(self.Vu, dl.Constant((0,0)), self.boundaryU)]

        facets = dl.MeshFunction("size_t", self.mesh, self.mesh.topology().dim()-1)
        dl.AutoSubDomain(self.boundaryQ).mark(facets, 1)
        self.ds = dl.Measure("ds", domain=self.mesh, subdomain_data=facets)
        ...
        self.nu = 0.25
        self.lam_fact = dl.Constant(self.nu / (1+self.nu)*(1-2*self.nu))
        self.mu_fact = dl.Constant(1/(2*(1+self.nu)))

        self.spatial_dim = self.u_fn.geometric_dimension()
        self.a_form = self.m_fn*dl.inner(self.lam_fact*dl.tr(dl.grad(self.u_trial))*dl
    .Identity(self.spatial_dim) \
                                         + 2*self.mu_fact * dl.sym(dl.grad(self.u_trial
    )), \
                                    dl.sym(dl.grad(self.u_test)))*dl.dx

        self.L_form = dl.inner(self.b, self.u_test)*dl.dx + dl.inner(self.t, self.
    u_test)*self.ds
        ...

    ...
    def solveFwd(self, u = None, m = None, transform_m = False):
        # similar to the solveFwd() in Listing 3

    def samplePrior(self, m = None, transform_m = False):
        # similar to the samplePriors() in Listing 3
```

Listing 4: Class for the linear elastic problem. Here, only the initialization part is shown, as all other functions are similar to the implementation of the Poisson problem in Listing 3.

4. Neural networks as surrogate of the forward problem

Over the years, several neural operator architectures have been introduced that leverage neural networks to build fast and efficient approximations of F. This section considers a few key neural operators, extracts core ideas and implementation details, and creates a strong working knowledge, as done in the subsequent sub-sections.

The following three subsections will introduce and go into implementation details of DeepONet, PCANet, and FNO.

4.1. Deep Operator Network (DeepONet)

DeepONet was first introduced in Lu et al. (2019), and over the years, various extensions of the general framework and applications have been realized, e.g., Lu et al. (2021b); Goswami et al.

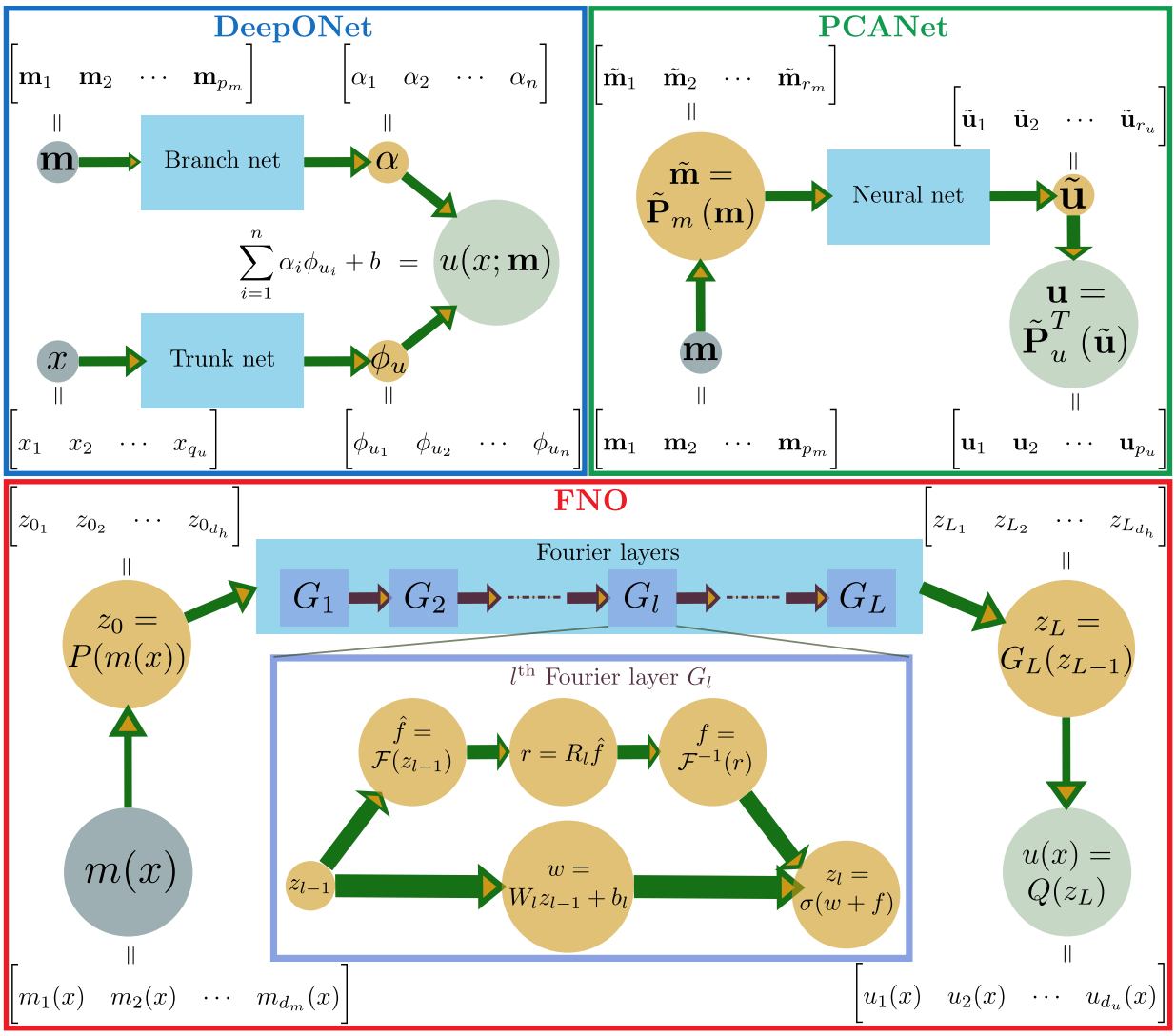


Figure 4: Schematics of three neural operators, DeepONet, PCANet, and FNO. Grey and light green circles represent the input and output of neural operators. The blue box includes a parameterized neural network-based map. In this work, the blue boxes for DeepONet and PCANet employ multilayer perceptron fully-connected neural networks. In the case of FNO, trainable parameters (namely, R_l, W_l, b_l) appear within each Fourier layer.

(2023). Consider some m, and suppose $\{\phi_u\}_{i=1}^{p_u} \subset U$ is finite collection of basis functions, then, at $x \in D_u$,

$$F(m)(x) = u(x) \approx \sum \underbrace{\langle F(m), \phi_{u_i} \rangle}_{=:\alpha_i(m)} \phi_{u_i}(x)\,, \tag{27}$$

where, coefficients $\alpha_i = \alpha_i(m)$ depend on m. DeepONet's underlying idea is to learn the above finite-dimensional representation of the output of operator F. That is, identify the linear bases or more precisely learn the values of basis functions $\phi_{u_i}(x)$ at coordinates $x \in D_u$, and the coefficients associated with the bases dependent on the input m, $\{\alpha_i(m)\}$ so that $\sum_i \alpha_i(m)\phi_{u_i}(x)$ is approximately equal to $F(m)(x) = u(x)$. Towards this, DeepONet considers two neural networks, so-called

branch and trunk networks; see Figure 4. The branch network takes discretization of the input function, denoted by $\mathbf{m} \in \mathbb{R}^{p_m}$, and the neural network produces N_{br} number of coefficients, which are used as $\{\alpha_i\}_{i=1}^{N_{br}}$. On the other hand, the trunk network takes as input the spatial coordinate, x, and outputs N_{tr} numbers, which play the role of $\{\phi_{u_i}(x)\}_{i=1}^{N_{tr}}$ in (27). Here, m and u are assumed to be scalar fields, and $N_{tr} = N_{br}$. Finally, the loss function is defined in terms of the norm of the difference between the ground truth and approximation by the joint output from the branch and trunk networks, $\sum_{i=1}^{N_{br}} \alpha_i(m)\phi_{u_i}(x)$. In summary, the operator approximation in DeepONet takes the form, for $\mathbf{m} \in \mathbb{R}^{p_m}$ and $x \in D_u$,

$$F_{NOp}(m)(x) = \sum_{i=1}^{N_{br}} \alpha_i(m;\theta_{br})\phi_{u_i}(x;\theta_{tr}) + b\,, \tag{28}$$

where

(i) $\alpha_i = \alpha_i(m;\theta_{br})$, $1 \le i \le N_{br}$, are outputs of the branch network with θ_{br} neural network trainable parameters,

(ii) $\phi_{u_i}(x;\theta_{tr})$, $1 \le i \le N_{tr} = N_{br}$, are outputs of the trunk network with θ_{tr} neural network trainable parameters, and

(iii) $b \in \mathbb{R}^{d_o}$ is a bias. Here, d_o is the dimension of the pointwise value of the output function u, i.e., d_o is such that $u(\xi) \in \mathbb{R}^{d_o}$.

As mentioned in several works Lu et al. (2021b); Goswami et al. (2023), it is interesting to note how the learning of the coefficients and learning pointwise values of bases are separated via branch and trunk networks. Another crucial property of DeepONet is that an approximation of $u = F(m)$ at any arbitrary point $x \in \Omega_u$ can be computed.

4.1.1. Implementation of DeepONet

To simplify the presentation, the input and output functions, $m \in M$ and $u \in U$, respectively, are assumed to be scalar-valued, and these functions are appropriately discretized, e.g., using finite element approximation. Extending the cases when one or both of these functions are vector-valued is trivial; see Remark 1.

Let $\mathbf{m} \in \mathbb{R}^{p_m}$ and $\mathbf{u} = \mathbf{F}(\mathbf{m}) \in \mathbb{R}^{p_u}$ are input and output (discretized) functions, $\mathbf{F}$ is the discretization of the operator F of interest. The following constitutes data for DeepONet:

1. Consider the $N_m \times p_m$-matrix, X_{br}, where N_m is the number of input function samples and I^{th} row of X_{br} is the sample $\mathbf{m}^I \in \mathbb{R}^{p_m}$. Each row of X_{br} will be the input to the branch network.

2. Select N_x number of locations, $\{x^I\}_{I=1}^{N_x}$, where $x^I \in D_u \subseteq \mathbb{R}^{q_u}$ and q_u is the dimension of the domain. Each location x^I will be the input to the trunk neural network so that the output of DeepONet is the prediction of the value of the target function at x^I. In the present implementation, the typical input coordinate x^I corresponds to the I^{th} discretization grid or nodes of a mesh so that the value of the output function $\mathbf{u}$ at x^I is simply the element $\mathbf{u}_I$ in the vector $\mathbf{u}$ corresponding to that grid/node. The matrix X_{tr} of size $N_x \times q_u$ is the data for the trunk network, and each row of X_{tr} is the input to the trunk network.

3. $N_m \times N_x$-matrix Y, such that an element Y_{IJ} is the value of output data $\mathbf{u}^I(x^J) = \mathbf{F}(\mathbf{m}^I)(x^J)$. I.e., Y_{IJ} is the value of the output data function $\mathbf{u}^I$ corresponding to the branch network input data function $\mathbf{m}^I$ (I^{th} row of X_{br}) at trunk network input location x^J (J^{th} row of X_{tr}).

Next, the goal is to find the combined training parameters $\theta = \{\theta_{br}, \theta_{tr}, b\} \in \mathbb{R}^{N_\theta}$ such that the error is minimized:

$$\theta^* = \underset{\theta=\{\theta_{br},\theta_{tr},b\}\in\mathbb{R}^{N_\theta}}{\arg\min} \frac{1}{N_m N_x} \sum_{I=1}^{N_m} \sum_{J=1}^{N_x} \left| Y_{IJ} - \underbrace{\left(\sum_{k=1}^{N_{br}} \alpha_k(\mathbf{m}^I;\theta_{br}) \phi_{u_k}(x^J;\theta_{tr}) + b \right)}_{\text{DeepONet output}} \right|^2 . \tag{29}$$

Remark 1. *The DeepONet framework described so far can be easily extended to the case when the target function of the operator is vector-valued. Suppose $u(x) = (u_1(x), u_2(x), ..., u_{d_o}(x)) \in \mathbb{R}^{d_o}$, u_j being the components. The approach used in this work is as follows. Suppose N_{tr} is the number of outputs from the trunk network. Then the branch network is designed to produce $N_{br} = d_o N_{tr}$ outputs, and the prediction, $u_{pred} = (u_{pred,1}, u_{pred,2}, ..., u_{pred,d_o})$, of u, for x^J row of X_{tr} and $\mathbf{m}^I$ row of X_{br}, is given by*

$$\begin{aligned} u_{pred,1} &= \sum_{k=1}^{N_{tr}} \alpha_k(\mathbf{m}^I;\theta^b) \phi_{u_k}(x^J;\theta^t) + b_1 \\ u_{pred,2} &= \sum_{k=1}^{N_{tr}} \alpha_{k+N_{tr}}(\mathbf{m}^I;\theta^b) \phi_{u_k}(x^J;\theta^t) + b_2 \\ &\cdots , \\ u_{pred,d_o} &= \sum_{k=1}^{N_{tr}} \alpha_{k+(d_o-1)N_{tr}}(\mathbf{m}^I;\theta^b) \phi_{u_k}(x^J;\theta^t) + b_{d_o} , \end{aligned} \tag{30}$$

i.e., the first N_{tr} outputs of the branch are used to predict u_1 component, the next N_{tr} branch outputs are used to predict u_2 component, and so on. In the above, the same trunk outputs are used for all components of the target functions.

Multi-layer perception (MLP) is used as branch and trunk networks. The implementation used in this work is based on the `DeepONet` Github repository[4] with some minor modifications. Listing 5 shows the implementation of MLP and Listing 6 implements the DeepONet framework.

```
1
2 import torch
3 import torch.nn as nn
4
5 class MLP(nn.Module):
6
7     def __init__(self, input_size, hidden_size, num_classes, depth, act):
8         super(MLP, self).__init__()
9         self.layers = nn.ModuleList()
```

[4] https://github.com/GideonIlung/DeepONet

```
        self.act = act

        # input layer
        self.layers.append(nn.Linear(input_size, hidden_size))

        # hidden layers
        for _ in range(depth - 2):
            self.layers.append(nn.Linear(hidden_size, hidden_size))

        # output layer
        self.layers.append(nn.Linear(hidden_size, num_classes))

    def forward(self, x, final_act=False):
        for i in range(len(self.layers) - 1):
            x = self.act(self.layers[i](x))

        # last layer
        x = self.layers[-1](x)
        if final_act == False:
            return x
        else:
            return torch.relu(x)
```

Listing 5: Multilayer Perceptron (MLP) implementation following DeepONet Github repository[4]

```
...
import torch
import torch.nn as nn
from torch_mlp import MLP
...

class DeepONet(nn.Module):

    def __init__(self, ...):

        super(DeepONet, self).__init__()
        ...
        # branch network
        self.branch_net = MLP(input_size=num_inp_fn_points, \
                              hidden_size=num_neurons, \
                              num_classes=num_br_outputs, \
                              depth=num_layers, \
                              act=act)
        self.branch_net.float()

        # trunk network
        self.trunk_net = MLP(input_size=out_coordinate_dimension, \
                             hidden_size=num_neurons, \
                             num_classes=num_tr_outputs, \
                             depth=num_layers, \
                             act=act)
        self.trunk_net.float()

        # bias added to the product of branch and trunk networks
        self.bias = [nn.Parameter(torch.ones((1,)),requires_grad=True) for i in range(
    num_Y_components)]
        ...
```

```
        # record losses
        self.train_loss_log = []
        self.test_loss_log = []

    def convert_np_to_tensor(self,array):
        if isinstance(array, np.ndarray):
            tensor = torch.from_numpy(array)
            return tensor.to(torch.float32)
        else:
            return array

    def forward(self, X, X_trunk):

        X = self.convert_np_to_tensor(X)
        X_trunk = self.convert_np_to_tensor(X_trunk)

        branch_out = self.branch_net.forward(X)
        trunk_out = self.trunk_net.forward(X_trunk,final_act=True)

        if self.num_Y_components == 1:
            output = branch_out @ trunk_out.t() + self.bias[0]
        else:
            # when d_o > 1, split the branch output and compute the product
            output = []
            for i in range(self.num_Y_components):
                output.append(branch_out[:,i*self.num_tr_outputs:(i+1)*self.
    num_tr_outputs] @ trunk_out.t() + self.bias[i])

            # stack and reshape
            output = torch.stack(output, dim=-1)
            output = output.reshape(-1, X_trunk.shape[0] * self.num_Y_components)

        return output

    def train(self, train_data, test_data, batch_size=32, epochs = 1000, lr=0.001,
    ...):
        ...
        # loss and optimizer setup
        criterion = nn.MSELoss()
        optimizer = optim.Adam(self.parameters(), lr=lr, weight_decay=1e-4)
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
        ...
        # training and testing loop
        start_time = time.perf_counter()

        self.trainable_params = sum(p.numel() for p in self.parameters() if p.
    requires_grad)
        ...
        for epoch in range(1, epochs+1):
            ...
            # training loop
            for X_train, _, Y_train in dataloader:
                ...
                # clear gradients
                optimizer.zero_grad()

```

```
                # forward pass through model
                Y_train_pred = self.forward(X_train, X_trunk)

                # compute and save loss
                loss = criterion(Y_train_pred, Y_train)
                train_losses.append(loss.item())

                # backward pass
                loss.backward()

                # update parameters
                optimizer.step()

            # update learning rate
            scheduler.step()

            # testing loop
            with torch.no_grad():
                for X_test, _, Y_test in test_dataloader:

                    # forward pass through model
                    Y_test_pred = self.forward(X_test, X_trunk)

                    # compute and save test loss
                    test_loss = criterion(Y_test_pred, Y_test)
                    test_losses.append(test_loss.item())

            # log losses
            self.train_loss_log[epoch-1, 0] = np.mean(train_losses)
            self.test_loss_log[epoch-1, 0] = np.mean(test_losses)
            ...
    def predict(self, X, X_trunk):
        with torch.no_grad():
            return self.forward(X, X_trunk)
```

Listing 6: DeepONet implementation

4.1.2. Architecture and preliminary results

A four-layer fully connected network is considered for both the branch and trunk networks. Neural network and optimization-related parameters are tabulated in Table 2. For the linear elasticity problem, $N_{br} = 200$ and $N_{tr} = 100$, and the formula for the prediction of a vector-valued is based on Remark 1.

For the Poisson problem, Figure 5 shows the typical prediction error using the DeepONet. In Figure 6, the results for the linear elasticity problem are shown. These figures also compare the accuracy of PCANet and FNO, which will be discussed in the following two subsections. The notebooks `DeepONet-Poisson`[5] and `DeepONet-Linear Elasticity`[6] show the steps involved in instantiating DeepONet and training and testing.

[5] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/poisson/DeepONet/training_and_testing.ipynb

[6] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/linear_elasticity/DeepONet/training_and_testing.ipynb

Parameter	**Value**	**Description**
Layers	4	Number of layers in DeepONet (branch and trunk each) and PCANet
Hidden layer width	128	Number of neurons in hidden layers
N_{train}, N_{test}	3500, 1000	Number of training and testing pairs $(\mathbf{m}^I, \mathbf{u}^I)$
N_m	3500	N_m is same as N_{train} (notation used for DeepONet);
N_x	2601	Number of coordinates for evaluation of u (DeepONet)
p_m, p_u	2601, $2601d_o$	Dimensions of discretized functions; $d_o = 1$ for Poisson and $d_o = 2$ for linear elasticity
r_m, r_u	100, 100	Reduced dimensions for the Poisson problem (PCANet)
r_m, r_u	50, 50	Reduced dimensions for the elasticity problem (PCANet)
N_{br}, N_{tr}	$100d_o$, 100	Number of branch and trunk outputs in DeepONet
Batch size	20	Neural networks are trained on "batch size" samples
Batch size (FNO)	20	Batch size specifically for FNO
Epochs	1000	Number of optimization steps
Epochs (FNO)	500	Number of optimization steps specifically for FNO
Learning rate	0.001	A parameter controlling the parameter update in gradient-based methods
Activation	ReLU	Activation function
d_h	20	Dimension of FNO layer outputs
L	3	Number of FNO layers
k_{max}	8	Number of Fourier modes to keep in FNO
n_1, n_2	51, 51	Number of grid points in grid mesh of $D_u = (0, 1)^2$ for FNO

Table 2: Summary of various parameters used in neural operator training and testing calculations.

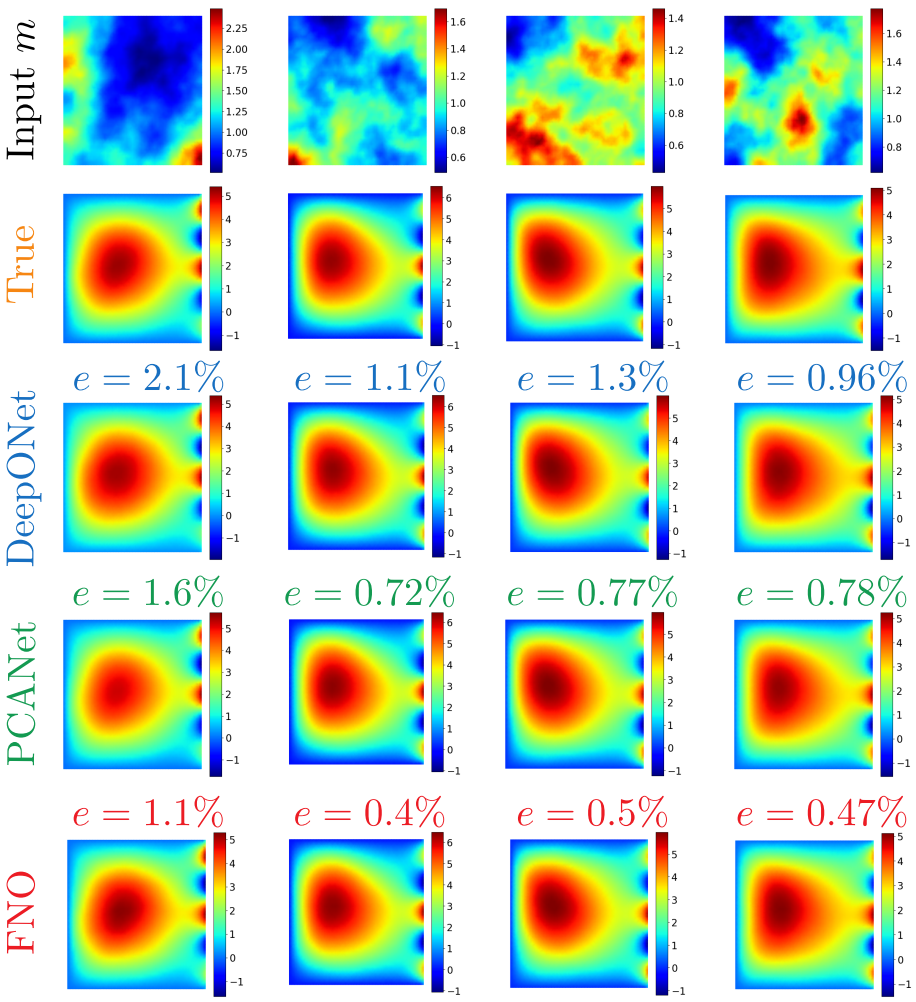


Figure 5: Comparing DeepONet, PCANet, and FNO predictions with the finite element solution for four random samples of diffusivity in the Poisson problem. Here, e is the relative percentage l^2 error corresponding to the sample.

4.2. Principal Component Analysis-based Neural Operator (PCANet)

The second neural operator of interest is the PCANet introduced in Bhattacharya et al. (2021), which utilizes PCA to reduce the dimensions of input and output functions and poses an operator learning problem in the reduced dimensional spaces, thereby making the learning efficient. Consider $\{(\mathbf{m}^I, \mathbf{u}^I = \mathbf{F}(\mathbf{m}^I))\}_{I=1}^N$ set of paired data, where $\mathbf{m}^I \in \mathbb{R}^{p_m}$ and $\mathbf{u}_i \in \mathbb{R}^{p_u}$. Let r_m and r_u denote the reduced dimensions for input and output functions, respectively, and $\tilde{\mathbf{P}}_m \in \mathbb{R}^{r_m \times p_m}$ and $\tilde{\mathbf{P}}_u \in \mathbb{R}^{r_u \times p_u}$ are the projectors based on SVD for dimension reduction. Learning the approximation of the target map $\mathbf{F} : \mathbb{R}^{p_m} \to \mathbb{R}^{p_u}$ becomes challenging if p_m and p_u are large. Moreover, the complexity of neural networks working with high-dimensional input and output is large. PCANet alleviates these challenges by introducing a low-dimensional approximation of $\mathbf{F}$. Specifically, in PCANet, the dimensions of input and output functions are reduced using SVD, and the neural network between reduced dimensional inputs and outputs is introduced to approximate the mapping. To make this precise, consider a parameterized neural network map $\tilde{\mathbf{F}}_\theta : \mathbb{R}^{r_m} \to \mathbb{R}^{r_u}$ to construct the

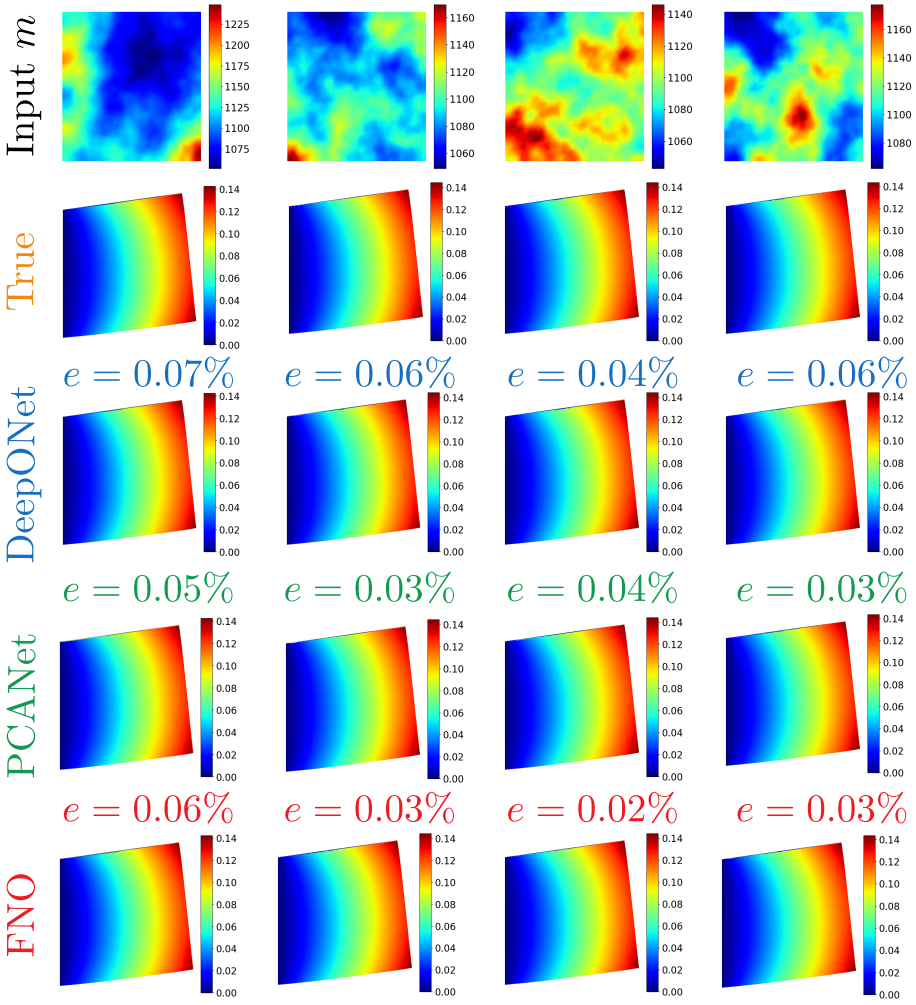


Figure 6: Comparing DeepONet, PCANet, and FNO predictions with the finite element solution for four random samples of Young's modulus in the linear elasticity problem. Here, e is the relative percentage l^2 error corresponding to the sample.

approximation $\mathbf{F}_{NOp}$ of $\mathbf{F}$ as follows, for given $\mathbf{m} \in \mathbb{R}^{p_m}$,

$$\mathbb{R}^{p_u} \ni \mathbf{u} = \mathbf{F}(\mathbf{m}) \qquad \approx \qquad \mathbf{F}_{NOp}(\mathbf{m}) = \underbrace{\tilde{\mathbf{P}}_u^T \left(\underbrace{\tilde{\mathbf{F}}_\theta \left(\underbrace{\tilde{\mathbf{P}}_m(\mathbf{m})}_{=:\tilde{\mathbf{m}} \text{ (project } \mathbf{m})} \right)}_{=:\tilde{\mathbf{u}} \text{ (low-dim map)}} \right)}_{\text{lift } \tilde{\mathbf{u}}} \in \mathbb{R}^{p_u}. \tag{31}$$

The parameters θ are determined via the optimization problem:

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{I=1}^{N} \left\| \mathbf{F}(\mathbf{m}^I) - \tilde{\mathbf{P}}_u^T \left(\tilde{\mathbf{F}}_\theta \left(\tilde{\mathbf{P}}_m(\mathbf{m}^I) \right) \right) \right\|^2, \tag{32}$$

where $\|\cdot\|$ denotes the l^2-norm, and $\Theta \subset \mathbb{R}^{p_\theta}$ appropriate space of neural networks parameters. Figure 4 presents the schematics of PCANet and the projection steps.

4.2.1. Implementation of PCANet

Let X and Y are two $N \times p_m$ and $N \times p_u$ matrices, respectively, such that rows of X and Y are input and output function samples, $\mathbf{m}^I \in \mathbb{R}^{p_m}$ and $\mathbf{u}^I \in \mathbb{R}^{p_u}$, where $1 \leq I \leq N$. The data for neural network-based map $\tilde{\mathbf{F}}$ is constructed using the following steps, which include the preprocessing to construct SVD-based projectors:

1. *Centering and scaling* X and Y data by subtracting the mean and dividing (elementwise) the sample standard deviation. E.g., if $\bar{X}$ and σ_X are $1 \times p_m$ mean and standard deviation matrices, then $\hat{X} = (X - \bar{X})/(\sigma_X + \text{tol})$, where the division is element-wise and tol small number. Similarly, $\hat{Y}$ is obtained.

2. *SVD projectors* for input and output data are determined following the procedure in Section 2.3.1. E.g., take $A = \hat{X}^T$ and compute a $r_m \times p_m$ matrix, $\tilde{\mathbf{P}}_m$, where r_m is the specified reduced dimension. Similarly, $\tilde{\mathbf{P}}_u$ can be obtained for the output data.

3. *Projected data for neural network* are computed by taking the rows of $\hat{X}$ and projecting them into reduced space. To be specific, let I^{th} row of matrix $\hat{X}$ is $(\hat{\mathbf{m}}^I)^T \in \mathbb{R}^{1\times p_m}$, where $\hat{\mathbf{m}}^I$ is the centered and scaled I^{th} input sample. Its projection is $\tilde{\mathbf{m}}^I = \tilde{\mathbf{P}}_m(\hat{\mathbf{m}}^I) \in \mathbb{R}^{r_m}$. Using the projection, a new matrix $\tilde{X}$ of size $N \times r_m$ is formed, where $(\tilde{\mathbf{m}}^I)^T \in \mathbb{R}^{1\times r_m}$ gives the I^{th} row of the matrix. Similarly, $\tilde{Y}$ of size $N \times r_u$ is obtained by transforming each row of $\hat{Y}$ by applying $\tilde{\mathbf{P}}_u$.

4. For the reduced dimensional map $\mathbb{R}^{r_u} \ni \tilde{\mathbf{u}} = \tilde{\mathbf{F}}_\theta(\tilde{\mathbf{m}})$, with $\tilde{\mathbf{m}} \in \mathbb{R}^{r_m}$, $\tilde{X}$ and $\tilde{Y}$ constitutes as input and output data, respectively.

Given data $\tilde{X}$ and $\tilde{Y}$ and a neural network-based map $\tilde{\mathbf{F}}_\theta$, the optimization problem to determine θ is given by

$$\theta^* = \arg\min_{\theta\in\Theta} \frac{1}{N} \sum_{I=1}^{N} \left|\left| \tilde{Y}_I - \tilde{\mathbf{F}}_\theta(\tilde{X}_I) \right|\right|^2 , \tag{33}$$

where $(\cdot)_I$ denotes the I^{th} row of matrix.

The core steps in implementing PCANet, i.e., $\tilde{\mathbf{F}}_\theta$, are shown in Listing 7.

```
...
import torch
import torch.nn as nn
from torch_mlp import MLP
...

class PCANet(nn.Module):

    def __init__(self, ...):

        super(PCANet, self).__init__()
        ...
        # network
        self.net = MLP(input_size=num_inp_red_dim, \
                               hidden_size=num_neurons, \
```

```
                            num_classes=num_out_red_dim, \
                            depth=num_layers, \
                            act=act)
        self.net.float()

        # record losses
        self.train_loss_log = []
        self.test_loss_log = []

    def convert_np_to_tensor(self,array):
        if isinstance(array, np.ndarray):
            tensor = torch.from_numpy(array)
            return tensor.to(torch.float32)
        else:
            return array

    def forward(self, X):
        X = self.convert_np_to_tensor(X)
        return self.net.forward(X)

    def train(self, train_data, test_data, batch_size=32, epochs = 1000, lr=0.001,
    ...):
        ...
        # loss and optimizer setup
        criterion = nn.MSELoss()
        optimizer = optim.Adam(self.parameters(), lr=lr, weight_decay=1e-4)
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
        ...
        # training and testing loop
        start_time = time.perf_counter()

        self.trainable_params = sum(p.numel() for p in self.parameters() if p.
    requires_grad)
        ...
        for epoch in range(1, epochs+1):
            ...
            # training loop
            for X_train, Y_train in dataloader:

                # clear gradients
                optimizer.zero_grad()

                # forward pass through model
                Y_train_pred = self.forward(X_train)

                # compute and save loss
                loss = criterion(Y_train_pred, Y_train)
                train_losses.append(loss.item())

                # backward pass
                loss.backward()

                # update parameters
                optimizer.step()

            # update learning rate
```

```
            scheduler.step()

            # testing loop
            with torch.no_grad():
                for X_test, Y_test in test_dataloader:

                    # forward pass through model
                    Y_test_pred = self.forward(X_test)

                    # compute and save test loss
                    test_loss = criterion(Y_test_pred, Y_test)
                    test_losses.append(test_loss.item())

            # log losses
            self.train_loss_log[epoch-1, 0] = np.mean(train_losses)
            self.test_loss_log[epoch-1, 0] = np.mean(test_losses)
            ...
    def predict(self, X):
        with torch.no_grad():
            return self.forward(X)
```

Listing 7: PCANet implementation. The interface is similar to the DeepONet, but with one key difference: PCANet does not require spatial coordinates as input.

4.2.2. Architecture and preliminary results

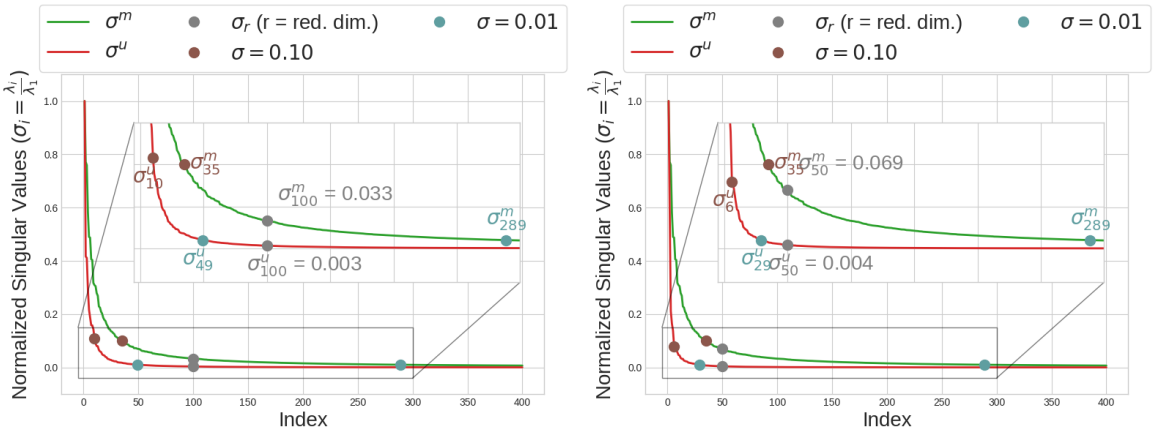
A fully connected neural network with four layers is considered for testing the PCANet. Other parameters, including the reduced dimension, are listed in Table 2. First, the singular values of the input and output data are analyzed. This helps decide the dimension of reduced space so that accuracy is not significantly compromised. In Figure 7, the plots of singular values for the input and output data show a rapid spectrum decay; the figure also shows the singular values at reduced dimensions for input and output data. The prediction and the performance of PCANet for the Poisson and linear elasticity problems are shown in Figure 5 and Figure 6, respectively. The notebooks `PCANet-Poisson`[7] and `PCANet-Linear Elasticity`[8] apply the PCANet to the two problems.

4.3. Fourier Neural Operator (FNO)

Fourier neural operator considers the composition of layers, with a typical layer involving affine transformation and an integral kernel operator followed by nonlinear activation; see Li et al. (2020a, 2021); Kovachki et al. (2021). These affine and integral kernel operations are parameterized. While there are multiple choices of integral kernel operator Kovachki et al. (2021), this work uses Fourier transform. Consider the case of $D_m = D_u = D \subset \mathbb{R}^q$, where $q_m = q_u = q$, and the neural operator

[7] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/poisson/PCANet/training_and_testing.ipynb

[8] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/linear_elasticity/PCANet/training_and_testing.ipynb



(a) Poisson equation

(b) Linear elasticity

Figure 7: Singular values of input and output data matrices $\hat{X}$ and $\hat{Y}$ (centered and scaled data). Green and red curves represent the normalized singular values for input and output data, respectively. For both curves, the annotation is displayed close to the 10% (brown dots) and 1% (blue dots), i.e., 0.1 and 0.01 fractions of the largest singular value. The grey dots correspond to the fixed reduced dimensions in PCANet. The rapid decay of singular values shows that both problems are inherently low-dimensional.

approximation F_{NOp} such that

$$u(x) = F(m)(x) \quad \approx \quad F_{NOp}(m)(x) := Q \left(\underbrace{G_L \left(G_{L-1} \cdots G_1 \left(\underbrace{P\left(m(x)\right)}_{=z_0 \in \mathbb{R}^{d_h}} \right) \right)}_{=z_L \in \mathbb{R}^{d_h}} \right), \tag{34}$$

or, concisely,

$$F_{\mathrm{NOp}} = Q \circ G_L \circ G_{L-1} \circ \cdots \circ G_1 \circ P\,. \tag{35}$$

The map F_{NOp} involves the following operations (see Li et al. (2020a); Kovachki et al. (2023)):

- Lifting $m(x) \in \mathbb{R}^{d_m}$ to $\mathbb{R}^{d_h}$, where d_h is the dimension of outputs from hidden layers via the trainable matrix P of size $d_h \times d_m$.
- Projecting the final hidden layer output $z_L \in \mathbb{R}^{d_h}$ onto $\mathbb{R}^{d_u}$ (space of $u(x)$) via the $d_u \times d_h$ matrix Q. Here, Q is also trainable.
- Application of operators G_l, $1 \le l \le L$, where G_l is defined via:

$$\mathbb{R}^{d_h} \ni z_l = G_l(z_{l-1}) = \sigma_l \left(\underbrace{W_l z_{l-1} + b_l}_{\text{linear/local operation}} + \underbrace{K_l(z_{l-1})}_{\text{nonlocal operation}} \right). \tag{36}$$

  Here, z_{l-1} is the output of the preceding layer, σ_l activation function, W_l weight matrix, b_l a bias vector, and $K_l(\cdot)$ a integral kernel operator.

Up until now, the map F_{NOp} in Equation (34) with the above definitions of P, Q, G_l, $l = 1, ..., L$, is abstract due to the generality of K_l. The linear operation in (36) captures the local effects on the reconstructed function, while the K_l is designed to capture the non-local effects (interaction with other degrees of freedom) via the integral kernel operation. There are various choices for K_l, as discussed in Kovachki et al. (2023). For example:

(1) Low-rank Neural Operator (LNO) in which K_l takes the form $K_l(z) = \sum_{j=1}^{r} \langle \psi^{(i)}, z \rangle \phi^{(i)}(x)$, where $\phi^{(i)}$ and $\psi^{(i)}$ are some parameterized functions;

(2) Graph Neural Operator (GNO) in which $K_l = \frac{1}{|B(x,r)|} \sum_{y_i \in B(x,r)} k(x, y_i) z(y_i)$, $k(\cdot, \cdot)$ kernel function; and

(3) Fourier Neural Operator (FNO) maps z to Fourier space, followed by mapping the weighted Fourier modes back to the real space. Since FNO is the main focus of this article, it is discussed in more detail next. Readers are referred to Li et al. (2020a, 2021); Kovachki et al. (2021) for further discussion of FNO and associated ideas.

In FNO, $K_l(z)$, for $z \in \mathbb{R}^{d_h}$, takes the form:

$$K_l(z) = \mathcal{F}^{-1} \left(R_l \mathcal{F}(z) \right) , \tag{37}$$

where

- $\mathcal{F}(z)$ is the Fourier transform applied to each components of z. Only the first k_{max} modes are retained, so the output of $\mathcal{F}(z)$ is in $\mathbb{R}^{d \times k_{max}}$.
- R_l applies the weighting to different Fourier modes. R_l is a complex-valued $d_h \times d_h \times k_{max}$ trainable matrix and $R_l \mathcal{F}(z) \in \mathbb{R}^{d_h \times k_{max}}$.
- $\mathcal{F}^{-1}(\cdot) \in \mathbb{R}^{d_h}$ is the inverse Fourier transform.

Figure 4 displays the FNO framework based on Fourier transforms. Summarizing, the trainable parameters in (34) with the specific forms of G_l and K_l are:

$$\theta := \left\{ P \in \mathbb{R}^{d_h \times d_i}, Q \in \mathbb{R}^{d_o \times d_h}, \left\{ W_l \in \mathbb{R}^{d_h \times d_h}, b_l \in \mathbb{R}^{d_h}, R_l \in \mathbb{C}^{d_h \times d_h \times k_{max}}, 1 \le l \le L \right\} \right\} . \tag{38}$$

Finally, in the discrete setting, one evaluates the FNO output at all grid points to have

$$\mathbb{R}^{p_u} \ni \mathbf{u} = \mathbf{F}(\mathbf{m}) \quad \approx \quad \mathbf{F}_{NOp}(\mathbf{m}) := Q \left(G_L \left(G_{L-1} \cdots G_1 \left(P(\mathbf{m}) \right) \right) \right) , \qquad \text{for } \mathbf{m} \in \mathbb{R}^{p_m} . \tag{39}$$

The optimization problem to determine θ is given by

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{N_m} \sum_{I=1}^{N_m} \left|\left| \mathbf{u}^I - \mathbf{F}_{NOp}(\mathbf{m}^I) \right|\right|^2 . \tag{40}$$

4.3.1. Implementation of FNO

The implementation of FNO requires function values at grid locations, and thus, preprocessing is required to obtain the function values at grid points from the finite element field. Suppose $D_m = D_u = D = (0,1)^2$ and consider the grid division of closure(D) consisting of n_1 and n_2 number of points in x_1 and x_2 directions, respectively. Linear interpolation is used to compute function values at grid points. The following describes the data:

1. Let X be $N \times n_1 \times n_2 \times 3$ matrix, where the outer dimension corresponds to the number of samples. The element of X at I^{th} outer index is a $n_1 \times n_2 \times 3$ matrix containing the interpolated values of a function m^I at all grid (x^1, x^2) points and the coordinates x^1 and x^2 of all grid points; thus, the inner dimension is three. Next, the function values are centered and scaled using the mean and standard deviation computed from the samples $1 \leq I \leq N$.

2. Let Y be $N \times n_1 \times n_2 \times d_o$ matrix such that the element of Y at I^{th} outer index is a $n_1 \times n_2 \times d_o$ matrix containing the interpolated values of d_o-valued function $\mathbf{u}^I$ at all grid points. Note that the element of X and Y at I^{th} outer index corresponds to $\mathbf{m}^I$ and $\mathbf{u}^I = \mathbf{F}(\mathbf{m}^I)$ on a grid, thereby establishing the correspondence between input and output. Function values at grid points are also centered and scaled using the sample mean and standard deviation.

Given a sample from X, denoted $\mathrm{x} = (m^I(x^1, x^2), x^1, x^2) \in \mathbb{R}^3$, it is first lifted into $\mathbb{R}^{d_h}$, where $d_h > 3$ is a dimension of the input and output spaces of FNO layers, to get $z_0 = P(\mathrm{x}) = W_P\,\mathrm{x} + b_P$, where $W_P \in \mathbb{R}^{d_h \times 3}$ and $b_P \in \mathbb{R}^{d_h}$. Next, z_0 goes through L FNO layers such that given z_{l-1} an output of $(l-1)^{\text{th}}$ layer, $z_l = G_l(z_{l-1})$. The Listing 8 presents the implementation of an FNO layer G_l based on the `Operator-Learning` repository[9] De Hoop et al. (2022). The l^{th} layer involves a linear transformation $z_l^1 = W_l z_{l-1} + b_l$ and Fourier-based transformation $z_L^2 = \mathcal{F}^{-1}(R_l \cdot \mathcal{F}(z_l))$, where R_l is a complex matrix of size $d_h \times d_h \times k_{max} \times k_{max}$, k_{max} denoting the number of Fourier modes that are retained after the Fourier transform. Here, W_l and b_l are weight and bias parameters. The output of the final layer $z_L = G_L(z_{L-1}) \in \mathbb{R}^{d_h}$ is projected to $u_{NOp}(x^1, x^2) = Q(z_L) \in \mathbb{R}^{d_o}$; the projection operator Q introduces $W_Q \in \mathbb{R}^{d_o \times d_h}$ and $b_Q \in \mathbb{R}^{d_o}$ parameters. In Listing 9, lift, FNO layer applications, and projection are combined to create an FNO model; see the `forward` method. The trainable parameters are:

$$\begin{aligned}\theta := \Big\{ &(W_P, b_P) \in \mathbb{R}^{d_h \times 3} \times \mathbb{R}^{d_h}, \quad (W_Q, b_Q) \in \mathbb{R}^{d_o \times d_h} \times \mathbb{R}^{d_o}, \\ &\Big\{ W_l \in \mathbb{R}^{d_h \times d_h}, b_l \in \mathbb{R}^{d_h}, R_l \in \mathbb{C}^{d_h \times d_h \times k_{max} \times k_{max}}, \quad 1 \leq l \leq L \Big\} \Big\}.\end{aligned} \tag{41}$$

It should be noted that extending the above to a vector-valued function (for a linear elasticity problem) as input and output is relatively straightforward.

Before writing the loss function, note that while an input to FNO is a triplet $(m(x^1, x^2), x^1, x^2)$, during training and testing, the input is applied in a batch. The FNO is applied to input matrix X of size $N \times n_1 \times n_2 \times 3$ altogether, i.e., N samples of m^I and grid locations. FNO produces $N \times n_1 \times n_2 \times d_o$ outputs, corresponding to N functions u^I at all grid locations. Noting this, the optimization problem to train parameters θ reads:

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{N\, n_1\, n_2} \sum_{I=1}^{N} \sum_{j=1}^{n_1} \sum_{k=1}^{n_2} \left| u^I(x_{jk}^1, x_{jk}^2) - F_{NOp}(m^I)(x_{jk}^1, x_{jk}^2) \right|^2. \tag{42}$$

[9] https://github.com/Zhengyu-Huang/Operator-Learning

```python
import torch
import torch.nn as nn
import torch.nn.functional as nnF

class FNO2DLayer(nn.Module):
    def __init__(self, in_channels, out_channels, \
                 modes1, modes2, \
                 apply_act = True, \
                 act = nnF.gelu):
        super(FNO2DLayer, self).__init__()
        ...
        # parameters in nonlocal transformation
        self.scale = (1 / (in_channels * out_channels))
        self.weights1 = nn.Parameter(self.scale * torch.rand(in_channels, out_channels, self.modes1, self.modes2, dtype = torch.cfloat))
        self.weights2 = nn.Parameter(self.scale * torch.rand(in_channels, out_channels, self.modes1, self.modes2, dtype = torch.cfloat))

        # parameters in linear transformation
        self.w = nn.Conv2d(self.out_channels, self.out_channels, 1)

    # complex multiplication
    def compl_mul2d(self, a, b):
        # (batch, in_channel, x,y ), (in_channel, out_channel, x,y) -> (batch, out_channel, x,y)
        op = torch.einsum("bixy,ioxy->boxy",a,b)
        return op

    def fourier_transform(self, x):
        batchsize = x.shape[0]

        # compute Fourier coeffcients
        x_ft = torch.fft.rfft2(x)

        # Multiply relevant Fourier modes
        out_ft = torch.zeros(batchsize, self.out_channels,  x.size(-2), x.size(-1)//2 + 1, device=x.device, dtype=torch.cfloat)
        out_ft[:, :, :self.modes1, :self.modes2] = \
            self.compl_mul2d(x_ft[:, :, :self.modes1, :self.modes2], self.weights1)
        out_ft[:, :, -self.modes1:, :self.modes2] = \
            self.compl_mul2d(x_ft[:, :, -self.modes1:, :self.modes2], self.weights2)

        # return to physical space
        x = torch.fft.irfft2(out_ft,s=(x.size(-2),x.size(-1)))
        return x

    def linear_transform(self, x):
        return self.w(x)

    def forward(self, x):
        x = self.fourier_transform(x) + self.linear_transform(x)
        if self.apply_act:
            return self.act(x)
        else:
            return x
```

Listing 8: Implementation of FNO layer based on Operator-Learning repository[9] De Hoop et al. (2022)

```
...
from torch_fno2dlayer import FNO2DLayer

class FNO2D(nn.Module):
    def __init__(self, num_layers, width, \
                 fourier_modes1, fourier_modes2, \
                 num_Y_components, save_file=None):
        super(FNO2D, self).__init__()
        ...
        # create hidden layers (FNO layers)
        self.fno_layers = nn.ModuleList()
        for _ in range(num_layers):
            self.fno_layers.append(FNO2DLayer(self.width, \
                                              self.width, \
                                              self.fourier_modes1, \
                                              self.fourier_modes2))

        # no activation in the last hidden layer
        self.fno_layers[-1].apply_act = False

        # define input-to-hidden projector
        # input has 3 components: m(x,y), x_1, x_2
        self.input_projector = nn.Linear(3, self.width)

        # define hidden-to-output projector
        # project to the dimension of u(x) \in R^d_o
        self.output_projector = nn.Linear(self.width, self.num_Y_components)

        # record losses
        self.train_loss_log = []
        self.test_loss_log = []
        ...

    def forward(self, X):
        x = self.convert_np_to_tensor(X)

        # input-to-hidden projector
        x = self.input_projector(x)

        # rearrange x so that it has the shape (batch, width, x, y)
        x = x.permute(0, 3, 1, 2)

        # pass through hidden layers
        for i in range(self.num_layers):
            x = self.fno_layers[i](x)

        # rearrange x so that it has the shape (batch, x, y, width)
        x = x.permute(0, 2, 3, 1)

        # hidden-to-output projector
        x = self.output_projector(x)

        return x

    def train(self, train_data, test_data, \
              batch_size=32, epochs = 1000, \
```

```
                lr=0.001, log=True, \
                loss_print_freq = 100, \
                save_model = False, save_file = None, save_epoch = 100):
        # similar to the train() of Listing 7 (PCANet).

    def predict(self, X):
        # similar to the predict() of Listing 5.
```

Listing 9: Implementation of FNO

4.3.2. Architecture and preliminary results

In the implementation, three FNO layers are considered with hidden layer output dimension $d_h = 20$. Other relevant parameters are listed in Table 2. Figure 5 and Figure 6 display the FNO prediction for test input samples, comparing it with the "ground truth". FNO training and testing for the two problems are implemented in the following two notebooks: `FNO-Poisson`[10] and `FNO-Linear Elasticity`[11].

5. Neural Operators applied to Bayesian inference problems

As an application of neural operator surrogates of the parametric PDEs, the Bayesian inference problem is considered to determine the posterior distribution of the parameter field from the synthetic data. Following the key contributions in this topic Dashti and Stuart (2017); Bui-Thanh et al. (2013); Stuart (2010), the Bayesian inference problem in an abstract setting is discussed. Subsequently, the setup and results for the cases of Poisson and linear elasticity models are considered.

5.1. Abstract Bayesian inference problem in infinite dimensions

Consider a parameter field $m \in M$ to be inferred from the data $\mathbf{o} \in \mathbb{R}^{d_o}$ and the corresponding solution of the PDE, $u = F(m) \in U$. Consider the state-to-observable map $\bar{\mathbf{B}} : U \to \mathbb{R}^{d_o}$ such that $\bar{\mathbf{B}}(u)$ gives the prediction of data $\mathbf{o}$ given u (u depends on m). It is possible to use the observational data to find the m that produces prediction $\bar{\mathbf{B}}(u)$ closely matching the data. In the Bayesian setting, this corresponds to finding the distribution of m such that

$$\mathbf{o} = \bar{\mathbf{B}}(u) + \boldsymbol{\eta} = \bar{\mathbf{B}}(F(m)) + \boldsymbol{\eta}\,, \tag{43}$$

where $\boldsymbol{\eta} \sim \mathbf{N}(\mathbf{0}, \boldsymbol{\Gamma}_o)$ is the combined noise due to noise in the data and modeling error, and $\boldsymbol{\Gamma}_o$ is a $d_o \times d_o$ covariance matrix. The covariance matrix is assumed to be of the form $\boldsymbol{\Gamma}_o = \sigma_o^2 \mathbf{I}$, where σ_o is the standard deviation and $\mathbf{I}$ the d_o-dimensional identity matrix.

Generally, the admissible space of parameter fields M_{ad} is a subspace of M, and it is obtained by introducing some restrictions on $m \in M$. For example, diffusivity and Young's modulus parameters in Poisson and linear elasticity problems must be strictly positive at all spatial locations for the

[10] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/poisson/FNO/training_and_testing.ipynb

[11] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/problems/linear_elasticity/FNO/training_and_testing.ipynb

problem to be well-posed. Focusing on our applications, to ensure m is positive, it is taken as a transformation of $w \in W$ as follows

$$m = \alpha_m \exp(w) + \beta_m \,, \tag{44}$$

where, α_m and β_m are constants. In what follows, the inference problem is posed on $w \in W$ with W assumed to be a Hilbert space and m given by (44). Introducing a parameter-to-observable map $\mathbf{B} : W \to \mathbb{R}^{d_o}$ such that $\mathbf{B}(w) = \bar{\mathbf{B}}(F(m(w)))$ leads to the problem of finding the probability distribution of w such that

$$\mathbf{o} = \mathbf{B}(w) + \boldsymbol{\eta} \,. \tag{45}$$

The above equation prescribes the conditional probability distribution of data $\mathbf{o}$ given w. The conditional probability distribution is referred to as likelihood, and using $\mathbf{o} - \mathbf{B}(w) = \boldsymbol{\eta}$, it is given by the translation of $\mathbf{N}(\mathbf{0}, \boldsymbol{\Gamma}_o)$ by $\mathbf{B}(w)$, i.e.,

$$\pi_{like}(\mathbf{o}|w) = \mathbf{N}(\mathbf{o} - \mathbf{B}(w), \boldsymbol{\Gamma}_o) = \frac{1}{\sqrt{\det(\boldsymbol{\Gamma}_o)\,(2\pi)^{\frac{d_o}{2}}}} \exp\left[-\frac{1}{2}(\mathbf{o} - \mathbf{B}(w))^T \boldsymbol{\Gamma}_o^{-1} (\mathbf{o} - \mathbf{B}(w))\right] . \tag{46}$$

It is useful to define a likelihood potential function Φ as follows:

$$\Phi(w) := -\log \pi_{like}(\mathbf{o}|w) \,, \tag{47}$$

where, the data $\mathbf{o}$ is assumed to be fixed so that Φ can be seen as a function of w.

Suppose $\mu^{\oslash}$ is a Gaussian measure on W with the mean function 0 and covariance operator C, i.e., $\mu^{\oslash} = N(0, C)$, and the parameters α_m and β_m in (44) are assigned some fixed values. This results in the log-normal prior distribution on model parameter field m. The choice of C, α_m, and β_m is based on the prior knowledge on m.

Bayes' rule relates the prior measure $\mu^{\oslash}$, the likelihood π_{like}, and posterior measure $\mu^{\mathbf{o}}$ (distribution of w conditioned on data $\mathbf{o}$) as follows:

$$\frac{\mathrm{d}\mu^{\mathbf{o}}}{\mathrm{d}\mu^{\oslash}}(w) = \frac{1}{Z}\pi_{like}(\mathbf{o}|w) = \frac{1}{Z}\exp\left(-\Phi(w)\right) \,, \tag{48}$$

where Z is the normalizing constant given by

$$Z = \int_W \exp\left(-\Phi(w)\right)\, \mathrm{d}\mu^{\oslash}(w) \,. \tag{49}$$

Bayes' rule (48) indicates how the new posterior and prior measures are related. In fact, given (48), it is straightforward to see

$$\begin{aligned} \mathbb{E}_{\mu^{\mathbf{o}}}[f] &= \int_W f(w)\, \mathrm{d}\mu^{\mathbf{o}}(w) \\ &= \int_W f(w) \frac{\mathrm{d}\mu^{\mathbf{o}}}{\mathrm{d}\mu^{\oslash}}(w)\, \mathrm{d}\mu^{\oslash}(w) = \int_W f(w) \frac{\exp\left(-\Phi(w)\right)}{Z}\, \mathrm{d}\mu^{\oslash}(w) \\ &= \mathbb{E}_{\mu^{\oslash}}\left[f(\cdot)\frac{\exp\left(-\Phi(\cdot)\right)}{Z}\right] . \end{aligned} \tag{50}$$

5.1.1. Markov chain Monte Carlo (MCMC) method to sample from the posterior measure

To sample from posterior measure $\mu^{\mathbf{o}}$, i.e., to generate samples $w \in W$ with measure $\mu^{\mathbf{o}}$, there are several algorithms available, see (Cotter et al., 2013, Section 4) and Dashti and Stuart (2017). This work utilizes the preconditioned Crank-Nicolson (pCN) method due to its simplicity. The pCN algorithm based on Cotter et al. (2013) is as follows:

1. Compute initial sample $w^{(0)} \sim \mu^{\oslash} = N(0, C)$
2. For $1 \leq k \leq k_{max}$
   (a) Propose $v^{(k)} = u^{(k)}\sqrt{1-\beta^2} + \beta\xi$, where $\xi \sim \mu^{\oslash}$
   (b) Compute $a = a(w^{(k)}, v^{(k)}) = \Phi(w^{(k)}) - \Phi(v^{(k)})$, and draw $b \sim \text{Uniform}[0, 1]$
   (c) If $\exp(a) > b$ (equivalently $a > \log(b)$), accept $v^{(k)}$ and set $w^{(k+1)} = v^{(k)}$. Else, $w^{(k+1)} = w^{(k)}$.
   (d) $k \to k + 1$
3. Burn initial k_{burn} samples, and take $\{w^{(k)}\}_{k=k_{burn}+1}^{k_{max}}$ as the samples from posterior. The mean of posterior samples is simply

$$\bar{w} = \frac{1}{k_{max} - k_{burn}} \sum_{k=k_{burn}+1}^{k_{max}} w^{(k)} \,. \tag{51}$$

In the above, β is the hyperparameter. To verify that the algorithm works as intended, consider two cases: (1) when the current sample $w^{(k)}$ has lower cost compared to the proposed sample $v^{(k)}$, i.e., $\Phi(u^{(k)}) < \Phi(v^{(k)})$. In this case, $\exp(a) < 1$, and therefore the proposed sample $v^{(k)}$ may be accepted or rejected depending on the draw b. (2) when the proposal has a lower cost compared to the current sample, $a > 0$ and $\exp(a) > 1$, the proposed sample will be accepted regardless of the draw b.

In Listing 10, the implementation of MCMC based on the pCN method is shown. The core implementation of the pCN method is in functions `proposal` and `sample`. The listing also shows the application of surrogate models in computing the forward solution in the `solveFwd` function.

```
...
import numpy as np
...
from state import State
from tracer import Tracer
...

class MCMC:
    def __init__(self, model, prior, data, sigma_noise, pcn_beta = 0.2, \
                 surrogate_to_use = None, surrogate_models = None, seed = 0):
        # model class that provides solveFwd()
        self.model = model
        ...
        # prior class that provides () and logPrior()
        self.prior = prior

        # data (dict) that provides x_obs, u_obs, m_true, u_true, etc.
        self.data = data
```

```
19
20          # noise (std-dev) in the observations
21          self.sigma_noise = sigma_noise
22
23          # preconditioned Crank-Nicolson parameter
24          self.pcn_beta = pcn_beta
25          ...
26          # current and proposed input parameter and state variables
27          self.current = State(self.m_dim, self.u_dim, self.u_obs_dim)
28          self.proposed = State(self.m_dim, self.u_dim, self.u_obs_dim)
29          self.init_sample = State(self.m_dim, self.u_dim, self.u_obs_dim)
30          ...
31          # tracer
32          self.tracer = Tracer(self)
33          self.log_file = None
34
35      def solveFwd(self, current):
36          if self.surrogate_to_use is not None:
37              current.u = self.surrogate_models[self.surrogate_to_use].solveFwd(current.
    m)
38          else:
39              current.u = self.model.solveFwd(u = current.u, m = current.m, transform_m
    = True)
40
41          return current.u
42
43      def state_to_obs(self, u):
44          # interpolate PDE solution on observation grid
45          if self.u_comps == 1:
46              return griddata(self.u_nodes, u, self.x_obs, method='linear')
47          else:
48              num_nodes = self.u_nodes.shape[0]
49              num_grid_nodes = self.x_obs.shape[0]
50              obs = np.zeros(num_grid_nodes*2)
51              for i in range(self.u_comps):
52                  obs[i*num_grid_nodes:(i+1)*num_grid_nodes] = griddata(self.u_nodes, u[
    i*num_nodes:(i+1)*num_nodes], self.x_obs, method='linear')
53
54              return obs
55
56      def logLikelihood(self, current):
57          current.u = self.solveFwd(current)
58          current.u_obs = self.state_to_obs(current.u)
59          current.err_obs = current.u_obs - self.u_obs
60          current.cost = 0.5 * np.linalg.norm(current.err_obs)**2 / self.sigma_noise**2
61          current.log_likelihood = -current.cost
62
63          return current.log_likelihood
64
65      def logPosterior(self, current):
66          current.log_prior = self.prior.logPrior(current.m)
67          current.log_likelihood = self.logLikelihood(current)
68          current.log_posterior = current.log_prior + current.log_likelihood
69
70          return current.log_posterior
71
```

```
    def proposal(self, current, proposed):
        # preconditioned Crank-Nicolson
        proposed.m, proposed.log_prior = self.prior(proposed.m)
        return self.pcn_beta * proposed.m + np.sqrt(1 - self.pcn_beta**2) * current.m

    def sample(self, current):
        # compute the proposed state
        self.proposed.m = self.proposal(current, self.proposed)
        self.proposed.log_posterior = self.logPosterior(self.proposed)

        # accept or reject (based on -log-likelihood
        alpha = current.cost - self.proposed.cost
        if alpha > np.log(np.random.uniform()):
            current.set(self.proposed)
            return 1

        return 0

    def run(self, init_m = None, n_samples = 1000, \
            n_burnin = 100, pcn_beta = 0.2, sigma_noise = 0.01, ...):
        ...
        # run the MCMC
        init_done = False
        for i in range(n_samples + n_burnin):
            # sample
            accept = self.sample(self.current)

            # postprocess/print
            self.process_and_print(i)

            if i < n_burnin: continue

            self.save(i, self.current, accept)

        # save the final state
        self.tracer.append(i, self.current, accept, force_save=True)

        # print final message
        end_time = time.perf_counter()
        self.logger('-'*50)
        self.logger('MCMC finished in {:.3e}s. \nTotal samples: {:4d}, Accepted
    samples: {:4d}, Acceptance Rate: {:.3e}, Cost mean: {:.3e}'.format(end_time -
    start_time, n_samples + n_burnin, self.tracer.acceptances, self.tracer.
    current_acceptance_rate, self.tracer.accepted_samples_cost_mean))
        self.logger('-'*50)
```

Listing 10: Core functions of MCMC implementation using pCN method

5.2. Inference of the diffusivity in Poisson problem

This section explores the application of neural operators as a surrogate in the Bayesian inference of the diffusivity field m in the Poisson problem. Suppose that $u = F(m)$ solves the Poisson problem given m, and $F_{NOp}(m)$ is the neural operator approximation of u, where $NOp \in \{DeepONet, PCANet, FNO\}$ The inference problem is posed on the function $w \in W$ with $m = \alpha_m \exp(w) + \beta_m$ to ensure that the diffusivity is positive. The following sub-section presents the prior measure and synthetic data generation, followed by the sub-section on inference results.

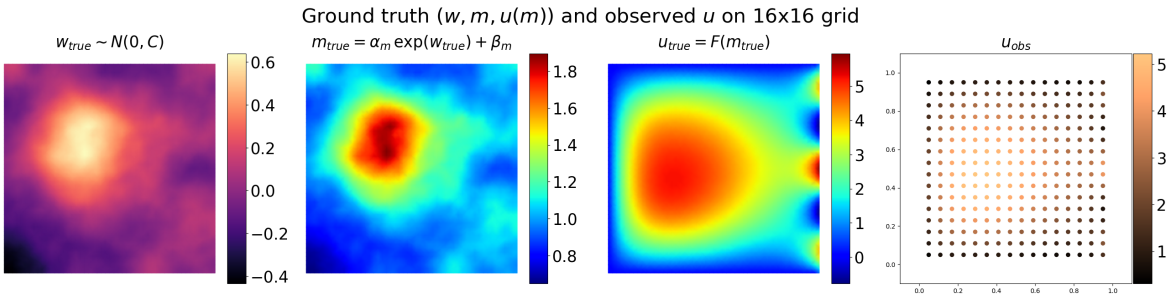


Figure 8: Synthetic w and corresponding diffusivity m and the solution u of the Poisson problem. The data $\mathbf{o} \in \mathbb{R}^{d_o}$ with $d_o = 16^2$ is obtained via the interpolation of u on 16×16 grid over $D_u = (0,1)^2$.

5.2.1. Setup of the forward problem, prior measure, and synthetic data

The setup of the forward problem is the same as in Section 3.1.1. For the prior measure $\mu^{\varnothing}$ on W, consider the probability measure μ_Z described in Section 3.1.1. The values of the α_m and β_m parameters in the transformation of w into m, see (44), and parameters of the covariance operator C are given in Section 3.1.1. ETheprior measure is the same as the probability measure $w \sim \mu_Z$ to generate the input data $\{m^I\}$ for the neural operator.

Synthetic data is obtained by taking the specific function w as shown in Figure 8 (see the figure on the left). Corresponding to w, the "true" diffusivity m is obtained by transforming w following (44), and the "true" solution u is obtained. The data $\mathbf{o}$ is the values of "true" u on 16×16 grid over the domain $D_u = (0,1)^2$. In Figure 8, synthetic data along with the w, m, and u fields are plotted; the implementation can be found in the notebook `Generate_GroundTruth.ipynb`[12].

5.2.2. Inference results

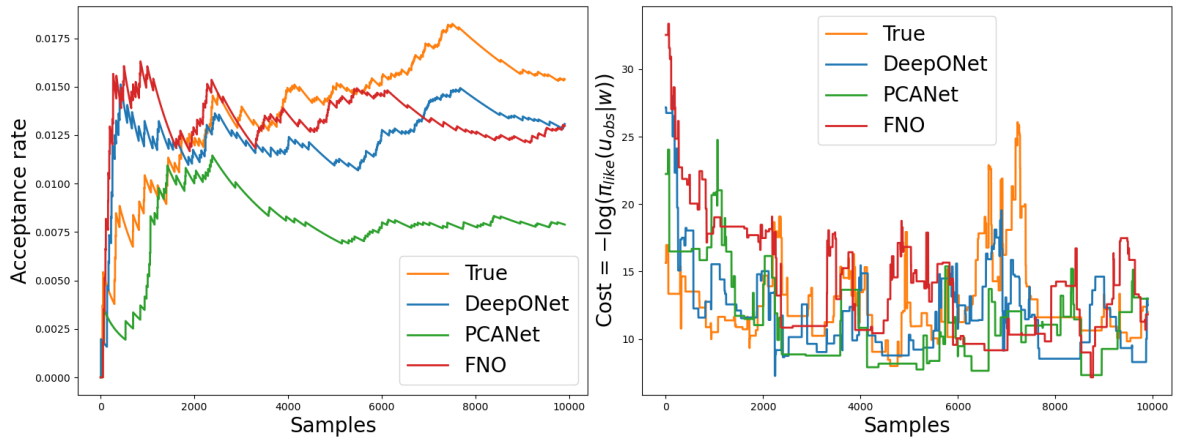
Using the setup, Gaussian prior measure, and synthetic data, MCMC was used to generate the posterior samples. Parameters of the MCMC simulation are as follows: $k_{max} = 10500$, $k_{burn} = 500$, $\beta = 0.2$, and $\sigma_o = 1.329$ (recall that the covariance matrix in the noise model is $\mathbf{\Gamma}_o = \sigma_o^2 \mathbf{I}$). Here, σ_o is taken to be 5% of the mean of the $\mathbf{o}$ (i.e., $\sigma_o = 0.05 \times \frac{1}{d_o} \sum_{i=1}^{d_o} \mathbf{o}_i$). The same MCMC simulation with four different forward models was performed. In the first case, the state field u was computed using the finite element approximation of the Poisson problem, i.e., using the "true" model. The remaining three simulations utilized DeepONet, PCANet, and FNO neural operator approximations of the forward problem. Here, the trained neural operators from the Section 4 are used; their accuracy for random samples from Gaussian prior measure is shown in Figure 5 for the Poisson problem. The notebook `BayesianInversion.ipynb`[13] sets up the problem, loads trained neural operators, and runs MCMC simulation.

Acceptance rate and cost during MCMC simulations for four inference problems corresponding to the "true" and surrogate models are shown in Figure 9. The results, including the posterior mean and a sample from the posterior, from the "true" model are shown in Figure 10. The results with DeepONet, PCANet, and FNO surrogates are shown in Figure 11. Comparing the results when the "true" model is used, all three surrogates produced a posterior mean of w with a slightly

[12] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/applications/bayesian_inverse_problem_poisson/Generate_GroundTruth.ipynb

[13] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/applications/bayesian_inverse_problem_poisson/BayesianInversion.ipynb

higher error. However, the error of posterior mean m obtained using the surrogates is about one percent. This demonstrates that neural operators are robust for the current Bayesian inference problem, even when the target parameter field has features that the prior samples can not produce.



(a) Acceptance rate

(b) Cost

Figure 9: Acceptance rate and cost during MCMC simulation for the Poisson problem.

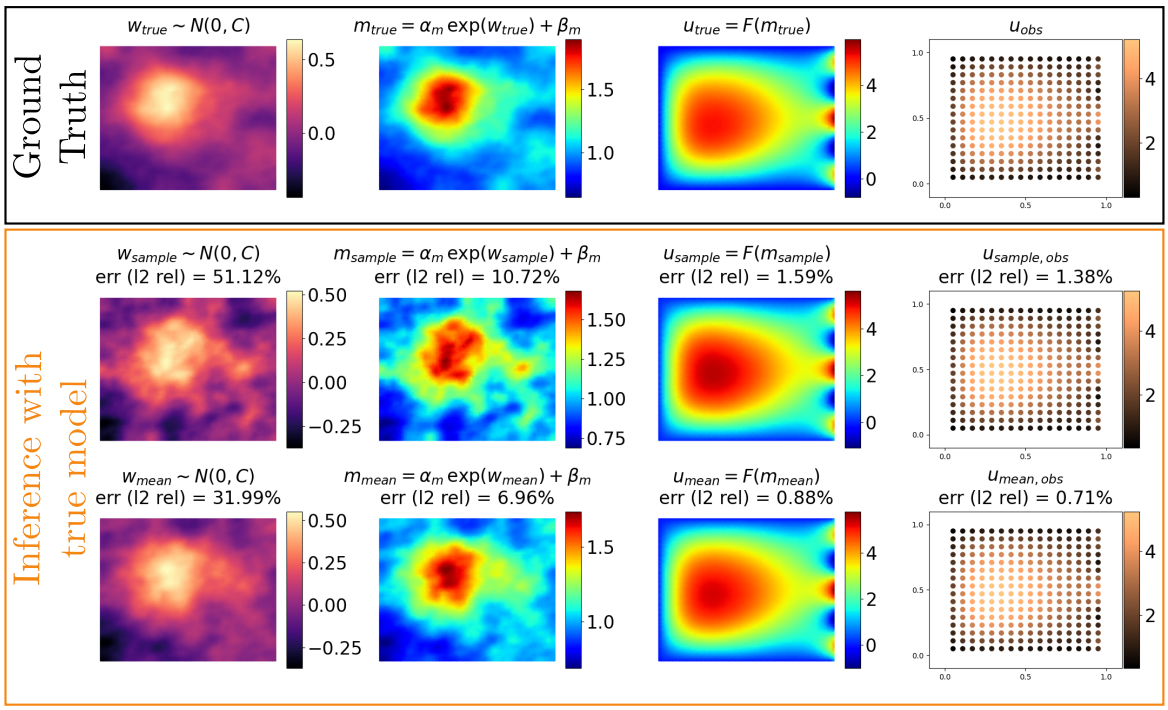


Figure 10: Bayesian inference of diffusivity in the Poisson problem using the "true" model (numerical approximation of PDE). The top panel shows the synthetic w field used to generate m, the corresponding PDE solution u, and the observation of u at 16×16 grid points. The panel below shows the posterior sample and posterior mean using the true model.

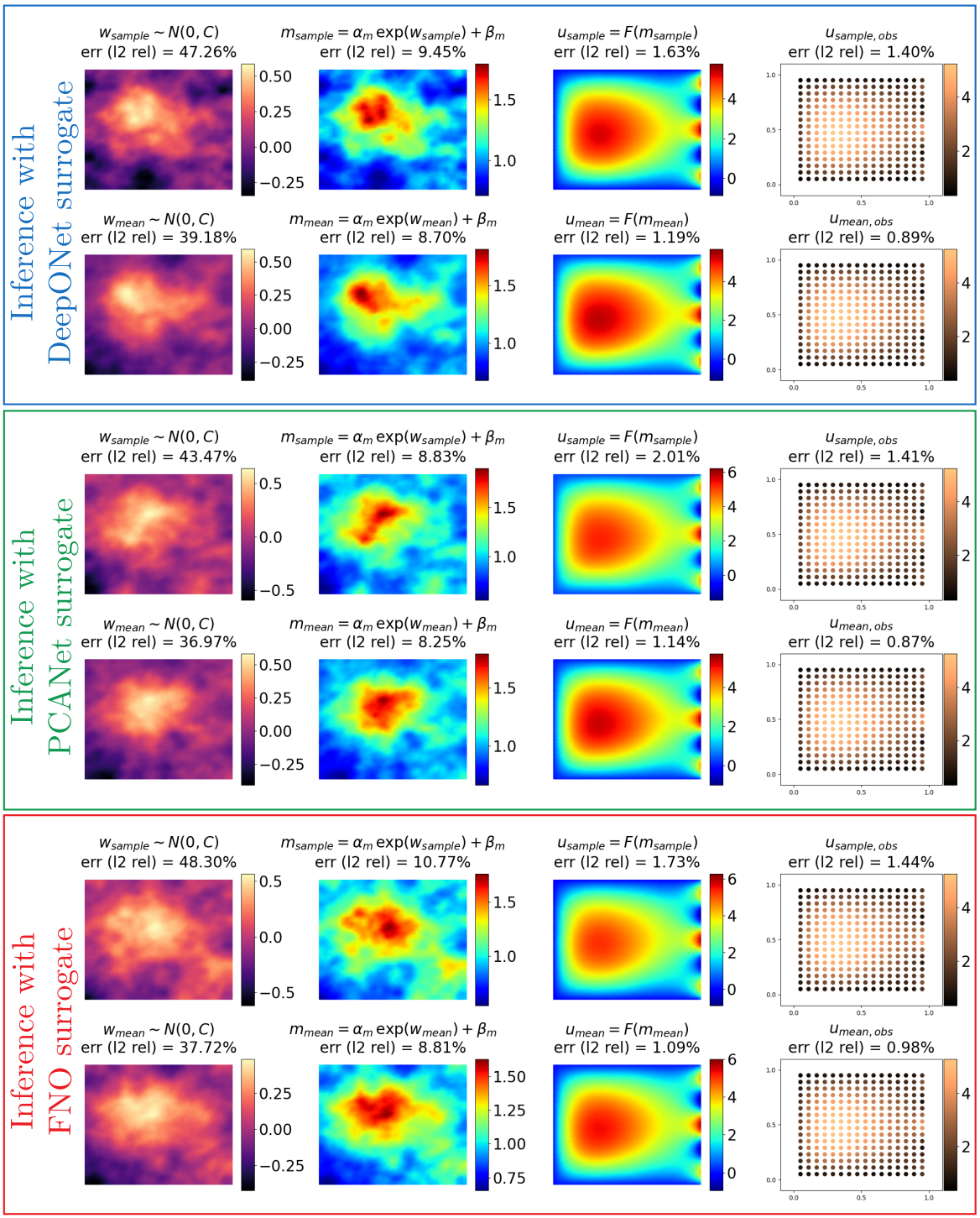


Figure 11: Comparing the Bayesian inference of diffusivity in the Poisson problem using DeepONet, PCANet, and FNO surrogates. The panels show the posterior sample and posterior mean using different surrogates. These results should be compared with the inference results using the "true" model in Figure 10.

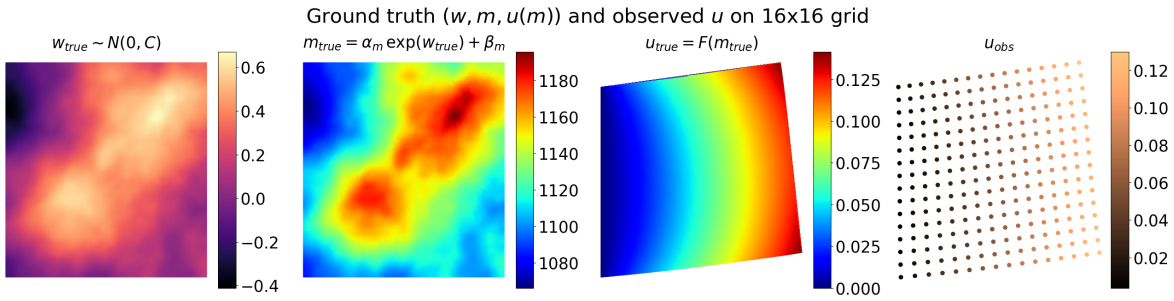


Figure 12: Synthetic w field and corresponding Young's modulus field m and the solution u of the linear elasticity problem. The data $\mathbf{o} \in \mathbb{R}^{d_o}$ with $d_o = 2 \times 16^2$ is obtained via the interpolation of displacement field u on 16×16 grid over $D_u = (0, 1)^2$.

5.3. Inference of Young's modulus in linear elasticity problem

This section explores the application of neural operators as a surrogate in the Bayesian inference of Young's modulus field m in the linear elasticity problem. As in the case of the Poisson problem, to ensure that Young's modulus is positive, the inference problem is posed for the function $w \in W$ such that $m = \alpha_m \exp(w) + \beta_m$. The following sub-section presents the prior measure and synthetic data generation, followed by the sub-section on inference results.

5.3.1. Setup of the forward problem, prior measure, and synthetic data

The setup of the forward problem is the same as in Section 3.2.1. The Gaussian prior measure $\mu^{\oslash}$ on W is the same as μ_Z used in generating training data for neural operators. The values of the α_m and β_m and the covariance operator parameters are given in Section 3.2.1.

The data is generated synthetically following the same procedure as in the case of the Poisson problem in Section 5.2.1. In this case, however, data at a grid point consists of two scalars corresponding to the displacement components, and, as a result, $d_o = 2 \times 16^2$ in $\mathbf{o} \in \mathbb{R}^{d_o}$. In Figure 12, synthetic data along with the w, m, and u fields are plotted; the implementation can be found in the link `Generate_GroundTruth.ipynb`[14].

5.3.2. Inference results

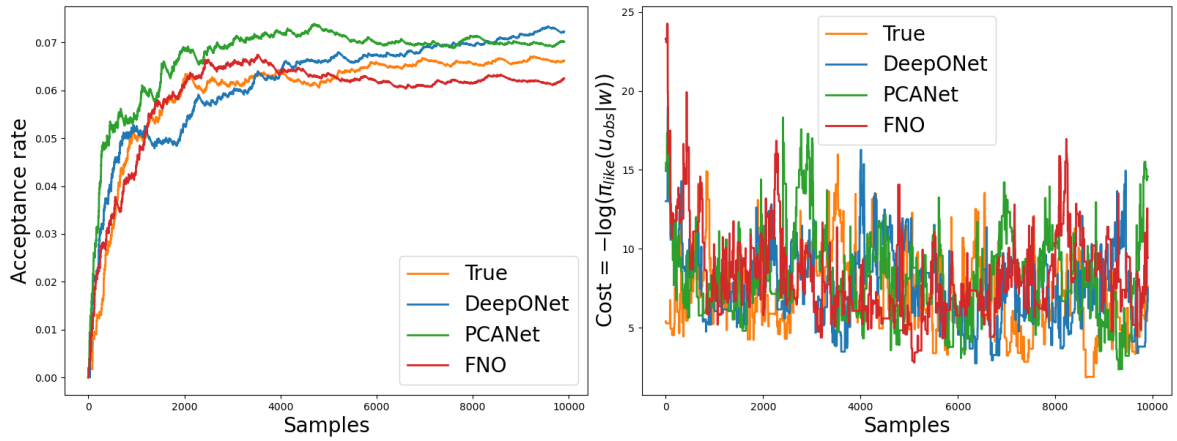
Parameters of the MCMC simulation are as follows: $k_{max} = 10500$, $k_{burn} = 500$, $\beta = 0.15$, and $\sigma_o = 0.0411$. Here, σ_o is taken to be 1% of the mean of the $\mathbf{o}$. The notebook in the link `BayesianInversion.ipynb`[15] sets up the problem, loads trained neural operators, and runs MCMC simulation.

Acceptance rate and cost during MCMC simulations for four inference problems corresponding to the "true" and surrogate models are shown in Figure 13. The results, including the posterior mean and a sample from the posterior, from the "true" model are shown in Figure 14. The results with DeepONet, PCANet, and FNO surrogates are shown in Figure 15. In this case, the posterior mean obtained using the "true" forward model and the surrogates have almost the same error. Also, the error of the posterior mean of m contained using the "true" model and surrogates is less

[14] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/applications/bayesian_inverse_problem_linear_elasticity/Generate_GroundTruth.ipynb

[15] https://github.com/CEADpx/neural_operators/blob/survey25_v1/survey_work/applications/bayesian_inverse_problem_linear_elasticity/BayesianInversion.ipynb

than one percent. Note that the synthetic field w had a slightly larger value along the diagonal, and the posterior mean w is seen to capture the variations in w.



(a) Acceptance rate (b) Cost

Figure 13: Acceptance rate and cost during MCMC simulation for the Linear Elasticity problem.

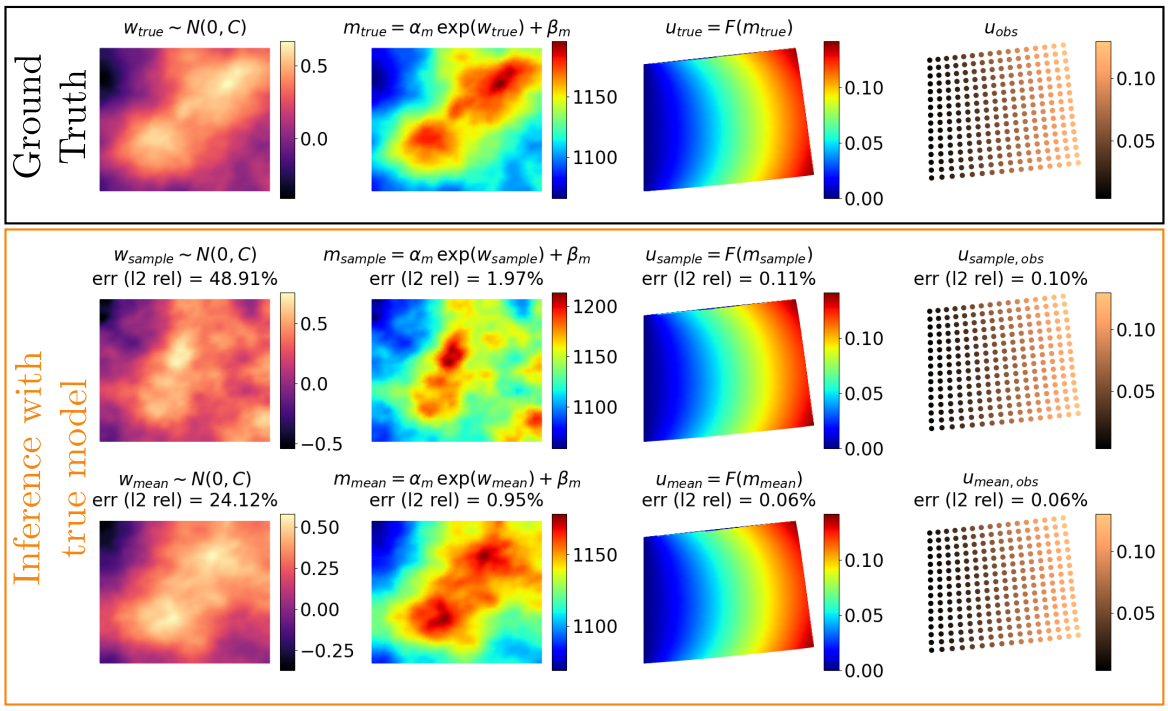


Figure 14: Bayesian inference of Young's modulus in the Linear Elasticity problem using the "true" model. The top panel shows the synthetic w field used to generate m, the corresponding PDE solution u, and the observation of u at 16×16 grid points. The panel below shows the posterior sample and posterior mean using the true model.

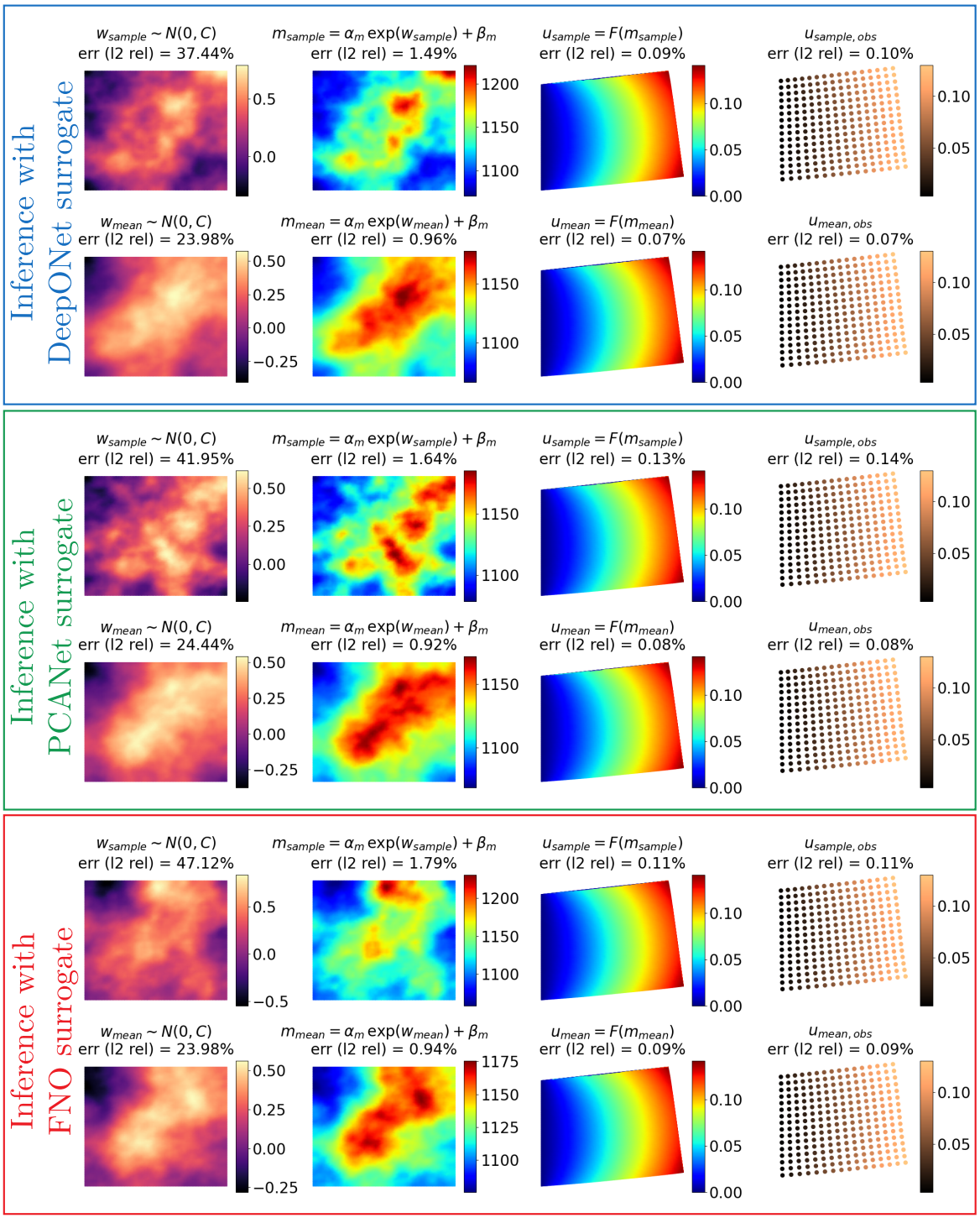


Figure 15: Comparing the Bayesian inference of Young's modulus in the Linear Elasticity problem using DeepONet, PCANet, and FNO surrogates. The panels show the posterior sample and posterior mean using different surrogates. These results should be compared with the inference results using the "true" model in Figure 14.

6. Conclusion

6.1. Growing field of neural operators

While this article has primarily focused on core architectures like DeepONet, PCANet, and the Fourier Neural Operator (FNO), the field of neural operators continues to evolve with several frameworks that draw from other fields and address specific challenges in scientific computing. Physics-Informed Neural Operators (PINO) integrate physical laws directly into the training process, enhancing generalization for complex physical systems; see Li et al. (2024); Goswami et al. (2023); Wang et al. (2021b). The main idea behind PINO is that the loss function includes the residual of the PDE problem in addition to the loss due to model output and data error. Graph-based Neural Operators (GNO) Li et al. (2020b); Kovachki et al. (2023) consider the graph-based discretization of the spatial domain and, therefore, are attractive for problems with irregular domains. Their structure is similar to the FNO structure shown in (34), with a different form of the nonlocal kernel operation. The Wavelet Neural Operator (WNO) Tripura and Chakraborty (2022) shares the structure of FNO with a different nonlocal operator (see (36)). In WNO, nonlocal operation is based on the wavelet transforms. Derivative-Informed Neural Operators (DINO) O'Leary-Roseberry et al. (2024) exploit derivative information to improve learning efficiency and accuracy in gradient-dominated systems. There is a considerable effort to extend neural operator frameworks into the Bayesian domain, integrating uncertainty quantification directly into the operator learning process; see Psaros et al. (2023); Jospin et al. (2022); Garg and Chakraborty (2022). These works aim to provide point estimates of the solution operator and probabilistic predictions that reflect the uncertainty in both the data and the model parameters.

Recently, the Kolmogorov-Arnold Neural Network (KAN) has been proposed Liu et al. (2024) that is inspired by the Kolmogorov-Arnold Representation Theorem, which states that for an n-dimensional function $f : [0, 1]^n \to \mathbb{R}$ the following holds

$$f(x) = f(x_1, x_2, ..., x_n) = \sum_{j=1}^{2n+1} \Phi_i \left(\sum_{i=1}^{n} \Phi_{ij}(x_j) \right) , \tag{52}$$

where $\Phi_i : \mathbb{R} \to \mathbb{R}$ and $\Phi_{ij}(\cdot) : [0, 1] \to \mathbb{R}$ are continuous functions of single variable. Note that the term inside the bracket on the right-hand side is the argument of the function Φ_i. The above mapping that takes input x to $f(x)$ can be seen as a two-layer network, where the first layer takes input x (just one component) and produces $n(2n + 1)$ outputs, each corresponding to $\{\{\Phi_{ij}\}_{i=1}^{n}\}_{j=1}^{2n+1}$, and the second layer takes $n(2n+1)$ and produces $2n+1$ outputs, each corresponding to $\{\Phi_i\}_{i=1}^{2n+1}$. The final output is then obtained by summing up the outputs of the second layer.

Therefore, KAN, as a direct extension of the above equation, can be written as

$$f(x) = f(x_1, x_2, ..., x_n) = \sum_{j=1}^{2n+1} \Phi_i^{\theta} \left(\sum_{i=1}^{n} \Phi_{ij}^{\theta}(x_j) \right) , \tag{53}$$

where, Φ_i^{θ} and Φ_{ij}^{θ} are the parameterized function with parameters θ (e.g., B-spline curve). The parameter θ can be obtained by solving the appropriate optimization problem. The reference Liu et al. (2024) mentions that the above framework is limited in representing arbitrary functions, which prompts the authors to propose the more general form below

$$f(x) \approx f_{KAN}(x) = \left(\Phi_L^{\theta} \circ \Phi_{L-1}^{\theta} \cdots \Phi_1^{\theta} \circ \Phi_0^{\theta} \right) x \, . \tag{54}$$

Here, for layer l, Φ_l^θ are the tensors (say, of size $n_{l+1} \times n_l$), with each component being a parameterized function; see (Liu et al., 2024, Section 2.2). KAN offers an alternative structure compared to multi-layer perception (MLP), and it has led to several new neural network and operator architectures.

Neural operators have demonstrated tremendous potential across multiple scientific and engineering domains. In Bayesian inverse problems, they accelerate computational processes by serving as efficient surrogates for expensive PDE solvers, enabling rapid posterior sampling in high-dimensional spaces Cao et al. (2023, 2024); Gao et al. (2024). Neural operators are attractive for optimization and control problems, where the bottleneck is the computation of the forward solution given the updated parameter field Shukla et al. (2024). The major hurdle in this direction is the unpredictability of the accuracy of neural operator predictions. This stems from the fact that the training data is generated based on prior knowledge of the distribution of the parameter field, and during the optimization and control iterations, the parameter fields for which solutions are sought could be far from the training regime; Jha (2024) shows that neural operators trained on a prior distribution perform poorly in the optimization problem.

Digital twins are another area that is seeing rapid expanse due to increased computational capacity, multiscale multiphysics modeling capabilities, innovation in sensors and actuators to measure and control the system, and growth in artificial intelligence; see the review on digital twins Juarez et al. (2021); Wagg et al. (2020). Neural operators show great potential in aiding the computations in digital twins. They can be used as a surrogate of the computationally demanding task of computing the next state of the system given the current state and model and control parameters Kobayashi and Alam (2024).

6.2. Controlling the neural operator prediction accuracy

Neural operators often struggle to meet high precision demands, particularly in complex PDEs and high-dimensional function spaces. Predictability of accuracy is the central issue in realizing the practical and robust applications of neural operators in uncertainty quantification, Bayesian inference, optimization, and control problems. Below, some of the key works in this direction are reviewed.

The multi-level neural network framework Aldirany et al. (2024) iteratively refines solutions by training successive networks to minimize residual errors from previous levels. This hierarchical approach progressively reduces approximation error. By capturing high-frequency components in subsequent neural networks and therefore reducing modeling error, it addresses the limitations of neural network approximations in capturing these modes due to the low sensitivity of network parameters to higher-frequency modes. The multi-stage neural network framework Wang and Lai (2024) addresses convergence plateauing in deep learning by dividing training into sequential stages, with each stage fitting the residual from the previous one. This approach progressively refines the approximation. The methods in references Aldirany et al. (2024); Wang and Lai (2024) achieve errors near machine precision (10^{-16}), far exceeding the typical 10^{-5} limit of single-stage training.

The Galerkin neural network framework Ainsworth and Dong (2021) integrates the classical Galerkin method with neural networks, using the neurons in a neural network layer as basis functions to approximate variational equations. It adaptively adds new neurons to a single-layer neural network; each neuron corresponds to the basis function, and by adding more neurons, the dimension of approximation is increased in a way that the new subspace produces less error compared to the subspace with smaller dimensions.

The residual-based error correction method Cao et al. (2023); Jha (2024), see Figure 16, building on the idea of using lower-fidelity solutions to estimate modeling errors Jha and Oden (2022), treats the neural operator's prediction as an initial guess and solves a variational problem to correct the residual error. This approach improves accuracy in applications like Bayesian inverse problems and topology optimization, where small errors can propagate and amplify. The trade-off in the residual-based error correction method is the introduction of the linear variational problem on the modeling error field, which must be solved and added to the neural operator prediction to get the improved prediction. For challenging nonlinear problems, the approach leads to significant speedups. In Bayesian settings, it enhances posterior estimates without increasing computational costs Cao et al. (2023). For the example optimization problem of seeking a diffusivity field in the Poisson equation that minimizes the compliance, neural operators, when used as a surrogate of the forward problem, produced minimizers with significant errors (about 80%). The error here is the norm of the difference between the minimizer obtained using the "true" forward model and the surrogate of the forward model. Neural operator surrogates with the residual-based error correction produced minimize with errors below 6%; see Figure 17, where the optimized diffusivity using the "true" model and surrogates are compared.

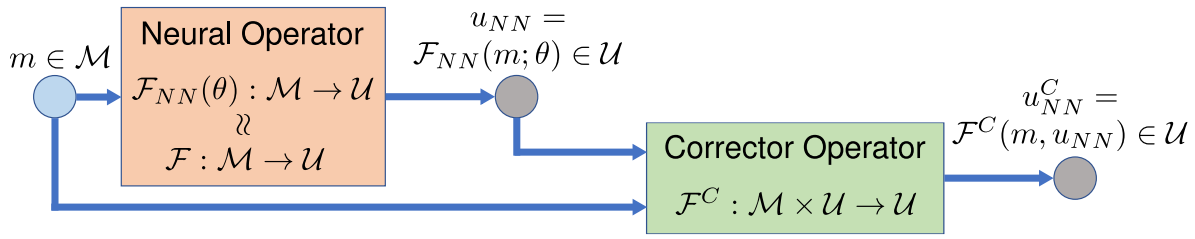


Figure 16: Residual-based error correction of neural operator predictions Cao et al. (2023); Jha (2024). The corrector problem is a linear BVP, and if the predictor u_{NN} is sufficiently close to the PDE solution u, the corrector gives quadratic error reduction Jha (2024).

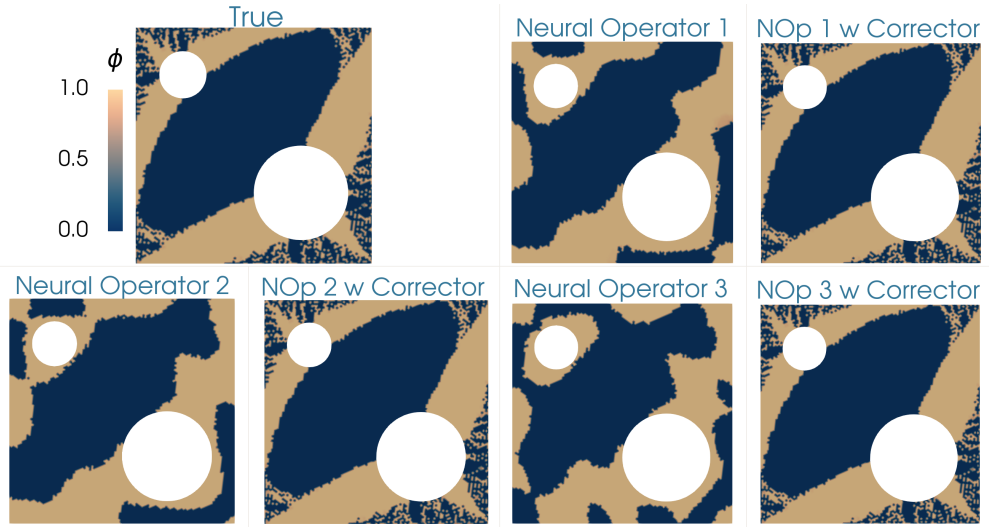


Figure 17: Optimized diffusivity in a Poisson equation and the optimized diffusivity using the neural operator surrogates with and without corrector. For more details about the problem and results, see (Jha, 2024, Section 4.3).

Some of the key limitations of the work mentioned above that address the issue of the reliability of neural networks include

- **Computational overhead.** Many error control methods, like residual-based correctors and multi-level networks, introduce additional computational steps, potentially offsetting the efficiency gains of neural operators.
- **Scalability challenges.** Iterative methods (e.g., multi-level and multi-stage networks) can become computationally intensive and less practical for high-dimensional or large-scale problems, leading to diminishing returns in accuracy improvement.
- **Risk of overfitting.** While multi-stage and multi-level networks aim to reduce residual errors and capture high-frequency components, they risk overfitting in later stages or failing to generalize if residuals are small or noisy.

This underscores the scope for new research in controlling the accuracy of neural operators without compromising the computational speedup associated with neural operators.

6.3. Final thoughts

Neural operators have emerged as transformative tools for solving parametric partial differential equations (PDEs). They offer efficient surrogates that bridge the gap between traditional numerical methods and modern machine learning. Their ability to map between infinite-dimensional function spaces and fast evaluations makes them attractive approximators for parametric PDEs; they can lead to significant speedups in uncertain quantification, Bayesian inference, optimization, and control problems. Even more, for problems where observational data exists and where the confidence in mathematical models is low or in doubt, trained neural operators can combine the features of the model and data in a way that the accuracy of the model does not limit the neural operator predictions.

Despite these advancements, challenges remain, particularly in controlling prediction accuracy and ensuring generalization across diverse problem domains. Strategies like residual-based error correction, multi-level neural networks, and adaptively building neural networks can significantly improve the robustness and precision. Yet, each method presents trade-offs between computational cost and accuracy, underscoring the need for further research into scalable and adaptive error control mechanisms.

Looking forward, the integration of uncertainty quantification, physics-informed constraints, and multi-fidelity modeling represents promising directions for future work. These enhancements and strategies to control the prediction errors will be crucial for deploying neural operators in complex, real-world systems where precision and reliability are paramount.

References

Ainsworth, M., Dong, J., 2021. Galerkin neural networks: A framework for approximating variational equations with error control. SIAM Journal on Scientific Computing 43, A2474–A2501.

Aldirany, Z., Cottereau, R., Laforest, M., Prudhomme, S., 2024. Multi-level neural networks for accurate solutions of boundary-value problems. Computer Methods in Applied Mechanics and Engineering 419, 116666.

Alnæs, M., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M.E., Wells, G.N., 2015. The fenics project version 1.5. Archive of Numerical Software 3.

Bhattacharya, K., Hosseini, B., Kovachki, N.B., Stuart, A.M., 2021. Model reduction and neural networks for parametric PDEs. SMAI Journal of Computational Mathematics, Volume 7 .

Bui-Thanh, T., Ghattas, O., Martin, J., Stadler, G., 2013. A computational framework for infinite-dimensional bayesian inverse problems part i: The linearized case, with application to global seismic inversion. SIAM Journal on Scientific Computing 35, A2494–A2523.

Cao, L., O'Leary-Roseberry, T., Ghattas, O., 2024. Derivative-informed neural operator acceleration of geometric mcmc for infinite-dimensional bayesian inverse problems. arXiv preprint arXiv:2403.08220 .

Cao, L., O'Leary-Roseberry, T., Jha, P.K., Oden, J.T., Ghattas, O., 2023. Residual-based error correction for neural operator accelerated infinite-dimensional bayesian inverse problems. Journal of Computational Physics 486, 112104.

Cotter, S.L., Roberts, G.O., Stuart, A.M., White, D., 2013. MCMC Methods for Functions: Modifying Old Algorithms to Make Them Faster. Statistical Science 28, 424 – 446. URL: https://doi.org/10.1214/13-STS421, doi:10.1214/13-STS421.

Dashti, M., Stuart, A.M., 2017. The Bayesian Approach to Inverse Problems. Springer International Publishing, Cham. pp. 311–428. URL: https://doi.org/10.1007/978-3-319-12385-1_7, doi:10.1007/978-3-319-12385-1_7.

De Hoop, M., Huang, D.Z., Qian, E., Stuart, A.M., 2022. The cost-accuracy trade-off in operator learning with neural networks. arXiv preprint arXiv:2203.13181 .

Fresca, S., Manzoni, A., 2022. POD-DL-ROM: Enhancing deep learning–based reduced order models for nonlinear parametrized PDEs by proper orthogonal decomposition. Computer Methods in Applied Mechanics and Engineering 388, 114181.

Gao, Z., Yan, L., Zhou, T., 2024. Adaptive operator learning for infinite-dimensional bayesian inverse problems. SIAM/ASA Journal on Uncertainty Quantification 12, 1389–1423.

Garg, S., Chakraborty, S., 2022. Variational bayes deep operator network: A data-driven bayesian solver for parametric differential equations. arXiv preprint arXiv:2206.05655 .

Goswami, S., Anitescu, C., Chakraborty, S., Rabczuk, T., 2020. Transfer learning enhanced physics informed neural network for phase-field modeling of fracture. Theoretical and Applied Fracture Mechanics 106, 102447.

Goswami, S., Bora, A., Yu, Y., Karniadakis, G.E., 2023. Physics-informed deep neural operator networks, in: Machine Learning in Modeling and Simulation: Methods and Applications. Springer, pp. 219–254.

Jha, P.K., 2024. Residual-based error corrector operator to enhance accuracy and reliability of neural operator surrogates of nonlinear variational boundary-value problems. Computer Methods in Applied Mechanics and Engineering 419, 116595.

Jha, P.K., Oden, J.T., 2022. Goal-oriented a-posteriori estimation of model error as an aid to parameter estimation. Journal of Computational Physics 470, 111575. doi:10.1016/j.jcp.2022.111575.

Jospin, L.V., Laga, H., Boussaid, F., Buntine, W., Bennamoun, M., 2022. Hands-on bayesian neural networks—a tutorial for deep learning users. IEEE Computational Intelligence Magazine 17, 29–48.

Juarez, M.G., Botti, V.J., Giret, A.S., 2021. Digital twins: Review and challenges. Journal of Computing and Information Science in Engineering 21, 030802.

Kobayashi, K., Alam, S.B., 2024. Deep neural operator-driven real-time inference to enable digital twin solutions for nuclear energy systems. Scientific reports 14, 2101.

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., Anandkumar, A., 2021. Neural operator: Learning maps between function spaces. arXiv preprint arXiv:2108.08481 .

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

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., Anandkumar, A., 2020a. Fourier neural operator for parametric partial differential equations. arXiv preprint arXiv:2010.08895 .

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., Anandkumar, A., 2020b. Neural operator: Graph kernel network for partial differential equations. arXiv preprint arXiv:2003.03485 .

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., Anandkumar, A., 2021. Fourier neural operator for parametric partial differential equations. International Conference on Learning Representations .

Li, Z., Zheng, H., Kovachki, N., Jin, D., Chen, H., Liu, B., Azizzadenesheli, K., Anandkumar, A., 2024. Physics-informed neural operator for learning partial differential equations. ACM/JMS Journal of Data Science 1, 1–27.

Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., Hou, T.Y., Tegmark, M., 2024. Kan: Kolmogorov-arnold networks. arXiv preprint arXiv:2404.19756 .

Lu, L., Jin, P., Karniadakis, G.E., 2019. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. arXiv preprint arXiv:1910.03193 .

Lu, L., Jin, P., Pang, G., Karniadakis, G.E., 2021a. DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. Nature Machine Intelligence .

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

Mandel, J., 2023. Introduction to infinite dimensional statistics and applications. arXiv preprint arXiv:2310.15818 .

O'Leary-Roseberry, T., Chen, P., Villa, U., Ghattas, O., 2024. Derivative-informed neural operator: an efficient framework for high-dimensional parametric derivative learning. Journal of Computational Physics 496, 112555.

Psaros, A.F., Meng, X., Zou, Z., Guo, L., Karniadakis, G.E., 2023. Uncertainty quantification in scientific machine learning: Methods, metrics, and comparisons. Journal of Computational Physics 477, 111902.

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

Stuart, A.M., 2010. Inverse problems: a bayesian perspective. Acta numerica 19, 451–559.

Tripura, T., Chakraborty, S., 2022. Wavelet neural operator: a neural operator for parametric partial differential equations. arXiv preprint arXiv:2205.02191 .

Wagg, D., Worden, K., Barthorpe, R., Gardner, P., 2020. Digital twins: state-of-the-art and future directions for modeling and simulation in engineering dynamics applications. ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering 6, 030901.

Wang, S., Wang, H., Perdikaris, P., 2021a. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. Science advances 7, eabi8605.

Wang, S., Wang, H., Perdikaris, P., 2021b. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. Science advances 7, eabi8605.

Wang, Y., Lai, C.Y., 2024. Multi-stage neural networks: Function approximator of machine precision. Journal of Computational Physics 504, 112865.