Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 393 (2022) 114778
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data

Lu Lu[a,1], Xuhui Meng[b,1], Shengze Cai[b,1], Zhiping Mao[c], Somdatta Goswami[b], Zhongqiang Zhang[d], George Em Karniadakis[b,e,*]

[a] *Department of Chemical and Biomolecular Engineering, University of Pennsylvania, United States of America*
[b] *Division of Applied Mathematics, Brown University, United States of America*
[c] *School of Mathematical Sciences, Xiamen University, United States of America*
[d] *Department of Mathematical Sciences, Worcester Polytechnic Institute, United States of America*
[e] *School of Engineering, Brown University, United States of America*

Received 10 November 2021; received in revised form 20 January 2022; accepted 15 February 2022
Available online 11 March 2022

Abstract

Neural operators can learn nonlinear mappings between function spaces and offer a new simulation paradigm for real-time prediction of complex dynamics for realistic diverse applications as well as for system identification in science and engineering. Herein, we investigate the performance of two neural operators, which have shown promising results so far, and we develop new practical extensions that will make them more accurate and robust and importantly more suitable for industrial-complexity applications. The first neural operator, *DeepONet*, was published in 2019 (Lu et al., 2019), and its original architecture was based on the universal approximation theorem of Chen & Chen (1995). The second one, named Fourier Neural Operator or *FNO*, was published in 2020 (Li et al., 2020), and it is based on parameterizing the integral kernel in the Fourier space. DeepONet is represented by a summation of products of neural networks (NNs), corresponding to the branch NN for the input function and the trunk NN for the output function; both NNs are general architectures, e.g., the branch NN can be replaced with a CNN or a ResNet. According to Kovachki et al. (2021), FNO in its continuous form can be viewed conceptually as a DeepONet with a specific architecture of the branch NN and a trunk NN represented by a trigonometric basis. In order to compare FNO with DeepONet computationally for realistic setups, we develop several extensions of FNO that can deal with complex geometric domains as well as mappings where the input and output function spaces are of different dimensions. We also develop an extended DeepONet with special features that provide inductive bias and accelerate training, and we present a faster implementation of DeepONet with cost comparable to the computational cost of FNO, which is based on the Fast Fourier Transform.

We consider 16 different benchmarks to demonstrate the relative performance of the two neural operators, including instability wave analysis in hypersonic boundary layers, prediction of the vorticity field of a flapping airfoil, porous media simulations in complex-geometry domains, etc. We follow the guiding principles of FAIR (Findability, Accessibility, Interoperability, and Reusability) for scientific data management and stewardship. The performance of DeepONet and FNO is comparable for relatively simple settings, but for complex geometries the performance of FNO deteriorates greatly. We also

* Corresponding author at: Division of Applied Mathematics, Brown University, United States of America.
E-mail address: george_karniadakis@brown.edu (G.E. Karniadakis).
[1] These authors contributed equally to this work.

https://doi.org/10.1016/j.cma.2022.114778
0045-7825/© 2022 Elsevier B.V. All rights reserved.

compare theoretically the two neural operators and obtain similar error estimates for DeepONet and FNO under the same regularity assumptions.
© 2022 Elsevier B.V. All rights reserved.

Keywords: Nonlinear mappings; Operator regression; Deep learning; DeepONet; FNO; Scientific machine learning

1. Introduction

While there have been rapid developments in machine learning in the last 20 years and there is currently plenty of euphoria and admiration about the so-called "unreasonable effectiveness of deep learning in artificial intelligence" [1], the development of physics-informed machine learning and its application to scientific applications is relatively recent [2]. Physics-informed neural networks (PINNs) were introduced first in 2017 [3,4] for forward, inverse and hybrid problems, and since then there have also been rapid developments in this area [5–13], although the achievements so far in scientific and engineering applications have been modest compared to applications in imaging, speech recognition and natural language processing. According to some estimates, last year alone close to 100,000 papers on machine learning were published, all of which were based on the assumption of the universal function approximation of neural networks — a theoretical work that goes back to the early 1990s [14,15].

At about the same time, a much less known theorem of Chen & Chen [16] on the universal operator approximation by single-layer neural networks was developed but remained largely unknown until our work on DeepONet in 2019 [17] with subsequent theoretical and computational extensions in [18]. Unlike function regression, operator regression aims to map infinite-dimensional functions (inputs) to infinite-dimensional functions (outputs). The work in [18] extended the theorem of Chen & Chen [16] to deep neural networks, which are more expressive and break the curse of dimensionality in the input space. More importantly, from the computational point of view, the new paradigm of operator regression allows for simulating the dynamics of complex nonlinear systems, e.g., fluid flows, corresponding to different boundary and initial conditions without the need for retraining the neural network. As noted independently by DeepMind researcher Irina Higgins [19], "Once DeepONet is trained, it can be applied to new input functions, thus producing new results substantially faster than numerical solvers. Another benefit of DeepONet is that it can be applied to simulation data, experimental data or both, and the experimental data may span multiple orders of magnitude in spatio-temporal scales, thus allowing scientists to estimate dynamics better by pooling the existing data". DeepONet can be applied to partial differential equations (PDEs) but also to learning explicit mathematical operators, e.g., integration, fractional derivatives, Laplace transforms, etc. [18].

Another parallel effort on operator regression started in 2020 with a paper on a graph kernel network (GKN) for PDEs [20]. The authors represented the infinite-dimensional mapping by composing nonlinear activation functions and a class of integral operators with the kernel integration computed by message passing on graph networks. Unfortunately, GKN was of limited use as it was shown to be unstable with the increase of the number of hidden layers [21]. In fact, a recent extension of GKN using nonlocal operator regression [21] shows that this type of neural operator is powerful if it is stable, but no extensive experiments have been performed yet. A different architecture was then proposed by the same group in [22], where they formulated the operator regression by parameterizing the integral kernel directly in Fourier space. They demonstrated very good accuracy and efficiency for relatively simple settings, including the Burgers' equation, Darcy flow, and the Navier–Stokes equations. However, they made the claim that "The Fourier neural operator is the first ML-based method to successfully model turbulent flows with zero-shot super-resolution. It is up to three orders of magnitude faster compared to traditional PDE solvers. Additionally, it achieves superior accuracy compared to previous learning-based solvers under fixed resolution". This is of course erroneous as the flow considered was a simple smooth laminar flow, and their comparison with DeepONet was not properly conducted. In fact, as we will demonstrate herein, DeepONet can achieve similar and in fact better accuracy for the same or similar benchmarks presented in [22] and even superior accuracy in realistic problems involving complex-geometry domains and noisy input data.

In this paper, we present a very systematic and transparent study of comparing the performance of DeepONet and FNO for 16 different benchmarks selected carefully to highlight both the advantages and the limitations of the two neural operators. According to an independent work by [23], FNO in its continuous form can be viewed as a DeepONet with a specific architecture of the branch and a trunk represented by a trigonometric basis. Hence, in the current work we introduce two significant enhancements to FNO so that they can deal with mappings of

different dimensionality as well as with complex-geometry domains, so that we can make sensible comparisons with DeepONet, which is a general neural operator. We also introduce various enhancements to DeepONet to accelerate its training and increase its accuracy, introducing for example the POD modes in the trunk net, obtained readily from the available training datasets. In addition to computational tests, we also perform a theoretical comparison of DeepONet versus FNO, following the published work on the theory of DeepONet in [16,24–27], and on the more recent theory of FNO in [23,28]. On this point, it is worth noted that DeepONet was based from the onset on the theorem of Chen & Chen [16], whereas the formulation of FNO was not theoretically justified originally, and the recent theoretical work covers only invariant kernels. There are also other methods for operator regression such as [29–32], but in this work we only consider DeepONet and FNO.

In the following, we summarize the new contributions of the current work in addition to designing 16 different benchmarks and obtaining new results for both DeepONet and FNO along with their new extensions. More specifically, the new developments for DeepONet are the following:

- We introduce extra features in the trunk net and the branch net.
- We impose hard-constraints for Dirichlet and periodic boundary conditions via a modified trunk net.
- We develop a new extension, POD-DeepONet, that employs the POD modes of the training data as the trunk net.
- We analyze and test a new DeepONet scaling that leads to accuracy improvement.
- We extend DeepONet to deal with multiple outputs.
- We present a new fast implementation of DeepONet, comparable to FNO for similar settings.

Similarly, the new developments for FNO are the following:

- dFNO+: We extend FNO to nonlinear mappings with inputs and outputs defined on different domains.
- gFNO+: We extend FNO to nonlinear mappings with inputs and outputs defined on a complex geometry.
- We add extra features by using them as extra network inputs.

Moreover, we employ normalization of both inputs and outputs for DeepONet and FNO and demonstrate its effect.

In our comparative studies, we designed benchmarks with important features typically encountered in real world applications, such as complex-geometry domains, non-smooth solutions, unsteadiness, and noisy data. On the theoretical side, our contribution is on developing error estimation of the network size by emulating Fourier methods both for DeepONet and FNO. We aim to make this study, including codes and data, accessible to all, and to the degree possible we have followed the FAIR (Findability, Accessibility, Interoperability, and Reusability) guiding principles for scientific data management and stewardship [33].

The paper is organized as follows. In Section 2, we define the problem setup and the data types. In Section 3, we describe the DeepONet and FNO architectures as well as their extensions. In Section 4, we present a theoretical comparison of the approximation theorems for the two neural operators and in addition we compare their error estimates for the solution of the Burgers' equation. In Section 5, we compare the performance of DeepONet and FNO for 16 different benchmarks listed in Table 3. Finally, we conclude with a summary in Section 6. The Appendix includes the proof of the theorem presented in the main text, description of the datasets, description of the architectures employed, data generation for the Darcy problems and the cavity flows, and a comparative study on the computational cost for different neural operators.

2. Operator learning

We first present the problem setup of operator learning and then discuss some important aspects of the dataset used for learning.

2.1. Problem setup

We consider a physical system, which involves multiple functions. Among these functions, we are usually interested in one function and aim to predict this function from other functions. For example, when the physical system is described by partial differential equations (PDEs), these functions typically include the PDE solution $u(x,t)$, the forcing term $f(x,t)$, the initial condition (IC) $u_0(x)$, and the boundary conditions (BCs) $u_b(x,t)$, where x and t are the space and time coordinates, respectively.

We denote the input function by v defined on the domain $D \subset \mathbb{R}^d$, e.g., $f(x,t)$ or $u_0(x)$,

$$v : D \ni x \mapsto v(x) \in \mathbb{R},$$

and denote the output function by u defined on the domain $D' \subset \mathbb{R}^{d'}$

$$u : D' \ni \xi \mapsto u(\xi) \in \mathbb{R}.$$

Let $\mathcal{V}$ and $\mathcal{U}$ be the spaces of v and u, respectively, and D and D' could be two different domains. Then, the mapping from the input function v to the output function u is denoted by an operator

$$\mathcal{G} : \mathcal{V} \ni v \mapsto u \in \mathcal{U}.$$

In this study, we aim to approximate $\mathcal{G}$ by neural networks and train the network from a training dataset $\mathcal{T} = \left\{\left(v^{(1)}, u^{(1)}\right), \left(v^{(2)}, u^{(2)}\right), \ldots, \left(v^{(m)}, u^{(m)}\right)\right\}$.

2.2. Data types

Deep learning is a powerful method for leveraging big data, but in many science and engineering problems, the available data is not "big" enough to ensure accuracy and reliability of deep learning models. What we may have instead is "dinky, dirty, dynamics, and deceptive data" as first characterized by Alexander Kott, chief of the Network Science Division of the US Army Research Laboratory [34]. On the other hand, scientific data should meet principles of **findability, accessibility, interoperability, and reusability** (FAIR) [33].

In addition, we have to facilitate multi-modality input data that may come from diverse sources, e.g., static images, videos, Schlieren photography, particle image velocimetry (PIV), particle tracking velocimetry (PTV), radar and satellite images, MRI, CT, X-ray, two-photon microscopy, satellite, synthetic/simulated data, scattered unstructured opportunistic data, etc. We also have to facilitate multi-fidelity data that not only include multi-resolution data but also data corresponding to different levels of physical complexity, and hence training of NN requires special multi-fidelity methods such as in [35–37]. Noisy data are omnipresent and effective NN training and inference should be stable to noise and provide answers with uncertainty quantification using, e.g., Bayesian NN as in [36,38–40].

3. Operator regression networks

We first introduce the two neural operators, namely the deep operator network (DeepONet) [17,18] and the Fourier neural operator (FNO) [22], and subsequently we propose several extensions of these two methods.

3.1. DeepONet

Next, we provide an introduction to the vanilla[2] DeepONet. Although vanilla DeepONet has demonstrated good performance in diverse applications [18,41–48], here, we propose several extensions of DeepONet to achieve better accuracy and faster training.

3.1.1. Vanilla DeepONet

Four slightly different versions of DeepONet have been developed in Ref. [18], but in this study we use the stacked DeepONet with bias, which exhibits the best performance in practice among all our versions of DeepONet. First of all, to work with the input function numerically, we need to discretize the input function v and represent v in a finite-dimensional space (Fig. 1A). We could evaluate v at a set of locations $\{x_1, x_2, \ldots, x_m\}$, i.e., the pointwise evaluations of $[v(x_1), v(x_2), \ldots, v(x_m)]$. Many alternative representations could be applied. For example, we could project v to a set of basis functions and an approximation of v is then characterized/parameterized by the coefficients.

In this work, we choose the evaluation approach to discretize v. One advantage of DeepONet is that we do not discretize the output function u because of the following network design. A DeepONet has two sub-networks: a

[2] In this work, "vanilla" means the standard or unmodified version.

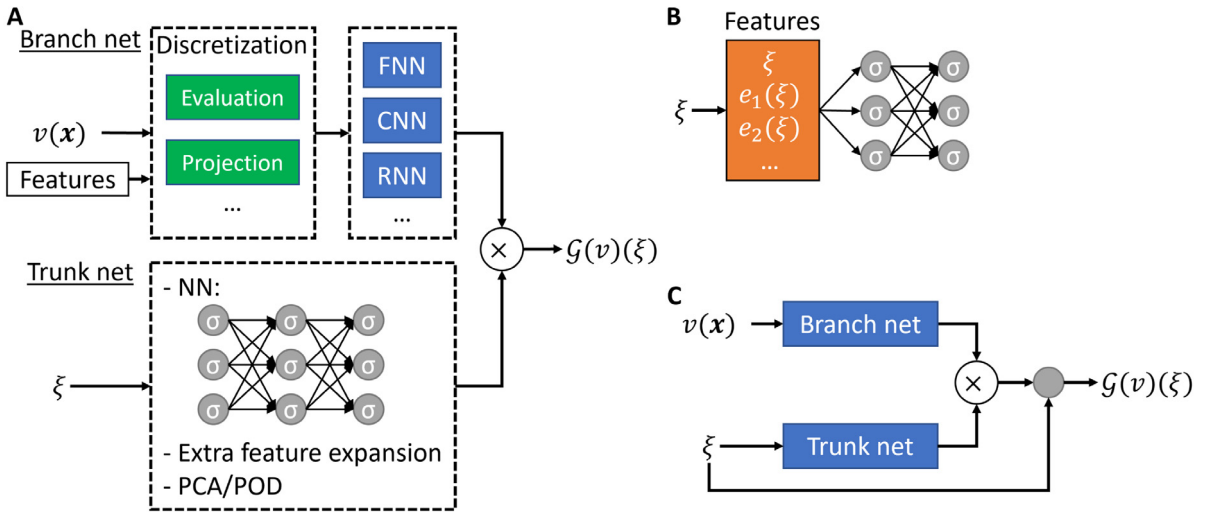

Fig. 1. Architecture of DeepONet. (**A**) DeepONet architecture. If the trunk net is a feed-forward neural network, then it is a vanilla DeepONet. (**B**) Feature expansion of the trunk-net input. Periodic BCs can also be strictly imposed into the DeepONet by using the Fourier feature expansion. (**C**) Dirichlet BCs are strictly enforced in DeepONet by modifying the network output.

"trunk" network and a "branch" network (Fig. 1A). The trunk net takes the coordinates $\xi \in D'$ as the input, while the branch net takes the discretized function v as input. Finally, the output of the network is expressed as:

$$\mathcal{G}(v)(\xi) = \sum_{k=1}^{p} b_k(v) t_k(\xi) + b_0,$$

where $b_0 \in \mathbb{R}$ is a bias. $\{b_1, b_2, \dots, b_p\}$ are the p outputs of the branch net, and $\{t_1, t_2, \dots, t_p\}$ are the p outputs of the trunk net.

We note that DeepONet is a high-level framework without restricting the branch net and the trunk net to any specific architecture. As ξ is usually low dimensional, a standard FNN is commonly used as the trunk net. The choice of the branch net depends on the structure of the input function v, and it can be chosen as a FNN, ResNet, CNN, RNN, or a graph neural network (GNN), etc. For example, if the discretization of v is on an equispaced 2D grid, then a CNN can be used; if the discretization of v is on an unstructured mesh, then a GNN can be used.

3.1.2. Feature expansion

In many applications, we not only have a dataset, but also have some prior knowledge about the underlying system. For example, we may know that the output functions have an oscillating nature or the functions have a fast decay. It is then beneficial to encode the knowledge directly into DeepONet by modifying the DeepONet architecture. The specific way to encode the knowledge is problem dependent, and here we introduce two ways of feature expansion either in the trunk net or in the branch net.

Feature expansion in the trunk net. If we know some features of the output function $u(\xi)$, then we can construct a feature expansion $(e_1(\xi), e_2(\xi), \dots)$ for the trunk net input (Fig. 1B), which was originally proposed and used in PINNs [9,49]. For example, as proposed in [43], for the problem with oscillating solutions, we can first perform a harmonic feature expansion on the input ξ of the trunk network as

$$\xi \mapsto (\xi, \cos(\xi), \sin(\xi), \cos(2\xi), \sin(2\xi), \dots),$$

which then becomes input to the trunk net. In this example, the features have analytical formulas. In general, we may not have the analytical formulas of features, and we can also use numerical values of the features as the extra input of the trunk net. For example, in dynamical systems, we can use some historical data as the feature. We note that here we usually still keep ξ as a feature.

Feature expansion in the branch net. If the feature is a function of x, then we cannot encode it in the trunk net, and instead we can use the feature as an extra input function of the branch net (Fig. 1A).

3.1.3. Hard-constraint boundary conditions

In some applications, we may know the boundary conditions of the system. Here, we introduce how to enforce the Dirichlet BCs and Periodic BCs as hard constraints in DeepONet.

Dirichlet BCs. Enforcing Dirichlet BCs in neural networks has been widely used in PINNs [9]. Let us consider a Dirichlet BC for the DeepONet output:

$$\mathcal{G}(v)(\xi) = g(\xi), \quad \xi \in \Gamma_D,$$

where Γ_D is a part of the boundary. To make the DeepONet output satisfy this BC automatically, we construct the solution as

$$\mathcal{G}(v)(\xi) = g(\xi) + \ell(\xi)\mathcal{N}(v)(\xi),$$

where $\mathcal{N}(v)(\xi)$ is the product of the branch and trunk nets, and $\ell(\xi)$ is a function satisfying the following condition:

$$\begin{cases} \ell(\xi) = 0, & \xi \in \Gamma_D, \\ \ell(\xi) > 0, & \text{otherwise.} \end{cases}$$

Here, we assume $g(\xi)$ is well defined for any ξ, otherwise we can construct a continuous extension for g.

Periodic BCs. Here, we first introduce how to enforce periodic BCs in neural networks in 1D [9,50] by constructing special feature expansion as discussed in Section 3.1.2, and then discuss how to extend it to 2D. If the solution $u(\xi)$ is periodic with respect to ξ of the period P, then $u(\xi)$ can be represented well by the Fourier series. Hence, we can replace the network input ξ with Fourier basis functions, i.e., the features in Fig. 1B are

$$\{1, \cos(\omega\xi), \sin(\omega\xi), \cos(2\omega\xi), \sin(2\omega\xi), \ldots\}$$

with $\omega = \frac{2\pi}{P}$. Compared to the aforementioned feature expansion, here we do not have the feature ξ any more. Because each Fourier basis function is periodic, it is easy to prove that the DeepONet output $\mathcal{G}(v)(\xi)$ is also periodic [50]. The number of Fourier features to be used is problem dependent, and we may use as few as the first two Fourier basis function $\{\cos(\omega\xi), \sin(\omega\xi)\}$.

Next, we discuss the 2D case, and with a slight abuse of the notation for x we denote $\xi = (x, y) \in \mathbb{R}^2$. The basis functions of the Fourier series on a 2D square are:

$$\left\{\cos(n\omega_x x)\cos\left(m\omega_y y\right), \cos(n\omega_x x)\sin\left(m\omega_y y\right), \sin(n\omega_x x)\cos\left(m\omega_y y\right), \sin(n\omega_x x)\sin\left(m\omega_y y\right)\right\}_{m,n=0,1,2,\ldots}$$

with $\omega_x = \frac{2\pi}{P_x}$ and $\omega_y = \frac{2\pi}{P_y}$. If we only choose $m, n \in \{0, 1\}$, the basis are:

$$\begin{aligned} &\{1, \cos(\omega_x x), \sin(\omega_x x), \cos(\omega_y y), \sin(\omega_y y), \\ &\quad \cos(\omega_x x)\cos(\omega_y y), \cos(\omega_x x)\sin(\omega_y y), \sin(\omega_x x)\cos(\omega_y y), \sin(\omega_x x)\sin(\omega_y y)\} \end{aligned}$$

or equivalently (due to trigonometric identities):

$$\left\{1, \cos(\omega_x x - \phi), \cos\left(\omega_y y - \phi\right), \cos\left(\omega_x x \pm \omega_y y - \phi\right) : \phi \in \{0, \frac{\pi}{2}\}\right\}.$$

Here, ϕ is the phase offset and has two values (0 and $\frac{\pi}{2}$). However, these phase offsets may not be optimal, and thus instead of using fixed phase offsets, we can use multiple phase offsets and set them as trainable parameters:

$$\begin{aligned} &\{1, \cos\left(\omega_x x - \phi_{1,1}\right), \ldots, \cos\left(\omega_x x - \phi_{1,m}\right), \cos\left(\omega_y y - \phi_{2,1}\right), \ldots, \cos(\omega_y y - \phi_{2,m}), \\ &\quad \cos(\omega_x x + \omega_y y - \phi_{3,1}), \ldots, \cos(\omega_x x + \omega_y y - \phi_{3,m}), \cos(\omega_x x - \omega_y y - \phi_{4,1}), \ldots, \cos(\omega_x x - \omega_y y - \phi_{4,m})\}, \end{aligned}$$

where $\phi_{i,j}$ $(i = 1, \ldots, 4$ and $j = 1, \ldots, m)$ are trainable.

3.1.4. POD-DeepONet: Precomputed POD basis as the trunk net

The vanilla DeepONet uses the trunk net to automatically learn the basis of the output function from the data. Here, we propose the POD-DeepONet, where we compute the basis by performing proper orthogonal decomposition (POD) on the training data (after first removing the mean). Then, we use this POD basis as the trunk net and only use neural networks for the branch net to learn the coefficients of the POD basis (Fig. 1A). The output can be written as:

$$\mathcal{G}(v)(\xi) = \sum_{k=1}^{p} b_k(v)\phi_k(\xi) + \phi_0(\xi),$$

where $\phi_0(\xi)$ is the mean function of $u(\xi)$ computed from the training dataset. $\{b_1, b_2, \dots, b_p\}$ are the p outputs of the branch net, and $\{\phi_1, \phi_2, \dots, \phi_p\}$ are the p precomputed POD modes of $u(\xi)$. The proposed POD-DeepONet shares a similar idea of using POD to represent functions as in [29].

3.1.5. Rescaling DeepONet with second moment analysis

It has been shown that it could be difficult to train neural networks if the variance of the output of each layer vanishes or explodes [51,52]. To facilitate network training, it is beneficial to have unit variance after the random initialization of neural networks. Based on this idea, different initialization methods have been developed for different activation functions, e.g., the Glorot initialization for tanh [51] and the He initialization for ReLU [53].

In DeepONet, the branch and trunk nets are two independent networks, and thus existing methods can be applied directly to maintain its variance. However, the DeepONet output is a product of the outputs of branch net and trunk net, which is not guaranteed to have unit variance. Hence, it is important to rescale the DeepONet output by its standard deviation as:

$$\mathcal{G}(v)(\xi) = \frac{1}{\sqrt{\mathrm{Var}[\sum_{k=1}^{p} b_k(v)t_k(\xi)]}}\left[\sum_{k=1}^{p} b_k(v)t_k(\xi) + b_0\right].$$

Next, we analyze the variance $\mathrm{Var}[\sum_{k=1}^{p} b_k(v)t_k(\xi)]$. We assume that both the branch net and the trunk net are standard fully-connected neural networks:

$$b_k(v(x_1), v(x_2), \dots, v(x_m)) = w^b_{k,L_b}\sigma_1(b^{L_b-1}(v(x_1), v(x_2), \dots, v(x_m))), \qquad (w^b_{k,L_b})^\top, b^{L_b-1} \in \mathbb{R}^{n^b_{k,L_b}},$$
$$t_k = \sigma_2(w^t_{k,L_t-1}t^{L_t-1}(x) + \theta^t_k), \qquad (w^t_{k,L_t-1})^\top, t^{L_t-1} \in \mathbb{R}^{n^t_{k,L_t-1}},$$

where b^{L_b-1} and t^{L_t-1} are the outputs of the last hidden layer of the branch net and the trunk net, respectively, and they are fully-connected ReLU networks.

We consider the ReLU activation function in both the branch and trunk nets, and thus use the He initialization [53]. We assume that all the biases are initialized to 0 and all the weights have the Gaussian distribution with mean 0 while all the random variables are independent and the weights of the same layer have the same variance. Then, at the initialization step we have [53]:

$$\mathbb{E}[b_k] = 0, \quad \mathbb{E}[b_k^2] = \mathbb{E}[(w^b_{k,L_b})^2]\mathbb{E}[\sigma_1^2(b^{L_b-1})] = \frac{1}{2}n^b_{k,L_b}\mathrm{Var}[(w^b_{k,L_b})_1]\mathrm{Var}[(b^{L_b-1})_1],$$
$$\mathbb{E}[t_k^2] = \frac{1}{2}\mathbb{E}[(w^t_{k,L_t-1}t^{L_t-1})^2] = \frac{1}{4}n^t_{k,L_t-1}\mathrm{Var}[(w^t_{k,L_t-1})_1]\mathrm{Var}[(t^{L_t-1})_1].$$

As in the He initialization, we initialize the weights such that $\mathrm{Var}[b_1^{(L_b-1)}] = 1/2$ and $\mathrm{Var}[(t^{L_t-1})_1] = 1/2$. Taking $\frac{1}{2}n^b_{k,L_b}\mathrm{Var}[(w^b_{k,L_b})_1] = 1$ and $\frac{1}{2}n^t_{k,L_t-1}\mathrm{Var}[(w^t_{k,L_t-1})_1] = 1$, we then have

$$\mathrm{Var}[b_k] = \frac{1}{2}, \quad \mathbb{E}[t_k^2] = \frac{1}{4}.$$

Observe that b_k and t_k are independent and that $\mathbb{E}[b_k] = 0$ and $\mathbb{E}[b_k t_k] = 0$. Moreover, $\mathbb{E}[b_i b_j] = 1/2$ if $i = j$ and $\mathbb{E}[b_i b_j] = 0$ if $i \neq j$. Then

$$\mathrm{Var}[\sum_{k=1}^{p} b_k t_k] = \mathbb{E}[(\sum_{k=1}^{p} b_k t_k)^2] = \mathbb{E}[\sum_{k,l=1}^{p} b_k b_l t_k t_l] = \sum_{k=1}^{p}\mathbb{E}[b_k^2]\mathbb{E}[t_k^2] = \sum_{k=1}^{p}\mathrm{Var}[b_k]\mathbb{E}[t_k^2] = \frac{p}{8}.$$

The analysis suggests that the scaling factor of the DeepONet should be $\mathcal{O}(1/\sqrt{p})$, i.e.,

$$\mathcal{G}(v)(\xi) = \frac{1}{\sqrt{p}}\left[\sum_{k=1}^{p} b_k(v)t_k(\xi) + b_0\right].$$

However, we note that the analysis only applies to the He initialization and considers the initialization stage, and thus in practice the scaling factor $1/\sqrt{p}$ may not be optimal for the final network accuracy.

The analysis is only suggestive, so we tested computationally the vanilla DeepONet and POD-DeepONet with a few different scaling factors, including no rescaling, $1/\sqrt{p}$, $1/p$, and $1/p^{1.5}$. From our experiments, we find that the scaling factor does not have a significant effect on the accuracy of vanilla DeepONet, but it is important to POD-DeepONet. In some of our experiments with POD-DeepONet, e.g., the Navier–Stokes equation in Section 5.6, among all scaling factors, we indeed obtain the best accuracy when it is scaled by $1/\sqrt{p}$ as suggested by our analysis. In some other experiments, e.g., the Burgers' equation in Section 5.1 and the advection problem (Case I) in Section 5.4.1, POD-DeepONet scaled by the factor $1/\sqrt{p}$ obtains a better accuracy than POD-DeepONet without rescaling, but it achieves an even better accuracy when scaled by the factor $1/p$. Therefore, the optimal scaling factor in practice is problem-dependent, and we use $1/p$ by default in our experiments, unless otherwise stated.

3.1.6. Multiple outputs

We have discussed the DeepONet for a single output function, but this can be extended to multiple output functions. Let us assume that we have n output functions. Here we propose a few possible approaches:

1. The simplest approach is that we can directly use n independent DeepONets, and each DeepONet outputs only one function.
2. The second approach is that we can split the outputs of both the branch net and the trunk net into n groups, and then the kth group outputs the kth solution. For example, if $n = 2$ and both the branch and trunk nets have 100 output neurons, then the dot product between the first 50 neurons of the branch and trunk nets generates the first function, and the remaining 50 neurons generate the second function.
3. The third approach is similar to the second approach, but we only split the branch net and share the trunk net. For example, for $n = 2$, we split the 100 output neurons of the branch net into 2 groups, but the trunk net only has 50 output neurons. Then, we use the first group of the branch net and the entire trunk net to generate the first output, and use the second group and the trunk net to generate the second output.
4. Similarly, we can also split the trunk net and share the branch net.

It is not easy to determine a priori which approach would work best, as this may be highly problem dependent. In this work, we use the second approach. More investigation and comparison between these four approaches both theoretically and computationally should be done in the future.

3.1.7. Fast algorithms and code implementation

For different types of datasets (depending on how the data is generated), we can use a different code implementation of DeepONet. The original implementation proposed in Ref. [18] works for all cases, but in some cases, we can implement DeepONet in a special way to greatly reduce the computational cost and memory usage by orders of magnitudes. We consider the following three cases of dataset types and the corresponding implementation:

1. **Case I:** Different output u has different number and different locations of ξ, see an example in Fig. 2 left. To train the DeepONet, we choose each data point as a triplet $(v, \xi, u(\xi))$, which is the original implementation in Ref. [18]. This implementation can be used for any problem setup, but it is expensive for the following Case II and Case III.
2. **Case II:** The number of trunk net input ξ is the same for all u (the locations of ξ can be different for different u), see an example in Fig. 2 middle. Then in the triplet $(v, \xi, u(\xi))$, for the same u, we can reuse the same input v of the branch net for all the corresponding ξ, and the output of the branch net can be stored and reused for these ξ. Hence, compared to the original implementation, we remove the computational redundancy in the branch net for different v.

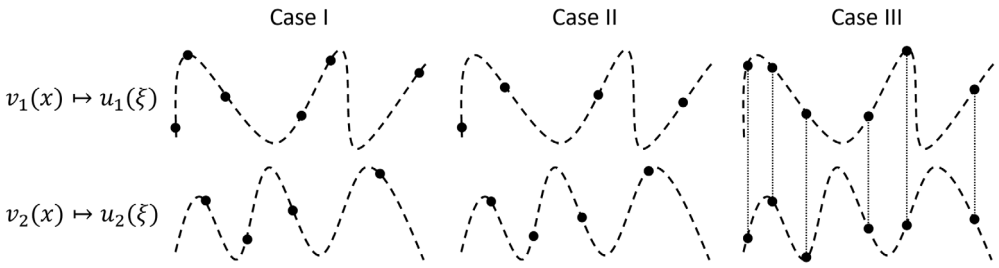

Fig. 2. Examples of three different types of datasets. u_1 and u_2 are two examples of the output function in the dataset. The black dots are the locations of measurements. (Case I) u_1 has measurements in 6 locations of ξ, and u_1 has measurements in 4 different locations. (Case II) Both u_1 and u_2 have 4 measurements, but the locations are different. (Case III) Both u_1 and u_2 have 6 measurements in the same locations.

Table 1
Comparison of three different types of datasets and code implementation.

	Case I	Case II	Case III
Do all u have the same number of locations of ξ?	No	Yes	Yes
Do all u have the same locations of ξ?	No	No	Yes
Flexibility	High	Medium	Low
Computational cost	High	Medium	Low

3. Another similar case as Case II is that all ξ are sampled from the points in the same grid, and each u uses only a portion of the grid points (the number of ξ may be different). Then, we can reuse the same input ξ of the trunk net and remove the computational redundancy.
4. **Case III:** The trunk net input ξ is exactly the same for all u, see an example in Fig. 2 right. Based on Case II, we can further reuse the same trunk net input ξ for all v. In this way, we remove the computational redundancy in both branch and trunk nets.

The comparison of the three different cases is summarized in Table 1. Here, we only introduce the underlying idea, but for the detailed code implementation, we refer the reader to our code.

DeepONet can work for the datasets in all the cases, but FNO only works for the dataset in Case III. Hence, in order to compare DeepONet and FNO, in this study we generate the data as in Case III. We also use the DeepONet implementation discussed in Case III, instead of the original implementation, to achieve comparable computational cost of FNO. We note that compared to Case III, Case I and Case II may lead to better accuracy of DeepONet.

3.2. FNO

Next, we introduce the vanilla FNO and then discuss how to generalize the vanilla FNO for problems whose inputs and outputs are defined on different domains or on a complex geometry.

3.2.1. Vanilla FNO

FNO is formulated by parameterizing the integral kernel directly in Fourier space. The main network parameters are defined and learned in the Fourier space rather than the physical space, i.e., the coefficients of the Fourier series of the output function are learned from the data.

Different from DeepONet, FNO discretizes both the input function $v(x)$ and the output function $u(\xi)$ by using pointwise evaluations in an equispaced mesh. Also, FNO requires that v and u are defined on the domain (i.e., $D = D'$, and x and ξ become the same variable), and the same discretization (i.e., the same mesh) is used for v and u. In short, FNO maps the discretization of v in an equispaced mesh to the discretization of u on the same mesh. For example, if D is a rectangular domain in 2D, then FNO input v is a 2D matrix (i.e., an image). When we present the architecture of FNO, we will discuss both the function value on one mesh location and also the function values on all mesh locations, so we use the following convention for a function f: $f(x)$ means the value of $f(x)$ in a mesh location x, and the bold font $\boldsymbol{f}$ means the values of $f(x)$ in all mesh locations.

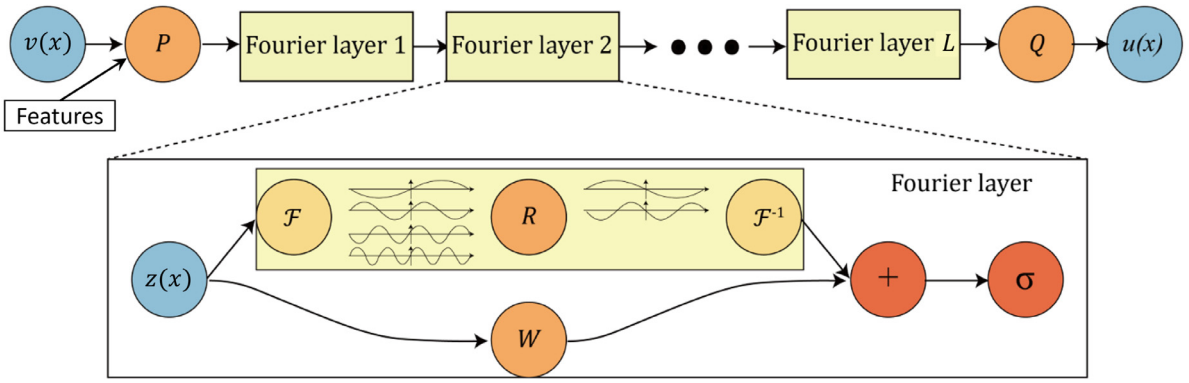

Fig. 3. Architecture of FNO with extra features added to the input.
Source: Figure is adapted from [22].

In FNO, for any location x on the mesh, the function value $v(x)$ is first lifted to a higher dimensional representation $z_0(x)$ by

$$z_0(x) = P(v(x)) \in \mathbb{R}^{d_z}$$

using a local transformation $P : \mathbb{R} \to \mathbb{R}^{d_z}$ (Fig. 3), which is parameterized by a shallow fully-connected neural network or simply a linear layer. We note that z_0 is defined on the same mesh as $\boldsymbol{v}$, and the values of z_0 in the mesh can be viewed as an image with d_z channels. Then L Fourier layers are applied iteratively to z_0. Let us denote z_L as the output of the last Fourier layer, and the dimension of $z_L(x)$ is also d_z. Hence, at the end, another local transformation $Q : \mathbb{R}^{d_z} \to \mathbb{R}$ is applied to project $z_L(x)$ to the output (Fig. 3) by

$$u(x) = Q(z_L(x)).$$

We parameterize Q by a fully-connected neural network.

Next we introduce the Fourier layer by using the Fast Fourier Transform (FFT). For the output of the lth Fourier layer z_l with d_v channels, we first compute the following transform by FFT $\mathcal{F}$ and inverse FFT $\mathcal{F}^{-1}$ (the top path of the Fourier layer in Fig. 3):

$$\mathcal{F}^{-1}\left(R_l \cdot \mathcal{F}(z_l)\right).$$

The details are as follows:

- $\mathcal{F}$ is applied to each channel of z_l separately, and we usually truncate the higher modes of $\mathcal{F}(z_l)$, keeping only the first k Fourier modes in each channel. So $\mathcal{F}(z_l)$ has the shape $d_v \times k$. For 2D functions, $k = k_1 \times k_2$, where k_1 and k_2 are the number of modes to keep in the first and second dimensions, respectively. For 3D functions, it can be done similarly.
- We apply a different (complex-number) weight matrix of shape $d_v \times d_v$ for each mode index of $\mathcal{F}(z_l)$, so we have k trainable matrices, which form a weight tensor $R_l \in \mathbb{C}^{d_v \times d_v \times k}$. Then, $R_l \cdot \mathcal{F}(z_l)$ has the same shape of $d_v \times k$ as $\mathcal{F}(z_l)$.
- Before we perform the inverse FFT, we need to append zeros to $R_l \cdot \mathcal{F}(z_l)$ to fill in the truncated modes.

Moreover, in each Fourier layer, a residual connection with a weight matrix $W_l \in \mathbb{R}^{d_v \times d_v}$ is used to compute a new set of d_v channels, and each new channel is a linear combinations between all the z_l channels (the bottom path of the Fourier layer in Fig. 3). $W_l \cdot z_l$ has the same shape as z_l. We can implement $W_l \cdot z_l$ by a matrix multiplication or by a convolution layer with the kernel size 1. Then, the output of the $(l+1)$th Fourier layer z_{l+1} is

$$z_{l+1} = \sigma\left(\mathcal{F}^{-1}\left(R_l \cdot \mathcal{F}(z_l)\right) + W_l \cdot z_l + \boldsymbol{b}_l\right),$$

where σ is a nonlinear activation function, and $\boldsymbol{b}_l \in \mathbb{R}^{d_v}$ is a bias. In this work, we use ReLU in all cases.

We note that in practice, to achieve good accuracy, we need to use the coordinates x as the input as well, in addition to $v(x)$, i.e., the FNO input is the values of $(x, v(x)) \in \mathbb{R}^{d+1}$ in a grid (i.e., an image with $d+1$ channels). This is a special case of the feature expansion that we will discuss in Section 3.2.4.

3.2.2. dFNO+: Operators with inputs and outputs defined on different domains

One limitation of FNO is that it requires D and D' to be the same domain, which is not always satisfied. Here, we discuss two scenarios where $D \neq D'$, and propose new extensions of FNO to address this issue.

Case I: The output space is a product space of the input space and another space D_0, i.e., $D' = D \times D_0$. We use a specific example to illustrate the idea. We consider the PDE solution operator mapping from the initial condition to the solution in the whole domain:

$$\mathcal{G} : v(x) = u(x, 0) \mapsto u(x, t),$$

where $x \in [0, 1]$ and $t \in [0, T]$. Here, $D = [0, 1]$, $D' = [0, 1] \times [0, T]$, and thus $D_0 = [0, T]$. In order to match the input and output domains, we propose the following two methods.

Method 1: Expand the input domain. We can extend the input function by adding the extra coordinate t, defining $\tilde{v}$ as

$$\tilde{v}(x, t) = v(x).$$

Then, FNO is used to learn the mapping from $\tilde{v}(x, t)$ to $u(x, t)$.

Method 2 [22]**: Shrink the output domain via RNN.** We can also reduce the dimension of the output by decomposing $\mathcal{G}$ into a series of operators. We denote a new time-marching operator

$$\tilde{\mathcal{G}} : u(x, t) \mapsto u(x, t + \Delta t), \quad x \in D,$$

i.e., $\tilde{\mathcal{G}}$ predicts the solution at $t + \Delta t$ from the solution at t. Then, we apply $\tilde{\mathcal{G}}$ to the input $v(x)$ repeatedly to obtain the solution in the whole domain, which is similar to a RNN.

Case II: The input space is a subset of the output space, i.e., $D \subset D'$. In general, we can always extend v from D to D' by padding zeros in the domain $D' \setminus D$, i.e., we define

$$\tilde{v}(\xi) = \begin{cases} v(\xi), & \text{if } \xi \in D \\ 0, & \text{if } \xi \in D' \setminus D, \end{cases}$$

and then learn the mapping from $\tilde{v}$ to u.

However, in some cases, this padding strategy may not be efficient. For example, we consider a PDE defined on a rectangular domain $(x, y) \in D' = [0, 1]^2$, and the operator is the mapping from the Dirichlet boundary condition v defined in the four boundaries ($D = \{0, 1\} \times [0, 1] \cup [0, 1] \times \{0, 1\}$) to the solution $u(x, y)$ inside the rectangular domain. In this example, D is essentially a 1D space and occupies zero area in D'. We propose a better strategy so that we first unfold the curve of v into a 1D function $\tilde{v}$ defined in $[0, 4]$:

$$\tilde{v}(\tilde{x}) = \begin{cases} v(\tilde{x}, 0), & \text{if } \tilde{x} \in [0, 1] \text{ (bottom boundary)} \\ v(1, \tilde{x} - 1), & \text{if } \tilde{x} \in [1, 2] \text{ (right boundary)} \\ v(3 - \tilde{x}, 1), & \text{if } \tilde{x} \in [2, 3] \text{ (top boundary)} \\ v(0, 4 - \tilde{x}), & \text{if } \tilde{x} \in [3, 4] \text{ (left boundary)} \end{cases},$$

and then, we use the method in Case I above to learn the operator from $\tilde{v}$ in 1D to u in 2D.

3.2.3. gFNO+: Operators with inputs and outputs defined on a complex geometry

FNO uses FFT, which requires the input and output functions to be defined on a Cartesian domain with a lattice grid mesh. However, for the PDEs defined on a complex geometry D (e.g., L-shape, triangular domain, etc.), an unstructured mesh is usually used, and thus we need to deal with two issues: (1) non-Cartesian domain, and (2) non-lattice mesh. For the second issue of unstructured mesh, we need to do interpolation between the unstructured mesh and a lattice grid mesh.

For the issue of the Cartesian domain, we first define the Cartesian domain $\tilde{D}$, which is the minimum bounding box of D, and then extend v (the same for u) from D to $\tilde{D}$ by

$$\tilde{v}(x) = \begin{cases} v(x), & \text{if } x \in D \\ v_0(x), & \text{if } x \in \tilde{D} \setminus D. \end{cases}$$

Table 2
Comparison between vanilla DeepONet and vanilla FNO.

	DeepONet	FNO
Input domain D' & Output domain D'	Arbitrary	Cuboid, $D = D'$
Discretization of output function u	No	Yes
Mesh	Arbitrary	Grid
Prediction location	Arbitrary	Grid points
Full field observation data	No	Yes
Discontinuous functions	Good	Questionable

Here, the choice of $v_0(x)$ is not unique. The simplest choice is $v_0(x) = 0$, i.e., zero padding. However, we find that such zero padding leads to large error of FNO, which may be due to the discontinuity from $v(x)$ to $v_0(x)$. We propose to compute $v_0(x)$ by "nearest neighbor":

$$\text{for } x \in \tilde{D} \setminus D, \quad v_0(x) = v(x_0), \quad \text{where } x_0 = \min_{p \in D} \|p - x\|,$$

so that $\tilde{v}(x)$ is continuous on the boundary of D. In the training, we use a mask to only consider the points inside D in the loss function.

3.2.4. Feature expansion

In FNO, the features can be applied by using them as extra network inputs. As we mentioned at the end of Section 3.2.1, in practice we need to use the coordinates x as the input, which is a special case of feature expansion. If we have a feature $f(x)$, then the FNO input is the values of $(x, v(x), f(x)) \in \mathbb{R}^{d+2}$, i.e., an image with $d + 2$ channels.

3.3. Comparison between DeepONet and FNO

Here, we list a comparison between DeepONet and FNO in Table 2 in terms of some of their properties instead of accuracy. The first three points have been discussed in the introduction above of DeepONet and FNO. Because FNO also discretizes the output function, then after the network training, it can only predict the solution in the same mesh as the input function, whereas DeepONet can make predictions at any location. For the training, FNO requires a full field observation data, but DeepONet is more flexible, except that POD-DeepONet requires a full field data to compute the POD modes. We also note that FNO relies on Fourier transformation, which may not be very accurate for discontinuous functions, see an example in Section 5.4.3.

3.4. Other technical details

There are several other useful techniques that may improve the performance of DeepONet and FNO, such as learning rate decay, L^2 regularization, and input normalization. Here, we emphasize the output normalization. Let us assume that in the training dataset the mean function and the standard deviation function of u is $\bar{u}(\xi)$ and $\text{std}[u](\xi)$, respectively. Then, we construct the surrogate model as

$$u(\xi) = \text{std}[u](\xi) \cdot \mathcal{N}(v)(\xi) + \bar{u}(\xi),$$

where $\mathcal{N}$ is a DeepONet or FNO. Hence, for any ξ, the mean value of $\mathcal{N}(v)(\xi)$ is zero and the standard deviation is one.

4. Theoretical comparison

In this section, we compare the universal approximation theorem of operators using DeepONet and FNO. We also compare their error estimates for the solution operator from the Burgers' equation. Here, we first present the general universal approximation theorem for DeepONet and FNO for completeness. Also, we emulate the same numerical method for DeepONet and FNO, while the comparison in [23] between two methods is abstract instead of specific equations such as the Burgers' equations we consider here.

4.1. Universal approximation theorem for operators

Theorem 4.1 (*Generalized universal approximation theorem for operators [18]*). *Suppose that X is a Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$ Assume that $\mathcal{G} : V \to C(K_2)$ is a nonlinear continuous operator. Then for any $\epsilon > 0$, there exist positive integers m, p, branch nets $b_k : V \to \mathbb{R}$, and trunk nets $t_k : \mathbb{R}^d \to \mathbb{R}$, and $x_1, x_2, \ldots, x_m \in K_1$, such that*

$$\sup_{v\in V}\sup_{x\in K_2}\left|\mathcal{G}(v)(x) - \sum_{k=1}^{p}\underbrace{b_k(v(x_1), v(x_2), \ldots, v(x_m))}_{branch}\underbrace{t_k(x)}_{trunk}\right| < \epsilon.$$

Furthermore, the functions b_k and t_k can be chosen as diverse classes of neural networks, satisfying the classical universal approximation theorem of functions, e.g., fully-connected neural networks, residual neural networks, and convolutional neural networks.

This theorem was proved in [16] with two-layer neural networks. Also, the theorem holds when the Banach space $C(K_1)$ is replaced by $L^q(K_1)$ and $C(K_2)$ replaced by $L^r(K_2)$, $q, r \geq 1$. Some extensions have been made in [24] for measurable operators, which contains discontinuous operators that can be approximated by continuous operators. The conclusion can be readily extended to the case of vector (multiple) output functions.

The FNO for operator regression is proposed in [22], and the corresponding universal approximation theorem is presented in [23]. We present the theorem for completeness. Define $\mathbb{T}^d = [0, 2\pi]^d$ and the Hilbert space with smooth index s by $H^s\left(\mathbb{T}^d; \mathbb{R}^{d_v}\right)$ for $\mathbb{R}^{d_v}$-valued functions defined on $\mathbb{T}^d$. Let $\mathcal{V}(\mathbb{T}^d; \mathbb{R}^{d_v})$ be a Banach space of $\mathbb{R}^{d_v}$-valued functions defined on $\mathbb{T}^d$. Let P be a lifting operator from $\mathcal{V}(\mathbb{T}^d; \mathbb{R}^{d_v}) \to \mathcal{U}(\mathbb{T}^d; \mathbb{R}^{d_a})$ and the projection $Q : \mathcal{U}(\mathbb{T}^d; \mathbb{R}^{d_a}) \to \mathcal{U}(\mathbb{T}^d; \mathbb{R}^{d_u})$. Let $\mathcal{I}_N$ denote the Fourier interpolation operator, i.e., $\mathcal{I}_N f(x) = \sum_{i=1}^{N} f(x_i)L_i(x)$, where x_i's are the Fourier collocation points on $\mathbb{T}^d$ and $L_i(x)$ are corresponding Lagrange interpolation trigonometric polynomials. Therefore the output of the lth Fourier block is defined as

$$\mathcal{L}_l(z)(x) = \sigma\left(W_l z(x) + b_l(x) + \mathcal{F}^{-1}\Big(R_l(k)\cdot\mathcal{F}(z)(k)\Big)(x)\right).$$

Here, $W_l \in \mathbb{R}^{d_l\times d_l}$ and the function $b_l(x)$ is $\mathbb{R}^{d_l}$-valued, and the coefficients $R_l(k) \in \mathbb{R}^{d_l\times d_l}$ define a convolution operator via the Fourier transform. Then, the FNO in its continuous form is defined as follows:

$$\mathcal{F}(v) = Q \circ \mathcal{I}_N \circ \mathcal{L}_L \circ \mathcal{I}_N \circ \cdots \circ \mathcal{L}_1 \circ \mathcal{I}_N \circ P(v). \tag{4.1}$$

Theorem 4.2 (*Universal approximation theorem of FNO [23]*). *Let $s, s' \geq 0$ and $\Omega \subset \mathbb{T}^d$ be a domain with Lipschitz boundary. Let $\mathcal{G} : H^s\left(\Omega; \mathbb{R}^{d_v}\right) \to H^{s'}\left(\Omega; \mathbb{R}^{d_u}\right)$ be a continuous operator. Let $V \subset H^s\left(\Omega; \mathbb{R}^{d_v}\right)$ be a compact subset. Then for any $\epsilon > 0$, there exists a FNO of the form (4.1), $\mathcal{F} : H^s\left(\mathbb{T}^d; \mathbb{R}^{d_v}\right) \to H^{s'}\left(\mathbb{T}^d; \mathbb{R}^{d_u}\right)$ such that*

$$\sup_{v\in V}\|\mathcal{G}(v) - \mathcal{F}(v)\|_{H^{s'}} \leq \epsilon.$$

The conclusion above can be found in Theorem 2.15 of [23].

Both DeepONet and FNO suffer from **the curse of dimensionality** if one uses ReLU or tanh networks for Lipschitz continuous operators, due to the approximation capacity of these networks for high dimensional inputs $(v(x_1), v(x_2), \ldots, v(x_m))$. However, rates of convergence are obtained for [24,25,27] for DeepONet and [28] for FNO, for some solution operators from PDEs. Next, we compare error estimates of DeepONet and FNO for 1D Burgers equations with periodic boundary conditions. As will be shown, the inputs of the operator will be first approximated by some numerical methods such that one has possibly high-dimensional inputs for the neural network operators. This will introduce approximation errors in the inputs and thus in the outputs. Then extra errors will be induced by the network approximation emulating the analytical (after approximation of the input function) method of the solution.

4.2. 1D Burgers' equation

Consider the Burgers' equation with periodic boundary condition

$$\begin{cases} u_t + uu_x = \kappa u_{xx}, \quad (x,t) \in \mathbb{R}\times(0,\infty), \quad \kappa > 0, \\ u(x-\pi, t) = u(x+\pi, t), \quad u(x,0) = u_0(x). \end{cases} \tag{4.2}$$

Let $M_0,\ M_1 > 0$. Define

$$\mathcal{S} = \mathcal{S}(M_0, M_1) := \{v \in W^{1,\infty}(-\pi,\pi) : \|v\|_{L^\infty} \le M_0, \|\partial_x v\|_{L^\infty} \le M_1, \bar{v} := \int_{-\pi}^{\pi} v(s)\,ds = 0\}.$$

We consider the Fourier interpolation of u_0, $u(x,0) = u_0(x) = \sum_{j=0}^{m-1} u_{0,j} L_j(x)$, where $-\pi = x_0 < x_1 < \cdots < x_m = \pi$, $u_{0,j} = u_0(x_j)$, and $L_j(x)$ is the Lagrange interpolation trigonometric polynomials at $\mathbf{u}_{0,m} = (u_{0,0}, u_{0,1}, \ldots, u_{0,m-1})^\top$.

Define $x_j^l = x_j^0 + 2\pi l,\ j = 0, 1, \ldots, m$, for each $l \in \mathbb{Z}$. Then $\{x_j^l\}_{j=0}^m$ form a partition of $[-\pi + 2\pi l, \pi + 2\pi l)$. For simplicity, we denote $x_j = x_j^0$. To make sure that $v_0(x)$ in (4.3) is 2π-periodic, we require that the initial condition has zero mean in a period $\bar{u}_0 := \int_{-\pi}^{\pi} u_0(s)ds = 0$. Then, by the Cole–Hopf transformation, the solution to (4.2) can be written as

$$u = \frac{-2\kappa v_x}{v}, \quad \text{where} \quad \begin{cases} v_t = \kappa v_{xx}, \\ v(x,0) = v_0(x) = \exp\Big(-\frac{1}{2\kappa}\int_{-\pi}^{x} u_0(s)ds\Big). \end{cases} \tag{4.3}$$

Since $0 < \exp(-\frac{\pi\|u_0\|_\infty}{\kappa}) \le v_0(x) \le \exp(\frac{\pi\|u_0\|_\infty}{\kappa})$, the solution u can be written explicitly as

$$u(\mathbf{x}) = \mathcal{G}(u_0)(\mathbf{x}) := -2\kappa \frac{\int_{\mathbb{R}} \partial_x \mathcal{K}(x,y,t) v_0(y) dy}{\int_{\mathbb{R}} \mathcal{K}(x,y,t) v_0(y) dy}, \quad \mathbf{x} = (x,t), \tag{4.4}$$

where $\mathcal{K}(x,y,t) = \frac{1}{\sqrt{4\pi\kappa t}} \exp\Big(-\frac{(x-y)^2}{4\kappa t}\Big)$ is the heat kernel. It can be readily checked that $u(\mathbf{x})$ is a unique solution to (4.2).

We may obtain a neural network for operators of regression of $\mathcal{G}$ by first approximating it with classical numerical methods and then emulating these methods using neural networks. Let us first approximate $\mathcal{G}$. Define $\mathbf{V}_m := \mathbf{V}(\mathbf{u}_{0,m}) = (V_0, V_1, \ldots, V_{m-1})^\top$, where $V_0 = 1$ and $V_j = \exp(-\sum_k \int_{-\pi}^{x_j} L_k(y)\,dy \frac{u_0(y_k)}{2\kappa})$, $j = 1, \ldots, m-1$. Define $\mathcal{G}_m(\mathbf{u}_{0,m})(\mathbf{x}) = \Big(\widetilde{\mathcal{G}}_m \circ \mathbf{V}(\mathbf{u}_{0,m})\Big)(\mathbf{x})$ where

$$\mathcal{G}_m(\mathbf{u}_{0,m})(\mathbf{x}) = \frac{-2\kappa \int_{\mathbb{R}} \partial_x \mathcal{K}(x,y,t)(\mathcal{I}_m v_0)(y) dy}{\int_{\mathbb{R}} \mathcal{K}(x,y,t)(\mathcal{I}_m v_0)(y) dy} = \frac{V_0 c_0^1(\mathbf{x}) + V_1 c_1^1(\mathbf{x}) + \cdots + V_{m-1} c_{m-1}^1(\mathbf{x})}{V_0 c_0^2(\mathbf{x}) + V_1 c_1^2(\mathbf{x}) + \cdots + V_{m-1} c_{m-1}^2(\mathbf{x})}, \tag{4.5}$$

where $\widetilde{\mathcal{G}}_m$ is a rational function with respect to $\mathbf{V}_m$, with both the numerator and the denominator being mth degree m-variate polynomials with m terms and $\mathcal{I}_m$ is the Fourier interpolation operator and for $j = 0, \ldots, m-1$,

$$c_j^1(\mathbf{x}) = -2\kappa \int_0^x \Big(\sum_{l\in\mathbb{Z}} \partial_x \mathcal{K}(x, y + 2\pi l, t)\Big) L_j(y)\,dy, \quad c_j^2(\mathbf{x}) = \int_0^x \Big(\sum_{l\in\mathbb{Z}} \mathcal{K}(x, y + 2\pi l, t)\Big) L_j(y)\,dy.$$

By the Lipschitz continuity of the solution operator $\mathcal{G}(\cdot)$ of the Burgers equation, we have the following estimate.

Theorem 4.3. *Let $u_0 \in \mathcal{S}' = \big\{u_0|_{[-\pi,\pi]} \in \mathcal{S} : u_0$ grows at most quadratically at $\infty\big\}$. Let $\mathcal{G}(u_0)(\mathbf{x})$ and $\mathcal{G}_m(\mathbf{u}_{0,m})(\mathbf{x})$ be defined in (4.4) and (4.5), respectively. Suppose h is small enough. Then there is a uniform constant C depending only on t, the lower and upper bounds M_0 and M_1 and κ, such that for any $t \in (0, +\infty)$, we have*

$$\sup_{u_0\in\mathcal{S}'} \big\|\mathcal{G}(u_0)(\cdot,t) - \mathcal{G}_m(\mathbf{u}_{0,m})(\cdot,t)\big\|_{L^2(-\pi,\pi)} \le Ch, \ \text{where } h = 2\pi/m.$$

Proof. The proof can be found in Appendix A. □

Second, we emulate $\mathcal{G}_m$ by a neural network. To this end, it is important to realize that a rational function can be approximated by a ReLU network according to the following theorem.

Theorem 4.4 ([54]). *Let $\varepsilon \in (0, 1]$ and nonnegative integer k be given. Let $p : [0,1]^d \to [-1,1]$ and $q : [0,1]^d \to [2^{-k}, 1]$ be polynomials of degree $\le r$, each with $\le s$ monomials. Then there exists a ReLU network f of size (number of total neurons) $\mathcal{O}\Big(k^7 \ln(\frac{1}{\varepsilon})^3 + \min\{srk\ln(sr/\varepsilon),\ sdk^2\ln(dsr/\varepsilon)^2\}\Big)$ such that $\sup_{x\in[0,1]^d} \Big|f(x) - \frac{p(x)}{q(x)}\Big| \le \varepsilon$.*

Observing that $\mathcal{G}_m(\mathbf{u}_{0,m}) = \widetilde{\mathcal{G}}(\mathbf{V}_m)$ is a rational function with respect to $\mathbf{V}_m$ while $\mathbf{V}_m$ is an exponential function in $\mathbf{u}_{0,m}$. Then by approximation of rational polynomials in Theorem 4.4, there exists a ReLU network of size

Table 3
16 problems tested in this study.

Section	Problems
Section 5.1	Burgers' equation
Section 5.2	5 Darcy problems in a rectangular domain and complex geometries
Section 5.3	Multiphysics electroconvection problem
Section 5.4.1	3 Advection problems
Section 5.4.2	Linear instability waves in high-speed boundary layers
Section 5.4.3	Compressible Euler equation with non-equilibrium chemistry
Section 5.5	Predicting surface vorticity of a flapping airfoil
Section 5.6	Navier–Stokes equation in the vorticity–velocity form
Section 5.7	2 problems of regularized cavity flows

$\mathcal{O}(m^2 \ln(m))$ ($s = r = m$ in Theorem 4.4) to obtain accuracy of $\mathcal{O}(m^{-1})$ for each x. For the approximation of 'exp' by ReLU networks, we only need a network size of $\mathcal{O}(\ln(m))$ to obtain accuracy of $\mathcal{O}(m^{-1})$, according to [55]. Thus, we need a ReLU network of size $\mathcal{O}(m^3 \ln(m))$ to emulate $\mathcal{G}_m(\mathbf{u}_{0,m})$ and denote this ReLU network by $\mathcal{G}_m^{\mathbb{N}}(\mathbf{u}_{0,m})$.

The network $\mathcal{G}_m^{\mathbb{N}}(\mathbf{u}_{0,m})(x_j)$ can be viewed as FNO, where for all the kernels $R_l \equiv 0$. Thus, the FNO of size $\mathcal{O}(m^3 \ln(m))$ can achieve the accuracy of $\mathcal{O}(m^{-1})$. According to [25], the network $\mathcal{G}_m^{\mathbb{N}}(\mathbf{u}_{0,m})(x_j)$ serves as a branch net while only a ReLU network of size $\mathcal{O}(\ln m)$ is needed for the trunk net. The size of the DeepONet is thus $\mathcal{O}(m^3 \ln(m))$. In conclusion, we find that

- both FNO and DeepONet of size $\mathcal{O}(m^3 \ln(m))$ can achieve the accuracy of $\mathcal{O}(m^{-1})$.

There are many ways to approximate the solution operator and emulate the corresponding numerical methods. Thus, the estimation of sizes of networks may be not optimal[3]. However, we can always connect FNO $\mathcal{F}(v)$ and DeepONet by $\sum_{i=1}^{N} \mathcal{F}(v)(x_j) L_j(x)$, where $L_j(x)$ is a basis of interpolation type, i.e., $L_j(x_j) = 1$ and $L_j(x_i) = 0$ for all $i \neq j$. Depending on the underlying problems, the basis $L_j(x)$ can be trigonometric polynomials, piecewise or global algebraic polynomials, splines, neural networks, etc. *In other words, FNO can be thought of as a special form of branch network in DeepONet.*

5. Numerical results

We compare the performance of DeepONet and FNO on 16 different problems listed in Table 3. To evaluate the performance of the networks, we compute the L^2 relative error of the predictions, and for each case, five independent training trials are performed to compute the mean error and the standard deviation. The dataset sizes for each problem are listed in Table B.1, and the network sizes of DeepONets are listed in Tables C.1 and C.2. All the data and codes are available in GitHub at https://github.com/lu-group/deeponet-fno.

5.1. Burgers' Equation

Problem setup. We first consider the one-dimensional Burgers' equation:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, 1), \ t \in (0, 1],$$

with periodic boundary condition, where $\nu = 0.1$ is the viscosity. Here, we learn the operator mapping from the initial condition $u(x, 0) = u_0(x)$ to the solution $u(x, t)$ at $t = 1$, i.e.,

$$\mathcal{G} : u_0(x) \mapsto u(x, 1).$$

We use the dataset generated in Ref. [22], where the initial condition is sampled from a Gaussian random field with a Riesz kernel, denoted by $\mu = \mathcal{R}(0, 625(-\Delta + 25I)^{-2})$. Here, μ is the probability measure on the function space, and Δ and I represent the Laplacian and the identity, respectively. We use a spatial resolution with 128 grids to represent the input and output functions.

[3] To make a fair comparison between DeepONet and FNO, we always emulate the same numerical methods.

Table 4
L^2 relative error for the Burgers' equation in Section 5.1. Here, p is the number of outputs of the branch and trunk nets.

	Section 5.1 Burgers'
DeepONet w/o normalization	2.29 ± 0.10%
DeepONet w/ normalization	2.15 ± 0.09%
FNO w/o normalization	2.23 ± 0.04%
FNO w/ normalization	**1.93** ± **0.04**%
POD-DeepONet (w/o rescaling)	3.46 ± 0.06%
POD-DeepONet (rescaling by $1/\sqrt{p}$)	2.40 ± 0.06%
POD-DeepONet (rescaling by $1/p$)	**1.94** ± **0.07**%
POD-DeepONet (rescaling by $1/p^{1.5}$)	2.41 ± 0.04%

Results. As discussed in Section 3.1.3, in DeepONet, we impose the periodic boundary condition by applying four Fourier basis $\{\cos(2\pi x), \sin(2\pi x), \cos(4\pi x), \sin(4\pi x)\}$ to the input of the trunk net. By using the output normalization, the error of DeepONet decreases from 2.29 ± 0.10% to 2.15 ± 0.09% (Table 4), and FNO achieves an error of 1.93 ± 0.04% (Table 4). Hence, the output normalization is helpful to both DeepONet and FNO.

As we showed in Section 3.1.5, DeepONet has variance proportional to p instead of 1 when each branch and trunk net has variance 1 and thus a scale of $1/\sqrt{p}$ is needed. A different scaling is required for POD-DeepONet as the precomputed POD modes are in place of the trunk nets and are not networks to be trained. We tested different scaling factors, including $1/\sqrt{p}$, $1/p$ and $1/p^{1.5}$ (Table 4), and POD-DeepONet with the factor $1/p$ has the smallest error of 1.94 ± 0.07%, which is almost the same as the error of FNO with the output normalization. For the following problems all POD-deepONet, we use the scale $1/p$, unless otherwise stated.

5.2. Darcy problem

We consider two-dimensional Darcy flows in different geometries filled with porous media, which can be described by the following equation:

$$-\nabla \cdot (K(x, y)\nabla h(x, y)) = f,\ (x, y) \in \Omega, \tag{5.1}$$

where K is the permeability field, h is the pressure, and f is a source term which can be either a constant or a space-dependent function. Boundary conditions will be described in the problem setup below. Four different geometries are considered in the present study, including a rectangular domain in Section 5.2.1, a pentagram with a hole in Section 5.2.2, a triangular domain in Section 5.2.3, and a triangular domain with notch in Section 5.2.4. We generate the dataset by solving Eq. (5.1) using the MATLAB Partial Differential Equation Toolbox (for more details see Appendix E).

5.2.1. Darcy problem in a rectangular domain

Problem setup. The first example of Darcy flow is defined in a rectangular domain $[0, 1]^2$ with zero Dirichlet boundary condition. We are interested in learning the mapping from the permeability field $K(x, y)$ to the pressure field $h(x, y)$, i.e.,

$$\mathcal{G} : K(x, y) \mapsto h(x, y).$$

Here, we consider two datasets of different types of permeability fields:

- **Piecewise constant (PWC).** The first dataset is from Ref. [22]. The coefficient field K is defined as $K = \psi(\mu)$, where $\mu = \mathcal{R}(0, (-\Delta + 9I)^{-2})$ is the Gaussian random field with zero Neumann boundary conditions on the Laplacian, and the mapping ψ performs the binarization on the function, namely it converts the positive values to 12 and the negative values to 3. The grid resolution of K and h is 29×29.
- **Continuous (Cont.).** We use a truncated Karhunen–Loève (KL) expansion to express the permeability field $K(x, y) = \exp(F(x, y))$, where $F(x, y)$ denotes a truncated KL expansion for a given Gaussian process.

Table 5
L^2 relative error for the Darcy problem in a rectangular domain in Section 5.2.1. PWC, piecewise constant. Cont, continuous.

	Section 5.2.1 Darcy (PWC)	Section 5.2.1 Darcy (Cont.)
DeepONet w/o normalization	2.91 ± 0.04%	2.04 ± 0.13%
DeepONet w/ normalization	2.98 ± 0.03%	1.36 ± 0.12%
FNO w/o normalization	4.83 ± 0.12%	2.38 ± 0.02%
FNO w/ normalization	2.41 ± 0.03%	**1.19 ± 0.05%**
POD-DeepONet	**2.32 ± 0.03%**	**1.26 ± 0.07%**

Specifically, we keep the leading 100 terms in the KL expansion for the Gaussian process with zero mean and the following covariance kernel:

$$\mathcal{K}((x, y), (x', y')) = \exp\left[\frac{-(x - x')}{2l_1^2} + \frac{-(y - y')^2}{2l_2^2}\right],$$

with $l_1 = l_2 = 0.25$. Both $K(\boldsymbol{x})$ and $h(\boldsymbol{x})$ have the same resolution of 20×20.

Examples of these two Darcy datasets can be found in Fig. E.1.

Results. We enforce the zero Dirichlet boundary condition on DeepONet by choosing the surrogate solution as

$$\hat{u}(x, y) = 20x(1 - x)y(1 - y)\mathcal{N}(x, y, K),$$

where $\mathcal{N}$ is a DeepONet, as we discussed in Section 3.1.3. We use the coefficient 20 such that $20x(1-x)y(1-y)$ is of order 1 for $x \in [0, 1]$ and $y \in [0, 1]$. Similar to the Burgers' problem, by using the output normalization, a better accuracy of DeepONet and FNO is obtained (Table 5). For the case PWC, all the methods have errors around 2%, and POD-DeepONet achieves the smallest error. For the case Cont., all the methods have the error around 1%, and FNO is slightly better. However, POD-DeepONet is only worse by one standard deviation of the error (Table 5), so there is no significant difference between the performance of POD-DeepONet and FNO.

5.2.2. Darcy problem in a pentagram with a hole

Problem setup. Here we consider the Darcy flow in a pentagram with a hole, where $K(x, y) = 0.1$ and $f = -1$. The following Gaussian process is employed to generate boundary conditions for each boundary in the pentagram:

$$\begin{aligned} &h(x) \sim \mathcal{GP}(0, \mathcal{K}(x, x')), \\ &\mathcal{K}(x, x') = \exp[-\frac{(x - x')^2}{2l^2}], \ l = 0.2, \\ &x, x' \in [0, 1], \end{aligned} \tag{5.2}$$

Specifically, we draw one sample function from the above Gaussian process at each time, and compute the boundary values based on x for each boundary of the pentagram. While the boundary values at the boundary of the hole are fixed as 1, i.e., $h(x) = 1$, for all test cases. In the present case, we generate 2000 sample functions from Eq. (5.2) as boundary conditions, then we employ the operator networks to learn the mapping from the boundary condition to the pressure field in the entire domain"

$$\mathcal{G} : h(x, y)|_{\partial\Omega} \mapsto h(x, y).$$

Two representative solutions with corresponding boundary conditions are displayed in Fig. 4(a).

Results. As shown in Table 5, the normalization of inputs/outputs helps to reduce the generalization error in DeepONet, and hence, we utilize the normalization in DeepONet for all the Darcy problems in what follows.

The results from the DeepONet, POD-DeepONet, and dgFNO+ are presented in Table 6. We note that FNO cannot be used for these problems, and instead we use dgFNO+, which is the combination of dFNO+ in Section 3.2.2 and gFNO+ in Section 3.2.3. In particular, the following grid size, i.e., 44×44, is employed in dgFNO+, which has almost the same nodes as in the original mesh, i.e., 1938. As shown, the results from DeepONet and POD-DeepONet are quite similar, while both are about 2% more accurate than the dgFNO+.

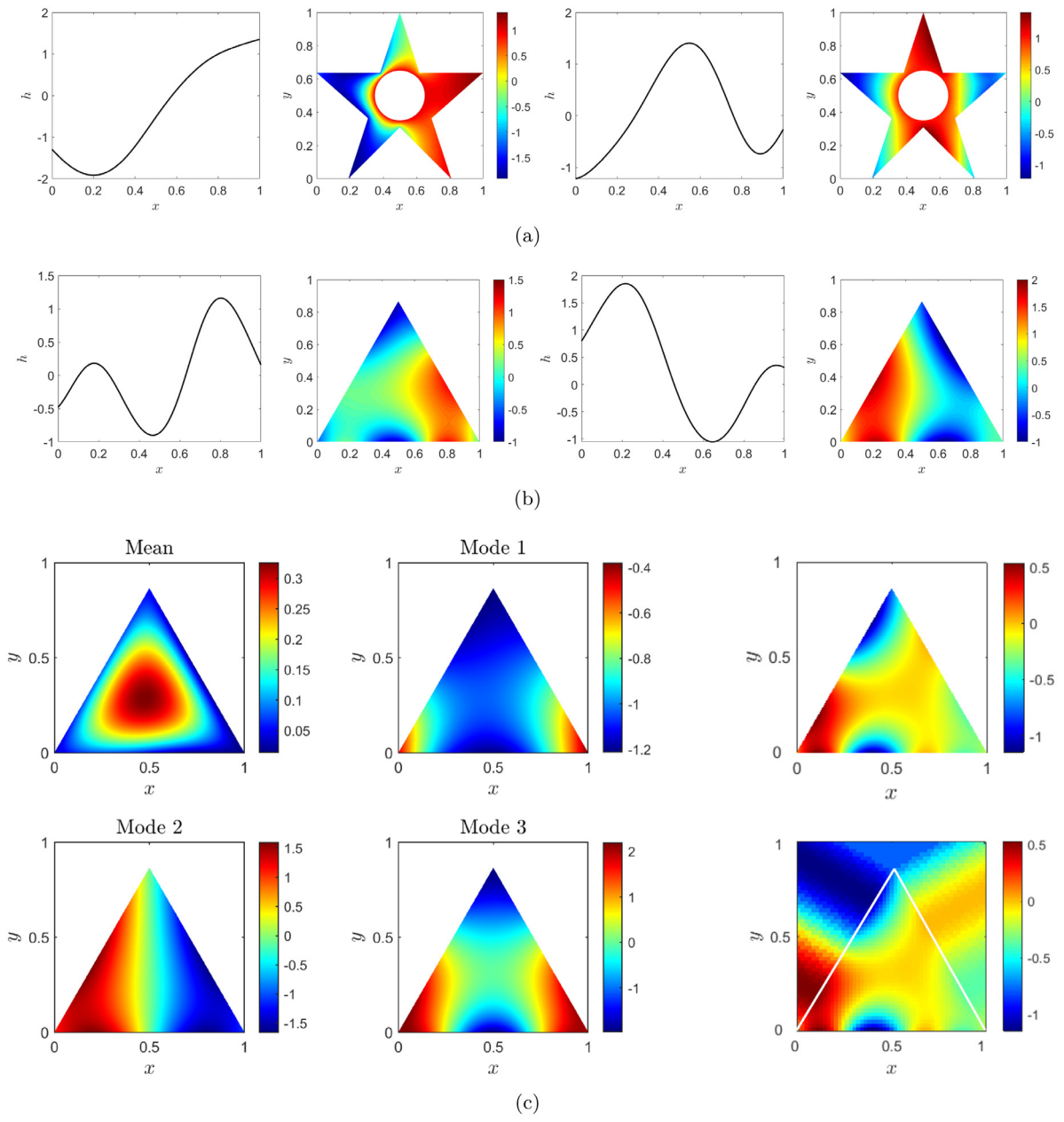

Fig. 4. Darcy flow in the domains of a pentagram and a triangle. (a) Darcy flow in a pentagram with a hole: two representative cases. For each case, Left: boundary condition; Right: pressure field. (b) Darcy flow in a triangular domain: two representative cases. Left: boundary condition; Right: pressure field. (c) Darcy flow in a triangular domain: Left: The first four POD modes used in POD-DeepONet. Right: An example of the augmented data for FNO training, where the padding is set by "nearest neighbor".

We further test an additional case using dgFNO+, in which we use a uniform grid with a resolution 50×50. The L^2 relative error for this case is $2.78 \pm 0.01\%$, which is better than the dgFNO+ with the resolution 44×44 but is still less accurate than the DeepONet and POD-DeepONet.

5.2.3. Darcy problem in a triangular domain

Problem setup. The setup for the case considered here is the similar as in Section 5.2.2, i.e., $K(x, y) = 0.1$, and $f = -1$. We also utilize the Gaussian process in Eq. (5.2) to generate the boundary conditions for each boundary

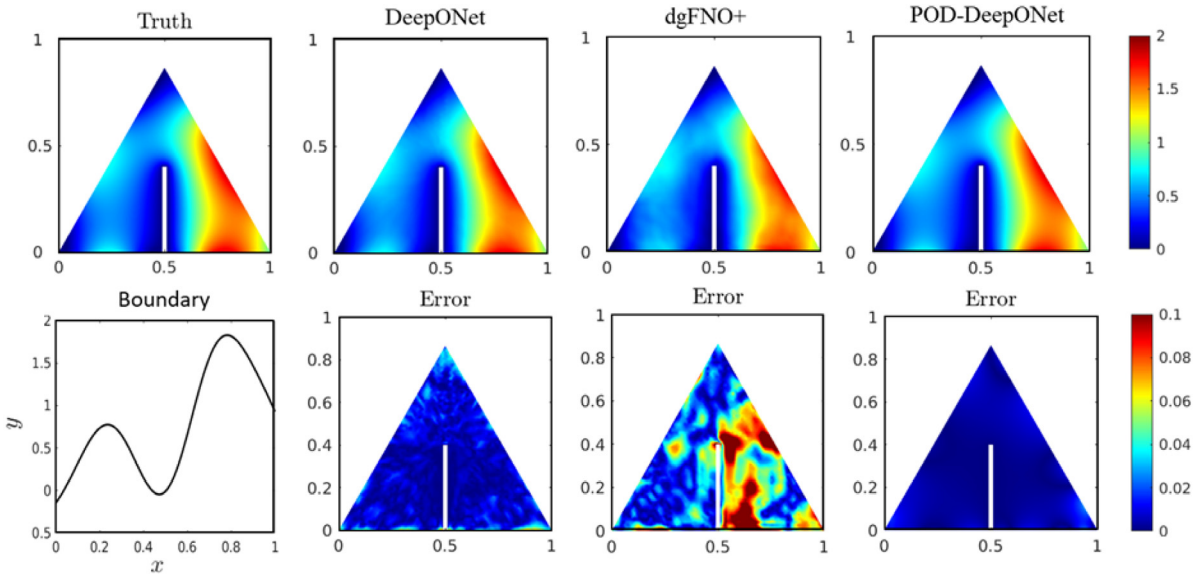

Fig. 5. Darcy flow in a triangular domain with a notch. For a representative boundary condition, the pressure field is obtained using DeepONet, dgFNO+, and POD-DeepONet. The prediction errors for the three operator networks are shown against the respective plots. The ground truth is simulated using the *PDE Toolbox* in Matlab. The predicted solutions and the ground truth share the same colorbar, while the errors corresponding to each of the neural operators are plotted on the same colorbar.

Table 6
L^2 relative error for the Darcy flows in complex geometries. dgFNO+ is the combination of dFNO+ and gFNO+.

	Section 5.2.2	Section 5.2.3	Section 5.2.4
	Darcy (Pentagram)	Darcy (Triangular)	Darcy (Notch)
DeepONet	1.19 ± 0.12%	0.43 ± 0.02%	2.64 ± 0.02%
FNO	–	–	–
POD-DeepONet	**0.82 ± 0.05%**	**0.18 ± 0.02%**	**1.00 ± 0.00%**
dgFNO+	3.34 ± 0.01%	1.00 ± 0.03%	7.82 ± 0.03%

of the triangular domain, and then use the operator networks to map the boundary conditions to the pressure field in the entire domain. Specifically, 861 nodes are employed in the numerical solver. Similarly, we also display two representative solutions with corresponding boundary conditions in Fig. 4(b).

Results. As demonstrated in Table 6, both the DeepONet and POD-DeepONet are more accurate than the dgFNO+. POD-DeepONet achieves the best accuracy (0.18%) among the three test approaches. The first four POD modes are demonstrated in Fig. 4(c)(Left), where we note that each mode is scaled by its L^2-norm. For dgFNO+, we use a uniform grid with a resolution of 51 × 51 and the "nearest neighbor" in Section 3.2.3, and an example of the solution is illustrated in Fig. 4(c) (Right).

5.2.4. Darcy problem in a triangular domain with notch

Problem setup. On the triangular domain discussed in Section 5.2.3, we now add a notch and also learn the operator mapping from the boundary conditions to the pressure field. The boundary conditions are generated using the Gaussian process discussed in Eq. (5.2). A representative case is shown in Fig. 5, where the predicted results and the errors corresponding to DeepONet, dgFNO+, and POD-DeepONet are shown for a specific boundary.

Results. The L_2 relative error for this example is given in Table 6. Amongst all the methods, POD-DeepONet outperforms the others, followed by DeepONet. The L_2 relative error for predicting the flow in a triangular domain with a notch is $2.64 \pm 0.02\%$ using DeepONet, and is $7.82 \pm 0.03\%$ using dgFNO+. The lowest prediction error is $1.0 \pm 0.00\%$, obtained using POD-DeepONet.

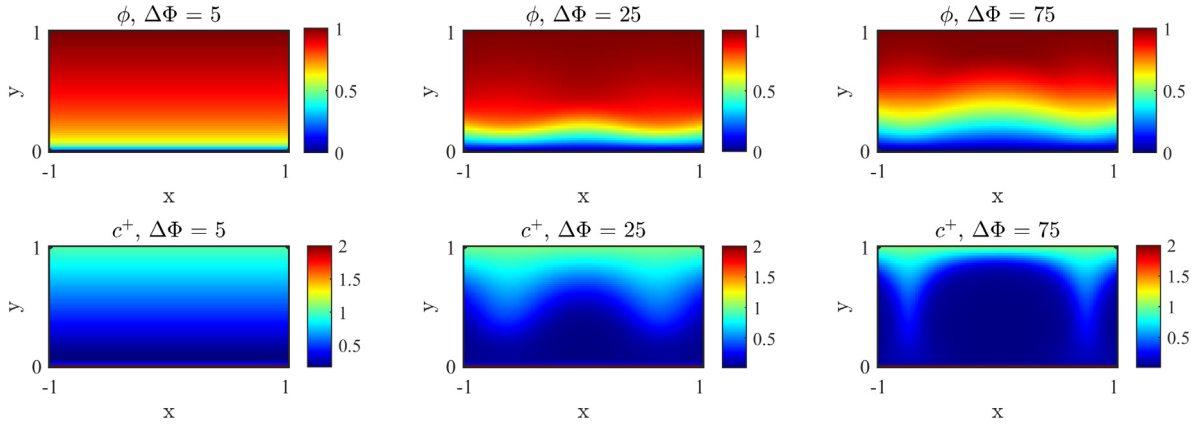

Fig. 6. Electroconvection problem: three representative cases. First row: electric potential field. Second row: cation concentration field. Note that there is a stiff boundary at $y = 0$ in the concentration field, where the concentration drops from $c^+ = 2$ to approximately zero.

5.3. Multiphysics electroconvection problem

Problem setup. Following the setup in [42], we consider a 2D electroconvection problem, which is a multiphysics phenomenon involving coupling of the flow field with the electric field, the cation and anion concentration fields. The full governing equations, including the Stokes equations, the electric potential and the ion transport equations, can be written as follows:

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} &= -\nabla p + \nabla^2 \boldsymbol{u} + \boldsymbol{f_e}, \\ \nabla \cdot \boldsymbol{u} &= 0, \\ -2\epsilon^2 \nabla^2 \phi &= \rho_e, \\ \frac{\partial c^{\pm}}{\partial t} &= -\nabla \cdot \left(c^{\pm} \boldsymbol{u} - \nabla c^{\pm} \mp c^{\pm} \nabla \phi \right), \end{aligned} \tag{5.3}$$

where $\boldsymbol{u}$ and p are the velocity and the pressure fields, respectively, and ϕ is the electric potential. Moreover, c^+ and c^- are the cation and anion concentrations, respectively. Also, $\rho_e = (c^+ - c^-)$ is the free charge density, $f_e = -0.5\rho_e \nabla\phi / 2\epsilon^2$ is the electrostatic body force, where ϵ is the Debye length. The investigated domain is defined as $\Omega : [-1, 1] \times [0, 1]$ with a regular mesh containing 101×51 grid points. By defining $\epsilon = 0.01$, the electroconvection described in Eq. (5.3) becomes a steady flow, where the flow pattern is uniquely dependent on the electric potential difference ($\Delta\Phi$) acted on the upper and lower boundaries.

In this problem, the operator networks are expected to learn the mapping from the 2D electric potential field $\phi(x, y)$ to the 2D cation concentration field $c^+(x, y)$:

$$\mathcal{G} : \phi(x, y) \mapsto c^+(x, y).$$

Different flow fields are generated by modifying the boundary condition of ϕ, namely using $\Delta\Phi = 5, 10, \ldots, 75$, which results in 15 steady states for network training. We also consider two unseen conditions, namely $\Delta\Phi = 13.4$ and $\Delta\Phi = 62.15$, which are applied for testing. The equations are solved by using a high-order spectral element method. Three training cases of the electroconvection flow are demonstrated in Fig. 6 and more details are included in [42].

Results. The evaluation errors for this example are given in Table 7, where we find that the testing errors of the different methods are all very small. Nevertheless, POD-DeepONet outperforms others, followed by DeepONet with data normalization. The reason that FNO performs relatively worse in this case is because of a stiff boundary at $y = 0$ in the output field, where the concentration drops from $c^+ = 2$ to approximately zero. Note that the output resolution of this dataset is 101×51, which means that there are 5151 sampling points for each output function used for training. However, it is reported in [42] that the DeepONet can be trained with much less data

Table 7
L^2 relative error for the multiphysics electroconvection problem.

	Section 5.3 Electroconvection
DeepONet w/o normalization	0.26 ± 0.04%
DeepONet w/ normalization	0.28 ± 0.02%
FNO w/o normalization	1.00 ± 0.01%
FNO w/ normalization	0.43 ± 0.01%
POD-DeepONet	**0.14** ± **0.03**%

Table 8
L^2 relative error of wave propagation for continuous and discontinuous problems in Section 5.4.

	Section 5.4.1 Advection (I)	Section 5.4.1 Advection (II)	Section 5.4.1 Advection (III)
DeepONet	0.22 ± 0.03%	0.27 ± 0.01%	**0.32** ± **0.04**%
FNO	0.66 ± 0.10%	54.4 ± 0.00%	47.7 ± 0.00%
POD-DeepONet	**0.04** ± **0.00**%	**0.08** ± **0.00**%	0.40 ± 0.00%
dFNO+1	–	0.22 ± 0.01%	0.60 ± 0.02%
dFNO+2	–	3.89 ± 1.26%	10.9 ± 2.08%
	Section 5.4.2 Instability waves	Section 5.4.3 Compressible Euler (p)	Section 5.4.3 Compressible Euler (N_2)
DeepONet	**8.90** ± **0.60**%	0.068 ± 0.011%	0.043 ± 0.006%
FNO	17.8 ± 0.92%	0.076 ± 0.005%	0.044 ± 0.004%
POD-DeepONet	20.8 ± 1.12%	**0.020** ± **0.004**%	**0.012** ± **0.002**%

measurements (i.e., 800 random points for each output function) to achieve a similar accuracy of 0.49 ± 0.04%. This also demonstrates the flexibility of DeepONet training, since FNO requires the output representation to be on a regular mesh, while DeepONet does not have such a stringent requirement.

5.4. Wave propagation for continuous and discontinuous problems

5.4.1. Advection equation

Problem setup. We consider the wave advection equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \qquad x \in [0, 1], \quad t \in [0, 1],$$

with periodic boundary condition. We choose the initial condition as a square wave centered at $x = c$ of width w and height h

$$u_0(x) = h 1_{\{c-\frac{w}{2}, c+\frac{w}{2}\}},$$

where (c, w, h) are randomly chosen from $[0.3, 0.7] \times [0.3, 0.6] \times [1, 2]$. We first learn the mapping from the initial condition $u_0(x)$ to the solution at $t = 0.5$:

$$\text{Case I:} \qquad \mathcal{G}_1 : u_0(x) \mapsto u(x, t = 0.5),$$

and then learn the operator mapping from the initial condition $u_0(x)$ to the solution $u(x, t) = u_0(x - t)$ of the whole domain

$$\text{Case II:} \qquad \mathcal{G}_2 : u_0(x) \mapsto u(x, t).$$

The mesh of the solution is chosen as 40 × 40 in space–time. We also test a more complicated initial condition (Case III):

$$u_0(x) = h_1 1_{\{c_1-\frac{w}{2}, c_1+\frac{w}{2}\}} + \sqrt{\max(h_2^2 - a^2(x - c_2)^2, 0)}.$$

Results. For case I, DeepONet, POD-DeepONet and FNO all have errors $< 1\%$, while POD-DeepONet achieves the smallest error ($< 0.1\%$, Table 8). Here, the output of POD-DeepONet is rescaled by a factor of $1/p$ and the effect of different rescaling factors is summarized in Table D.1.

For Cases II and III, DeepONet and POD-DeepONet still obtain good accuracy of the order of 0.1% (Table 8). Because FNO cannot directly map from the 1D initial condition to the 2D solution, we considered the following three approaches.

- **Brute-force FNO (2D):** We first use FNO in a brute-force way. Because the input u_0 is only a function of x, we directly repeat the same u_0 multiple times to match the 2D size of the output function, so that it is a valid 2D input. We find that the training loss is always very large $\sim \mathcal{O}(0.1)$ and cannot be improved, even if we do not truncate Fourier modes, use a large hidden dimension, and add more hidden layers. The L^2 relative error is $\sim$50% (Table 8).
- **dFNO+1:** We use dFNO+ in Section 3.2.2 for Case I using Method 1. Compared to the brute-force FNO above, here we also add the t coordinate as input, and the optimization of FNO improves significantly. dFNO+1 achieves an error $< 1\%$ (Table 8), but it is still worse than DeepONet and POD-DeepONet.
- **dFNO+2:** We also use dFNO+ in Section 3.2.2 for Case I using Method 2, i.e., instead of using FNO in 2D, we test FNO in 1D with RNN for time-marching. We find that dFNO+2 is very hard to train, and the final result is unstable, as also observed in [22]. In our 10 independent experiments, dFNO+2 gets stuck at a bad local minimum with (L^2 relative error $> 20\%$) for 4 times. Here we truncate the FFT to the first 16 modes, but using more modes does not improve the accuracy or reduce the probability of training failure. We also note that the training cost of dFNO+2 is much more expensive than FNO in 2D.

5.4.2. Linear instability waves in high-speed boundary layers

Problem setup. The early stages of transition to turbulence in high-speed aerodynamics often involve exponential amplification of linear instability waves. A visualization of an instability wave in a high-speed and spatially developing boundary layer is shown in Fig. 7A. We aim to predict the evolution of linear instability waves in a compressible boundary layer, i.e., how the upstream instability wave will amplify or decay within a region of interest downstream. Here, we consider small-amplitude instability waves, which can be accurately described by the linear parabolized stability equations (PSE) derived from the Navier–Stokes equations by decomposing the flow into the sum of a base flow and a perturbation.

The instability waves depend on the flow parameters. Here we consider air with Prandtl number $Pr = 0.72$ and ratio of specific heats $\gamma = 1.4$. The free-stream Mach number is $Ma = 4.5$ and the free-stream temperature is $T_0 = 65.15$ K. We set the inflow location of our configuration slightly upstream at $\sqrt{Re_{x_0}} = 1800$. When the perturbation frequency is ω, the instability frequency is $\omega 10^6/\sqrt{Re_{x_0}}$. In our dataset, the Reynolds number of the domain of integration spans from 1800^2 to 2322^2, and the problem was solved for 59 different perturbation frequencies in the range $[100, 125]$ with arbitrarily sampled phase. Moreover, the input function is defined on a mesh with resolution (20, 47), and the output function is on a mesh with resolution (111, 47). Because the output functions in the dataset differ in amplitudes by more than two orders of magnitude, a weighted MSE is used, where each loss term is weighted by the amplitude of each function. More details about this problem and the dataset can be found in Ref. [43].

Results. This problem has been solved by DeepONet with 4 Fourier features in Ref. [43]. The accuracy of DeepONet is $8.90 \pm 0.60\%$ (the forward case with two-mode combinations F_2 in Fig. 18 in Ref. [43]), and the training trajectory is shown in Fig. 7B (extracted from Fig. 15a in Ref. [43]).

The output functions in the dataset differ by more than two orders of magnitude, and thus if we compute the POD modes, the functions with large magnitude dominate. Hence, we first normalize all the functions in the training dataset such that the maximum value of each function is 1, and then compute the POD modes. We note that this normalization is only used to compute POD modes, and when we train and test the network, we still use the original dataset. This POD normalization makes POD-DeepONet work better, but POD-DeepONet still has a larger error ($20.8 \pm 1.12\%$) than DeepONet. A better strategy to compute the POD modes for problems with multiple scales needs to be developed in the future.

For FNO, because the input and outputs functions have different mesh resolution, we first use a linear interpolation to interpolate the input function from (20, 47) to (111, 47). FNO performs better than POD-DeepONet,

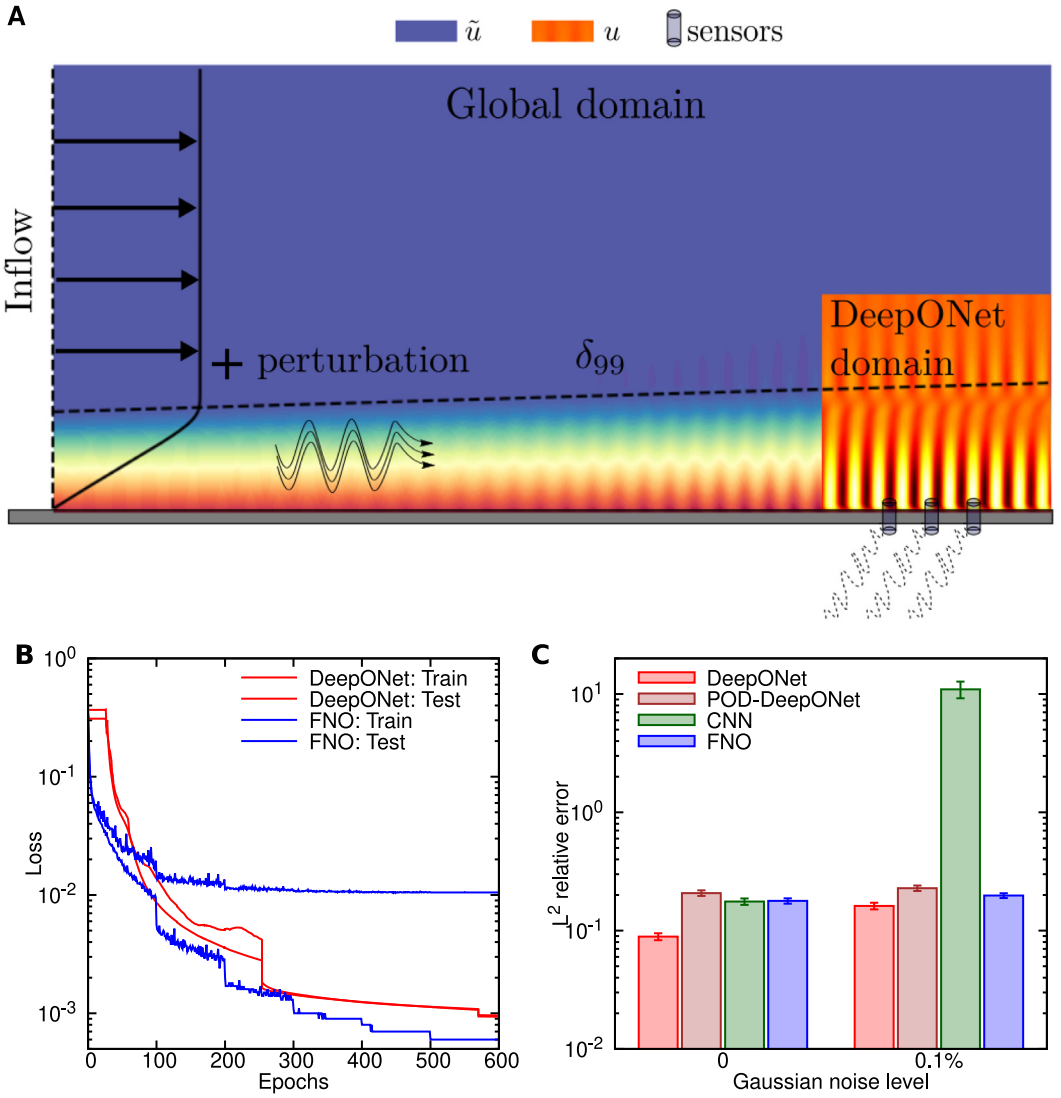

Fig. 7. Linear instability waves in high-speed boundary layers. (**A**) Visualization of an instability wave in a spatially developing boundary layer. At the inlet to the computational domain, the base flow is superposed with instability waves. The dashed line marks the 99% thickness of the boundary layer. The objective is to accurately predict the downstream evolution of the instability wave. (**B**) The training and testing losses during the training process. (**C**) The L^2 relative errors of different networks for noiseless inputs and inputs with a 0.1% Gaussian noise. Panel A is adapted from Ref. [43].

but is much worse than DeepONet with Fourier features. Although DeepONet and FNO have a similar training error, DeepONet has a smaller testing error than FNO (Fig. 7B). In FNO, the generalization gap between the training and testing errors is more than one order of magnitude gap, while there is almost no generalization gap for DeepONet.

We further analyze the robustness of different networks to input uncertainties by adding a Gaussian noise of 0.1% to the input functions during testing. We note that training data is still noiseless. We also add the result of CNN in Ref. [43] for the comparison. The errors of DeepONet, POD-DeepONet and FNO only increase slightly for noisy inputs and remain satisfactory, but the error of CNN increases by two orders of magnitudes, respectively (Fig. 7C). This implies that the mapping that CNN has learned is unstable, but DeepONet, POD-DeepONet and FNO are more stable.

5.4.3. Compressible Euler equations with non-equilibrium chemistry

Problem setup. We consider one-dimensional inviscid high-speed flows with dissociation involving three species and non-equilibrium chemistry described by a general equation of state as studied in the works of [56,57]:

$$\begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho u \\ \rho E \end{pmatrix}_t + \begin{pmatrix} \rho_1 u \\ \rho_2 u \\ \rho_3 u \\ \rho u^2 + p \\ (E+p)u \end{pmatrix}_x = \begin{pmatrix} 2M_1\omega \\ -M_2\omega \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

and

$$\rho = \sum_{s=1}^{3} \rho_s, \qquad p = RT \sum_{s=1}^{3} \frac{\rho_s}{M_s}, \qquad E = \sum_{s=1}^{3} \rho_s e_s(T) + \rho_1 h_1^0 + \frac{1}{2}\rho u^2,$$

where the enthalpy h_1^0 is a constant, R is the universal gas constant, M_s is the molar mass of species s, and the internal energy $e_s(T) = 3R/2M_s$ and $5R/2M_s$ for mono-atomic and diatomic species, respectively. The rate of the chemical reaction is given by

$$\omega = \left(k_f(T)\frac{\rho_2}{M_2} - k_b(T)\left(\frac{\rho_1}{M_1}\right)^2 \right) \sum_{s=1}^{3} \frac{\rho_s}{M_s}, \quad k_f = CT^{-2}e^{-\mathcal{E}/T},$$
$$k_b = k_f / e^{b_1 + b_2 \log z + b_3 z + b_4 z^2 + b_5 z^3}, \quad z = 10,000/T,$$

where b_i, C and $\mathcal{E}$ are constants which can be found in [56,58]. The model involves three species, namely, O_2, O and N_2 ($\rho_1 = \rho_O$, $\rho_2 = \rho_{O_2}$ and $\rho_3 = \rho_{N_2}$) with the reaction:

$$O_2 + N_2 \Longleftrightarrow O + O + N_2.$$

We consider the following initial condition:

$$T_0(x) = 8000 \text{ K}, \quad p_0(x) = \frac{p_L - p_R}{2}(1 - \tanh(x/\eta)) + p_L,$$

where η is a parameter in the range of $[0.02, 0.2]$, and p_L and p_R are constant left and right pressure levels. This is a multi-physics problem involving different quantities. Here, we only consider the pressure field and the Nitrogen density to demonstrate the comparison between different operator learners. The neural operators are expected to learn two mappings: from the initial condition $p_0(x)$ to the solution $p(x)$ at $t = 0.0002$, and from $N_2(x, t = 0)$ to $N_2(x, t = 0.0002)$.

To generate the training data, we use the positivity-preserving high order discontinuous Galerkin (DG) schemes developed by Zhang and Shu in [57]. In the simulation of the DG solver, we set $\Delta x = 10^{-3}$ and set Δt according to the CFL condition of the DG scheme.

Results. The L^2 relative errors of DeepONet and FNO are in Table 8 and a testing example is demonstrated in Fig. 8. All the networks provide results with good accuracy, and POD-DeepONet performs the best. We note that the FNO relies on Fourier transformation, which is not very accurate for discontinuous functions. On the contrary, the DeepONet performs well for functions with discontinuity, as shown in the insets of Fig. 8.

5.5. Predicting surface vorticity of a flapping airfoil

Problem setup. Here, we predict the surface vorticity of a flapping airfoil based on the angle of attack. We perform simulation of the flow over a NACA0012 airfoil. The flapping airfoil setup is achieved by defining an oscillating inflow velocity, which is expressed as:

$$u = U_\infty \cos\left(\frac{\alpha_0 \pi}{180} \times \frac{\sin(2f\pi t) + 1.0}{2}\right),$$
$$v = U_\infty \sin\left(\frac{\alpha_0 \pi}{180} \times \frac{\sin(2f\pi t) + 1.0}{2}\right),$$

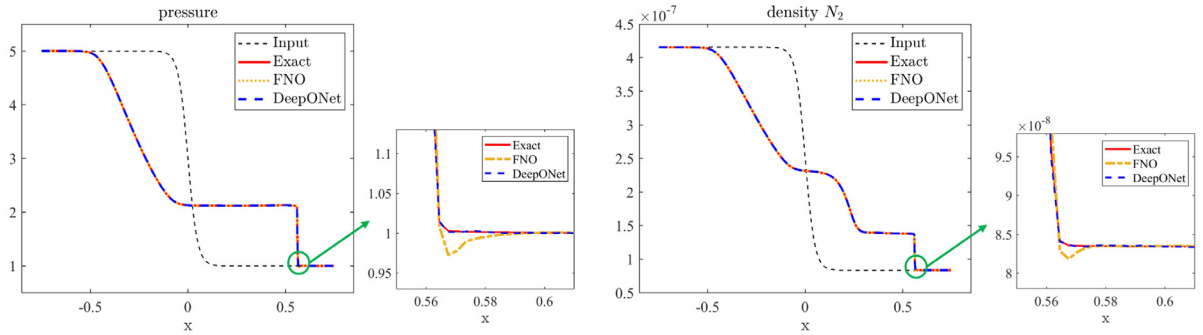

Fig. 8. Compressible Euler equations: results of a testing example. Left: neural operators for pressure. Right: neural operators for N_2 density. The black dash line is the initial condition. The red, orange, blue lines denote the exact solution, the FNO prediction and the DeepONet prediction, respectively.

where $\alpha_0 = 15^\circ$ is the reference angle of attack (AOA), $f = 0.2$ is the frequency and $U_\infty = 1$. In this case, the Reynolds number based on the chord length is $Re = 2500$. We can simply define the normalized time-dependent AOA as $\alpha(t) = \sin(2f\pi t)$, which is used as the input of the learning algorithms hereafter. Specifically, we aim to map the AOA $\alpha(t)$ to the vorticity on the airfoil surface, which is denoted by $\omega(s, t)$ and s is the location index on the surface. The airfoil geometry is illustrated in Fig. 9A, where the surface of the airfoil is discretized by using 152 points.

In this example, we only have one time-dependent signal, which covers about 56.4 time units. We apply approximately 36.6 time units for neural network training and the rest are used for prediction and validation. Moreover, we divide the training data into multiple independent signals, which involve two periods and have different phases. An example of the input–output functions for standard DeepONet and FNO training is illustrated in Fig. 9B. By doing this, the training data contain 28 input–output pairs, while the testing data (about 20 time units) are composed of two signals whose phases are not included in training.

For vanilla DeepONet and FNO, the operator can be expressed as: $\omega(s, t) = \mathcal{G}(\alpha)(s, t)$. In this example, we also adopt a modified DeepONet with feature expansion, where a couple of historical states of the time signal are fed into the trunk net as the features and replace the time coordinate (Section 3.1.2). Such an operator is denoted as:

$$\omega(s, t) = \mathcal{G}(\alpha)(s, \omega(s, t-1), \ldots, \omega(s, t-k)),$$

where k denotes the number of historical states. We note that in the prediction stage, only the initial data of ω is given. The predictions from the network itself are concatenated and fed into the trunk net for the future predictions. A schematic of this DeepONet is shown in Fig. 9C.

Results. The L^2 relative errors of DeepONet and FNO for the testing data are given in Table 9. Note that here we learn an operator mapping from a 1D function (i.e., function of time) to a 2D function (i.e., function of time and space). As mentioned above, FNO requires dimension augmentation for the input function in this case. If the input only involves the time coordinate, i.e., the brute-force way in Section 5.4.1, then FNO fails to predict correctly the output (16.20% error). However, when the input is augmented to involve both time and space coordinates (Section 3.2.2 Case I Method 1), the relative error of dFNO+ reduces to 3.56%.

The vanilla DeepONet predicts the output with satisfactory result (3.65%). In addition, when we apply the modified DeepONet with 5 historical states of the investigated quantity as the features, the error becomes even smaller (2.87%). The prediction results of this modified DeepONet are illustrated in Figs. 9D and E, where (D) shows the time-dependent signals and (E) shows the vorticity profiles over the airfoil surface. It is worth noting again that in the prediction process, the predictions from DeepONet are concatenated to the input of the trunk net in order to perform future evaluation. We can observe great consistency between the truth and the DeepONet prediction from the figures.

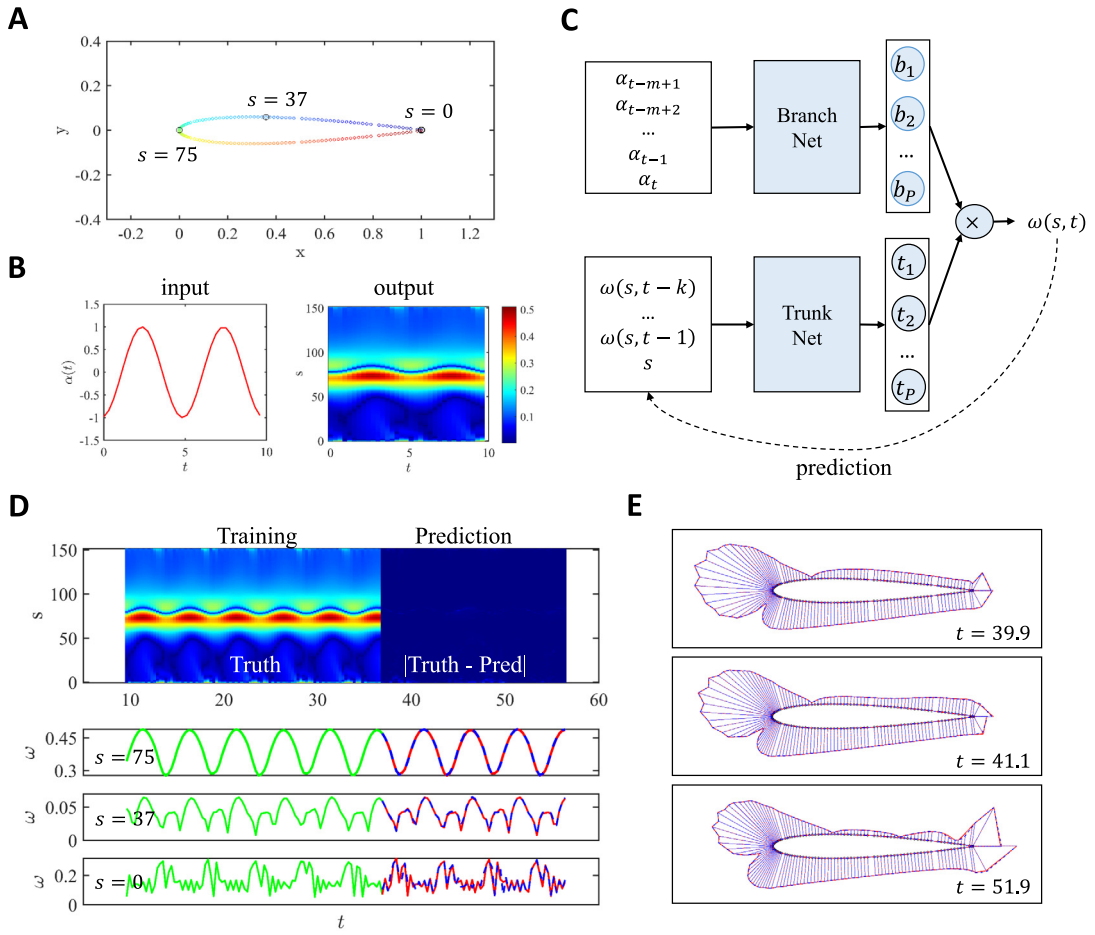

Fig. 9. Predicting vorticity on the surface of a flapping airfoil. (**A**) Geometry of NACA0012 airfoil. (**B**) An example of the input function $\alpha(t)$ and output function $\omega(s,t)$ that are used to train the operator networks. (**C**) Schematic of the modified DeepONet, which includes a few historical states in the trunk net as the features to replace the time coordinate. (**D**) Testing result of the modified DeepONet. Top: 2D visualization in space–time; bottom: 1D signals at different surface locations. (**E**) Vorticity profiles on the airfoil surface at three time stamps. The red and blue colors in (D) and (E) represent the truth and the DeepONet prediction, respectively.

Table 9
L^2 relative error of predicting surface vorticity of a flapping airfoil.

	Section 5.5 Flapping airfoil
DeepONet	3.65 ± 0.02%
FNO	16.20 ± 0.01%
DeepONet (feature expansion)	**2.87** ± **0.24**%
dFNO+	3.56 ± 0.10%

Table 10
L^2 relative error for the Navier–Stokes equation in the vorticity–velocity form.

	Section 5.6 Navier–Stokes
DeepONet w/o normalization	2.51 ± 0.07%
DeepONet w/ normalization	1.78 ± 0.02%
FNO w/o normalization	2.62 ± 0.03%
FNO w/ normalization	1.81 ± 0.02%
POD-DeepONet	**1.36** ± **0.03**%

5.6. Navier–Stokes Equation in the vorticity–velocity form

Problem setup. Following the problem setup in [22], we consider the 2D incompressible Navier–Stokes equation in the vorticity–velocity form:

$$\begin{aligned} \partial_t \omega + \boldsymbol{u} \cdot \nabla \omega &= \nu \Delta \omega + \boldsymbol{f}, && x \in [0,1]^2,\ t \in [0,T], \\ \nabla \cdot \boldsymbol{u} &= 0, && x \in [0,1]^2,\ t \in [0,T], \\ \omega(x,0) &= \omega_0(x), && x \in [0,1]^2, \end{aligned}$$

where $\omega(x, y, t)$ and $\boldsymbol{u}(x, y, t)$ are the vorticity and velocity, respectively. The viscosity ν in our experiment is 0.001. The forcing term is defined as:

$$f(x, y) = 0.1 \sin(2\pi(x + y)) + 0.1 \cos(2\pi(x + y))$$

and the periodic boundary condition is imposed. The dataset we use in this paper is identical to that in [22]. We are interested in learning the operator mapping the vorticity field at the first ten time steps $t \in [0, 10]$ to the vorticity at a target time step $T = 20$. The solution is determined by modifying the initial condition ω_0, which is given by a Gaussian random field $\mathcal{R}(0, 7^{3/2}(-\Delta + 49I)^{-2.5})$. The spatial resolution is fixed to be 64×64 both for training and testing.

Results. DeepONet in this example is imposed with periodic boundary condition in Section 3.1.3 and a CNN is used to process the input function (namely $\omega(t_1 - t_{10})$) in the branch net. For FNO, the input is the concatenation of $\omega(t_1 - t_{10})$ and the grid points. The L^2 relative errors of DeepONet and FNO are illustrated in Table 10, indicating that the accuracy of DeepONet and FNO are comparable in this case. First, we find that the networks without data normalization perform the worst (both giving more than 2% errors). The data normalization improves the testing accuracy for both DeepONet and FNO, resulting in the errors of 1.78% and 1.81%, respectively. The best result is obtained by POD-DeepONet (1.36%). Here, the output of POD-DeepONet is rescaled by a factor of $1/\sqrt{p}$, and the effect of different rescaling factors is summarized in Table D.1. Furthermore, we also computed the enstrophy spectra (i.e., the vorticity variance) based on the vorticity information reconstructed by different networks. As shown in Fig. 10, the spectrum of POD-DeepONet is more consistent with the spectrum of reference data, followed by the DeepONet and FNO. The discrepancy of FNO may be caused by the truncation in the Fourier layers.

5.7. Regularized cavity flows

Here we consider a two-dimensional lid-driven flow in a square cavity (i.e., $x, y \in [0, 1]$), which can be described by the incompressible Navier–Stokes equations as

$$\begin{aligned} \nabla \cdot \boldsymbol{u} &= 0, \\ \partial_t \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} &= -\nabla P + \nu \nabla^2 \boldsymbol{u}, \end{aligned}$$

where $\boldsymbol{u} = (u, v)$ denotes the velocity in $x-$ and $y-$ directions, respectively; P is the pressure; and ν is the kinematic viscosity. We consider two cases with different boundary conditions for the upper wall, i.e., a time-independent (Case A) and a time-dependent one (Case B). In particular, the boundary conditions are expressed as

$$\text{Case A: } u = U\left(1 - \frac{\cosh[r(x - \frac{L}{2})]}{\cosh(\frac{rL}{2})}\right),\ v = 0,$$

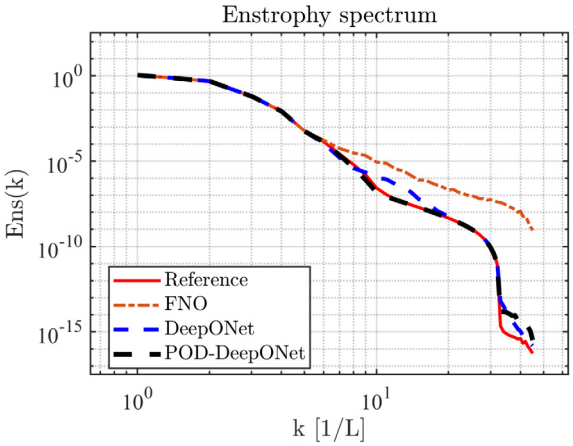

Fig. 10. Enstrophy spectra of the predicted vorticity fields by different methods. We note that the spatial resolution of the mesh for training is 64 × 64.

$$\text{Case B: } u = U\left(1 - \frac{\cosh[r(x-\frac{L}{2})]}{\cosh(\frac{rL}{2})} + 0.8\sin(2\pi x)\sin(5t)\right),\ v = 0,$$

where U, r and L are constant. Specifically, $r = 10$, $L = 1$ is the length of the cavity. In addition, the remaining walls are stationary in both cases. The aforementioned equations are then solved using the lattice Boltzmann method (LBM) [59] to generate training data (see more details for the data generation in Appendix F).

For Case A, we generate 100 velocity flow fields at different Reynolds numbers (Re = UL/ν), i.e., spanning from 100 to 2080 with a step size 20. We then simulate the flow fields for 10 randomly generated Reynolds numbers within the range [100, 2080], which are not seen in the training dataset and employ them as the testing dataset. We take the boundary condition on the upper wall as the input for the operator networks, and the corresponding output is the converged velocity field.

In Case B, we fix the Re as 1000, and set the maximum iteration number in LBM as 1,500,000, i.e., T = 1,500,000 dt, where dt is the time step used in LBM. We then save the velocity fields every 100 time steps in the last 10,000 iterations, suggesting that we have a total number of 100 snapshots for the velocity fields. Similarly, we train the operator networks to learn the mapping from the boundary condition to the corresponding velocity field. Specifically, we consider the following two cases:

- Unsteady I: We utilize the first 90 snapshots in the training and the last 10 as testing dataset.
- Unsteady II: We utilize the first 10 snapshots in the training and the last 10 as testing dataset. A representative velocity field for Re = 1000 is illustrated in Fig. 11.

Note that the boundary condition is assumed to be known in both Cases A and B. We then use $\sin(5t)$, which is introduced in the boundary condition in Case B as an extra feature for the networks to enhance the predicted accuracy. In particular, we employ $\sin(5t)$ as (1) an additional input for the branch network of DeepONet (by concatenating the boundary condition and $\sin(5t)$), and (2) an additional channel in dFNO+, i.e., we change the inputs from (x, y, t, u_{bc}) to $(x, y, t, \sin(5t), u_{bc})$, where u_{bc} represents the velocity along the x direction at the upper wall. See more details on the feature expansion of DeepONet and FNO in Sections 3.1.2 and 3.2.4.

Results. As displayed in Table 11: (1) For Case A, the results from dFNO+ are slightly more accurate than DeepONet, while POD-DeepONet achieves the best accuracy among them. (2) For Case B, DeepONet, dFNO+, and POD-DeepONet without feature expansion cannot provide accurate predictions for the velocity fields, and the relative errors from these three methods are quite similar. However, the extra feature can significantly reduce the predicted accuracy as compared to the results without feature expansion for all operator networks. In addition, DeepONet and dFNO+ with feature expansion have similar accuracy, and POD-DeepONet provides the most accurate predictions among all the methods. It is also interesting to observe that we can still get quite accurate predictions (around 2% relative errors for all methods) for the velocity field as we reduce the number of training dataset from 90 to 10 with the feature expansion in Unsteady II.

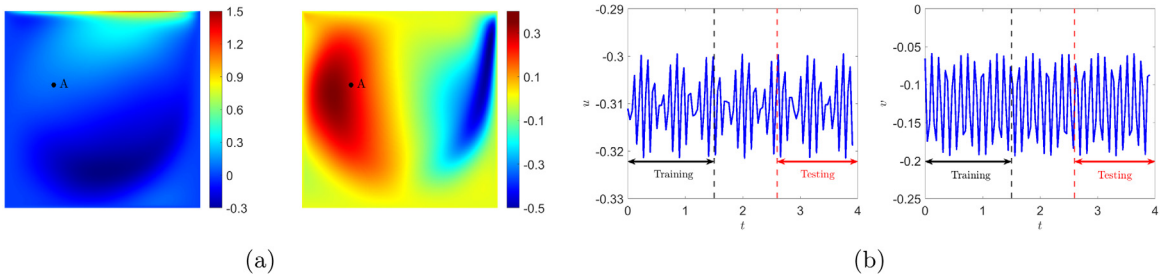

Fig. 11. Unsteady cavity flows for Re $= 1000$ at $T = 1{,}500{,}000\ dt$. (a) Velocity field obtained from LBM. Left: u, Right: v. (b) Time series for the velocity at location A. u and v are normalized by U, and t is normalized by L/U.

Table 11
L^2 relative error for the regularized cavity flows. Unsteady I: number of training data: 90; number of testing data: 10. Unsteady II: number of training data: 10; number of testing data: 10.

	Section 5.7 Cavity (Steady)	Section 5.7 Cavity (Unsteady I)	Section 5.7 Cavity (Unsteady II)
DeepONet	1.20 ± 0.23%	14.4 ± 0.53%	
FNO	–	–	–
POD-DeepONet	**0.33 ± 0.08%**	14.2 ± 0.51%	
dFNO+	0.63 ± 0.04%	15.0 ± 0.17%	
DeepONet (feature expansion)		0.51 ± 0.12%	2.24 ± 0.21%
POD-DeepONet (feature expansion)		**0.18 ± 0.02%**	**1.51 ± 0.27%**
dFNO+ (feature expansion)		0.56 ± 0.03%	1.78 ± 0.22%

6. Summary

In this study, we have investigated the performance of two neural operators that have shown early promising results: the deep operator network (DeepONet) and the Fourier neural operator (FNO). The main difference between DeepONet and FNO is that DeepONet does not discretize the output, but FNO does. Moreover, DeepONet can employ any type of neural network architectures in the branch net whereas FNO has a fixed architecture, and hence DeepONet is more flexible than FNO in terms of problem settings and datasets (see comparison details in Table 2). Here, we have designed 16 benchmarks that have elements of industrial-complexity applications, e.g., unsteadiness, complex geometry, noisy data, and have generated data that will be available publicly so that interested researchers can test their own ideas against the results presented herein. In particular, we have shown that the vanilla DeepONet and the vanilla FNO may lead to suboptimal results in several of the 16 benchmarks that exhibit multiscale behavior and non-smooth solutions. To this end, we have proposed several extensions of both DeepONet and FNO (e.g., POD-DeepONet, dFNO+, gFNO+, and feature expansion) to either improve their accuracy or expand their capability to tackle diverse PDE based applications, especially for FNO to be able to deal with problems involving complex-geometry domains and mappings with different dimensionality of the input–output spaces.

We have compared DeepONet and FNO both theoretically and computationally. We have theoretically showed that FNO and DeepONet of the same size exhibit the same accuracy when emulating the Cole–Hopf transformation for Burgers' equation. In particular, FNO in its continuous form can be thought of as a subcase of DeepONet with a specially-designed branch network and a discrete trigonometric basis to replace the trunk net. On the computational side, we have performed extensive experiments of 16 PDE problems, and we demonstrated that when proper extensions were employed, both DeepONet and FNO exhibited good accuracy for diverse applications, with similar performance in most problems.

Training neural operators could be very expensive, but the theoretical works in [23–25] have shown that both DeepONet and FNO can break the curse of dimensionality in the input space for solution operators arising from the majority of partial differential equations. We have also demonstrated exponential convergence of the DeepONet error with respect to the size of the training data, see [18], although for larger data sizes the convergence rate switches to algebraic due to the finite size of the neural network architecture. We believe that this is a possible

direction for future work, i.e., to design new architectures that can sustain exponential convergence rate for all data sizes. Moreover, the demand for large datasets could be reduced significantly by incorporating physics-informed learning in the loss function of the neural operators as it was demonstrated in the works of [46,47]. However, as demonstrated in [46], a hybrid physics-data training is more effective for realistic problems of the type considered herein, and simply using the governing equations in the loss as suggested in [47] without using data may lead to erroneous results, e.g., for the compressible Euler equations considered in the current work.

Declaration of competing interest

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

Acknowledgments

This work was supported by DARPA/CompMods HR00112090062, DOE PhILMs project (no. DE-SC0019453), and OSD/AFOSR MURI grant FA9550-20-1-0358, United States of America. Zhang was also partially supported by AFOSR under award number FA9550-20-1-0056, United States of America.

Appendix A. Proof of Theorem 4.3

Proof. By the fact that $\partial_x \mathcal{K}(x, y, t) = -\partial_y \mathcal{K}(x, y, t)$ and integration by parts, we have

$$\begin{aligned}\mathcal{G}(u_0)(\mathbf{x}) - \mathcal{G}_m(\mathbf{u}_{0,m}; \mathbf{x}) &= u(\mathbf{x})\frac{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0 - v_0)(y)dy}{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0)(y)dy} - \frac{2\kappa \int_{\mathbb{R}} \partial_x \mathcal{K}(x, y, t)(\mathcal{I}_m v_0 - v_0)(y)dy}{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0)(y)dy} \\ &= u(\mathbf{x})\frac{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0 - v_0)(y)dy}{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0)(y)dy} - \frac{2\kappa \int_{\mathbb{R}} \mathcal{K}(x, y, t)\partial_y(\mathcal{I}_m v_0 - v_0)(y)dy}{\int_{\mathbb{R}} \mathcal{K}(x, y, t)(\mathcal{I}_m v_0)(y)dy} \\ &=: I_1 + I_2.\end{aligned}$$

Then by the fact that $u_0 \in W^{1,\infty}$ and properties of the heat kernel, we have $u \in W^{1,\infty}$ and also

$$\|I_1\|_{L^2(-\pi,\pi)} \le C \|\mathcal{I}_m v_0 - v_0\|_{L^2(-\pi,\pi)}, \quad \|I_2\|_{L^2(-\pi,\pi)} \le C \|\mathcal{I}_m v_0 - v_0\|_{H^1(-\pi,\pi)}.$$

We observe that

$$\begin{aligned}\left|\partial_x v_0(s)\right| &= \left|v_0(s)\frac{1}{2\kappa}u_0(s)\right| \le \frac{M_0}{2\kappa}\left|v_0(s)\right|, \\ \left|\partial_{xx} v_0(s)\right| &= \left|\left(\frac{u_0^2(s)}{4\kappa^2} - \frac{\partial_x u_0(s)}{2\kappa}\right)v_0(s)\right| \le \left(\frac{M_0^2}{4\kappa^2} + \frac{M_1}{2\kappa}\right)\left|v_0(s)\right|.\end{aligned}$$

By the estimates of the Fourier interpolation error

$$\|\mathcal{I}_m v_0 - v_0\|_{H^1(-\pi,\pi)} \le Ch \left\|\partial_x^2 v_0\right\|_{L^2(-\pi,\pi)} \le C(M_0, M_1, \kappa)h,$$

we then have $\|I_1\|_{L^2(-\pi,\pi)} \le Ch$ and $\|I_2\|_{L^2(-\pi,\pi)} \le Ch$ and whence for some small h (large m)

$$\|\mathcal{G}(u_0)(\cdot, t) - \mathcal{G}_m(\mathbf{u}_{0,m})(\cdot, t)\|_{L^2(-\pi,\pi)} \le Ch.$$

Here $C > 0$ depends on t, M_0, M_1, and κ and C may be very large when κ is small. □

Appendix B. Dataset size

See Table B.1.

Appendix C. Architectures of DeepONets

We list the architectures of DeepONet and POD-DeepONet for each example in Tables C.1 and C.2.

Table B.1
Dataset size for each problem, unless otherwise stated.

	No. of training data	No. of testing data
Section 5.1	1000	200
Section 5.2.1 PWC	1000	200
Section 5.2.1 Cont.	1000	200
Section 5.2.2	1900	100
Section 5.2.3	1900	100
Section 5.2.4	1900	100
Section 5.3 Electroconvection	15	2
Section 5.4.1 Advection I	1000	1000
Section 5.4.1 Advection II/III	1000	200
Section 5.4.2	40800	10000
Section 5.4.3 Euler	100	50
Section 5.5 Airfoil	28	2
Section 5.6 NS	1000	200
Section 5.7 Cavity (Steady)	100	10
Section 5.7 Cavity (Unsteady I)	90	10
Section 5.7 Cavity (Unsteady II)	10	10

Table C.1
DeepONet architecture for each problem, unless otherwise stated.

	Branch net	Trunk net	Activation function
Section 5.1 Burgers	Depth 4 & Width 128	Depth 4 & Width 128	tanh
Section 5.2.1 PWC	CNN	Depth 5 & Width 128	ReLU
Section 5.2.1 Cont.	CNN	[128, 128, 100]	tanh
Section 5.2.2	[128, 128]	[128, 128, 128, 128]	tanh
Section 5.2.3	[128, 128]	[128, 128, 128]	ReLU
Section 5.2.4	[128, 128]	[128, 128, 128]	ReLU
Section 5.3 Electroconvection	[256, 256, 128]	[128, 128, 128]	ReLU
Section 5.4.1 Advection I	Depth 2 & Width 256	Depth 4 & Width 256	ReLU
Section 5.4.1 Advection II/III	Depth 2 & Width 512	Depth 4 & Width 512	ReLU
Section 5.4.2	Depth 6 & Width 200	Depth 7 & Width 200	ELU
Section 5.4.3 Euler	Depth 2 & Width 256	Depth 4 & Width 256	ReLU
Section 5.5 Airfoil	Depth 2 & Width 200	Depth 4 & Width 200	ReLU
Section 5.6 NS	CNN	[128, 128, 64]	tanh
Section 5.7 Cavity	CNN	[128, 128, 128, 100]	tanh

Appendix D. Rescaling effect for POD-DeepONet

We test computationally POD-DeepONet with different scaling factors. The examples of Burgers' equation, advection equation and Navier–Stokes equations are investigated. While the results for Burgers' equation are already given in Table 4, the other results are summarized in Table D.1 here.

Appendix E. Data generation for Darcy flows

We employ the *PDEtoolbox* in Matlab to solve Eq. (5.1) for the Darcy flows in different geometries of Section 5.2. The meshed domain used for data generation is shown in Fig. E.2.

- **Rectangular domain:** The Dirichlet boundary conditions are imposed on the inlet and the outlet, i.e., $h(x = 0, y) = 1$, and $h(x = 1, y) = 0$, and the Neumann boundary conditions are employed on the upper and bottom walls. We employ 1893 unstructured meshes in our simulations, and then interpolate the solution to a 20×20 uniform grid to make it consistent with the generated permeability field. In addition, we obtain 1200 solutions with different permeability fields, and use 1000 of them for training and 200 for testing.
- **Pentagram with a hole:** The Dirichlet boundary conditions are imposed on all boundaries, which are generated from the Gaussian processes in Eq. (5.2). We employ 1938 unstructured meshes in our simulations, and obtain

Table C.2
POD-DeepONet architecture for each problem, unless otherwise stated. The activation functions are the same as those in Table C.1.

	Branch net	No. of POD modes
Section 5.1 Burgers	Depth 3 & Width 128	32
Section 5.2.1 PWC	CNN	115
Section 5.2.1 Cont.	CNN	10
Section 5.2.2	[64, 64]	8
Section 5.2.3	Depth 3 & Width 128	32
Section 5.2.4	CNN	20
Section 5.3 Electroconvection	Depth 3 & Width 256	12
Section 5.4.1 I	Depth 2 & Width 256	38
Section 5.4.1 II	Depth 2 & Width 512	38
Section 5.4.1 III	Depth 2 & Width 512	32
Section 5.4.2	Depth 6 & Width 256	9
Section 5.4.3 Euler	Depth 2 & Width 256	16
Section 5.6 NS	CNN	29
Section 5.7	CNN	6

Table D.1
L^2 relative error for POD-DeepONet with different rescaling factors. Here, p is the number of outputs of the branch and trunk nets.

	Section 5.4.1 Advection (I)	Section 5.6 Navier–Stokes
POD-DeepONet (w/o rescaling)	0.215 ± 0.027%	3.37 ± 0.22%
POD-DeepONet (rescaling by $1/\sqrt{p}$)	0.078 ± 0.009%	**1.36 ± 0.03%**
POD-DeepONet (rescaling by $1/p$)	**0.041 ± 0.003%**	1.71 ± 0.03%
POD-DeepONet (rescaling by $1/p^{1.5}$)	0.042 ± 0.002%	5.35 ± 0.17%

solutions for 2000 different boundary conditions. We then use 1900 of them for training the operator networks, and utilize the remaining for testing.

- **Triangular domain:** We utilize the same Gaussian processes in Eq. (5.2) to generate the Dirichlet boundary conditions for all boundaries here, and then employ 861 unstructured meshes in our simulations. Similarly, we generate 2000 solutions with different boundary conditions, and use 1900 of them for training and 100 for testing.
- **Triangular domain with a notch:** We employ 1084 unstructured meshes in our simulations to generate 2000 solutions with different boundary conditions. From the generated datasets, 1900 samples are used for training, while the remaining 100 are employed as testing samples.

Appendix F. Data generation for regularized cavity flow

For both the steady and unsteady cases, we employ the multi-relaxation-time lattice Boltzmann equation model (MRT-LBE) [59] with a $N \times N = 256 \times 256$ uniform grid in our simulations. In addition, the non-equilibrium extrapolation [60] is employed for all boundary conditions. The time step is set as $dt = dx = L/N$.

For the steady case (Case A), we terminate the iteration as the following criterion is met:

$$E = \frac{\sum |\boldsymbol{u}_{T+1000} - \boldsymbol{u}_T|}{\sum \boldsymbol{u}_T} \leq 10^{-6},$$

where T denotes the number of iterations. For the unsteady case (Case B), we set the maximum iteration number as 1,500,000, and use the velocity fields in the last 10,000 iterations as the training/testing dataset in the DeepONet/FNO.

Appendix G. Computational cost

We have conducted a comparative study on the computational cost as well as the GPU memory usage for training different networks using the examples in Sections 5.7 and 5.2.2. For the problem in Section 5.2.2, the batch size is

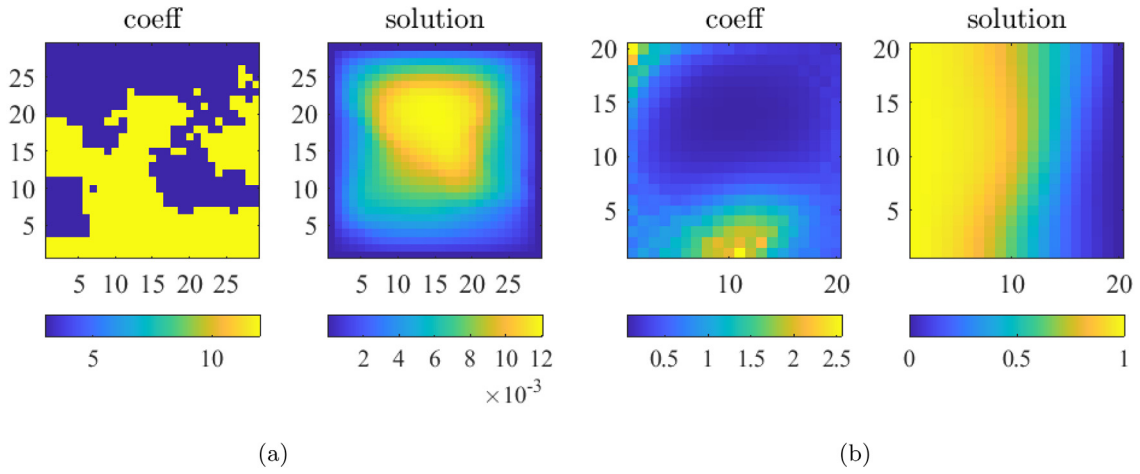

Fig. E.1. Two datasets of Darcy flow. (a) Data from [22]. (b) Newly-generated data.

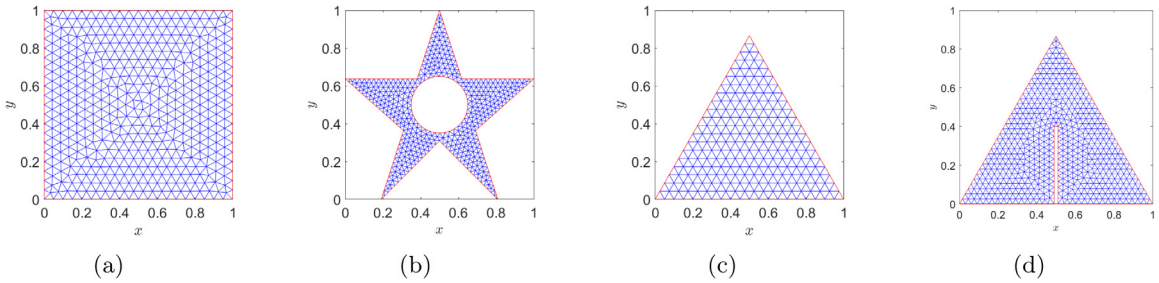

Fig. E.2. Darcy flows: Unstructured meshes for solving Eq. (5.1) in different geometries.

Table G.1
Computational cost (second) of one iteration for training different networks.

	dgFNO+/dFNO+	DeepONet (vanilla)	DeepONet	POD-DeepONet
Section 5.2.2 Darcy (Pentagram)	0.0566	3.4862	0.0250	0.0085
Section 5.7 Cavity (case B)	0.0135	0.9072	0.0085	0.0048

Table G.2
GPU memory usage (MiB) for training different networks.

	dgFNO+/dFNO+	DeepONet (vanilla)	DeepONet	POD-DeepONet
Section 5.2.2 Darcy (Pentagram)	1485	3311	1324	289
Section 5.7 Cavity (case B)	2183	1441	541	285

100, and we keep the first 32 and 8 modes in the dgFNO+ and POD-DeepONet, respectively. While for the problem in Section 5.7, the batch size is 10 and we keep 64 and 5 modes in the dFNO+ and POD-DeepONet, respectively. All computations are performed on a workstation with 2 CPUs (Intel Xeon E5-2643) and a GPU (NVIDIA TITAN Xp). We list the computational time of one iteration for all the cases in Table G.1 and the GPU memory usage in Table G.2.

References

[1] T.J. Sejnowski, The unreasonable effectiveness of deep learning in artificial intelligence, Proc. Natl. Acad. Sci. 117 (48) (2020) 30033–30038.

[2] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (6) (2021) 422–440.
[3] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017, arXiv preprint arXiv:1711.10561.
[4] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations, 2017, arXiv preprint arXiv:1711.10566.
[5] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, DeepXDE: A deep learning library for solving differential equations, SIAM Rev. 63 (1) (2021) 208–228.
[6] G. Pang, L. Lu, G.E. Karniadakis, FPINNs: Fractional physics-informed neural networks, SIAM J. Sci. Comput. 41 (4) (2019) A2603–A2626.
[7] D. Zhang, L. Lu, L. Guo, G.E. Karniadakis, Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems, J. Comput. Phys. 397 (2019) 108850.
[8] D. Zhang, L. Guo, G.E. Karniadakis, Learning in modal space: Solving time-dependent stochastic PDEs using physics-informed neural networks, SIAM J. Sci. Comput. 42 (2) (2020) A639–A665.
[9] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, S.G. Johnson, Physics-informed neural networks with hard constraints for inverse design, SIAM J. Sci. Comput. 43 (6) (2021) B1105–B1132.
[10] X. Meng, Z. Li, D. Zhang, G.E. Karniadakis, PPINN: Parareal physics-informed neural network for time-dependent PDEs, Comput. Methods Appl. Mech. Engrg. 370 (2020) 113250.
[11] A.D. Jagtap, G.E. Karniadakis, Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations, Commun. Comput. Phys. 28 (5) (2020) 2002–2041.
[12] J. Yu, L. Lu, X. Meng, G.E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems, 2021, arXiv preprint arXiv:2111.02801.
[13] S. Cai, Z. Mao, Z. Wang, M. Yin, G.E. Karniadakis, Physics-informed neural networks (PINNs) for fluid mechanics: A review, 2021, arXiv preprint arXiv:2105.09506.
[14] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Systems 2 (4) (1989) 303–314.
[15] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366.
[16] T. Chen, H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems, IEEE Trans. Neural Netw. 6 (4) (1995) 911–917.
[17] L. Lu, P. Jin, G.E. Karniadakis, DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, 2019, arXiv preprint arXiv:1910.03193.
[18] L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators, Nat. Mach. Intell. 3 (3) (2021) 218–229.
[19] I. Higgins, Generalizing universal function approximators, Nat. Mach. Intell. 3 (3) (2021) 192–193.
[20] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Neural operator: Graph kernel network for partial differential equations, 2020, arXiv preprint arXiv:2003.03485.
[21] H. You, Y. Yu, M. DÉlia, T. Gao, S. Silling, Nonlocal kernel network (NKN): a stable and resolution-independent deep neural network, Preprint (2021).
[22] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, 2020, arXiv preprint arXiv:2010.08895.
[23] N. Kovachki, S. Lanthaler, S. Mishra, On universal approximation and error bounds for Fourier neural operators, 2021, arXiv preprint arXiv:2107.07562.
[24] S. Lanthaler, S. Mishra, G.E. Karniadakis, Error estimates for DeepONets: A deep learning framework in infinite dimensions, 2021, arXiv preprint arXiv:2102.09618.
[25] B. Deng, Y. Shin, L. Lu, Z. Zhang, G.E. Karniadakis, Convergence rate of DeepONets for learning operators arising from advection-diffusion equations, 2021, arXiv preprint arXiv:2102.10621.
[26] A. Yu, C. Becquey, D. Halikias, M.E. Mallory, A. Townsend, Arbitrary-depth universal approximation theorems for operator neural networks, 2021, arXiv preprint arXiv:2109.11354.
[27] C. Marcati, C. Schwab, Exponential Convergence of Deep Operator Networks for Elliptic Partial Differential Equations, Tech. Rep. 2021–42, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2021.
[28] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, A. Anandkumar, Neural operator: Learning maps between function spaces, 2021, arXiv preprint arXiv:2108.08481.
[29] K. Bhattacharya, B. Hosseini, N.B. Kovachki, A.M. Stuart, Model reduction and neural networks for parametric PDEs, 2020, arXiv preprint arXiv:2005.03180.
[30] N. Trask, R.G. Patel, B.J. Gross, P.J. Atzberger, GMLS-Nets: A framework for learning from unstructured data, 2019, arXiv preprint arXiv:1909.05371.
[31] H. You, Y. Yu, N. Trask, M. Gulian, M. D'Elia, Data-driven learning of nonlocal physics from high-fidelity synthetic data, Comput. Methods Appl. Mech. Engrg. 374 (2021) 113553.
[32] R.G. Patel, N.A. Trask, M.A. Wood, E.C. Cyr, A physics-informed operator regression framework for extracting data-driven continuum models, Comput. Methods Appl. Mech. Engrg. 373 (2021) 113500.
[33] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L.B. da Silva Santos, P.E. Bourne, et al., The FAIR guiding principles for scientific data management and stewardship, Sci. Data 3 (1) (2016) 1–9.

[34] H. Maupin, US Army Research Laboratory South Partnership Summit 2018, Tech. Rep., Army Research Lab Aberdeen Proving Ground United States, 2019.
[35] X. Meng, G.E. Karniadakis, A composite neural network that learns from multi-fidelity data: Application to function approximation and inverse PDE problems, J. Comput. Phys. 401 (2020) 109020.
[36] X. Meng, H. Babaee, G.E. Karniadakis, Multi-fidelity Bayesian neural networks: Algorithms and applications, J. Comput. Phys. 438 (2021) 110361.
[37] L. Lu, M. Dao, P. Kumar, U. Ramamurty, G.E. Karniadakis, S. Suresh, Extraction of mechanical properties of materials through deep learning from instrumented indentation, Proc. Natl. Acad. Sci. 117 (13) (2020) 7052–7062.
[38] L. Yang, X. Meng, G.E. Karniadakis, B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data, J. Comput. Phys. 425 (2021) 109913.
[39] A. Olivier, M.D. Shields, L. Graham-Brady, BayesIan neural networks for uncertainty quantification in data-driven materials modeling, Comput. Methods Appl. Mech. Engrg. 386 (2021) 114079.
[40] X. Meng, L. Yang, Z. Mao, J.d.Á. Ferrandis, G.E. Karniadakis, Learning functional priors and posteriors from data and physics, J. Comput. Phys. 457 (2022) 111073.
[41] C. Lin, Z. Li, L. Lu, S. Cai, M. Maxey, G.E. Karniadakis, Operator learning for predicting multiscale bubble growth dynamics, J. Chem. Phys. 154 (10) (2021) 104118.
[42] S. Cai, Z. Wang, L. Lu, T.A. Zaki, G.E. Karniadakis, DeepM&Mnet: Inferring the electroconvection multiphysics fields based on operator approximation by neural networks, J. Comput. Phys. 436 (2021) 110296.
[43] P.C. Di Leoni, L. Lu, C. Meneveau, G. Karniadakis, T.A. Zaki, DeepONet prediction of linear instability waves in high-speed boundary layers, 2021, arXiv preprint arXiv:2105.08697.
[44] Z. Mao, L. Lu, O. Marxen, T.A. Zaki, G.E. Karniadakis, DeepM&Mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators, J. Comput. Phys. 447 (2021) 110698.
[45] C. Lin, M. Maxey, Z. Li, G.E. Karniadakis, A seamless multiscale operator neural network for inferring bubble dynamics, J. Fluid Mech. 929 (2021).
[46] S. Goswami, M. Yin, Y. Yu, G.E. Karniadakis, A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials, Comput. Methods Appl. Mech. Engrg. 391 (2022) 114587.
[47] S. Wang, H. Wang, P. Perdikaris, Learning the solution operator of parametric partial differential equations with physics-informed DeepONets, 2021, arXiv preprint arXiv:2103.10974.
[48] M. Yin, E. Ban, B.V. Rego, E. Zhang, C. Cavinato, J.D. Humphrey, G.E. Karniadakis, Simulating progressive intramural damage leading to aortic dissection using an operator-regression neural network, 2021, arXiv preprint arXiv:2108.11985.
[49] A. Yazdani, L. Lu, M. Raissi, G.E. Karniadakis, Systems biology informed deep learning for inferring parameters and hidden dynamics, PLoS Comput. Biol. 16 (11) (2020) e1007575.
[50] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, J. Comput. Phys. 435 (2021) 110242.
[51] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
[52] L. Lu, Y. Shin, Y. Su, G. E. Karniadakis, Dying ReLU and initialization: Theory and numerical examples, Commun. Comput. Phys. 28 (5) (2020) 1671–1706.
[53] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1026–1034.
[54] M. Telgarsky, Neural networks and rational functions, in: 34th International Conference on Machine Learning, ICML 2017, International Machine Learning Society (IMLS), 2017, pp. 5195–5210.
[55] J.A.A. Opschoor, C. Schwab, J. Zech, Exponential reLU DNN expression of holomorphic maps in high dimension, Constr. Approx. (2021).
[56] W. Wang, C.-W. Shu, H.C. Yee, B. Sjögreen, High-order well-balanced schemes and applications to non-equilibrium flow, J. Comput. Phys. 228 (18) (2009) 6682–6702.
[57] X. Zhang, C.-W. Shu, Positivity-preserving high order discontinuous Galerkin schemes for compressible Euler equations with source terms, J. Comput. Phys. 230 (4) (2011) 1238–1248.
[58] P.A. Gnoffo, R.N. Gupta, J.L. Shinn, Conservation equations and physical models' for hypersonic air flows in thermal and chemical nonequilibrium, 1989, NASA Technical Paper 2869.
[59] X. Meng, Z. Guo, Multiple-relaxation-time lattice Boltzmann model for incompressible miscible flow with large viscosity ratio and high Péclet number, Phys. Rev. E 92 (4) (2015) 043305.
[60] Z. Guo, C. Zheng, B. Shi, Non-equilibrium extrapolation method for velocity and pressure boundary conditions in the lattice Boltzmann method, Chin. Phys. 11 (4) (2002) 366.