Operator Learning Part II: DeepONet and Physical-Informed Operator Learning

Presenter: Shibo Li, March 2nd 2022

References

DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators

Lu Lu[1], Pengzhan Jin[2], and George Em Karniadakis[1]

LEARNING THE SOLUTION OPERATOR OF PARAMETRIC PARTIAL DIFFERENTIAL EQUATIONS WITH PHYSICS-INFORMED DEEPONETS

Sifan Wang
Graduate Group in Applied Mathematics
and Computational Science
University of Pennsylvania
Philadelphia, PA 19104
sifanw@sas.upenn.edu

Hanwen Wang
Graduate Group in Applied Mathematics
and Computational Science
University of Pennsylvania
Philadelphia, PA 19104
wangh19@sas.upenn.edu

Paris Perdikaris
Department of Mechanichal Engineering
and Applied Mechanics
University of Pennsylvania
Philadelphia, PA 19104
pgp@seas.upenn.edu

Physics-Informed Neural Operator for Learning Partial Differential Equations

Zongyi Li*, Hongkai Zheng*, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, Anima Anandkumar

November 9, 2021

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*
[b] *Division of Applied Mathematics, Brown University*
[c] *School of Mathematical Sciences, Xiamen University*
[d] *Department of Mathematical Sciences, Worcester Polytechnic Institute*
[e] *School of Engineering, Brown University*

Outline

- Recap of Operator Learning
- Motivation
- Methods
- Physics-Informed DeepONet
- DeepONet vs. Neural Operator
- Physical Informed Neural Operator

Previous on Operator Learning

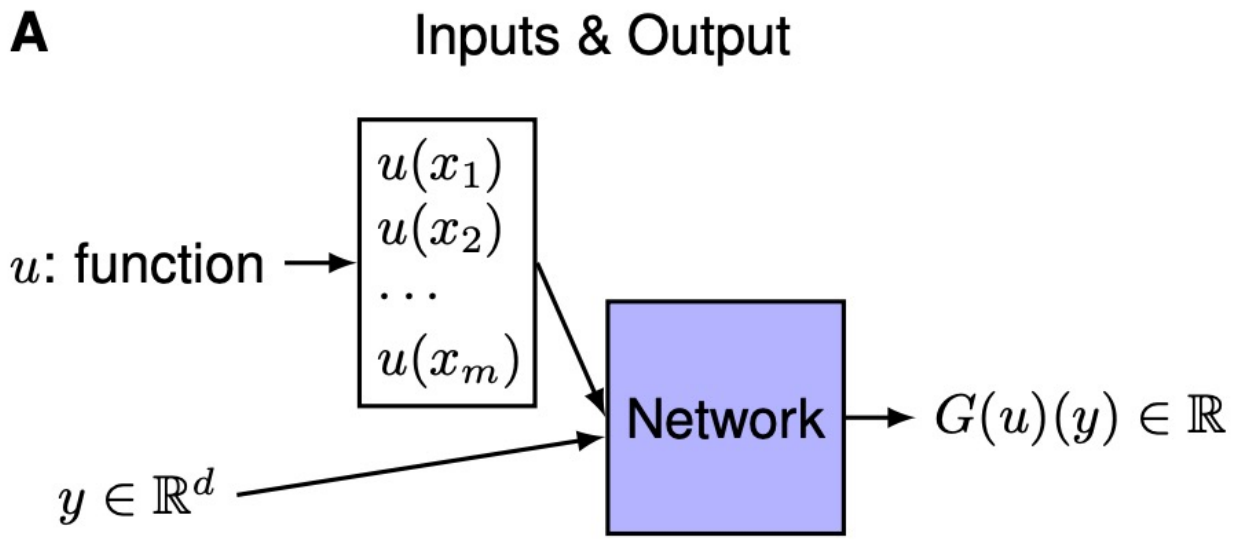
Introduction

- Well known:
  - Neural Networks can be used to approximate any continuous function to arbitrary accuracy if no constraint is placed on with width and depth of the hidden layers (G. Cybenko et al. 1989, K Hornik et al. 1989)
- Yet more supervising and has not been appreciated so far
  - A neural network with a single hidden layer can approximate accurately any nonlinear continuous functional or (nonlinear) operator

Introduction

- Notations:

Let G be an operator taking an input function u, and then $G(u)$ is the corresponding output function. For any point y in the domain of $G(u)$, the output $G(u)(y)$ is a real number. Hence, the network takes inputs composed of two parts: u and y, and outputs $G(u)(y)$ (Fig. 1A). Although our goal is to learn operators, which take a function as the input, we have to represent the input functions discretely, so that network approximations can be applied.



Universal Approximation Theorem for Operator

- (Chen & Chen 1995)

Theorem 1 (Universal Approximation Theorem for Operator). *Suppose that σ is a continuous non-polynomial function, 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 $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \ldots, n$, $k = 1, \ldots, p$, $j = 1, \ldots, m$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{branch} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{trunk} \right| < \epsilon \tag{1}$$

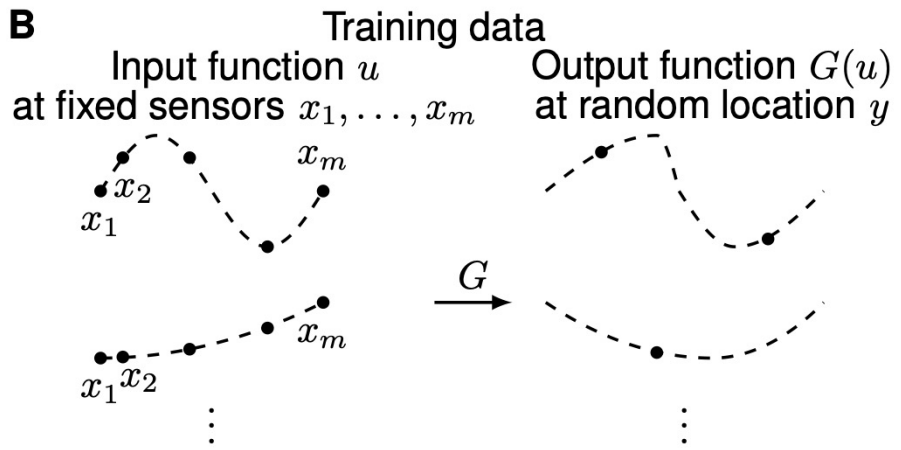
holds for all $u \in V$ and $y \in K_2$.

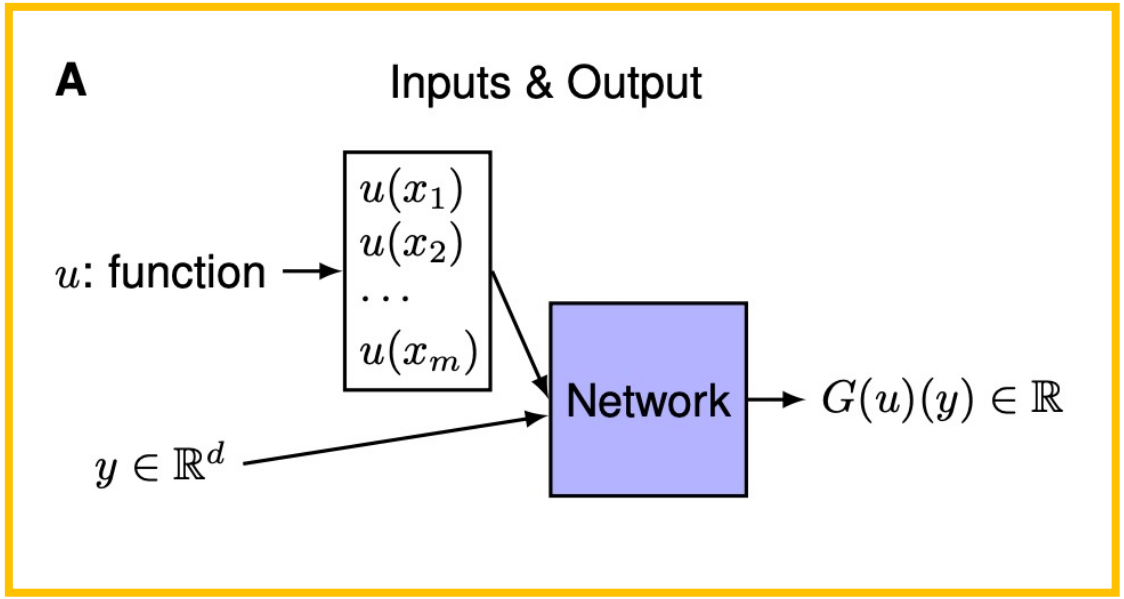
Indicates the potential application of neural networks to learn nonlinear operator from data

DeepONet

We focus on learning operators in a more general setting, where the only requirement for the training dataset is the consistency of the sensors $\{x_1, x_2, \ldots, x_m\}$ for input functions. In this general setting, the network inputs consist of two separate components: $[u(x_1), u(x_2), \ldots, u(x_m)]^T$ and y

???

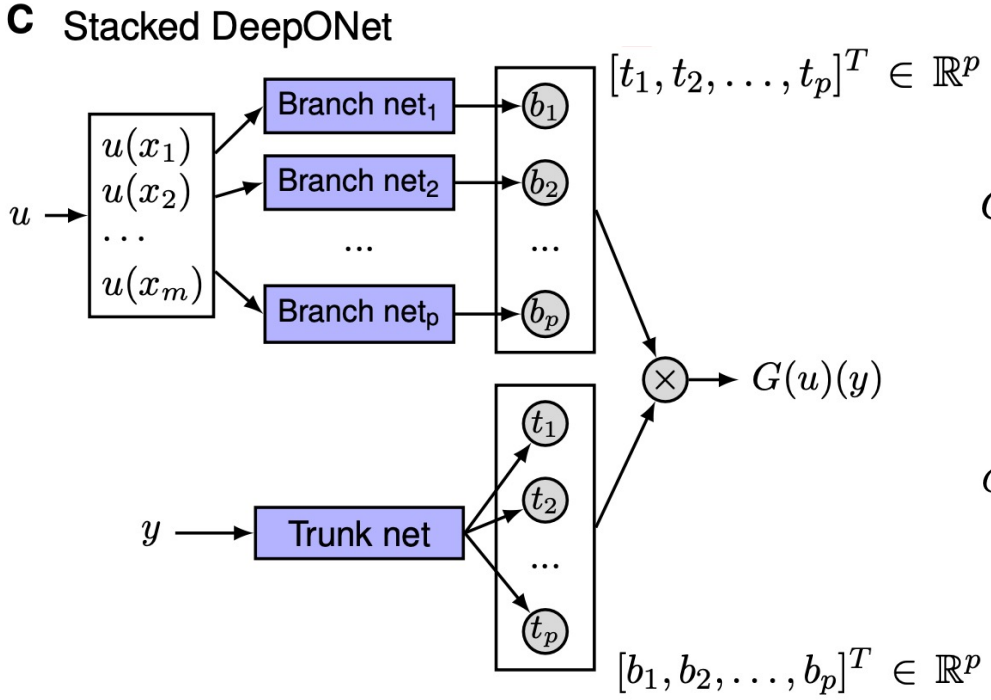




- Lost the input structure
- Cannot handle high dimensional output space

DeepONet

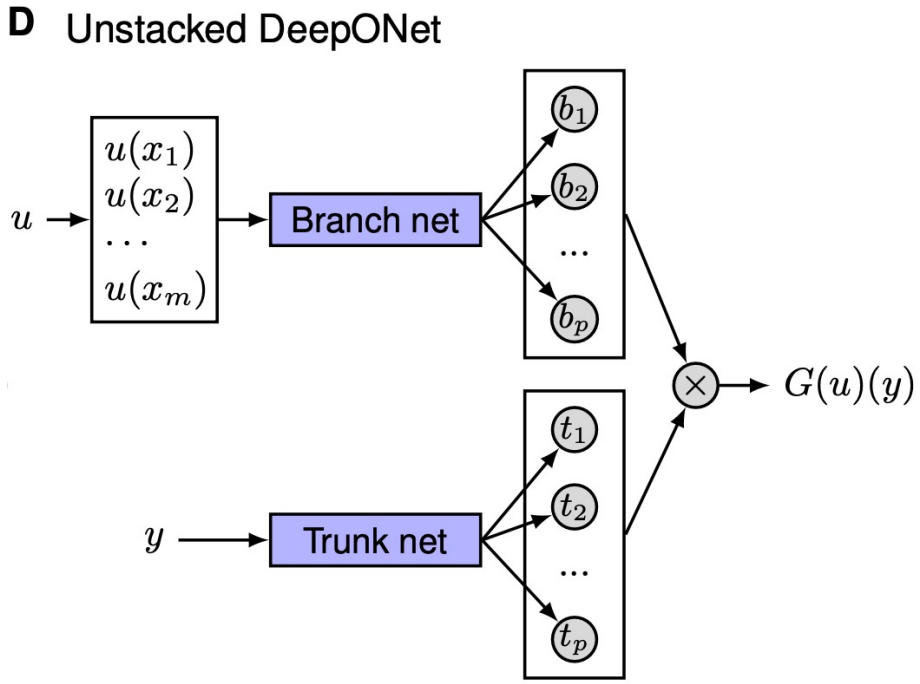
- **Key Idea**: Handle them separately into
  - *Branch Net*: takes the input function(s)
  - *Trunk Net*: takes the input of the output function(s)



$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k.$$

OR

$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k + b_0.$$



Theoretical Analysis of Number of Sensors

Suppose that the dynamic system is subject to the following ODE system:

$$\begin{cases} \frac{d}{dx}\boldsymbol{s}(x) = \boldsymbol{g}(\boldsymbol{s}(x), u(x), x) \\ \boldsymbol{s}(a) = \boldsymbol{s_0} \end{cases},$$

where $u \in V$(a compact subset of $C[a,b]$) is the input signal, and $\boldsymbol{s} : [a,b] \to \mathbb{R}^K$ is the solution of system (3) serving as the output signal.

Let G be the operator mapping the input u to the output $\boldsymbol{s}$, i.e., Gu satisfies

$$(Gu)(x) = \boldsymbol{s_0} + \int_a^x \boldsymbol{g}((Gu)(t), u(t), t)dt.$$

Now, we choose uniformly $m+1$ points $x_j = a + j(b-a)/m, j = 0, 1, \cdots, m$ from $[a,b]$, and define the function $u_m(x)$ as follows:

$$u_m(x) = u(x_j) + \frac{u(x_{j+1}) - u(x_j)}{x_{j+1} - x_j}(x - x_j),\ x_j \le x \le x_{j+1},\ j = 0, 1, \cdots, m-1.$$

When V is GRF with RBF kernel, we have $\kappa(m, V) \sim \frac{1}{m^2 l^2}$, see Appendix C for the proof. Based on the these concepts, we have the following theorem.

Theorem 2. *Suppose that m is a positive integer making $c(b-a)\kappa(m,V)e^{c(b-a)}$ less than ε, then for any $d \in [a,b]$, there exist $\mathcal{W}_1 \in \mathbb{R}^{n\times(m+1)}, b_1 \in \mathbb{R}^{m+1}, \mathcal{W}_2 \in \mathbb{R}^{K\times n}, b_2 \in \mathbb{R}^K$, such that*

$$\|(Gu)(d) - (\mathcal{W}_2 \cdot \sigma(\mathcal{W}_1 \cdot [u(x_0) \quad \cdots \quad u(x_m)]^T + b_1) + b_2)\|_2 < \varepsilon$$

holds for all $u \in V$.

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

Experiments of DeepONet

- Default Settings

Table 1: **Default parameters for each problem, unless otherwise stated.** We note that one data point is a triplet $(u, y, G(u)(y))$, and thus one specific input u may generate multiple data points with different values of y.

	Case	u space	# Sensors m	# Training	# Test	# Iterations	Other parameters
Linear ODE	4.1.1	GRF ($l = 0.2$)	100	10000	100000	50000	
Nonlinear ODE	4.1.2	GRF ($l = 0.2$)	100	10000	100000	100000	
Practical ODE: Pendulum	4.2	GRF ($l = 0.2$)	100	10000	100000	100000	$k = 1, T = 1$
Practical PDE: Diff-React	4.3	GRF ($l = 0.2$)	100		1000000	500000	

Table 2: **DeepONet size for each problem, unless otherwise stated.**

Case	Network type	Trunk depth	Trunk width	Branch depth	Branch width
4.1	Stacked/Unstacked	3	40	2	40
4.2	Unstacked	3	40	2	40
4.3	Unstacked	3	100	2	100

Experiments of DeepONet

A 1D dynamic system is described by

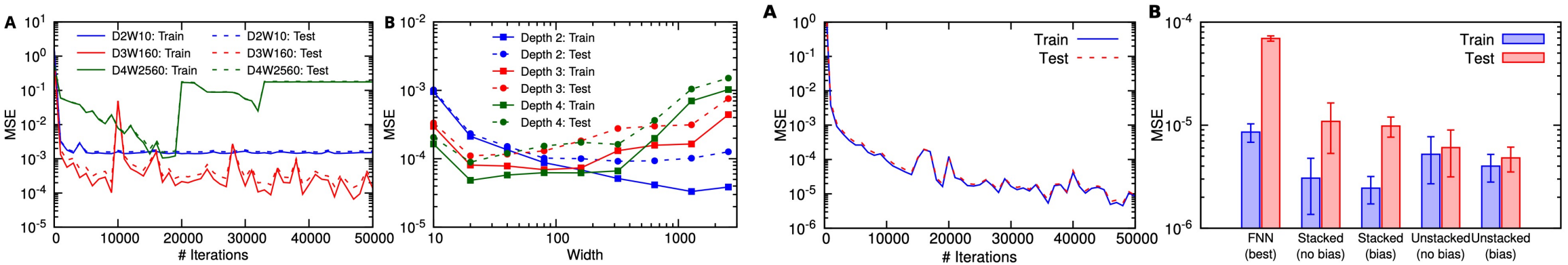
$$\frac{ds(x)}{dx} = g(s(x), u(x), x), \quad x \in [0, 1],$$

with an initial condition $s(0) = 0$. Our goal is to predict $s(x)$ over the whole domain $[0, 1]$ for any $u(x)$.

Linear case: $g(s(x), u(x), x) = u(x)$

We first consider a linear problem by choosing $g(s(x), u(x), x) = u(x)$, which is equivalent to learning the antiderivative operator

$$G : u(x) \mapsto s(x) = \int_0^x u(\tau) d\tau.$$



FNN

DeepONet

Physics-Informed DeepONet

- **Key Idea**: (Partially) known physical laws can be incorporated to achieve (reduced) zero training examples required (just like what PINN does).
- When DeepONet fails

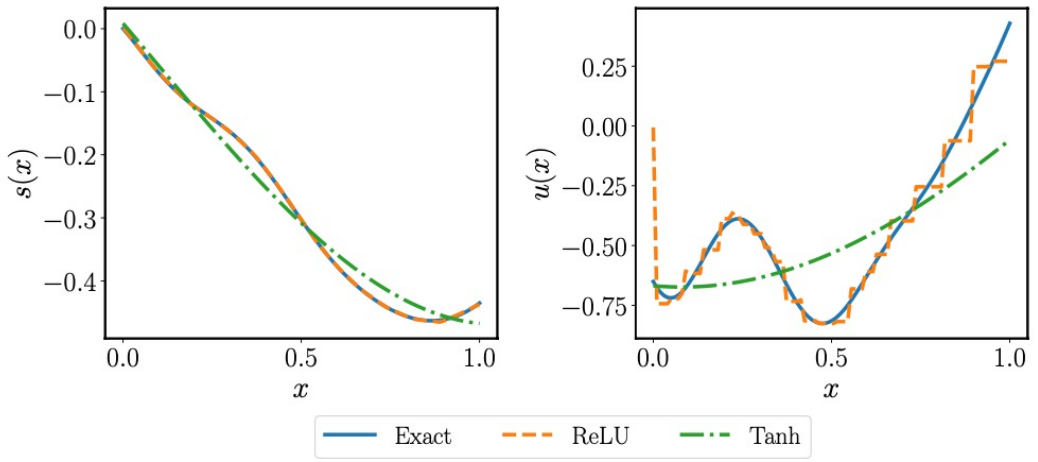
a simple initial value problem

$$\frac{ds(x)}{dx} = u(x), \quad x \in [0, 1],$$

with an initial condition $s(0) = 0$. Here, our goal is to learn the anti-derivative operator

$$G : u(x) \longrightarrow s(x) = s(0) + \int_0^x u(t)dt, \quad x \in [0, 1].$$

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left| G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(y^{(i)}) - s^{(i)}(y^{(i)}) \right|^2,$$



Simply minimizing the operator loss could be problematic

Physics-Informed DeepONet

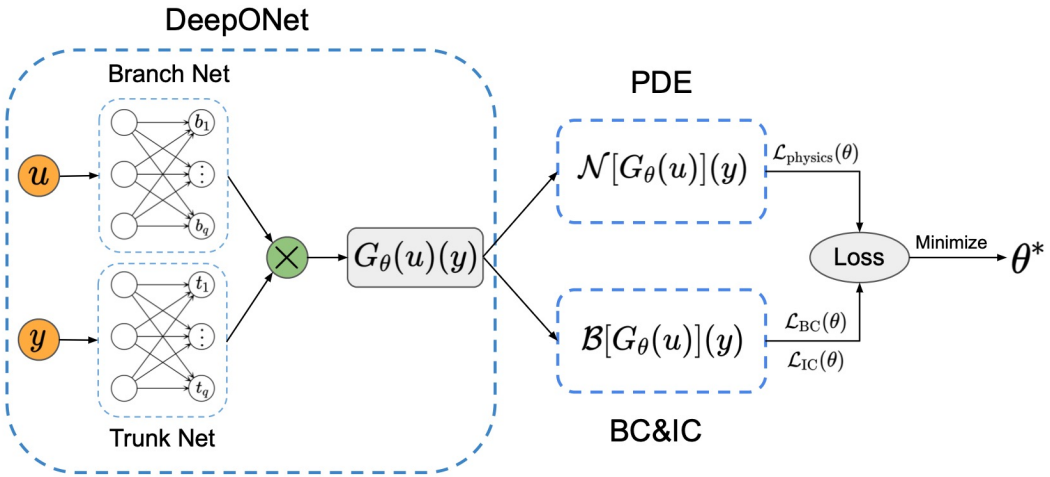
we propose a novel model class referred to as "physics-informed" DeepONets that enables the DeepONet output functions to be consistent with physical constraints via minimizing the residual of the underlying governing laws in the same manner as PINNs. Specifically, we consider minimizing the following composite loss function

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{\text{operator}}(\boldsymbol{\theta}) + \mathcal{L}_{\text{physics}}(\boldsymbol{\theta}), \tag{3.1}$$

where $\mathcal{L}_{\text{operator}}(\boldsymbol{\theta})$ is defined exactly the same as in equation 2.4 and

$$\mathcal{L}_{\text{physics}}(\boldsymbol{\theta}) = \frac{1}{NQm} \sum_{i=1}^{N} \sum_{j=1}^{Q} \sum_{k=1}^{m} \left\| \mathcal{N}(u^{(i)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_j^{(i)})) \right\|^2, \tag{3.2}$$

where $\{\boldsymbol{y}_j^{(i)}\}_{i=1}^{Q}$ denotes a set of collocation points that are randomly sampled from the domain of $G(\boldsymbol{u}^{(i)})$, and used to approximately enforce a set of given physical constraints, typically described by systems of PDEs.



let $(\mathcal{U}, \mathcal{V}, \mathcal{S})$ be a triplet of Banach spaces and $\mathcal{N} : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ be a linear or nonlinear differential operator. We consider a parametric PDEs taking the form

$$\mathcal{N}(\boldsymbol{u}, \boldsymbol{s}) = 0, \tag{2.1}$$

where $\boldsymbol{u} \in \mathcal{U}$ denotes the parameters (i.e. input functions), and $\boldsymbol{s} \in \mathcal{S}$ is the corresponding unknown solutions of the PDE system.

Physics-Informed DeepONet

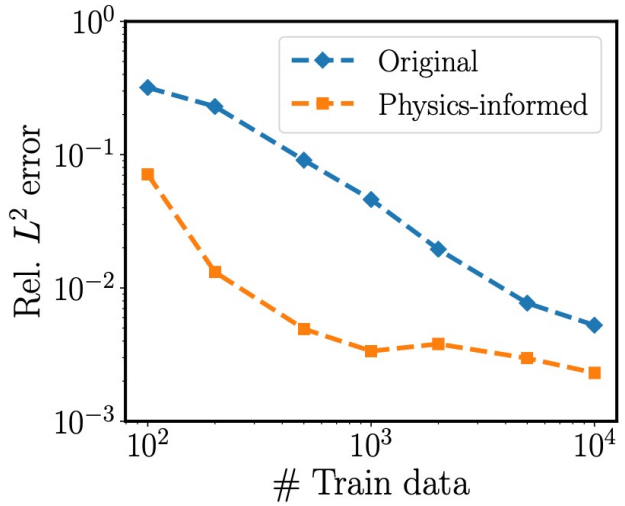
a simple initial value problem

$$\frac{ds(x)}{dx} = u(x), \quad x \in [0, 1], \tag{2.6}$$

with an initial condition $s(0) = 0$. Here, our goal is to learn the anti-derivative operator

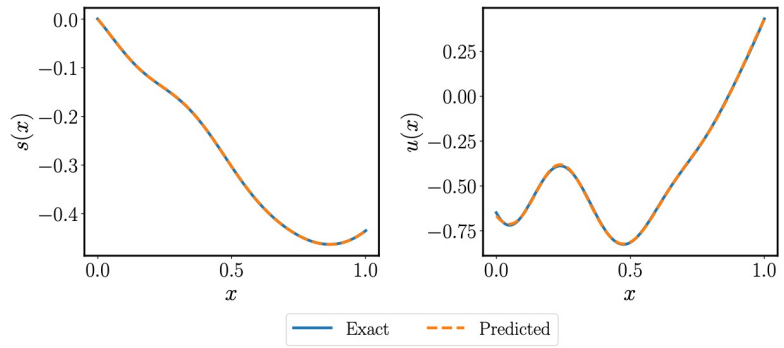
$$G : u(x) \longrightarrow s(x) = s(0) + \int_0^x u(t)dt, \quad x \in [0, 1]. \tag{2.7}$$

$$\mathcal{L}_{\text{physics}}(\boldsymbol{\theta}) = \frac{1}{Nm} \sum_{i=1}^{N} \sum_{j=1}^{m} \left| \frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(y)}{dy} \Big|_{y=x_j} - u^{(i)}(x_j) \right|^2$$



Model \ Relative L^2 error	Relative L^2 error of s	Relative L^2 error of u
DeepONet (ReLU)	$5.16e-03 \pm 4.58e-03$	$1.39e-01 \pm 5.58e-02$
DeepONet (Tanh)	$1.89e-01 \pm 1.51e-01$	$6.14e-01 \pm 2.36e-01$
Physics-informed DeepONet (Tanh)	$2.49e-03 \pm 2.74e-03$	$6.29e-03 \pm 3.65e-03$

Table 1: *Learning anti-derivative operator:* Mean and standard deviation of relative L^2 prediction errors of DeepONet and physics-informed DeepONet equipped with ReLU or Tanh activations over 1,000 examples in the test data-set.



Burgers Equation

To highlight the ability of the proposed framework to handle nonlinearity in the governing PDEs, we consider the 1D Burgers' equation benchmark investigated in Li *et. al.* [32]

$$\frac{ds}{dt} + s\frac{ds}{dx} - \nu\frac{d^2s}{dx^2} = 0, \quad (x,t) \in (0,1) \times (0,1], \tag{4.10}$$

$$s(x,0) = u(x), \quad x \in (0,1), \tag{4.11}$$

with periodic boundary conditions

$$s(0,t) = s(1,t), \tag{4.12}$$

$$\frac{ds}{dx}(0,t) = \frac{ds}{dx}(1,t), \tag{4.13}$$

where $t \in (0,1)$, the viscosity is set to $\nu = 0.01$, and the initial condition $u(x)$ is generated from a GRF $\sim \mathcal{N}\left(0, 25^2(-\Delta + 5^2 I)^{-4}\right)$ satisfying the periodic boundary conditions. Our goal here is to use the proposed physics-informed DeepONet model to learn the solution operator mapping initial conditions $u(x)$ to the full spatio-temporal solution $s(x,t)$ of the 1D Burgers' equation.

Suppose that the solution operator is approximated by a physics-informed DeepONet $G_{\boldsymbol{\theta}}$. For a specific input function $\boldsymbol{u}^{(i)}$, the PDE residual is defined by

$$R_{\boldsymbol{\theta}}^{(i)}(x,t) = \frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dt} + G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)\frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dx} - \nu\frac{d^2G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dx^2}, \tag{4.14}$$

Burgers Equation

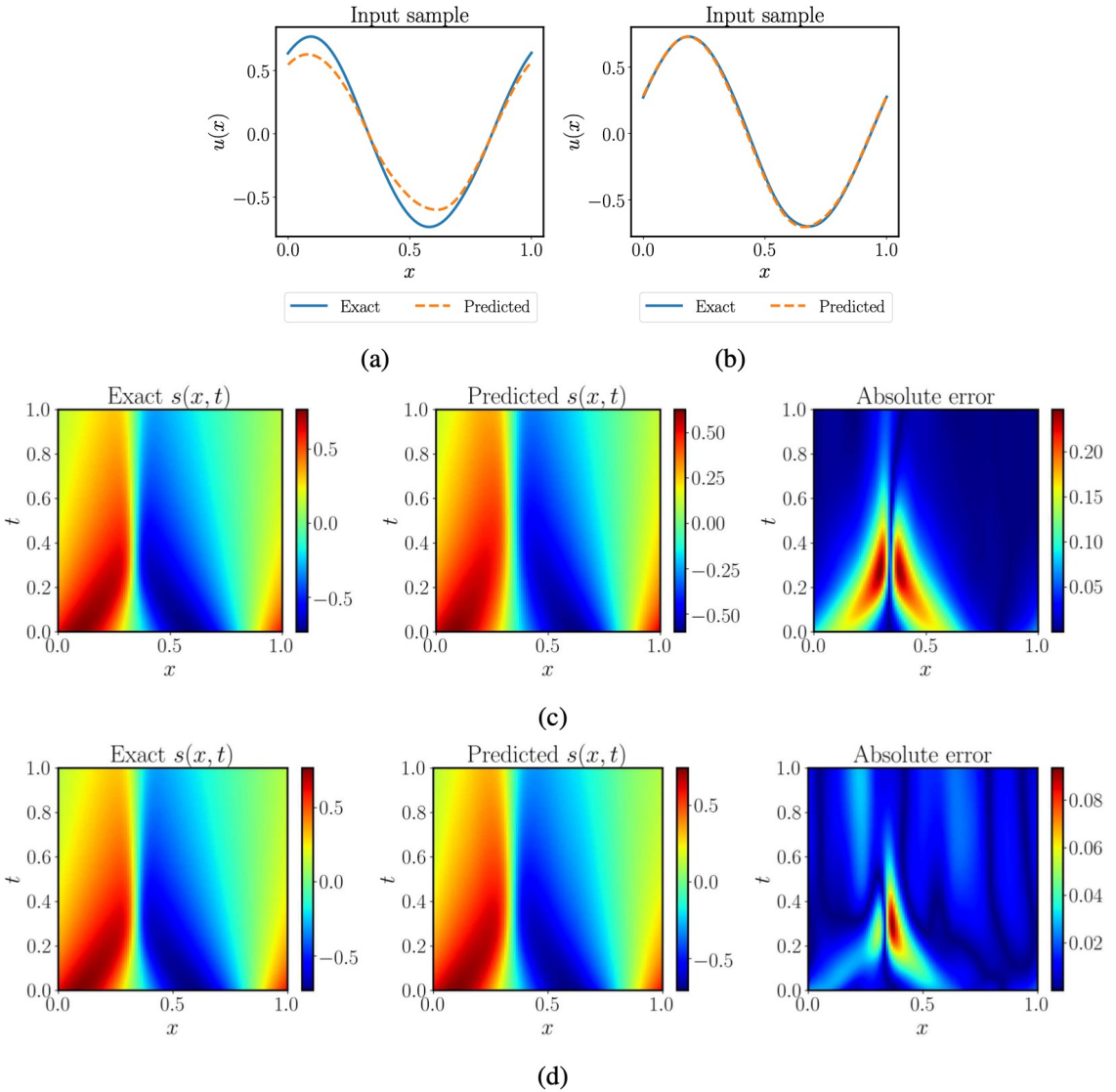
$$\mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{ic,j}^{(i)},0) - u^{(i)}(x_{ic,j}^{(i)})\right|^2$$

$$\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(0,t_{bc,j}^{(i)}) - G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(1,t_{bc,j}^{(i)})\right|^2$$

$$+ \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|\frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dx}\bigg|_{(0,t_{bc,j}^{(i)})} - \frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dx}\bigg|_{(1,t_{bc,j}^{(i)})}\right|^2$$

$$\mathcal{L}_{\text{physics}}(\boldsymbol{\theta}) = \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|R_{\boldsymbol{\theta}}^{(i)}(x_{r,j}^{(i)},t_{r,j}^{(i)})\right|^2.$$

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) + \mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) + \mathcal{L}_{\text{physics}}(\boldsymbol{\theta})$$



DeepONet

Neural Operator

New Developments

- 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.

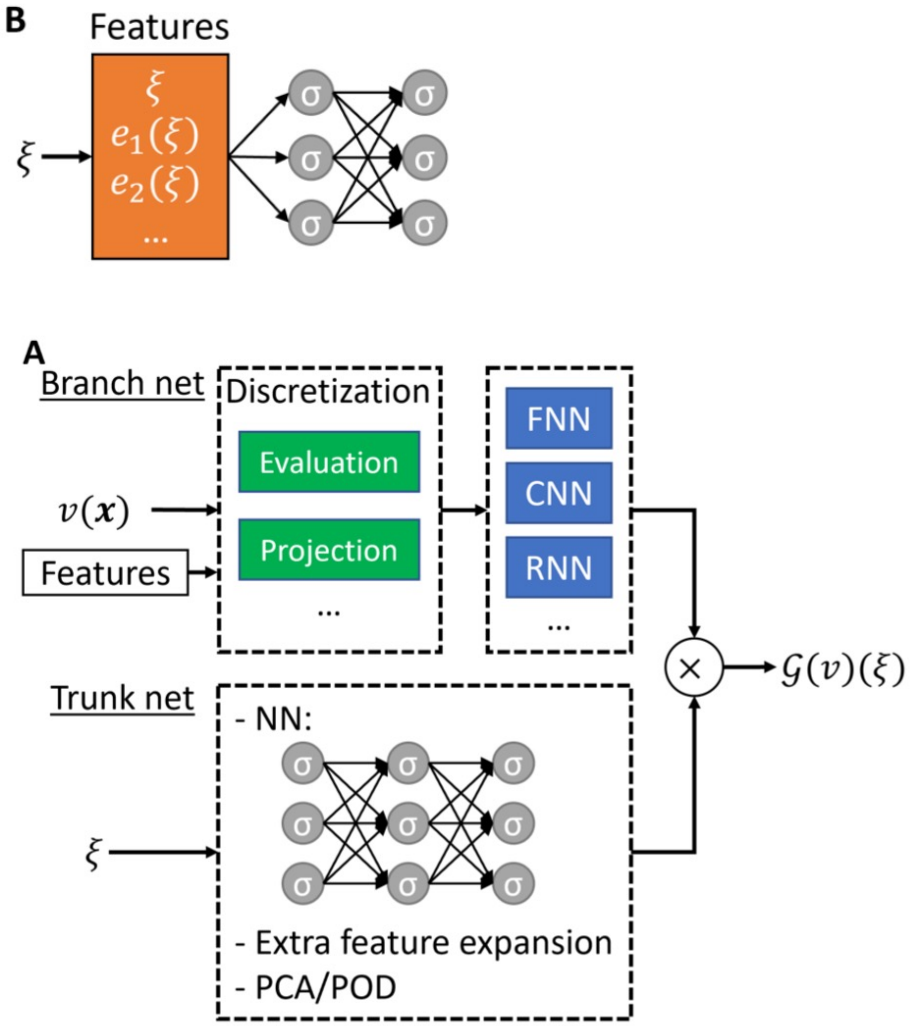
Feature Expansion

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 [49, 13]. 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).



Hard-constraint boundary conditions

Dirichlet BCs.. Enforcing Dirichlet BCs in neural networks has been widely used in PINNs [13]. 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:

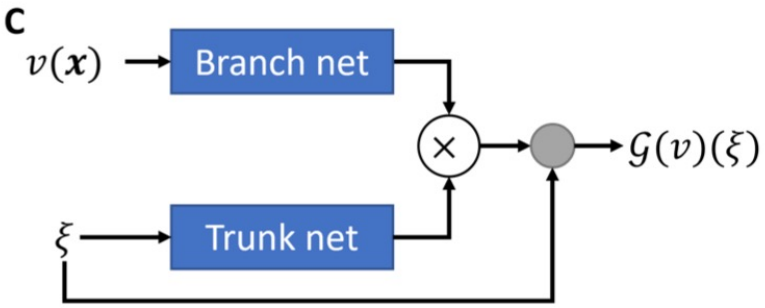
$$\begin{cases} \ell(\xi) = 0, & \xi \in \Gamma_D, \\ \ell(\xi) > 0, & \text{otherwise.} \end{cases}$$

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 [13, 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

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

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 $\{\cos(\omega\xi), \sin(\omega\xi)\}$.



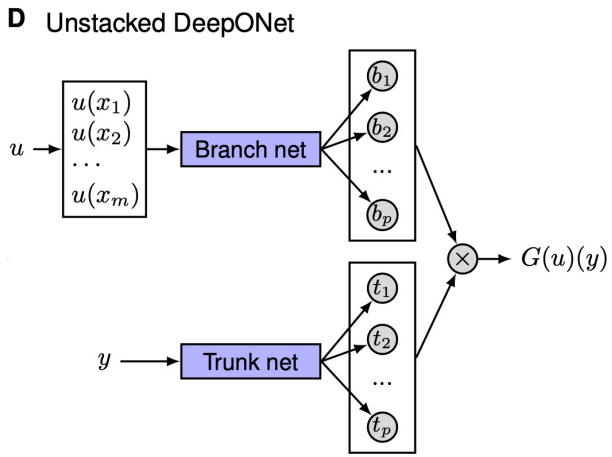
Similarly, in 2D case

Multi-dimension DeepONET

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.



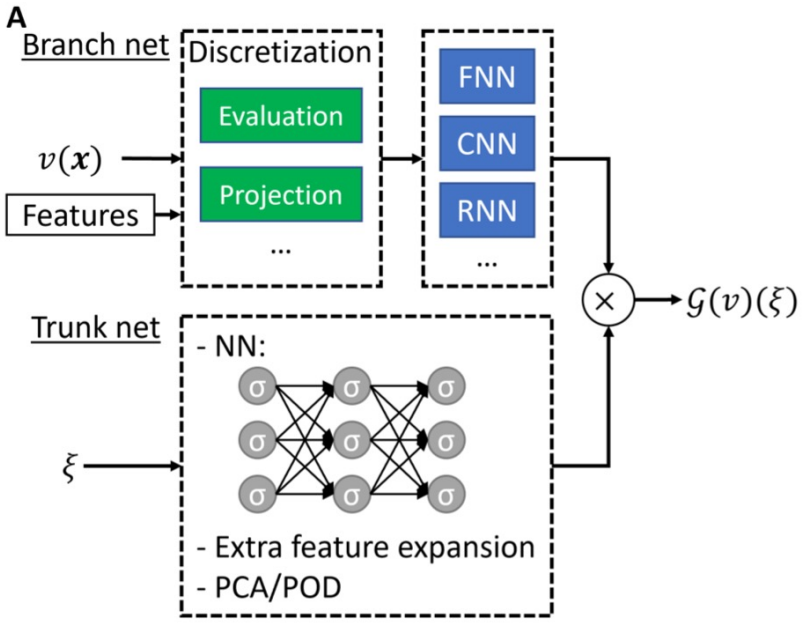
$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k.$$

POD-DeepONET

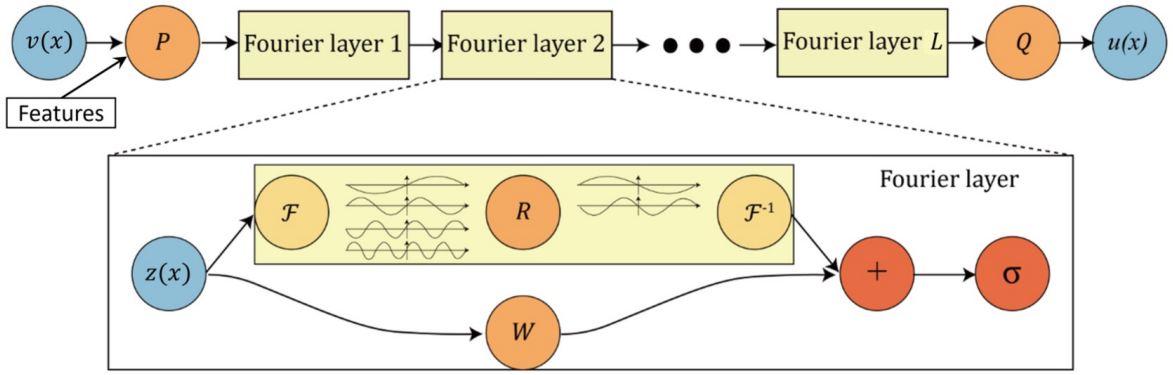
The vanilla DeepONet uses the trunk net to automatically learn the basis of the output function from the data. Here, we propose the POD-DeepONet, where we compute the basis by performing proper orthogonal decomposition (POD) on the training data (after first removing the mean). Then, we use this POD 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:

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

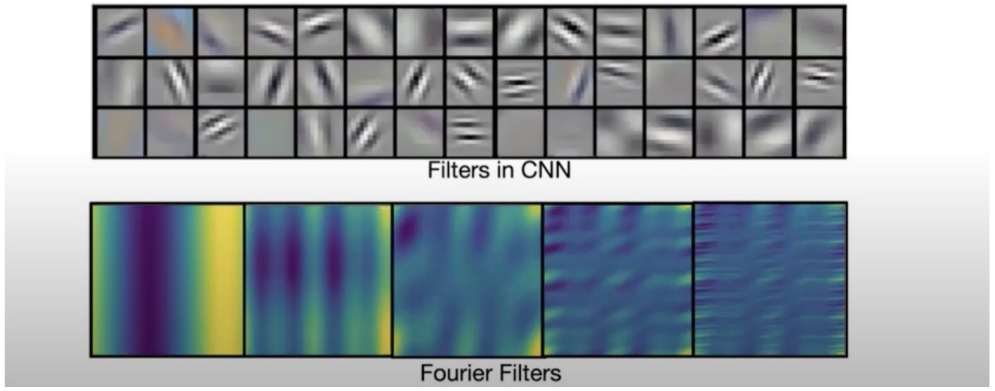
where $\phi_0(\xi)$ is the mean function of $u(\xi)$ computed from the training dataset. $\{b_1, b_2, \ldots, b_p\}$ are the p outputs of the branch net, and $\{\phi_1, \phi_2, \ldots, \phi_p\}$ are the p precomputed POD modes of $u(\xi)$. The proposed POD-DeepONet shares a similar idea of using POD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



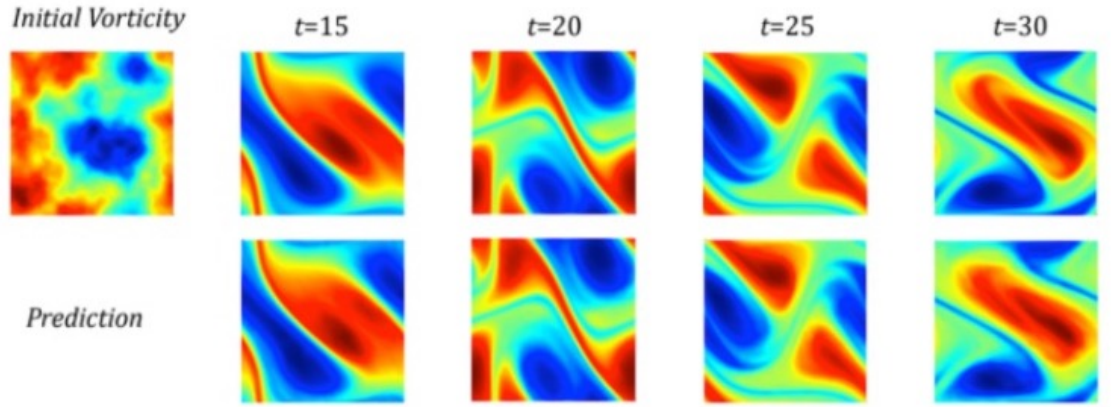
Fourier Neural Operator



Fourier representation is more efficient than CNN.



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



dFNO+

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 [3]: 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.

gFNO+

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

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

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.

Summary of DeepONet vs. 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

Table 2: **Comparison between vanilla DeepONet and vanilla FNO.**

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. [3], 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.

	§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%

Networks	$s = 256$	$s = 512$	$s = 1024$	$s = 2048$	$s = 4096$	$s = 8192$
NN	0.4714	0.4561	0.4803	0.4645	0.4779	0.4452
GCN	0.3999	0.4138	0.4176	0.4157	0.4191	0.4198
FCN	0.0958	0.1407	0.1877	0.2313	0.2855	0.3238
PCANN	0.0398	0.0395	0.0391	0.0383	0.0392	0.0393
DeepONet	0.0569	0.0617	0.0685	0.0702	0.0833	0.0857
GNO	0.0555	0.0594	0.0651	0.0663	0.0666	0.0699
LNO	0.0212	0.0221	0.0217	0.0219	0.0200	0.0189
MGNO	0.0243	0.0355	0.0374	0.0360	0.0364	0.0364
FNO	**0.0018**	**0.0018**	**0.0018**	**0.0019**	**0.0020**	**0.0019**

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.

- **Piecewise constant (PWC).** The first dataset is from Ref. [3]. 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. Specifically, we keep the leading 100 terms in the KL expansion for the Gaussian process with zero mean and the following covariance kernel:

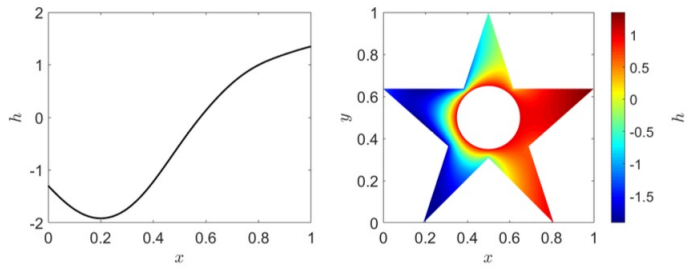
$$\mathcal{K}((x,y),(x',y')) = \exp\left[\frac{-(x-x')}{2l_1^2} + \frac{-(y-y')^2}{2l_2^2}\right],$$

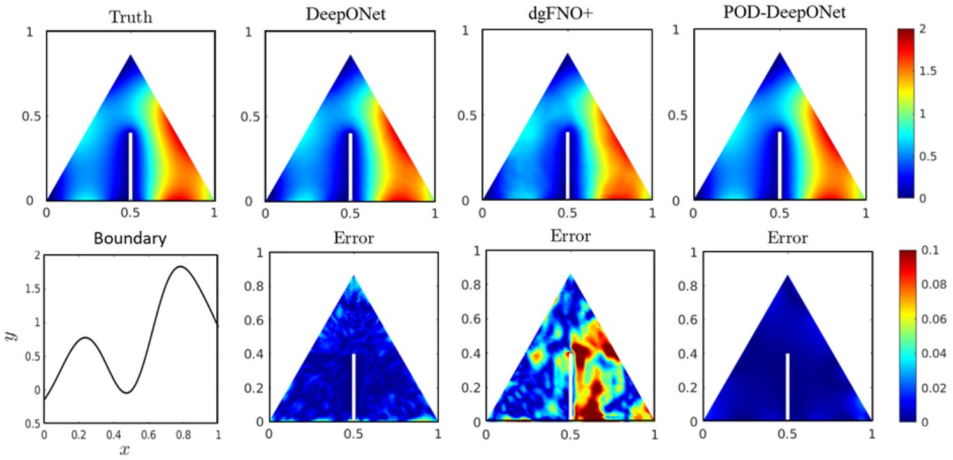
with $l_1 = l_2 = 0.25$. Both $K(\boldsymbol{x})$ and $h(\boldsymbol{x})$ have the same resolution of 20×20.

Networks	$s = 85$	$s = 141$	$s = 211$	$s = 421$
NN	0.1716	0.1716	0.1716	0.1716
FCN	0.0253	0.0493	0.0727	0.1097
PCANN	0.0299	0.0298	0.0298	0.0299
RBM	0.0244	0.0251	0.0255	0.0259
DeepONet	0.0476	0.0479	0.0462	0.0487
GNO	0.0346	0.0332	0.0342	0.0369
LNO	0.0520	0.0461	0.0445	–
MGNO	0.0416	0.0428	0.0428	0.0420
FNO	**0.0108**	**0.0109**	**0.0109**	**0.0098**

	§5.2.1 Darcy (PWC)	§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%**

Darcy Problem

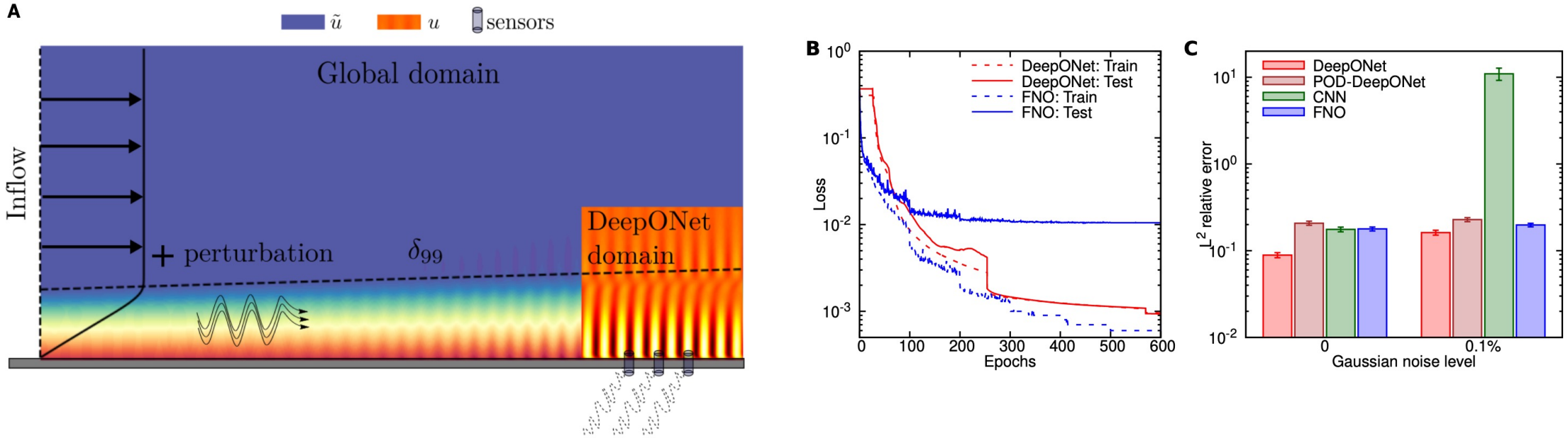




	§5.2.2 Darcy (Pentagram)	§5.2.3 Darcy (Triangular)	§5.2.4 Darcy (Notch)
DeepONet	1.19±0.12%	0.43±0.02%	2.64±0.02%
FNO	—	—	—
POD-DeepONet	**0.82±0.05%**	**0.18±0.02%**	**1.00±0.00%**
dgFNO+	3.34±0.01%	1.00±0.03%	7.82±0.03%

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.



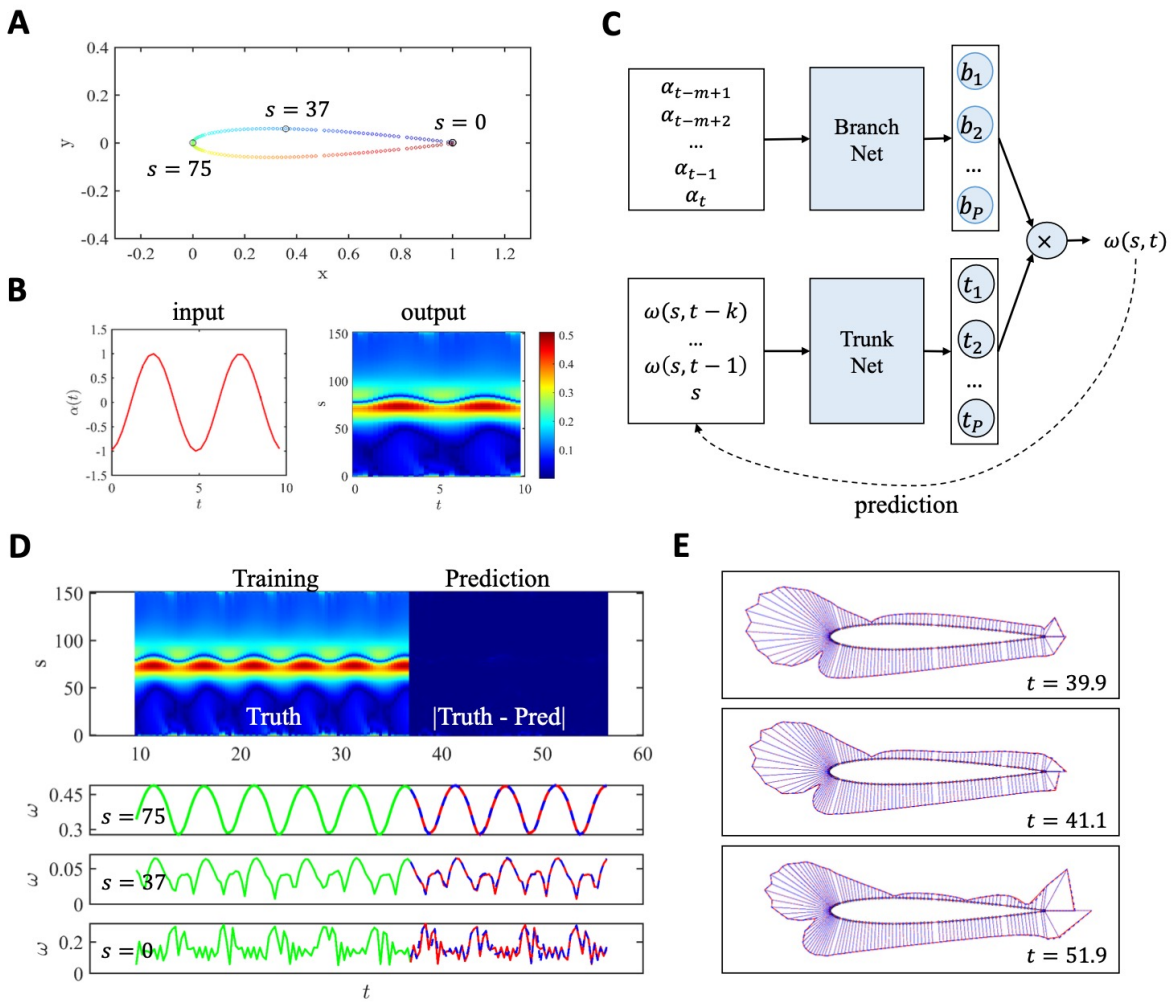
Surface Vorticity of a Flapping airfoil

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 a oscillating inflow velocity, which is expressed as:

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

where $\alpha_0 = 15°$ 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.

	§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%



Navier Stock Equations

Problem setup. Following the problem setup in [3], 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:

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

	§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%**

Configs	Parameters	Time per epoch	$\nu = 1e{-}3$	$\nu = 1e{-}4$	$\nu = 1e{-}5$
FNO-3D	6,558,537	38.99s	**0.0086**	**0.0820**	0.1893
FNO-2D	414,517	127.80s	0.0128	0.0973	**0.1556**
U-Net	24,950,491	48.67s	0.0245	0.1190	0.1982
TF-Net	7,451,724	47.21s	0.0225	0.1168	0.2268
ResNet	266,641	78.47s	0.0701	0.2311	0.2753

Physics-Informed Neural Operator(PINO)

2.1 Problem settings

We consider two natural class of PDEs. In the first, we consider the stationary system

$$\begin{aligned} \mathcal{P}(u, a) &= 0, && \text{in } D \subset \mathbb{R}^d \\ u &= g, && \text{in } \partial D \end{aligned} \tag{1}$$

where D is a bounded domain, $a \in \mathcal{A} \subseteq \mathcal{V}$ is a PDE coefficient/parameter, $u \in \mathcal{U}$ is the unknown, and $\mathcal{P} : \mathcal{U} \times \mathcal{A} \to \mathcal{F}$ is a possibly non-linear partial differential operator with $(\mathcal{U}, \mathcal{V}, \mathcal{F})$ a triplet of Banach spaces. Usually the function g is a fixed boundary condition (potentially can be entered as a parameter). This formulation gives rise to the solution operator $\mathcal{G}^\dagger : \mathcal{A} \to \mathcal{U}$ defined by $a \mapsto u$. A prototypical example is the second-order elliptic equation $\mathcal{P}(u, a) = -\nabla \cdot (a \nabla u) + f$.

$$\begin{aligned} \mathcal{L}_{\text{pde}}(a, u_\theta) &= \left\| \mathcal{P}(a, u_\theta) \right\|^2_{L^2(D)} + \alpha \left\| u_\theta|_{\partial D} - g \right\|^2_{L^2(\partial D)} \\ &= \int_D |\mathcal{P}(u_\theta(x), a(x))|^2 \mathrm{d}x + \alpha \int_{\partial D} |u_\theta(x) - g(x)|^2 \mathrm{d}x \end{aligned}$$

$$\mathcal{L}_{\text{data}}(u, \mathcal{G}_\theta(a)) = \|u - \mathcal{G}_\theta(a)\|^2_{\mathcal{U}} = \int_D |u(x) - \mathcal{G}_\theta(a)(x)|^2 \mathrm{d}x$$

where we assume the setting of (1) for simplicity of the exposition. The operator data loss is defined as the average error across all possible inputs

$$\mathcal{J}_{\text{data}}(\mathcal{G}_\theta) = \|\mathcal{G}^\dagger - \mathcal{G}_\theta\|^2_{L^2_\mu(\mathcal{A};\mathcal{U})} = \mathbb{E}_{a \sim \mu}[\mathcal{L}_{\text{data}}(a, \theta)] \approx \frac{1}{N} \sum_{j=1}^{N} \int_D |u_j(x) - \mathcal{G}_\theta(a_j)(x)|^2 \mathrm{d}x. \tag{6}$$

Similarly, one can define the operator PDE loss as

$$\mathcal{J}_{\text{pde}}(\mathcal{G}_\theta) = \mathbb{E}_{a \sim \mu}[\mathcal{L}_{\text{pde}}(a, \mathcal{G}_\theta(a))]. \tag{7}$$

Physics-Informed Neural Operator

We propose the PINO framework that uses one neural operator model $\mathcal{G}_\theta$ for solving both operator learning problems and equation solving problems. It consists of two phases

- **Pre-train the solution operator**: learn a neural operator $\mathcal{G}_\theta$ to approximate $\mathcal{G}^\dagger$ using either/both the data loss $\mathcal{J}_{data}$ and/or the PDE loss $\mathcal{J}_{pde}$ which can be additional to the dataset.
- **Test-time optimization**: use $\mathcal{G}_\theta(a)$ as the ansatz to approximate $u^\dagger$ with the pde loss $\mathcal{L}_{pde}$ and/or an additional operator loss $\mathcal{L}_{op}$ obtained from the pre-train phase.

3.1 Pre-train: operator learning with PDE loss

Given a fixed amount of data $\{a_j, u_j\}$, the data loss $\mathcal{J}_{data}$ offers a stronger constraint compared to the PDE loss $\mathcal{J}_{pde}$. However, the PDE loss does not require a pre-existing dataset. One can sample virtual PDE instances by drawing additional initial conditions or coefficient conditions $a_j \sim \mu$ for training, as shown in Algorithm 1. In this sense, we have access to the unlimited dataset by sampling new a_j in each iteration.

3.2 Test-time optimization: solving equation with operator ansatz

Given a learned operator $\mathcal{G}_\theta$, we use $\mathcal{G}_\theta(a)$ as the ansatz to solve for $u^\dagger$. The optimization procedure is similar to PINNs where it computes the PDE loss $\mathcal{L}_{pde}$ on a, except that we propose to use a neural operator instead of a neural network. Since the PDE loss is a soft constraints and challenging to optimize, we also add an optional operator loss $\mathcal{L}_{op}$ (anchor loss) to bound the further optimized model from the pre-trained model

$$\mathcal{L}_{op}\big(\mathcal{G}_{\theta_i}(a), \mathcal{G}_{\theta_0}(a)\big) := \|\mathcal{G}_{\theta_i}(a) - \mathcal{G}_{\theta_0}(a)\|_{\mathcal{U}}^2$$

Experiment of PINO

A.1.2 Darcy Flow

We use the 1000 coefficient conditions a to train the solution operator where $a \sim \mu$ where $\mu = \psi_{\#}\mathcal{N}(0, (-\Delta + 9I)^{-2})$, $\psi(a(x)) = 12$ if $a(x) \geq 0$; $\psi(a(x)) = 3$ if $a(x) < 0$. The zero boundary condition is enforced by multiplying a mollifier $m(x) = \sin(\pi x)\sin(\pi y)$ for all methods. The parameter of PINO on Darcy Flow is the same as in the Burgers equation above. Regarding the implementation detail of the baselines: as for FNO, we use the same hyperparameters as its paper did [20]; DeepONet [24] did not study Darcy flow so we grid search the hyperparameters of DeepONet: depth from 2 to 12, width from 50 to 100. The best result of DeepONet is achieved by depth 8, width 50. The results are shown in Table 3.

Method	Solution error	Equation error
DeepONet with data [24]	$6.97 \pm 0.09\%$	-
FNO with data [20]	$1.98 \pm 0.05\%$	1.3645 ± 0.014
PINO with data	$1.22 \pm 0.03\%$	0.5740 ± 0.008
PINO w/o data	$1.50 \pm 0.03\%$	0.4868 ± 0.007

Table 3: Operator learning on Darcy Flow.

Experiment of PINO

4.2 Solve equation using operator ansatz

# data samples	# additional PDE instances	Solution error	Equation error
0	100k	74.36%	0.3741
0.4k	0	33.32%	1.8779
0.4k	40k	31.74%	1.8179
0.4k	160k	**31.32%**	1.7840
4k	0	25.15%	1.8223
4k	100k	**24.15%**	1.6112
4k	400k	**24.22%**	1.4596

Table 1: Operator-learning on Kolmogorov flow $Re = 500$. Each relative L_2 test error is averaged over 300 instances, which is evaluated with resolution $128 \times 128 \times 65$. Complete results of other resolutions are reported in Table 4.

Based on the solution operators learned in the above operator-learning section, we continue to do test-time optimization. The results are shown in Figure and Table 2. Overall, PINO outperforms PINN by **20x** smaller error and **25x** speedup. Using a pre-trained model makes PINO converge faster. The implementation detail and the search space of parameters are included in the Appendix A.2.1.

Method	# data samples	# additional PDE instances	Solution error (w)	Time cost
PINNs	-	-	18.7%	4577s
PINO	0	0	**0.9%**	608s
PINO	4k	0	**0.9%**	536s
PINO	4k	160k	**0.9%**	**473s**

Table 2: Equation-solving on Kolmogorov flow $Re = 500$.

Experiment of PINO

A.2 Transfer learning across Reynolds numbers

We study the test-time optimization with different Reynolds numbers on the 1s Kolmogorov flow. For the higher Reynolds number problem $Re = 500, 400$, the pre-training operator shows better convergence accuracy. In all cases, the pre-training operator shows better convergence speed as demonstrated in Figure 5. The results are shown in Table 5 where the error is averaged over 40 instances. Each row is a testing case and each column is a pre-trained operator.

Testing RE	From scratch	100	200	250	300	350	400	500
500	0.0493	0.0383	0.0393	0.0315	0.0477	0.0446	0.0434	0.0436
400	0.0296	0.0243	0.0245	0.0244	0.0300	0.0271	0.0273	0.0240
350	0.0192	0.0210	0.0211	0.0213	0.0233	0.0222	0.0222	0.0212
300	0.0168	0.0161	0.0164	0.0151	0.0177	0.0173	0.0170	0.0160
250	0.0151	0.0150	0.0153	0.0151	0.016	0.0156	0.0160	0.0151
200	0.00921	0.00913	0.00921	0.00915	0.00985	0.00945	0.00923	0.00892
100	0.00234	0.00235	0.00236	0.00235	0.00239	0.00239	0.00237	0.00237

Table 5: Reynolds number transfer learning. Each row is a test set of PDEs with corresponding Reynolds number. Each column represents the operator ansatz we use as the starting point of test-time optimization. For example, column header '100' means the operator ansatz is trained over a set of PDEs with Reynolds number 100. The relative L_2 errors is averaged over 40 instances of the corresponding test set.