SIAM J. SCI. COMPUT.
Vol. 46, No. 4, pp. C273–C296

ON THE TRAINING AND GENERALIZATION OF DEEP OPERATOR NETWORKS*

SANGHYUN LEE† AND YEONJONG SHIN‡

Abstract. We present a novel training method for deep operator networks (DeepONets), one of the most popular neural network models for operators. DeepONets are constructed by two subnetworks, namely the branch and trunk networks. Typically, the two subnetworks are trained simultaneously, which amounts to solving a complex optimization problem in a high dimensional space. In addition, the nonconvex and nonlinear nature makes training very challenging. To tackle such a challenge, we propose a two-step training method that trains the trunk network first and then sequentially trains the branch network. The core mechanism is motivated by the divide-and-conquer paradigm and is the decomposition of the entire complex training task into two subtasks with reduced complexity. Therein the Gram–Schmidt orthonormalization process is introduced which significantly improves stability and generalization ability. On the theoretical side, we establish a generalization error estimate in terms of the number of training data, the width of DeepONets, and the number of input and output sensors. Numerical examples are presented to demonstrate the effectiveness of the two-step training method, including Darcy flow in heterogeneous porous media.

Key words. deep operator networks, divide-and-conquer, sequential training method, generalization error analysis

MSC codes. 65K05, 65B99, 93E24, 42C05

DOI. 10.1137/23M1598751

1. Introduction.

Influenced by huge empirical successes of deep learning and artificial intelligence, numerous neural network–based computational models have been proposed. The effectiveness of these approaches has been documented and reported in various research papers [31, 29, 8, 26, 34, 9, 19], with references included therein. Among them, operator learning has garnered significant attention, particularly because of its potential applications in addressing forward and backward problems involving partial differential equations (PDEs). One feature that sets the deep learning approach apart from traditional numerical methods is the interplay between the training process and the inference on unseen data. While the training could be very challenging and time-consuming and require huge computational resources, once it is complete, the inference can be done almost instantaneously.

The deep operator networks (DeepONets) [31] represent pioneering examples of neural operator models [18], with their fundamental structures drawing inspiration from the universal approximation theorem [10] for nonlinear operators. Since then,

*Submitted to the journal's Machine Learning Methods for Scientific Computing section September 5, 2023; accepted for publication (in revised form) March 25, 2024; published electronically July 1, 2024.

https://doi.org/10.1137/23M1598751

Funding: The work of the first author was partially supported by the U.S. National Science Foundation grant DMS-2208402 and by the U.S. Department of Energy, Office of Science, Energy Earthshots Initiatives under Award DE-SC-0024703. The work of the second author was partially supported by an NRF grant funded by the Ministry of Science and ICT of Korea (RS-2023-00219980).

†Department of Mathematics, Florida State University, Tallahassee, FL 32306 USA (lee@math.fsu.edu).

‡Corresponding author. Department of Mathematics, North Carolina State University, Raleigh, NC 27695-8205 USA, and Mathematical Institute for Data Science, Pohang University of Science and Technology (POSTECH), Pohang, 37673, Republic of Korea (yeonjong_shin@ncsu.edu).

various neural network–based operator models have emerged, primarily aimed at enhancing performance on previously unseen data. Notable models encompass different versions of DeepONets [32, 42, 25, 30, 23, 41] and Fourier neural operator adaptations [5, 28, 20, 7, 39]. Nevertheless, the accuracy of the deduced solutions significantly hinges on multiple factors, including but not limited to network architectures, training data, hyperparameters, and training methodologies.

The current study aims to comprehend the aforementioned interplay by focusing on DeepONets, given its status as one of the most widely used network models for operators. This work introduces two main contributions. First, a novel training method is developed, enhancing both training/optimization and generalization performances. The DeepONets consist of two subnetworks, namely trunk and branch networks. Standard approaches usually train these two subnetworks monolithically as a unified entity using a first-order optimization method, such as `Adam` [24] and other variants [37]. Accordingly, the training amounts to solving a complex optimization problem defined in a very high dimensional space in addition to it being nonconvex and nonlinear. Many difficulties encountered in practice are related to training, despite the inherent expressive capabilities of DeepONets [13, 27, 33].

To improve the conventional monolithic optimization approach, we propose a two-step training method. The method is motivated by the divide-and-conquer paradigm. The core mechanism is the decomposition of the entire complex optimization problem into two subproblems with reduced complexity. As the name implies, this method involves two steps. The first step is devoted to learning the trunk network. The trunk network can be thought of as the basis representing the output functions, while the branch network corresponds to appropriate coefficients. The first step aims at finding the basis representation through the trunk network together with the corresponding coefficients without introducing the branch network. The coefficients found in the first step shall be the desired values for the branch networks to learn. Consequently, in the second step, the branch network is trained to learn the values obtained from the first step. The Gram–Schmidt orthonormalization is applied to the trunk network in the first step, which significantly improves stability and generalization ability. This separation of the learning process results in each step involving an optimization task with considerably reduced complexity. In addition, we show that the minimized loss achieved by the two-step method is identical to the one by the monolithic approach (Theorem 3.4). We also prove that a zero training loss can be achieved under mild overparameterization of DeepONets (Theorem 3.6). Numerical examples demonstrate that the two-step training method significantly enhances the performance of DeepONets over the same dataset when compared to results obtained through monolithic training methods.

The other contribution is the generalization error estimate. The total error of DeepONets can be decomposed by approximation, estimation, and optimization errors. The approximation error of DeepONets is relatively well established (e.g., see [13, 27, 33]), while the remaining two errors remain elusive. One reason may deviate from that (1) every input function has to be transformed into a finite-dimensional representation (e.g., discretization or generalized Fourier coefficients), and (2) DeepONets learn only from a finite number of data. Unlike function approximation tasks, the input space for operators is, in principle, infinite-dimensional, which requires one to appropriately extend the space-filling argument [38] to the infinite-dimensional counterpart. This is closely related to the input domain on which DeepONets can learn and generalize. To mathematically characterize these features, we introduce Assumption 4.3 and present multiple examples that satisfy it. We then connect the DeepONet structures to the least-squares regression [11, 12] with respect to the

number of random sensor points for the output functions. Since the two-step training allows the branch network to learn the optimal least-squares coefficients directly, an existing error bound of the least-squares regression [11, 12] provides the generalization error estimate for DeepONets. Yet, there are certain conditions to be met for the trunk network, which we formulate in Assumption 4.7.

The rest of the paper is organized as follows. Upon introducing the problem set and preliminaries in section 2, the proposed two-step training method for DeepONets is presented in section 3. A generalization error analysis is given in section 4. Numerical examples are presented in section 5 before conclusions.

2. Problem setup and preliminaries. Let $\Omega_x \subset \mathbb{R}^{d_x}$ and $\Omega_y \subset \mathbb{R}^{d_y}$ be computational (compact) domains of interest. Let $(\mathcal{X}, d_{\mathcal{X}})$ be a metric space of functions defined on Ω_x and let $(\mathcal{Y}, \|\cdot\|_{\mathcal{Y}})$ be a normed vector space of functions on Ω_y. Let

$$\mathcal{G} : \mathcal{X} \ni f \mapsto \mathcal{G}[f] \in \mathcal{Y}$$

be the operator of interest to be approximated by neural networks.

In order to utilize any input function $f \in \mathcal{X}$ with DeepONets, it must first undergo a conversion into a finite-dimensional quantity. There are various ways of extracting information of f. One naive way is to consider a fixed set of discretization points $\{x_i\}_{i=1}^{m_{\mathrm{x}}} \subset \Omega_x$ and discretize $f \in \mathcal{X}$ by $\boldsymbol{f} = (f(x_1), \ldots, f(x_{m_{\mathrm{x}}})) \in \mathbb{R}^{m_{\mathrm{x}}}$ [31]. Another popular way is to extract finitely many (generalized) Fourier coefficients of f [28]. For example, suppose $f(x) = \sum_{i=1}^{\infty} \hat{f}_i \phi_i(x)$ where $\langle \cdot, \cdot \rangle$ is an appropriate inner product, $\{\phi_i\}$ is an associated orthogonal basis, and $\hat{f}_i = \langle f, \phi_i \rangle$. Then, $\boldsymbol{f} = (\hat{f}_1, \ldots, \hat{f}_{m_{\mathrm{x}}})$. For the rest of the paper, bold font denotes a representation in a finite-dimensional space, and m_{x} is referred to as the number of input function sensors.

The goal of operator learning is to construct an operator network $\mathcal{G}_{\mathrm{NN}}$ that approximates the target operator $\mathcal{G}$. An operator network is a mapping defined by

$$\mathcal{G}_{\mathrm{NN}} : \mathbb{R}^{m_{\mathrm{x}}} \ni \boldsymbol{f} \mapsto \mathcal{G}_{\mathrm{NN}}[\boldsymbol{f}](\cdot) \in \mathcal{Y}. \tag{2.1}$$

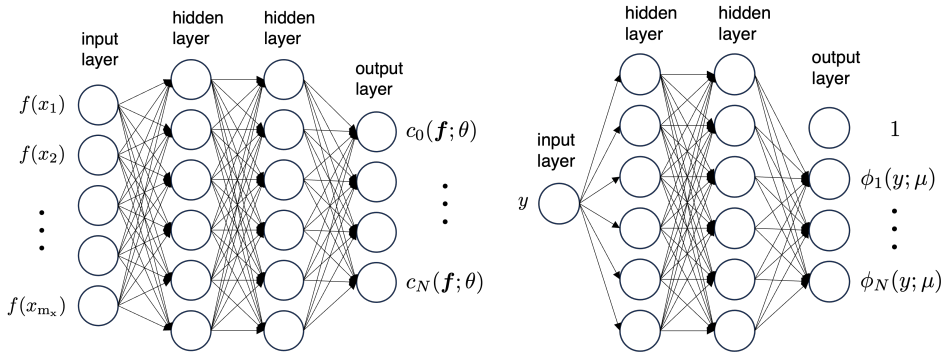
While various methodologies for designing operator networks are available, DeepONets [31, 32] are the pioneering neural operator model proposed in the literature. This study centers its attention on DeepONets, and a detailed explanation thereof is given in the following subsection.

2.1. Deep operator networks. For $L \in \mathbb{N}$ and $\vec{\boldsymbol{n}} = (n_0, n_1, \ldots, n_L) \in \mathbb{N}^{L+1}$, an L-layer feed-forward neural network is the mapping $\mathbb{R}^{n_0} \ni x \mapsto z^L \in \mathbb{R}^{n_L}$ where z^L is defined recursively by

$$z^\ell = W^\ell \sigma(z^{\ell-1}) + b^\ell, \quad 2 \le \ell \le L, \quad z^1 = W^1 x + b^1.$$

Here, σ is a nonlinear activation function that applies elementwise, and $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ and $b^\ell \in \mathbb{R}^{n_\ell}$ are the weight matrix and the bias vector of the ℓth layer, respectively. The vector $\vec{\boldsymbol{n}}$ is referred to as the network architecture and $\{W^\ell, b^\ell\}_{\ell=1}^{L}$ as the network parameters.

The DeepONet framework [31] integrates two distinct neural networks to effectively approximate the target operator, referred to as the *branch* and *trunk* networks. The construction of the branch network leads to two variants of DeepONet: unstacked and stacked versions [31]. While the presented materials can be applicable to both versions, for the sake of simplicity, this paper concentrates on the unstacked variant, denoting it as DeepONet (with a slight abuse of notation) throughout the subsequent sections. Throughout this work, N shall be referred to as the *width* of DeepONet.



(a) Branch network (b) Trunk network

FIG. 1. *An illustration of the unstacked DeepONet architecture.*

The branch network is a vector-valued L_b-layer neural network

$$\boldsymbol{c}(\cdot;\theta) = (c_0(\cdot;\theta), \dots, c_N(\cdot;\theta))^\top, \tag{2.2}$$

whose architecture is $\vec{\boldsymbol{n}}_b = (m_{\mathrm{x}}, n_1^{(b)}, \dots, n_{L_b-1}^{(b)}, N+1)$, and θ represents the network parameters. The trunk network is a vector-valued L_t-layer neural network defined on $\Omega_y \subset \mathbb{R}^{d_y}$

$$\boldsymbol{\phi}(\cdot;\mu) = (1, \boldsymbol{\phi}_0(\cdot;\mu))^\top, \tag{2.3}$$

where $\boldsymbol{\phi}_0(\cdot;\mu) = (\phi_1(\cdot;\mu), \dots, \phi_N(\cdot;\mu))$ is an L_t-layer neural network whose architecture is $\vec{\boldsymbol{n}}_t = (d_y, n_1^{(t)}, \dots, n_{L_t-1}^{(t)}, N)$ and μ represents the network parameters.

Then, a DeepONet is defined as the inner product of the branch and the trunk networks, i.e.,

$$O_{\mathrm{net}}[\boldsymbol{f};\Theta](y) := \boldsymbol{\phi}^\top(y;\mu)\boldsymbol{c}(\boldsymbol{f};\theta) = c_0(\boldsymbol{f};\theta) + \sum_{j=1}^{N} c_j(\boldsymbol{f};\theta)\phi_j(y;\mu), \tag{2.4}$$

where $\Theta = \{\mu, \theta\}$ is the set of trainable DeepONet parameters. See Figure 1 for more details of the DeepONet architecture.

2.2. Training of deep operator networks. Let $\{f_k\}_{k=1}^K$ be a set of input functions from $\mathcal{X}$ and let $u_k(\cdot) = \mathcal{G}[f_k](\cdot)$ be the corresponding output functions in $\mathcal{Y}$. Ideally, one wishes to minimize

$$\mathcal{L}_{\mathrm{ideal}}(\Theta) = \frac{1}{K}\sum_{k=1}^{K} \|O_{\mathrm{net}}[\boldsymbol{f}_k;\Theta](\cdot) - u_k(\cdot)\|_{\mathcal{Y}}^p,$$

where $\|\cdot\|_{\mathcal{Y}}$ is the norm in $\mathcal{Y}$, and p is a positive number that may depend on $\|\cdot\|_{\mathcal{Y}}$. For example, if $\|\cdot\|_{\mathcal{Y}}$ is the L_2-norm, $p = 2$. In practice, however, the norm $\|\cdot\|_{\mathcal{Y}}$ has to be discretized, and let $\|\cdot\|_{\mathcal{Y}_{m_y}}$ be a discretized norm. We then seek to find parameters of the DeepONet that minimize the training loss $\mathcal{L}$ defined by

$$\mathcal{L}(\Theta) = \frac{1}{K}\sum_{k=1}^{K} \|O_{\mathrm{net}}[\boldsymbol{f}_k;\Theta](\cdot) - u_k(\cdot)\|_{\mathcal{Y}_{m_y}}^p. \tag{2.5}$$

Typically, the optimization problem is tackled using first-order optimization methods such as stochastic gradient descent and its variants [37]. However, the nonconvex and nonlinear nature of the loss frequently obstructs these methods from achieving satisfactory loss minimization, leading to failures in some applications. However, as evidenced by multiple extant studies [10, 31, 13, 33], DeepONets have the capacity to approximate a multitude of nonlinear operators, including those commonly encountered in physical and engineering challenges, particularly those involving PDEs. Such an expressivity of DeepONets, however, becomes null if an effective training mechanism is not available.

3. Method. To address the aforementioned challenge, we present a novel training method for DeepONets. For ease of discussion, we confine ourselves to the case of $\mathcal{Y} = L^p_\omega(\Omega_y)$ whose norm is defined by

$$\|g\|_{\mathcal{Y}} = \left(\int_{\Omega_y} |g(y)|^p d\omega(y) \right)^{\frac{1}{p}} \qquad \forall g \in \mathcal{Y},$$

where ω is a probability measure satisfying $\int_{\Omega_y} d\omega(y) = 1$, and consider the corresponding discrete norm by Monte Carlo sampling,

$$\|g\|_{\mathcal{Y}_{m_{\mathrm{y}}}} = \left(\frac{1}{m_{\mathrm{y}}} \sum_{i=1}^{m_{\mathrm{y}}} |g(y_i)|^p \right)^{\frac{1}{p}} \qquad \forall g \in \mathcal{Y},$$

where $\{y_i\}_{i=1}^{m_{\mathrm{y}}}$ are independent and identically distributed random samples from ω. Here m_{y} is referred to as the number of output function sensors.

For any $g \in \mathcal{Y}$, let $\boldsymbol{g} = (g(y_1), \ldots, g(y_{m_{\mathrm{y}}}))^\top$ be the discretization of g. Note that the discrete norm $\|g\|_{\mathcal{Y}_{m_{\mathrm{y}}}}$ is the standard ℓ_p-norm $\|\boldsymbol{g}\|_{\ell_p}$ up to a multiplicative constant. Then the set of training data can be written as

$$(\boldsymbol{f}_k, \boldsymbol{u}_k) = (f_k(x_1), \ldots, f_k(x_{m_{\mathrm{x}}}), u_k(y_1), \ldots, u_k(y_{m_{\mathrm{y}}})), \quad k = 1, \ldots, K.$$

It follows from (2.4) that the training loss function (2.5) is

$$\mathcal{L}(\{\mu, \theta\}) = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{m_{\mathrm{y}}} \sum_{i=1}^{m_{\mathrm{y}}} |\boldsymbol{\phi}^\top(y_i; \mu) \boldsymbol{c}(\boldsymbol{f}_k; \theta) - u_k(y_i)|^p, \tag{3.1}$$

where θ and μ represent all the network parameters of branch $\boldsymbol{c}(\cdot; \theta)$ and trunk $\boldsymbol{\phi}(\cdot; \mu)$ networks, respectively.

3.1. Loss function via matrix representation. By exploiting matrix representation, the loss function (3.1) can be written in a simple matrix form. Let

$$\boldsymbol{\Phi}(\mu) = \begin{bmatrix} \boldsymbol{\phi}^\top(y_1; \mu) \\ \vdots \\ \boldsymbol{\phi}^\top(y_{m_{\mathrm{y}}}; \mu) \end{bmatrix} \in \mathbb{R}^{m_{\mathrm{y}} \times (N+1)}, \ \boldsymbol{C}(\theta) = [\boldsymbol{c}(\boldsymbol{f}_1; \theta), \ldots, \boldsymbol{c}(\boldsymbol{f}_K; \theta)] \in \mathbb{R}^{(N+1) \times K},$$

and $\boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_K] \in \mathbb{R}^{m_{\mathrm{y}} \times K}$. The matrix $\boldsymbol{\Phi}(\mu)$ may be viewed as a Vandermonde-like matrix, while $\boldsymbol{C}(\theta)$ may be viewed as the corresponding coefficient-like matrix. It can be checked that the loss (3.1) can be expressed as

$$\mathcal{L}(\{\mu, \theta\}) = \frac{1}{K m_{\mathrm{y}}} \|\boldsymbol{\Phi}(\mu) \boldsymbol{C}(\theta) - \boldsymbol{U}\|_{p,p}^p,$$

where $\|\cdot\|_{p,p}$ is the entrywise matrix norm. Hence, training DeepONets means solving the following optimization problem:

$$\min_{\mu,\theta} \mathcal{L}(\{\mu,\theta\}) := \|\boldsymbol{\Phi}(\mu)\boldsymbol{C}(\theta) - \boldsymbol{U}\|_{p,p}^p. \tag{3.2}$$

3.2. Reparameterization of DeepONets. A key part of the proposed training method is a specific (re)parameterization of DeepONets. Let T be a trainable square matrix of size $N+1$ and consider a new trunk network $\hat{\boldsymbol{\phi}}$ which has the form of

$$\hat{\boldsymbol{\phi}}(\cdot;\mu,T) = T^\top \boldsymbol{\phi}(\cdot;\mu),$$

where $\boldsymbol{\phi}(\cdot;\mu)$ is the standard trunk network defined in (2.3). The resulting DeepONet is then

$$O_{\text{net}}[\boldsymbol{f}](y) = \hat{\boldsymbol{\phi}}^\top(\cdot;\mu,T)\boldsymbol{c}(\boldsymbol{f};\theta) = \boldsymbol{\phi}^\top(\cdot;\mu)T\boldsymbol{c}(\boldsymbol{f};\theta).$$

This can also be viewed as a DeepONet with the same trunk network $\boldsymbol{\phi}(\cdot;\mu)$ as before but a new branch network $T\boldsymbol{c}(\cdot;\theta)$. In fact, if $\boldsymbol{c}$ is a standard vector-valued neural network constructed by the architecture $\vec{\boldsymbol{n}}_b$, $T\boldsymbol{c}(\boldsymbol{f};\theta)$ can be viewed as a reparameterization of the last layer's weight matrix and bias vector as

$$T\boldsymbol{c}(\boldsymbol{f};\theta) = TW^{L_b}\sigma(z^{L_b-1}) + Tb^{L_b}.$$

With this new parameterization, we consider the loss function

$$\min_{\mu,T,\theta} \mathcal{L}(\mu,T,\theta) := \|\boldsymbol{\Phi}(\mu)T\boldsymbol{C}(\theta) - \boldsymbol{U}\|_{p,p}^p, \tag{3.3}$$

and the proposed training method aims at solving (3.3). We note that $\boldsymbol{\Phi}(\mu)T$ can be viewed as a Vandermonde-like matrix constructed from the new trunk network $\hat{\boldsymbol{\phi}}$ as basis.

3.3. Proposed two-step training method. Assuming $m_{\text{y}} > N$, we propose a two-step training method for solving (3.3).

Step 1. The first step trains the new trunk network $\hat{\boldsymbol{\phi}}$ through the following minimization problem:

$$\min_{\mu,A} \mathcal{L}(\mu,A) := \|\boldsymbol{\Phi}(\mu)A - \boldsymbol{U}\|_{p,p}^p, \quad \text{where} \quad A \in \mathbb{R}^{(N+1)\times K}. \tag{3.4}$$

Let (μ^*, A^*) be an optimal solution and let $\boldsymbol{\Phi}(\mu^*)$ be full rank. We then set $T^* = (R^*)^{-1}$ as the inverse of R^* obtained from a QR-factorization of $\boldsymbol{\Phi}(\mu^*)$, i.e., $Q^*R^* = \boldsymbol{\Phi}(\mu^*)$. The trunk network is then fully determined as $\hat{\boldsymbol{\phi}}(\cdot;\mu^*,T^*)$.

Step 2. The second step trains the branch network to fit R^*A^*. Specifically, we consider the optimization problem of

$$\min_{\theta} \|\boldsymbol{C}(\theta) - R^*A^*\|_{2,2}^2. \tag{3.5}$$

Assuming θ^* to be an optimal solution, the fully trained branch network is given by $\boldsymbol{c}(\cdot;\theta^*)$.

Remark 3.1. The first step replaces the use of the branch network from (3.1) to the corresponding value matrix A. The trunk network's parameters μ and the value matrix A are then trained simultaneously to minimize the loss (3.4). Since the trunk

loss function is convex with respect to A (assuming $p \geq 1$), the first step shall avoid any difficulties caused by nonlinearity and nonconvexity from the branch network. In addition, the number of parameters involved with it is $|\mu| + |A|$, while the one for the standard loss (3.1) is $|\mu| + |\theta|$. Hence, as long as $|A| = (N+1)K < |\theta|$, the first step yields an optimization problem whose dimension is smaller than the one of the standard loss. Altogether, the first step (3.4) is designed to facilitate training of the trunk network.

Remark 3.2. The role of T^* may be viewed as applying the Gram–Schmidt orthonormalization process on the standard trunk network with respect to the discretization points $\{y_j\}$. The approximation capability remains the same with or without T^*; however, we found numerically that the introduction of T^* significantly improves stability and generalization ability.

Remark 3.3. The proposed method splits the entire optimization problem into two smaller problems, the complexities of which (computational and/or optimization) are significantly reduced. In the numerical test section, we demonstrate the advantage of our proposed method by comparing the results of a monolithic approach that trains both networks simultaneously.

3.4. Optimization error analysis. We first show that the loss of (3.3) at an optimal solution from the two-step training via (3.4) and (3.5) is equal to the minimum loss of the original problem (3.2).

THEOREM 3.4. *Suppose that the branch network's architecture is sufficiently large so that for any $M \in \mathbb{R}^{(N+1)\times K}$, there exists $\tilde{\theta}$ such that $\boldsymbol{C}(\tilde{\theta}) = M$. Let (μ^*, A^*) and θ^* be optimal solutions of* (3.4) *and* (3.5)*, respectively. Then,*

$$\mathcal{L}(\mu^*, T^*, \theta^*) = \min_{\mu,\theta} \mathcal{L}(\{\mu, \theta\}).$$

Proof. We prove the theorem by contradiction. Suppose that there exists $\{\hat{\mu}, \hat{\theta}\}$ that gives $\mathcal{L}(\{\hat{\mu}, \hat{\theta}\}) < \mathcal{L}(\mu^*, T^*, \theta^*)$. By letting $\hat{A} := \boldsymbol{C}(\hat{\theta})$, it can be checked that

$$\begin{aligned}\mathcal{L}(\mu^*, T^*, \theta^*) &= \|\boldsymbol{\Phi}(\mu^*)T^*\boldsymbol{C}(\theta^*) - \boldsymbol{U}\|_{p,p}^p = \|\boldsymbol{\Phi}(\mu^*)A^* - \boldsymbol{U}\|_{p,p}^p \\ &\leq \|\boldsymbol{\Phi}(\hat{\mu})\hat{A} - \boldsymbol{U}\|_{p,p}^p = \|\boldsymbol{\Phi}(\hat{\mu})\boldsymbol{C}(\hat{\theta}) - \boldsymbol{U}\|_{p,p}^p = \mathcal{L}(\{\hat{\mu}, \hat{\theta}\}),\end{aligned}$$

which is a contradiction. This shows that $\mathcal{L}(\mu^*, T^*, \theta^*) \leq \min_{\mu,\theta} \mathcal{L}(\{\mu, \theta\})$.

Let $\hat{\mu} = \mu^*$ and $\hat{\theta}$ be the branch parameter that satisfies $\boldsymbol{C}(\hat{\theta}) = A^*$. It then can be checked that

$$\mathcal{L}(\{\hat{\mu}, \hat{\theta}\}) = \|\boldsymbol{\Phi}(\mu^*)\boldsymbol{C}(\hat{\theta}) - \boldsymbol{U}\|_{p,p}^p = \|\boldsymbol{\Phi}(\mu^*)A^* - \boldsymbol{U}\|_{p,p}^p = \mathcal{L}(\mu^*, T^*, \theta^*),$$

which shows that $\min_{\mu,\theta} \mathcal{L}(\{\mu, \theta\}) \leq \mathcal{L}(\mu^*, T^*, \theta^*)$. ∎

Theorem 3.4 assumes that the branch network can interpolate any given data. This is indeed possible under mild conditions (e.g., sufficiently large width) as already shown in [35, 43, 14, 15, 3]. In addition, if the branch network is built based on two-layer networks, the minimization problem (3.5) can be effectively solved by a recently developed training method, active neuron least squares (ANLS) [1, 2].

Last, for the training of the trunk network (3.4), a zero loss can also be obtained if the architecture of the trunk network is appropriately chosen.

THEOREM 3.5. *Suppose that $p = 2$, $\boldsymbol{U}$ has rank r and the trunk network $\boldsymbol{\phi}_0$ of (2.3) is a $(2m_y + 1)$-layer rectified linear unit (ReLU) network whose architecture is given by*

$$\vec{\boldsymbol{n}}_t = (d_y, 4, 4, \overbrace{\tilde{n}, \ldots, \tilde{n}}^{(2m_y-2)\ times}, N), \quad \textit{where} \quad \tilde{n} = 2\min\{N, r\} + 4.$$

Then, there exist μ^* and A^* satisfying

$$\mathcal{L}(\mu^*, A^*) := \|\boldsymbol{\Phi}(\mu^*)A^* - \boldsymbol{U}\|_{2,2}^2 \leq \min_{rank(\boldsymbol{Z}) \leq \min\{N,r\}} \|\boldsymbol{Z} - \boldsymbol{U}\|_{2,2}^2.$$

In particular, if $N \geq r$, we have $\mathcal{L}(\mu^*, A^*) = 0$.

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

By combining Theorems 3.4 and 3.5, a zero training loss for DeepONets can be achieved under overparameterization of both trunk and branch networks.

THEOREM 3.6. *Suppose that the architecture of the trunk network is set as described in Theorem 3.5 with $N \geq r$. Suppose also that the branch network's architecture is sufficiently large so that for any $M \in \mathbb{R}^{(N+1)\times K}$, there exists $\tilde{\theta}$ such that $\boldsymbol{C}(\tilde{\theta}) = M$. Then,*

$$0 = \min_{\mu,\theta} \mathcal{L}(\{\mu, \theta\}) := \|\boldsymbol{\Phi}(\mu)\boldsymbol{C}(\theta) - \boldsymbol{U}\|_{p,p}^p.$$

Proof. It follows from Theorem 3.5 that there exist μ^*, A^* satisfying $\mathcal{L}(\mu^*, A^*) = 0$ as $N \geq r$. Since the branch network can achieve a zero loss from (3.5), there exists θ^* such that $\boldsymbol{C}(\theta^*) = R^* A^*$. It then can be checked that

$$0 = \mathcal{L}(\mu^*, A^*) = \|\boldsymbol{\Phi}(\mu^*)A^* - \boldsymbol{U}\|_{p,p}^p = \|\boldsymbol{\Phi}(\mu^*)T^*\boldsymbol{C}(\theta^*) - \boldsymbol{U}\|_{p,p}^p = \mathcal{L}(\mu^*, T^*, \theta^*).$$

The proof is then completed by Theorem 3.4. ∎

4. Generalization error analysis. The generalization error refers to a quantity that measures how well the *learned* DeepONet performs on *unseen* functions (data). To be more precise, let $\mathcal{X}_K = \{f_1, \ldots, f_K\} \subset \mathcal{X}$ be a set of functions and $\{u_j := \mathcal{G}[f_j] : j = 1, \ldots, K\}$ be the corresponding output functions of the operator $\mathcal{G}$ of interest, which are all used for the training of DeepONets. Let O_{net} be a fully trained DeepONet. For $f \in \mathcal{X} \backslash \mathcal{X}_K$, the generalization error of the DeepONet O_{net} at f is defined to be

$$\mathcal{E}_{\text{gen}}(O_{\text{net}}[\boldsymbol{f}]) := \|\mathcal{G}[f] - O_{\text{net}}[\boldsymbol{f}]\|_{L^2_\omega(\Omega_y)}. \tag{4.1}$$

The end goal of operator learning is to construct a neural operator O_{net} from finitely many data that yield a small generalization error uniformly over $\mathcal{X}$.

In what follows, we present a generalization error analysis for DeepONets in terms of the number K of training data, the number m_{x} of input function sensors, the number m_{y} of output function sensors, and the width N of DeepONets. The presented analysis is motivated by an approximation error analysis of DeepONets for the coefficient-to-solution map of elliptic second-order PDEs [33] and we combine it with the analysis [11, 12] of the least-squares approximations to incorporate the output training data.

Let us consider a class $\mathcal{C}_{\text{op}}$ of operators from $\mathcal{X}$ to $\mathcal{Y} = L^2_\omega(\Omega_y)$ which has a spectral form of

$$\mathcal{G}[f](y) = \sum_{j=0}^{\infty} \mathfrak{c}_j(f)\psi_j(y),$$

where $\{\mathfrak{c}_j\}$'s are L_j-Lipschitz functionals in $\mathcal{X}'$ such that $\sum_{j=0}^{\infty} L_j^2 < \infty$, and $\{\psi_j(\cdot)\}_j$ is an orthogonal basis for $L^2_\omega(\Omega_y)$ satisfying

$$\langle \psi_i, \psi_j \rangle_{L^2_\omega} := \int_{\Omega_y} \psi_i(y)\psi_j(y) d\omega(y) = \delta_{ij}.$$

Here δ_{ij} is a Kronecker delta function, and ω is a probability measure on Ω_y. It then can be checked that every operator in $\mathcal{C}_{\text{op}}$ is Lipschitz.

PROPOSITION 4.1. *Any operator $\mathcal{G} \in \mathcal{C}_{op}$ is Lipschitz continuous. We denote $L_{\mathcal{G}}$ as the Lipschitz constant of $\mathcal{G}$.*

Proof. For any $f, f' \in \mathcal{X}$, observe that $\|\mathcal{G}[f] - \mathcal{G}[f']\|^2_{L^2_\omega} = \sum_{j=0}^{\infty} |\mathfrak{c}_j(f) - \mathfrak{c}_j(f')|^2 \le \sum_{j=0}^{\infty} L_j^2 d^2_{\mathcal{X}}(f, f')$, which gives $\|\mathcal{G}[f] - \mathcal{G}[f']\|_{L^2_\omega} \le L_{\mathcal{G}} d_{\mathcal{X}}(f, f')$ with $L_{\mathcal{G}} \le \sqrt{\sum_{j=0}^{\infty} L_j^2}$. ∎

Let $\mathcal{G}_N[f](y) = \sum_{j=0}^{N} \mathfrak{c}_j(f)\psi_j(y)$ be the best N-term approximation of $\mathcal{G}[f]$, and let $\mathcal{E}_N(\mathcal{G}[f]) := \|\mathcal{G}[f] - \mathcal{G}_N[f]\|_{L^2_\omega}$ be the corresponding best N-term approximation error. In what follows, we make a couple of assumptions on $\mathcal{G}$ that guarantee the *uniform boundedness* and the *uniform decay rate* of the best N-term approximation error, inspired by [33].

Assumption 4.2 (operators). For any $\mathcal{G} \in \mathcal{C}_{\text{op}}$, the following are assumed.

1. There is a constant $M > 0$ such that $|\mathcal{G}[f](y)| \le M$ for any $f \in \mathcal{X}$, and for almost every y with respect to ω.
2. Let $\mathcal{E}_N(\mathcal{X}) := \sup_{f \in \mathcal{X}} \mathcal{E}_N(\mathcal{G}[f])$ be the supremum of the best N-term approximation errors over $\mathcal{X}$. Assume that $\mathcal{E}_N(\mathcal{X}) \le N^{-r_{\mathcal{G},\mathcal{X}}}$ for some $r_{\mathcal{G},\mathcal{X}} > 0$ that depends on $\mathcal{X}$, $\mathcal{G}$ and the choice of basis $\{\psi_j\}$.

We note that there are many operators of interest satisfying Assumption 4.2. For example, [33] considered the elliptic boundary value problem

$$-\nabla \cdot (a \nabla u^a) = f$$

for some fixed source term f and studied approximation rates of the data-to-solution operator $\mathcal{G} : a \mapsto u^a$. It was shown that the operator $\mathcal{G}$ satisfies Assumption 4.2.

Since DeepONet requires one to extract finite-dimensional information from an infinite-dimensional class $\mathcal{X}$ for the input, in order to quantify how many input functions are needed to fill up the target domain, we make the following assumptions.

Assumption 4.3 (input functions and sensors). The symbol $\lesssim$ is used to suppress constants that depend only on $(\mathcal{X}, d_{\mathcal{X}})$.

1. For all but finitely many $m_{\text{x}} \in \mathbb{N}$, there exist m_{x} discretization points $\{x_j\}_{j=1}^{m_{\text{x}}}$ in Ω_x satisfying

$$\|\boldsymbol{f} - \boldsymbol{g}\|_{w,2} \lesssim d_{\mathcal{X}}(f, g) + m_{\text{x}}^{-s} \quad \forall f, g \in \mathcal{X},$$

where $\|\cdot\|_{w,2}$ is a weighted Euclidean norm, and $s > 0$ is a constant that depends only on $\mathcal{X}$.

2. For any $K \in \mathbb{N}$, there exist K input functions $\mathcal{X}_K := \{f_1, \dots, f_K\}$ in $\mathcal{X}$ satisfying for any $f \in \mathcal{X}$,

$$\min_{1 \le k \le K} d_{\mathcal{X}}(f, f_k) \lesssim K^{-\alpha} + m_{\mathrm{x}}^{-s} \tag{4.2}$$

for some $\alpha > 0$ which may depend $\mathcal{X}$ and m_{x}.

Many input function spaces $\mathcal{X}$ used in the literature (e.g., [31, 32, 5]) satisfy Assumption 4.3. For the sake of clarity, we present three detailed examples.

Example 4.4. Let

$$\mathcal{X} = \left\{ f : [-1,1]^2 \to \mathbb{R} \mid \kappa \in [0,1], f(x) = \kappa \text{ if } \|x\|_2 \le 1, \text{ and 1 otherwise} \right\},$$

and define a map $d_{\mathcal{X}}$ over $\mathcal{X} \times \mathcal{X}$ such that $d_{\mathcal{X}}(f,g) := |f(0) - g(0)|$ for any $f, g \in \mathcal{X}$. It can be checked that $d_{\mathcal{X}}$ is a metric on $\mathcal{X}$. For a given set of points $\{x_j\}_{j=1}^{m_{\mathrm{x}}}$ where there exists a point whose $\|\cdot\|_2$-norm is less than or equal to 1, let $w_j = 0$ if $\|x_j\| > 1$ and $w_j = \frac{1}{|\{x_i : \|x_i\| \le 1\}|}$ if $\|x_j\| \le 1$. For $f \in \mathcal{X}$, let $\boldsymbol{f} = (f(x_1), \dots, f(x_{m_{\mathrm{x}}}))$ and $\|\boldsymbol{f}\|_{w,2}^2 := \sum_{j=1}^{m_{\mathrm{x}}} w_j f(x_j)^2$. It then can be checked that $\|\boldsymbol{f} - \boldsymbol{g}\|_{w,2} = d_{\mathcal{X}}(f,g)$ for all $f, g \in \mathcal{X}$, which satisfies Assumption 4.3.1.

For any $K \in \mathbb{N}$, let $\mathcal{X}_K = \{f_1, \dots, f_K\}$, where $f_i \in \mathcal{X}$ and $f_i(0) = \frac{i}{K}$. It then can be checked that for any $f \in \mathcal{X}$, there exists $g \in \mathcal{X}_K$ such that $d_{\mathcal{X}}(f,g) \le K^{-1}$, which shows that Assumption 4.3.2 holds.

Example 4.5. Let

$$\mathcal{X} = \{ f \in C^1([-1,1]) \mid \|f\|_{C^1} \le 1 \}$$

with $d_{\mathcal{X}}(f,g) = \|f - g\|_{L^2}$ for any $f, g \in \mathcal{X}$. Let $\{(x_j, w_j)\}_{j=1}^{m_{\mathrm{x}}}$ be the Gauss–Legendre quadrature points and weights. Observe that for any $f \in \mathcal{X}$, we have $\|\boldsymbol{f}\|_{w,2} = \|\Pi_{m_{\mathrm{x}}} f\|_{L^2}$, where $\Pi_{m_{\mathrm{x}}} f$ is the Lagrange interpolation of f. Therefore, for any $f, g \in \mathcal{X}$,

$$\begin{aligned} \|\boldsymbol{f} - \boldsymbol{g}\|_{w,2} = \|\Pi_{m_{\mathrm{x}}}(f-g)\|_{L^2} &\le \|f - g\|_{L^2} + \|(f-g) - \Pi_{m_{\mathrm{x}}}(f-g)\|_{L^2} \\ &\lesssim d_{\mathcal{X}}(f,g) + m_{\mathrm{x}}^{-1}, \end{aligned}$$

which shows that Assumption 4.3.1 holds.

For any $n \in \mathbb{N}$, let $K = (n+1)^{m_{\mathrm{x}}}$, and consider $\mathcal{X}_K = \{f \in \mathcal{X} : \boldsymbol{f} \in \{-1 + \frac{i}{n}(2) : i = 0, \dots, n\}^{m_{\mathrm{x}}}\}$. It then can be checked that for any $f \in \mathcal{X}$, there exists $g \in \mathcal{X}_K$ such that

$$\begin{aligned} d_{\mathcal{X}}(f,g) &\le \|\Pi_{m_{\mathrm{x}}}(f-g)\|_{L^2} + C m_{\mathrm{x}}^{-1} = \|\boldsymbol{f} - \boldsymbol{g}\|_{w,2} + C m_{\mathrm{x}}^{-1} \\ &\le \|\boldsymbol{f} - \boldsymbol{g}\|_{\infty} + C m_{\mathrm{x}}^{-1} \lesssim K^{-\frac{1}{m_{\mathrm{x}}}} + m_{\mathrm{x}}^{-1}, \end{aligned}$$

which shows that Assumption 4.3.2 holds.

Example 4.6. Let $\mathcal{X} = \{f \in H_\omega^p([-1,1]) : \|f\|_{H_\omega^p} \le 1\}$ where $H_\omega^p([-1,1])$ is a weighted Sobolev space. For any $f \in H_\omega^p$, let $\boldsymbol{f} = (\hat{f}_0, \dots, \hat{f}_{m_{\mathrm{x}}})$ with $\hat{f}_k = \langle f, p_k \rangle_{L_\omega^2([-1,1])}$ where p_k is the orthonormal polynomial of degree k with respect to ω. It then follows from the well-known spectral convergence [22] that for any $f, g \in \mathcal{X}$,

$$\|\boldsymbol{f} - \boldsymbol{g}\|_2 \lesssim \|f - g\|_{L_\omega^2} + m_{\mathrm{x}}^{-p},$$

implying Assumption 4.3.1.

For any $n \in \mathbb{N}$, let $K = (n+1)^{m_{\mathrm{x}}}$, and consider $\mathcal{X}_K = \{f \in \mathcal{X} : \hat{f} \in \{\frac{i}{n} : i = 0, \dots, n\}^{m_{\mathrm{x}}}$. Let $P_{m_{\mathrm{x}}} f = \sum_{j=0}^{m_{\mathrm{x}}} \hat{f}_j p_k$. It then can be checked that for any $f \in \mathcal{X}$, there exists $g \in \mathcal{X}_K$ such that

$$\begin{aligned} d_{\mathcal{X}}(f,g) &\le \|P_{m_{\mathrm{x}}} f - g\|_{L^2_\omega} + \|f - P_{m_{\mathrm{x}}} f\|_{L^2_\omega} \lesssim \|\boldsymbol{f} - \boldsymbol{g}\|_2 + m_{\mathrm{x}}^{-p} \\ &\le \|\boldsymbol{f} - \boldsymbol{g}\|_\infty + m_{\mathrm{x}}^{-p} \lesssim K^{-\frac{1}{m_{\mathrm{x}}}} + m_{\mathrm{x}}^{-p}, \end{aligned}$$

implying Assumption 4.3.2.

The following assumption is the one that draws a connection between the proposed two-step training method and the generalization analysis. Roughly speaking, we generalize the assumption on the number of output sensors introduced in [11, 12] for a class of trunk neural networks on which orthonormal basis can be formed with respect to a given measure defined on Ω_y. This assumption is crucial as it allows one to utilize a classical least-squares analysis in the context of DeepONets.

Assumption 4.7 (trunk networks and sensors). Let F be a feasible set of trunk network parameters defined by

$$\mathrm{F} = \left\{ \mu \in \mathrm{F}_t : \sup_{y \in \Omega_y} \|\hat{\boldsymbol{\phi}}(\cdot; \mu, T_\mu)\|_2^2 < \infty \right\}, \tag{4.3}$$

where $\mathrm{F}_t = \{\mu : \exists\, T_\mu$ such that $\hat{\boldsymbol{\phi}}(\cdot; \mu, T_\mu)$ forms orthonormal basis in $L^2_\omega(\Omega_y)\}$. Let $\{y_1, \dots, y_{m_{\mathrm{y}}}\}$ be a set of discretization points randomly independently drawn from the probability measure ω. For $r_t > 0$, suppose m_{y} is sufficiently large enough to satisfy

$$\sup_{\mu \in \mathrm{F}} \left(\sup_{y \in \Omega_y} \|\hat{\boldsymbol{\phi}}(\cdot; \mu, T_\mu)\|_2^2 \right) \le \kappa \frac{m_{\mathrm{y}}}{\log m_{\mathrm{y}}}, \qquad \kappa := \frac{3\log(3/2) - 1}{2 + 2r_t}. \tag{4.4}$$

Last, we introduce assumptions for branch networks. For simplicity, we confine ourselves to a two-layer neural network of sufficiently large width so that one achieves a zero loss on (3.5).

Assumption 4.8 (branch networks). The following are assumed for branch networks.

1. The branch network is a two-layer neural network whose activation function σ is Lipschitz continuous with the Lipschitz constant L_σ.
2. For each K, there exists a two-layer branch network of width n_K that achieves a zero loss (3.5). That is, there exists θ^* such that $\boldsymbol{C}(\theta^*) = R^* A^*$. Specifically, let $\theta^* = \{\gamma_\ell, \beta_\ell, w_\ell\}_{\ell=1}^{n_K}$ where $\gamma_\ell \in \mathbb{R}^{N+1}$, $w_\ell \in \mathbb{R}^{m_{\mathrm{x}}}$, $\beta_\ell \in \mathbb{R}$. Then, for $k = 1, \dots, K$,
$$\boldsymbol{c}(\boldsymbol{f}_k; \theta^*) := \sum_{\ell=1}^{n_K} \gamma_\ell \sigma\left(\langle w_\ell, \boldsymbol{f}_k \rangle + \beta_\ell\right) = (R^* A^*)_k \in \mathbb{R}^{N+1},$$
where $(R^* A^*)_k$ is the kth column of $R^* A^*$ from (3.5).
3. Let $L_c(K, N, m_{\mathrm{x}}) := \sum_{\ell=1}^{n_K} \|\gamma_\ell\|_2 \|w_\ell\|_2$. Suppose $L_c(K, N, m_{\mathrm{x}})$ is uniformly bounded independent of K, N, and m_{x}, and denote its upper bound by $\overline{L}_c$.

Remark 4.9. The assumptions of Assumption 4.8 are mild and easily satisfied in many practical setups. The last assumption corresponds to the uniform boundedness of the Lipschitz constant for the branch networks, which is often used in the literature (e.g., [38]) to establish a convergence.

We are now in a position to present the main theorem that characterizes the generalization error of the fully trained DeepONets in terms of the number K of training data, the number m_{x} of input function domain sensors, the number m_{y} of output function domain sensors, and the width N of DeepONets.

THEOREM 4.10. *Suppose Assumptions* 4.2, 4.3, 4.7, *and* 4.8 *hold. Let* O_{net} *be the fully trained DeepONet, that is, the trunk networks are obtained from* (3.4) *with* $p=2$ *and* $\mu^* \in F$ *defined as in Assumption* 4.7, *and the branch network solves* (3.5). *Given a truncation operator* $\mathfrak{T}_M(z) = sign\ (z)\max\{M, |z|\}$, *let* $\widetilde{O}_{net}[\boldsymbol{f}](y) := \mathfrak{T}_M(O_{net}[\boldsymbol{f}](y))$. *Then, for any* $f \in \mathcal{X}$,

$$\mathbb{E}\left[\mathcal{E}^2_{gen}(\widetilde{O}_{net}[\boldsymbol{f}])\right] \lesssim C(m_y, r_t)N^{-r_{\mathcal{G},\mathcal{X},\mu^*}} + K^{-\alpha} + m_x^{-s} + m_y^{-r_t}, \tag{4.5}$$

where $C(m_y, r_t) = 1 + \frac{6\log(3/2)-2}{(1+r_t)\log m_y}$ *and the expectation is taken over all random output function sensors* $\{y_i\}_{i=1}^{m_y}$. *All the hidden constants are independent of* K, m_x, m_y, *and* N *but may only depend on* M, $L_{\mathcal{G}}$, L_σ, $\overline{L}_c$, *and* $(\mathcal{X}, d_{\mathcal{X}})$.

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

Remark 4.11. The rate $r_{\mathcal{G},\mathcal{X},\mu^*}$ of convergence with respect to the width N of DeepONet is affected by the trunk network.

5. Numerical examples. In this section, we present several numerical experiments to demonstrate the performance of the proposed two-step training method. Throughout, the two-step training method as described in section 3.3 is referred to as `2ST` and the vanilla monolithic training method is referred to as `VAN`. To illustrate the importance of the Gram–Schmidt orthonormalization (implemented by QR-factorization) in the first step, the one without it shall be referred to as `2STw/oQR`. In all the numerical tests, we employ the standard unstacked DeepONet structure proposed in [31].

Darcy's flow equation. In the following numerical examples, we consider Darcy's flow equation in a bounded domain $\Omega = (-1, 1)^2$ with Lipschitz boundary $\partial\Omega$:

$$\begin{aligned} -\nabla \cdot (\alpha(p)\nabla p) &= f \quad \text{in } \Omega, \\ p &= g \quad \text{on } \Gamma_D, \\ -\alpha(p)\nabla p \cdot \mathbf{n} &= h \quad \text{on } \Gamma_N, \end{aligned} \tag{5.1}$$

where $p : \Omega \to \mathbb{R}$ is the scalar pressure and f is the body force. A Dirichlet boundary condition is imposed on Γ_D and a Neumann boundary condition is given on $\Gamma_N = \partial\Omega\backslash\Gamma_D$ with $\mathbf{n}$ as the unit outward normal vector on Γ_N. We note that the conductivity $\alpha := \alpha(p)$ could yield the equation to be nonlinear. In what follows, we will consider three different operators that arise from (5.1).

Data generation. We employ the classical Lagrange continuous Galerkin linear finite element method (FEM) to generate the data. Both finite element libraries, deal.II [6] and FEniCS [4], were utilized. Once the data is generated, we split it into the training data and the test data. The training data is used for training of DeepONet and the test data is used to evaluate the performance of the trained DeepONet.

Inference on unseen data. For a test input function $\boldsymbol{f}_{\mathrm{test}}$, the DeepONet produces an approximation to the corresponding output function $u_{\mathrm{test}} := \mathcal{G}[f_{\mathrm{test}}]$. Let $\{y_i^{(\mathrm{test})}\}_{i=1}^{M_{\mathrm{test}}}$ be a set of points from Ω_y to be used for evaluating the generalization ability. Let $\boldsymbol{u}_{\mathrm{test}} = (u_{\mathrm{test}}(y_1^{(\mathrm{test})}), \ldots, u_{\mathrm{test}}(y_{M_{\mathrm{test}}}^{(\mathrm{test})}))^\top$ be the discretization of u_{test}, which is not available in practice. We measure the generalization ability of the DeepONet by means of the relative ℓ_2 error defined by

$$\mathcal{E}_{\mathrm{rel}}(O_{\mathrm{net}}[\boldsymbol{f}_{\mathrm{test}}]) := \frac{\sqrt{\sum_{i=1}^{M_{\mathrm{test}}} \left(O_{\mathrm{net}}[\boldsymbol{f}_{\mathrm{test}}](y_i^{(\mathrm{test})}) - u_{\mathrm{test}}(y_i^{(\mathrm{test})})\right)^2}}{\|\boldsymbol{u}_{\mathrm{test}}\|_2}. \tag{5.2}$$

Conditional optimality. If $\boldsymbol{u}_{\mathrm{test}}$ were known, by fixing the trunk network, one can obtain the optimal value $\boldsymbol{a}^*_{\mathrm{test}}$ for the branch network at $\boldsymbol{f}_{\mathrm{test}}$ by solving

$$\boldsymbol{a}^*_{\mathrm{test}} = \begin{cases} \mathrm{argmin}_{\boldsymbol{a}\in\mathbb{R}^{N+1}} \|\boldsymbol{\Phi}_{\mathrm{test}}(\boldsymbol{\mu}^*)\boldsymbol{a} - \boldsymbol{u}_{\mathrm{test}}\|_2 & \text{with the monolithic method,} \\ \mathrm{argmin}_{\boldsymbol{a}\in\mathbb{R}^{N+1}} \|\boldsymbol{\Phi}_{\mathrm{test}}(\boldsymbol{\mu}^*)T^*\boldsymbol{a} - \boldsymbol{u}_{\mathrm{test}}\|_2 & \text{with the two-step method,} \end{cases}$$

where $\boldsymbol{\Phi}_{\mathrm{test}}(\boldsymbol{\mu}^*)$ is the matrix whose ith row is $\boldsymbol{\phi}^\top(y_i^{(\mathrm{test})}; \mu^*)$. Let

$$O^*_{\mathrm{net}}[\boldsymbol{f}_{\mathrm{test}}](y) := \begin{cases} \boldsymbol{\phi}^\top(y;\mu^*)\boldsymbol{a}^*_{\mathrm{test}} & \text{with the monolithic method,} \\ \boldsymbol{\phi}^\top(y;\mu^*)T^*\boldsymbol{a}^*_{\mathrm{test}} & \text{with the two-step method.} \end{cases} \tag{5.3}$$

We then define $\mathcal{E}_{\mathrm{rel}}(O^*_{\mathrm{net}}[\boldsymbol{f}_{\mathrm{test}}])$ as the *optimal* relative ℓ_2 error. Here the optimality shall be understood as conditional in the sense that given $\{y_i^{(\mathrm{test})}\}$ and the trunk network $\boldsymbol{\phi}(\cdot;\mu^*)$, $O^*_{\mathrm{net}}[\boldsymbol{f}_{\mathrm{test}}]$ is the least-squares approximation to u_{test}. However, this optimality is not available in practice as O^*_{net} requires the target function $\boldsymbol{u}_{\mathrm{test}}$ to obtain $\boldsymbol{a}^*_{\mathrm{test}}$. The optimal relative ℓ_2 errors by `2ST` and `VAN` are referred to as `Opt-2ST` and `Opt-VAN`, respectively.

5.1. Forward problem: Nonlinear conductivity. Let us consider a specific case of (5.1). Let $f = 1$, $g = \cos(x)$, $\partial\Omega = \Gamma_D$ and the conductivity coefficient be $\alpha(p) = \kappa p$, where κ is a constant function. The operator $\mathcal{G}$ of interest is

$$\mathcal{G} : \mathcal{X} \ni \kappa(\cdot) \mapsto p(\cdot) \in \mathcal{Y},$$

where $\mathcal{X} = \{\kappa \mid \kappa(x,y) = \beta\ \forall (x,y) \in \Omega, \beta \in [1,1000]\}$ and $\mathcal{Y}$ is an appropriate space where the solution p lies. Note that for any $\kappa \in \mathcal{X}$, it is well-known [16] that there exists a unique solution $p(\cdot)$ of the system (5.1).

Since the input functions are constant functions, we simply set $m_{\mathrm{x}} = 1$, e.g., $x_1 = (0,0)$. Accordingly, the input data are generated as the collection of 1000 equidistant β values in $[1,1000]$, i.e., $\{1,2,\ldots,1000\}$. The corresponding output data are obtained by the FEM solver on 2049 grid points, i.e., $m_{\mathrm{y}} = 2049$. The data are then randomly split in two—900 of them are used as training and the remaining 100 are used as test data. We employ a DeepONet whose branch and trunk architectures are $\vec{\boldsymbol{n}}_b = (1,500,51)$ and $\vec{\boldsymbol{n}}_t = (2,50,50,50,50)$, respectively. Both branch and trunk networks use the ReLU activation function and were initialized by the He initialization scheme [21]. Throughout, we employ the `Adam` optimizer [24] with full-batch.

In Figure 2(a), we plot the training loss versus the number of iterations by both `2ST` and `VAN`. Specifically, the training loss refers to the trunk network loss which is defined in (3.4) for `2ST` and the standard overall loss of (3.2) for `VAN`. It can be clearly seen that the loss by `2ST` is roughly two orders of magnitude smaller than the one by `VAN`. As a matter of fact, the smallest loss attained by `2ST` is 2.33×10^{-7}, while the one by `VAN` is 2.08×10^{-5}. This is not a single isolated case. We tested five independent simulations, and the averaged smallest loss achieved by the two methods is 2.11×10^{-7} and 1.84×10^{-5} for `2ST` and `VAN`, respectively. This demonstrates the effectiveness of the proposed two-step method for learning the trunk network. The remaining task for `2ST` is then to learn the branch network following (3.5).

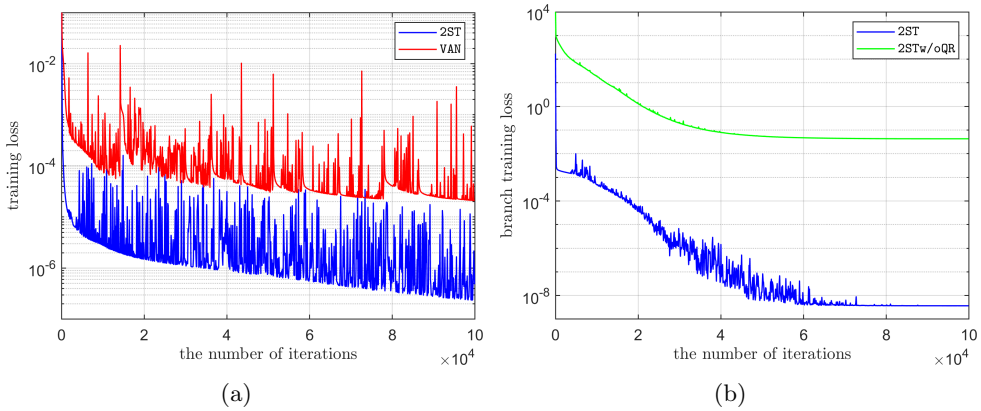


FIG. 2. *Example* 5.1. *(Left) The training loss versus the number of iterations. Here the loss refers to* (2.5) *for* `VAN` *and* (3.4) *for* `2ST` *(thus* `2STw/oQR`*). (Right) The branch loss* (3.5) *is reported for* `2ST` *and* `2STw/oQR`*. This shows the effectiveness of orthogonalization in the second step of the proposed training method. (Color online.)*

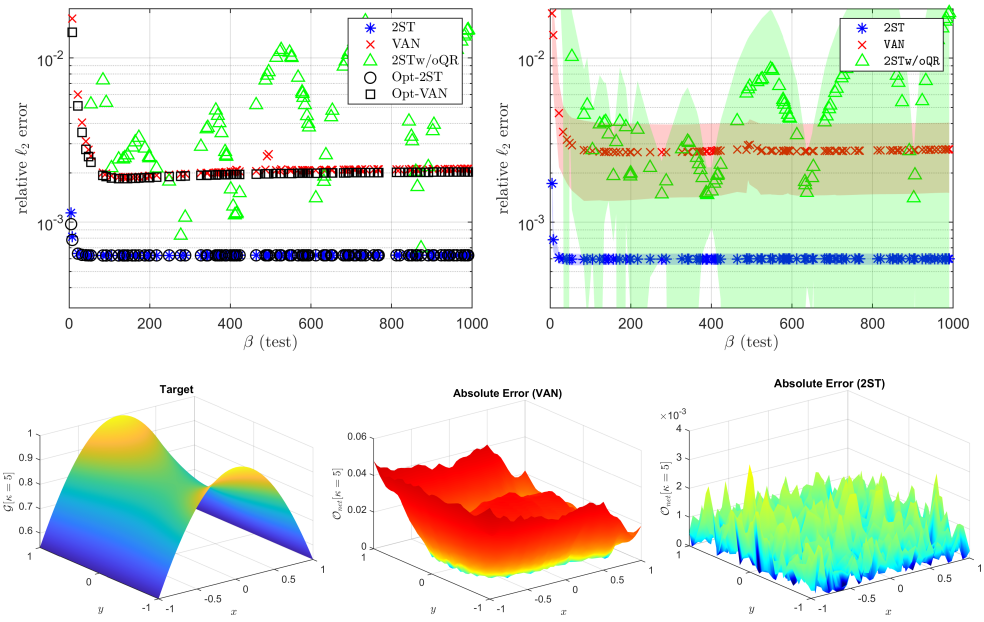


FIG. 3. *Example* 5.1. *(Top) The relative* ℓ_2 *errors versus the* 100 *test* β *values by* `2ST`*,* `VAN`*, and* `2STw/oQR`*. (Bottom) Left: The graph of the target operator evaluated at a test function* $\kappa = 5$*, which is the smallest value lying in the test data. Middle and right: The absolute error maps by* `VAN` *(middle) and* `2ST` *(right).*

In Figure 2(b), the training loss for the branch network versus the number of iterations is plotted. We can see that the loss reaches the level of 10^{-9} by `2ST` which utilizes QR-factorization, while the one without QR cannot reach a similar level. This demonstrates the effectiveness of the orthogonalization in the proposed two-step training method.

In the top left of Figure 3, the relative ℓ_2 errors defined in (5.2) are reported. The errors are plotted with respect to the 100 test β values, where the β are constant input

functions. The results of `2ST`, `VAN`, and `2STw/oQR` are marked as asterisks ($*$), crosses ($\times$), and triangles ($\triangle$), respectively. The optimal relative ℓ_2 errors defined in (5.3) are shown as circles ($\circ$, `Opt-2ST`) for `2ST` and squares ($\square$, `Opt-VAN`) for `VAN`, and they serve as references. We note that having small optimal relative errors means the trunk network was well-trained to represent unseen output functions. In a similar vein, it is also worth noting that if the relative errors of `VAN` (`2ST`) are close to the optimal ones of `VAN` (`2ST`), it implies that the branch network was well-trained, allowing the DeepONet to generalize effectively. It is clearly observed that `2ST` achieves the smallest relative ℓ_2 errors being almost identical to the optimal ones, while `2STw/oQR` yields much higher and unstable errors. This again demonstrates the effectiveness of orthogonalization in the proposed method. `VAN` is able to produce relative errors similar to optimal ones; however, the relative optimal errors of `VAN` are much higher than the ones by `2ST`. This indicates the ineffectiveness of the monolithic training for learning the trunk network. On the contrary, the proposed two-step training method `2ST` effectively trains not only the trunk network but also the branch network. On the top right, we report the means of the relative ℓ_2 errors from five independent simulations. The shaded area is the area that falls within one standard deviation of the mean. On the bottom left, we report the graph of the target output function for $\beta = 5$ (the smallest value belongs to the test data). On the bottom middle and right, the absolute error maps by `VAN` and `2ST` are shown, respectively. It is clearly observed that the absolute error by `2ST` is at least one order magnitude smaller than the one by `VAN`.

5.2. Inverse problem: Discontinuous conductivity. In this case, let $f = 0$ and consider a piecewise constant (discontinuous) conductivity $\alpha := \kappa(\cdot;\beta)$ where κ is defined by

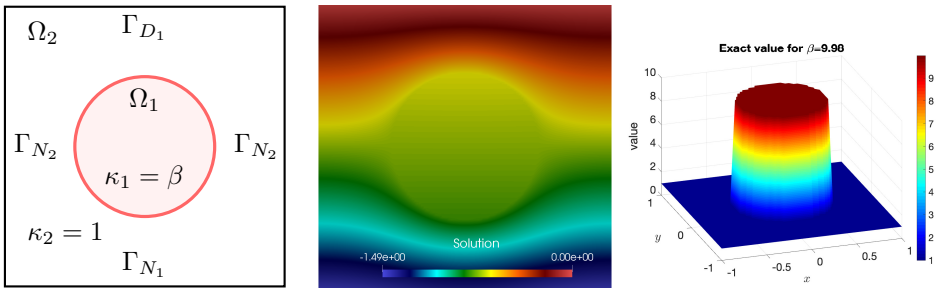
$$\kappa(x,y;\beta) = \begin{cases} \beta & \text{if } (x,y) \in \Omega_1, \\ 1 & \text{if } (x,y) \in \Omega \backslash \Omega_1, \end{cases} \tag{5.4}$$

where Ω_1 is a disk centered at the origin $(0,0)$ with radius $r = 0.5$. The Dirichlet and Neumann boundary conditions are imposed by

$$p = 0 \text{ on } \Gamma_{D_1}, \quad -\kappa \nabla p \cdot \mathbf{n} = 0 \text{ on } \Gamma_{N_2}, \quad -\kappa \nabla p \cdot \mathbf{n} = 1 \text{ on } \Gamma_{N_1}. \tag{5.5}$$

Figure 4(a) shows the detailed geometry of the problem.

We are interested in the inverse problem of (5.1) with the boundary conditions of (5.5). That is, the operator $\mathcal{G}$ of interest maps a given solution p to the corresponding



(a) Domain Geometry. (b) $p(\cdot)$. (c) $\kappa(\cdot;\beta = 9.98)$.

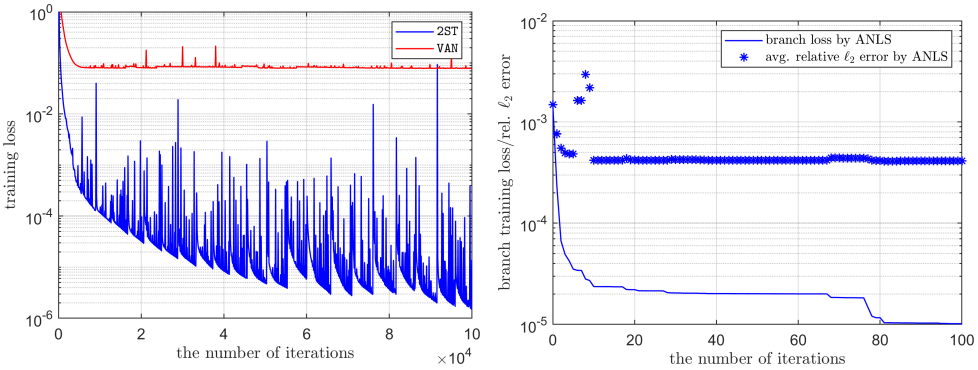
FIG. 4. *Example* 2. (a) *Problem domain with detailed geometries.* (b) *The solution* p *of* (5.1) *with the boundary conditions* (5.5) *at* $\kappa(\cdot;\beta = 9.98)$. (c) *The graph of* $\kappa(\cdot;\beta = 9.98)$.

conductivity coefficient κ as a function. Specifically, $\mathcal{G} : \mathcal{X} \ni p(\cdot) \mapsto \kappa(\cdot) \in \mathcal{Y}$ where $\mathcal{X}$ is the collection of solutions obtained at various $\kappa(\cdot;\beta)$ defined in (5.4) where $\beta \in [0.01, 10]$ for κ, and $\mathcal{Y}$ is the collection of the corresponding $\kappa(\cdot;\beta)$. Figures 4(b) and 4(c) show an input-output pair for the operator.

For data generation, we consider the collection of 1000 β values—$\{\beta = j \times 0.01 : j \in \{1, 2, \ldots, 1000\}\}$—and solve the corresponding equation (5.1) to obtain p. This is done by the FEM solver on $m_{\mathrm{x}} = 4225$ points in Ω. On that exact grid, i.e., $m_{\mathrm{y}} = 4225$, the output function data are generated according to (5.4). The data are then randomly split into 900 training data and 100 test data. We employ a DeepONet with the trunk and branch architectures of $\vec{\boldsymbol{n}}_t = (2, 25, 25, 25, 25, 25)$ and $\vec{\boldsymbol{n}}_b = (4225, 100, 26)$, respectively, with the ReLU activation function. Both are initialized with the He scheme [21]. Since the effectiveness of QR was already demonstrated in the previous example, here we only consider `2ST`.

Figure 5(a) shows the training loss versus the number of iterations by both `2ST` and `VAN`. Specifically, it displays the trunk network training loss for `2ST` and the overall standard loss for `VAN`. We employ the `Adam` optimizer [24] with full-batch and its default hyperparameters. It is clearly observed that the two-step training method `2ST` minimizes the loss to the level of 10^{-6}, while the monolithic standard training `VAN` stagnates at the level of 10^{-1} after 10,000 iterations. This again indicates that `2ST` effectively trains the trunk network to represent the output functions, which are the piecewise constant functions (5.4).

For the second step (3.5) of `2ST`, we employ the ANLS training method developed in [1, 2]. This is possible because the second step is merely a standard regression task on which ANLS is applicable. In Figure 5(b), the branch training loss is plotted with respect to the number of ANLS iterations. ANLS minimizes the branch loss to the level of 1.02×10^{-5} merely within 100 iterations. The averaged relative ℓ_2 errors over the 100 test data are also reported. It can be seen that the average test errors are saturated after merely 20 ANLS iterations. This indicates that the branch network is successfully trained by ANLS. We remark that ANLS is not applicable to the monolithic training of DeepONets.



(a) (b)

FIG. 5. *Example* 5.2. *(Left) The training loss versus the number of iterations. Here the loss refers to* (2.5) *for* `VAN` *and* (3.4) *for* `2ST`*. (Right) The branch training loss of* `2ST` *versus the number of ANLS* [1, 2] *iterations. Also, the average relative* ℓ_2 *errors over the* 100 *test data are marked by asterisks* $(*)$*.*

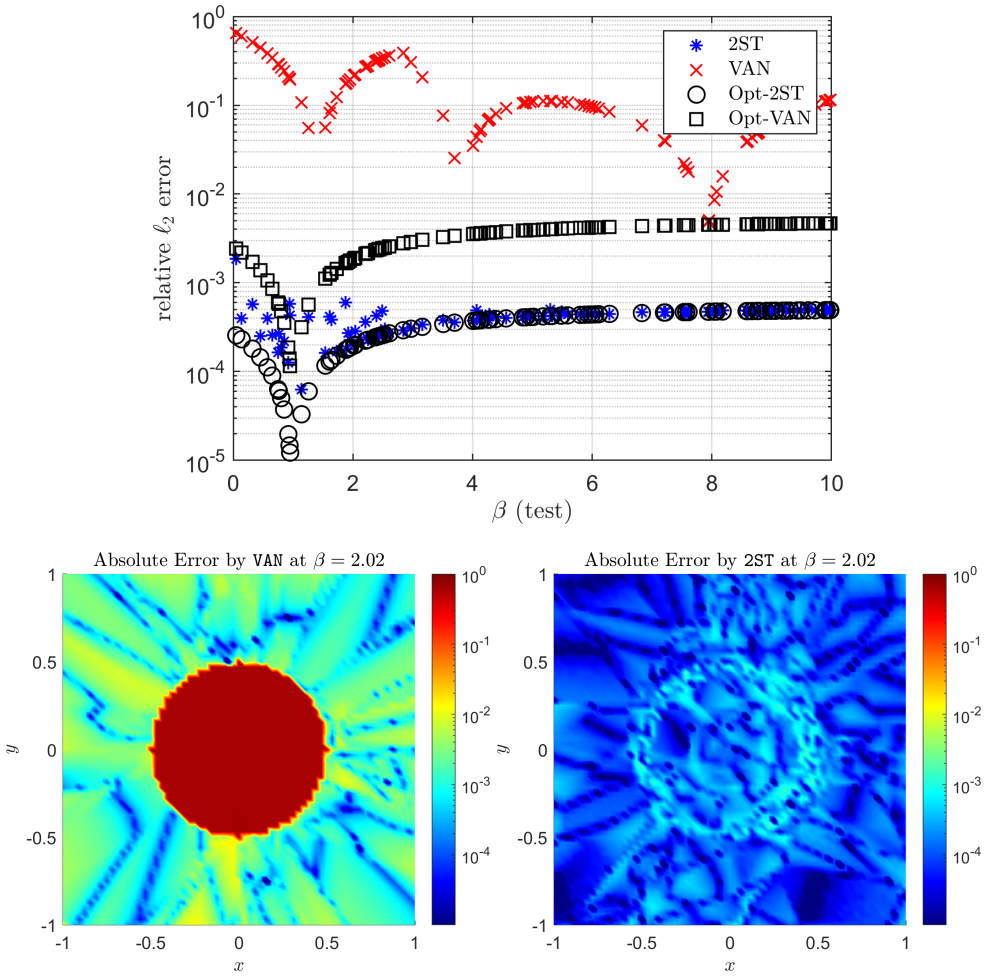


FIG. 6. *Example* 5.2. *(Top) The relative ℓ_2 errors on the test data are reported for* 2ST *and* VAN. *(Bottom) The absolute error maps for $\beta = 2.02$ (test) by* VAN *(left) and* 2ST *(right).*

At the top of Figure 6, we plot the relative ℓ_2 errors for the 100 test data. Since each test datum is determined by its corresponding β value, the errors are plotted with respect to the test β values. The optimal relative ℓ_2 errors (5.3) are also reported and serve as reference benchmarks. The optimal errors for VAN and 2ST are indicated by squares (□, Opt-VAN) and circles (○, Opt-2ST), respectively. Again note that these optimal values are not available in practice as the underlying target output function data are required. Therefore, the closer to the optimal values from the trained DeepONet, the better the generalization performance it implies. It can be seen that the optimal errors by 2ST are roughly one order of magnitude smaller than those by VAN. This indicates that 2ST can train the trunk network more effectively than VAN. Furthermore, it is clearly observed that the relative test errors by 2ST are close to the optimal ones and are even almost identical especially if $\beta \in [3, 10]$. On the other hand, the test errors by VAN are way off from the corresponding optimal ones. The averaged relative error over the 100 test data by VAN and 2ST are 1.48×10^{-1} and 4.14×10^{-4}, respectively. On the bottom of Figure 6, the absolute error maps of

DeepONets trained by both `VAN` and `2ST` at $\beta = 2.02$ (test) are shown. It can be seen that while both capture well the discontinuity, `2ST` can accurately predict the value on the circle. This again demonstrates the effectiveness of the proposed two-step training method over the standard monolithic one.

5.3. Multiple inputs and nonlinear conductivity. Let us consider the nonlinear conductivity $\alpha(p) := \kappa p$ and the Dirichlet boundary condition (i.e., $\Gamma_N = \emptyset$) in (5.1). In this case, we illustrate the capabilities of the proposed algorithm by considering the solution operator whose inputs are the triplet of the right-hand-side source term f, the conductivity κ, and the boundary value g. That is, the operator of interest is

$$\mathcal{G} : \mathcal{X} \ni (f, \kappa, g) \mapsto p \in \mathcal{Y}.$$

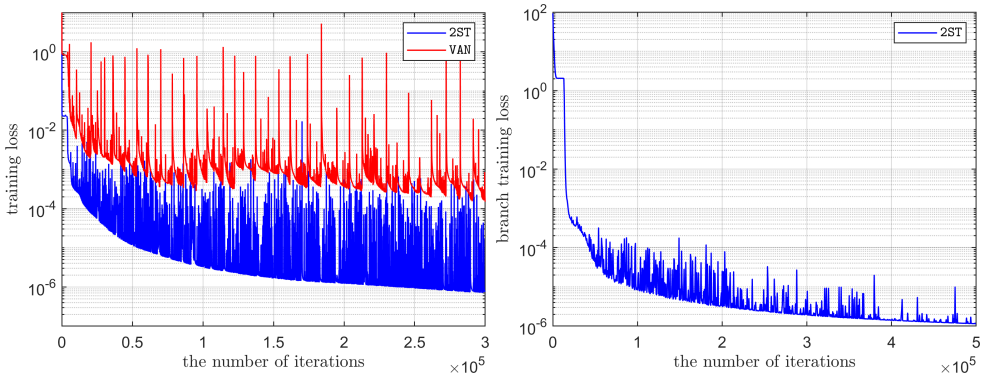
The input function space is the set of triplets $\mathcal{X} = \{(f, \kappa, g) | f \in \mathcal{F}, \kappa \in \mathcal{K}, g \in \mathcal{G}\}$, where $\mathcal{F}$, $\mathcal{K}$, $\mathcal{G}$ are all the collection of constant functions in the range of $[0.1, 10]$. The output space $\mathcal{Y}$ is the collection of the corresponding solutions p to the system (5.1). Note that for any $(f, \kappa, g) \in \mathcal{X}$, there exists a unique solution p to (5.1).

To generate a dataset, we consider the grid of 1 M points in $[0.1, 10]^3$,

$$\left\{ \left(\frac{i}{10}, \frac{j}{10}, \frac{k}{10} \right) : i, j, k \in \{1, \ldots, 100\} \right\},$$

and each element represents the triplet of the three constant functions (f, κ, g). Then, we randomly select 100,000 grid points out of 1 M and solve (5.1) to obtain the corresponding solutions p on $m_y = 541$ points. The 100,000 data are split into 90,000 training data and 10,000 test data. Since the input functions are the triplet of constant functions, we use the corresponding grid as the input for DeepONets. We employ the trunk and branch networks whose architectures are $\vec{n}_t = (2, 100, 100, 100, 200)$ and $\vec{n}_b = (3, 100, 100, 100, 201)$, respectively. The hyperbolic tangent (tanh) activation function is used for both and the Xavier initialization scheme [17] and the `Adam` optimizer [24] is utilized.

In Figure 7(a), the training loss versus the number of iterations is reported. Again, it is clearly seen that `2ST` can effectively minimize the trunk network loss reaching the



(a) (b)

FIG. 7. *Example* 5.3. *(Left) The training loss versus the number of iterations by* `VAN` *and* `2ST`*. (Right) The branch training loss versus the number of iterations. This is the second step of the proposed two-step training method.*

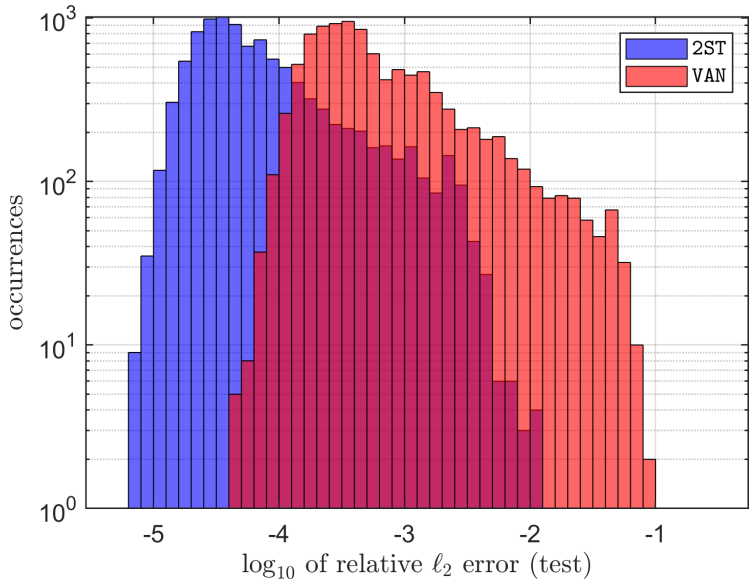


FIG. 8. *Example* 5.3. *The histogram of* $\log_{10}$ *of the relative* ℓ_2 *errors on the* 10,000 *test data obtained by* `2ST` *(blue) and* `VAN` *(red).*

level of 10^{-7} at the end of the training, while `VAN` stagnates at the level of 10^{-4}. This again confirms that `2ST` effectively trains the trunk network when it is compared with `VAN`. The remaining job for `2ST` is to train the branch network according to (3.5).

Figure 7(b) shows the branch loss in the second step of `2ST` with respect to the number of `Adam` iterations. Here, we use a learning rate scheduler that starts at 10^{-3} and reduces the learning rate by a factor of 2 for every 100K iteration. It is observed that the branch loss is sufficiently minimized at the end of the training and reaches the level of 10^{-6}.

Last, we report the histogram of $\log_{10}$ of the relative ℓ_2 errors on the 10,000 test data in Figure 8. It is clearly observed that the DeepONet trained by `2ST` yields a much better generalization performance than the one by `VAN`. The average relative test error by the proposed two-step method is 2.9×10^{-4}, while the one by the vanilla monolithic training is 2.5×10^{-3}. This clearly indicates that how DeepONets are trained makes a significant impact on generalization performance. We emphasize that the only change we make is the training method, while the network architecture, data, and initialization schemes were identical throughout.

6. Conclusions. In this study, we explored a novel training technique for DeepONets. The newly introduced sequential two-step training approach involves initial training of the trunk network that involves the Gram–Schmidt orthonormalization by means of QR-factorization, followed by the training of the branch network. The efficacy of the two-step training method was assessed through various numerical experiments, contrasting its performance against the conventional monolithic training approach, involving both forward and inverse Darcy problems within porous media contexts. The efficacy and robustness of the proposed approach were clearly showcased in these representative examples, underscoring the significance of having robust training algorithms tailored to a specific neural network architecture. Moreover, the significance of pretraining the trunk network was emphasized, as it provided valuable

insights into the outcomes of the complete training process. This approach resulted in crucial improvements in accuracy while also simplifying the overall complexity of the training. Last, a generalization error estimate is established by leveraging the least-squares error analysis of [11, 12] in terms of the number of training data, the number of input and output sensor points, and the width of DeepONets.

Appendix A. Proof of Theorem 3.5.

Proof. Let the rank of $\boldsymbol{U}$ be r and let $Z\Sigma_r V^\top$ be an SVD of $\boldsymbol{U}$ where $Z \in \mathbb{R}^{m_{\mathrm{y}} \times r}$, $\Sigma_r \in \mathbb{R}^{r\times r}$, and $V \in \mathbb{R}^{K\times r}$. Let

$$\tilde{A} = \begin{cases} \begin{pmatrix} \Sigma_r V^\top \\ \mathbf{0}_{(N-r)\times K} \end{pmatrix} & \text{if } N \geq r, \\ \Sigma_{1:N} V_{1:N}^\top & \text{if } N < r, \end{cases}$$

where $\Sigma_{1:s}$ is a diagonal matrix of size $s \times s$ obtained from Σ_r by collecting the first s rows and columns, and $V_{1:s}$ is obtained by collecting the first s columns of V. It then can be checked that if the trunk network satisfies

$$\boldsymbol{\phi}_0^\top(y_i;\mu) = \begin{cases} (Z^{(i)}, \mathbf{0}_{1\times(N-r)}) & \text{if } N \geq r, \\ Z_{1:N}^{(i)} & \text{if } N < r, \end{cases} \tag{A.1}$$

where $Z^{(i)}$ is the ith row of Z and $Z_{1:s}^{(i)}$ is the first s entries of $Z^{(i)}$, the desired statement is obtained by letting $A^* = [\mathbf{0}, \tilde{A}]$ as $\boldsymbol{\Phi}(\mu)A^* = \boldsymbol{\Phi}_0(\mu)\tilde{A} = Z_{1:\tilde{r}}\Sigma_{1:\tilde{r}}V_{1:\tilde{r}}^\top$ where $\tilde{r} = \min\{N, r\}$ and $\boldsymbol{\Phi}_0(\mu)$ is the matrix whose ith row is $\boldsymbol{\phi}_0^\top(y_i;\mu)$.

For the rest of the proof, we explicitly construct a deep ReLU network satisfying (A.1). We closely follow the construction that appeared in [40]. Note that for any distinct $y_1, \dots, y_{m_{\mathrm{y}}}$ in $\mathbb{R}^{d_y}$, there exists a unit vector $v \in \mathbb{R}^{d_y}$ (see, e.g., [36]) such that

$$\sqrt{\frac{8}{\pi d}}\frac{1}{m_{\mathrm{y}}^2}\|y_i - y_j\| \leq |v^\top(y_i - y_j)| \leq \|y_i - y_j\| \quad \forall i \neq j.$$

Let $\mathrm{y}_i = \tilde{v}^\top y_i$ where $\tilde{v} = 2\delta^{-1}\sqrt{\frac{\pi d}{8}}m_{\mathrm{y}}^2 v$, and let $\mathrm{y}_1 < \cdots < \mathrm{y}_{m_{\mathrm{y}}}$ (after reordering if necessary). It then can be checked that $|\mathrm{y}_i - \mathrm{y}_j| \geq 2$ for all $i \neq j$.

For $a < b$, let $N_{a,b}$ be a three-layer ReLU network of width 2 defined by

$$N_{a,b}(\mathrm{y}) = A^3\sigma(A^2\sigma(A^1\mathrm{y} + b^1) + b^2) + b^3,$$

where $A^1 = \left(\begin{smallmatrix} -2 \\ 2 \end{smallmatrix}\right)$, $b^1 = \left(\begin{smallmatrix} 2a \\ -2b \end{smallmatrix}\right)$, $A^2 = \left(\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, $b^2 = \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right)$, $A^3 = \left(\begin{smallmatrix} 1 & 1 \end{smallmatrix}\right)$, $b^3 = -1$, which emulates the hat-like function, i.e.,

$$N_{a,b}(\mathrm{y}) = \begin{cases} 1 & \text{if } a \leq \mathrm{y} \leq b, \\ 0 & \text{if } \mathrm{y} \leq a - \frac{1}{2} \quad \text{or} \quad b + \frac{1}{2} \leq \mathrm{y}, \\ 2(\mathrm{y} - (a - \frac{1}{2})) & \text{if } a - \frac{1}{2} < \mathrm{y} < a, \\ -2(\mathrm{y} - b) & \text{if } b < \mathrm{y} < b + \frac{1}{2}. \end{cases}$$

For $k = 1, \dots, \tilde{r}$, let $N^{(k)}(\mathrm{y}) = \sum_{j=1}^{m_{\mathrm{y}}} Z_k^{(j)} N_{\mathrm{y}_j, \mathrm{y}_{j+1}}(\mathrm{y})$ which satisfies $N^{(k)}(\mathrm{y}_j) = Z_k^{(j)}$ for all $j \in [m_{\mathrm{y}}]$. The remaining task is to construct a deep ReLU network $\boldsymbol{\phi}_0$ such that $\boldsymbol{\phi}_0(y;\mu) = (N^{(1)}(\tilde{v}^\top y), \dots, N^{(\tilde{r})}(\tilde{v}^\top y), \mathbf{0}_{N-\tilde{r}})^\top$.

Let $\boldsymbol{z} = (z_1, \dots, z_{\tilde{r}})^\top$, $\boldsymbol{c} = (c_1, \dots, c_{\tilde{r}})^\top$, and consider a three-layer ReLU network $F_{a,b,\boldsymbol{c}}$ of width $\tilde{n} = 2\tilde{r} + 4$ defined by

$$F_{a,b,\boldsymbol{c}}\left(\begin{bmatrix} \mathrm{y} \\ \boldsymbol{z} \end{bmatrix}\right) = \tilde{A}^3\sigma\left(\tilde{A}^2\sigma\left(\tilde{A}^1\begin{bmatrix} \mathrm{y} \\ \boldsymbol{z} \end{bmatrix} + \tilde{b}^1\right) + \tilde{b}^2\right) + \tilde{b}^3 = \begin{pmatrix} \mathrm{y} \\ \boldsymbol{z} + \boldsymbol{c}N_{a,b}(\mathrm{y}) \end{pmatrix},$$

where $P = (1, -1)^\top$,

$$\tilde{A}^1 = \begin{pmatrix} A^1 & \mathbf{0}_{2\times\tilde{r}} \\ P & \mathbf{0}_{2\times\tilde{r}} \\ \mathbf{0}_{2\tilde{r}\times 1} & \mathbf{P}_{\tilde{r}} \end{pmatrix} \in \mathbb{R}^{\tilde{n}\times(\tilde{r}+1)} \text{ with } \mathbf{P}_s = \begin{pmatrix} P & \mathbf{0}_{2\times 1} & \cdots & \mathbf{0}_{2\times 1} \\ \mathbf{0}_{2\times 1} & P & \cdots & \mathbf{0}_{2\times 1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{2\times 1} & \mathbf{0}_{2\times 1} & \cdots & P \end{pmatrix} \in \mathbb{R}^{2s\times s},$$

$$\tilde{b}^1 = \begin{pmatrix} b^1 \\ \mathbf{0}_{2(\tilde{r}+1)\times 1} \end{pmatrix},$$

$$\tilde{A}^2 = \begin{pmatrix} A^2 & \mathbf{0}_{2\times 2(\tilde{r}+1)} \\ \mathbf{0}_{2(\tilde{r}+1)\times 2} & I_{2(\tilde{r}+1)} \end{pmatrix} \in \mathbb{R}^{\tilde{n}\times\tilde{n}},$$

$$\tilde{b}^2 = \begin{pmatrix} b^2 \\ \mathbf{0}_{2(\tilde{r}+1)\times 1} \end{pmatrix} \in \mathbb{R}^{\tilde{n}},$$

$$\tilde{A}^3 = \begin{pmatrix} \mathbf{0}_{1\times 2} & P^\top & \mathbf{0}_{1\times 2\tilde{r}} \\ \boldsymbol{c}A^3 & \mathbf{0}_{\tilde{r}\times 2} & \mathbf{P}_{\tilde{r}}^\top \end{pmatrix} \in \mathbb{R}^{(\tilde{r}+1)\times\tilde{n}},$$

$$\tilde{b}^3 = \begin{pmatrix} 0 \\ \boldsymbol{c} \end{pmatrix} \in \mathbb{R}^{\tilde{r}+1}.$$

Last, let us consider a three-layer ReLU network $F^{(0)}_{a,b,\boldsymbol{c}}$ of width 4 defined by

$$F^{(0)}_{a,b,\boldsymbol{c}}(y) = \hat{A}^3\sigma(\hat{A}^2\sigma(\hat{A}^1 y + \hat{b}^1) + \hat{b}^2) + \hat{b}^3 = \begin{pmatrix} x \\ \boldsymbol{c}N_{a,b}(\tilde{v}^\top y) \end{pmatrix},$$

where $\hat{A}^1 = \begin{pmatrix} A^1 \\ P \end{pmatrix}\tilde{v}^\top \in \mathbb{R}^{4\times d_y}$, $\hat{b}^1 = \begin{pmatrix} b^1 \\ \mathbf{0}_{2\times 1} \end{pmatrix}$, $\hat{A}^2 = \begin{pmatrix} A^2 & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & I_2 \end{pmatrix}$, $\hat{b}^2 = \begin{pmatrix} b^2 \\ \mathbf{0}_{2\times 1} \end{pmatrix}$, $\hat{A}^3 = \begin{pmatrix} \mathbf{0}_{1\times 2} & P^\top \\ \boldsymbol{c}A^3 & \mathbf{0}_{\tilde{r}\times 2} \end{pmatrix}$, $\hat{b}^3 = \begin{pmatrix} 0 \\ \boldsymbol{c} \end{pmatrix}$.

Let us consider

$$\boldsymbol{\phi}_0(y; \mu^*) := AF_{\mathrm{y}_{m_\mathrm{y}-1}, \mathrm{y}_{m_\mathrm{y}}, Z^{(m_\mathrm{y})}_{1:\tilde{r}}} \circ \cdots \circ F_{\mathrm{y}_2, \mathrm{y}_3, Z^{(2)}_{1:\tilde{r}}} \circ F^{(0)}_{\mathrm{y}_1, \mathrm{y}_2, Z^{(1)}_{1:\tilde{r}}}(y),$$

where $A = \begin{bmatrix} \mathbf{0}_{\tilde{r}\times 1} & I_{\tilde{r}} \\ \mathbf{0}_{(N-\tilde{r})\times 1} & \mathbf{0}_{(N-\tilde{r})\times(N-\tilde{r})} \end{bmatrix}$. Then, it is a $(2m_\mathrm{y} + 1)$-layer ReLU network whose architecture is $\vec{\boldsymbol{n}}_t$ as shown in the statement, and μ^* represents the corresponding network parameters. It then can be checked that $\boldsymbol{\phi}_0(y; \mu^*) = (N^{(1)}(\tilde{v}^\top y), \dots, N^{(\tilde{r})}(\tilde{v}^\top y), \mathbf{0}_{N-\tilde{r}})^\top$, which completes the proof. ∎

Appendix B. Proof of Theorem 4.10.

Proof. Let $V_n(\mu) = \text{span}\{(\boldsymbol{\phi}(\cdot;\mu))_j : j = 1, \dots, N+1\}$. The minimization problem of (3.4) is equivalent to

$$\mu^* = \underset{\mu \in \mathrm{M}}{\text{argmin}} \, \|(\boldsymbol{\Phi}(\mu)\boldsymbol{\Phi}(\mu)^\dagger - I)\boldsymbol{U}\|_{2,2}^2,$$

where M is a feasible set for the trunk networks defined on Assumption 4.7. Since $\mu^* \in \mathrm{M}$, there exists T_{μ^*} such that $\hat{\boldsymbol{\phi}}(\cdot;\mu^*, T_{\mu^*})$ form an orthonormal basis in $L^2_\omega(\Omega_y)$, which we denote by $\{\psi_j^*\}_{j=0}^N$. We then rewrite the operator $\mathcal{G}$ of interest in terms of the new orthonormal basis, i.e.,

$$\mathcal{G}[f](y) = \sum_{j=0}^{\infty} \mathfrak{c}_j^*(f)\psi_j^*(y).$$

Observe that

$$\begin{aligned}
\|\mathcal{G}[f] - \widetilde{O}_{\text{net}}[f]\|_{L^2_\omega(\Omega_y)}^2 &= \|\mathcal{G}[f] - \mathcal{G}_N[f] + \mathcal{G}_N[f] - \widetilde{O}_{\text{net}}[f]\|_{L^2_\omega(\Omega_y)}^2 \\
&= \|\mathcal{G}[f] - \mathcal{G}_N[f]\|_{L^2_\omega(\Omega_y)}^2 + \|\mathcal{G}_N[f] - \widetilde{O}_{\text{net}}[f]\|_{L^2_\omega(\Omega_y)}^2 \\
&= \|\mathcal{G}[f] - \mathcal{G}_N[f]\|_{L^2_\omega(\Omega_y)}^2 + \sum_{j=0}^{N} (\mathfrak{c}_j^*(f) - \boldsymbol{c}_j(\boldsymbol{f};\theta^*))^2.
\end{aligned}$$

The second term on the right-hand side of the above can be further bounded as follows:

$$\begin{aligned}
&(\mathfrak{c}_j^*(f) - \boldsymbol{c}_j(\boldsymbol{f};\theta^*))^2 \\
&= (\mathfrak{c}_j^*(f) - \mathfrak{c}_j^*(f_k) + \mathfrak{c}_j^*(f_k) - \boldsymbol{c}_j(\boldsymbol{f};\theta^*) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*) + \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2 \\
&\le 3\left\{(\mathfrak{c}_j^*(f) - \mathfrak{c}_j^*(f_k))^2 + (\mathfrak{c}_j^*(f_k) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2 + (\boldsymbol{c}_j(\boldsymbol{f};\theta^*) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2\right\} \\
&\le 3\left\{L_j^2 d_{\mathcal{X}}^2(f, f_k) + (\mathfrak{c}_j^*(f_k) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2 + L_\sigma^2 L_c^2(K, N, m_{\text{x}})\|\boldsymbol{f} - \boldsymbol{f}_k\|_{w,2}^2\right\}.
\end{aligned}$$

Thus, we have

$$\begin{aligned}
\|\mathcal{G}[f] - \widetilde{O}_{\text{net}}[f]\|_{L^2_\omega(\Omega_y)}^2 \le{}& \|\mathcal{G}[f] - \mathcal{G}_N[f]\|_{L^2_\omega(\Omega_y)}^2 \\
&+ 3\left\{L_{\mathcal{G}}^2 d_{\mathcal{X}}^2(f, f_k) + L_\sigma^2 L_c^2 \|\boldsymbol{f} - \boldsymbol{f}_k\|_{w,2}^2\right\} \\
&+ 3\sum_{j=0}^{N} (\mathfrak{c}_j^*(f_k) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2.
\end{aligned}$$

Recall that since the trunk networks are an orthogonal basis, the optimal solution A^* of (3.4) is the least-squares solution as $p = 2$. Under Assumptions 4.2 and 4.7, it follows from Theorem 2 of [11, 12] that we have

$$\mathbb{E}\left[\sum_{j=0}^{N} (\mathfrak{c}_j^*(f_k) - \boldsymbol{c}_j(\boldsymbol{f}_k;\theta^*))^2\right] \lesssim C'(m_{\text{y}}, r_t)\|\mathcal{G}[f_k] - \mathcal{G}_N[f_k]\|_{L^2_\omega(\Omega_y)}^2 + m_{\text{y}}^{-r_t},$$

where $C'(m_y, r_t) = \frac{6\log(3/2) - 2}{(1+r_t)\log m_{\text{y}}}$. By combining the above with Assumptions 4.2 and 4.3, we have

$$\mathbb{E}\left[\|\mathcal{G}[f] - \widetilde{O}_{\text{net}}[f]\|_{L^2_\omega}^2\right] \lesssim C(m_{\text{y}}, r_t) N^{-r_{\mathcal{G},\mathcal{X},\mu^*}} + m_{\text{y}}^{-r_t} + K^{-\alpha} + m_{\text{x}}^{-s},$$

which completes the proof. ∎

REFERENCES

[1] M. Ainsworth and Y. Shin, *Plateau phenomenon in gradient descent training of RELU networks: Explanation, quantification, and avoidance*, SIAM J. Sci. Comput., 43 (2021), pp. A3438–A3468.

[2] M. Ainsworth and Y. Shin, *Active neuron least squares: A training method for multivariate rectified neural networks*, SIAM J. Sci. Comput., 44 (2022), pp. A2253–A2275.

[3] Z. Allen-Zhu, Y. Li, and Z. Song, *A convergence theory for deep learning via over-parameterization*, in Proceedings of the 36th International Conference on Machine Learning, Vol. 97, PMLR, 2019, pp. 242–252.

[4] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, *The FEniCS project version* 1.5, Arch. Num. Soft., 3 (2015).

[5] A. Anandkumar, K. Azizzadenesheli, K. Bhattacharya, N. Kovachki, Z. Li, B. Liu, and A. Stuart, *Neural operator: Graph kernel network for partial differential equations*, in Proceedings of the ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations, 2020, https://openreview.net/forum?id=fg2ZFmXFO3.

[6] D. Arndt, W. Bangerth, B. Blais, T. C. Clevenger, M. Fehling, A. V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, et al., *The deal.*II *library, version* 9.2, J. Numer. Math., 28 (2020), pp. 131–146.

[7] J. A. L. Benitez, T. Furuya, F. Faucher, X. Tricoche, and M. V. de Hoop, *Out-of-Distributional Risk Bounds for Neural Operators with Applications to the Helmholtz Equation*, preprint, arXiv:2301.11509, 2023.

[8] K. Bi, L. Xie, H. Zhang, X. Chen, X. Gu, and Q. Tian, *Accurate medium-range global weather forecasting with* 3*D neural networks*, Nature, 619 (2023), pp. 533–538.

[9] K. A. Boster, S. Cai, A. Ladrón-de Guevara, J. Sun, X. Zheng, T. Du, J. H. Thomas, M. Nedergaard, G. E. Karniadakis, and D. H. Kelley, *Artificial intelligence velocimetry reveals in vivo flow rates, pressure gradients, and shear stresses in murine perivascular flows*, Proc. Natl. Acad. Sci. USA, 120 (2023), e2217744120.

[10] T. Chen and 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 (1995), pp. 911–917.

[11] A. Cohen, M. A. Davenport, and D. Leviatan, *On the stability and accuracy of least squares approximations*, Found. Comput. Math., 13 (2013), pp. 819–834.

[12] A. Cohen, M. A. Davenport, and D. Leviatan, *Correction to: On the stability and accuracy of least squares approximations*, Found. Comput. Math., 19 (2019), pp. 239–239.

[13] B. Deng, Y. Shin, L. Lu, Z. Zhang, and G. E. Karniadakis, *Approximation rates of DeepONets for learning operators arising from advection–diffusion equations*, Neural Netw., 153 (2022), pp. 411–426.

[14] S. Du, J. Lee, H. Li, L. Wang, and X. Zhai, *Gradient descent finds global minima of deep neural networks*, in Proceedings of the 36th International Conference on Machine Learning, PMLR, 2019, pp. 1675–1685.

[15] S. S. Du, X. Zhai, B. Poczos, and A. Singh, *Gradient descent provably optimizes over-parameterized neural networks*, in Proceedings of the International Conference on Learning Representations, 2019, https://openreview.net/forum?id=S1eK3i09YQ.

[16] L. C. Evans, *Partial Differential Equations*, Grad. Stud. Math. 19, AMS, Providence, RI, 2022.

[17] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, Vol. 9, PMLR, 2010, pp. 249–256.

[18] S. Goswami, A. Bora, Y. Yu, and G. E. Karniadakis, *Physics-informed deep neural operator networks*, in Machine Learning in Modeling and Simulation: Methods and Applications, T. Rabczuk and K.-J. Bathe, eds., Springer, 2023, pp. 219–254, https://doi.org/10.1007/978-3-031-36644-4_6.

[19] S. Goswami, K. Kontolati, M. D. Shields, and G. E. Karniadakis, *Deep transfer operator learning for partial differential equations under conditional shift*, Nat. Mach. Intell., 4 (2022), pp. 1155–1164.

[20] J. Guibas, M. Mardani, Z. Li, A. Tao, A. Anandkumar, and B. Catanzaro, *Efficient token mixing for transformers via adaptive fourier neural operators*, in Proceedings of the International Conference on Learning Representations, 2022, https://openreview.net/forum?id=EXHG-A3jlM.

[21] K. He, X. Zhang, S. Ren, and 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.

[22] J. S. Hesthaven, S. Gottlieb, and D. Gottlieb, *Spectral Methods for Time-Dependent Problems*, Cambridge Monogr. Appl. Comput. Math. 21, Cambridge University Press, Cambridge, UK, 2007.
[23] P. Jin, S. Meng, and L. Lu, *MIONet: Learning multiple-input operators via tensor product*, SIAM J. Sci. Comput., 44 (2022), pp. A3490–A3514.
[24] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, in Proceedings of the International Conference on Learning Representations, 2015.
[25] G. Kissas, J. H. Seidman, L. F. Guilhoto, V. M. Preciado, G. J. Pappas, and P. Perdikaris, *Learning operators with coupled attention*, J. Mach. Learn. Res., 23 (2022), pp. 1–63.
[26] T. Kurth, S. Subramanian, P. Harrington, J. Pathak, M. Mardani, D. Hall, A. Miele, K. Kashinath, and A. Anandkumar, *FourCastNet: Accelerating global high-resolution weather forecasting using adaptive Fourier neural operators*, in Proceedings of the Platform for Advanced Scientific Computing Conference, Vol. 13, ACM, 2023, pp. 1–11, https://doi.org/10.1145/3592979.3593412.
[27] S. Lanthaler, S. Mishra, and G. E. Karniadakis, *Error estimates for DeepONets: A deep learning framework in infinite dimensions*, Trans. Math. Appl., 6 (2022), tnac001.
[28] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier neural operator for parametric partial differential equations*, in Proceedings of the International Conference on Learning Representations, 2021, https://openreview.net/forum?id=c8P9NQVtmnO.
[29] C. Lin, Z. Li, L. Lu, S. Cai, M. Maxey, and G. E. Karniadakis, *Operator learning for predicting multiscale bubble growth dynamics*, J. Chem. Phys., 154 (2021).
[30] L. Liu and W. Cai, *Multiscale Deeponet for Nonlinear Operators in Oscillatory Function Spaces for Building Seismic Wave Responses*, preprint, arXiv:2111.04860, 2021.
[31] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*, Nat. Mach. Intell., 3 (2021), pp. 218–229.
[32] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, and G. E. Karniadakis, *A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data*, Comput. Methods Appl. Mech. Engrg., 393 (2022), 114778.
[33] C. Marcati and C. Schwab, *Exponential convergence of deep operator networks for elliptic partial differential equations*, SIAM J. Numer. Anal., 61 (2023), pp. 1513–1545.
[34] V. Oommen, K. Shukla, S. Goswami, R. Dingreville, and G. E. Karniadakis, *Learning two-phase microstructure evolution using neural operators and autoencoder architectures*, npj Comput. Mater., 8 (2022), 190.
[35] S. Oymak and M. Soltanolkotabi, *Toward moderate overparameterization: Global convergence guarantees for training shallow neural networks*, IEEE J. Sel. Areas Inf. Theory, 1 (2020), pp. 84–105.
[36] S. Park, J. Lee, C. Yun, and J. Shin, *Provable memorization via deep neural networks using sub-linear parameters*, in Proceedings of the Conference on Learning Theory, Vol. 134, PMLR, 2021, pp. 3627–3661, https://proceedings.mlr.press/v134/park21a.html.
[37] S. Ruder, *An Overview of Gradient Descent Optimization Algorithms*, preprint, arXiv:1609.04747, 2016.
[38] Y. Shin, J. Darbon, and G. E. Karniadakis, *On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs*, Commun. Comput. Phys., 28 (2020), pp. 2042–2074.
[39] A. Tran, A. Mathews, L. Xie, and C. S. Ong, *Factorized fourier neural operators*, in Proceedings of the International Conference on Learning Representations, 2023, https://openreview.net/forum?id=tmIiMPl4IPa.
[40] G. Vardi, G. Yehudai, and O. Shamir, *On the optimal memorization power of ReLU neural networks*, in Proceedings of the International Conference on Learning Representations, 2022, https://openreview.net/forum?id=MkTPtnjeYTV.
[41] S. Venturi and T. Casey, *SVD perspectives for augmenting DeepONet flexibility and interpretability*, Comput. Methods Appl. Mech. Engrg., 403 (2023), 115718.
[42] S. Wang, H. Wang, and P. Perdikaris, *Improved architectures and training algorithms for deep operator networks*, J. Sci. Comput., 92 (2022), 35.
[43] D. Zou, Y. Cao, D. Zhou, and Q. Gu, *Gradient descent optimizes over-parameterized deep ReLU networks*, Mach. Learn., 109 (2020), pp. 467–492.