Computer Methods in Applied Mechanics and Engineering 436 (2025) 117699

Contents lists available at ScienceDirect

Comput. Methods Appl. Mech. Engrg.

journal homepage: www.elsevier.com/locate/cma

DeepOKAN: Deep operator network based on Kolmogorov Arnold networks for mechanics problems

Diab W. Abueidda [a,b], Panos Pantidis [a], Mostafa E. Mobasher [a,*]

[a] Civil and Urban Engineering Department New York University, Abu Dhabi, United Arab Emirates
[b] National Center for Supercomputing Applications University of Illinois at Urbana-Champaign, United States of America

ARTICLE INFO

Dataset link: https://github.com/DiabAbu/DeepOKAN

Keywords:
Computational solid mechanics
Deep operator networks
Gaussian radial basis functions
Neural networks
Orthotropic elasticity
Transient analysis

ABSTRACT

The modern digital engineering design often requires costly repeated simulations for different scenarios. The prediction capability of neural networks (NNs) makes them suitable surrogates for providing design insights. However, only a few NNs can efficiently handle complex engineering scenario predictions. We introduce a new version of the neural operators called DeepOKAN, which utilizes Kolmogorov Arnold networks (KANs) rather than the conventional neural network architectures. Our DeepOKAN uses Gaussian radial basis functions (RBFs) rather than the B-splines. RBFs offer good approximation properties and are typically computationally fast. The KAN architecture, combined with RBFs, allows DeepOKANs to represent better intricate relationships between input parameters and output fields, resulting in more accurate predictions across various mechanics problems. Specifically, we evaluate DeepOKAN's performance on several mechanics problems, including 1D sinusoidal waves, 2D orthotropic elasticity, and transient Poisson's problem, consistently achieving lower training losses and more accurate predictions compared to traditional DeepONets. This approach should pave the way for further improving the performance of neural operators.

1. Introduction

Contemporary science and engineering increasingly depend on advanced physics-based computational models. Numerical simulations offer critical insights for understanding and predicting complex physical phenomena and engineering systems. Real-world problems often involve varying loads, boundary and initial conditions, material properties, and domain geometries. Due to their inherent complexity, multi-physics nature, dimensionality, time dependency, and fidelity requirements, finite element analysis (FEA) models can be computationally intensive, even with high-performance computing platforms [1–3]. Consequently, relying solely on traditional high-fidelity simulation models for tasks such as computer-aided design, material discovery, and digital twins is often impractical, especially when exploring numerous design scenarios or geometries [4]. In this context, surrogate neural network (NN) models emerge as a promising machine learning approach, capable of rapidly inferring solutions to physical problems without requiring expensive numerical simulations once they are trained [5–7]. These models hold significant potential across various application domains, including real-time simulations for predictions and controls, design, topology and shape optimizations, sensitivity analysis, and uncertainty quantification, which require extensive forward evaluations with varying parameters [8–10]. For a comprehensive overview of the application of neural networks in computational mechanics, see [11].

Despite their advantages, most existing surrogate models still necessitate retraining or transfer learning when input parameters such as loads, boundary conditions, material properties, or geometry are altered. This limitation highlights the need for more

* Corresponding author.
E-mail addresses: da3205@nyu.edu (D.W. Abueidda), mostafa.mobasher@nyu.edu (M.E. Mobasher).

https://doi.org/10.1016/j.cma.2024.117699
Received 6 September 2024; Received in revised form 20 December 2024; Accepted 20 December 2024
Available online 14 January 2025

robust solutions that adapt to changing conditions without extensive reconfiguration. Recently, researchers devised NNs trained to approximate the underlying physics or mathematical operator for a class of problems, a process known as operator learning [12]. Researchers have proposed various architectures for operator learning, with two notable examples being the Fourier neural operator (FNO) and the deep operator network (DeepONet). The Fourier Neural Operator (FNO) was first introduced by Li et al. [13] to solve partial differential equations with parametric inputs. Drawing inspiration from the Fourier transform used in differential equation solutions, the input function is processed through multiple Fourier layers. The encoded information is then mapped onto the output function space. Each Fourier layer applies the fast Fourier transform (FFT) to its input and filters out high-frequency modes. FNO and its enhanced versions have been successfully used for Burger's equation, Darcy flow, and the Navier–Stokes equation [13], as well as for elasticity and plasticity problems in solid mechanics [14]. Nonetheless, since the FNO employs FFT, it encounters challenges when dealing with complex geometries or intricate non-periodic boundary conditions. Additionally, Fourier-based operations are computationally expensive when addressing high-dimensional problems.

Lately, Lu et al. [15,16] introduced the DeepONet, which emerged as another capable architecture for operator learning. DeepONet comprises two sub-networks: a branch network for encoding input functions and a trunk network for encoding input domain geometry. Initially, both networks were designed as multi-layer perceptions (MLP) networks. In the seminal study, DeepONet effectively mapped between unknown parametric functions and solution spaces for several linear and nonlinear partial differential equations (PDEs) while also learning explicit operators such as integrals. DeepONets have been increasingly used to tackle scientific and engineering challenges, such as the inverse design of nanoscale heat transfer systems [17], brittle fracture [18], digital twins [19], and uncertainty quantification [20]. Lu et al. [21] thoroughly compared FNO and DeepONet. Several researchers have proposed different architectures for the branch and/or trunk networks to enhance DeepONet's versatility and capability to capture complex scenarios [22–27].

The significance of MLPs is immense, as they are the standard models in machine learning for approximating nonlinear functions, owing to their expressive power as assured by the universal approximation theorem [28,29]. Recently, Liu et al. [30] proposed a promising alternative to MLPs called Kolmogorov-Arnold networks (KANs). While MLPs are based on the universal approximation theorem [29], KANs are based on the Kolmogorov-Arnold representation theorem [31–33]. Similar to MLPs, KANs have fully connected architectures. Nevertheless, unlike MLPs, which use fixed activation functions on nodes (neurons), KANs employ activation functions with learnable weights on edges. This difference can make KANs more accurate and interpretable than MLPs for several modeling scenarios. The potential of using the Kolmogorov-Arnold representation theorem to construct neural networks has been explored in various studies [34–38] before the work of Liu et al. [30]. However, most research has adhered to the original depth-2, width-$(2n+1)$ representation and has not utilized modern techniques such as backpropagation for training. Liu et al. [30] generalized the original Kolmogorov-Arnold representation to arbitrary widths and depths, thereby revitalizing and contextualizing it within the contemporary deep learning framework. They also conducted extensive empirical experiments to demonstrate its potential as a foundational artificial intelligence and science model, highlighting its accuracy and interoperability. Although the work of Liu et al. [30] was published recently, it has already inspired other researchers to investigate the topic further [39–43].

This study presents DeepOKAN, a model where both the branch and trunk of DeepONet are replaced with KANs instead of traditional MLPs. The motivation behind this change is to capitalize on the inherent advantages of KANs, which help address several limitations associated with MLPs. Specifically, KANs have shown promising results in overcoming issues such as lack of interpretability and catastrophic forgetting—challenges that are commonly faced by MLPs in both supervised and unsupervised learning tasks [30,44,45]. By leveraging the theoretical foundations laid by Kolmogorov and Arnold [31–33], KANs offer a compact and efficient representation of functions. This efficiency, combined with higher accuracies, positions KANs as a promising alternative to traditional MLPs across various applications.

In addition to this novel combination, we have integrated Radial Basis Functions (RBFs) for function approximation within the architecture. RBFs have emerged as a powerful tool in numerical methods, providing a flexible and effective way to approximate multivariate functions, especially in the absence of structured grid data [46,47]. RBFs are typically used to build up function approximations as a sum of weighted radial basis functions, each associated with a different center point. These functions have been employed in interpolation, approximation, and solving differential equations [47,48]. RBFs are advantageous because they provide good approximation properties, handle scattered data, do not require triangulations of the data points offering a meshless approach, and leverage known fundamental solutions (Green's functions) of the underlying PDE operators [46]. These characteristics make RBFs particularly advantageous in applications involving complex geometries or higher-dimensional problems, where traditional mesh-based methods may be computationally prohibitive. In the context of FEA, RBFs produce shape functions with simplicity and have shown promise in analyzing solid mechanics problems [49]. Their application extends to dynamic analysis, where RBF-based approaches have shown promise in capturing transient behavior [50–53]. Also, RBFs have been successfully applied in fracture mechanics, demonstrating their ability to model complex stress fields and crack propagation [54,55]. RBFs have been used before in the context of machine learning [56], but their success was limited. However, the recent development of KANs promises more effective integration.

This paper uses the DeepOKAN to solve several mechanics problems, including 1D sinusoidal waves, 2D orthotropic elasticity, and the 2D transient Poisson's problem. The paper's structure is outlined as follows: Section 2 provides an overview of KANs and Gaussian RBFs. It also scrutinizes the DeepOKAN. In Section 3, the DeepOKAN is developed as a neural operator for a few numerical examples. The paper concludes in Section 4 by summarizing the key results and indicating potential directions for future research.

2. Methods

2.1. Kolmogorov-Arnold representation theorem

The Kolmogorov-Arnold representation theorem, also known as the superposition theorem, is a fundamental result in approximation theory. It asserts that any continuous multivariate function on a bounded domain can be represented as a composition of a finite number of univariate functions and a set of linear operations (additions). Specifically, for any smooth function $f : [0,1]^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$, the Kolmogorov-Arnold theorem asserts that there exist continuous univariate functions ψ_q and $\phi_{p,q}$ such that:

$$f(\boldsymbol{x}) = \sum_{q=1}^{2n_{\text{in}}+1} \psi_q \left(\sum_{p=1}^{n_{\text{in}}} \phi_{p,q}(x_p) \right), \tag{1}$$

where $\boldsymbol{x} = (x_1, \dots, x_{d_{\text{in}}})$, $\phi_{q,p} : [0,1] \to \mathbb{R}$, and $\psi_q : \mathbb{R} \to \mathbb{R}$.

2.2. Kolmogorov-Arnold network

The Kolmogorov-Arnold representation theorem states that any continuous multivariable function on a bounded domain can be represented as a superposition of continuous functions of one variable and addition. While no strict restrictions exist on choosing these univariate functions beyond their continuity, practical considerations may influence their specific forms based on the problem context and desired properties. In the KAN implementation by Liu et al. [30], each KAN layer comprises the sum of the spline function and the sigmoid linear unit, known as the SiLU activation function, with learnable coefficients. Splines face a significant curse of dimensionality (COD) due to their inability to leverage compositional structures. On the other hand, MLPs are less affected by COD because of their feature learning capabilities. Still, they are less accurate than splines in low-dimensional settings due to their inefficiency in optimizing univariate functions. A model must capture the compositional structure (external degrees of freedom) and effectively approximate univariate functions (internal degrees of freedom) to learn a function accurately. A more detailed description of KANs can be found in the work of [30], and for the sake of completeness, we provide a brief description below.

A KAN comprises several layers, and in Fig. 1, we show a KAN with two such layers. A KAN layer with n_{in}-dimensional inputs and n_{out}-dimensional outputs can be represented as a matrix of 1D functions, denoted as:

$$\boldsymbol{\Psi} = \{\phi_{l,q,p}\}, \quad p = 1, 2, \dots, n_{\text{in}}, \quad q = 1, 2, \dots, n_{\text{out}}, \tag{2}$$

where the functions $\phi_{q,p}$ have learnable parameters. The shape of KAN can be expressed as an integer array $[n_0, n_1, \dots, n_L]$, where n_l is the number of nodes (neurons) in the lth layer (see Fig. 1). The ith neuron in the lth layer is denoted by (l, i), while the activation value of the (l, i)-neuron is represented by $x_{l,i}$. There are $n_l n_{l+1}$ activation functions between the consecutive layers l and $l+1$. The activation function connecting the neurons (l, i) and $(l+1, j)$ is denoted by:

$$\phi_{l,i,j}, \quad l = 0, \dots, L-1, \quad i = 1, \dots, n_l, \quad j = 1, \dots, n_{l+1}. \tag{3}$$

As shown in Fig. 1, the activation functions appear on the edges rather than the nodes, unlike MLPs. Specifically, the pre-activation of $\phi_{l,i,j}$ is $x_{l,i}$, while the post-activation of $\phi_{l,i,j}$ is $\tilde{x}_{l,i,j}$. Then, the $\tilde{x}_{l,i,j}$'s are summed to determine $x_{l+1,j}$, which is the activation value at the $(l+1, j)$-neuron:

$$x_{l+1,j} = \sum_{i=1}^{n_l} \tilde{x}_{l,i,j} = \sum_{i=1}^{n_l} \phi_{l,i,j}(x_{l,i}). \tag{4}$$

This can be presented in a matrix form as follows:

$$\boldsymbol{x}_{l+1} = \underbrace{\begin{pmatrix} \phi_{l,1,1}(\cdot) & \phi_{l,1,2}(\cdot) & \cdots & \phi_{l,1,n_l}(\cdot) \\ \phi_{l,2,1}(\cdot) & \phi_{l,2,2}(\cdot) & \cdots & \phi_{l,2,n_l}(\cdot) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{l,n_{l+1},1}(\cdot) & \phi_{l,n_{l+1},2}(\cdot) & \cdots & \phi_{l,n_{l+1},n_l}(\cdot) \end{pmatrix}}_{\boldsymbol{\Psi}_l} \boldsymbol{x}_l, \tag{5}$$

where $\boldsymbol{\Psi}_l$ is the function matrix corresponding to the lth KAN layer. Then, a general KAN is expressed as a composition of L layers:

$$\text{KAN}(\boldsymbol{x}) = (\boldsymbol{\Psi}_{L-1} \circ \boldsymbol{\Psi}_{L-2} \circ \cdots \circ \boldsymbol{\Psi}_1 \circ \boldsymbol{\Psi}_0)\boldsymbol{x}. \tag{6}$$

The original Kolmogorov–Arnold representation (see Eq. (1)) is deemed a special case of the KAB, with two layers and a shape $[n_0, 2n_0 + 1, 1]$. On the other hand, the well-known MLPs alternate between linear transformation $\boldsymbol{W}$ and nonlinear activation functions σ:

$$\text{MLP}(\boldsymbol{x}) = (\boldsymbol{W}_{L-1} \circ \sigma \circ \boldsymbol{W}_{L-2} \circ \sigma \circ \cdots \circ \boldsymbol{W}_1 \circ \sigma \circ \boldsymbol{W}_0)\boldsymbol{x}. \tag{7}$$

MLPs distinctly handle linear transformations $\boldsymbol{W}$ and nonlinearities σ, whereas KANs combine both within $\boldsymbol{\Psi}$.

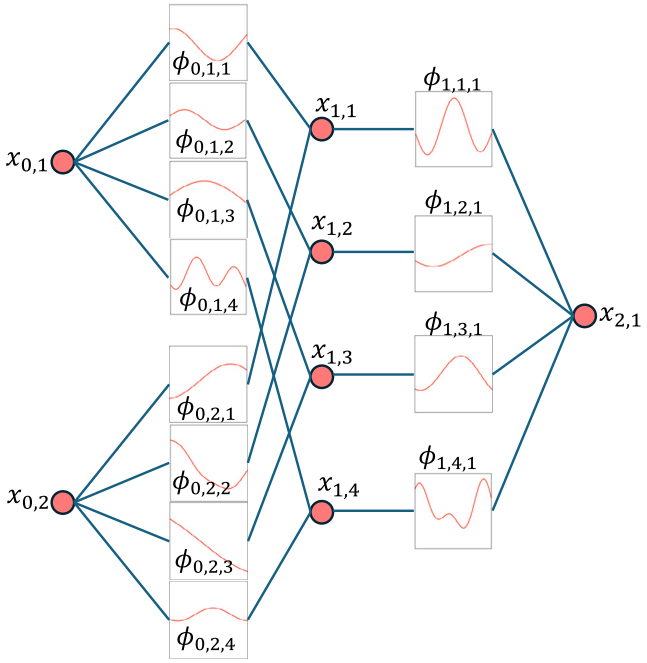

Fig. 1. Illustration of the activation functions flowing through the network.

2.3. RBF-KAN

As mentioned earlier, Liu et al. [30] proposed using B-splines as activation functions within the KAN framework. Here, we propose using RBFs for KANs. An RBF is a real-valued function whose value depends only on the distance from a central point. The Gaussian RBF is a specific type from the radial basis function family. The transformation of each input x_i using a Gaussian RBF $\boldsymbol{R}$ is given by:

$$R^l(x_i^l, g_j^l) = \exp\left(-\left(\frac{x_i^l - g_j^l}{\beta}\right)^2\right), \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, m \tag{8}$$

where n is the input dimension (layer l size), and m represents the number of grid points (RBF centers). The superscript l denotes the layer number: $l = 1, \ldots, L$, where L is the number of layers.. g_j represents the jth point in the RBF grid; $g_j \in \boldsymbol{G}$. The RBF centers $\boldsymbol{G}$ can be learnable parameters or fixed points (non-learnable) defined during the initialization of the RBF transformation within the RBF-KAN framework. β is the scaling factor for the RBF, defined as:

$$\beta = \frac{g_{\max} - g_{\min}}{m - 1}. \tag{9}$$

$g_{\max}$ and $g_{\min}$ denote the maximum and minimum values of the grid, respectively. After transforming the inputs, the transformed features are combined linearly using some learnable weight matrix $\boldsymbol{W}$. The output $\boldsymbol{x}^{l+1}$ is computed as:

$$\boldsymbol{x}^{l+1} = \boldsymbol{W}^l \boldsymbol{R}^l \left(\boldsymbol{x}^l, \boldsymbol{G}^l\right) \tag{10}$$

where $\boldsymbol{R}$ is a vector of size $(n \times m,)$, $\boldsymbol{W}$ is a matrix with a size of $(o, n \times m)$, where o is the length of the vector $\boldsymbol{x}^{l+1}$. Fig. 2 illustrates the behavior of the RBF-KAN layer, showcasing a series of Gaussian-like curves, each representing a basis function centered at different points along the input range. These basis functions highlight the localized response characteristic of the RBF layer, which can effectively capture and represent complex, non-linear patterns in the data. Similar curves can be produced using B-spline layers, as shown in the work of Liu et al. [30].

While constructing an RBF-KAN, one can stack several RBF-KAN layers. During the training of an RBF-KAN, $\boldsymbol{W}$ and $\boldsymbol{G}$ are obtained by minimizing a loss function $\mathcal{L}$, which requires finding the gradients of $\mathcal{L}$ with respect to $\boldsymbol{W}$ and $\boldsymbol{G}$ using backpropagation:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^L} \cdot \boldsymbol{W}^{L-1} \cdot \frac{\partial \boldsymbol{R}^{L-1}}{\partial \boldsymbol{x}^{L-1}} \cdot \boldsymbol{W}^{L-2} \cdot \ldots \cdot \boldsymbol{W}^{l+1} \cdot \frac{\partial \boldsymbol{R}^{l+1}}{\partial \boldsymbol{x}^{l+1}} \cdot \boldsymbol{R}^l \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{G}^l} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^L} \cdot \boldsymbol{W}^{L-1} \cdot \frac{\partial \boldsymbol{R}^{L-1}}{\partial \boldsymbol{x}^{L-1}} \cdot \boldsymbol{W}^{L-2} \cdot \ldots \cdot \boldsymbol{W}^{l+1} \cdot \frac{\partial \boldsymbol{R}^{l+1}}{\partial \boldsymbol{x}^{l+1}} \cdot \boldsymbol{W}^l \cdot \frac{\partial \boldsymbol{R}^l}{\partial \boldsymbol{G}^l}
\end{aligned} \tag{11}$$

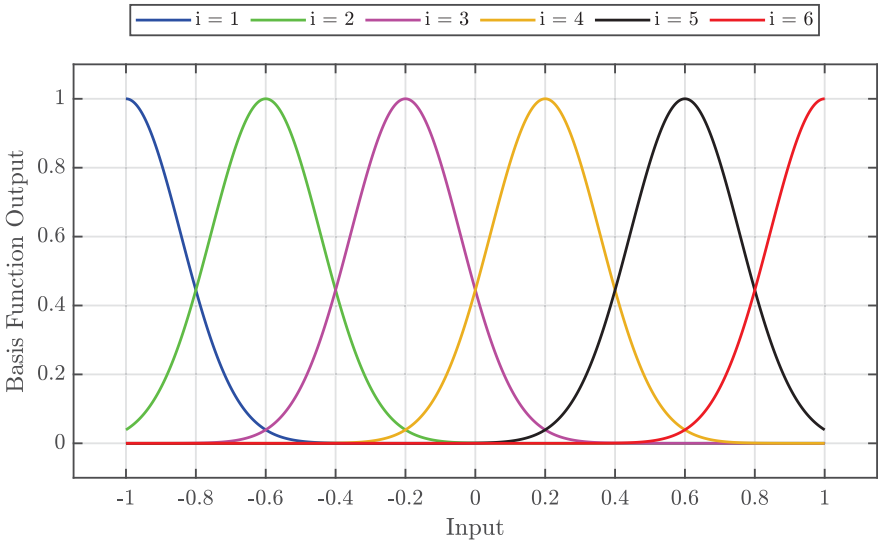

Fig. 2. Visualization of RBF-KAN Layer: Each curve represents an individual basis function.

where $\boldsymbol{x}^L$ is the final output of the RBF-KAN, and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^L}$ depends on the choice of the loss function. $\frac{\partial \boldsymbol{R}^l}{\partial \boldsymbol{x}^l}$ and $\frac{\partial \boldsymbol{R}^l}{\partial \boldsymbol{G}^l}$ are found by differentiating Eq. (8):

$$\begin{aligned}\frac{\partial R^l}{\partial x_i^l} &= -\frac{2(x_i^l - g_j^l)}{\beta^2}\exp\left(-\left(\frac{x_i^l - g_j^l}{\beta}\right)^2\right)\\ \frac{\partial R^l}{\partial g_j^l} &= \frac{2(x_i^l - g_j^l)}{\beta^2}\exp\left(-\left(\frac{x_i^l - g_j^l}{\beta}\right)^2\right).\end{aligned} \tag{12}$$

2.4. Neural operators

Researchers developed neural networks trained to approximate the underlying physics or mathematical operator for a class of problems, a process known as operator learning. Specifically, neural operators were proposed to learn mapping input functions into corresponding output functions. For neural operators, in the infinite functional space Q, the parameter $q \in Q$ represents the input functions and $s \in S$ refers to the unknown solutions of the PDE in the functional space S. It is assumed that for each q in Q, a unique solution exists $s = s(q)$ in S corresponding to the governing PDE, which also satisfies the boundary conditions (BCs). Consequently, the mapping solution operator $F : Q \to S$ can be defined as:

$$F(q) = s(q). \tag{13}$$

Specifically, for a collection of N points $\boldsymbol{X}$ on a domain, each denoted by its coordinates (x_i, y_i), the neural operator considers both the functions q_j in its branch and positions $\boldsymbol{X}$ in its trunk. The branch and trunk networks yield outputs with length r. The solution operator $\hat{F}(q)(\boldsymbol{X})$ is predicted by fusing intermediate encoded outputs b_i (from branch) and t_i (from trunk) in a dot product enhanced by bias B, as shown in Fig. 3. In a larger sense, it is possible to think of $\hat{F}(q)(\boldsymbol{X})$ as a function of $\boldsymbol{X}$ conditioning on input q.

The parameter q can encompass a variety of functional inputs that influence the underlying PDE solution. For instance, q may represent material properties such as thermal conductivity or elastic modulus. Additionally, q could denote load constants or time-dependent input load function, discretized at T time steps to form an input load vector $\boldsymbol{q}$. This allows the modeling of transient processes where the applied loads vary over time, as shown in one of the examples later. This study leverages neural operators as regressors. Thus, one needs to devise an appropriate loss function. Typical loss functions for regression problems are mean square error and mean absolute error. This paper uses the root mean square deviation (RMSD) as the loss function to be minimized and determine the optimized network parameter. The RMSD is defined as:

$$\mathcal{L} = \text{RMSD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(s_i - \hat{s}_i)^2} \tag{14}$$

where $\hat{s}$ denotes the solution obtained from the neural operator. This paper considers two types of neural operators: DeepONet and DeepOKAN.

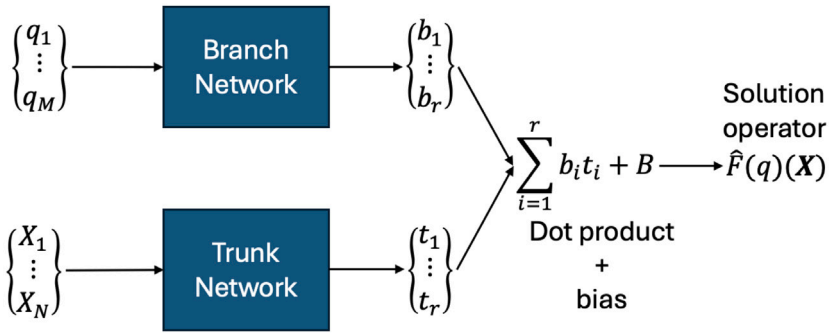

Fig. 3. Schematic for neural operators.

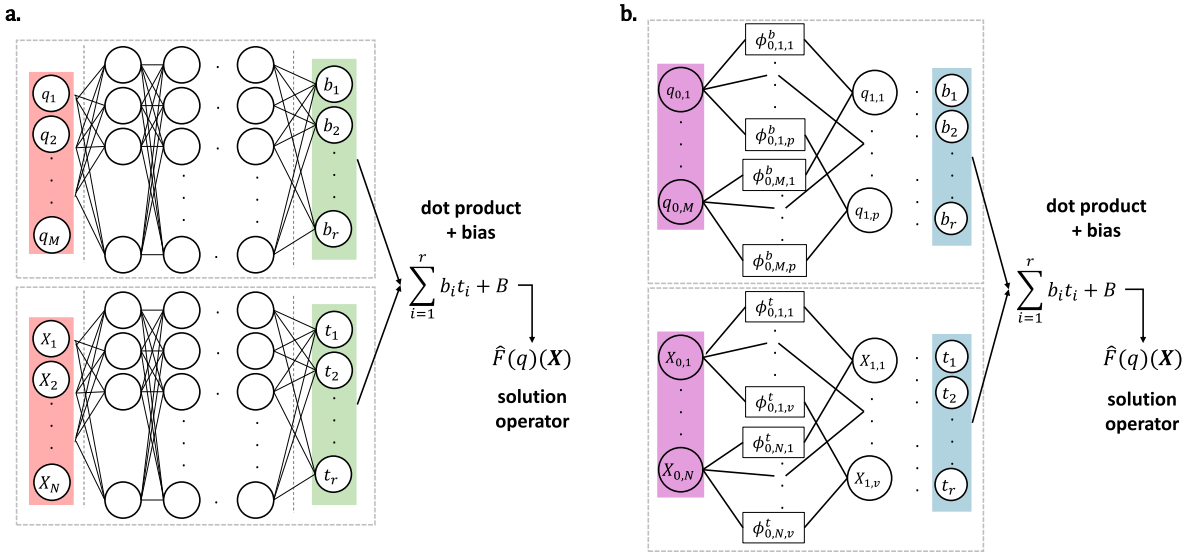

Fig. 4. Illustration of (a.) DeepONets and (b.) DeepOKANs. ϕ^l and ϕ^b represent the activation functions corresponding to the trunk and branch networks, respectively. p and v are hyperparameters defining the width of layer $l = 1$ in the branch and trunk, respectively. Different layers can possess different widths.

2.5. DeepOKAN vs. DeepONet

A DeepONet (see Fig. 4a) model, which uses MLP networks in both its branch and trunk, serves as a performance baseline. This paper introduces the DeepOKAN (see Fig. 4b), which replaces the MLP networks with KANs. Specifically, it employs Gaussian RBF KANs, though other KAN types, such as multi-quadratic RBF KANs and B-splines KANs, could also be explored in the context of neural operators.

3. Numerical examples

To create a fair evaluation platform for the two operators, DeepOKAN and DeepONet, we adopt the following approach across all the numerical examples. We constrain the depth d of the trunk and branch networks for both operators, as well as the number of neurons in their output layer r, and we then vary the number of neurons n in the hidden layers to arrive at the same number of learnable parameters w. This approach ensures that essential aspects of the network expressivity, such as the depth, neurons of the output layer, and learnable weights, remain the same across the two architectures. We also maintain identical training settings regarding the choice of batch size, optimizer, number of epochs, and starting learning rate lr. The specific implementation details of each example are provided in the following sections. In this study, we use either the 1- L-BFGS optimizer or 2- the Adam optimizer with a learning rate scheduler. The learning rate scheduler is designed to reduce the starting learning rate by a factor of γ at regular epochs T_{step}, allowing for more precise adjustments as training progresses. This approach helps prevent the model from overshooting the optimal solution. We finally note that throughout the rest of the manuscript and for notation convenience, we will also refer to the number of learnable parameters as *network complexity*, which we denote as nc. Summaries of the used hyperparameters for the different examples are available in the Appendix.

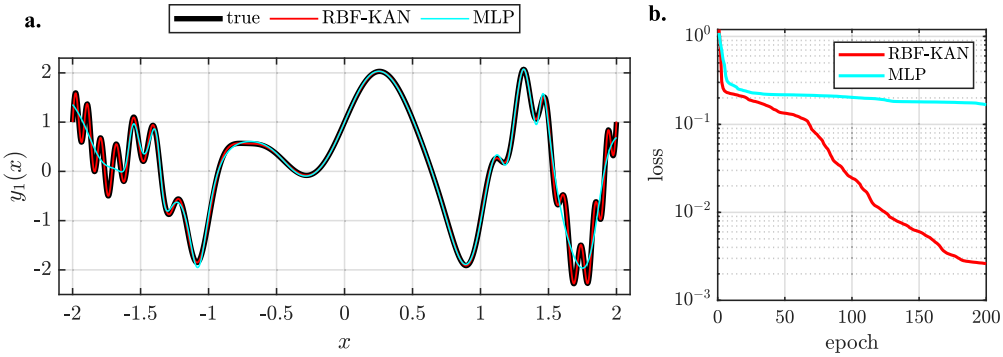

Fig. 5. Results of the 1st sinusoidal wave example (Wave-Case1): a. true and predicted values, b. evolution of the training loss.

3.1. 1D sinusoidal waves

Before we evaluate the performance of the two operators, DeepOKAN and DeepONet, we compare their baseline networks (RBF-KAN and MLP, respectively). This is an intuitive but important check since we first need to elucidate the behavior of the networks that serve as the backbone of their operator versions. For this task, we use two 1D sinusoidal waves. The equation of the first one is given below:

$$y_1(x) = \sin(2\pi x) + \cos(\pi x^2) + \cos(\pi x^3) \times \sin(\pi x^3) \tag{15}$$

Eq. (15) is sampled with 1000 uniformly distributed points between $[-2, 2]$. The RBF-KAN has 2 hidden layers with 8 neurons each, $r = 1$, and $w = 640$ trainable parameters. The MLP is constructed with 2 hidden layers and 24 neurons each, $r = 1$ and $w = 673$ learnable parameters. In this example, we train both networks with the L-BFGS optimizer for 200 epochs, using a learning rate $lr = 1$ (default) and applying the *tanh()* activation function for the MLP. The results of this study (Wave-Case1) are shown in Fig. 5, where the left graph compares the true and predicted values from the two networks. We observe that RBF-KAN approximates the true field with much higher accuracy than the MLP, particularly in the high-frequency regions. The improved expressivity of the RBF-KAN is also confirmed by the greater reduction of its loss function compared to the MLP, as shown in Fig. 5b.

Next, we create a more challenging situation where the wave experiences more intense oscillations. The second wave expression is given below:

$$y_2(x) = \cos(4\pi x) - \sin(\pi x^2) \times \cos(\pi x^3) \tag{16}$$

Eq. (16) is sampled with 1000 uniformly distributed points between $[-3, 3]$. The same architectures for RBF-KAN and MLP are utilized as in the previous case. Here, we explore two training variations. In the first case (Wave-Case2A), we use the Adam optimizer with a learning rate $lr = 10^{-2}$ for 15 000 epochs, and in the second case (Wave-Case2B), we use Adam with $lr = 10^{-3}$ for 20 000 epochs. The results of this analysis are shown in Fig. 6. For Wave-Case2A, we observe an almost excellent match between the RBF-KAN and the exact solution, even in the more demanding regions of the spectrum. On the contrary, the MLP fails to capture the high-frequency regimes on both sides of the spectrum. This figure demonstrates the ability of RBF-KAN to approximate the queried field in this problem with much better resolution than the MLP, given that they both have the same number of learnable weights and are trained for the same number of epochs. A similar picture is observed for Wave-Case2A. Here, we note that the match between RBF-KAN and the analytical solution is slightly worse than in the previous case, since there are still some mild discrepancies at the far outermost regimes of the wave. This signifies that training for further epochs could potentially improve the quality of the model predictions, which is also implied by the still descending loss function of RBF-KAN. However, despite the beneficial impact of the lower learning rate on the MLP, it is once again evident that RBF-KAN outperforms MLP in this problem as well.

Having showcased early signs of the superior performance of RBF-KAN compared to the vanilla MLP architecture, we now proceed with an in-depth comparison of their operator-based versions (DeepOKAN and DeepONet, respectively). The target function is defined as:

$$y_3(x; c) = \cos(c_1 \pi x) - \sin(c_2 \pi x^2) \cos(c_3 \pi x^3), \tag{17}$$

where x ranges from -3 to 3, and the parameters c_1, c_2, and c_3 are sampled from a uniform distribution within the interval $[-1, 1]$. For the data generation, we created 20,000 samples. The dataset was subsequently split into training and testing sets, with 80% of the data used for training and the remaining 20% for testing. A batch size of 1024 is used for both models. The DeepONet and DeepOKAN have 2 layers in the branch and trunk. The DeepOKAN has 50 neurons in each layer, whereas the DeepONet has 350 neurons with *tanh*() activation. These configurations yield 275 880 and 276 560 learnable parameters for the DeepONet and DeepOKAN, respectively. Both networks have an $r = 40$ (see Fig. 4). We consider two independent seeds for the weight initialization of both operators, henceforth termed as *seed*1 and *seed*2. The Adam optimizer with a scheduler is used to optimize the weights of

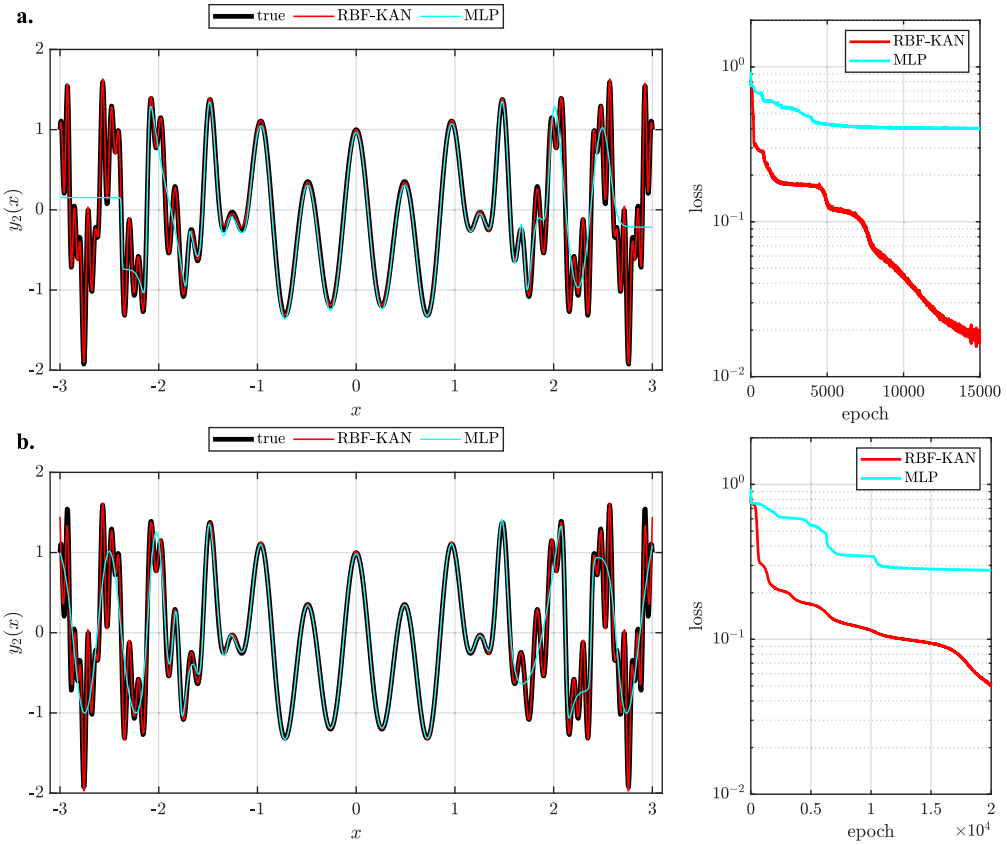

Fig. 6. Results of the 2nd sinusoidal wave example, training with: a. Adam, 15 000 epochs, $lr = 10^{-2}$ (Wave-Case2A), and b. Adam, 20 000 epochs, $lr = 10^{-3}$ (Wave-Case2B). In both (a) and (b), the left graphs show the true and predicted values, and the right graphs depict the evolution of the training loss function.

the two models. In this study, we apply the scheduler with $\gamma = 0.9$ every $T_{step} = 500$ epochs for a total training epochs of 20,000. We examine two cases of starting learning rate values, $lr = 10^{-2}$ and $lr = 10^{-3}$.

The loss convergence histories of all models are shown in Fig. 7. Overall, the DeepOKAN model demonstrates faster and more stable convergence in both learning rate settings than the DeepONet model. Specifically, for the learning rate of $lr = 10^{-2}$, the DeepONet model exhibits higher spikes and less smooth convergence, indicating potential instability at this higher learning rate. Conversely, the DeepOKAN model consistently converges to a lower training loss faster and with fewer spikes. Both models show improved stability at the lower learning rate of $lr = 10^{-3}$; however, DeepOKAN maintains superior performance. The training loss for DeepOKAN is reduced smoothly and reaches a lower value than the DeepONet models.

The L2-norm of the absolute error values for all the test samples are computed, and they are plotted as shown in the histogram of Fig. 8. We emphasize here that the same number of samples is plotted in all cases, and overall, this figure illustrates that DeepOKAN's errors are more tightly clustered around smaller values compared to the DeepONet. In particular, the DeepOKAN errors in this case are one order of magnitude smaller than DeepONet. Finally, to further visualize the difference in the performance between the two operators, we plot in Fig. 9 randomly sampled examples from the testing dataset showing the true functions alongside the predictions from DeepONet and DeepOKAN. Across all samples, DeepOKAN's predictions are consistently closer to the true values, particularly in regions with rapid oscillations and sharp peaks. This indicates its superior ability to capture intricate details and complexities for the 1D sinusoidal wave problem.

In the previous case, high frequencies appear at the boundaries. Next, we study DeepOKAN and DeepONet for a different class of sinusoidal waves with higher frequencies at the center and relatively lower frequencies at the boundaries. Consider a target function defined as:

$$y_4(x;c) = \left(\cos(4\pi c_1 x) + \sin(5\pi c_2 x)\right) \cdot e^{-x^2/2} + \cos(\pi c_3 x) \cdot (1 - e^{-x^2/2}), \tag{18}$$

where x ranges from -3 to 3, and the parameters c_1, c_2, and c_3 are sampled from a uniform distribution within the interval $[-2, 2]$. Similar to the previous case, 20,000 samples were generated. The dataset was subsequently split into training and testing datasets, with 80% of training and 20% for testing. A batch size of 1024 is used for both models. Starting from the same configurations used in the previous operator example, i.e., the DeepONet and DeepOKAN have two layers in the branch and trunk, where the DeepOKAN has 50 neurons in each layer, and the DeepONet has 350 neurons. $r = 40$ (see Fig. 4) is used for the DeepOKAN and DeepONet,

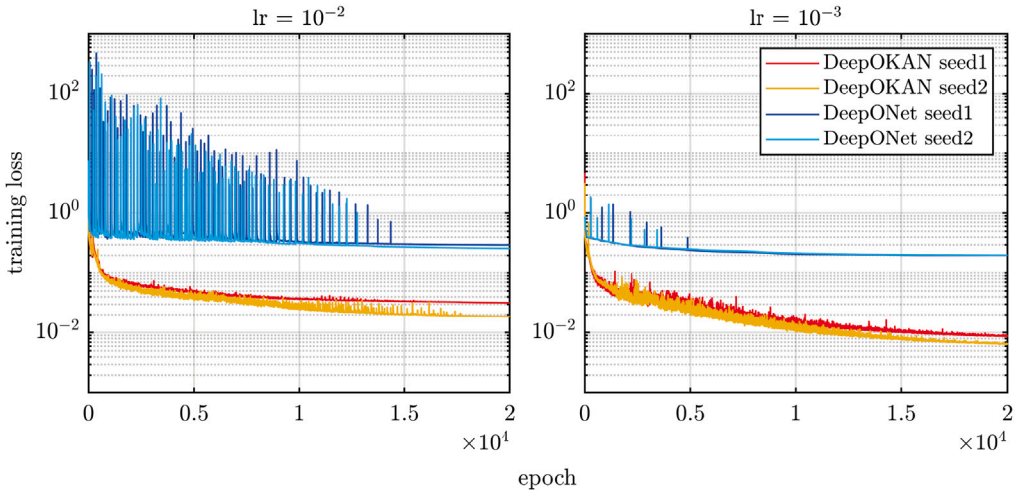

Fig. 7. DeepONet and DeepOKAN loss convergence for y_3 using different seeds and learning rates. Using a starting learning rate of a. $1e-2$ and b. $1e-3$.

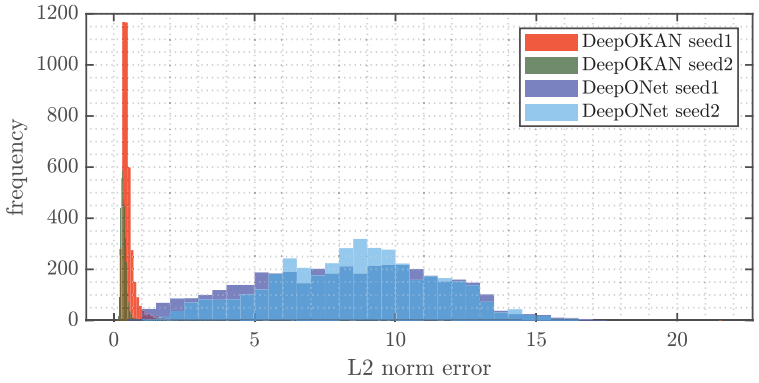

Fig. 8. Histogram of the L2-norm values of the absolute errors for y_3.

leading to the same number of learnable parameters used in the previous example. We use two seeds for the weight initialization of both operators, referred to as $seed1$ and $seed2$. The Adam optimizer, in conjunction with a learning rate scheduler, is employed to optimize the weights of the two models. In this study, the scheduler is applied with a starting learning rate of $lr = 10^{-3}$ and a decay factor of $\gamma = 0.9$ every $T_{step} = 500$ epochs for a total training duration of 20,000 epochs. In this example, we consider two activation functions for the DeepONet: $ReLU()$ and $tanh()$.

Fig. 10 shows the loss convergence histories of all models. The DeepOKAN demonstrates faster and more stable convergence than both DeepONets (with $ReLU()$ and $tanh()$ activation functions) with lower converged loss value. Such lower loss values are reflected in lower L2-norm values of the absolute errors for the DeepOKAN than the two DeepONets, as shown in Fig. 11. Fig. 12 depicts the predictions of the different models for three samples, along with the corresponding absolute error. Also, it is worth mentioning that for purely data-driven learning, i.e., without physics-informed loss terms, the DeepONet with $ReLU$ as the activation function yields more accurate results than the DeepONet with $tanh$ (for the shallow DeepONet).

Next, we consider relatively deeper branch and trunk networks for the DeepOKAN and DeepONet. Specifically, five hidden layers and $r = 40$ (see Fig. 4) are used for both operators, with 240 neurons in each layer for the DeepONet and 40 neurons in each layer for the DeepOKAN. These configurations lead to 483 440 learnable parameters for the DeepONet and 485 760 learnable parameters for the DeepOKAN. For these deep operators, we use data splits, optimizers, learning rate schedulers, and epochs similar to those of the shallow operators discussed above.

Fig. 13 illustrates the loss convergence of the DeepOKAN and two DeepONets (with different activations). For the deeper operators, we observe that the DeepONets possess faster convergence. However, the operators converge to relatively close training and testing loss values. Fig. 14 shows the L2-norm of the absolute errors for the DeepOKAN and DeepONet (with two different

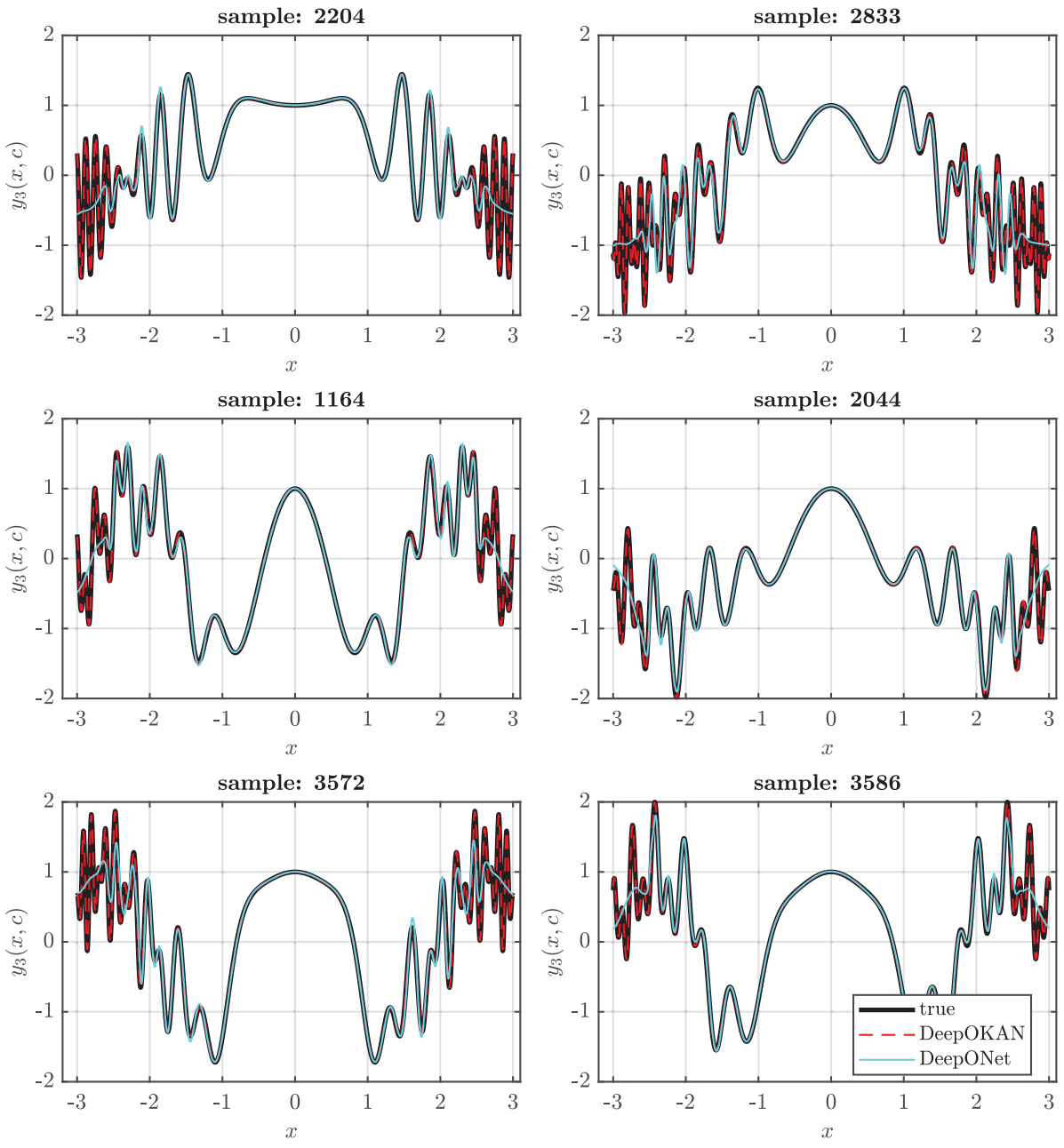

Fig. 9. y_3: Predictions of DeepONet and DeepOKAN for different samples randomly picked from the testing dataset. The c coefficients for each example are: (a.) c = [0.2276, −0.1843, −0.6270], (b.) c = [0.3257, 0.679, 0.829], (c.) c = [−0.8976, 0.7085, 0.4797], (d.) c = [−0.4736, 0.5641, −0.4129], (e.) c = [−0.7583, −0.5544, −0.7268], (f.) c = [−0.8268, −0.5951, 0.4899].

activations). Fig. 14 indicates that the operators with deeper branches and trunks perform similarly. When comparing Figs. 11 and 14, it can be observed that both DeepONets show significant improvement in prediction accuracy, while the DeepOKAN maintains the same level of accuracy. For the deep DeepONets, both *tanh* and *ReLU* activations lead to similar error levels [57]. Additionally, the shallow DeepOKAN performs similarly to the deep DeepONets, which indicates more efficient learning in favor of the DeepOKAN. Fig. 15 presents the predictions and corresponding absolute error of the DeepOKAN and DeepONet for a few samples from the testing dataset. Appendix A.2. shows a study on how the size of the training dataset affects DeepOKAN's learning process while learning y_4.

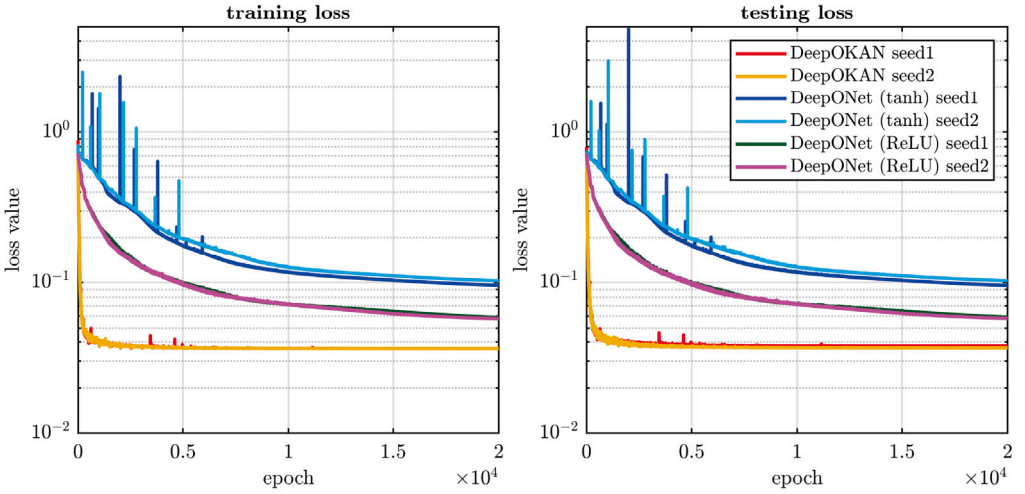

Fig. 10. Shallow DeepONet and DeepOKAN loss convergence for y_4 using different seeds.

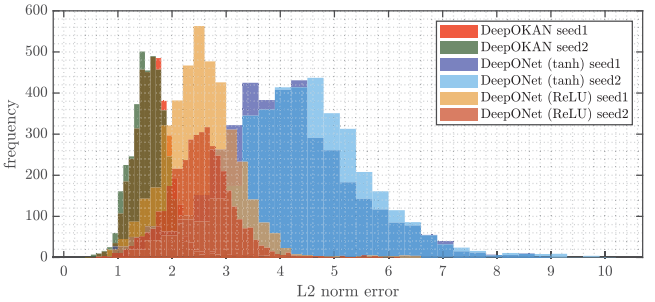

Fig. 11. Histogram of the L2-norm values of the absolute errors for shallow operators predicting y_4.

3.2. 2D orthotropic elasticity

3.2.1. Problem description, FEA, and data generation

Next, we consider a homogeneous, orthotropic, elastic body undergoing small deformations. In the absence of body forces and inertial forces, the equilibrium equation can be expressed as follows:

$$\begin{aligned} \boldsymbol{\nabla}\cdot\boldsymbol{\sigma} &= \mathbf{0}, \quad \boldsymbol{x}\in\Omega, \\ \boldsymbol{u} &= \bar{\boldsymbol{u}}, \quad \boldsymbol{x}\in\Gamma_u, \\ \boldsymbol{\sigma}\cdot\boldsymbol{n} &= \bar{\boldsymbol{t}}, \quad \boldsymbol{x}\in\Gamma_t. \end{aligned} \tag{19}$$

Here, $\boldsymbol{\sigma}$ represents the Cauchy stress tensor, $\boldsymbol{n}$ denotes the normal unit vector, $\boldsymbol{\nabla}\cdot$ signifies the divergence operator, and $\boldsymbol{\nabla}$ indicates the gradient operator. Ω denotes the domain of the problem, Γ_u denotes the part of the boundary where Dirichlet boundary conditions (DBCs) are applied, and Γ_t denotes the part of the boundary where Neumann boundary conditions (NBCs) are applied. Given the assumption of small deformations, the strain $\boldsymbol{\epsilon}$ can be described by:

$$\boldsymbol{\epsilon} = \frac{1}{2}(\boldsymbol{\nabla u} + \boldsymbol{\nabla u}^T) \tag{20}$$

The relationship between the strain and stress is written as:

$$\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\epsilon}, \tag{21}$$

where $\mathbb{C}$ is the fourth-order elastic tensor defining linear elastic behavior. In the case of plane-stress orthotropy, it is assumed that the two-dimensional material has two planes of symmetry aligned with the global x and y axes. For the orthotropic plane-stress

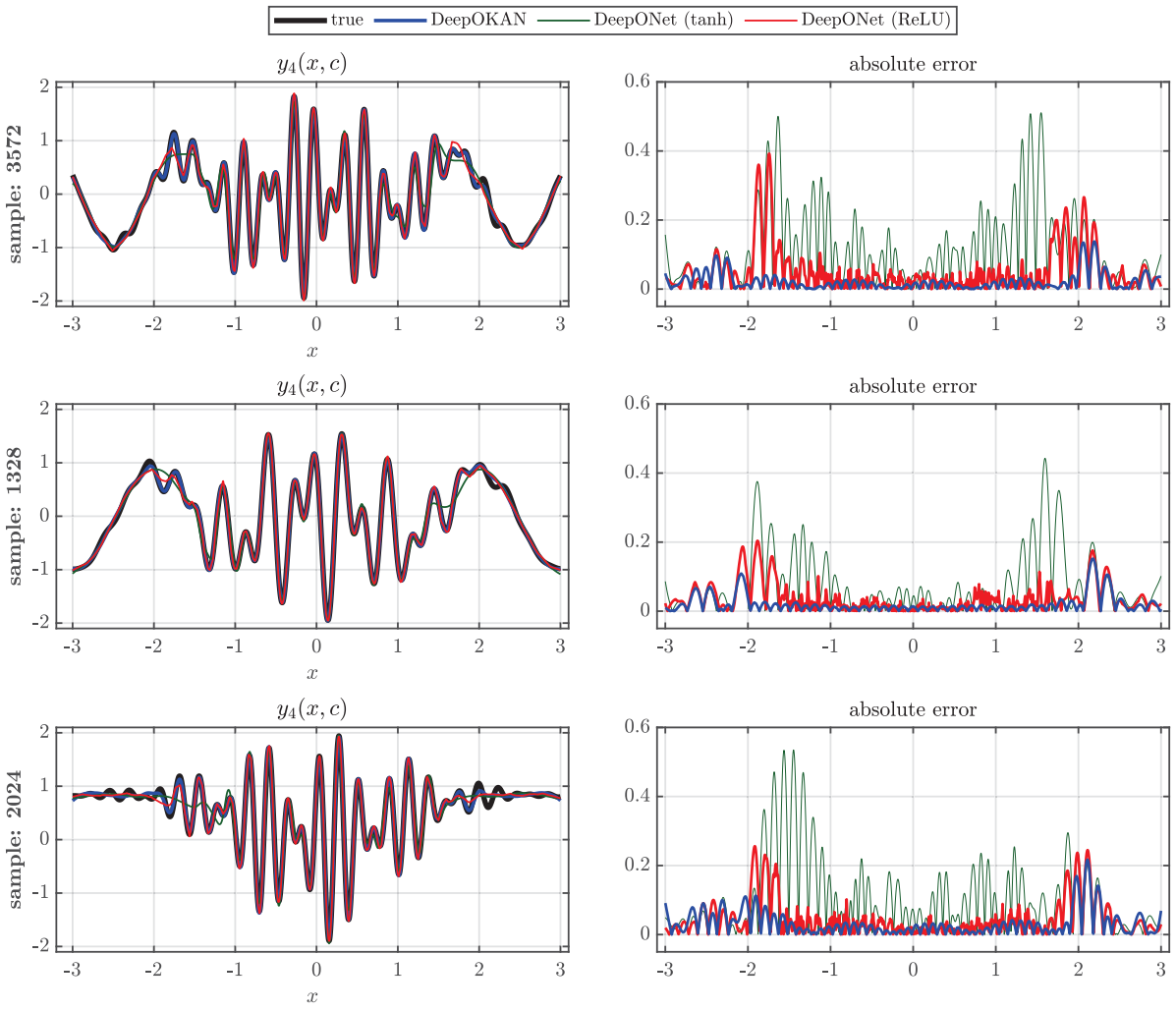

Fig. 12. y_4: Predictions of shallow DeepONet and DeepOKAN for different samples randomly picked from the testing dataset.

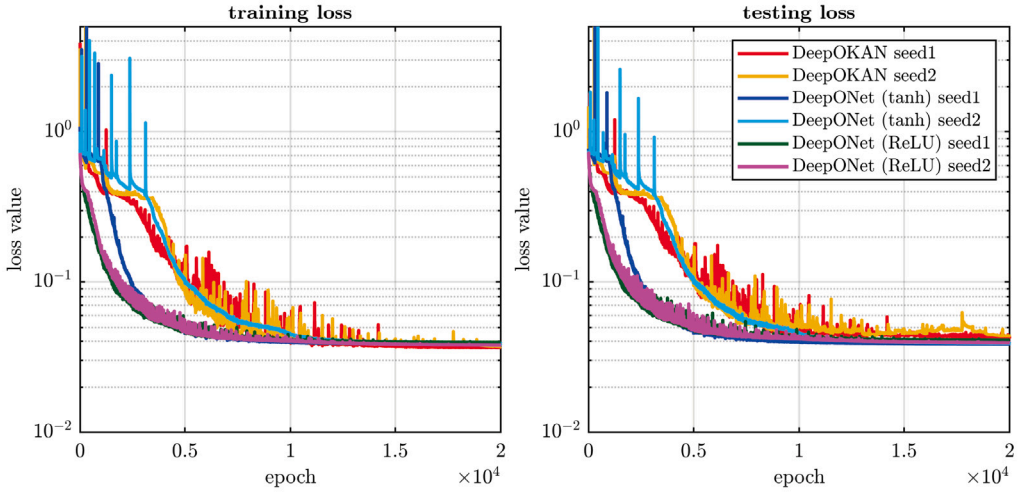

Fig. 13. Deep DeepONet and DeepOKAN loss convergence for y_4 using different seeds.

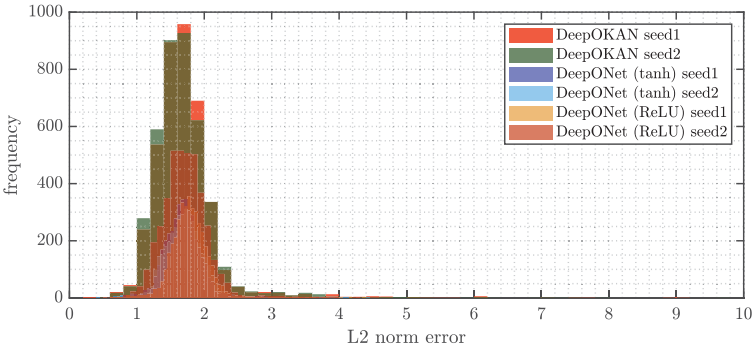

Fig. 14. Histogram of the L2-norm values of the absolute errors for deep operators predicting y_4.

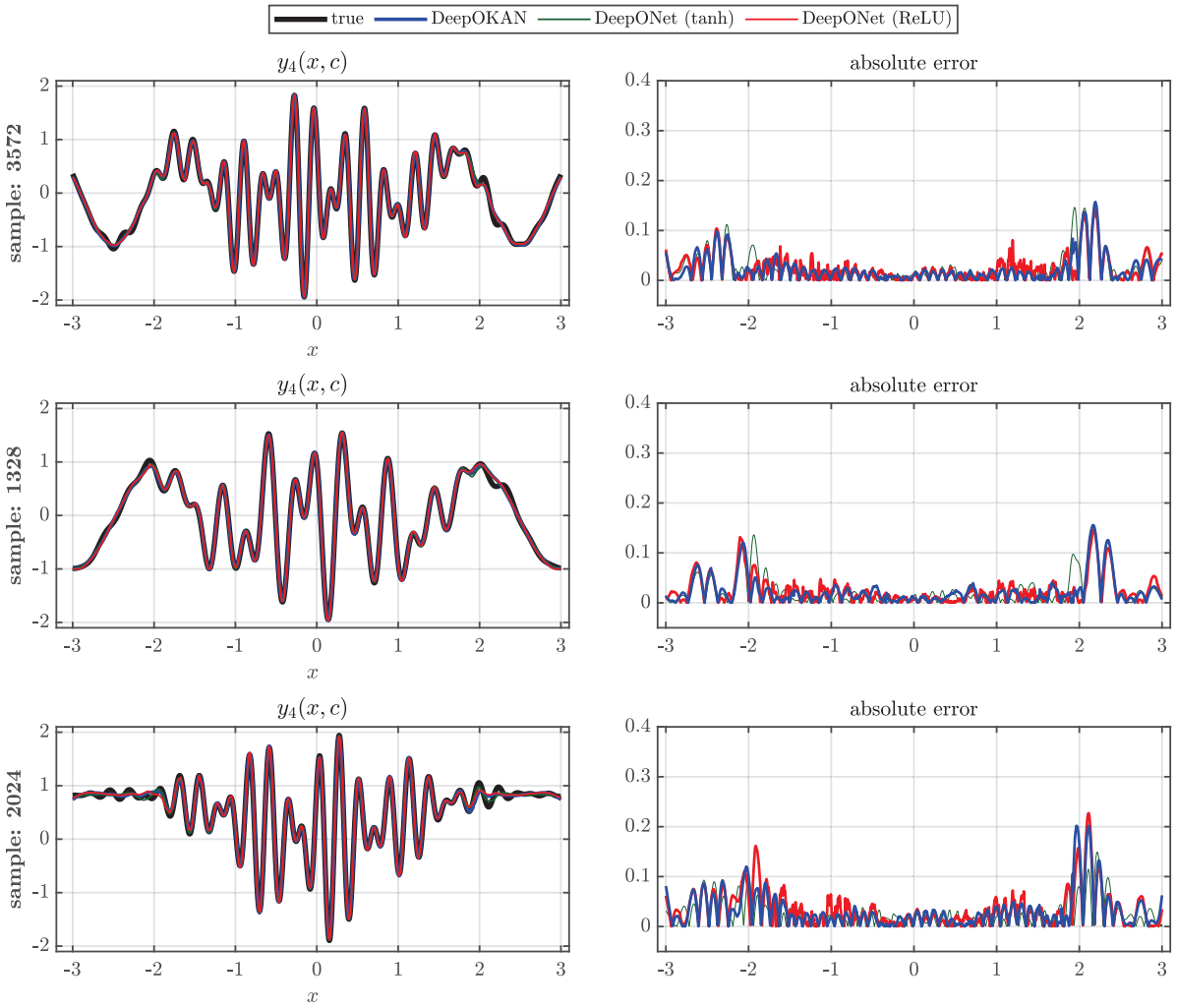

Fig. 15. y_4: Predictions of deep DeepONet and DeepOKAN for different samples randomly picked from the testing dataset.

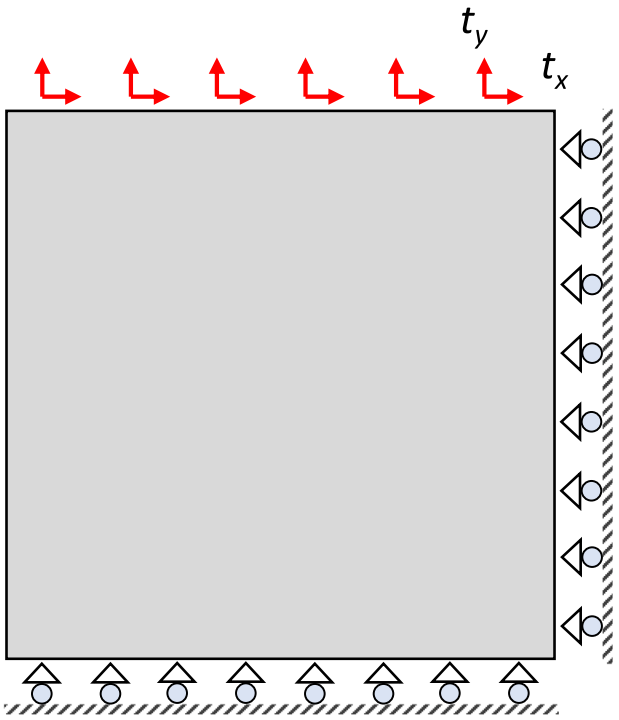

Fig. 16. Schematic of the geometry and boundary conditions of the 2D orthotropic material example.

case, the compliance matrix **S** is expressed as:

$$[\mathbf{S}] = \begin{bmatrix} \frac{1}{E_x} & -\frac{\nu_{xy}}{E_x} & 0 \\ -\frac{\nu_{yx}}{E_y} & \frac{1}{E_y} & 0 \\ 0 & 0 & \frac{1}{G_{xy}} \end{bmatrix}, \tag{22}$$

with E_x and E_y representing Young's moduli in the orthotropic directions, G_{xy} being the shear modulus, and ν_{xy} being the in-plane Poisson's ratio. The constitutive relation symmetry is ensured by:

$$\nu_{xy} = \frac{E_x}{E_y}\nu_{yx}. \tag{23}$$

S has to be inverted to obtain the stiffness matrix.

In this example, we consider a two-dimensional (2D) domain subjected to boundary conditions and traction forces shown in Fig. 16. The bottom and right edges of the domain are subjected to roller boundary conditions. Traction forces are applied at the top edge of the domain, represented by two components, t_x and t_y. These forces are sampled from uniform distributions within a specified range: $t_x, t_y \in [-0.3, 0.3]$. As demonstrated in Eqs. (22) and (23), for a 2D orthotropic material, four constants are required: E_x, E_y, ν_{xy}, and G_{xy}. We sample E_x, E_y, and ν_{xy} from specified uniform distributions: $E_x, E_y \in [5, 20]$, and $\nu_{xy} \in [0.15, 0.35]$. G_{xy} is computed using the following relationship:

$$G_{xy} = \frac{1}{2}\left(\frac{E_x}{2\left(1+\nu_{xy}\right)} + \frac{E_y}{2\left(1+\nu_{yx}\right)}\right). \tag{24}$$

Eq. (24) does not generally hold for orthotropic materials. However, we compute G_{xy} using this equation to ensure that it remains within a physically plausible range, as for most materials, it is observed that they are less stiff in shear than in tension or compression. Hence, we would like to highlight that it is an assumption we made, and one might choose a different range to generate the dataset for training. Although G_{xy} is calculated using Eq. (24) in the data generation phase, it is still fed to the neural operators to learn its impact on the material deformation. We assume zero body force and negligible inertia. This leaves us with six parameters to be fed to the branch network: t_x, t_y, E_x, E_y, ν_{xy}, and G_{xy}, while the coordinates are fed to the trunk network. A total of 5000 samples were generated using this approach. The dataset was then split into 80% for training and 20% for testing.

Table 1
Average training time for different network complexities.

Complexity	Network	Average time (s)
Low	DeepOKAN	6454
	DeepONet	5284
Medium	DeepOKAN	8957
	DeepONet	7280
High	DeepOKAN	17 905
	DeepONet	14 216

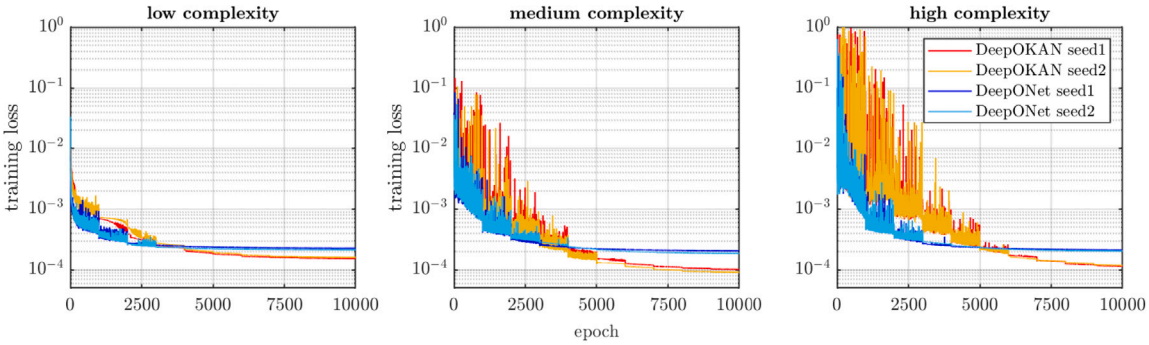

Fig. 17. Evolution of the training loss function for the 2D orthotropic elasticity problem.

3.2.2. Training and results

In this example, we investigate three cases of network complexity: $nc = [low,\ medium,\ high]$. Both DeepOKAN and DeepONet have a single hidden layer in their trunk and branch networks, as well as 5 neurons in the output layer. Across the three levels of complexity, the DeepOKAN has $n = [14,\ 80,\ 190]$ neurons in its hidden layer and $w = [1260,\ 7200,\ 17100]$ trainable parameters. Respectively, the DeepONet has $n = [62,\ 358,\ 855]$ neurons and $w = [1250,\ 7170,\ 17110]$ parameters. Both operators are trained with Adam for 10 000 epochs, using a learning rate $lr = 10^{-3}$ and a batch size of 64. A learning scheduler is applied with $\gamma = 0.5$ and $T_{step} = 1000$ epochs to enhance the training process. Finally, we perform two independent simulations for each operator, using two different seed numbers to initialize their weights ($seed1$ and $seed2$).

Table 1 summarizes the training time of the different network complexities. It is observed that the DeepONet training is slightly faster than the DeepOKAN for the same network complexity. The evolution of the training loss functions for these models is shown in Fig. 17. Across all levels of complexity, we observe that DeepONets experience a sharper decrease in their loss function at the beginning of their training. However, the learning capacity of the network seems to be stalled rather quickly, as evidenced by the almost negligible loss reduction after a few thousand epochs. On the contrary, all DeepOKAN models experience a smoother reduction in the training loss function, ultimately arriving at smaller values at the end of the training. This is the first indication that DeepOKAN may also bear higher accuracy than its DeepONet counterparts for this problem. We compute the L2-norm of the absolute error values for all the test samples and plot their histograms in Fig. 18 to evaluate this assumption. For visual clarity, the top row of graphs represents the $seed1$ models, while the bottom graphs represent $seed2$. Here, we observe that DeepOKANs attain smaller values in the L2-norm, clearly outperforming their DeepONets counterparts. This observation holds true for both seed cases, regardless of the level of complexity.

To further solidify these observations, we randomly select three test samples and in Fig. 19 we plot their true values, predictions and absolute error maps. The medium complexity DeepOKAN and DeepONet with $seed1$ are chosen for evaluation purposes. Even though both operators capture the true field with sufficient accuracy, both qualitatively and quantitatively, DeepOKANs are clearly more accurate than DeepONets, as shown by the absolute error contour maps. Here we emphasize that the colorbar of the absolute error graphs is capped between 0 and 0.0005.

3.3. Transient Poisson's problem

3.3.1. Problem description, FEA, and data generation

In this example, we will demonstrate how the DeepOKAN can solve the transient Poisson problem. Consider a homogeneous unit square experiencing a time-dependent DBC at the right edge, while zero DBC is imposed at the left edge of the square. The top and bottom edges experience zero NBCs. Fig. 20a summarizes the domain and boundary conditions used for this example. This transient

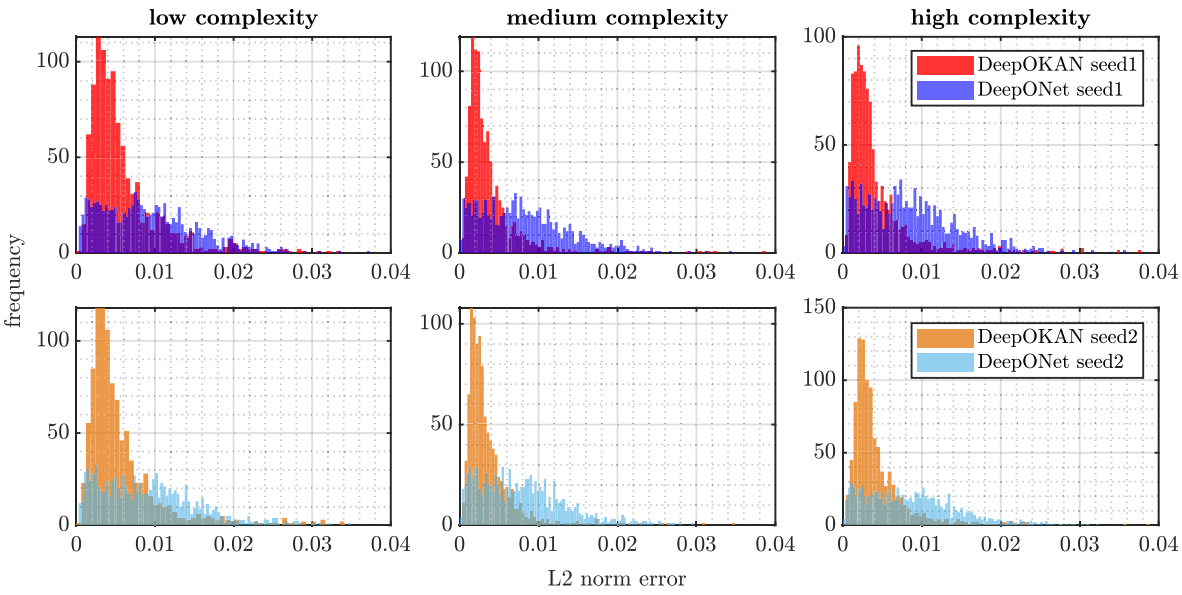

Fig. 18. Histograms of the L2-norm values of the absolute errors for the 2D orthotropic elasticity problem.

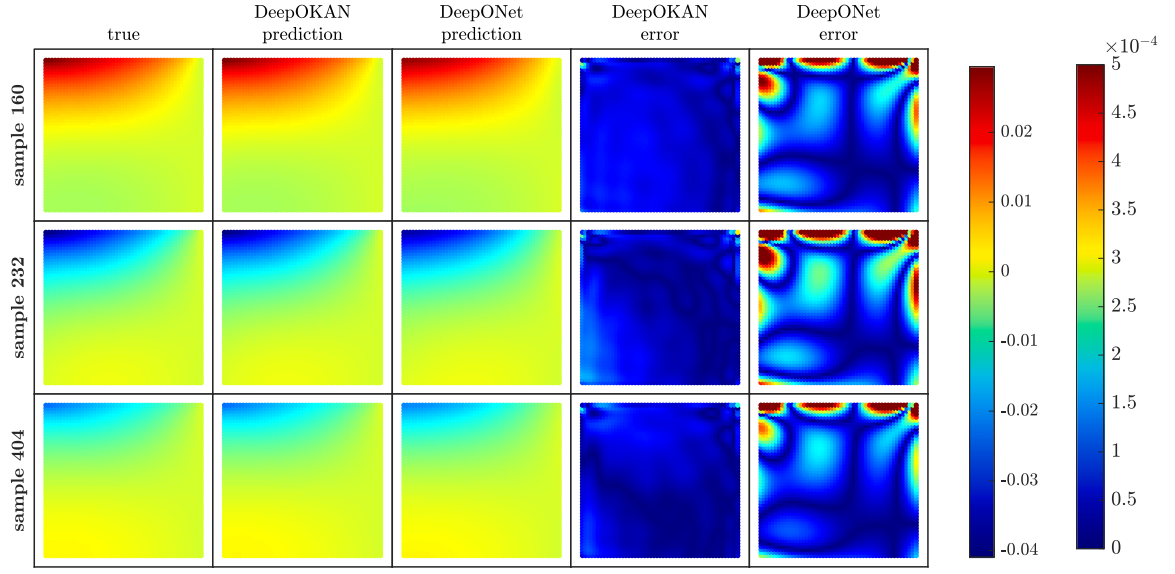

Fig. 19. Comparison between true and predicted values for the 2D orthotropic elasticity problem. The left color bar is for the solution field (true/prediction) contours, while the right color bar is for the error contours. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Poisson problem can be described by the partial differential equation (PDE):

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u &= f \qquad \boldsymbol{x} \in \Omega,\ t \in (0, T], \\ u(x, 0) &= u_0(x) \qquad \boldsymbol{x} \in \Omega \\ u &= \overline{u}(t) \qquad \boldsymbol{x} \in \Gamma_u \\ \nabla u \cdot \mathbf{n} &= 0 \qquad \boldsymbol{x} \in \Gamma_t \end{aligned} \tag{25}$$

where Δ is the Laplacian, and f is the source term, set to zero for this example. $u_0(x)$ is the initial condition, and it is set to zero, i.e., $u_0(x) = 0$.

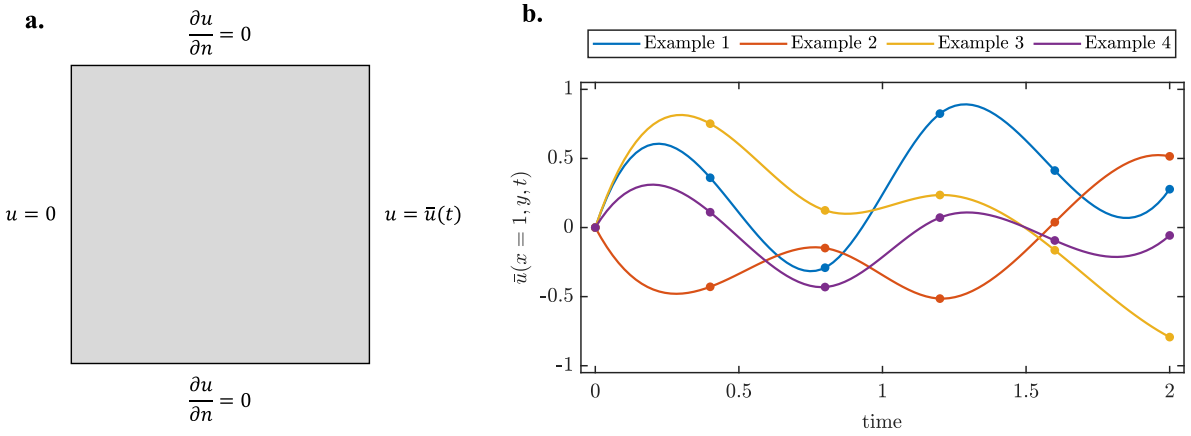

Fig. 20. a. Schematic of the geometry and boundary conditions of the 2D transient Poisson example. b. Examples of the DBCs imposed on the right edge of the unit square domain. The points appearing on the curves represent the control points sampled from the normal distribution and then used for interpolation.

To generate the dataset for training and testing neural operators on the transient Poisson problem with a time-dependent Dirichlet boundary condition applied to the right edge of a unit square domain, we begin by randomly sampling five control points. These control points are generated by first creating a sequence of time stamps linearly spaced between the initial time $t = 0$ and the final time $t = T$ and then sampling corresponding u values from a normal distribution $\mathcal{N}(0, \sigma = 0.5)$, which are clipped to remain within the range $[-1, 1]$. The initial condition at $t = 0$ with $u = 0$ is explicitly included. The complete set of six points (origin plus five randomly sampled points) is then interpolated using a cubic spline function, denoted as $CS(t)$, to ensure a smooth transition between these points. Mathematically, this can be expressed as:

$$CS(t) = \sum_{i=0}^{n} c_i B_i(t) \tag{26}$$

where $B_i(t)$ are the basis functions of the cubic spline, and c_i are the coefficients determined by the control points. This interpolation is evaluated at 100 evenly spaced time points between $t = 0$ and $t = T$ to generate a time series of u values. Fig. 20b shows a few examples of the generated boundary conditions imposed on the right edge of the domain.

Next, these time-dependent DBCs are used in the FEA to solve for the corresponding transient solutions. The FEA is performed using FEniCSx. A total of 4500 samples were generated using this approach. The dataset was then divided into 80% for training and 20% for testing. The branch network takes the boundary condition history at the right edge $\bar{u}(x = 1, y, t)$. The length of the branch input vector is $M = 100$, where each entry corresponds to one point in time. On the other hand, the trunk takes the coordinates of the domain. Since we are interested in transient response at the M time increments, we must devise the branch output to reflect such a dimensionality. The branch output has a length of $M \times r$. Then, we reshape the output to (M, r) to prepare it for the multiplication with the trunk output with a length of r. Unlike the previous examples, the multiplication yields an output of length M representing the solution history:

$$\hat{F}(q)(\boldsymbol{X}) = \sum_{i=1}^{r} b_{mi} t_i + B_m \tag{27}$$

where $m = [1, 2, \ldots, M]$ is the time increment.

3.3.2. Training and results

In this example, we consider three cases of network complexity: nc = low, nc = medium, and nc = high. For the low complexity case, the DeepOKAN has $n = 5$ neurons in the hidden layer, while the DeepONet has $n = 25$. This leads to $w = 13\,104$ and $w = 12\,650$ learnable parameters for the DeepONet and DeepOKAN, respectively. For the medium complexity case, the DeepOKAN has $n = 10$ with $w = 25\,300$ learnable parameters, while the DeepONet has $n = 50$ with $w = 25\,804$ learnable parameters. For the high complexity case, $n = 20$ are assigned to the DeepOKAN, leading to $w = 50\,600$ learnable parameters, and $n = 100$ are assigned to the DeepONet, yielding $w = 51\,204$ learnable parameters. In all cases, r is set to $r = 4$. Fig. 21 presents training loss curves for the DeepOKAN and DeepONet across low, medium, and high complexity levels. Each subplot displays the training loss over epochs for two seeds for each neural operator. DeepOKAN consistently achieves lower training losses than DeepONet. This trend is evident across low, medium, and high complexity levels, indicating DeepOKAN's superior training efficiency and performance stability regardless of complexity.

Next, we perform a statistical analysis for the DeepONet and DeepOKAN using the testing dataset, where the analysis is done using the high-complexity network of each model. The analysis reveals that DeepOKAN consistently outperforms DeepONet. This is evident from Fig. 22a and Table 2. Table 2 presents the statistics: the mean L2-norm error for DeepONet is 0.02980 with a standard deviation of 0.0302, whereas DeepOKAN has a mean error of 0.0047 and a standard deviation of 0.0052. The median,

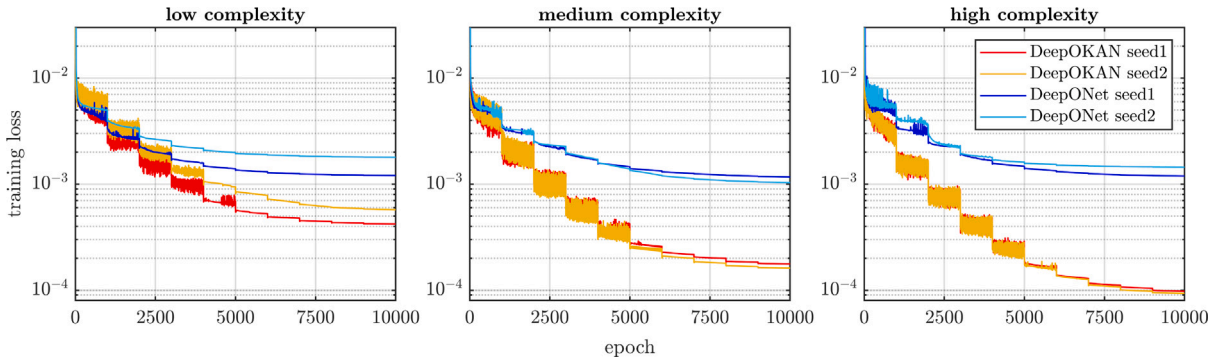

Fig. 21. Training loss for the transient Poisson problem: Two different seeds for each network complexity level.

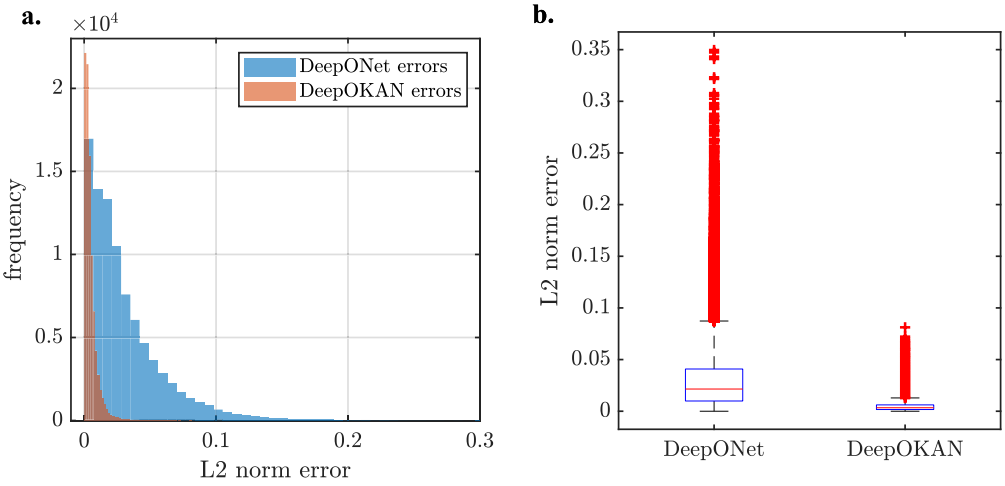

Fig. 22. Histogram and box plot of L2-norm errors for DeepONet and DeepOKAN on the testing dataset.

Table 2
Statistical analysis of L2-norm errors for the DeepONet and DeepOKAN.

Network	Mean	Std deviation	Median	25th percentile	75th percentile
DeepONet	0.02980	0.0302	0.0214	0.0099	0.0408
DeepOKAN	0.0047	0.0052	0.0033	0.0016	0.0061

Table 3
Average training time for different network complexities.

Complexity	Network	Average time (s)
Low	DeepONet	16 325
	DeepOKAN	16 853
Medium	DeepONet	16 369
	DeepOKAN	16 889
High	DeepONet	16 423
	DeepOKAN	16 957

25th percentile, and 75th percentile values also highlight DeepOKAN's superior performance. Fig. 22b illustrates that DeepONet has more outliers and greater variability, indicating less stable performance. The histogram further demonstrates that DeepOKAN's errors are more tightly clustered around smaller values. Finally, Figs. 23 and 24 compare the solutions obtained from the DeepONet and DeepOKAN and their corresponding errors for two sample examples. Altogether, these figures further manifest the conclusion that DeepOKAN outperforms the DeepONet. Regarding the training time, both operators show similar average training time, while DeepONet is slightly faster, as presented in Table 3.

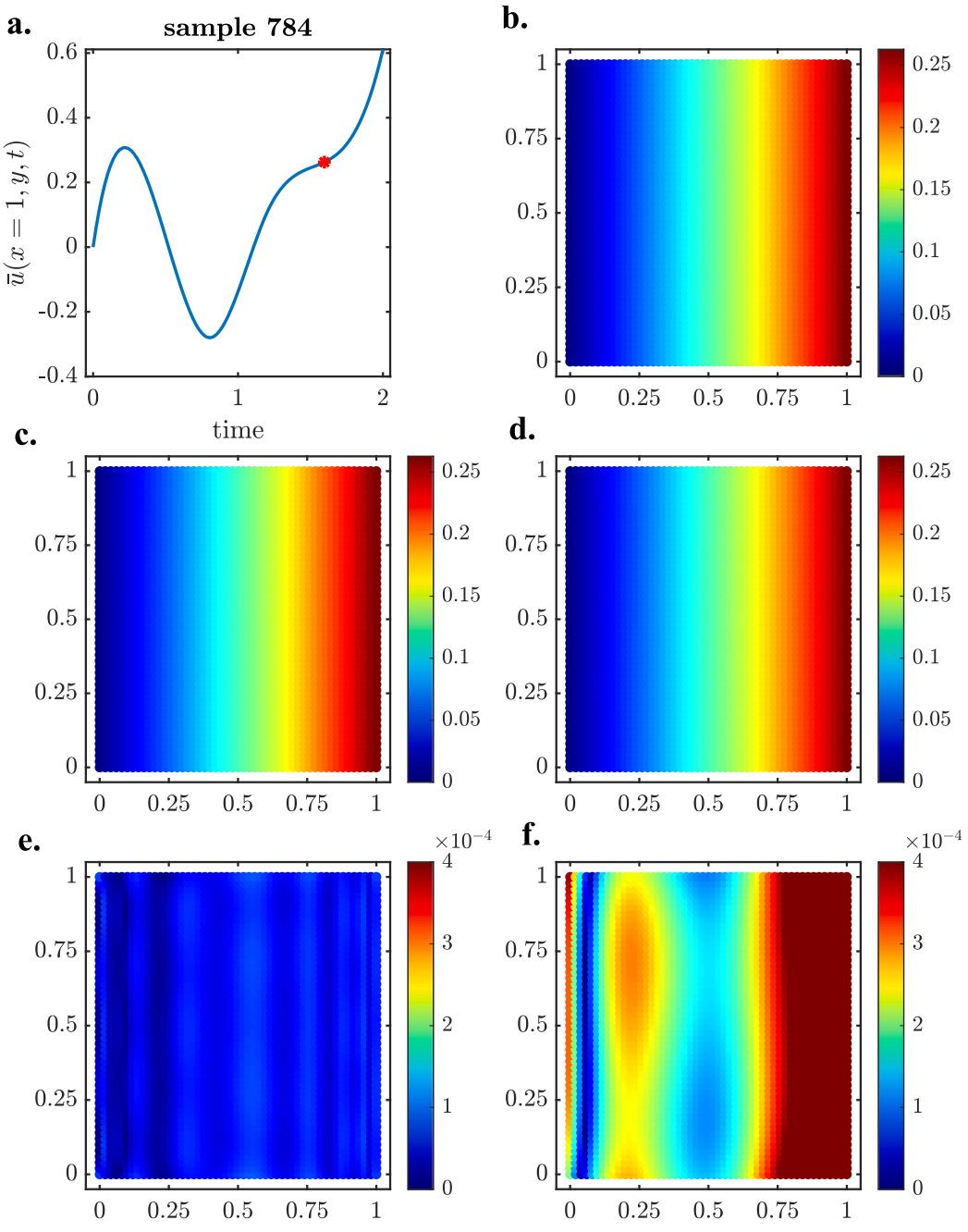

Fig. 23. Comparison between the solutions and errors of the DeepONet and DeepOKAN. The contours are for a randomly selected point from the testing dataset and time increment (shown as a red point in a.). a. DBC history imposed on the right edge of the unit square domain. b. Ground-truth solution obtained using the FEA. c. DeepOKAN prediction. d. DeepONet prediction. e. DeepOKAN error. f. DeepONet error. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4. Conclusions, limitations, and future work

Neural operators are machine learning models that approximate the solutions of partial differential equations by learning the mapping between function spaces, providing an efficient and accurate tool for solving complex, high-dimensional problems in physics and engineering. One of the most popular neural operators is the DeepONet, which utilizes the Multi-Layer Perceptron (MLP) architecture in the branch and trunk, where the MLP is based on interleaving affine transformations and nonlinear activation

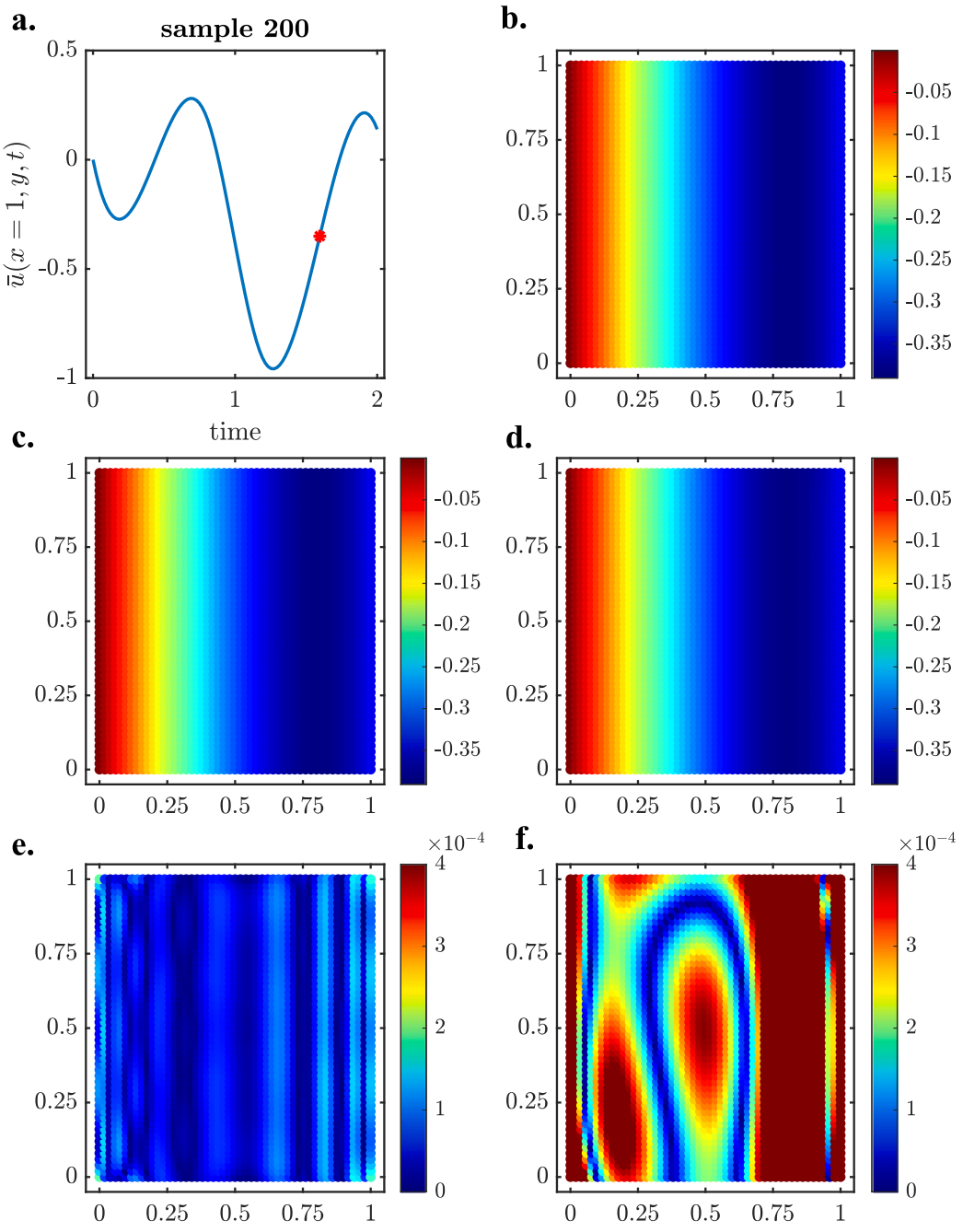

Fig. 24. Comparison of DeepOKAN and DeepONet for another test example. The figure layout is identical to Fig. 23.

functions. MLPs treat the linear transformations and activation functions separately. This paper proposes a new neural operator based on KANs called DeepOKAN. The proposed operator uses KANs in the branch and trunk components, where KANs have learnable activation functions on the edges of the network and sum operations on the nodes. One may consider different bases for these activation functions. This paper uses Gaussian RBFs as a basis for the KANs, appearing in both the branch and trunk. The classical data-driven DeepONet framework is effective; however, the proposed DeepOKAN outperforms it.

As discussed earlier, the present paper uses Gaussian RBF as a basis function for the KANs. RBF kernels naturally capture local patterns in the input space through their distance-based activation, making them particularly effective for learning localized features compared to other basis functions [58]. Having said that, several researchers studied KANs with different basis function

Table A.1
Hyperparameters of the DeepOKAN used for studying y_3 in Section 3.1.

Number of branch hidden layers	2
Number of trunk hidden layers	2
Number of neurons	50
r	4
Total number of learnable parameters	276 560
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-2}, 10^{-3}

Table A.2
Hyperparameters of the DeepONet used for studying y_3 in Section 3.1.

Number of branch hidden layers	2
Number of trunk hidden layers	2
Number of neurons	350
r	4
Total number of learnable parameters	275 880
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-2}, 10^{-3}

choices [30,39–43,59,60]. It would be intriguing to investigate the effect of the basis function choice on the performance of the developed KANs in future work. Additionally, further investigations involving comprehensive benchmarks and comparisons with other neural operator architectures, such as the Fourier Neural Operator (FNO) and different types of DeepONets, are necessary [21,24,26]. These additional benchmarks would help evaluate DeepOKAN's relative performance, highlighting its strengths and potential limitations compared to other state-of-the-art methods.

CRediT authorship contribution statement

Diab W. Abueidda: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Panos Pantidis:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation. **Mostafa E. Mobasher:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no conflict of interest.

Acknowledgments

This work was partially supported by the Sand Hazards and Opportunities for Resilience, Energy, and Sustainability (SHORES) Center, United Arab Emirates, funded by Tamkeen under the NYUAD Research Institute Award CG013. The authors wish to thank the NYUAD Center for Research Computing for providing resources, services, and skilled personnel. This work was partially supported by Sandooq Al Watan Applied Research and Development (SWARD), United Arab Emirates, funded by Grant No. : SWARD-F22-018.

Appendix. Supporting information

A.1. Summary of hyperparameters

See Tables A.1–A.10.

A.2. Effect of dataset size

See Fig. 25.

Data availability

Data available at https://github.com/DiabAbu/DeepOKAN.

Table A.3
Hyperparameters of the shallow DeepOKAN used for y_4 in Section 3.1.

Number of branch hidden layers	2
Number of trunk hidden layers	2
Number of neurons	50
r	40
Total number of learnable parameters	276 560
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-3}

Table A.4
Hyperparameters of the shallow DeepONet used for y_4 in Section 3.1.

Number of branch hidden layers	2
Number of trunk hidden layers	2
Number of neurons	350
r	40
Total number of learnable parameters	275 880
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-3}

Table A.5
Hyperparameters of the deep DeepOKAN used for y_4 in Section 3.1.

Number of branch hidden layers	5
Number of trunk hidden layers	5
Number of neurons	40
r	40
Total number of learnable parameters	485 760
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-3}

Table A.6
Hyperparameters of the deep DeepONet used for y_4 in Section 3.1.

Number of branch hidden layers	5
Number of trunk hidden layers	5
Number of neurons	240
r	40
Total number of learnable parameters	483 440
Scheduler γ	0.9
Scheduler T_{step}	500
Starting lr	10^{-3}

Table A.7
Hyperparameters of the DeepOKAN used in Section 3.2.

	Low complexity	Medium complexity	High complexity
Number of branch hidden layers	1	1	1
Number of trunk hidden layers	1	1	1
Number of neurons	14	80	190
r	5	5	5
Total number of learnable parameters	1260	7200	17 100
Scheduler γ	0.5	0.5	0.5
Scheduler T_{step}	1000	1000	1000
Starting lr	10^{-3}	10^{-3}	10^{-3}

Table A.8
Hyperparameters of the DeepONet used in Section 3.2.

	Low complexity	Medium complexity	High complexity
Number of branch hidden layers	1	1	1
Number of trunk hidden layers	1	1	1
Number of neurons	62	358	855
r	5	5	5
Total number of learnable parameters	1250	7170	17 110
Scheduler γ	0.5	0.5	0.5
Scheduler T_{step}	1000	1000	1000
Starting lr	10^{-3}	10^{-3}	10^{-3}

Table A.9
Hyperparameters of the DeepOKAN used in Section 3.3.

	Low complexity	Medium complexity	High complexity
Number of branch hidden layers	1	1	1
Number of trunk hidden layers	1	1	1
Number of neurons	5	10	20
r	4	4	4
Total number of learnable parameters	12 650	25 300	50 600
Scheduler γ	0.5	0.5	0.5
Scheduler T_{step}	1000	1000	1000
Starting lr	10^{-3}	10^{-3}	10^{-3}

Table A.10
Hyperparameters of the DeepONet used in Section 3.3.

	Low complexity	Medium complexity	High complexity
Number of branch hidden layers	1	1	1
Number of trunk hidden layers	1	1	1
Number of neurons	25	50	100
r	4	4	4
Total number of learnable parameters	13 104	25 804	51 204
Scheduler γ	0.5	0.5	0.5
Scheduler T_{step}	1000	1000	1000
Starting lr	10^{-3}	10^{-3}	10^{-3}

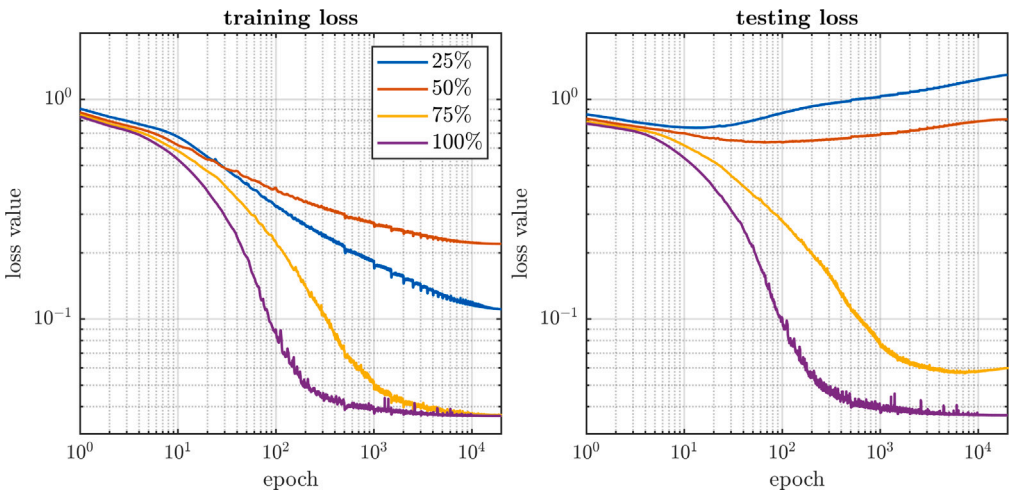

Fig. 25. Effect of dataset size for the shallow DeepOKAN. It demonstrates the training and testing loss histories for y_4 with different training dataset sizes, while the testing dataset (4000 samples)is fixed for the different trials. The total size of the training dataset is 16 000 samples, where a different percentage is used for each trial.

References

[1] D. Ammosov, M. Vasilyeva, Online multiscale finite element simulation of thermo-mechanical model with phase change, Computation 11 (4) (2023) 71.
[2] Y. Efendiev, T.Y. Hou, Multiscale Finite Element Methods: Theory and Applications, vol. 4, Springer Science & Business Media, 2009.
[3] M.E. Mobasher, H. Waisman, Dual length scale non-local model to represent damage and transport in porous media, Comput. Methods Appl. Mech. Engrg. 387 (2021) 114154.
[4] M. Torzoni, M. Tezzele, S. Mariani, A. Manzoni, K.E. Willcox, A digital twin framework for civil engineering structures, Comput. Methods Appl. Mech. Engrg. 418 (2024) 116584.
[5] S. Kushwaha, J. He, D. Abueidda, I. Jasiuk, Designing impact-resistant bio-inspired low-porosity structures using neural networks, J. Mater. Res. Technol. 27 (2023) 767–779.
[6] M. Valizadeh, S.J. Wolff, Convolutional neural network applications in additive manufacturing: A review, Adv. Ind. Manuf. Eng. 4 (2022) 100072.
[7] P. Pantidis, H. Eldababy, D. Abueidda, M.E. Mobasher, I-FENN with Temporal Convolutional Networks: Expediting the load-history analysis of non-local gradient damage propagation, Comput. Methods Appl. Mech. Engrg. 425 (2024) 116940.
[8] B. Paermentier, D. Debruyne, R. Talemi, A machine learning based sensitivity analysis of the GTN damage parameters for dynamic fracture propagation in X70 pipeline steel, Int. J. Fract. 227 (1) (2021) 111–132.
[9] A. Olivier, M.D. Shields, L. Graham-Brady, Bayesian neural networks for uncertainty quantification in data-driven materials modeling, Comput. Methods Appl. Mech. Eng. 386 (2021) 114079.
[10] C.M. Parrott, D.W. Abueidda, K.A. James, Multidisciplinary topology optimization using generative adversarial networks for physics-based design enhancement, J. Mech. Des. 145 (6) (2023) 061704.
[11] L. Herrmann, S. Kollmannsberger, Deep learning in computational mechanics: A review, Comput. Mech. (2024) 1–51.
[12] Z. Li, Neural operator: Learning maps between function spaces, in: 2021 Fall Western Sectional Meeting, AMS, 2021.
[13] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, 2020, arXiv preprint arXiv:2010.08895.
[14] Z. Li, D.Z. Huang, B. Liu, A. Anandkumar, Fourier neural operator with learned deformations for PDEs on general geometries, J. Mach. Learn. Res. 24 (388) (2023) 1–26.
[15] L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators, Nat. Mach. Intell. 3 (3) (2021) 218–229.
[16] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, DeepXDE: A deep learning library for solving differential equations, SIAM Rev. 63 (1) (2021) 208–228.
[17] L. Lu, R. Pestourie, S.G. Johnson, G. Romano, Multifidelity deep neural operators for efficient learning of partial differential equations with application to fast inverse design of nanoscale heat transport, Phys. Rev. Res. 4 (2) (2022) 023210.
[18] S. Goswami, M. Yin, Y. Yu, G.E. Karniadakis, A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials, Comput. Methods Appl. Mech. Engrg. 391 (2022) 114587.
[19] K. Kobayashi, J. Daniell, S.B. Alam, Improved generalization with deep neural operators for engineering systems: Path towards digital twin, Eng. Appl. Artif. Intell. 131 (2024) 107844.
[20] S. Garg, S. Chakraborty, VB-DeepONet: A Bayesian operator learning framework for uncertainty quantification, Eng. Appl. Artif. Intell. 118 (2023) 105685.
[21] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, G.E. Karniadakis, A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data, Comput. Methods Appl. Mech. Engrg. 393 (2022) 114778.
[22] J. He, S. Koric, S. Kushwaha, J. Park, D. Abueidda, I. Jasiuk, Novel DeepONet architecture to predict stresses in elastoplastic structures with variable complex geometries and loads, Comput. Methods Appl. Mech. Engrg. 415 (2023) 116277.
[23] J. He, S. Kushwaha, J. Park, S. Koric, D. Abueidda, I. Jasiuk, Sequential Deep Operator Networks (S-DeepONet) for predicting full-field solutions under time-dependent loads, Eng. Appl. Artif. Intell. 127 (2024) 107258.
[24] J. He, S. Kushwaha, J. Park, S. Koric, D. Abueidda, I. Jasiuk, Predictions of transient vector solution fields with sequential deep operator network, Acta Mech. (2024) 1–16.
[25] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Neural operator: Graph kernel network for partial differential equations, 2020, arXiv preprint arXiv:2003.03485.
[26] J. He, S. Koric, D. Abueidda, A. Najafi, I. Jasiuk, Geom-DeepONet: A point-cloud-based deep operator network for field predictions on 3D parameterized geometries, Comput. Methods Appl. Mech. Eng. 429 (2024) 117130.
[27] W. Zhong, H. Meidani, Physics-informed Mesh-independent Deep Compositional Operator Network, 2024, arXiv preprint arXiv:2404.13646.
[28] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst. 2 (4) (1989) 303–314.
[29] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366.
[30] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T.Y. Hou, M. Tegmark, KAN: Kolmogorov-Arnold networks, 2024, arXiv preprint arXiv:2404.19756.
[31] N. Kolmogorov, On the Representation of Continuous Functions of Several Variables by Superpositions of Continuous Functions of a Smaller Number of Variables, American Mathematical Society, 1961.
[32] V.I. Arnol'd, On the representation of continuous functions of three variables by superpositions of continuous functions of two variables, Mat. Sb. 90 (1) (1959) 3–74.
[33] J. Braun, M. Griebel, On a constructive proof of Kolmogorov's superposition theorem, Constr. Approx. 30 (2009) 653–675.
[34] D.A. Sprecher, S. Draghici, Space-filling curves and Kolmogorov superposition-based neural networks, Neural Netw. 15 (1) (2002) 57–67.
[35] M. Köppen, On the training of a Kolmogorov network, in: Artificial Neural Networks—ICANN 2002: International Conference Madrid, Spain, August 28–30, 2002 Proceedings 12, Springer, 2002, pp. 474–479.
[36] J.-N. Lin, R. Unbehauen, On the realization of a Kolmogorov network, Neural Comput. 5 (1) (1993) 18–20.
[37] M.-J. Lai, Z. Shen, The Kolmogorov superposition theorem can break the curse of dimensionality when approximating high dimensional functions, 2021, arXiv preprint arXiv:2112.09963.
[38] D. Fakhoury, E. Fakhoury, H. Speleers, ExSpliNet: An interpretable and expressive spline-based neural network, Neural Netw. 152 (2022) 332–346.
[39] R. Genet, H. Inzirillo, TKAN: Temporal Kolmogorov-Arnold Networks, 2024, arXiv preprint arXiv:2405.07344.
[40] Z. Li, Kolmogorov-Arnold networks are radial basis function networks, 2024, arXiv preprint arXiv:2405.06721.
[41] S. SS, Chebyshev polynomial-based Kolmogorov-Arnold networks: An efficient architecture for nonlinear function approximation, 2024, arXiv preprint arXiv:2405.07200.
[42] J. Xu, Z. Chen, J. Li, S. Yang, W. Wang, X. Hu, E.C.-H. Ngai, FourierKAN-GCF: Fourier Kolmogorov-Arnold network–An effective and efficient feature transformation for graph collaborative filtering, 2024, arXiv preprint arXiv:2406.01034.
[43] K. Shukla, J.D. Toscano, Z. Wang, Z. Zou, G.E. Karniadakis, A comprehensive and FAIR comparison between MLP and KAN representations for differential equations and operator networks, 2024, arXiv preprint arXiv:2406.02917.
[44] C.J. Vaca-Rubio, L. Blanco, R. Pereira, M. Caus, Kolmogorov-Arnold networks (KANs) for time series analysis, 2024, arXiv preprint arXiv:2405.08790.

[45] M.E. Samadi, Y. Müller, A. Schuppert, Smooth Kolmogorov Arnold networks enabling structural knowledge representation, 2024, arXiv preprint arXiv:2405.11318.
[46] M.D. Buhmann, Radial basis functions, Acta Numer. 9 (2000) 1–38.
[47] G. Arora, KiranBala, H. Emadifar, M. Khademi, A review of radial basis function with applications explored, J. Egyptian Math. Soc. 31 (1) (2023) 6.
[48] M.E. Chenoweth, A Local Radial Basis Function Method for the Numerical Solution of Partial Differential Equations (Master's thesis), Marshall University, 2012.
[49] D.N. Kien, X. Zhuang, Radial basis function based finite element method: Formulation and applications, Eng. Anal. Bound. Elem. 152 (2023) 455–472.
[50] E.M. Elsheikh, T.H. Naga, Y.F. Rashed, Efficient fundamental solution based finite element for 2-d dynamics, Eng. Anal. Bound. Elem. 148 (2023) 376–388.
[51] Y.F. Rashed, Transient dynamic boundary element analysis using Gaussian-based mass matrix, Eng. Anal. Bound. Elem. 26 (3) (2002) 265–279.
[52] W. Li, Y. Guan, D. Huang, N. Trisovic, Gaussian RBFNN method for solving FPK and BK equations in stochastic dynamical system with FOPID controller, Int. J. Non-Linear Mech. 153 (2023) 104403.
[53] Q. Li, R. Li, D. Huang, Dynamic analysis of a new 4D fractional-order financial system and its finite-time fractional integral sliding mode control based on RBF neural network, Chaos Solitons Fractals 177 (2023) 114156.
[54] L. Wang, J.-S. Chen, H.-Y. Hu, Subdomain radial basis collocation method for fracture mechanics, Int. J. Numer. Methods Eng. 83 (7) (2010) 851–876.
[55] P. Wen, M. Aliabadi, Meshless method with enriched radial basis functions for fracture mechanics, Struct. Durab. Health Monit. 3 (2) (2007) 107.
[56] S. Satapathy, S. Dehuri, A. Jagadev, S. Mishra, Empirical study on the performance of the classifiers in EEG classification, EEG Brain Signal Classif. Epileptic Seizure Disord. Detect. (2019) 45–65.
[57] S. Wang, H. Wang, P. Perdikaris, Learning the solution operator of parametric partial differential equations with physics-informed DeepONets, Sci. Adv. 7 (40) (2021) eabi8605.
[58] L. Yingwei, N. Sundararajan, P. Saratchandran, Performance evaluation of a sequential minimal radial basis function (RBF) neural network learning algorithm, IEEE Trans. Neural Netw. 9 (2) (1998) 308–318.
[59] A.A. Aghaei, fKAN: Fractional Kolmogorov-arnold networks with trainable Jacobi basis functions, 2024, arXiv preprint arXiv:2406.07456.
[60] Z. Bozorgasl, H. Chen, Wav-KAN: Wavelet Kolmogorov-Arnold networks, 2024, arXiv preprint arXiv:2405.12832.