Journal of Computational Physics 520 (2025) 113433

Contents lists available at ScienceDirect

Journal of Computational Physics

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

RandONets: Shallow networks with random projections for learning linear and nonlinear operators

Gianluca Fabiani [a], Ioannis G. Kevrekidis [b,c,d], Constantinos Siettos [e,*], Athanasios N. Yannacopoulos [f]

[a] Modelling Engineering Risk and Complexity, Scuola Superiore Meridionale, Naples, 80138, Italy
[b] Department of Chemical and Biomolecular Engineering, Johns Hopkins University, Baltimore, MD, USA
[c] Department of Applied Mathematics and Statistics, Johns Hopkins University, Baltimore, MD, USA
[d] Medical School, Department of Urology, Johns Hopkins University, Baltimore, MD, USA
[e] Dipartimento di Matematica e Applicazioni "Renato Caccioppoli", Università degli Studi di Napoli Federico II, Naples, Italy
[f] Department of Statistics, Athens University of Economics and Business, Athens, Greece

ARTICLE INFO

Dataset link: https://github.com/GianlucaFabiani/RandONets

MSC:
65M32
65D12
65J22
41A35
68T20
65D15
68T07
68W20

Keywords:
RandONets
Interpretable machine learning
Random projections
Shallow neural networks
Linear and nonlinear operators
Numerical analysis

ABSTRACT

Deep neural networks have been extensively used for the solution of both the forward and the inverse problem for dynamical systems. However, their implementation necessitates optimizing a high-dimensional space of parameters and hyperparameters. This fact, along with the requirement of substantial computational resources, pose a barrier to achieving high numerical accuracy, but also interpretability. Here, to address the above challenges, we present Random Projection-based Operator Networks (RandONets): shallow networks with random projections and tailor-made numerical analysis methods that learn accurately and fast linear and nonlinear operators. Building on previous works, we prove that RandOnets are universal approximators of linear and nonlinear operators. Due to their simplicity, RandONets provide a one-step transformation of the input space, facilitating interpretability. For the evaluation of their performance, we focus on operators of PDEs. We show, that RandONets outperform by several orders of magnitude, both in terms of numerical approximation accuracy and computational cost, the "vanilla" DeepONets. Hence, we believe that our method will trigger further developments in the field of scientific machine learning, for the development of new ''light"schemes that will provide high accuracy while reducing dramatically the computational cost. A MATLAB toolbox for RandONets, including demos, is available on GitHub at https://github.com/GianlucaFabiani/RandONets.

1. Introduction

In recent years, significant advancements in machine learning (ML) have broadened our computational toolkit with the ability to solve both the forward and, importantly, the inverse problem in differential equations and multiscale/complex systems. For the forward problem, ML algorithms such as Gaussian process and physics-informed neural networks (PINNs) are trained to approximate the solutions of nonlinear differential equations, with a particular interest in stiff and high-dimensional systems of nonlinear differential

* Corresponding author at: Dipartimento di Matematica e Applicazioni "Renato Caccioppoli", Università degli Studi di Napoli "Federico II", Naples, Italy.
E-mail address: constantinos.siettos@unina.it (C. Siettos).

https://doi.org/10.1016/j.jcp.2024.113433
Received 9 June 2024; Accepted 9 September 2024

Available online 1 October 2024

equations [1–8], as well as for the solution of nonlinear functional equations [9–14]. The solution of the inverse problem leverages the ability of ML algorithms to learn the physical laws, their parameters and closures among scales from data [15,3,16,9,17–20,10,11]. To the best of our knowledge, the first neural network-based solution of the inverse problem for identifying the evolution law (the right-hand-side) of parabolic Partial Differential Equations (PDEs), using spatial partial derivatives as basis functions, was presented in Gonzalez et al. (1998) [21]. In the same decade, such inverse identification problems for PDEs, were investigated through reduced order models (ROMs) for PDEs, using data-driven Proper Orthogonal Decomposition (POD) basis functions [22] and Fourier basis functions (in a context of approximate inertial manifolds) in [23].

Over the last few years several advanced ML-based approaches for the approximation of nonlinear operators, focused on partial differential operators, stand out: the Deep Operator Networks (DeepONets) [10], the Fourier Neural Operators (FNOs) [24], and the Graph-based Neural Operators [25,26] are the most prominent ones. DeepONets extend the universal approximation theorem for dynamical systems –given back in 90's by Chen and Chen [27]– employing the so-called branch and trunk networks that handle input functions and spatial variables and/or parameters separately, thus providing a powerful and versatile framework for operator learning in dynamical systems. Various architectures can be used for either/both networks, thus offering new avenues for tackling challenging problems in the modeling of complex dynamical systems [10,28–30]. FNOs [24] exploit the Fourier transform to capture global patterns and dependencies in the data. FNOs employ convolutional layers in the frequency domain and the inverse Fourier transform to map back to the original domain. This transformation allows the neural network to efficiently learn complex, high-dimensional inputs and outputs with long range correlations. This method is particularly advantageous for problems involving large spatial domains. The family of graph-based neural Operators [26,25] model the nonlinear operator as a graph –where nodes represent spatial locations of the output function– learning the kernel of the network which approximates the PDE. They define a sequence of compositions, where each layer is a map between infinite-dimensional spaces with a finite-dimensional parametric dependence. The authors also prove a universal approximation theorem, showing that the formulation can approximate any given nonlinear continuous operator. Building on the above pioneering methods, other approaches include wavelet neural operators (WNOs) [31] and spectral neural operators (SNOs) [32] using a fixed number of basis functions for both the input and the output functions, which can be either Chebyshev polynomials or complex exponentials. For a comprehensive review on the applications of neural operators, the interested reader can refer to the recent work in [33].

Here, we present *Random Projection-based Operator Networks* (RandONets) to deal with the "curse of dimensionality" in the training process of DNNs schemes such as DeepONets [10] and facilitate interpretability. DeepONets, while being a powerful methodology that revolutionized scientific machine learning for dynamical systems, are not without their limitations. Their training often involves iterating over large datasets multiple times to update the high-dimensional space of the deep learning network parameters, requiring significant computational time and memory. Additionally, the complexity of the underlying nonlinear operators and the size of the problem domain can further increase the computational burden. Moreover, hyperparameter tuning, regularization techniques, and model selection procedures contribute to additional computational overhead. As a result, training DeepONets can require substantial computational resources, including high-performance computing clusters or GPUs, while due to their deep architecture are not interpretable. Importantly, the computational demands of training DNNs can significantly impact convergence behavior and numerical approximation accuracy. The high-dimensional parameter space may lead to challenges in converging to a (near) global optimum. In some cases, the optimization algorithm may get stuck in local minima or plateaus, hindering the network's ability to approximate the underlying nonlinear or even linear, as we will show, operators with a high accuracy. Addressing these challenges requires careful consideration of optimization strategies, regularization methods and dataset size, balancing computational efficiency with the desired level of approximation accuracy (see also critical discussions and approaches to deal with this cost-accuracy tradeoff in [34–37]).

Our proposed RandONets deal with these challenges, leveraging shallow (just one hidden layer) feedforward neural networks, random projections [38–42,17,6] and numerical analysis methods to enable a computationally efficient framework for the approximation of linear and nonlinear operators, thus enhancing interpretability. In particular, we integrate established numerical analysis methods, such as the Tikhonov regularization, and pivoted QR decomposition with regularization, for the solution of an ill-posed linear problem. These methods offer highly efficient numerical approximations with guaranteed (near) optimal convergence properties. Importantly, based on previous works, we prove that RandONets are universal approximators of linear and nonlinear operators. Furthermore, we assess their performance with a focus on the approximation of linear and nonlinear evolution operators (right-hand-sides (RHS)) of PDEs. We demonstrate that, RandONets outperform the vanilla DeepONets both in terms of numerical approximation accuracy (reaching for linear operators machine precision) and computational cost by orders of magnitude. In summary, our work shows that carefully designed "light" neural networks, aided by tailor-made numerical methods, can provide significantly faster and more accurate approximations of nonlinear operators compared to DNNs, while also enhancing interpretability.

The paper is organized as follows. In Section 2, we describe the problem. In Sections 3.1 and 3.2, we present the preliminaries regarding the fundamentals and necessary notation for DeepONets and the random projection neural networks (RPNNs) approaches, respectively. In section 3.3 we introduce RandONets, and then, in Section 3.3.1, we extend the theorem of Chen and Chen [27] on the universal approximation of Operator to RandONets architectures. In Section 4, we assess the performance of RandONets and various linear and nonlinear benchmark problems and compare its performance with the vanilla DeepONet. We start with some simple problems of ODEs, where we approximate the solution operator, and then we proceed with the presentation of the results on the approximation of the evolution operator of PDEs. We conclude, thus giving future perspectives, in Section 5.

2. Description of the problem

In this study, we focus on the challenging task of learning linear and nonlinear functional operators $\mathcal{F} : \mathsf{U} \to \mathsf{V}$ which constitute maps between two infinite-dimensional function spaces U and V. Here, for simplicity, we consider both U and V to be subsets of the set $\mathsf{C}(\mathbb{R}^d)$ of continuous functions on $\mathbb{R}^d$. The elements of the set U are functions $u : \mathsf{X} \subseteq \mathbb{R}^d \to \mathbb{R}$ that are transformed to other functions $v = \mathcal{F}[u] : \mathsf{Y} \subseteq \mathbb{R}^d \to \mathbb{R}$ through the application of the operator $\mathcal{F}$. We use the following notation for an operator evaluated at a location $\boldsymbol{y} \in \mathsf{Y} \subseteq \mathbb{R}^d$

$$v(\boldsymbol{y}) = \mathcal{F}[u](\boldsymbol{y}). \tag{1}$$

These operators play a pivotal role in various scientific and engineering applications, particularly in the context of (partial) differential equations. By effectively learning (discovering from data) such nonlinear operators, we seek to enhance our understanding and predictive capabilities in diverse fields, ranging from fluid dynamics and materials science to financial and biological systems and beyond [43,10,9,34,29,17,44,19,45,18]. One prominent example is the right-hand side (RHS) evolution operators $\mathcal{L}$ associated with differential equations (PDEs), which govern the temporal evolution of the associated system dynamics. We denote these *evolution* operators in the following way:

$$v(\boldsymbol{x}, t) = \frac{\partial u(\boldsymbol{x}, t)}{\partial t} = \mathcal{L}[u](\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \Omega, \quad t \in [0, T], \tag{2}$$

where $u : \Omega \times [0, T] \subseteq \mathbb{R}^d \times \mathbb{R}_+ \to \mathbb{R}$ is the unknown solution of the PDE (methods for the identification of such PDEs and in general RHS of dynamical systems with ML can be traced back to the '90s [22,46,47,23,48]). Given a state profile $u(\cdot, t) : \Omega \to \mathbb{R}$ at each time t, e.g., the initial condition u_0 of the system at time $t = 0$, the *evolution operator* (the right-hand-side) of differential equations) $\mathcal{L}$ provides the corresponding time derivative (the output $v(\cdot, t)$) of the system at that time t. Again, a method for learning the RHS of PDEs with a different ANN architecture than DeepONet was proposed back in '90s in [21]. There, the RHS was estimated in terms of spatial derivatives.

One can also learn the corresponding *solution operators* S_t, which embody both the time integration and the satisfaction of boundary conditions, of the underlying physical phenomena. Given the initial condition u_0 at time $t = 0$, the solution operator outputs the state profile $u(\cdot, t) : \Omega \to \mathbb{R}$ after a certain amount of time t:

$$v(\boldsymbol{x}) = u(\boldsymbol{x}, t) = S_t[u_0](\boldsymbol{x}). \tag{3}$$

We will deal with this problem in the part II that will follow.

Although our objective is to learn functional operators from data, which take functions (u) as input, we must discretize them to effectively represent them and be able to apply network approximations. One practical approach, as implemented in the DeepONet framework, is to use the function values ($u(\boldsymbol{x}_j,\ j = 1, \cdots, m)$) at a sufficient, but finite, number of locations $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m$, where $\boldsymbol{x}_j \in \mathsf{X} \subseteq \mathbb{R}^d$; these locations are referred to as "sensors." Other methods to represent functions in functional spaces include the use of Fourier coefficients [24], wavelets [31], spectral Chebychev basis [32], reproducing kernel Hilbert spaces (RKHS) [49], graph neural operators [26] or meshless representations [50]. Regarding the availability of data for the output function, we encounter two scenarios. In the first scenario, the functions in the output are known at the same fixed grid $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n$, where $\boldsymbol{y}_i \in Y$; this case is termed as "aligned" data. Conversely, there are cases where the output grid may vary randomly for each input function, known as "unaligned" data. If this grid is uniformly sampled and dense enough, interpolation can be used to approximate the output function at fixed locations. Thus, this leads us back to the aligned data case. However, if the output is only available at sparse locations, interpolation becomes impractical. As we will see later in the text, despite this challenge, our approach can address this scenario, albeit with a higher computational cost for training the machine learning model (since, in such cases, the fixed structure of the data cannot be fully leveraged).

3. Methods

For the completeness of the presentation, we start with some preliminaries on the use of neural networks and DeepONets for the approximation of nonlinear continuous functional operators. We then introduce the concepts of random projections for neural networks (RPNN) and extend them to the DeepONet framework, arriving at the proposed Random Projection-based Operator networks (RandONets).

3.1. Preliminaries on DeepONets

As universal approximators, feedforward neural networks (FNN) have the capability to approximate continuous functions effectively [51–55]. However, a lesser-known theorem by Chen & Chen (1995) [27], which gained prominence with the advent of DeepONet by Lu et al. (2021) [10] and Fourier Neural Operator (FNO) by Li et al. (2020) [24], asserts the existence of a neural network architecture capable of approximating any continuous nonlinear operator to an arbitrary degree of accuracy. Before introducing this theorem, let us introduce the following definition:

Definition 3.1 *(Tauber-Wiener function [27]).* A function $\psi : \mathbb{R} \to \mathbb{R}$ is called a Tauber-Wiener function if, for any interval $[a, b] \subset \mathbb{R}$, the set of finite linear combinations

$$\left\{ \sum_{i=1}^{N} w_i \psi(\xi_i x + \theta_i) \mid N \in \mathbb{N}, \xi_i \in \mathbb{R}, \theta_i \in \mathbb{R}, w_i \in \mathbb{R} \right\}$$

is dense in $C[a, b]$.

Theorem 3.1 *(Universal approximation for functions [27]). Suppose K is a compact set in $\mathbb{R}^d$, U is a compact set in $\mathsf{C}(K)$ and ψ is a Tauber-Wiener function, then $\forall f \in \mathsf{U}$ and any $\epsilon > 0$, there exist scaling factors $\{\xi_i\}_{i=1}^N$ and shifts $\{\theta_i\}_{i=1}^N$ both independent of f, and also coefficients $\{w_i[f]\}_{i=1}^N$ depending on f, such that*

$$\left\| f(x) - \sum_{i=1}^{N} w_i[f] \psi(\xi_i x + \theta_i) \right\|_{\infty} < \epsilon. \tag{4}$$

Moreover, the coefficients $w_i[f]$, $i = 1, \ldots, N$, are continuous functionals on U.

In other words, any function in $\mathsf{C}(K)$ can be approximated arbitrarily closely by a finite linear combination of scaled and shifted versions of ψ. It has been demonstrated that continuous non-polynomial functions are Tauber-Wiener functions [54,27]. Then, the following theorem holds:

Theorem 3.2 *(Universal approximation for operators [27]). Suppose that ψ is a Tauber-Wiener function, X is a Banach space, and $\mathsf{K}_1 \subset \mathsf{X}$, $\mathsf{K}_2 \subset \mathbb{R}^d$ are two corresponding compact sets. Let U be a compact set in $\mathsf{C}(\mathsf{K}_1)$, and let $\mathcal{F} : \mathsf{U} \to \mathsf{C}(\mathsf{K}_2)$ be a nonlinear continuous operator. Then, for any $\epsilon > 0$, there are $N, M, m \in \mathbb{N}$, and network parameters w_{ki}, ξ_{kij}, θ_{ki}, $\beta_k \in \mathbb{R}$, $\boldsymbol{c}_k \in \mathbb{R}^d$, $\boldsymbol{x}_j \in \mathsf{K}_1$, with $k = 1, \ldots, N$, $i = 1, \ldots, M$, $j = 1, \ldots, m$, such that $\forall u \in U, \boldsymbol{y} \in \mathsf{K}_2$:*

$$\left| \mathcal{F}(u)(\boldsymbol{y}) - \sum_{k=1}^{N} \sum_{i=1}^{M} w_{ki} \psi\Big(\sum_{j=1}^{m} \xi_{kij} u(\boldsymbol{x}_j) + \theta_{ki} \Big) \cdot \psi(\boldsymbol{c}_k \cdot \boldsymbol{y} + \beta_k) \right| < \epsilon. \tag{5}$$

To briefly describe how the above Theorem in the original paper of Chen & Chen in [27] works, let us assume that our goal is to approximate an operator $\mathcal{F}$, acting on functions $u \in \mathsf{U}$. These functions u (which are inputs to the DeepONet) are assumed to be known and sampled at m fixed locations x_j in the domain K_1. The vector $\boldsymbol{U} = (u(\boldsymbol{x}_1), u(\boldsymbol{x}_2), \ldots, u(\boldsymbol{x}_m)) \in \mathbb{R}^{m \times 1}$ (a column vector) is the input of a single-hidden layer FNN with M neurons, the so-called *branch network*, that process the function values space. At the same time there is a second single-hidden layer FNN with N neurons, the so-called *trunk network*, that process the new location $\boldsymbol{y} \in K_2 \subset \mathbb{R}^{1 \times d}$ (for convenience let us assume it as a row vector) in which we have to evaluate the transformed function $\mathcal{F}[u]$. For convenience, let us define the vector $\boldsymbol{B} = (B_1, B_2, \ldots, B_M) \in \mathbb{R}^{M \times 1}$ (column vector) of hidden layers value of the branch network:

$$B_i(\boldsymbol{U}) = \psi\Big(\sum_{j=1}^{m} \xi_{kij} u(\boldsymbol{x}_j) + \theta_{ki} \Big), i = 1, 2, \ldots, M, \tag{6}$$

and let us define the vector $\boldsymbol{T} = (T_1, T_2, \ldots, T_N) \in \mathbb{R}^{1 \times N}$ (row vector):

$$T_k(\boldsymbol{y}) = \psi({\boldsymbol{c}_k}^T \cdot \boldsymbol{y} + \beta_k), k = 1, 2, \ldots, N. \tag{7}$$

Then, the output of the network as in Eq. (5) can be written as:

$$\mathcal{F}[u](\boldsymbol{y}) \simeq \sum_{k=1}^{N} \sum_{i=1}^{M} w_{ki} B_i(\boldsymbol{U}) T_k(\boldsymbol{y}) \Leftrightarrow \mathcal{F}[u](y) = \boldsymbol{T} W \boldsymbol{B} = \langle \boldsymbol{T}, \boldsymbol{B} \rangle_W, \tag{8}$$

where the matrix $W \in \mathbb{R}^{N \times M}$ has elements w_{ki}. As can be seen, the output of the scheme is a weighted inner product $\langle \cdot, \cdot \rangle_W$ of the trunk and branch networks. In the next section, we will take advantage of this formulation for an efficient and accurate training of the network through the use of random bases.

We note, that the original Theorem 3.2 considers only two shallow feedforward neural networks with a single hidden layer. On the other hand, DeepONet uses deep networks instead, but also can incorporate any other type of networks such as CNNs. An extension of the Theorem 3.2, given by Lu et al. [10], states that the branch network and the trunk network can be chosen by diverse classes of ANNs, which satisfy the classical universal approximation theorem. Also, while the Chen and Chen architecture in (5) does not include an output bias, the DeepONet usually utilize biases to improve generalization performance [10]. More broadly, DeepONets can be considered conditional models, where $\mathcal{F}[u](\boldsymbol{y})$ represents a function of $\boldsymbol{y}$ given u. These two independent inputs, u and $\boldsymbol{y}$, are given as inputs to the trunk and branch networks, respectively. At the end, the embeddings of u and $\boldsymbol{y}$ are combined through an inner product operation. However, the challenge remains in finding efficient approaches to train these networks. DeepONets do not come without limitations. While they enhance the models' ability to capture complex relationships, they introduce challenges in the optimization process. It is also worth noting that, as it happens for shallow FNNs, while the universal approximation theorem for operators ensures the existence of an effective approximating DeepONet, it does not offer a constructive numerical method for the computation of the specific weights and biases of the DNNs. Furthermore, computing the values of the networks *parameters and hyperparameters* requires significant computational resources, entailing complexity that can lead to moderate generalization ability and/or numerical

approximation accuracy. Hence, it is nearly implicit that training such DeepONets heavily relies on parallel computing and GPU-based computations.

Here we present a computationally efficient method for approximating nonlinear operators, based on shallow networks with a single hidden layer, as in the paper of Chen and Chen in [27], coupled with random projections, that relaxes the "curse of dimensionality" in the training process. First, we give some preliminaries for random projection neural networks, and then we proceed with the presentation of the RandONets and building on previous works, we prove its universal approximation property for linear and nonlinear operators.

3.2. Preliminaries on random projection neural networks

Random Projection Neural Networks (RPNN) are a type of single-hidden-layer ANNs with randomized activation functions to facilitate the training process. The family of RPNNs includes random weights neural networks (RWNN) [56], Random Vector Functional Link Network (RVFLN) [57,76,58], Reservoir Computing (RC) [59,60], Extreme Learning Machines (ELM) [61] and Random Fourier Features Networks (RFFN) [39,41]. Some seeds of this idea can be also found in *gamba perceptron* proposed initially by Frank Rosenblatt (1962) [62] and the Distributed Method algorithm proposed by Gallant (1987) [63]. For a review on random projection neural networks see in [64,17,6].

Here we consider for simplicity, and refer with the acronym RP-FNN to, a single hidden layer feed-forward neural network, denoted by a vector function $\boldsymbol{f} \equiv (f_1, f_2, \dots, f_n) : \mathbb{R}^d \to \mathbb{R}^n$, with n outputs f_k, N neurons and with an activation function $\psi : \mathbb{R} \to \mathbb{R}$. To simplify our notation, we consider here each scalar output $y_k = f_k(\boldsymbol{x})$ of the RP-FNN:

$$f_k(\boldsymbol{x}; W, \boldsymbol{\beta}, C, \boldsymbol{P}) = \sum_{j=1}^{N} w_{kj} \psi(\boldsymbol{c}_j \cdot \boldsymbol{x} + \beta_j; \boldsymbol{P}), \qquad k = 1, \dots, n, \tag{9}$$

In RP-FNNs, the weights W are the only trainable parameters of the network. While the internal weights/parameters and hyperparameters of ψ_j are randomly pre-determined and fixed. In order to simplify the notation, let us group the set of parameters and hyperparameters in the vector of random variables $\boldsymbol{\alpha}$ over the set $\mathsf{A} \subseteq \mathbb{R}^q$, containing the stacking of all parameters in $\{C, \boldsymbol{\beta}, \boldsymbol{P}\}$. The vector of random variables $\boldsymbol{\alpha}$ is in general sampled from a probability distribution function $p_{\boldsymbol{\alpha}}$ on A. Thus, we can rewrite Eq. (9) as: where the matrix $C \in \mathbb{R}^{N \times d}$, with rows $\boldsymbol{c}_j \in \mathbb{R}^{1 \times d}$, contains *a priori* randomly-fixed *internal weights*, sampled appropriately from a probability distribution p_c, connecting the input layer with the hidden layer (in other configurations, they can also be set all to ones, see e.g. [6]); the vector $\boldsymbol{\beta} = (\beta_1, \dots, \beta_N) \in \mathbb{R}^{N \times 1}$ includes *a priori* randomly-fixed vector of *biases* (shifts), sampled appropriately from a probability p_β; the vector $\boldsymbol{x} \in \mathbb{R}^{d \times 1}$ represents the input, the matrix $W \in \mathbb{R}^{n \times N}$, with elements w_{kj}, contains the *external* weights that connect the hidden layer to the output; the vector $\boldsymbol{P}$ includes any additional required hyperparameters (either deterministically or randomly fixed) for the activation function, for example the shape parameters of Gaussian Radial Basis functions.

$$f_k(\boldsymbol{x}; \boldsymbol{\alpha}) = \sum_{j=1}^{N} w_{kj} \psi(\boldsymbol{x}; \boldsymbol{\alpha}_j), \tag{10}$$

where $\boldsymbol{\alpha}_j \in \mathsf{A}$ are N realizations of the random variables $\boldsymbol{\alpha}$.

Having $\{\boldsymbol{x}_i, f_k(\boldsymbol{x}_i)\}$, $i = 1, 2, \dots, m$ pairs of training data for each f_k, the unknown parameters $\boldsymbol{w}_k \in \mathbb{R}^N$, which are the rows of the matrix W, can be computed for example using truncated SVD, preconditioned QR decomposition with regularization, and/or Tikhonov regularization. For example, with truncated SVD, the least-squares solution reads:

$$\boldsymbol{w}_k = \sum_{i=1}^{k} \frac{{u_i}^T Y_k}{\sigma_i} v_i, \tag{11}$$

with u_i, v_i being the first k right and left singular vectors of Ψ is the $N \times m$ matrix with entries $\Psi_{ji} = \psi(\boldsymbol{x}_i; \boldsymbol{\alpha}_j)$, and σ_i, the corresponding singular values. On the other hand, the Tikhonov regularization reads:

$$\boldsymbol{w}_k = \arg\min_{\boldsymbol{w}_k} \left\{ \|\boldsymbol{w}_k \Psi - Y_k\|^2 + \lambda \|\boldsymbol{w}_k L\|^2 \right\}, \tag{12}$$

where Y_k is the vector of dimension m, containing the values (samples) of f_k at m sampling points $\boldsymbol{x}_i$, $\lambda > 0$ is the regularization parameter and $L \in \mathbb{R}^{N \times N}$ is a regularization operator, often taken as the identity matrix I. The Tikhonov regularized solution can be expressed as:

$$\boldsymbol{w}_k = Y_k \Psi^T (\Psi \Psi^T + \lambda L L^T)^{-1}. \tag{13}$$

Setting, $L = I$, the above problem can be solved, e.g., by substituting the truncated SVD of Ψ into the Tikhonov regularization formula to get: [66]:

$$\boldsymbol{w}_k = \sum_{i=1}^{r} \frac{{\sigma_i}^2}{{\sigma_i}^2 + \lambda^2} \frac{{u_i}^T Y_k}{\sigma_i} v_i, \tag{14}$$

where r is the rank of Ψ. Let us now define the *hidden layer map* $\boldsymbol{\phi}_N : \mathbb{R}^d \times \mathsf{A} \to \mathbb{R}^N$, that maps the input layer to the (random) features $\boldsymbol{z}$ of the hidden layer, as:

$$\boldsymbol{z} = \boldsymbol{\phi}_N(\boldsymbol{x};\boldsymbol{\alpha}) = [\psi(\boldsymbol{c}_1 \cdot \boldsymbol{x} + \beta_1; P), \psi(\boldsymbol{c}_2 \cdot \boldsymbol{x} + \beta_2; P), \dots, \psi(\boldsymbol{c}_N \cdot \boldsymbol{x} + \beta_N; P)]. \tag{15}$$

In its simplest form (taking just a linear projection of the input space), the above is – conceptually equivalent – to the celebrated Johnson-Lindenstrauss (JL) lemma [38], which states that there exists an approximate isometry projection $\boldsymbol{\phi}_N : \mathbb{R}^d \to \mathbb{R}^N$ of input data $\boldsymbol{x} \in \mathbb{R}^d$ induced by a random matrix $R \in \mathbb{R}^{N\times d}$:

$$\boldsymbol{z}^{JL} = \boldsymbol{\phi}_N^{JL}(\boldsymbol{x}; R) = \frac{1}{\sqrt{N}} R\,\boldsymbol{x}, \tag{16}$$

where the matrix $R = [R_{ji}] \in \mathbb{R}^{N\times d}$ has components which are i.i.d. random variables sampled from a standard normal distribution. Let us assume X to be a set of m sample points $\boldsymbol{x} \in \mathbb{R}^d$, such that $N \geq O\big(\log(m)/\epsilon^2\big)$. Then with probability $\mathbb{P}$, for every $\epsilon \in (0,1)$, we obtain:

$$\mathbb{P}\left(\left|\|\boldsymbol{x}\|_2 - \|\boldsymbol{z}^{JL}\|_2\right| \leq \epsilon\|\boldsymbol{x}\|_2\right) \geq 1 - 2\exp\left(-(\epsilon^2 - \epsilon^3)\frac{N}{4}\right). \tag{17}$$

Let us consider a simple regression problem, with training data $(\boldsymbol{x}^{(s)}, \boldsymbol{y}^{(s)}) \in \mathbb{R}^d \times \mathbb{R}^n$, $s = 1, \dots, m$. Let us call the matrix $X \in \mathbb{R}^{d\times m}$ the collection of inputs, and the matrix $Y \in \mathbb{R}^{n\times m}$ the collection of outputs. A simple approach can consist in considering linear random JL projections of the input, and then approximating the output as a weighted linear combination of random JL features. Thus, one finds $W \in \mathbb{R}^{n\times N}$ such that:

$$Y = \frac{1}{\sqrt{N}} W R X. \tag{18}$$

At this stage, one can solve for the unknown parameters in W using a linear regularization problem as briefly described above. When considering nonlinear projections, the training of an RP-FNN, involves solving a system of $n \times m$ linear algebraic equations, with $n \times N$ unknowns:

$$W\Phi_N(X;\boldsymbol{\alpha}) = Y, \qquad \Phi_{ji} = \psi(\boldsymbol{x}_i;\boldsymbol{\alpha}_j), \tag{19}$$

where $\Phi_N(X) \in \mathbb{R}^{N\times m}$ is the random matrix of the hidden layer features, with elements Φ_{ji}. Note that despite the nonlinearity of ψ, the problem is still linear with respect to the external weights W.

While JL linear random projections are appealing due to their simplicity, studies have highlighted that well-designed *nonlinear* random projections can outperform such linear embeddings [53,58,42,65]. In this context, back in '90s Barron [53] proved that for functions with integrable Fourier transformations, a random sample of the parameters of sigmoidal basis functions from an appropriately chosen distribution results to an approximation error on the order of $O(1/(n^{(2/d)}))$. Igelnik and Pao [58] extended Barron's proof [53] for any family of L^2 integrable basis functions, thus addressing the so-called RVFLNs [57]. Later on, the works of Rahimi and Recht [39,41] have explored the effectiveness of nonlinear random bases for preserving any shift-invariant kernel distances. It is also worth mentioning that the "kernel trick" [67,68], a common feature approach in machine learning, including Support Vector Machines (SVMs) and Gaussian Processes (GPs), provides a straightforward method to generate features for algorithms that rely solely on the inner product between pairs of input points:

$$\langle \tilde{\boldsymbol{\phi}}(\boldsymbol{u}), \tilde{\boldsymbol{\phi}}(\boldsymbol{v}) \rangle = K(\boldsymbol{u},\boldsymbol{v}), \tag{20}$$

where $\tilde{\boldsymbol{\phi}}$ represents a generic implicit lifting and K is a kernel distance function. This technique is commonly employed to effectively handle high-dimensional data without explicitly computing the feature vectors $\tilde{\boldsymbol{\phi}}(\boldsymbol{u})$ and $\tilde{\boldsymbol{\phi}}(\boldsymbol{v})$.

However, large training sets often lead to significant computational and storage costs. Instead of depending on the implicit lifting provided by the kernel trick, we seek an *explicit* mapping of the data to a low-dimensional Euclidean inner product space, using a nonlinear randomized (randomly parametrized) feature map $\boldsymbol{\phi}_N(\cdot,\boldsymbol{\alpha}) : \mathbb{R}^d \times \mathsf{A} \to \mathbb{R}^N$:

$$K(\boldsymbol{u},\boldsymbol{v}) \approx \boldsymbol{\phi}_N(u;\boldsymbol{\alpha})^T \boldsymbol{\phi}_N(v;\boldsymbol{\alpha}), \tag{21}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^q$ is a set of hyperparameters (random variables) sampled from a probability distribution $p_{\boldsymbol{\alpha}}$.

Here, for the sake of completeness of the presentation, we restate the following theorem [39],

Theorem 3.3 *(Low-distortion of kernel-embedding [39]). Let K be a positive definite shift-invariant kernel $K(\boldsymbol{u},\boldsymbol{v}) = K(\boldsymbol{u}-\boldsymbol{v})$. Consider the Fourier transform $p_{K,\boldsymbol{\alpha}} = \hat{\mathcal{F}}[K]$ of the kernel K, resulting a probability density function (pdf) $p_{K,\boldsymbol{\alpha}}$ in the frequency space A: $p_{K,\boldsymbol{\alpha}}(\alpha) = \frac{1}{2\pi}\int e^{j\alpha\Delta} K(\Delta) d\Delta$, and draw N i.i.d. samples weights $\boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_N \in \mathbb{R}^d$ from $p_{K,\boldsymbol{\alpha}}$. Define*

$$\boldsymbol{\phi}_N(\boldsymbol{u};\boldsymbol{\alpha}) \equiv \sqrt{\frac{1}{N}}[\cos(\boldsymbol{\alpha}_1^T\boldsymbol{u}), \dots, \cos(\boldsymbol{\alpha}_n^T\boldsymbol{u}), \sin(\boldsymbol{\alpha}_1^T\boldsymbol{u}), \dots, \sin(\boldsymbol{\alpha}_n^T\boldsymbol{u})]. \tag{22}$$

Then, $\forall \epsilon > 0$

$$\mathbb{P}\left[(1-\epsilon)K(\boldsymbol{u},\boldsymbol{v}) \le \boldsymbol{\phi}_N(\boldsymbol{u})^T\boldsymbol{\phi}_N(\boldsymbol{v}) \le (1+\epsilon)K(\boldsymbol{u},\boldsymbol{v})\right] \ge 1 - O\left(\exp\left(-\frac{N\epsilon^2}{4(d+2)}\right)\right), \tag{23}$$

where $\mathbb{P}$ stands for the probability function.

The above approach for the kernel approximation, employing trigonometric activation functions, is also known as Random Fourier Features (RFFN). An equivalent result can be obtained by employing only cosine as the activation function and random biases in $[0, 2\pi]$ [39]. More generally, there is no constraint in considering trigonometric activation functions, as sigmoid and radial basis functions have equivalently shown remarkable results [4,6,65,7,8,20,13]. Here we restate, the following theorem [41,40]:

Theorem 3.4. *(cf. Theorem 3.1 and 3.2 in [40])) Consider the parametric set activation functions on $\mathsf{X} \subseteq \mathbb{R}^d$, $\psi(\boldsymbol{x};\boldsymbol{\alpha}) : \mathsf{X}\times\mathsf{A} \to \mathbb{R}$ parametrized by random variables $\boldsymbol{\alpha}$ in A, that satisfy $\sup_{\boldsymbol{x},\boldsymbol{\alpha}} |\phi(\boldsymbol{x},\boldsymbol{\alpha})| \le 1$. Let p be a probability distribution on A and μ be a probability measure on X and the corresponding norm $\|f\|_{L^2(\mu)} = \int_{\mathsf{X}} f(\boldsymbol{x})^2\mu(d\boldsymbol{x})$. Define the set:*

$$\mathsf{G}_p \equiv \left\{ g(\boldsymbol{x}) = \int_{\mathsf{A}} w(\boldsymbol{\alpha})\phi(\boldsymbol{x};\boldsymbol{\alpha})d\boldsymbol{\alpha} \,:\, \|g\|_{p(\boldsymbol{\alpha})} < \infty \right\},\ \|g\|_p := \sup_{\boldsymbol{\alpha}\in\mathsf{A}} \|w(\boldsymbol{\alpha})/p(\boldsymbol{\alpha})\|. \tag{24}$$

Fix a function g^* in G_p. Then, for any $\delta > 0$, there exist $N \in \mathbb{N}$, and $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \dots, \boldsymbol{\alpha}_N$ of $\boldsymbol{\alpha}$ drawn i.i.d. from p, and a function $\hat{g}$ in the random set of finite sums

$$\hat{\mathsf{G}}_{\boldsymbol{\alpha}} \equiv \left\{ \hat{g} \,:\, \hat{g}(\boldsymbol{x}) = \sum_{j=1}^{N} w_j\phi(\boldsymbol{x};\boldsymbol{\alpha}_j) \right\} \tag{25}$$

such that

$$\sqrt{\int_{\mathsf{X}} (g^*(\boldsymbol{x}) - \hat{g}(\boldsymbol{x}))^2 d\boldsymbol{\mu}(\boldsymbol{x})} \le \frac{||g^*||_p}{\sqrt{N}}\left(1 + \sqrt{2\log\frac{1}{\delta}}\right), \tag{26}$$

holds with probability at least $1-\delta$. Moreover, if $\phi(\boldsymbol{x};\boldsymbol{\alpha}) = \varphi(\boldsymbol{\alpha}\cdot\boldsymbol{x})$, for a L-Lipschitz function ψ, the above approximation is uniform (i.e. in the supremum norm).

The above Theorem implies that the function class G_p can be approximated to any accuracy when $N \to \infty$. Moreover (see [40]) this class of functions is dense in Reproducing Kernel Hilbert Spaces defined by ϕ and p. For a detailed discussion on the pros and cons of function approximation with such random bases see [42]. Finally, very recently, Fabiani (2024) [65] have theoretically proved the existence and uniqueness of RP-FNN of the best approximation and their exponential convergence rate when approximating low-dimensional infinitely differentiable functions. These theoretical results are also numerically validated through extensive benchmarks in [65]. This showcases a concrete possibility of bridging the gap between theory and practice in ANN-based approximation [69,65].

3.3. Random projection-based operator networks (RandONets)

In this section, we present RandONets for approximating nonlinear operators. Building on previous works, we first demonstrate that the proposed shallow–single hidden layer–random projection neural networks are universal approximators of non-linear operators. Then we discuss how RandONets can be used in both the aligned and unaligned data cases, and finally present their numerical implementation for the solution of the inverse problem: the approximation of the differential operator, i.e., the RHS of the differential equations as well as their solution operator.

3.3.1. RandONets as universal approximators of nonlinear operators

In this section, we prove that RandONets are universal approximators of nonlinear operators. Following the methodological thread in [27], we first state the following proposition:

Proposition 1. *Let $\mathsf{K} \subset \mathbb{R}^d$ compact and $\mathsf{U} \subset \mathsf{C}(\mathsf{K})$ compact and consider a parametric family of random activation functions $\{\psi(\boldsymbol{x};\boldsymbol{\alpha}) : \boldsymbol{x}\in\mathbb{R}^d,\ \boldsymbol{\alpha}\in\mathsf{A}\}$, where $\boldsymbol{\alpha}\in\mathsf{A}$ is a vector of randomly chosen (hyper) parameters, and assume that ψ are uniformly bounded in $\mathbb{R}^d\times\mathsf{A}$. Let p be a probability distribution on A. Given any ϵ, there exists a $N\in\mathbb{N}$ and i.i.d. sample $\boldsymbol{\alpha}_1,\cdots,\boldsymbol{\alpha}_N$ from p, chosen independently of f, such that for every $f\in\mathsf{U}$ the random approximation*

$$f_\epsilon(\boldsymbol{x}) = \sum_{j=1}^{N} c_j[f]\psi(\boldsymbol{x};\boldsymbol{\alpha}_j), \tag{27}$$

approximates f in the sense that with high probability

$$\|f - f_\epsilon\|_{L^2(\mu)} < \epsilon, \tag{28}$$

for a suitable probability measure μ over K. Moreover, if $\psi(\boldsymbol{x};\boldsymbol{\alpha}) = \varphi(\boldsymbol{\alpha}\cdot\boldsymbol{x})$, for a L-Lipschitz function ψ, the above approximation is uniform (i.e. in the supremum norm).

Proof. We only show the uniform approximation result. The $L^2(\mu)$ approximation follows similar arguments.

We apply Theorem 3 in [27]. Given a continuous sigmoidal (non-polynomial) function σ, for every $\epsilon > 0$, there exist $N \in \mathbb{N}$, $(\hat{\theta}_i, \boldsymbol{\omega}_i) \in \mathbb{R} \times \mathbb{R}^d$, $i = 1, \cdots, N$ such that for every $f \in \mathsf{U} \subset \mathsf{C}(\mathsf{K})$, there exists a linear continuous functional on U, $f \mapsto c_i[f]$, with the property that

$$|f(\boldsymbol{x}) - \sum_{i=1}^{N} c_i[f]\sigma(\boldsymbol{\omega}_i \cdot \boldsymbol{x} + \hat{\theta}_i)| < \epsilon, \quad \forall \boldsymbol{x} \in K. \tag{29}$$

This approximation is deterministic. Note that even though according to Theorem 3 [27] this approximation holds for any Tauber-Wiener function (i.e. even for non-continuous σ) here we must insist on the continuity of σ.

In the above, we obtain a uniform approximation f_ϵ to f in terms of

$$f_\epsilon(\boldsymbol{x}) = \sum_{i=1}^{N} c_i[f]\sigma(\boldsymbol{\omega}_i \cdot \boldsymbol{x} + \hat{\theta}_i) \tag{30}$$

We emphasize that σ is chosen to be a continuous and bounded sigmoidal function. Note that the choice of N, (θ_i, ω_i), are independent of f, while the coefficients $c_i[f]$ depend on f and in fact $f \mapsto c_i[f]$ is a linear continuous functional on U.

To connect with the random features approach of Rahimi and Rechts we first employ an expansion of each sigmoid in (29) in terms of a radial basis function (RBF) (which is eligible to a random features expansion a la Rahimi and Rechts) and then follow with the expansion of the RBFs in random features. We follow the subsequent steps:

(a) For each $i = 1, \cdots, N$, we consider the function ϕ_i, defined by $\mathsf{K} \ni \boldsymbol{x} \mapsto \phi_i(\boldsymbol{x}; \boldsymbol{\omega}_i, \hat{\theta}_i) := \sigma(\boldsymbol{\omega}_i \cdot \boldsymbol{x} + \hat{\theta}_i)$. By the properties of σ, the functions $\phi_i \in \mathsf{C}(\mathsf{K})$. Hence, we may apply the approximation of each ϕ_i in terms of an RBF neural network. Using standard results (e.g. Theorem 3 [70]) we have that if $g \in \mathsf{C}(\mathbb{R}^d)$ is a bounded radial basis function $S := \mathrm{span}\{g(a\boldsymbol{x} + \boldsymbol{b}) \ : \ a \in \mathbb{R}, \ \boldsymbol{b} \in \mathbb{R}^d\}$ is dense in $\mathsf{C}(\mathsf{K})$. Note that without loss of generality we may impose the extra assumption that g can be expressed in terms of the inverse Fourier transform of some function (i.e. an element of a function space on which the Fourier transform is surjective, for example, g belongs in the Schwartz space). This assumption also allows us to invoke the standard results of [71], [72] leading to the same density result). Note that this step does not affect the generality of our results, as it is only used in the intermediate step (a) which re-expands the general sigmoids used in (29) into a more convenient basis on which the step (b) is applicable. Moreover, the choice of g is not unique.

Using the above result, we can approximate each ϕ_i in terms of the functions

$$\phi_{i,\epsilon}(\boldsymbol{x}) = \sum_{j=1}^{M_i} w_{ij} g(a_{ij}\boldsymbol{x} + \boldsymbol{b}_{ij}), \tag{31}$$

where importantly, the $(w_{ij}, a_{ij}, \boldsymbol{b}_{ij}) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d$ are independent of the choice of the function f (as they only depend on $(\boldsymbol{\omega}_i, \hat{\theta}_i) \in \mathbb{R}^d \times \mathbb{R}$, which are independent of f – see (29)). The function $\phi_{i,\epsilon}$ satisfies the property $\|\phi_i - \psi_{i,\epsilon}\| < \frac{\epsilon}{N}$, in the uniform norm $\forall \epsilon > 0$. Combining (29) (and (30)) with (31), we obtain an approximation f_ϵ^{RBF} to f_ϵ in terms of

$$f_\epsilon^{RBF}(\boldsymbol{x}) = \sum_{i=1}^{N} \sum_{j=1}^{M_i} c_i[f] w_{ij} g(a_{ij}\boldsymbol{x} + \boldsymbol{b}_{ij}), \tag{32}$$

such that $\|f_\epsilon - f_\epsilon^{RBF}\| < \epsilon$, hence satisfying by (29) that $\|f - f_\epsilon^{RBF}\| < 2\epsilon$ (in the uniform norm).

(b) Now, by appropriate choice of the RBF function g we apply Rahimi and Recht for a further expansion of each term $g(a_{ij}\boldsymbol{x} + \boldsymbol{b}_{ij})$, using the random features RPNN. By the choice of g as above, this is now possible, and the results of Rahimi and Recht [40] (Theorem 3.4), are now applicable to g. This holds, since RBFs can be elements of the RKHS that the Rahimi and Recht framework applies to, i.e., they belong to the space of functions G_p, defined in (24). For such a choice Theorem 3.1 in [40] can be directly applied to each of the RBF g in the function (32). Under the extra assumption that $\phi(\boldsymbol{x}; \boldsymbol{\alpha}) = \varphi(\boldsymbol{\alpha} \cdot \boldsymbol{x})$, for φ L-Lipschitz (which without loss of generality can be shifted so that $\varphi(0) = 0$ and scaled so that $\sup|\varphi| \leq 1$), we can also apply Theorem 3.2 [40] for a corresponding uniform approximation. We only present the second case, the first one being similar. There are two equivalent ways to proceed.

b1) Using Theorem 3.2 [40], for an L-Lipschitz φ as defined above, for any $\delta > 0$ there exists a random function g_δ of the form

$$g_\delta(\boldsymbol{x}) = \sum_{k=1}^{K} \hat{c}_k \varphi(\boldsymbol{\alpha}_k \cdot \boldsymbol{x}), \tag{33}$$

where $\boldsymbol{\alpha}_k$ is i.i.d. randomly sampled from a chosen distribution p, which approximates $\hat{\epsilon}(\delta)$ close g with probability at least $1 - \delta$.

Using (33) into (32) we obtain

$$f_{\epsilon,\delta}(\boldsymbol{x}) = \sum_{i=1}^{N} \sum_{j=1}^{M_i} \sum_{k=1}^{K} c_i[f] w_{ij} \hat{c}_k \varphi(\boldsymbol{\alpha}_k \cdot (a_{ij}\boldsymbol{x} + \boldsymbol{b}_{ij})), \tag{34}$$

which if δ is chosen such that $\hat{\epsilon}(\delta) < \frac{\epsilon}{NM}$ satisfies $\|f^{RBF} - f_{\epsilon,\delta}\| < \epsilon$, hence, $\|f - f_{\epsilon,\delta}\| < 3\epsilon$.

Using a resummation of (34) in terms of a single summation index ℓ we end up with an approximation $f_{\epsilon,\delta}$ for f in the form

$$f_{\epsilon,\delta}(\boldsymbol{x}) = \sum_{\ell=1}^{\hat{N}} \hat{w}_\ell[f]\varphi(\boldsymbol{\alpha}_\ell \cdot (a_\ell \boldsymbol{x} + \boldsymbol{b}_\ell)), \tag{35}$$

where $\hat{w}_\ell[f] = c_i[f]w_{ij}\hat{c}_k$ are continuous functionals on U. Hemce, we obtain an approximation in terms of shifted and re-scaled L-Lipschitz random feature functions.

b2) One possible drawback of this expansion is that – see (34) – it depends both on the a_j, $\boldsymbol{b}_j$ and the $\boldsymbol{\alpha}_k$ and not on the $\boldsymbol{\alpha}_k$ only. An alternative could be to expand each one of the $g_j(\boldsymbol{x}) := g(\alpha_j \boldsymbol{x} + \boldsymbol{b}_j)$ separately. If it holds that $g_j \in \mathcal{G}$ for each j then

$$g_j(\boldsymbol{x}) = \sum_{k=1}^{K} \hat{c}_{jk}\varphi(\boldsymbol{\alpha}_k^{(j)} \cdot \boldsymbol{x}), \;\; j = 1, \cdots, M, \tag{36}$$

where $\boldsymbol{\alpha}^{(j)} := \{\boldsymbol{\alpha}_k^{(j)}, \;:\; k = 1, \cdots, K\} \sim_{i.i.d} p$ for each $j = 1, \cdots, M$ and with the $\hat{\boldsymbol{\alpha}}^{(j)}$ for different j being independent. When using this approach we get the expansion (35) with α_k i.i.d. from our initial distribution p. Upon resummation the stated result follows. □

Based on the Proposition 1, we can now prove the following proposition for universal approximation of functional $\mathcal{F} : U \to \mathbb{R}$ in terms of the random projection neural network (RPNN):

Proposition 2 *(Random Projection Neural Networks (RPNNs) for functionals). Adopting the framework from Proposition 1, and additionally, let* U *be a compact subset of* C(K) *and* $\mathcal{F}$ *be a continuous functional in* U*. Let us define the compact set* $\mathsf{U}_m \subseteq \mathbb{R}^d$ *of vectors, whose elements consist of the values of the function* $u \in \mathsf{U}$ *on a finite set of* m *grid points* $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m \in \mathbb{R}^d$ *and denote the vector* $\boldsymbol{u} := [u(\boldsymbol{x}_1), \ldots, u(\boldsymbol{x}_m)] \in \mathbb{R}^m$.

Then, with high probability, w.r.t. p*, for any* $\epsilon > 0$*, there exist* $M, m \in \mathbb{N}$*,* $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_M \in \mathsf{A}$*, i.i.d. distributed from* p*, such that:*

$$\left\| \mathcal{F}(u) - \sum_{i=1}^{M} w_i \varphi\left(\boldsymbol{\alpha}_i \cdot \boldsymbol{u}(\boldsymbol{x})\right) \right\|_\infty < \epsilon, \quad \forall u \in \mathsf{U}. \tag{37}$$

Proof. The representation of the function $u \in \mathsf{U}$ through a finite set of m evaluations $u(\boldsymbol{x}_j)$ is possible by the Tietze Extension Theorem for functionals from the set U_m to U (see [27] for more details). Then the proof comes directly from Proposition 1. □

Finally, using the above ideas and results and possibly allowing for different random embeddings for the branch and trunk networks, we can prove the following theorem:

Theorem 3.5 *(RandONets universal approximation for operators). Adopting the framework of Propositions 1, 2 and the notation of Theorem 3.2, and additionally, let:* X *be a Banach Space, and* $\mathsf{K}_1 \subset \mathsf{X}$, $\mathsf{K}_2 \subset \mathbb{R}^d$, $\mathsf{U} \subset C(\mathsf{K}_1)$ *be compact sets, and* $\mathcal{F} : \mathsf{U} \to \mathsf{C}(\mathsf{K}_2)$ *be a continuous (in the general case nonlinear) operator. Then, with high probability w.r.t.* p*, for any* $\epsilon > 0$*, there exist positive integers* $M, N, m \in \mathbb{N}$*, and network (hyper)parameters* $\boldsymbol{\alpha}_1^{br,tr}, \ldots, \boldsymbol{\alpha}_N^{br,tr} \in \mathsf{A}^{br,tr}$*, i.i.d. distributed from* $p_{\boldsymbol{\alpha}}$ *such that:*

$$\left\| \mathcal{F}(u)(\boldsymbol{y}) - \sum_{k=1}^{N}\sum_{i=1}^{M} w_{ki}\varphi^{br}\left(\boldsymbol{\alpha}_i^{br} \cdot \boldsymbol{u}(\boldsymbol{x})\right)\varphi^{tr}(\boldsymbol{\alpha}_k^{tr} \cdot \boldsymbol{y}) \right\|_\infty < \epsilon, \quad \forall u \in \mathsf{U}, \boldsymbol{y} \in \mathsf{K}_2, \tag{38}$$

where the superscripts br, tr *correspond to branch and trunk networks and can be chosen in generally independently.*

Proof. From the Proposition 1, we have that with high probability, for any $\epsilon_1 > 0$, there are $N \in \mathbb{N}$, $\tilde{w}_k[\mathcal{F}[u]]$ and $\boldsymbol{\alpha}_k^{tr} \in \mathsf{A}^{tr}$, such that

$$\left\| \mathcal{F}(u)(\boldsymbol{y}) - \sum_{k=1}^{N} \tilde{w}_k[\mathcal{F}[u]]\varphi^{tr}(\boldsymbol{\alpha}^{tr} \cdot \boldsymbol{y}) \right\|_\infty < \epsilon_1. \tag{39}$$

Moreover, from Proposition 1, we have that for any $k = 1, \ldots, N$, $\tilde{w}_k[\mathcal{F}[u]]$ is a continuous functional on U. We can therefore repeatedly apply Proposition 2, for each k, and obtain approximations of each functional $\tilde{w}_k[\mathcal{F}[u]]$ on U_m. Thus, with high probability, for any $\epsilon_2 > 0$ there exist $m, M \in \mathbb{N}$, w_{ki}, $\boldsymbol{\alpha}_i^{br} \in \mathsf{A}^{br}$, such that:

$$\left\| \tilde{w}_k[\mathcal{F}[u]] - \sum_{i=1}^{M} w_{ik}\varphi^{br}\left(\boldsymbol{\alpha}_i^{br} \cdot \boldsymbol{u}(\boldsymbol{x})\right) \right\|_\infty < \epsilon_2. \tag{40}$$

Combining (39) and (40) we obtain Eq. (38). This completes the proof. □

3.3.2. Implementation of RandONets

In this section, we present the architecture of RandONets, depicted in Fig. 1. As in Theorem 3.2, we use two single-hidden-layer FNNs with appropriate random bases as activation functions. We employ (nonlinear) random based projections for embedding, in the

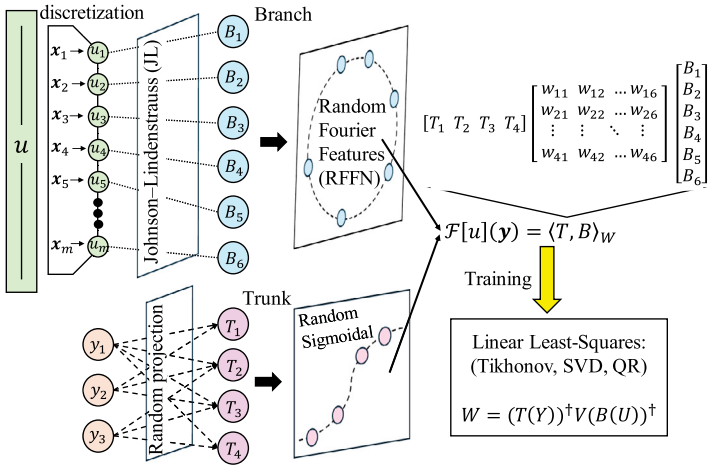


Fig. 1. Schematic of the random projection-based operator network (RandONet). The RandONets first discretizes the input function (u) over a fixed grid of spatial points. Then it separately embeds the space of the spatial locations ($\boldsymbol{y}$) into a random hidden layer (e.g., with sigmoidal activations functions) and the space of the discretized functions into low-distortion kernel-embedding (e.g., with Johnson-Lindenstrauss random projections [38] or Rahimi and Recht Random Fourier Features [39]). Finally, the output is composed of a weighted (W) inner product of the branch (B) and trunk (T) features. The training can be performed through linear least-squares techniques (e.g., Tikhonov regularization, SVD and QR decomposition).

two separate hidden layer features, both the (high-dimensional) space of the discretized function (u) and the domain (low-dimensional) of spatial locations ($\boldsymbol{y}$) of the transformed output ($v(\boldsymbol{y}) = \mathcal{F}[u](\boldsymbol{y})$).

Specifically, we propose leveraging nonlinear random projections to embed the space of spatial locations efficiently, employing parsimoniously chosen random bases. Thus, the random projected-based trunk feature vector $\boldsymbol{T} = (T_1, \dots, T_N)$, as denoted in Eq. (7), can be re-written as:

$$\boldsymbol{T} = \boldsymbol{\varphi}_n^{tr}(\boldsymbol{y}; \boldsymbol{\alpha}^{tr}) = [\varphi(\boldsymbol{y} \cdot \boldsymbol{\alpha}_1^{tr} + b_1), \dots, \varphi(\boldsymbol{y} \cdot \boldsymbol{\alpha}_N^{tr} + b_N)], \tag{41}$$

where $\boldsymbol{\alpha}_k^{tr} \in \mathbb{R}^d$, b_k, $j = 1, \dots, N$ are i.i.d. randomly sampled from a continuous probability distribution function.

Here, when the domain of $\mathcal{F}[u]$ is a one-dimensional interval $[a, b] \subseteq \mathbb{R}$, we select the activation function φ of the trunk network to be the hyperbolic tangent, and we utilize a parsimonious function-agnostic randomization of the weights as explained in [65,4,6]. In particular, the weights $\boldsymbol{\alpha}_j^{tr}$ are uniformly distributed as $\mathcal{U}[-a_U, a_U]$. The bounds a_U, of the uniform distributions, have been optimized in [65,4,6]. For the branch network, we have implemented two types of embeddings:

- Linear random Johnson-Lindenstrauss (JL) embeddings [38], in which case, we denote the branch feature vector $\boldsymbol{B} = (B_1, \dots, B_M)$ as:

$$\boldsymbol{B} = \boldsymbol{\phi}_M^{br}(\boldsymbol{U}) = \boldsymbol{\phi}_M^{JL}(\boldsymbol{U}) = \frac{1}{\sqrt{M}} R \boldsymbol{U}, \tag{42}$$

  where R is a matrix with elements that are sampled from a standard Gaussian distribution and $\boldsymbol{U}$ is the vector of function evaluation in the input grid.
- Nonlinear random embeddings [40]. Here, for our illustrations, we use a random Fourier feature network (RFFN) [39], as embedding of the functional space:

$$\begin{aligned} \boldsymbol{B} &= \boldsymbol{\varphi}_M^{br}(\boldsymbol{U}; \boldsymbol{\alpha}^{br}) = \boldsymbol{\varphi}_M^{RFFN}(\boldsymbol{U}; \boldsymbol{\alpha}^{br}, \boldsymbol{b}^{br}) = \\ &= \frac{1}{m} \sqrt{\frac{2}{M}} [\cos(\boldsymbol{\alpha}_1^{br} \cdot \boldsymbol{U} + b_1^{br}), \dots, \cos(\boldsymbol{\alpha}_M^{br} \cdot \boldsymbol{U} + b_M^{br})], \end{aligned} \tag{43}$$

  where we have two vectors of random variables $\boldsymbol{\alpha}^{br}$ and $\boldsymbol{b}^{br}$, from which we sample M realizations. The weights $\boldsymbol{\alpha}_i^{br}$ are i.i.d. sampled from a standard Gaussian distribution, and the biases $\boldsymbol{b}_i^{br}$ are uniformly distributed in $\mathcal{U}[0, 2\pi]$. This explicit random lifting has a low distortion for a Gaussian shift-invariant kernel distance.

The training of RandONets reduces to the solution of a linear least-squares problem in the unknowns W, i.e., the external weights of the weighted inner product as in Eq. (8). In what follows, before presenting the numerical implementation, we first present the treatment of aligned and unaligned data. The aligned data case refers to datasets where the training pairs are consistently organized, say in a grid, facilitating the learning process. In such scenarios, the network can more effectively learn the underlying nonlinear operator mappings due to the structured form of the data. On the other hand, training for unaligned data presents challenges compared to the aligned data case. In such scenarios, where the input and output pairs are not consistently organized, the network must learn to identify and map the complex relationships between disjoint datasets. This lack of alignment can make it more difficult for the

network to capture the underlying nonlinear operator mappings accurately. In general, achieving high accuracy in this context often demands greater computational resources and more extensive hyperparameter tuning to ensure the network converges to an optimal solution.

RandONets for aligned data Let us assume that the training dataset consists of s sampled input functions at m collocation/grid points. Thus, the input is included in a matrix $U \in \mathbb{R}^{m\times s}$. Let us also assume that the output function can be evaluated on a fixed grid of n points $\boldsymbol{y}_k \in \mathbb{R}^d$, which are stored in a matrix $Y \in \mathbb{R}^{n\times d}$ (row-vector); d is the dimension of the domain. In this case, we assume that for each input function u, we have function evaluations v at the grid Y stored in matrix $V = \mathcal{F}[U] \in \mathbb{R}^{n\times s}$.

While this assumption may appear restrictive at a first glance (as for example some values in the matrix V could be missing, or Y can be nonuniform, and may change in time), nonetheless, for many problems in dynamical systems and numerical analysis, such as the numerical solution of PDEs, entails employing a fixed grid where the solution is sought. This is clearly the case in methods like Finite Difference or Finite Elements-based numerical schemes without mesh adaptation. Additionally, even in cases where the grid is random or adaptive, there is still the opportunity to construct a "regular" output matrix V through "routine" numerical interpolation of outputs on a fixed regular grid. Now, given that the data are aligned, following Eq. (8), we can solve the following linear system (double-sided) of $n \times s$ algebraic equations in $N \times M$ unknowns:

$$V = \mathcal{F}[U] = \boldsymbol{\varphi}_n^{tr}(Y;\boldsymbol{\alpha}^{tr})\, W\, \boldsymbol{\varphi}_m^{br}(U;\boldsymbol{\alpha}^{br}) = T(Y)\, W\, B(U). \tag{44}$$

Let us observe that –differently from a classical system of equations (e.g., $Ax = b$), here we have two matrices from the trunk and the branch features, that multiply the readout weights W on both sides.

Although the number of unknowns and equations appears large due to the product, the convenient alignment of the data allows for effective operations that involve separate and independent (pseudo-) inversion of the trunk/branch matrices $T(Y) \in \mathbb{R}^{n\times N}$ and $B(U) \in \mathbb{R}^{M\times s}$. Thus, the solution weights of Eq. (44), can be found by employing methods such as the Tikhonov regularization [73], truncated singular value decomposition (SVD), QR/LQ decomposition and Complete Orthogonal Decomposition (COD) [74] of the two matrices, as we will detail later, obtaining:

$$W = \big(T(Y)\big)^{\dagger}\, V\, \big(B(U)\big)^{\dagger}. \tag{45}$$

As one might expect, the trunk matrix typically features smaller dimensions compared to the branch matrix. This is because the branch matrix may involve numerous samples s of functions (usually exceeding the number n of points in the output grid), along with a higher number of neurons M required to represent the high-dimensional function input, as opposed to the N neurons of the RP-FNN trunk which embeds the input space. At the end, the computational cost associated with the training (i.e., the solution of the linear least-squares problem) of RandONets is of the order $O(M^2 s + s^2 M)$. Here, we use the COD algorithm [74] for the inversion of both T and B matrices (for a comparison of truncated SVD and COD algorithms for RP-FNN training see in [65]).

RandONets for unaligned data In contrast to the aligned data, the output V cannot be usually stored in a matrix, but we have to consider a (long) vector. To address learning with unaligned data, it is sufficient to assume that for each input function u, the output $v(\boldsymbol{y})$ is available at a *single* random location in the output domain. This encompasses scenarios with a sparse random grid, where each output in the grid is treated separately, yet necessitating the introduction of multiple copies of the function u. Thus, let us assume we have stored the input functions in a matrix $U \in \mathbb{R}^{m\times S}$ and a vector of outputs $V \in \mathbb{R}^{1\times S}$.

Here, $S \in \mathbb{N}$ denotes both the total number of output functions and the total number of input functions. Unlike the aligned case, we store the random points for each input in the matrix $Y \in \mathbb{R}^{d\times S}$, where d now represents the columns instead of the rows, and S reflect the total number of (single) random locations where the individual outputs are sought.

Now, returning to Eq. (8), we notice that with the current format of inputs (both column-wise), we can express the output using the Hadamard (Shur) product ($\otimes$):

$$V = \sum_{k=1}^{N}\sum_{i=1}^{M} w_{ki} T_k(Y) B_i(U) = \sum_{k=1}^{n} T_k(Y) \otimes \boldsymbol{w}_k\, B(U), \tag{46}$$

where $\boldsymbol{w}_k$ are the rows of the matrix W. This corresponds to the original formulation of the DeepONet by Lu et al. (2021) [10], where instead of considering the merging of the branch and trunk networks as a weighted inner product, the focus is on the individual output at a single location, rather than treating the output as an entire transformed function. In this scenario, to solve the linear least-squares problem in terms of the $N \times M$ unknown weights W, we need to reshape the matrix W into a row vector $\boldsymbol{\omega}$, where the elements $\omega_q = w_{ki}$, with $q = k + (i-1)n$. Then we construct the full collocation matrix $Z \in \mathbb{R}^{NM\times S}$, such that the rows $\boldsymbol{z}_q$ are obtained as:

$$\boldsymbol{z}_q = T_k \otimes B_i, \qquad q = k + (i-1)N. \tag{47}$$

Note that the Hadamard product $T_k \otimes B_i$ is possible as both lie in $\mathbb{R}^S$.

At this point, the weights $\boldsymbol{\omega}$ can be computed, through a (pseudo-) inversion of the matrix Z, in analogy to what was detailed in section 3.2, resulting in:

$$\boldsymbol{\omega} = Y\, Z^{\dagger}. \tag{48}$$

We note that the total number S of single outputs can be viewed as proportional to the product of s (the number of different input functions) and N (the number of points in the output grid), as explained in the aligned case. In particular, employing an unaligned training algorithm for aligned data (by augmenting the input function with copies and reshaping the output) will result exactly in $S = Ns$. Now, the pseudo-inversion of the matrix Z will result in a computational cost of the order $O\big((Ns)^2MN + (MN)^2Ns\big)$, which is significantly higher. For instance, in the case the values of M, s are similar/proportional to N, we obtain a transition, in terms of computational complexity, from an order $O(N^3)$ for the aligned case to an order $O(N^6)$ for the unaligned case.

To this end, we argue that the unaligned approach described here should only be considered if the output data display substantial sparsity, suggesting that the random output grid does not adequately represent the output function. Conversely, we advocate for prioritizing the aligned approach in other scenarios. Even if it entails performing interpolation on a fixed grid to generate new aligned output data.

Numerical implementation of the training of RandONets Below, we provide more details on the training process of RandONets. From a numerical point of view, the resulting random trunk $T(Y)$ and branch $B(U)$ matrices, tend to be ill-conditioned. Therefore, in practice, we suggest solving Eq. (44) as described in 3.2 via a truncated SVD (tSVD), Tikhonov regularization, QR decomposition or regularized Complete Orthogonal decomposition (COD) [74,65]. Here, we describe the procedure for the branch network. For the trunk matrix, the procedure is similar.

The regularized pseudo-inverse $(B(U))^{\dagger}$, for the solution of the problem in Eq. (45) is computed as:

$$B(U) = U\Sigma V^T = [U_r \quad \tilde{U}] \begin{bmatrix} \Sigma_k & 0 \\ 0 & \tilde{\Sigma} \end{bmatrix} [V_k \quad \tilde{V}]^T, \qquad (B(U))^{\dagger} = V_k \Sigma_k^{-1} U_r^T, \tag{49}$$

where the matrices $U = [U_k \quad \tilde{U}] \in \mathbb{R}^{k\times n} \oplus \mathbb{R}^{(n-k)\times n}$ and $V = [V_k \quad \tilde{V}] \in \mathbb{R}^{k\times s} \oplus \mathbb{R}^{(s-k)\times s}$ are orthogonal and $\Sigma \in \mathbb{R}^{n\times s}$ is a diagonal matrix containing the singular values $\sigma_i = \Sigma_{(i,i)}$. Here, we select the k largest singular values exceeding a specified tolerance $0 < \epsilon \ll 1$, i.e., $\sigma_1, \ldots, \sigma_k > \epsilon$, effectively filtering out insignificant contributions and improving numerical stability.

Alternatively, a more robust method that further enhances numerical stability, involves utilizing a rank-revealing LQ decomposition (transposition of the QR decomposition, which can be used for the trunk matrix $T(Y)$) with column-pivoting:

$$P\,B(U) = [L \quad 0] \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, \tag{50}$$

where (e.g., if $n > s$) the matrix $Q = [Q_1 \quad Q_2] \in \mathbb{R}^{s\times n} \oplus \mathbb{R}^{(n-s)\times n}$ is orthogonal, $L \in \mathbb{R}^{s\times s}$ is a lower triangular square matrix and the matrix $P \in \mathbb{R}^{n\times n}$ is an orthogonal permutation of the columns. The key advantage of the column permutations lies in its ability to automatically identify and discard small values that contribute to instability. Indeed, in case of ill-conditioned matrices, we have that effectively the rows of the matrix Q do not span the same space as the rows of the matrix $B(U)$. As a result, the matrix L is not full lower triangular, but we have:

$$B(U) = P^T \begin{bmatrix} L_{11} & 0 \\ L_{21} & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, \tag{51}$$

where, if $rank(B(U)) = r < n$, the matrix $L_{11} \in \mathbb{R}^{r\times r}$ is effectively lower triangular and $L_{21} \in \mathbb{R}^{(s-r)\times r}$ are the remaining rows. Note that numerically, one selects a tolerance $0 < \epsilon << 1$ to estimate the rank r of the matrix $B(U)$ and set values of $B(U)$ below the threshold to zero. Then, to find the pseudo inverse of the branch matrix, we can additionally use the Complete Orthogonal Decomposition (COD) [74], by also computing the LQ decomposition of the transposed non-zero elements in L (for the trunk matrix this will correspond to a second QR decomposition):

$$[L_{11}^T \quad L_{21}^T] = [T_{11}^T \quad 0]V. \tag{52}$$

Finally, by setting $S^T = VP$, one obtains:

$$B(U) = S \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, \tag{53}$$

where T_{11} is an upper triangular matrix of size $r \times r$. Note that the inversion of the matrix T_{11}, can be efficiently computed using the back substitution algorithm that is numerically stable.

4. Numerical results

In this section, we present numerical results focusing exclusively on the aligned case within the framework of RandONets. This choice stems from (a) the observed superior computational cost of the proposed aligned approach; and (b) the aligned case fits well with the nature of the considered PDEs problems in dynamical systems and numerical analysis approaches. For a first proof of concept, our investigation encompasses a selection of benchmark problems originally addressed in the paper introducing DeepONet by Lu et al. (2021) [10]. Additionally, we focus on other benchmark problems concerning the evolution operator (right-hand-side (RHS)) of PDEs. Through these numerical experiments, we aim to demonstrate the effectiveness and versatility of RandONets in tackling a diverse array of challenging problems in dynamical systems and scientific computing for the solution of the inverse problem.

In [10], there is a discussion on how to generate the dataset of functions: they compare Gaussian Random Fields and other random parametrized orthogonal polynomial sets. Here, we decided to generate the input-output data functions without using precomputed datasets. We consider a random parametrized RP-FNN with 200 neurons and Gaussian radial basis functions combined with a few additional random polynomial terms.

$$u(t) = \boldsymbol{w} \exp\left(\boldsymbol{s}(t-\boldsymbol{c})^2\right) + a_0 + a_1\, t + a_2\, t^2, \tag{54}$$

where the parameters $\boldsymbol{w}, \boldsymbol{s}, \boldsymbol{c} \in \mathbb{R}^{200}$, $a_0, a_1, a_2 \in \mathbb{R}$ are all randomized to generate the dataset of input functions.

In all numerical examples, we select many different realizations of these functions. In some of these, in selecting the training datasets, we distinguish the case in which we utilize *limited-data* for training. In particular, we utilize 15% of the data for the training set and 85% for the test set. In the case in which we assume that we have available *extensive-data*, we utilize 80% of the data for the training set and 20% for the test set.

The range of the values of the parameters $(\boldsymbol{w}, \boldsymbol{s}, \boldsymbol{c}, a_0, a_1, a_2)$ is detailed for each case study. When possible, the output function is computed analytically. Otherwise, a well-established numerical method (with sufficiently small tolerances) is used to compute accurate solutions as the "ground truth". To represent both the input functions u and output functions v, we use an equally spaced grid of 100 points in the domains K_1 and K_2 of interest. In particular, in all examples considered here, we take as input, one-dimensional domains (intervals in $[a, b]$).

Regarding the architecture of the RandONets, as also detailed in Section 3.3.2, we investigate and compare the performance of two different RandONets architectures, with two well-established embedding techniques, respectively: linear random Johnson-Lindenstrauss (JL) embeddings denoted by ϕ_M^{JL} (as presented in Eq. (42)) and Random Fourier Feature Network (RFFN) embeddings denoted by ϕ_M^{RFFN} (as presented in Eq. (43)). These architectures will be subsequently referred to as RandONets-JL and RandONets-RFFN, respectively. We will explore the impact of varying the number of neurons (M) within the single hidden layer of the branch, effectively controlling the dimension of the branch embedding. For both RandONets-JL and RandONets-RFFN, the trunk embedding leverages a non-linear RP-FNN architecture denoted by ϕ_N^{tr} (as presented in (41)), which utilizes hyperbolic tangent activation functions ψ and parsimoniously function-agnostic randomization of the internal weights (as described in [65,6,4]). Throughout the experiments, we will maintain a consistent number of neurons ($N = 200$) within the trunk's hidden layer, thus ensuring a fixed size for the trunk embedding. It is important to note that the RandONets-JL architecture incorporates a combination of linear and non-linear embedding techniques, whereas the RandONets-RFFN architecture is entirely non-linear.

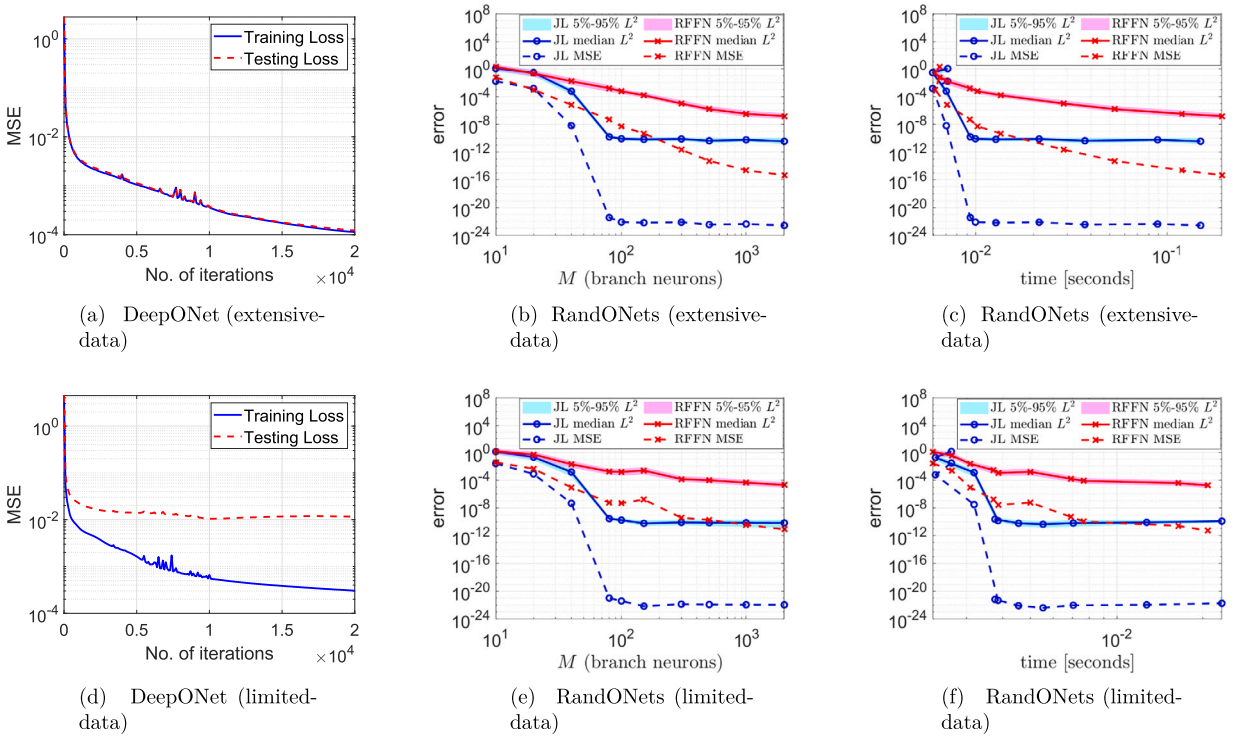
Given the big difference in the computational cost for the inversion of the branch matrix compared to the trunk matrix, we decided to fix the number of neurons $N = 200$ in the trunk RP-FNN embedding for both the RandONets-JL and RandONets-RFFN. The inversion of the corresponding trunk matrix $\boldsymbol{T} \in \mathbb{R}^{100\times 200}$ thus it is, for any practical purposes, relatively negligible. Here, we investigate the performance of the scheme for $M = 10, 20, 40, 80, 100, 150, 300, 500, 1000, 2000$ neurons in the branch embedding of both RandONets-JL and RandONets-RFFN and the corresponding increment in computational cost.

Metrics To assess the performance of the RandONets, we utilize both the mean squared error (MSE) for the entire test set, as well the L^2–error for each output-function in the test set. In particular, we report the median L^2 and the percentiles $5\% - 95\%$. Importantly, we report the execution time in seconds of the scheme, thus indicating when the computations are performed with GPU or CPU.

Remark on the DeepONet architectures used Given the high-computational cost associated when training DeepONets with the Adaptive Moment Estimation (Adam) algorithm, (even if we employ a GPU hardware), we do not focus now on performing a convergence diagram of the scheme or finding the best architecture. For our illustrations, we just selected a few configurations. In particular, we selected 2 hidden layers for both trunk and branch networks, each layer with a prescribed number $N = \{5, 10, 20, 40\}$ of neurons. Also, we employ hyperbolic tangent as activation functions for both branch and trunk networks. We will refer to the performance of these vanilla DeepONets in Tables with the notation $[N, N]$. We remark that the number of free trainable parameters ζ of such DeepONet configurations is $m \times N + 3N \times N + 5N$, (e.g., $N = 40, m = 100$, then $\zeta = 9000$) which is not higher than the biggest considered RandONets. Indeed, RandONets have a number of free trainable parameters equal to $N \times M$ (e.g., in the biggest case considered here, $N = 200$, $M = 2000$, it corresponds to $\zeta = 400,000$ parameters). However, as we will show, despite the high number of parameters, such RandONets can be trained in around one second. Finally, we also remark that when employing a gradient-descent based algorithm, like the Adam one, there is no guarantee of convergence, and the generalization of the network can be moderate. We anyway decided to train DeepONet for a fixed number of iterations equal to $20,000$ for ODE benchmarks and to $50,000$ for the PDEs.

Remark on the hardware and software used In our experiments, we utilized the DeepONet framework implemented in Python with the DeepXde library [2], leveraging TensorFlow as the back-end for computations. These computations are executed on a GPU NVIDIA GeForce RTX 2060, harnessing its parallel processing capabilities to expedite training and evaluation. Additionally, we ran the Python code on `Google Colab` using a Tesla K80 GPU, resulting in computational times approximately 6 to 7 times slower than those reported in the main text. While we do not include these execution times in the main text, we mention them here as a reference. In contrast, the RandONets framework is implemented in `MATLAB` 2024a and executed on a single CPU of an Intel(R) Core(TM) i7-10750H CPU @ 2.60 GHz, with 16 GB of RAM. The code is available at https://github.com/GianlucaFabiani/RandONets.

It is worth noting that while the hardware and software environments for the two frameworks differ significantly, hindering a direct comparison of computational times, we report the computational times for both approaches for transparency. Despite the



(a) DeepONet (extensive-data) (b) RandONets (extensive-data) (c) RandONets (extensive-data)

(d) DeepONet (limited-data) (e) RandONets (limited-data) (f) RandONets (limited-data)

Fig. 2. Case study 1: antiderivative operator, in Eq. (56). (First row) extensive-data case, 800 training input functions; (Second row) limited-data case, 150 training input functions. (a), (d) MSE for the training and test sets, with DeepONets with 2 hidden layers (indicatively) with 40 neurons each, for both the branch and trunk networks. (b), (c), (e), (f) MSE and L^2 error, $5\% - 95\%$ range and median, of the RandONets, for different size M of the branch embedding. The errors are computed w.r.t. only the output functions in the test dataset. Comparison of Johnson-Lindenstrauss (JL) branch embedding with random Fourier features (RFFN) embeddings. We set the size of the Trunk network to $N = 200$ and the grid of input points to $m = 100$. Numerical approximation accuracy vs. (b)-(e) the number of neurons M in the hidden layer of the branch network; and (c)-(f) vs. computational times in seconds.

disparity in hardware and software, it is noteworthy that the single CPU utilized is not suitable for the Python code, whereas it proves to be efficient for the RandONets implemented in `MATLAB`. Additionally, while we acknowledge that differences in computational times between `MATLAB` and Python platforms may arise due to a range of factors beyond software differences alone, we believe that the reported differences cannot be just explained by the two different software implementations. We provide these computational costs as they are observed, recognizing that, while a hand-made `MATLAB` implementation of DeepONet is feasible, our objective is to compare with the professionally implemented and widely used DeepXde library [2] to ensure a reliable comparison.

4.1. Some simple (pedagogical) ODE benchmark problems

We start with some very simple benchmark problems involving non-autonomous ODEs subject to a time dependent source term $u(t)$, in the form:

$$\frac{dv}{dt}(t) = f(t, v(t)) + u(t), \quad t \in [0, T], \qquad v(0) = v_0, \tag{55}$$

with some forcing time-dependent input function $u(t)$. The solution function $v(t)$ depends directly on the forcing term, and the function $u(t)$. Thus, there exists an operator that maps $u(t)$ into $v(t)$. The task here, different from the one for the PDEs, is to learn the "solution operator" for one initial condition. Of course, learning the full solution operator would need a set of different initial conditions as in [10] but as mentioned these serve purely for pedagogical purposes. Still, the focus, in this first work, is on the approximation of the evolution operators of PDEs; we will present these results in the subsection 4.2.

4.1.1. Case study 1: anti-derivative operator

The first benchmark problem that we consider is the anti-derivative operator [75], thus the solution $v(t)$ given the function $u(t)$ of the following (phase-independent, $f(t, v) \equiv 0$) ODE problem (see Fig. 2 and Table 1):

$$\frac{dv}{dt}(t) = u(t), \quad t \in [0, T], \qquad v(0) = v_0, \quad t \in [0, 1], \tag{56}$$

thus learning the linear operator $\mathcal{F}[u](t) = v(t)$. The corresponding analytical anti-derivatives are:

Table 1
Case study 1: Anti-derivative operator in Eq. (56). We report Mean Squared Error (MSE), percentiles (median, 5%, 95% of L^2 error across the test set. The extensive-data case comprises 800 training functions, while the limited-data case uses 150 functions as training. We employed vanilla DeepONets with 2 hidden layers, denoted as $[N, N]$ neurons, for both the branch and the trunk. We set $N = 5, 10, 20, 40$. DeepONets are trained with 20,000 Adam iterations (with learning rate 0.001 and then 0.0001). We report the RandONets encompassing Johnson-Lindenstrauss (JL) Featured branch network (with $M = 100$ neurons) as well as the Random Fourier Feature branch Network (RFFN) (with $M = 2000$ neurons).

data	ML-model	MSE	5% L^2	median–L^2	95% L^2	comp. time
80%	DeepONet [5,5]	9.83E−01	2.31E+00	6.90E+00	1.80E+01	4.75E+02 (GPU)
	DeepONet [10,10]	2.28E−03	2.43E−01	4.37E−01	7.46E−01	4.62E+02 (GPU)
	DeepONet [20,20]	4.39E−04	1.26E−01	2.00E−01	2.97E−01	4.79E+02 (GPU)
	DeepONet [40,40]	1.22E−04	7.04E−02	1.03E−01	1.60E−01	5.23E+02 (GPU)
	RandONets–JL (100)	9.43E−23	4.33E−11	8.01E−11	1.68E−10	1.02E−02 (CPU)
	RandONets–RFFN (2000)	8.09E−16	6.81E−08	1.73E−07	5.99E−07	1.96E−01 (CPU)
15%	DeepONet [5,5]	8.88E−02	1.08E+00	2.48E+00	5.15E+00	1.05E+02 (GPU)
	DeepONet [10,10]	2.99E−03	2.73E−01	4.91E−01	8.51E−01	9.78E+01 (GPU)
	DeepONet [20,20]	7.48E−04	1.51E−01	2.54E−01	4.07E−01	1.13E+02 (GPU)
	DeepONet [40,40]	1.16E−02	2.57E−01	7.36E−01	2.12E+00	1.24E+02 (GPU)
	RandONets–JL (100)	1.66E−21	2.22E−10	3.74E−10	6.11E−10	3.60E−03 (CPU)
	RandONets–RFFN (2000)	8.12E−12	8.14E−06	2.03E−05	5.25E−05	1.88E−02 (CPU)

$$v(t) = \boldsymbol{w}\left(-\sqrt{\pi}\,\mathrm{erfi}\left(\sqrt{\boldsymbol{s}}(\boldsymbol{c}-t)\right)/(2\sqrt{\boldsymbol{s}})\right) + a_0 t + a_1 t^2/2 + a_2 t^3/3 + C, \tag{57}$$

where erfi is the error function and C is a constant that has to be fit by the initial condition $v(0) = v_0$. We select $v_0 = 0$ and we set $\tilde{v}(t) = v(t) - v(0)$ as the output function.

The values of the parameters $\boldsymbol{w}$, a_0, a_1, $a_2 \sim \mathcal{U}[-1, 1]$, $\boldsymbol{s} \sim \mathcal{U}[0, 500]$ and $\boldsymbol{c} \sim \mathcal{U}[0, 1]$, of the RP-FNN based function dataset, as in Eq. (54), are (element-wise) sampled from the corresponding aforementioned uniform distributions. To generate the data, we used 1000 random realizations. We considered two different sizes of training sets. In particular, we used 15% for the training and 85% for the test set (we remind the reader that we call this limited-data case). As described above, for the extensive data-case we used 80% for the training and 20% for the test set.

In Fig. 2, we depict the numerical approximation accuracy for the test set in terms of the MSE and percentiles median, $5\% - 95\%$ of L^2–errors. As shown, the training of all RandONets takes approximately less than 1 second and is performed without iterations. In Table 1, we summarize the comparison results with the vanilla DeepONet in terms of the best accuracy and best computational times. For the RandONets, we used 100 neurons for the JL embedding, and 2000 neurons for the RFFN embedding for the branch network. As shown, the JL-based RandONets gets an astonishing almost machine-precision accuracy of $MSE \simeq 1E\text{-}23$, $L^2 \simeq 1E\text{-}11$ with just 40 neurons in the branch with a computational time of $\simeq 0.01$ seconds. Such "perfect" results are due to the simplicity of the problem and its linearity. The nonlinear RFFN embedding result in a lower performance with respect to the JL RandONets, for this linear problem, obtaining an $MSE \simeq 1E\text{-}16$ and a median $L^2 \simeq 1E\text{-}08$, using 2000 neurons in the branch, with a computational time of less of the order 0.1 seconds. We employ vanilla DeepONets with two hidden layers, denoted as $[N, N]$ neurons, in both Trunk and Branch. We observe a rather slow convergence in accuracy by increasing the size of the Vanilla DeepONets. However, the vanilla DeepONets need many iterations to reach an adequate accuracy. After 20,000 iterations, the accuracy in the extensive-data case, for $N = 40$, is around $1E\text{-}04$ in terms of MSE, but the L^2 error is on the order of $1E\text{-}01$. In the limited-data case, the vanilla DeepONet, with $N = 40$, gives a rather poor performance: the MSE on test data is stuck at $1E\text{-}02$ thus, overfitting. The corresponding L^2 error is on the order of 1. Indeed, the DeepONet with $N = 20$ performs better. We can explain such failure due to difficult dataset considered, that needs sufficient input-output functions to be well represented.

Our results for the two hidden layers vanilla DeepONet are in line with the ones presented in [10]. Also, for investigations on different architectures one can refer to the same paper. In particular, there, for the vanilla DeepONet with 4 hidden layers and $[2560, 2560, 2560, 2560]$ neurons in both trunk and branch, they report an MSE of around $1E\text{-}08$ after 50,000 iterations.

As a matter of fact, for this case study, the execution times (training times) for RandONet, utilizing both linear JL and nonlinear RP-FNN random embeddings, are 10,000 times faster, while achieving L^2 accuracy that is 6 to 10 orders of magnitude higher.

4.1.2. Case study 2: pendulum with external force

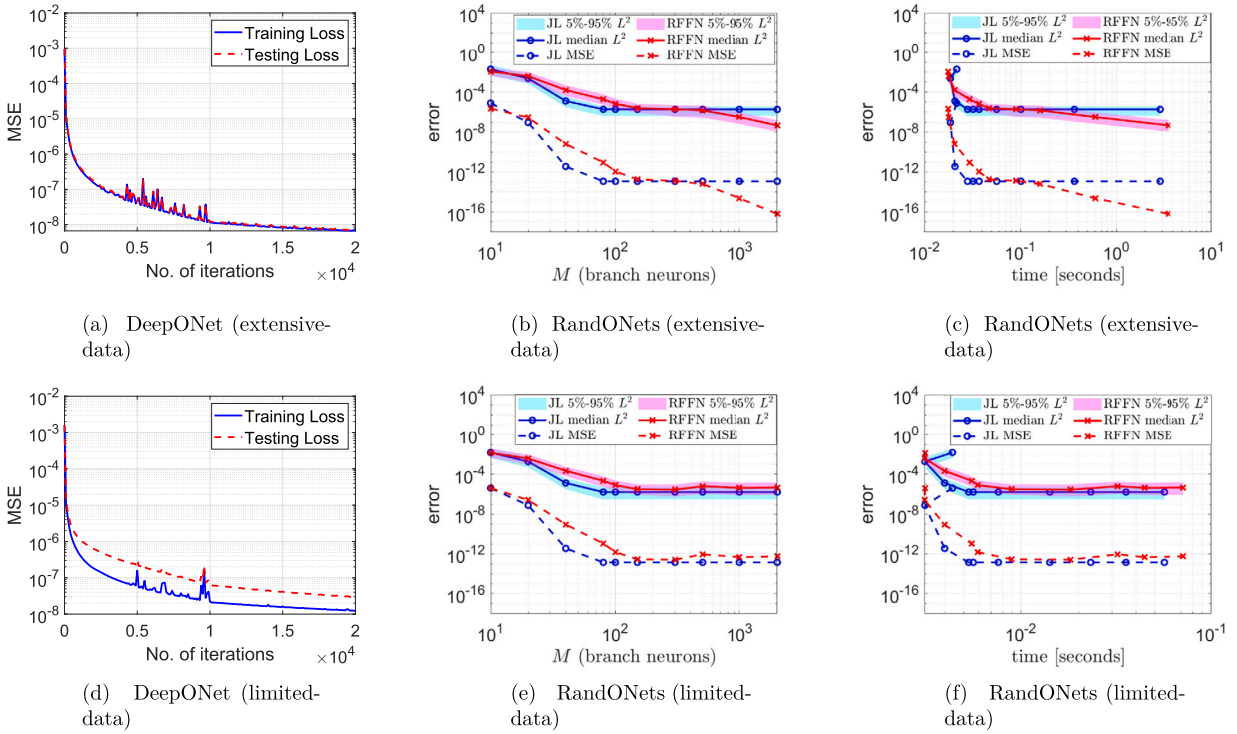
We consider the motion of a simple pendulum, consisting of a point mass suspended from a support by a mass-less string of length $l = 1$, on which act the gravity force and an additional external force $u(t)$. The system is described by the following second order ODE:

$$\frac{d^2 v}{dt^2}(t) = -k \sin v(t) + u(t), \qquad t \in [0, 1], \quad v(0) = 0, \quad v'(0) = 0, \tag{58}$$

where v represents the angle with the vertical, and $k = 9.81$. (See Fig. 3 and Table 2.)

Again, the goal is to learn the operator that maps the input function $u(t)$ to the output function $v(t)$.

The values of the parameters, of the RP-FNN based function dataset (see in Eq. (54)), are randomly sampled from uniform distributions as follows: The values of $\boldsymbol{w}$, a_0, a_1, a_2 are sampled uniformly in $\mathcal{U}[-0.05, 0.05]$, the value of $\boldsymbol{s}$ from $\mathcal{U}[0, 500]$ and the value of $\boldsymbol{c}$ is uniformly sampled from $\mathcal{U}[0, 1]$.



(a) DeepONet (extensive-data) (b) RandONets (extensive-data) (c) RandONets (extensive-data)

(d) DeepONet (limited-data) (e) RandONets (limited-data) (f) RandONets (limited-data)

Fig. 3. Case study 2: pendulum with external force, in Eq. (58). (First row) extensive-data case, 2400 training input functions; (Second row) limited-data case, 450 training input functions. (a), (d) Convergence of training and test set MSE of the DeepONet with two hidden layers (indicatively) with 40 neurons each, for both the branch and trunk networks. (b), (c), (e), (f) MSE and L^2 error percentiles (median, $5\% - 95\%$), of the RandONets, for different size M of the branch embedding. The errors are computed w.r.t. only the output functions in the test dataset. Comparison of Johnson-Lindenstrauss (JL) branch embedding, in Eq. (42), with Random Fourier Feature Networks (RFFN) embeddings, in Eq. (43). We set the size of the Trunk network to $N = 200$ and the grid of input points to $m = 100$. Numerical approximation accuracy vs. (b)-(e) number of neurons M in the hidden layer of the branch network; and (c)-(f) vs. computational time in seconds.

Table 2
Case study 2: simple pendulum with external force, as in Eq. (58). We report Mean Squared Error (MSE) and percentiles (median, $5\% - 95\%$) of L^2 error across the test set. The extensive-data case comprises 2400 training functions, while the limited-data case uses 450 functions as training. We employed vanilla DeepONets with 2 hidden layers, with $[N, N]$ neurons, in both trunk and branch. We set $N = 5, 10, 20, 40$. DeepONets are trained with 20,000 Adam iterations (with learning rate 0.001 and then 0.0001). We report the RandONets encompassing Johnson-Lindenstrauss (JL) Featured branch network (with $M = 100$ neurons) and the Random Fourier Feature branch Network (RFFN) (with $M = 150$ neurons for few data and with $M = 2000$ neurons for many data).

data	ML-model	MSE	5% L^2	median–L^2	95% L^2	comp. time
80%	DeepONet [5, 5]	2.63E−07	2.41E−03	4.50E−03	8.40E−03	1.25E+03 (GPU)
	DeepONet [10, 10]	1.50E−07	1.71E−03	3.48E−03	5.88E−03	1.19E+03 (GPU)
	DeepONet [20, 20]	1.83E−08	7.92E−04	1.27E−03	1.91E−03	1.36E+03 (GPU)
	DeepONet [40, 40]	7.02E−09	5.37E−04	7.96E−04	1.13E−03	1.32E+03 (GPU)
	RandONets–JL (100)	1.19E−13	3.96E−07	1.89E−06	3.75E−06	3.31E−02 (CPU)
	RandONets–RFFN (2000)	6.52E−17	1.09E−08	4.66E−08	1.58E−07	3.91E+00 (CPU)
15%	DeepONet [5, 5]	2.30E−07	2.09E−03	4.10E−03	7.87E−03	2.97E+02 (GPU)
	DeepONet [10, 10]	1.08E−07	1.48E−03	2.98E−03	5.02E−03	2.6170E+02 (GPU)
	DeepONet [20, 20]	2.27E−08	8.57E−04	1.37E−03	2.24E−03	2.77E+02 (GPU)
	DeepONet [40, 40]	2.92E−08	8.77E−04	1.45E−03	2.81E−03	2.89E+02 (GPU)
	RandONets–JL (100)	1.41E−13	2.82E−07	1.62E−06	5.03E−06	5.63E−03 (CPU)
	RandONets–RFFN (150)	2.91E−13	8.55E−07	3.14E−06	9.59E−06	8.91E−03 (CPU)

To obtain the "ground-truth" corresponding output functions, we employ the `MATLAB` solver `ode45` with absolute tolerance set to 1E−12 and relative tolerance set to 1E−10. Here, we consider, compared to case 1, relatively smaller amplitudes in the functions, since a high forcing term can lead the system far from the initial condition. It is important to note also that in this case, the solution v is not a simple primitive of the function u: it corresponds for a given initial condition to a nonlinear solution operator.

To generate the data, we used 3000 random realizations for the values of the parameters. We consider two different sizes for the training set. As described above, we used 15% of the data for training and 85% for testing (for the limited-data case) and 80% of the data for training and 20% for testing (for the extensive-data case).

In Fig. 3, we report the numerical approximation accuracy for the test set in terms of the MSE and the median (and corresponding percentiles $5\% - 95\%$) of the L^2–error. As shown, the training time of RandONets takes approximately less than one second and is performed without iterations. In Table 2, we also summarize the comparative results in terms of the approximation errors, and computational times between RandONets, and vanilla DeepONets. For our illustrations, for RandONets, we have used $M = 100$ neurons for the JL embeddings and $M = 150$ neurons for the RFFN, in the limited-data case, and $M = 2000$ neurons in the extensive-data case. Compared to case 1, the nonlinear RFFN embeddings perform slightly better than the linear JL embeddings, especially for higher sizes of the branch network. This is due to nonlinearity of the benchmark, yet they still both schemes exhibit comparable performance for smaller sizes of the networks. Actually, JL embeddings converge faster, generalizing better with few neurons compared to the nonlinear RFFN embeddings. This indicates that, despite the JL embedding being linear, the nonlinearity of the operator can be effectively approximated by the hyperbolic tangent RP-FNN based trunk embedding for the spatial locations y. Also, the DeepONets performed better than in case 1.

Finally, for this case study, the execution times when using RandONets (with JL and nonlinear RFFN embeddings) are 3 to 5 orders faster, than DeepONets, while achieving a 3 to 4 orders higher numerical approximation accuracy in terms of the L^2 error.

4.2. Approximation of evolution operators (RHS) of time-dependent PDEs

Here, we consider some benchmark problems relative to the identification of the evolution operator, i.e., the right-hand-side (RHS) of time-dependent PDEs:

$$v(x,t) = \frac{\partial u}{\partial t}(x,t) = \mathcal{L}(u)(x,t). \tag{59}$$

The output function, the time derivative, (i.e., the right-hand-side of the evolutionary PDE) depends on the current state profile $u(x,t)$ at a certain time t. In the following examples, we do not consider the limited-data case.

4.2.1. Case study 3: 1D linear diffusion-advection-reaction PDE

As a first example for the learning of the evolution operator, we consider a simple 1D linear Diffusion-advection-reaction problem, described by:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \gamma \frac{\partial u}{\partial x} + \zeta u, \qquad x \in [-1, 1] \tag{60}$$

where $\nu = 0.1$, $\gamma = 0.4$ and $\zeta = -1$. (See Fig. 4 and Table 3.)

The output function can be computed analytically/symbolically based on Eq. (54). The values of the parameters of the RP-FNN based function dataset, in Eq. (54), are uniformly sampled as follows: $\boldsymbol{w}$, a_0, a_1, $a_2 \sim \mathcal{U}[-1, 1]$, $\boldsymbol{s} \sim \mathcal{U}[0, 50]$ and $\boldsymbol{c} \sim \mathcal{U}[0, 1]$. Here we select w in a smaller range, as higher values may correspond to high derivatives resembling singularity in the second derivative.

To generate the data, we used 2000 random realizations of the values of the parameters of the RP-FNNs based function dataset, as in (54). We set 80% for training and 20% for testing.

In Fig. 4, we depict the approximation accuracy w.r.t. the test set in terms of the MSE and the percentiles (median, $5\% - 95\%$) of L^2–approximation errors. As shown, the training of all RandONets takes approximately less or around 0.1 seconds and it is performed without iterations. In Table 3, we report the comparative results w.r.t. the numerical approximation accuracy and computational times of the RandONets, here for JL with 100 neurons and RFFN with 500 neurons in the branch embedding. In this case, the numerical results suggest that DeepONets architectures exhibit limitations in achieving satisfactory accuracy, even if the operator is a simple linear RHS of a PDE. This is evident despite extensive training with Adam for $50,000$ iterations. Furthermore, we observed a concerning trend of slow convergence in DeepONet performance as the hidden layer size (N) increased from 5 to 40 neurons. In contrast, RandONets architectures demonstrate significantly faster learning and superior performance. RandONets-JL achieves exceptional accuracy with a modest number of neurons ($M = 100$). This is reflected in the Mean Squared Error (MSE) on the order of 10^{-17} and the L^2 error on the order of 10^{-8}. While RandONets-RFFN requires slightly more neurons ($M = 500$) to reach its best performance. This still delivers respectable results with an MSE on the order of 10^{-11}.

As a matter of fact, also for this case study, RandONets, utilizing both JL and RFFN, exhibits execution times (training times) that are on the order of 1000 times faster, while achieving L^2 accuracy that is 5 to 8 orders of magnitude higher than the vanilla DeepONets.

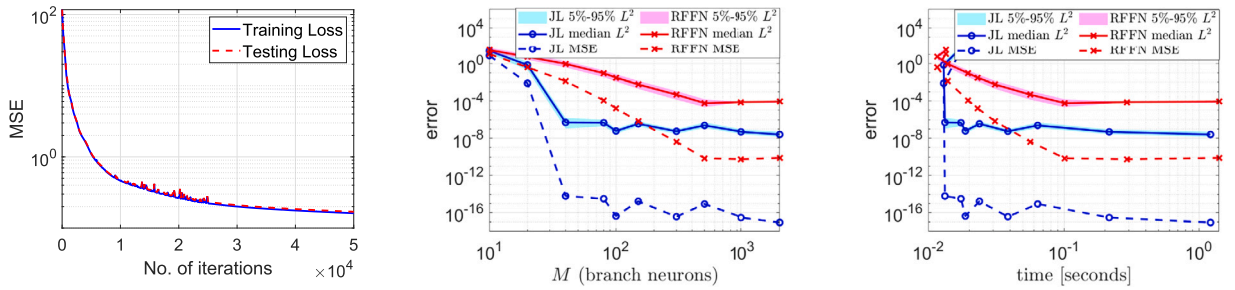
4.2.2. Case study 4: 1D viscous Burgers PDE

We consider the nonlinear evolution operator of the Burgers' equation given by:

$$v = \frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} - u \frac{\partial u}{\partial x}, \tag{61}$$

where $\nu = 0.01$. (See Fig. 5 and Table 4.)

The output function can be computed analytically/symbolically based on (54). The parameters $\boldsymbol{w}$, a_0, a_1, $a_2 \sim \mathcal{U}[-0.05, 0.05]$, $\boldsymbol{s} \sim \mathcal{U}[0, 50]$ and $\boldsymbol{c} \sim \mathcal{U}[-1, 1]$, of the RP-FNN based function dataset, as in Eq. (54), to represent the functional space are (element-wise) uniformly distributed. Here we select w in a smaller range, as higher values may correspond to high second derivatives approaching singularity.

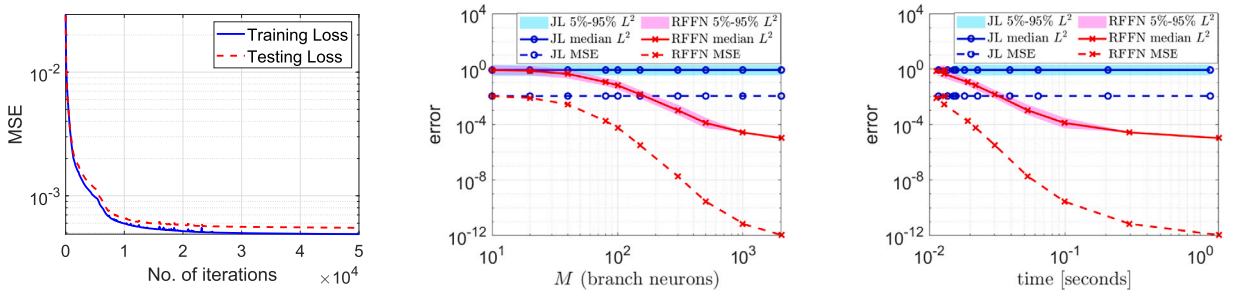


(a) DeepONet (b) RandONets (c) RandONets

Fig. 4. Case study 3: 1D diffusion-advection-reaction linear PDE in Eq. (60). We use 1600 training input functions; (a) Convergence of training and test MSE of the DeepONet with two hidden layers (indicatively) with 40 neurons each, for both branch and trunk networks. (b), (c), MSE and L^2 error percentiles (median, $5\%-95\%$), of the RandONets, for different size M of the branch embedding. The errors are computed w.r.t. only the output functions in the test dataset. Comparison of Johnson-Lindenstrauss (JL) random features, in Eq. (42), with random Fourier features (RFFN) embeddings, in Eq. (43). We set the size of the trunk network to $N = 200$ and the grid of input points to $m = 100$. Numerical approximation accuracy vs. (b) number of neurons M in the hidden layer of the branch network; and (c) computational time in seconds.

Table 3
Case study 3: 1D diffusion-advection-reaction linear PDE (Eq. (60)). We report the mean squared error (MSE) and percentiles (median, $5\%-95\%$) of L^2 error for the test set. We used 1600 training functions. We employed a vanilla DeepONet with 2 hidden layers with $[N, N]$ neurons. We set $N = 5, 10, 20, 40$. DeepONets are trained with 50,000 Adam iterations (with learning rate 0.001 and then 0.0001). We report the RandONets encompassing Johnson-Lindenstrauss (JL) Featured branch network (with $M = 100$ neurons) and the random Fourier features (RFFN) (with $M = 500$ neurons).

ML-model	MSE	5% L^2	median–L^2	95% L^2	comp. time
DeepONet [5, 5]	4.74E+01	3.42E+01	6.30E+01	1.04E+02	2.18E+03 (GPU)
DeepONet [10, 10]	2.03E+01	1.88E+01	4.15E+01	6.81E+01	2.20E+03 (GPU)
DeepONet [20, 20]	1.57E+00	6.68E+00	1.09E+01	1.95E+01	2.31E+03 (GPU
DeepONet [40, 40]	1.69E−01	2.29E+00	3.73E+00	6.30E+00	2.30E+03 (GPU)
RandONets–JL (100)	4.33E−17	3.14E−08	5.98E−08	1.02E−07	1.83E−02 (CPU)
RandONets–RFFN (500)	7.03E−11	2.14E−05	5.87E−05	1.56E−04	1.02E−01 (CPU)



(a) DeepONet (b) RandONets (c) RandONets

Fig. 5. Case study 4: 1D nonlinear Burgers' PDE (Eq. (61)). We used 1600 training input functions: (a) MSE when using a vanilla DeepONet with 2 hidden layers with (indicatively) 40 neurons each, for both branch and trunk networks. (b), (c), MSE and L^2 error percentiles (median, $5\%-95\%$), of the RandONets for different size M of the branch embedding. Comparison of Johnson-Lindenstrauss random embeddings, as in Eq. (42), with random Fourier features (RFFN) embeddings, as in Eq. (43). We have set the size of the trunk network to $N = 200$ and the grid of input points to $m = 100$. Numerical approximation accuracy vs. (b) number of neurons M in the hidden layer of the branch network; and (c) vs. computational time in seconds.

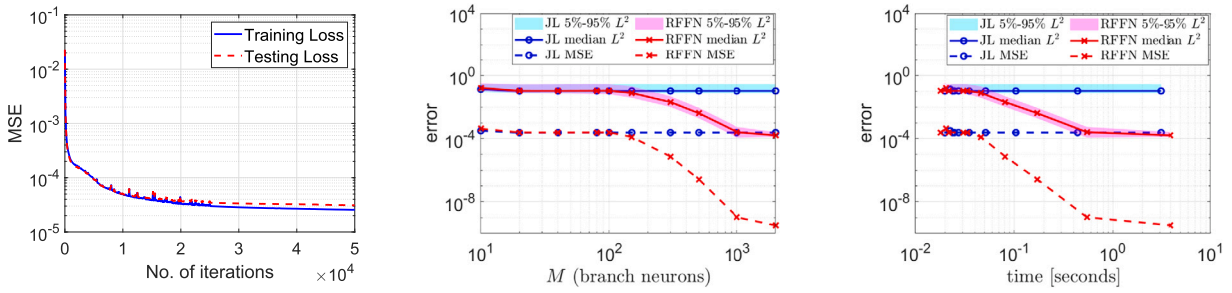
To generate the data, we used 2000 random realizations of the parameters. We set 80% for training and 20% for testing. In Fig. 5, we report the accuracy w.r.t. the test set in terms of the MSE and the median (and percentiles $5\%-95\%$) of L^2–errors. As shown, the training of all RandONets takes approximately less or around one second. In Table 4, we also report the comparison results in terms of the numerical approximation accuracy and computational times.

Due to the inherent non-linearity of this example, linear JL random embeddings exhibit limitations in efficiently approximating the non-linear operator. This observation aligns with the theoretical understanding of JL embeddings being most effective in capturing linear relationships. Unlike case study 2, the non-linearity within only the trunk architecture appears insufficient for this specific problem. Therefore, incorporating non-linearity also in the branch embedding becomes crucial for achieving optimal performance.

Interestingly, the performance of JL embeddings approaches that of fully trained, entirely non-linear vanilla DeepONets. While DeepONets can achieve a minimum Mean Squared Error (MSE) on the order of 1E−04 and an L^2 error on the order of 1E−01, their performance is not significantly better than the JL approach. In contrast, the RandONets-RFFN architecture emerges as the clear

Table 4
Case study 4: Burgers' nonlinear PDE in Eq. (61). We report Mean Squared Error (MSE) and percentiles (median, $5\%-95\%$) of L^2 approximation errors, for the test set. We use 1600 training functions. We employ DeepONets with 2 hidden layers with $[N, N]$ neurons in both the branch and trunk. We set $N = 5, 10, 20, 40$. DeepONets are trained with 50,000 Adam iterations (with learning rate 0.001 and then 0.0001). We report the RandONets results encompassing Johnson-Lindenstrauss (JL) Featured branch network (with $M = 40$ neurons) and the Random Fourier Feature branch Network (RFFN) (with $M = 2000$ neurons).

ML-model	MSE	5% L^2	median–L^2	95% L^2	comp. time
DeepONet [5, 5]	9.00E−03	3.70E−01	7.60E−01	1.66E+00	2.16E+03 (GPU)
DeepONet [10, 10]	4.75E−03	3.43E−01	5.59E−01	1.20E+00	2.01E+03 (GPU)
DeepONet [20, 20]	1.51E−03	2.24E−01	3.28E−01	6.16E−01	2.40E+03 (GPU)
DeepONet [40, 40]	5.50E−04	1.30E−01	2.03E−01	3.82E−01	2.34E+03 (GPU)
RandONets–JL (40)	1.09E−02	3.32E−01	8.11E−01	1.91E+00	1.29E−02 (CPU)
RandONets–RFFN (2000)	1.12E−12	1.01E−05	1.04E−05	1.19E−05	1.51E+00 (CPU)



(a) DeepONet (b) RandONets (c) RandONets

Fig. 6. Case study 5: 1D Allen-Cahn phase-field PDE in Eq. (62). We use 800 training input functions; (a) Convergence of training and test MSE of the DeepONet with 2 hidden layer (indicatively) with 40 neurons each, for both branch and trunk networks. (b), (c), MSE and L^2 approximation errors percentiles (median, $5\%-95\%$), of the RandONets, for different size M of the branch embedding. The errors are computed w.r.t. only the output functions in the test dataset. Comparison of RandONets with Johnson-Lindenstrauss random features, as in Eq. (42), RandONets with Random Fourier Feature Networks (RFFN), as in Eq. (43). We set the size of the Trunk network to $N = 200$ and the grid of input points to $m = 100$. Numerical approximation accuracy vs. (b) number of neurons M in the hidden layer of the branch network; and (c) vs. computational time in seconds.

Table 5
Case study 5: 1D Allen-Cahn phase-field PDE in Eq. (62). We report the mean Squared Error (MSE) and percentiles (median, $5\%-95\%$) of the L^2 approximation error for the test set. We use 2400 training functions. Here, we depict, indicatively, the results with a vanilla DeepONet with 2 hidden layers with $[N, N]$ neurons. We set $N = 5, 10, 20, 40$. DeepONets are trained with 50′000 Adam iterations (with learning rate 0.001 and then 0.0001). We report the RandONets encompassing Johnson-Lindenstrauss (JL) Featured branch network (with $M = 40$ neurons) and the Random Fourier Feature branch Network (RFFN) (with $M = 2000$).

ML-model	MSE	5% L^2	median–L^2	95% L^2	comp. time
DeepONet [5, 5]	1.75E−03	1.57E−01	3.50E−01	7.28E−01	3.20E+03 (GPU)
DeepONet [10, 10]	1.60E−04	6.46E−02	1.04E−01	2.12E−01	3.03E+03 (GPU)
DeepONet [20, 20]	5.62E−05	3.49E−02	5.52E−02	1.28E−01	3.39E+03 (GPU)
DeepONet [40, 40]	3.10E−05	2.39E−02	4.06E−02	9.38E−02	3.49E+03 (GPU)
RandONets–JL (40)	2.42E−04	7.61E−02	1.04E−01	2.77E−01	2.32E−02 (CPU)
RandONets–RFFN (2000)	3.15E−10	1.28E−04	1.60E−04	2.60E−04	3.20E+00 (CPU)

leader in this specific case study. It achieves a remarkably low MSE on the order of 1E−12, demonstrating its superior capability in handling the non-linearities present in this example.

Also for this case study, RandONets, utilizing both JL and RP-FFN random embeddings, demonstrates execution times (training times) that are 3 to 5 order times faster, while achieving L^2 accuracy that is 4 orders of magnitude higher in the case of RandONets-RFFN, and of a similar level of accuracy in the case of JL random embeddings.

4.2.3. Case study 5: 1D Allen-Cahn phase-field PDE

Here we consider the nonlinear evolution operator of the Allen-Cahn equation, described by:

$$v = \frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + (u - u^3) \tag{62}$$

where we set the parameter $\nu = 0.01$. (See Fig. 6 and Table 5.)

The output function can be computed analytically/symbolically based on (54). The parameters $\boldsymbol{w}$, a_0, a_1, $a_2 \sim \mathcal{U}[-0.05, 0.05]$, $\boldsymbol{s} \sim \mathcal{U}[0, 50]$ and $\boldsymbol{c} \sim \mathcal{U}[-1, 1]$, of the RP-FNN based function dataset, as in Eq. (54), to represent the functional space are (element-

wise) uniformly distributed. Here, similarly to case study 4, we select w in a smaller range as higher values may correspond to high second derivatives approaching singularity.

To generate the data, we used 3000 random realizations of the parameters. We set 80% for training and 20% for testing. We report the accuracy w.r.t. the test set in terms of MSE and the percentiles (median, $5\% - 95\%$) of L^2 approximation errors in Fig. 6. As shown, the training of the all RandONets takes approximately less or around 3 seconds and it is performed without iterations. In Table 5, we also report the comparison results with the vanilla DeepONets, denoted as $[N, N]$. We set $N = 5, 10, 20, 40$.

Consistent with our observations in case study 4, linear JL random embeddings exhibit limitations in efficiently approximating the non-linear Allen-Cahn operator. Like in case Study 4, the performance of JL embeddings approaches that of fully trained, entirely non-linear vanilla DeepONets. While DeepONets can achieve a minimum Mean Squared Error (MSE) on the order of 1E−04 and an L^2 error on the order of 1E−01, their performance is not significantly better than the RandONets-JL approach. Once again, the RandONets-RFFN architecture emerges as the superior method. It achieves a remarkably low MSE on the order of 1E−10, demonstrating its capability in handling the non-linearities present in this example.

Finally, RandONets, utilizing both JL and RFFN embeddings, exhibit execution times (training times) that are of 3 to 5 orders faster, while achieving L^2 approximation accuracy that is of 2 orders of magnitude higher when using RFFN embeddings, and of the same level of accuracy when using the linear JL random embeddings.

5. Conclusions

In this work, we presented RandONets to approximate linear and nonlinear operators. Our work builds on three keystones: (a) Linear and nonlinear random embeddings of the input function space, (b) shallow (one-hidden layer) random neural networks, and, (c) tailor-made numerical analysis methods for the solution of a linear ill-posed problem. First, buliding on previous works, we prove that RandOnets are universal approximators. We furthermore assessed their performance by comparing them with vanilla DeepONets on various benchmark problems, including, simple problems of the approximation of the solution operator of ODEs, and linear and nonlinear evolution operators (right-hand-sides) of PDEs. We show that RandOnets outperform the vanilla DeepONets in both computational cost and numerical accuracy by orders of magnitude. In particular, we show that RandONets with JL random embeddings are unbeatable when approximating linear evolution operators of PDEs, reaching almost machine-precision accuracy for aligned data. RandONets with nonlinear random embeddings are on the order of $10^2 - 10^3$ more accurate and 10^3 times faster than the vanilla DeepONets.

Here, we aimed at introducing to the community RandONets. So, an extensive comparison, with more advanced versions of DeepONets and other approaches such as FNOs and others, is beyond the scope of this current work. However, in a following work, we aim at assessing the performance of RandONets considering high-dimensional nonlinear operators, and the approximation of the solution operator of PDEs, thus performing “a comprehensive and fair comparison” of the various machine learning schemes as performed also in [28].

We believe our work will inspire further advancements in the field, encouraging exploration into how numerical analysis can improve the applicability and interpretability of shallow neural networks. This could potentially enable shallow networks to outperform DNNs by several orders of magnitude in both accuracy and computational efficiency for specific scientific machine learning tasks. Towards to this direction, for example, in [77] the authors show that “accurate” optimization of shallow feedforward neural networks can yield superior results compared to training deep neural networks for PDEs.

CRediT authorship contribution statement

Gianluca Fabiani: Conceptualization, Mathematical analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Ioannis G. Kevrekidis:** Methodology, Supervision, Validation, Writing – review & editing. **Constantinos Siettos:** Conceptualization, Mathematical analysis, Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing. **Athanasios N. Yannacopoulos:** Mathematical analysis, Methodology, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

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

Data availability

The `MATLAB` code developed for this study is accessible on the GitHub repository at https://github.com/GianlucaFabiani/RandONets.

Acknowledgements

I.G.K. acknowledges partial support from the US AFOSR FA9550-21-0317 and the US Department of Energy SA22-0052-S001. C.S. acknowledges partial support from the PNRR MUR, projects PE0000013-Future Artificial Intelligence Research-FAIR & CN0000013

CN HPC - National Centre for HPC, Big Data and Quantum Computing. A.N.Y. acknowledges the use of resources from the Stochastic Modelling and Applications Laboratory, AUEB.

References

[1] J. Han, A. Jentzen, E. Weinan, Solving high-dimensional partial differential equations using deep learning, Proc. Natl. Acad. Sci. 115 (2018) 8505–8510.
[2] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, DeepXDE: a deep learning library for solving differential equations, SIAM Rev. 63 (2021) 208–228.
[3] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.
[4] G. Fabiani, F. Calabrò, L. Russo, C. Siettos, Numerical solution and bifurcation analysis of nonlinear partial differential equations with extreme learning machines, J. Sci. Comput. 89 (2021) 44.
[5] F. Calabrò, G. Fabiani, C. Siettos, Extreme learning machine collocation for the numerical solution of elliptic PDEs with sharp gradients, Comput. Methods Appl. Mech. Eng. 387 (2021) 114188.
[6] G. Fabiani, E. Galaris, L. Russo, C. Siettos, Parsimonious physics-informed random projection neural networks for initial value problems of odes and index-1 daes, Chaos 33 (2023).
[7] S. Dong, J. Yang, On computing the hyperparameter of extreme learning machines: algorithm and application to computational pdes, and comparison with classical and high-order finite elements, arXiv preprint, arXiv:2110.14121, 2021.
[8] S. Dong, Z. Li, Local extreme learning machines and domain decomposition for solving linear and nonlinear partial differential equations, Comput. Methods Appl. Mech. Eng. 387 (2021) 114129.
[9] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (2021) 422–440.
[10] 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 (2021) 218–229.
[11] M. Kalia, S.L. Brunton, H.G. Meijer, C. Brune, J.N. Kutz, Learning normal form autoencoders for data-driven discovery of universal, parameter-dependent governing equations, arXiv preprint, arXiv:2106.05102, 2021.
[12] H. Vargas Alvarez, G. Fabiani, N. Kazantzis, C. Siettos, I.G. Kevrekidis, Discrete-time nonlinear feedback linearization via physics-informed machine learning, J. Comput. Phys. 492 (2023) 112408.
[13] D.G. Patsatzis, G. Fabiani, L. Russo, C. Siettos, Slow invariant manifolds of singularly perturbed systems via physics-informed machine learning, arXiv preprint, arXiv:2309.07946, 2023.
[14] H. Vargas Alvarez, G. Fabiani, I.G. Kevrekidis, N. Kazantzis, C. Siettos, Nonlinear discrete-time observers with physics-informed neural networks, arXiv preprint, arXiv:2402.12360, 2024.
[15] M. Raissi, Deep hidden physics models: deep learning of nonlinear partial differential equations, J. Mach. Learn. Res. 19 (2018) 932–955.
[16] S. Lee, M. Kooshkbaghi, K. Spiliotis, C.I. Siettos, I.G. Kevrekidis, Coarse-scale pdes from fine-scale observations via machine learning, Chaos 30 (2020) 013141.
[17] E. Galaris, G. Fabiani, I. Gallos, I. Kevrekidis, C. Siettos, Numerical bifurcation analysis of pdes from lattice Boltzmann model simulations: a parsimonious machine learning approach, J. Sci. Comput. 92 (2022) 1–30.
[18] G. Fabiani, N. Evangelou, T. Cui, J.M. Bello-Rivas, C.P. Martin-Linares, C. Siettos, I.G. Kevrekidis, Task-oriented machine learning assisted surrogates for tipping points of agent-based models, Nat. Commun. 15 (2024) 1–13, https://doi.org/10.1038/s41467-024-48024-7.
[19] S. Lee, Y.M. Psarellis, C.I. Siettos, I.G. Kevrekidis, Learning black- and gray-box chemotactic pdes/closures from agent based Monte Carlo simulation data, J. Math. Biol. 87 (2023) 15.
[20] F. Dietrich, A. Makeev, G. Kevrekidis, N. Evangelou, T. Bertalan, S. Reich, I.G. Kevrekidis, Learning effective stochastic differential equations from microscopic simulations: linking stochastic numerics to deep learning, Chaos 33 (2023) 023121.
[21] R. González-García, R. Rico-Martìnez, I.G. Kevrekidis, Identification of distributed parameter systems: a neural net based approach, Comput. Chem. Eng. 22 (1998) S965–S968.
[22] K. Krischer, R. Rico-Martinez, I. Kevrekidis, H. Rotermund, G. Ertl, J. Hudson, Model identification of a spatiotemporally varying catalytic reaction, AIChE J. 39 (1993) 89–98.
[23] S.Y. Shvartsman, C. Theodoropoulos, R. Rico-Martınez, I. Kevrekidis, E.S. Titi, T. Mountziaris, Order reduction for nonlinear dynamic models of distributed reacting systems, J. Process Control 10 (2000) 177–184.
[24] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, arXiv preprint, arXiv:2010.08895, 2020.
[25] N.B. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A.M. Stuart, A. Anandkumar, Neural operator: learning maps between function spaces with applications to pdes, J. Mach. Learn. Res. 24 (2023) 1–97.
[26] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, A. Stuart, K. Bhattacharya, A. Anandkumar, Multipole graph neural operator for parametric partial differential equations, Adv. Neural Inf. Process. Syst. 33 (2020) 6755–6766.
[27] T. Chen, H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems, IEEE Trans. Neural Netw. 6 (1995) 911–917.
[28] 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. Eng. 393 (2022) 114778.
[29] 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. Eng. 391 (2022) 114587.
[30] P. Jin, S. Meng, L. Lu, Mionet: learning multiple-input operators via tensor product, SIAM J. Sci. Comput. 44 (2022) A3490–A3514.
[31] T. Tripura, S. Chakraborty, Wavelet neural operator for solving parametric partial differential equations in computational mechanics problems, Comput. Methods Appl. Mech. Eng. 404 (2023) 115783, https://doi.org/10.1016/j.cma.2022.115783, https://www.sciencedirect.com/science/article/pii/S0045782522007393.
[32] V. Fanaskov, I.V. Oseledets, Spectral neural operators, Dokl. Math. 108 (Suppl 2) (2023) S226–S232, Springer.
[33] K. Azizzadenesheli, N. Kovachki, Z. Li, M. Liu-Schiaffini, J. Kossaifi, A. Anandkumar, Neural operators for accelerating scientific simulations and design, Nat. Rev. Phys. (2024) 1–9.
[34] S. Wang, H. Wang, P. Perdikaris, Learning the solution operator of parametric partial differential equations with physics-informed deeponets, Sci. Adv. 7 (2021) eabi8605, https://doi.org/10.1126/sciadv.abi8605, https://www.science.org/doi/abs/10.1126/sciadv.abi8605, arXiv: https://www.science.org/doi/pdf/10.1126/sciadv.abi8605.
[35] M.V. de Hoop, D.Z. Huang, E. Qian, A.M. Stuart, The cost-accuracy trade-off in operator learning with neural networks, arXiv preprint, arXiv:2203.13181, 2022.
[36] S. Venturi, T. Casey, Svd perspectives for augmenting deeponet flexibility and interpretability, Comput. Methods Appl. Mech. Eng. 403 (2023) 115718.
[37] S. Goswami, A. Bora, Y. Yu, G.E. Karniadakis, Physics-informed deep neural operator networks, in: Machine Learning in Modeling and Simulation: Methods and Applications, Springer, 2023, pp. 219–254.
[38] W.B. Johnson, J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, Contemp. Math. 26 (1984) 189–206.
[39] A. Rahimi, B. Recht, Random features for large-scale kernel machines, Adv. Neural Inf. Process. Syst. 20 (2007).

[40] A. Rahimi, B. Recht, Uniform approximation of functions with random bases, in: 2008 46th Annual Allerton Conference on Communication, Control, and Computing, IEEE, 2008, pp. 555–561.
[41] A. Rahimi, B. Recht, Weighted sums of random kitchen sinks: replacing minimization with randomization in learning, in: Nips, Citeseer, 2008, pp. 1313–1320.
[42] A.N. Gorban, I.Y. Tyukin, D.V. Prokhorov, K.I. Sofeikov, Approximation with random bases: pro et contra, Inf. Sci. 364 (2016) 129–145.
[43] I.G. Kevrekidis, C.W. Gear, J.M. Hyman, P.G. Kevrekidis, O. Runborg, C. Theodoropoulos, Equation-free, coarse-grained multiscale computation: enabling mocroscopic simulators to perform system-level analysis, Commun. Math. Sci. 1 (2003) 715–762, https://doi.org/10.4310/cms.2003.v1.n4.a5.
[44] P.G. Papaioannou, R. Talmon, I.G. Kevrekidis, C. Siettos, Time-series forecasting using manifold learning, radial basis function interpolation, and geometric harmonics, Chaos 32 (2022) 083113.
[45] I.K. Gallos, D. Lehmberg, F. Dietrich, C. Siettos, Data-driven modelling of brain activity using neural networks, diffusion maps, and the Koopman operator, Chaos 34 (2024).
[46] J.C. Patra, R.N. Pal, B. Chatterji, G. Panda, Identification of nonlinear dynamic systems using functional link artificial neural networks, IEEE Trans. Syst. Man Cybern., Part B, Cybern. 29 (1999) 254–262.
[47] C. Siettos, C. Kiranoudis, G. Bafas, Advanced control strategies for fluidized bed dryers, Dry. Technol. 17 (1999) 2271–2291.
[48] C.I. Siettos, G.V. Bafas, A.G. Boudouvis, Truncated Chebyshev series approximation of fuzzy systems for control and nonlinear system identification, Fuzzy Sets Syst. 126 (2002) 89–104.
[49] N.H. Nelsen, A.M. Stuart, The random feature model for input-output maps between Banach spaces, SIAM J. Sci. Comput. 43 (2021) A3212–A3243.
[50] Z. Zhang, L. Wing Tat, H. Schaeffer, Belnet: basis enhanced learning, a mesh-free neural operator, Proc. R. Soc. A 479 (2023) 20230043.
[51] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst. 2 (1989) 303–314.
[52] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (1989) 359–366.
[53] A.R. Barron, Universal approximation bounds for superpositions of a sigmoidal function, IEEE Trans. Inf. Theory 39 (1993) 930–945.
[54] M. Leshno, V.Y. Lin, A. Pinkus, S. Schocken, Multilayer feedforward networks with a nonpolynomial activation function can approximate any function, Neural Netw. 6 (1993) 861–867.
[55] A. Pinkus, Approximation theory of the MLP model, Acta Numer. 8 (1999) 143–195.
[56] W. Schmidt, M. Kraaijveld, R. Duin, Feedforward neural networks with random weights, in: Proceedings 11th IAPR International Conference on Pattern Recognition. Vol. II. Conference B: Pattern Recognition Methodology and Systems, IEEE Computer Society Press, 1992, pp. 1–4.
[57] Y.-H. Pao, Y. Takefuji, Functional-link net computing: theory, system architecture, and functionalities, Computer 25 (1992) 76–79.
[58] B. Igelnik, Y.-H. Pao, Stochastic choice of basis functions in adaptive function approximation and the functional-link net, IEEE Trans. Neural Netw. 6 (1995) 1320–1329.
[59] H. Jaeger, The “echo state” approach to analysing and training recurrent neural networks-with an erratum note, Technology GMD Technical Report 148, German National Research Center for Information, Bonn, Germany, 2001, 13.
[60] H. Jaeger, Adaptive nonlinear system identification with echo state networks, Adv. Neural Inf. Process. Syst. 15 (2002) 609–616.
[61] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (2006) 489–501.
[62] F. Rosenblatt, Perceptions and the Theory of Brain Mechanisms, Spartan Books, 1962.
[63] S. Gallant, D. Smith, Random cells: an idea whose time has come and gone... and come again?, in: IEEE International Conference on Neural Networks, vol. 2, IEEE, 1987, pp. 671–678.
[64] S. Scardapane, D. Wang, Randomness in neural networks: an overview, Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 7 (2017) e1200.
[65] G. Fabiani, Random projection neural networks of best approximation: convergence theory and practical applications, arXiv preprint, arXiv:2402.11397, 2024.
[66] R.D. Fierro, G.H. Golub, P.C. Hansen, D.P. O’Leary, Regularization by truncated total least squares, SIAM J. Sci. Comput. 18 (1997) 1223–1241.
[67] B. Schölkopf, The kernel trick for distances, Adv. Neural Inf. Process. Syst. 13 (2000).
[68] B. Schölkopf, A. Smola, K.-R. Müller, Kernel principal component analysis, in: Artificial Neural Networks—ICANN’97: 7th International Conference Lausanne, Switzerland, October 8–10, 1997 Proceedings, Springer, 2005, pp. 583–588.
[69] B. Adcock, N. Dexter, The gap between theory and practice in function approximation with deep neural networks, SIAM J. Math. Data Sci. 3 (2021) 624–655.
[70] Y. Liao, S.-C. Fang, H.L. Nuttle, Relaxed conditions for radial-basis function networks to be universal approximators, Neural Netw. 16 (2003) 1019–1028.
[71] J. Park, I.W. Sandberg, Universal approximation using radial-basis-function networks, Neural Comput. 3 (1991) 246–257.
[72] J. Park, I.W. Sandberg, Approximation and radial-basis-function networks, Neural Comput. 5 (1993) 305–316.
[73] G.H. Golub, P.C. Hansen, D.P. O’Leary, Tikhonov regularization and total least squares, SIAM J. Matrix Anal. Appl. 21 (1999) 185–194.
[74] P.D. Hough, S.A. Vavasis, Complete orthogonal decomposition for weighted least squares, SIAM J. Matrix Anal. Appl. 18 (1997) 369–392.
[75] Y. Lu, R. Maulik, T. Gao, F. Dietrich, I.G. Kevrekidis, J. Duan, Learning the temporal evolution of multivariate densities via normalizing flows, Chaos 32 (2022) 033121.
[76] Y.-H. Pao, G.-H. Park, D.J. Sobajic, Learning and generalization characteristics of the random vector functional-link net, Neurocomputing 6 (1994) 163–180.
[77] P.A. Mistani, S. Pakravan, R. Ilango, F. Gibou, Jax-dips: neural bootstrapping of finite discretization methods and application to elliptic problems with discontinuities, J. Comput. Phys. 493 (2023) 112480.