Improved architectures and training algorithms for deep operator networks

Preprint · October 2021

CITATIONS
0

READS
85

3 authors:

Sifan Wang
University of Pennsylvania
13 PUBLICATIONS **101** CITATIONS
SEE PROFILE

Hanwen Wang
University of Pennsylvania
7 PUBLICATIONS **26** CITATIONS
SEE PROFILE

Paris Perdikaris
University of Pennsylvania
82 PUBLICATIONS **4,817** CITATIONS
SEE PROFILE

IMPROVED ARCHITECTURES AND TRAINING ALGORITHMS FOR DEEP OPERATOR NETWORKS

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

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

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

October 4, 2021

ABSTRACT

Operator learning techniques have recently emerged as a powerful tool for learning maps between infinite-dimensional Banach spaces. Trained under appropriate constraints, they can also be effective in learning the solution operator of partial differential equations (PDEs) in an entirely self-supervised manner. In this work we analyze the training dynamics of deep operator networks (DeepONets) through the lens of Neural Tangent Kernel (NTK) theory, and reveal a bias that favors the approximation of functions with larger magnitudes. To correct this bias we propose to adaptively re-weight the importance of each training example, and demonstrate how this procedure can effectively balance the magnitude of back-propagated gradients during training via gradient descent. We also propose a novel network architecture that is more resilient to vanishing gradient pathologies. Taken together, our developments provide new insights into the training of DeepONets and consistently improve their predictive accuracy by a factor of 10-50x, demonstrated in the challenging setting of learning PDE solution operators in the absence of paired input-output observations. All code and data accompanying this manuscript are publicly available at https://github.com/PredictiveIntelligenceLab/ImprovedDeepONets.

***K*eywords** Deep learning · Partial differential equations · Computational physics

1 Introduction

Motivation: Operator learning techniques have recently attracted significant attention thanks to their effectiveness and favorable complexity in approximating maps between infinite-dimensional Banach spaces [1, 2, 3]. Techniques such as deep operator networks (DeepONets) [4], the family of neural operator methods [5], and operator-valued kernel methods [6, 7] have demonstrated promise in building fast surrogate models for emulating complex physical processes, opening new avenues for sampling, inference and uncertainty quantification in very high-dimensional parameter spaces. More recently, Wang *et al.* [8, 9] introduced physics-informed DeepONets; a self-supervised framework for learning the solution operator of parametric partial differential equations (PDEs), even in the absence of labelled training data. These new approaches allow deep neural networks to seamlessly generalize across different scenarios (e.g. initial/boundary

conditions, random inputs, different geometries, etc.), making it easier to model complex physical and engineering systems, and to do so orders of magnitude faster.

Open challenges and related work: Network initialization and data normalization are known to be extremely important factors in facilitating the effectiveness of deep neural networks. Effective heuristics such as the Glorot and He initialization schemes [10, 11], batch and weight normalization [12, 13], and data standardization [14] have been become standard practice in training neural networks for supervised learning tasks in finite-dimensional vector spaces. The main goal of such techniques is to reduce inter-dependencies between examples in the training data-set, ultimately leading to more efficient back-propagation and faster convergence of the training loss under gradient descent. However, it is still unclear whether such techniques can be effective or need to be revised in the context of learning operators in infinite-dimensional function spaces. For instance, Di Leoni *et. al.* [15] observed that DeepONets are biased towards approximating target functions with larger magnitudes, and proposed to address this by using the L^2 norm of each output function to normalize the training data. A similar approach was adopted by Li *et. al.* [16], where the relative L^2 norm was used as a training objective, instead of the commonly used mean square error. Although the authors do not discuss the motivation behind this choice, here we conjecture that it serves the exact same purpose: mitigate unbalanced gradients during back-propagation that bias the model towards approximating target functions with larger magnitudes. Here we must notice that, while it is evident that such heuristics are crucial, they may not always be applicable and effective in practice. For instance, if only sparse measurements of the target functions are available one cannot accurately estimate their norm and employ the aforementioned normalization. Moreover, in a semi-supervised or self-supervised setting (as is the case for physics-informed DeepONets [8, 9]), the training data-set may contain no measurements of the target functions at all. Therefore, there is a pressing need to develop a deeper mathematical understanding of the training dynamics of deep operator networks, and use it to guide the development of new effective and robust architectures and training algorithms.

Contributions of this work: Motivated by prior work on physics-informed neural networks [17, 18, 19, 20], in this work we employ Neural Tangent Kernel (NTK) theory [21, 22, 23] to investigate the training dynamics of DeepONets and physics-informed DeepONets. Our analysis provides theoretical insight on the magnitude bias of DeepONets observed by Di Leoni *et. al.* [15], and provides a principled framework for re-weighting the importance of each training example, even in the absence of any measurements for the target output functions. This results into an adaptive learning rate annealing scheme that can continuously interpolate between balancing the magnitude of back-propagated gradients during training and balancing the convergence rate of the training loss associated with each individual example. Motivated by these findings, we also put forth a novel DeepONet architecture that is more resilient to vanishing gradient pathologies and is demonstrated to outperform conventional DeepONet for all benchmarks considered in this work. Taken together, the proposed developments introduce a 10-50x improvement in the predictive accuracy of DeepONets, which is demonstrated in the challenging setting of learning PDE solution operators in an entirely self-supervised manner (i.e. in the absence of any paired input-output observations). The computational infrastructure developed in this work can have a broad technical impact in enhancing the performance of DeepONets and physics-informed DeepONets in emulating PDE systems, paving a new way to accelerating scientific modeling of complex non-linear, non-equilibrium processes in science and engineering.

Structure of this paper: In section 2, we provide an overview of the Neural Tangent Kernel (NTK) theory following the original formulation of Jacot *et. al.* [21], the DeepONet architecture [4], and the physics-informed DeepONet framework [8]. Section 3.1 introduces a simple benchmark problem that will guide our analysis and highlight the bias of DeepONets towards learning output functions with a larger magnitude. To address this issue, in section 3.2 we analyze the training dynamics of physics-informed DeepONets through the lens of NTK theory. Based on it, we propose an NTK-guided weighting scheme along with a novel DeepONet architecture in sections 3.3-3.4. To validate the effectiveness the proposed techniques, in section 4 we present a series of comprehensive numerical studies across a range of representative benchmarks for which conventional physics-informed DeepONets struggle. Finally, section 5 concludes with a discussion of our main findings, potential pitfalls, and shortcomings, as well as future research directions emanating from this study.

2 Preliminaries

2.1 The Neural Tangent Kernel (NTK) of a fully-connected neural network

The Neural Tangent Kernel (NTK) is a recently proposed theoretical framework for establishing provable convergence and generalization guarantees for wide (over-parametrized) neural networks [21, 22, 23]. In this section, we will present a brief introduction to NTK theory and its connection to spectral bias [24, 25, 26] in the context of training multi-layer

perceptron networks (MLPs). We begin by considering a scalar-valued MLP network $f(\boldsymbol{x}, \boldsymbol{\theta})$ with inputs $\boldsymbol{x} \in \mathbb{R}^{d_0}$ recursively defined by,

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{W}^{(L+1)} \boldsymbol{f}^{(L)}(\boldsymbol{x}) + \boldsymbol{b}^{(L+1)}, \tag{2.1}$$

$$\boldsymbol{f}^{(l)}(\boldsymbol{x}) = \frac{1}{\sqrt{d_{l-1}}} \boldsymbol{W}^{(l)} \sigma(\boldsymbol{f}^{(l-1)}(\boldsymbol{x})) + \boldsymbol{b}^{(l)} \in \mathbb{R}^{d_l}, \text{ for } k = 2, \ldots, L, \tag{2.2}$$

$$\boldsymbol{f}^{(1)}(\boldsymbol{x}) = \frac{1}{\sqrt{d_0}} \boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{b}^{(1)}, \tag{2.3}$$

where $\boldsymbol{\theta} = \{\boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)}\}_{k=1}^{L+1}$ denotes all trainable parameters of the network, with all weights and biases initialized by sampling a Gaussian distribution $\mathcal{N}(0, 1)$. Now we can define the NTK as

$$k_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}') = \left\langle \frac{df(\boldsymbol{x}, \boldsymbol{\theta})}{d\boldsymbol{\theta}}, \frac{df(\boldsymbol{x}', \boldsymbol{\theta})}{d\boldsymbol{\theta}} \right\rangle. \tag{2.4}$$

The seminal work of Jacot *et. al.* [21] shows that the NTK converges to a deterministic kernel under the infinite width limit and training via gradient descent with an infinitesimally small learning rate. As a consequence the training dynamics of a sufficiently wide MLP network behave like a linearized model governed by the NTK.

Now consider a set of training data $\{\boldsymbol{X}_{\text{train}}, \boldsymbol{Y}_{\text{train}}\}$ with $\boldsymbol{X}_{\text{train}} = (\boldsymbol{x}_i)_{i=1}^N$, $\boldsymbol{Y}_{\text{train}} = (y_i)_{i=1}^N$, and the ℓ^2 regression loss $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N |f(\boldsymbol{x}_i, \boldsymbol{\theta}) - y_i|^2$. Under the asymptotic conditions stated in Lee *et. al.* [27], one may derive

$$\frac{df\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right)}{dt} \approx -\boldsymbol{K} \cdot \left(f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right) - \boldsymbol{Y}_{\text{train}}\right), \tag{2.5}$$

where $\boldsymbol{\theta}$ denotes the parameters of the network at gradient descent iteration t and $f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right) = \left(f\left(\boldsymbol{x}_i, \boldsymbol{\theta}(t)\right)_{i=1}^N\right)$ is a vector in $\mathbb{R}^N$. Besides, $\boldsymbol{K} \in \mathbb{R}^{N \times N}$ is the NTK matrix whose entries are defined by

$$\boldsymbol{K}_{ij} = k_{\boldsymbol{\theta}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left\langle \frac{df(\boldsymbol{x}_i, \boldsymbol{\theta})}{d\boldsymbol{\theta}}, \frac{df(\boldsymbol{x}_j, \boldsymbol{\theta})}{d\boldsymbol{\theta}} \right\rangle, \quad \text{for } i, j = 1, 2, \ldots, N. \tag{2.6}$$

One can then obtain that

$$f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right) \approx e^{-\boldsymbol{K}t} f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(0)\right) + \left(I - e^{-\boldsymbol{K}t}\right) \cdot \boldsymbol{Y}_{\text{train}}. \tag{2.7}$$

From the NTK definition, it is easy to see that $\boldsymbol{K}$ is positive semi-definite. Therefore, there exists an orthogonal matrix $\boldsymbol{Q} = [\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_N]$ such that $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$, where $\boldsymbol{\Lambda}$ is a diagonal matrix with diagonal entries λ_i being the eigenvalue corresponding to the eigenvector $\boldsymbol{q}_i$. Note that $e^{-\boldsymbol{K}t} = \boldsymbol{Q} e^{-\boldsymbol{\Lambda}t} \boldsymbol{Q}^T$, yielding

$$\boldsymbol{Q}^T \left(f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right) - \boldsymbol{Y}_{\text{train}}\right) \approx e^{-\boldsymbol{\Lambda}t} \boldsymbol{Q}^T \left(f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(0)\right) - \boldsymbol{Y}_{\text{train}}\right). \tag{2.8}$$

Therefore,

$$\begin{bmatrix} \boldsymbol{q}_1^T \\ \boldsymbol{q}_2^T \\ \vdots \\ \boldsymbol{q}_N^T \end{bmatrix} \left(f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(t)\right) - \boldsymbol{Y}_{\text{train}}\right) \approx \begin{bmatrix} e^{-\lambda_1 t} & & & \\ & e^{-\lambda_2 t} & & \\ & & \ddots & \\ & & & e^{-\lambda_N t} \end{bmatrix} \begin{bmatrix} \boldsymbol{q}_1^T \\ \boldsymbol{q}_2^T \\ \vdots \\ \boldsymbol{q}_N^T \end{bmatrix} \left(f\left(\boldsymbol{X}_{\text{train}}, \boldsymbol{\theta}(0)\right) - \boldsymbol{Y}_{\text{train}}\right). \tag{2.9}$$

The above equation reveals that the i-th term of the left-hand side will converge to 0 at the speed of $e^{-\lambda_i t}$ if $\lambda_i > 0$. In other words, the eigenvalues of the NTK characterize the convergence rate of the total training error. We remark that this conclusion holds for any network architecture in the NTK regime [18, 28].

2.2 DeepONets and Physics-informed DeepONets

Deep operator networks (DeepONets) [4] are a specialized deep learning architecture that aims to learn abstract nonlinear operators between infinite-dimensional function spaces. This network architecture is inspired and validated by the rigorous universal approximation theorem for operators [29, 4]. Now we present a brief overview of DeepONets in the setting of learning the solution operator of parametric partial differential equations (PDEs) [8]. Let $(\mathcal{U}, \mathcal{V}, \mathcal{S})$ be a triplet of abstract function spaces and $\mathcal{N} : \mathcal{U} \times \mathcal{S} \to \mathcal{V}$ be a linear or nonlinear differential operator. We consider general parametric PDEs of the form

$$\mathcal{N}(\boldsymbol{u}, \boldsymbol{s}) = 0, \tag{2.10}$$

subject to boundary conditions

$$\mathcal{B}(\boldsymbol{u}, \boldsymbol{s}) = 0, \tag{2.11}$$

where $\boldsymbol{u} \in \mathcal{U}$ denotes the parameters (i.e. input functions), and $\boldsymbol{s} \in \mathcal{S}$ denotes an unknown latent function that is governed by the PDE system in equation 2.10. Moreover, $\mathcal{B}[\cdot]$ denotes a boundary conditions operator that enforces any Dirichlet, Neumann, Robin, or periodic boundary conditions. For time-dependent problems, we consider time t as an extra component of $\boldsymbol{x}$, and Ω is extended to contain the temporal domain. In that case, initial conditions can be simply treated as a special type of boundary condition defined in the joint spatio-temporal domain. Assume that, for any $\boldsymbol{u} \in \mathcal{U}$, there exists an unique solution $\boldsymbol{s} = \boldsymbol{s}(\boldsymbol{u}) \in \mathcal{U}$ to equation (2.10), subject to appropriate initial and boundary conditions. Then, one can define the PDE solution operator $G : \mathcal{U} \to \mathcal{S}$ as

$$G(\boldsymbol{u}) = \boldsymbol{s}(\boldsymbol{u}). \tag{2.12}$$

Now we can proceed by representing the solution map with a DeepONet $G_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta}$ denotes all trainable parameters of the DeepOnet network. As illustrated in Fig 1, the DeepONet is composed of two separate neural networks referred as the "branch" and "trunk" networks, respectively. The branch network takes a function $\boldsymbol{u}$ as input and returns a features embedding $[b_1, b_2, \ldots, b_q]^T \in \mathbb{R}^q$ as output, where $\boldsymbol{u} = [\boldsymbol{u}(\boldsymbol{x}_1), \boldsymbol{u}(\boldsymbol{x}_2), \ldots, \boldsymbol{u}(\boldsymbol{x}_m)]$ represents a function $\boldsymbol{u} \in \mathcal{U}$ evaluated at a collection of fixed locations $\{\boldsymbol{x}_i\}_{i=1}^m$. The trunk network takes the continuous coordinates $\boldsymbol{y}$ as inputs, and outputs a features embedding $[t_1, t_2, \ldots, t_q]^T \in \mathbb{R}^q$. The final DeepONet output is obtained by merging the outputs of branch and trunk networks together via an inner product. Mathematically, one can express the DeepONet $G_{\boldsymbol{\theta}}$ representation of a function $\boldsymbol{u}$ evaluated at $\boldsymbol{y}$ as

$$G_{\boldsymbol{\theta}}(\boldsymbol{u})(\boldsymbol{y}) = \sum_{k=1}^{q} \underbrace{b_k\left(\boldsymbol{u}\left(\boldsymbol{x}_1\right), \boldsymbol{u}\left(\boldsymbol{x}_2\right), \ldots, \boldsymbol{u}\left(\boldsymbol{x}_m\right)\right)}_{\text{branch}} \underbrace{t_k(\boldsymbol{y})}_{\text{trunk}}, \tag{2.13}$$

From this definition, one may note that the output of a DeepONet is a scalar function. However, sometimes we may require a DeepONet to return a vector-valued function. To resolve this issue, Wang *et. al* [9] modified the original forward pass as follows

$$G_{\boldsymbol{\theta}}^{(i)}(\boldsymbol{u})(\boldsymbol{y}) = \sum_{k=q_{i-1}+1}^{q_i} \underbrace{b_k\left(\boldsymbol{u}\left(\boldsymbol{x}_1\right), \boldsymbol{u}\left(\boldsymbol{x}_2\right), \ldots, \boldsymbol{u}\left(\boldsymbol{x}_m\right)\right)}_{\text{branch}} \underbrace{t_k(\boldsymbol{y})}_{\text{trunk}}, \quad i = 1, \ldots, n, \tag{2.14}$$

where $0 = q_0 < q_1 < \cdots < q_n = q$. This simple modification enables a DeepONet to output an n-dimensional vector.

Now let us define

$$\mathcal{L}(\boldsymbol{u}, \boldsymbol{\theta}) = \frac{1}{P} \sum_{j=1}^{P} \left| G_{\boldsymbol{\theta}}(\boldsymbol{u})(\boldsymbol{y}_j) - s(\boldsymbol{y}_j) \right|^2, \tag{2.15}$$

where $\{s(y_j)\}_{j=1}^P$ denotes the associated PDE solution of equation (2.10) evaluated at P locations $\{\boldsymbol{y}_j\}_{j=1}^P$ in the domain of $G(u)$. Then a DeepONet model can be trained by minimizing the following loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta}), \tag{2.16}$$

where

$$\mathcal{L}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta}) = \frac{1}{P} \sum_{j=1}^{P} \left| G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_j^{(i)}) - s^{(i)}(\boldsymbol{y}_j^{(i)}) \right|^2 \tag{2.17}$$

$$= \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_j^{(i)}) - s^{(i)}(\boldsymbol{y}_j^{(i)}) \right|^2. \tag{2.18}$$

We remark that that the locations $\boldsymbol{y}$ may vary for different input samples $\boldsymbol{u}$.

In follow up work, Wang *et. al.* [8] developed physics-informed DeepONets, which leverages automatic differentiation [30] to constrain the outputs of a DeepONet model to satisfy a given governing PDE. Specifically, a physics-informed DeepONet can be trained by minimizing

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{\text{operator}}(\boldsymbol{\theta}) + \mathcal{L}_{\text{physics}}(\boldsymbol{\theta}), \tag{2.19}$$

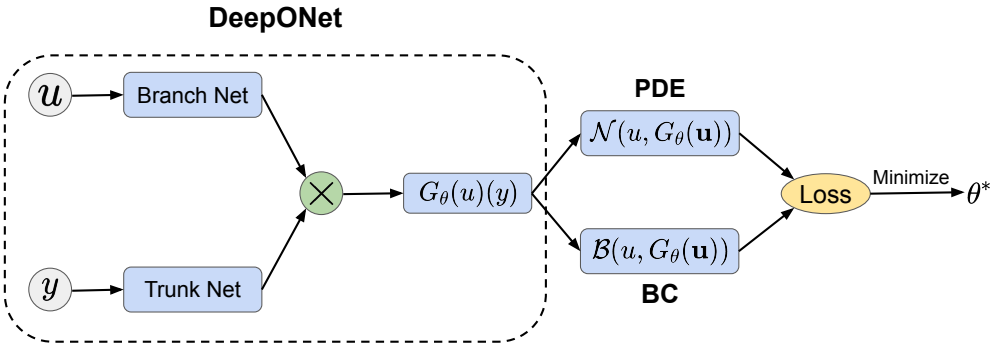

Figure 1: *Physics-informed DeepONets:* The DeepONet architecture [4] consists of two sub-networks referred as the branch network and the trunk network, which extract latent representations of input functions $\boldsymbol{u}$ and input coordinates $\boldsymbol{y}$ at which the output functions are evaluated, respectively. A continuously differentiable representation of the output functions is then obtained by merging the outputs of each sub-network via a dot product. Automatic differentiation can then be employed to formulate appropriate regularization mechanisms for biasing the DeepONet outputs to satisfy a given system of PDEs.

where $\mathcal{L}_{\text{operator}}(\boldsymbol{\theta})$ can be defined exactly the same as in equation (2.16), which aims to fit available experimental data and numerical estimations. Notice that this enables one to train DeepONet models even if no paired input-output observations are available, except for assuming knowledge of the initial and boundary conditions 2.11. Then one can define

$$\sum_{i=1}^{N} \mathcal{L}(\boldsymbol{u}, \boldsymbol{\theta}) = \frac{1}{P} \sum_{j=1}^{P} \left| \mathcal{B}(u(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u})(\boldsymbol{y}_j)) \right|^2 . \tag{2.20}$$

For each input sample $\boldsymbol{u}$, $\{\boldsymbol{x}_j\}_{j=1}^P$ and $\{\boldsymbol{y}_j\}_{j=1}^P$ denote two sets of points that are randomly sampled from the domain of $\boldsymbol{u}$ and the boundary of $G(\boldsymbol{u})$, respectively, for imposing boundary conditions. Consequently we can define

$$\mathcal{L}_{\text{operator}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta}) = \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| \mathcal{B}(u^{(i)}(\boldsymbol{x}_j^{(i)}), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_j^{(i)})) \right|^2 , \tag{2.21}$$

and

$$\mathcal{L}_{\text{physics}}(\boldsymbol{u}, \boldsymbol{\theta}) = \frac{1}{Q} \sum_{j=1}^{Q} \left| \mathcal{N}(u(\boldsymbol{x}_{r,j}), G_{\boldsymbol{\theta}}(\boldsymbol{u})(\boldsymbol{y}_{r,j})) \right|^2 , \tag{2.22}$$

where $\{\boldsymbol{x}_{r,j}\}_{j=1}^Q$ and $\{\boldsymbol{y}_{r,j}\}_{j=1}^Q$ denote two sets of collocation points that are randomly sampled from the domain of u and $G(\boldsymbol{u})$, respectively, for enforcing the set of parametric PDE constraints described equation (2.10). It also follows that

$$\mathcal{L}_{\text{physics}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{\text{physics}}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta}) \tag{2.23}$$

$$= \frac{1}{NQ} \sum_{i=1}^{N} \sum_{j=1}^{Q} \left| \mathcal{N}(u^{(i)}(\boldsymbol{x}_{r,j}^{(i)}), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_{r,j}^{(i)})) \right|^2 \tag{2.24}$$

As shown in Wang *et. al.* [8, 9], it is worth pointing out that physics-informed DeepONets are capable of learning the solution operator of parametric PDEs in an entirely self-supervised manner, i.e. without any paired input-output observations.

3 Methods

3.1 Gradients pathologies in DeepONets

Although DeepONets and physics-informed DeepONets have demonstrated a series of promising results in learning nonlinear operators and solving multi-physics problems [4, 31, 15, 8, 9], it is worth noting that the original formulation

may have difficulties in tackling cases with multi-scale input parameters. To this end, poor approximations can arise even in the simplest possible setting. As an pedagogical example, let us consider the following one-dimensional initial value problem

$$\frac{ds(x)}{dx} = u(x), \quad x \in [0,1], \tag{3.1}$$

$$s(0) = 0. \tag{3.2}$$

Here, our goal is to learn the solution operator G mapping the forcing term $u(x)$ to the corresponding solution $s(x)$ in $[0,1]$. Indeed, G is the so-called anti-derivative operator with an explicit expression of

$$G : u(x) \longrightarrow s(x) = s(0) + \int_0^x u(t)dt, \quad x \in [0,1]. \tag{3.3}$$

To assess the performance of DeepONets in learning the anti-derivative operator, we generate a training data-set by sampling $N = 10^4$ forcing terms u from a Gaussian random field (GRF) with a fixed length scale $l = 0.2$, and a random output scale $k \in [0.01, 100]$. To balance the training data-set, we set $k = 10^\alpha$ where α is sampled from a uniform distribution $\mathcal{U}(-2,2)$. The associated ODE solutions s are obtained by integrating the ODE 3.1 using an explicit Runge-Kutta method (RK45) [32]. Input functions u are represented via point-wise evaluations at a set of $m = 100$ fixed sensors $\{x_i\}_{i=1}^m$ evenly spaced in $[0,1]$. For each observed pair (u,s), we randomly select $P = 1$ observation of $s(\cdot)$ in $[0,1]$.

We now proceed by approximating the anti-derivative operator with a DeepONet $G_{\boldsymbol{\theta}}$, where the branch and trunk networks are two 3-layer MLP networks with ReLU activation functions and 100 neurons per hidden layer. We train this model by minimizing the loss function of equation (2.16) for 4×10^4 iterations of gradient descent using the Adam optimizer [33]. Figure 2 shows the predicted solutions for representative input samples with different output scales. One may observe that the model prediction corresponding to the large output scale is more accurate than the one corresponding to the small output scale. To further validate this observation, we compute the relative L^2 errors over $N = 10^3$ random input functions sampled from a GRF with a different fixed output scale k and length scale $l = 0.2$. The results are summarized in Table 1. Intuitively, since the DeepONet is trained on a balanced training data-set, one would expect that the trained model is capable of achieving approximately the same predictive accuracy across all input samples. However, this is contradicted with the observed positive correlation between the test error and the output scale k reported in Table 1.

To investigate the inner mechanism that triggers this interesting behavior, we draw motivation from Wang *et. al.* [17] and monitor the back-propagated gradients with respect to the network parameters during training. Rather than tracking the gradients of the aggregate loss, we monitor the gradients of each individual term in $\mathcal{L}(\boldsymbol{u}, \boldsymbol{\theta})$. In Figure 3, we present a histogram of the gradients $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{u}, \boldsymbol{\theta})$ for three representative input functions $\boldsymbol{u}$ with different output scales, at the end of training. One key observation is that the gradients corresponding to small output scales are sharply concentrated at the origin and overall attain significantly smaller values than the gradients corresponding to large output scales. In fact, if we take a closer look at the definition of the loss function 2.16, then (for $P = 1$) the corresponding update rule of gradient descent is given by

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \frac{1}{N}\sum_{i=1}^{N} \frac{d\mathcal{L}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta})}{d\boldsymbol{\theta}} = \frac{2}{N}\sum_{i=1}^{N} (G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(y_j^{(i)}) - s^{(i)}(y_j^{(i)})) \frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(y^{(i)})}{d\boldsymbol{\theta}}. \tag{3.4}$$

From this we can deduce that for any input sample $\boldsymbol{u}$

$$\|G_{\boldsymbol{\theta}}(\boldsymbol{u}) - s\| \propto \|s\|, \propto \|u\|$$

where $s(y) = s(0) + \int_0^y u(x)dx$ denotes the associated ODE solution. As a result, the solutions corresponding to large-value input functions yield large changes to the loss function and thus large magnitudes in the corresponding back-propagated gradients during training. For this specific example, it is no surprise that minimizing the mean square error corresponding to large-value input samples dominates the total training process, and, consequently, our trained model is biased towards yielding accurate solutions corresponding to input samples with large output scales. Therefore, it is natural to consider assigning some weight to each term of the loss function to balance the back-propagated gradients, i.e

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \lambda_i \mathcal{L}(\boldsymbol{u}^{(i)}, \boldsymbol{\theta}) \tag{3.5}$$

Here we must point out that this issue has been also noticed by Di Leoni *et. al.* [15], and the authors used $\lambda_i = \frac{1}{\|s^{(i)}\|_\infty}$ as weights to mitigate it. However, this solution is inapplicable to physics-informed DeepONets [8], not only due to

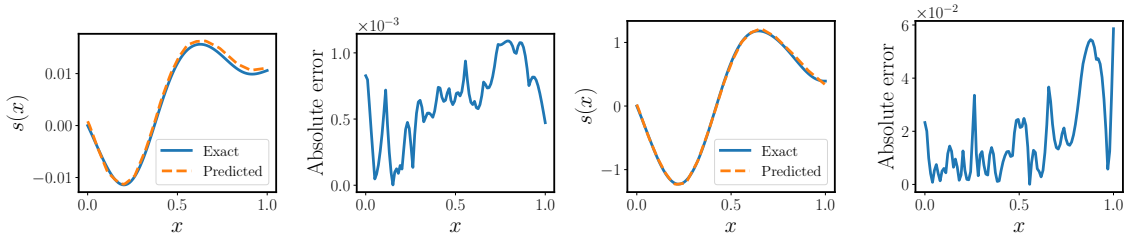

Figure 2: *Anti-derivative operator:* Exact solution versus the predicted solution of a trained physics-informed DeepONet for the same input sample with different output scale $k = 0.01$ (left) and $k = 100$ (right).

Output scale	$k = 0.01$	$k = 0.1$	$k = 1$	$k = 10$	$k = 100$
Rel. L^2 error	$3.29\% \pm 3.01\%$	$1.73\% \pm 1.40\%$	$1.36\% \pm 1.09\%$	$1.27\% \pm 1.01\%$	$1.17\% \pm 0.86\%$

Table 1: *Anti-derivative operator:* Mean and standard deviation of the relative L^2 errors over the test data sampled from a GRF with different output scale $k \in [10^{-2}, 10^2]$.

the absence of paired input-output solution measurements, but also because the interplay between the operator loss $\mathcal{L}_{\text{operator}}$ and the physics loss $\mathcal{L}_{\text{physics}}$ is not taken into consideration. As illustrated in [8, 9], the latter generally plays an important role in the loss convergence and the predictive accuracy of a DeepONet model. In the following section, we will propose a novel algorithm that assigns adaptive weights to the loss function via the lens of NTK theory, which is capable of tackling all the observed issues in a unified manner.

3.2 NTK analysis of physics-informed DeepONets

In this section, we employ the NTK framework to analyze and understand the training dynamics of physics-informed DeepONets, in the specific context of learning the solution operator for a given parametric PDE system (see equations (2.10) - (2.11) in section 2.2). Without loss of generality, here we will assume $P = Q = R$ and rewrite the loss function of equation (2.19) by combining all the individual terms in one summation by rearranging notation and indices as

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N^*} \sum_{k=1}^{N^*} \left| T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right|^2 \tag{3.6}$$

where $N^* = 2NR$, and for every k, $T^{(k)}(\cdot, \cdot)$ can be the identity operator, differential operator $\mathcal{N}(\cdot, \cdot)$ or the boundary condition $\mathcal{B}(\cdot, \cdot)$. We will refer to a loss function written in the above equation as a "fully-decoupled" loss. Similarly, we will refer to a loss function written in the form of equation (2.19) as a "partially-decoupled" loss.

Now we generalize the original definition of the NTK to physics-informed DeepONets.

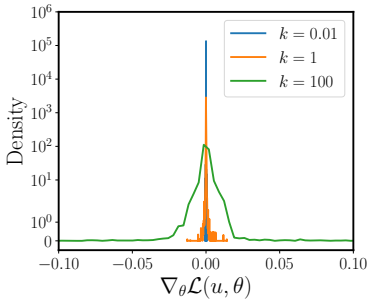

Figure 3: *Anti-derivative operator:* Histogram of back-propagated gradients corresponding to the input samples with different output scale $k \in [10^{-2}, 10^2]$.

Definition 3.1. *Given the loss function 3.6, the Neural Tangent Kernel of physics-informed DeepONets (NTK of PI-DeepONets) is a matrix $\boldsymbol{H}(\boldsymbol{\theta}) \in \mathbb{R}^{N^* \times N^*}$ whose entries are defined by*

$$\boldsymbol{H}_{ij}(\boldsymbol{\theta}) = \left\langle \frac{dT^{(i)}\left(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i)\right)}{d\boldsymbol{\theta}}, \frac{dT^{(j)}\left(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j)\right)}{d\boldsymbol{\theta}} \right\rangle, \tag{3.7}$$

for $i, j = 1, 2, \ldots, N^*$.

Lemma 3.2. *The NTK of PI-DeepONets $\boldsymbol{H}(\boldsymbol{\theta})$ has the following properties*

(a) $\boldsymbol{H}(\boldsymbol{\theta})$ is a positive semi-definite matrix.

(b) $\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty$ is achieved by some diagonal entry of $\boldsymbol{H}(\boldsymbol{\theta})$, i.e

$$\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty = \max_{1 \le k \le N^*} \boldsymbol{H}_{kk}(\boldsymbol{\theta}). \tag{3.8}$$

Proof. The proof is provided in Appendix D. □

For simplicity, we also introduce the following notation.

Definition 3.3. *For the loss function 3.6, we define*

(a) $\boldsymbol{U} = \left(\boldsymbol{u}^{(k)}\right)_{k=1}^{N^*}$ $\boldsymbol{X} = (\boldsymbol{x}_k)_{k=1}^{N^*}$, $\boldsymbol{Y} = (\boldsymbol{y}_k)_{k=1}^{N^*}$.

(b) $\boldsymbol{U}(\boldsymbol{X}) = \left(u^{(k)}(\boldsymbol{x}_k)\right)_{k=1}^{N^*}$, $\boldsymbol{G}_{\boldsymbol{\theta}}(\boldsymbol{U})(\boldsymbol{Y}) = \left(G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)\right)_{k=1}^{N^*}$.

(c) $\boldsymbol{T}(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}}(\boldsymbol{U})(\boldsymbol{Y}))) = \left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right)_{k=1}^{N^*}$.

(d) $\mathcal{L}^{(k)}(\boldsymbol{\theta}) = \left|T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right|^2$, then $\mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N^*} \sum_{k=1}^{N^*} \mathcal{L}^{(k)}(\boldsymbol{\theta})$.

With these definitions, we can prove the following lemma, which characterizes the training dynamics of physics-informed DeepONets.

Lemma 3.4. *Given the loss function 3.6, the physics-informed DeepONet outputs is governed by the following ODE:*

$$\frac{d\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y}))\right)}{dt} = -\frac{4}{N^*}\boldsymbol{H}(\boldsymbol{\theta}) \cdot \boldsymbol{T}(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y}))) \tag{3.9}$$

where $\boldsymbol{H} \in \mathbb{R}^{N^* \times N^*}$ is the NTK of PI-DeepONets defined in equation 3.7

Proof. The proof is provided in Appendix E. □

Although the recent work of [34, 35] has empirically observed that the NTK of finite-width MLP networks evolves at a constant slow velocity after a rapid change at the early training stages, the behavior of the NTK of physics-informed DeepONets is unclear during training, and worth of further investigation in the future. Nevertheless, performing a similar analysis as in Section 2.1 we can show that

$$\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y}))\right) \approx e^{-\boldsymbol{H}t}\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(0)}(\boldsymbol{U})(\boldsymbol{Y}))\right). \tag{3.10}$$

Letting $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}Q^T$ be an orthogonal decomposition of $\boldsymbol{H}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix whose entries are the eigenvalues of $\boldsymbol{H}$, we obtain

$$\boldsymbol{Q}^T\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y}))\right) \approx e^{-\boldsymbol{\Lambda}\boldsymbol{t}}\boldsymbol{Q}^T\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(0)}(\boldsymbol{U})(\boldsymbol{Y}))\right). \tag{3.11}$$

Consequently, the eigenvalues of $\boldsymbol{H}(\boldsymbol{\theta})$ not only determine the stiffness of the ODE system 3.9, but also characterizes the convergence rate of total training error.

Algorithm 1: NTK-guided weighting scheme for training deep operator networks

Consider a physics-informed DeepONet $G_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$ and the corresponding weighted loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N^*}\sum_{k=1}^{N^*} \lambda_k \left| T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right|^2, \tag{3.13}$$

where all weights $\{\lambda_k\}_{k=1}^{N^*}$ are initialized to 1. Then, use S steps of a gradient descent algorithm to update the parameters $\boldsymbol{\theta}$ as: **for** $n = 1, \ldots, S$ **do**

(a) Compute and update λ_k by

$$\lambda_k = \left(\frac{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}{\boldsymbol{H}_{kk}(\boldsymbol{\theta}_n)} \right)^\alpha = \left(\frac{\max_{1\le k\le N^*} \boldsymbol{H}_{kk}(\boldsymbol{\theta}_n)}{\boldsymbol{H}_{kk}(\boldsymbol{\theta}_n)} \right)^\alpha \tag{3.14}$$

where $\boldsymbol{\theta}_n$ denotes the DeepONet parameters at n-th step, and $\boldsymbol{H}_{kk}$ is the k-th diagonal entry of the NTK matrix defined in Definition 3.1. Here $\alpha \in [0, 1]$ is a user-defined hyper-parameter that determines the magnitude of each weight.

(b) Update the parameters $\boldsymbol{\theta}$ via gradient descent

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_n) \tag{3.15}$$

The recommended hyper-parameter values are: $\alpha = 1$ or $\alpha = \frac{1}{2}$.

end

3.3 NTK-guided weights for physics-informed DeepONets

Recall that our goal is to assign appropriate weights to each individual term in the loss function of physics-informed DeepONets to balance the back-propagated gradients, as well as calibrate the convergence rate of each term. To this end, we consider a weighted loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N^*}\sum_{k=1}^{N^*} \lambda_k \mathcal{L}^{(k)}(\boldsymbol{\theta}) = \frac{2}{N^*}\sum_{k=1}^{N^*} \lambda_k \left| T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right|^2. \tag{3.12}$$

Motivated by Wang *et. al.* [18], we propose Algorithm 1 to calibrate the interplay between each individual term of the loss function. One may note that Algorithm 1 reduces to training a conventional physics-informed DeepONet when $\alpha = 0$. However, in this work we will mainly consider the case of $\alpha = \frac{1}{2}$, and $\alpha = 1$. In the rest of this work, we may refer to the NTK-guided weights with $\alpha = 1$ as NTK weights while referring the NTK-guided weights with $\alpha = \frac{1}{2}$ as moderate NTK weights.

To get a deeper understanding of this algorithm and the role of the hyper-parameter α, let us boldly assume that each individual term of the loss function 3.6 does not affect one another, and is minimized independently via gradient descent. This simplifying assumption enables us to decouple the total loss into individual terms, thus enabling us to analyze their training dynamics separately. For every $1 \le k \le N^*$, we then obtain

$$\frac{dT^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))}{dt} = -\frac{4\lambda_k}{N^*} \left\| \frac{dT^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))}{d\boldsymbol{\theta}} \right\|_2^2 \cdot \left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right) \tag{3.16}$$

$$= -\frac{4}{N^*}\lambda_k \boldsymbol{H}_{kk}(\boldsymbol{\theta}) \cdot \left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right) \tag{3.17}$$

This implies that (under our independence assumption) $\lambda_k \boldsymbol{H}_{kk}(\boldsymbol{\theta})$ essentially quantifies the convergence rate of every weighted individual loss. Then, taking $\alpha = 1$ in equation 3.14 yields

$$\frac{dT^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))}{dt} = -\frac{4}{N^*}\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty \cdot \left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k)) \right), \ \ 1 \le k \le N^*. \tag{3.18}$$

Consequently, computing the NTK weights by setting $\alpha = 1$ enables each individual loss to achieve the same maximum convergence rate $\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty$.

Next, let us examine the back-propagated gradients of each individual weighted term when applying gradient descent updates. For $k = 1, 2, \ldots, N^*$, we have

$$\left\|\nabla_{\boldsymbol{\theta}}\lambda_k\mathcal{L}^{(k)}(\boldsymbol{\theta})\right\|_2 = 2\frac{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}{\boldsymbol{H}_{kk}(\boldsymbol{\theta}_n)}\left\|\left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right)\nabla_{\boldsymbol{\theta}}T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right\|_2 \tag{3.19}$$

$$= 2\frac{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}{\left\|\nabla_{\boldsymbol{\theta}}T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right\|_2}\left|T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right|. \tag{3.20}$$

As a result, the NTK weights obtained by setting $\alpha = 1$ tend to penalize the gradients with large magnitude and to amplify the gradients with small magnitude. However, in practice we observe that this update rule sometimes yields extremely large weights , potentially due to the presence of vanishing gradients, which can lead to an unstable training process. In contrast, updating the NTK weights by setting $\alpha = \frac{1}{2}$ gives

$$\left\|\nabla_{\boldsymbol{\theta}}\lambda_k\mathcal{L}^{(k)}(\boldsymbol{\theta})\right\|_2 = 2\sqrt{\frac{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}{\boldsymbol{H}_{kk}(\boldsymbol{\theta}_n)}}\left\|\left(T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right)\nabla_{\boldsymbol{\theta}}T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right\|_2 \tag{3.21}$$

$$= 2\sqrt{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}\left|T^{(k)}(u^{(k)}(\boldsymbol{x}_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(\boldsymbol{y}_k))\right|. \tag{3.22}$$

This implies that NTK weights computed with $\alpha = \frac{1}{2}$ normalize the loss function such that the gradients of each weighted individual loss are of the same magnitude $\sqrt{\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty}$.

In summary, Algorithm 1 with $\alpha = 1$ aims to balance the convergence rate derived from the associated gradient flow, while Algorithm 1 with $\alpha = \frac{1}{2}$ aims to balance the back-propagated gradients. Rather than constructing the whole NTK matrix, it is worth emphasizing that the Algorithm 1 only requires one to compute the NTK diagonal entries, which will greatly save computational costs and reduce the total training time in practice.

To compare the performance of different weight schemes mentioned in Section 3.1 - 3.2, let us first revisit the example of learning anti-derivative operator shown in Section 3.1. In all cases we will employ the same DeepONet architecture to represent the solution operator G (see Appendix B for additional details). The DeepONet model can be trained by minimizing the weighted loss function with different weighting schemes under exactly the same hyper-parameter settings (i.e. # of iterations, learning rate, batch size, etc.). The left panel of Figure 4 summarizes the averaged resulting relative L^2 prediction errors along with their standard deviation corresponding to input functions with different output scales in the test data-set. Among all weighting schemes, NTK weights with $\alpha = 1$ yields the best predictive accuracy as well as the smallest standard deviation, which indicates that the trained DeepONet is unbiased towards different output-scale input samples. Moreover, using $\lambda = \frac{1}{\|s\|_\infty}$ as weights slightly mitigates the gradient pathology issue and achieves a similar performance to the original un-weighted loss. To explore the underlying reasons, we investigate the weight distribution of different update rules after training. Particularly, we compute the weights for each input sample $\boldsymbol{u}$ in a random mini-batch and visualize them in a descending order of $\|\boldsymbol{u}\|_2$. As shown in the right panel of Figure 4, all weight distributions are inversely proportional to the magnitude of input samples, which potentially balance the back-propagated gradients of each individual term in the loss function. However, one may note that the weights for $\lambda = \frac{1}{\|s\|_\infty}$ exhibit high variance. This is because of the inaccurate estimate of $\|s\|_\infty$ since only one corresponding observation is provided for each input sample. We believe that such inaccurate estimation of $\|s\|_\infty$ leads to a poor model performance.

3.4 An improved DeepONet architecture

Besides effective training algorithms, the network architecture itself also plays a significant role in deep learning. In this section, we propose a novel DeepONet architecture, which is empirically proved to be uniformly better than the conventional DeepONet architecture of Lu *et al.* [4]. To demonstrate our motivation, we start with a perspective of deep information propagation [36], which hypothesizes that a necessary condition for a random neural network to be trainable is that information should be able to pass through it in a stable manner. One may notice that the conventional DeepONet architecture merges the latent representations of DeepONet inputs $\boldsymbol{u}$ and $\boldsymbol{y}$ only in the last layer via an inner product. Consequently, the final information fusion may be inefficient if the DeepONet input signals fail to propagate through a deep branch network or trunk network at initialization, leading to an ineffective training process and poor model performance. To design a network architecture that is more resilient to vanishing signals, we draw motivation

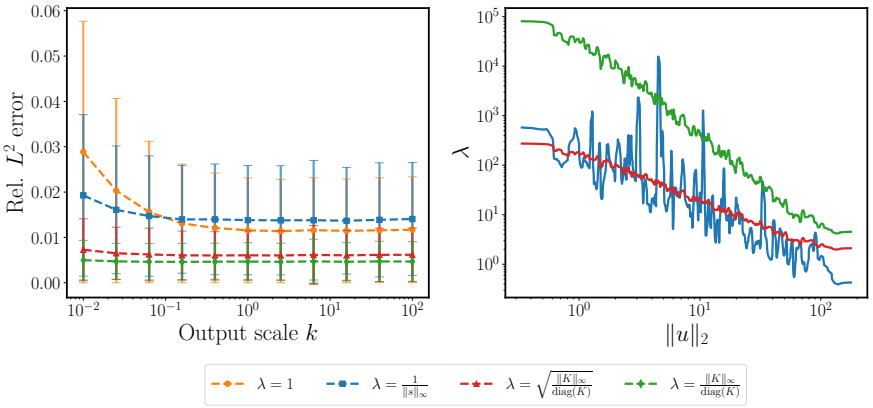

Figure 4: *Anti-derivative operator: Left:* Mean and standard deviation of the relative L^2 prediction errors of the trained DeepONets using different weighting schemes, averaged over the test data sampled from a GRF with different output scale $k \in [10^{-2}, 10^2]$. A detailed description of the legend is summarized in Table 2. *Right:* Weight distributions of trained models with different weighting schemes in increasing order of $\|u\|_2$.

from Wang *et. al.* [17] and modify the forward pass of an L-layer DeepONet as follows

$$\boldsymbol{U} = \phi(\boldsymbol{W}_u \boldsymbol{u} + \boldsymbol{b}_u), \quad \boldsymbol{V} = \phi(\boldsymbol{W}_y \boldsymbol{y} + \boldsymbol{b}_y) \tag{3.23}$$

$$\boldsymbol{H}_u^{(1)} = \phi(\boldsymbol{W}_u^{(1)} \boldsymbol{u} + \boldsymbol{b}_u^{(1)}), \quad \boldsymbol{H}_y^{(1)} = \phi(\boldsymbol{W}_y^{(1)} \boldsymbol{y} + \boldsymbol{b}_y^{(1)}) \tag{3.24}$$

$$\boldsymbol{Z}_u^{(l)} = \phi(\boldsymbol{W}_u^{(l)} \boldsymbol{H}_u^{(l)} + \boldsymbol{b}_u^{(l)}), \quad \boldsymbol{Z}_y^{(l)} = \phi(\boldsymbol{W}_y^{(l)} \boldsymbol{H}_y^{(l)} + \boldsymbol{b}_y^{(l)}), \quad l = 1, 2, \ldots, L-1 \tag{3.25}$$

$$\boldsymbol{H}_u^{(l+1)} = (1 - \boldsymbol{Z}_u^{(l)}) \odot \boldsymbol{U} + \boldsymbol{Z}_u^{(l)} \odot \boldsymbol{V}, \quad l = 1, \ldots, L-1 \tag{3.26}$$

$$\boldsymbol{H}_y^{(l+1)} = (1 - \boldsymbol{Z}_y^{(l)}) \odot \boldsymbol{U} + \boldsymbol{Z}_y^{(l)} \odot \boldsymbol{V}, \quad l = 1, \ldots, L-1 \tag{3.27}$$

$$\boldsymbol{H}_u^{(L)} = \phi(\boldsymbol{W}_u^{(L)} \boldsymbol{H}_u^{(L-1)} + \boldsymbol{b}_u^{(L)}), \quad \boldsymbol{H}_y^{(L)} = \phi(\boldsymbol{W}_y^{(L)} \boldsymbol{H}_y^{(L-1)} + \boldsymbol{b}_y^{(L)}) \tag{3.28}$$

$$G_{\boldsymbol{\theta}}(\boldsymbol{u})(\boldsymbol{y}) = \left\langle \boldsymbol{H}_u^{(L)}, \boldsymbol{H}_y^{(L)} \right\rangle, \tag{3.29}$$

where $\odot$ denotes point-wise multiplication, ϕ denotes a activation function, and $\boldsymbol{\theta}$ represents all trainable parameters of the DeepONet model. In particular, $\{\boldsymbol{W}_u^{(l)}, \boldsymbol{b}_u^{(l+1)}\}_{l=1}^{L+1}$ and $\{\boldsymbol{W}_y^{(l)}, \boldsymbol{b}_y^{(l+1)}\}_{l=1}^{L+1}$ are the weights and biases of the branch and trunk networks, respectively.

At first glance, this novel architecture seems to appear a bit complicated. However, notice that the proposed architecture is almost the same as a standard DeepONet model using MLPs as backbone, with the addition of two encoders and a minor modification in the forward pass. As illustrated in Figure 5, we embed the DeepONet inputs $\boldsymbol{u}$ and $\boldsymbol{y}$ into a high-dimensional feature space via two encoders $\boldsymbol{U}, \boldsymbol{V}$, respectively. Instead of just merging the propagated information in the output layer of the branch and trunk networks, we merge the embeddings $\boldsymbol{U}, \boldsymbol{V}$ in each hidden layer of these two sub-networks using a point-wise multiplication (equation (3.26) - (3.27)). Heuristically, this design may not only help input signals propagate through the DeepONet, but also enhance its capability of representing non-linearity due to the extensive use of point-wise multiplications. In section 4, we will demonstrate that this simple modification uniformly leads to more accurate predictions than the original DeepONet architecture across all benchmarks considered in this work.

4 Results

To validate the proposed training algorithms and network architectures, we present a series of comprehensive numerical studies across a range of representative benchmarks for which the conventional physics-informed DeepONets struggle. More specifically, we quantify the predictive accuracy of trained physics-informed DeepONets using the different weighting schemes listed in Table 2, as well as the different DeepONet architectures listed in Table 3.

The error metric employed throughout all numerical experiments to assess model performance is the relative L^2 norm. Specifically, the reported test errors correspond to the mean of the relative L^2 error of a trained model over all examples

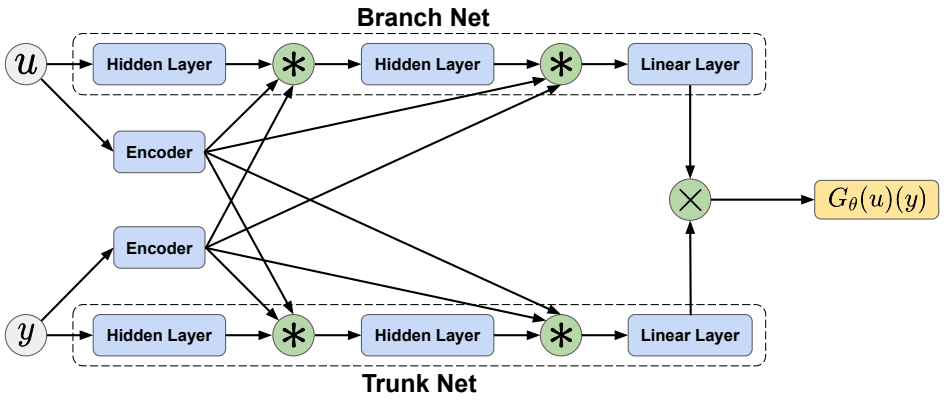

Figure 5: *Modified DeepONet architecture:* Two encoders are applied to the input samples and the input coordinates, respectively. The embedded features are then inserted into each hidden layer of the branch network and the trunk network using a $*$ operation, which is defined in equation (3.26) - (3.27). The final outputs is obtained by the same way as conventional DeepONets, i.e, merging the outputs of the branch and trunk networks via a dot production.

Weighting scheme	Notation	Loss form	Update rule
No weights	$\lambda = 1$	Fully-decoupled	$\lambda_k = 1$
Fixed weights	$\lambda_{\text{bc}} = \lambda$	Partially-decoupled	$\lambda_{\text{bc}} = \lambda, \lambda_{\text{r}} = 1$
Data-guided weights	$\lambda = \frac{1}{\|s\|_\infty}$	Fully-decoupled	$\lambda_k = \frac{1}{\|s^{(k)}\|_\infty}$
NTK-guided weights	$\lambda = \left(\frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}\right)^\alpha$	Fully-decoupled	$\lambda_k = \left(\frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H}_{kk})}\right)^\alpha$

Table 2: Weighting schemes and their associated notation, as considered in the numerical studies presented in this work.

in the test data-set, i.e

$$\text{Test error} := \frac{1}{N}\sum_{i=1}^{N}\frac{\|G_\theta(\boldsymbol{u}^{(i)})(y) - G(\boldsymbol{u}^{(i)})(y)\|_2}{\|G(\boldsymbol{u}^{(i)})(y)\|_2}, \tag{4.1}$$

where N denotes the number of examples in the test data-set and y is typically a set of equi-spaced points in the domain of $G(u)$. Here $G_\theta(\boldsymbol{u}^{(i)})(\cdot)$ denotes a predicted DeepONet output function, while $G(\boldsymbol{u}^{(i)})(\cdot)$ corresponds to the ground truth target functions.

Throughout all benchmarks we employ hyperbolic tangent activation functions (Tanh) and initialize the DeepOnet networks using the Glorot normal scheme [10], unless otherwise stated. All networks are trained via mini-batch stochastic gradient descent using the Adam optimizer [33] with default settings. Particularly, we set the batch size to be 10,000 and use exponential learning rate decay with a decay-rate of 0.9 every 2, 000 training iterations. The detailed hyper-parameters and computational cost for all examples are listed in Appendix B and C. We can see that the proposed methods generally incur a higher computational costs than the original DeepONet formulation. For example, training physics-informed DeepONets with NTK weights is $\sim$3-5x slower than without any weighting schemes, while employing the modified DeepONet architecture takes about twice the training time as the conventional DeepONet architecture. The code and data accompanying this manuscript will be made publicly available at `https://github.com/PredictiveIntelligenceLab/ImprovedDeepONets`.

4.1 Advection equation

We start with a one-dimensional linear advection equation with variable coefficients of the form

$$\frac{\partial s}{\partial t} + u(x)\frac{\partial s}{\partial x} = 0, \quad (x,t) \in (0,1)\times(0,1), \tag{4.2}$$

DeepONet architecture	Description
DeepONet with MLP	The branch and trunk networks are MLPs
DeepONet with modified MLP	The branch and trunk networks are modified MLPs, see [17].
Modified DeepONet	The proposed novel DeepONet architecture in section 3.4.

Table 3: DeepONet architectures considered in the numerical studies presented in this work. We remark that DeepONet with MLP refers to a conventional DeepONet model, while DeepONet with modified MLP consists of two independent sub-networks employing modified MLP networks as backbone that does not share any parameters. One main difference between the modified DeepONet and conventional DeepONet architectures is that two encoders are employed to exchange information between the branch and the trunk networks before merging them together. To simplify the notations, we may omit "DeepONet" in the rest of this work.

subject to the initial and boundary condition

$$s(x,0) = f(x), \quad x \in [0,1], \tag{4.3}$$
$$s(0,t) = g(t), \quad t \in [0,1] \tag{4.4}$$

where $f(x) = \sin(\pi x)$ and $g(t) = \sin(\frac{\pi}{2}t)$. To ensure numerical stability, we make the input function $u(x)$ strictly positive by letting $u(x) = v(x) - \min_x v(x) + 1$ where $v(x)$ is sampled from a GRF with a length scale $l = 0.2$. Our goal is to learn the solution operator G mapping the variable coefficient $u(x)$ to the associated spatio-temporal PDE solution $s(x,t)$.

We represent the solution map by a DeepONet $G_{\boldsymbol{\theta}}$ and define PDE residual as

$$\mathcal{R}_{\boldsymbol{\theta}}(\boldsymbol{u})(x,t) = \frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u})(x,t)}{\partial t} - u(x)\frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u})(x,t)}{\partial x}. \tag{4.5}$$

Then, we formulate a weighted physics-informed as follows

$$\mathcal{L}(\boldsymbol{\theta}) = \lambda_{\text{bc}}\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) + \lambda_{\text{ic}}\mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) + \lambda_{\text{r}}\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}), \tag{4.6}$$

where

$$\mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{ic,j}^{(i)},0) - f(x_{ic,j}^{(i)})\right|^2, \tag{4.7}$$

$$\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(0,t_{bc,j}^{(i)}) - g(t_{bc,j}^{(i)})\right|^2, \tag{4.8}$$

$$\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}) = \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|\mathcal{R}_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{r,j}^{(i)},t_{r,j}^{(i)})\right|^2, \tag{4.9}$$

and $\lambda_{\text{bc}}, \lambda_{\text{ic}}, \lambda_{\text{r}}$ can be set to 1 or some other user-specified value, and remain fixed during training. For using different weighting schemes, we reformulate the loss function into a fully-decoupled form

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N^*}\sum_{k=1}^{N^*}\lambda_k\left|T^{(k)}(u^{(k)}(x_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(x_k,t_k))\right|^2, \tag{4.10}$$

where $T^{(k)}$ is a operator corresponding to either boundary, initial conditions or differential operators (e.g $\mathcal{R}_{\boldsymbol{\theta}}$), and λ_k are weights that will be automatically updated by Algorithm 1 during training.

Here, we take $N = 2,000$, $P = 200$ and $Q = 2,500$. The input samples $\boldsymbol{u}^{(i)}$ are evaluated at equi-spaced points $\{x_i\}_{i=1}^m$ in $[0,1]$. To generate a set of test data, we sample $N = 100$ input functions from the same GRF and solve the advection equation using the Lax–Wendroff scheme [32] on a 100×100 uniform grid.

We train the physics-informed DeepONet using different weighting schemes and DeepONet architectures for 3×10^5 iterations, and report the resulting test error in Table 4. Despite some improvements brought by all weighting schemes, the results obtained with both the MLP and modified MLP architectures are unsatisfactory. In contrast, the modified

Method \ Architecture	MLP	Modified MLP	Modified DeepONet
$\lambda = 1$	$6.39\% \pm 1.76\%$	$6.83\% \pm 1.82\%$	$1.78\% \pm 0.42\%$
$\lambda_{\text{bc}} = \lambda_{\text{ic}} = \lambda$	$4.73\% \pm 1.43\%$	$3.74\% \pm 1.17\%$	$1.03\% \pm 0.25\%$
$\lambda = \frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}$	$3.88\% \pm 1.42\%$	$2.74\% \pm 1.04\%$	$1.19\% \pm 0.35\%$
$\lambda = \sqrt{\frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}}$	$4.22\% \pm 1.28\%$	$3.37\% \pm 1.12\%$	$\mathbf{0.95\% \pm 0.23\%}$

Table 4: *Advection equation:* Test errors of trained physics-informed DeepONets using different weighting schemes and network architectures. In particular, the results of $\lambda_{\text{ic}} = \lambda_{\text{bc}} = \lambda$ shows the best accuracy obtained by training the physics-informed DeepONets for different $\lambda \in [10^{-2}, 10^2]$. The resulting test errors and their standard deviations are summarized in Figure 17.

DeepONet architecture with moderate NTK weights ($\alpha = 1/2$) achieves the best accuracy, which has been improved by a factor of x6 compared to the baseline physics-informed DeepONets [8]. A more detailed visual assessment of this comparison is shown in Figure 6. In contrast to the inaccurate approximation of sharp gradients by the conventional physics-informed DeepONet, the result of the improved one shows excellent agreement between the predictions and the numerical estimations with a relative L^2 error of 0.73%.

To further elucidate the important role played by the adaptive per-example weights λ_k, we can the weight distribution over the computational domain during training. Specifically, for some input sample $\boldsymbol{u}$, we can define the weight distribution associated with the PDE residual loss as

$$\lambda_r(\boldsymbol{u})(x, y) = \sqrt{\frac{\|\boldsymbol{H}\|_\infty}{\|\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\boldsymbol{\theta}}(\boldsymbol{u})(x, y)\|_2^2}}. \tag{4.11}$$

Figure 7 shows the distribution of the residual weights corresponding to the same input sample as in Figure 6, at the end of training. Interestingly, the computed weights resemble an "attention map" that identifies regions of sharp gradients, in which small weights are assigned. This implies that Algorithm 1 tends to relax the PDE constraint in such regions, hence making the residual loss easier to minimize. More visualizations of the model predictions and the associated weight distribution maps can be found in Appendix Figure 18. We can observe that this phenomenon is not accidental and strikingly manifests itself across all input samples.

4.2 Burgers' equation

To demonstrate the effectiveness of the proposed approaches in tackling nonlinear PDEs, let us consider the one-dimensional Burgers' equation

$$\frac{ds}{dt} + s\frac{ds}{dx} - \nu\frac{d^2 s}{dx^2} = 0, \quad (x, t) \in (0, 1) \times (0, 1], \tag{4.12}$$

subject to initial and boundary conditions

$$s(x, 0) = u(x), \quad x \in (0, 1), \tag{4.13}$$
$$s(0, t) = s(1, t), \quad t \in (0, 1), \tag{4.14}$$
$$\frac{ds}{dx}(0, t) = \frac{ds}{dx}(1, t), \quad t \in (0, 1), \tag{4.15}$$

where the viscosity is set to $\nu = 10^{-3}$, and the initial condition $u(x)$ is generated from a GRF [37].

Our objective here is to learn the solution operator mapping initial conditions $u(x)$ to the associated full spatio-temporal solution $s(x, t)$. We proceed by representing the solution operator by a DeepONet $G_{\boldsymbol{\theta}}$. For any input sample $\boldsymbol{u}$, the PDE residual is then defined by

$$R_{\boldsymbol{\theta}}(\boldsymbol{u})(x, t) = \frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u})(x, t)}{\partial t} + G_{\boldsymbol{\theta}}(\boldsymbol{u})(x, t)\frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u})(x, t)}{\partial x} - \nu\frac{\partial^2 G_{\boldsymbol{\theta}}(\boldsymbol{u})(x, t)}{\partial x^2}, \tag{4.16}$$

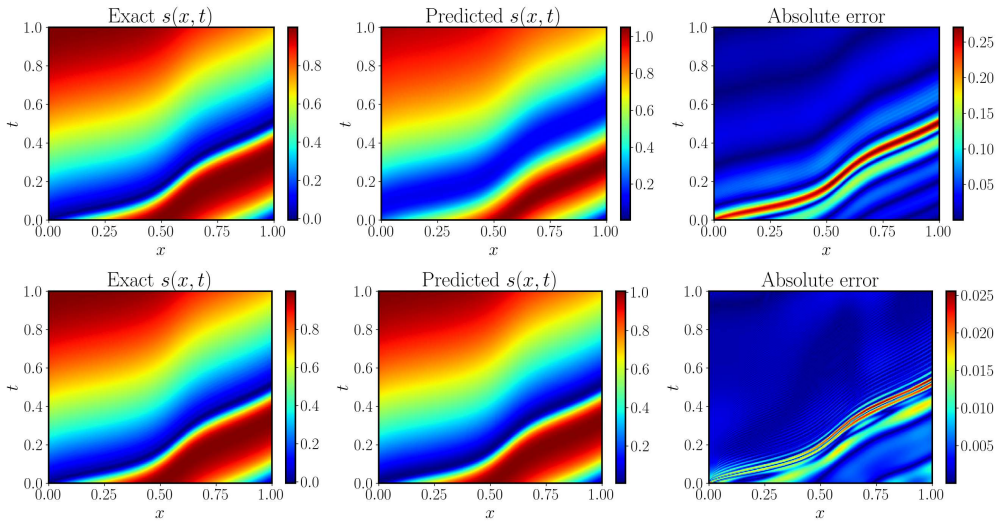

Figure 6: *Advection equation: Top:* Exact solution versus the prediction of a trained conventional physics-informed DeepONet for a representative example in the test data-set. The resulting relative L^2 error is 9.32%. *Bottom:* Exact solution versus the prediction of a physics-informed DeepONet represented by modified DeepONet architecture and trained using Algorithm 1 with $\alpha = \frac{1}{2}$ for the same example. The resulting relative L^2 error is 0.73%, which is 10x more accurate than the original formulation.

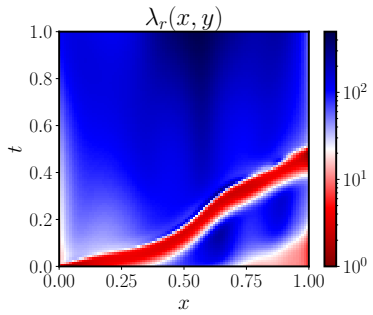

Figure 7: *Advection equation:* Weight distribution defined in equation (4.11) for the same input sample as in Figure 6.

Consequently, a physics-informed DeepONet can be trained by minimizing the following weighted loss function with two equivalent forms

$$\mathcal{L}(\boldsymbol{\theta}) = \lambda_{\text{bc}}\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) + \lambda_{\text{ic}}\mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) + \lambda_{\text{r}}\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}), \tag{4.17}$$

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N^*}\sum_{k=1}^{N^*} \lambda_k \left| T^{(k)}(u^{(k)}(x_k), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(k)})(x_k,t_k)) \right|^2, \tag{4.18}$$

where

$$\mathcal{L}_{\text{IC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{\text{ic},j}^{(i)},0) - u^{(i)}(x_{\text{ic},j}^{(i)})\right|^2 \tag{4.19}$$

$$\mathcal{L}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(0,t_{\text{bc},j}^{(i)}) - G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(1,t_{\text{bc},j}^{(i)})\right|^2 \tag{4.20}$$

$$+\frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|\frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(0,t_{\text{bc},j}^{(i)})}{\partial x} - \frac{\partial G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(1,t_{\text{bc},j}^{(i)})}{\partial x}\right|^2 \tag{4.21}$$

$$\mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}) = \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|R_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{\text{r},j}^{(i)},t_{\text{r},j}^{(i)})\right|^2. \tag{4.22}$$

To obtain a set of training data, we randomly sample $N = 1,000$ input functions from a GRF $\sim \mathcal{N}\left(0, 25^2(-\Delta + 5^2 I)^{-4}\right)$ and for every input sample $\boldsymbol{u}^{(i)}$, we uniformly sample $P = 100$ locations $x_{ic,j}^{(i)} = x_j$ and $\{(0,t_{ic,j}^{(i)})\}_{j=1}^{P}, \{(1,t_{ic,j}^{(i)})\}_{j=1}^{P}$ from each boundary, and $Q = 2500$ collocation points $\{(x_{r,j}^{(i)}, t_{r,j}^{(i)})\}_{j=1}^{Q}$ from the domain interior. We generate the test data-set by sampling another 500 input functions from the same GRF and solve the Burgers' equation (4.12) using the Chebfun package [38] with a spectral Fourier discretization and a fourth-order stiff time-stepping scheme (ETDRK4) [39] with a time-step size of 10^{-4}. Temporal snapshots of the solution are saved every $\Delta t = 0.01$ to give us 101 snapshots in total. Consequently, the test data-set contains 500 realizations evaluated at a 100×101 spatio-temporal grid.

Table 5 summarizes the test error of the trained physics-informed DeepONets with different weighting schemes and architectures after 2×10^5 iterations of gradient descent. Compared to the poor performance of the MLP and modified MLP architectures, the proposed modified DeepONet achieves a test error of 8.10% even without any weights. The error can be further reduced to 3.69% by using NTK weights with $\alpha = \frac{1}{2}$, yielding a result that is at least 5x more accurate than the baseline physics-informed DeepONets [8]. To further compare the performance of the different trained models, we present a representative predicted solution obtained from the worst and the best trained models in Figure 8 and Figure 9, respectively. We can observe that the latter can accurately capture the viscous shock wave and is in excellent agreement with the ground truth. The resulting relative L^2 error is 1.19%, which is about 30x more accurate than the former one (30.72%). In Figure 10, we also visualize the weights distribution of the PDE residual corresponding to the same input sample as in Figure 9. We can observe a similar behavior as in the previous benchmark, i.e., the region of viscous shocks are identified and assigned small weights. Additional representative predictions with their associated residual weight distribution maps corresponding to different input functions are shown in Figure 22.

Furthermore, we also investigate the effect of the viscosity parameter in the Burgers' equation on the performance of physics-informed DeepONets. In particular, we vary the viscosity ν from 10^{-2} to 10^{-4} and train physics-informed DeepONets with different architectures and weighting schemes, under exactly the same hyper-parameter settings (i.e. learning rate, batch size, optimizer, etc). Table 6 summarizes the test error of the best and the worst trained models. These results further validate the remarkable improvements of the proposed architecture and adaptive training algorithms. It is no surprise that the accuracy decreases as we reduce the viscosity since smaller viscosity typically results in sharper gradients and stiffer dynamics, leading to a harder optimization problem.

4.3 Stokes flow

Our last example aims to highlight the capability of the proposed methods on solving PDEs in domains of varying shapes. To this end, we consider an example of a two-dimensional Stokes flow over an obstacle. The associated PDE system takes the form

$$-\Delta \boldsymbol{u} + \nabla p = 0, \quad (x,y) \in \Omega/\Gamma, \tag{4.23}$$

$$\nabla \cdot \boldsymbol{u} = 0, \quad (x,y) \in \Omega/\Gamma, \tag{4.24}$$

subject to boundary conditions

$$\boldsymbol{u} = \boldsymbol{0}, \quad (x,y) \in \Lambda_1 \cup \partial\Gamma, \tag{4.25}$$

$$\boldsymbol{u} = \boldsymbol{g}, \quad (x,y) \in \Lambda_2 \tag{4.26}$$

$$p = 0, \quad (x,y) \in \Lambda_3, \tag{4.27}$$

Method \ Architecture	MLP	Modified MLP	Modified DeepONet
$\lambda = 1$	$26.91\% \pm 12.26\%$	$27.46\% \pm 16.54\%$	$8.10\% \pm 6.44\%$
$\lambda_{\text{ic}} = \lambda_{\text{bc}} = \lambda$	$22.20\% \pm 9.50\%$	$22.87\% \pm 9.22\%$	$8.10\% \pm 6.44\%$
$\lambda = \frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}$	$23.85\% \pm 8.08\%$	$15.76\% \pm 12.27\%$	$5.53\% \pm 5.95\%$
$\lambda = \sqrt{\frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}}$	$19.87\% \pm 11.06\%$	$20.21\% \pm 14.86\%$	$\mathbf{3.69}\% \pm \mathbf{3.63}\%$

Table 5: *Burger's equation:* Test errors of trained physics-informed DeepONets using different weighting schemes and network architectures. In particular, the results of $\lambda_{\text{ic}} = \lambda_{\text{bc}} = \lambda$ shows the best accuracy obtained by training the physics-informed DeepONets for different $\lambda \in [10^{-2}, 10^{2}]$. The resulting test errors and their standard deviations are visualized in Figure 20.

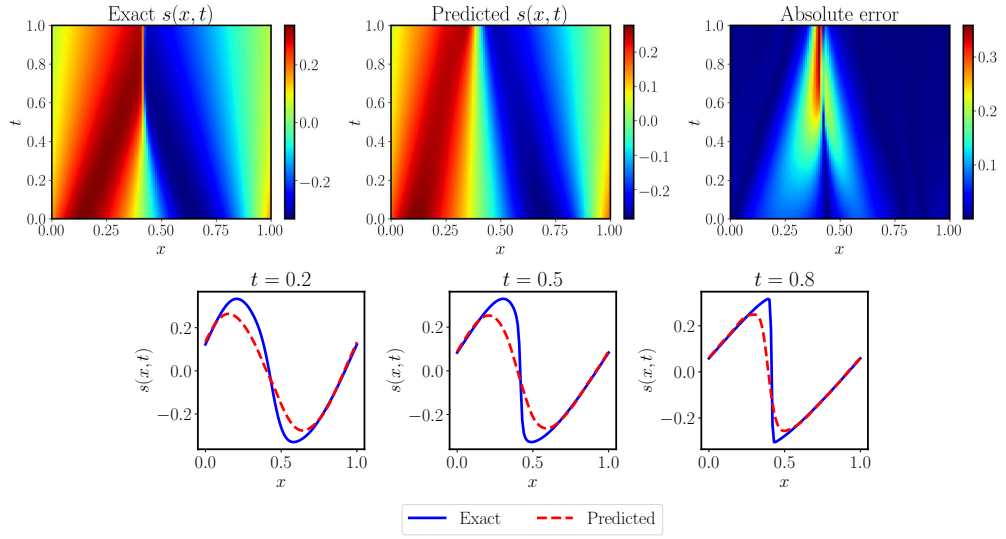

Figure 8: *Burger's equation: Top:* Exact solution versus the prediction of a trained conventional physics-informed DeepONet for a representative example in the test data-set. *Bottom:* Comparison of the predicted and exact solutions corresponding to the three temporal snapshots at $t = 0.2, 0.5, 0.8$. The resulting relative L^2 error is 30.72%

Model \ Viscosity	$\nu = 10^{-2}$	$\nu = 10^{-3}$	$\nu = 10^{-4}$
Plain PI-DeepONet	$3.48\% \pm 3.79\%$	$26.91\% \pm 12.26\%$	$29.21\% \pm 12.38\%$
Improved PI-DeepONet	$1.03\% \pm 1.35\%$	$3.69\% \pm 3.63\%$	$7.95\% \pm 5.29\%$

Table 6: *Solving a parametric Burger's equation:* Relative L^2 prediction error of a trained physics-informed DeepONet averaged over all examples in the test data-set, for different viscosity values.

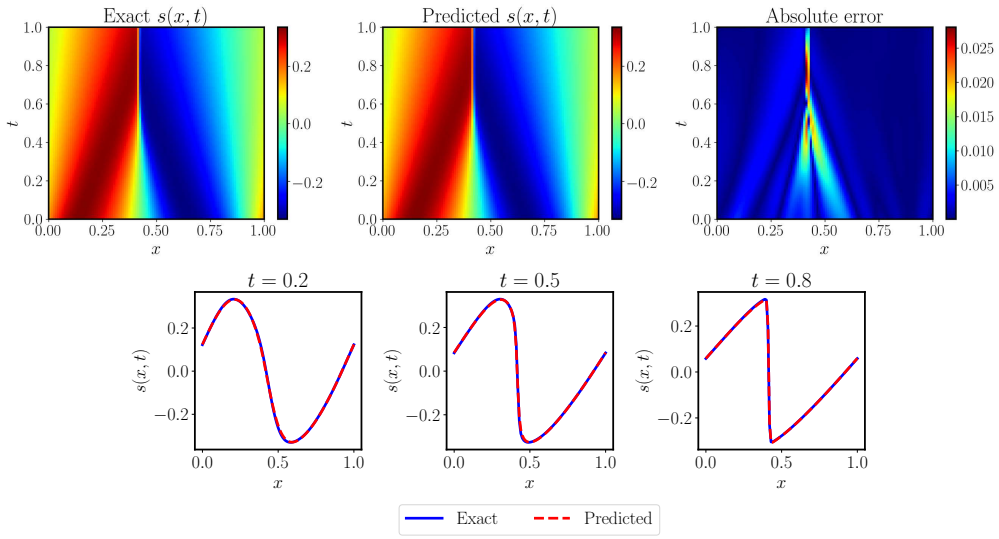

Figure 9: *Burger's equation: Top:* Exact solution versus the prediction of a physics-informed DeepONet represented by modified DeepONet architecture and trained using Algorithm 1 with $\alpha = \frac{1}{2}$ for the same example as in Figure 8.. *Bottom:* Comparison of the predicted and exact solutions corresponding to the three temporal snapshots at $t = 0.2, 0.5, 0.8$. The resulting relative L^2 error is 1.19%, which is 30x more accurate than the original formulation.

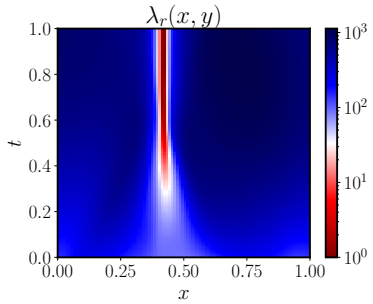

Figure 10: *Burger's equation:* Weight distribution defined in equation (4.11) for the same input sample as in Figure 8.

where $\boldsymbol{u} = (u, v)$ denotes the velocity field, and ∇p is the gradient of the pressure. In addition, $\Omega = [0, 1] \times [0, 1]$ denotes the computational domain, Γ represents an obstacle, and Λ_1 denotes wall boundaries, while Λ_2 and Λ_3 correspond to the inlet and the outlet of the channel, respectively. As illustrated in Figure 11, we assume a no-slip boundary condition at the walls and the obstacle surface, impose an initial velocity as $\boldsymbol{g} = (\sin(\pi y), 0)$ at the inlet, and a zero pressure condition at the outlet.

Our goal is to learn the solution operator G that maps different obstacle shapes $\partial\Gamma$ to the associated PDE solution triplet (u, v, p). In this example, we parameterize $\partial\Gamma$ by a family of ellipses centered at $(\frac{1}{2}, \frac{1}{2})$, i.e

$$\partial\Gamma = \partial\Gamma(\phi) = (a\cos(\phi) + \frac{1}{2}, b\sin(\phi) + \frac{1}{2}), \quad \phi \in [0, 2\pi), \tag{4.28}$$

where $a, b > 0$ denote the length of the long or the short axis.

We approximate the solution operator by a DeepONet $G_{\boldsymbol{\theta}}$ with 3-dimensional vector-valued outputs for representing u, v, p, respectively (see equation (2.14)),

$$\partial\Gamma \xrightarrow{G_{\boldsymbol{\theta}}} [G_{\boldsymbol{\theta}}^{(u)}, G_{\boldsymbol{\theta}}^{(v)}, G_{\boldsymbol{\theta}}^{(p)}]. \tag{4.29}$$

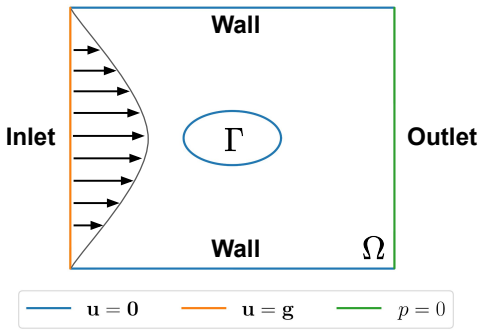

Figure 11: *Stokes equation:* Illustration of the computational domain and boundary conditions.

Now we can define the following PDE residuals

$$\mathcal{R}^{(1)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y) = -\frac{\partial^2 G^{(u)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial x^2} - \frac{\partial^2 G^{(u)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial y^2} + \frac{\partial G^{(p)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial x}, \tag{4.30}$$

$$\mathcal{R}^{(2)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y) = -\frac{\partial^2 G^{(v)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial x^2} - \frac{\partial^2 G^{(v)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial y^2} + \frac{\partial G^{(p)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial y}, \tag{4.31}$$

$$\mathcal{R}^{(3)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y) = \frac{\partial G^{(u)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial x} + \frac{\partial G^{(v)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma})(x,y)}{\partial y}, \tag{4.32}$$

where $\partial\boldsymbol{\Gamma} = [\partial\Gamma(\phi_1), \partial\Gamma(\phi_2), \ldots, \partial\Gamma(\phi_m)]$ denotes an input closed curve evaluated at evenly-spaced grid points $\{\phi\}_{i=1}^m$ in $[0, 2\pi]$.

According to equation (4.23) - (4.27), a physics-informed loss can be formulated as follows

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{k=1}^{3} \mathcal{L}^{(k)}_{\text{BC}}(\boldsymbol{\theta}) + \sum_{k=1}^{3} \mathcal{L}^{(k)}_{\text{PDE}}(\boldsymbol{\theta}), \tag{4.33}$$

where

$$\mathcal{L}^{(1)}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left[\left|G^{(u)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{bc1},j}, y^{(i)}_{\text{bc1},j})\right|^2 + \left|G^{(v)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{bc1},j}, y^{(i)}_{\text{bc1},j})\right|^2\right], \tag{4.34}$$

$$\mathcal{L}^{(2)}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left[\left|G^{(u)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{bc2},j}, y^{(i)}_{\text{bc2},j}) - \sin(\pi y^{(i)}_{\text{bc2},j})\right|^2 + \left|G^{(v)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{bc2},j}, y^{(i)}_{\text{bc2},j})\right|^2\right], \tag{4.35}$$

$$\mathcal{L}^{(3)}_{\text{BC}}(\boldsymbol{\theta}) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G^{(p)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{bc3},j}, y^{(i)}_{\text{bc3},j})\right|^2, \tag{4.36}$$

and

$$\mathcal{L}^{(k)}_{\text{PDE}}(\boldsymbol{\theta}) = \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|\mathcal{R}^{(k)}_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(i)})(x^{(i)}_{\text{r},j}, y^{(i)}_{\text{r},j})\right|^2, \quad k = 1, 2, 3. \tag{4.37}$$

Here, for each input sample $\partial\boldsymbol{\Gamma}^{(i)}$, $\{x^{(i)}_{\text{bc1},j}, y^{(i)}_{\text{bc1},j}\}_{j=1}^P$, $\{x^{(i)}_{\text{bc2},j}, y^{(i)}_{\text{bc2},j}\}_{j=1}^P$, $\{x^{(i)}_{\text{bc3},j}, y^{(i)}_{\text{bc3},j}\}_{j=1}^P$ and $\{x^{(i)}_{\text{r},j}, y^{(i)}_{\text{r},j}\}_{j=1}^Q$ are uniformly sampled at the boundaries $\Lambda_1 \cup \partial\Gamma$, Λ_2, Λ_3 and inside the domain Ω/Γ, respectively. In this example, we set $N = 1000, m = P = 100, Q = 2500$. To prepare the training data-set, we randomly sample a, b from uniform distribution $\mathcal{U}(0.05, 0.3)$ and obtain a set of input curves $\{\partial\boldsymbol{\Gamma}^{(i)}\}_{i=1}^N$ using equation (4.28). To generate a set of test data, we repeat the same procedure to obtain 500 new curves and obtain the corresponding PDE solutions using the FEniCS solver [40].

One can see that there are several terms in the loss function that it is almost impossible to assign weights manually. Therefore, here we will attempt any hyper-parameter tuning and only test the performance of Algorithm 1 for different

Method \ Architecture	MLP	Modified MLP	Modified DeepONet
$\lambda = 1$	$73.53\% \pm 3.75\%$ $93.17\% \pm 11.58\%$ $100.00\% \pm 1.06\%$	$73.72\% \pm 3.83\%$ $92.97\% \pm 11.84\%$ $100.00\% \pm 1.14\%$	$55.21\% \pm 7.79\%$ $62.07\% \pm 5.63\%$ $71.78\% \pm 6.21\%$
$\lambda = \frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}$	$54.12\% \pm 61.17\%$ $100.00\% \pm 0.01\%$ $90.45\% \pm 2.79\%$	$2.94\% \pm 1.12\%$ $10.36\% \pm 2.15\%$ $5.22\% \pm 2.91\%$	$\mathbf{1.69\% \pm 0.34\%}$ $\mathbf{6.05\% \pm 1.31\%}$ $\mathbf{3.83\% \pm 1.89\%}$
$\lambda = \sqrt{\frac{\|\boldsymbol{H}\|_\infty}{\text{diag}(\boldsymbol{H})}}$	$20.49\% \pm 7.42\%$ $30.87\% \pm 11.91\%$ $28.13\% \pm 7.51\%$	$17.10\% \pm 7.17\%$ $31.34\% \pm 14.11\%$ $22.73\% \pm 7.57\%$	$4.19\% \pm 1.80\%$ $10.86\% \pm 2.53\%$ $6.45\% \pm 2.98\%$

Table 7: *Stokes equation:* Test errors of trained physics-informed DeepONets using different weighting schemes and network architectures. We remark that, in each cell, three rows represents the test error corresponding to the velocity u, v and the pressure p, respectively.

choices of underlying network architectures. To this end, we reformulate the above loss function into a weighted fully-decoupled form

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N^*} \sum_{k=1}^{N^*} \lambda_k \left| T^{(k)}(\partial\Gamma^{(k)}, G_{\boldsymbol{\theta}}(\partial\boldsymbol{\Gamma}^{(k)})(x_k, y_k)) \right|^2, \tag{4.38}$$

where all weights $\{\lambda_k\}_{k=1}^{N^*}$ can be either set to 1, or be updated using Algorithm 1 at each training iteration.

We train the DeepONet by minimizing the loss function 4.33 for 2×10^5 iterations of gradient descent using Adam optimizer. The test error of the trained DeepONets with different network architectures and weighting schemes is summarized in Table 7. In contrast to the two previous examples, the results of "Fixed weights" are not included because there are too many terms in the loss function to tune weights manually. Among all trained models, one can see that the modified DeepONet architecture with NTK weights yields the best predictive accuracy. Remarkably, this result is 50x more accurate than the baseline physics-informed DeepONet [8]. Another key observation is that using NTK weights or the modified DeepONet architecture alone is not sufficient for accurately approximating the solution operator. This strongly suggests that appropriate weighting schemes and network architectures together play a vital role in enhancing the trainability, as well as the performance of physics-informed DeepONets. Figure 12 and Figure 13 provide a more detailed visual assessment of one representative predicted solution obtained by the best and the worst trained models. As it can be seen, the baseline DeepONet completely fails to predict the correct velocity and pressure field, while the modified DeepONet architecture produces predictions that achieve an excellent agreement with the corresponding reference solution. Additional predicted solutions for different input shapes are shown in Appendix Figure 25 - 27.

To further elucidate the important role played by the NTK weights, we compute the weight distribution over the domain during training. According to the PDE residual loss, we can define

$$\lambda_r^{(k)}(\partial\boldsymbol{\Gamma})(x, y) = \frac{\|\boldsymbol{H}\|_\infty}{\|\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\boldsymbol{\theta}}^{(k)}(\partial\boldsymbol{\Gamma})(x, y)\|_2^2}, \quad k = 1, 2, 3, \tag{4.39}$$

which denote the weights for the PDE residuals enforcing the momentum equations and the continuity equation, respectively. The weight distributions for the best trained model are presented in Figure 14. One can observe red belts surrounding the obstacle, which implies that these weights attain small magnitude near it. Another key observation is that the weight distribution $\lambda_{\text{r}}^{(3)}$ corresponding to the continuity equation is two orders of magnitude greater than the other two weight distributions $\lambda_{\text{r}}^{(1)}$ and $\lambda_{\text{r}}^{(2)}$, respectively; a fact that suggests the difficulty and the necessity of imposing the divergence-free condition for physics-informed DeepONets. More visualizations for different obstacle geometries are provided in Appendix Figure 25 - 27 from which a similar behavior can be observed.

5 Discussion

This work proposes a novel training algorithm to calibrate the convergence rate and back-propagated gradients of DeepONets and physics-informed DeepONets, by assigning appropriate weights to each individual term in their loss

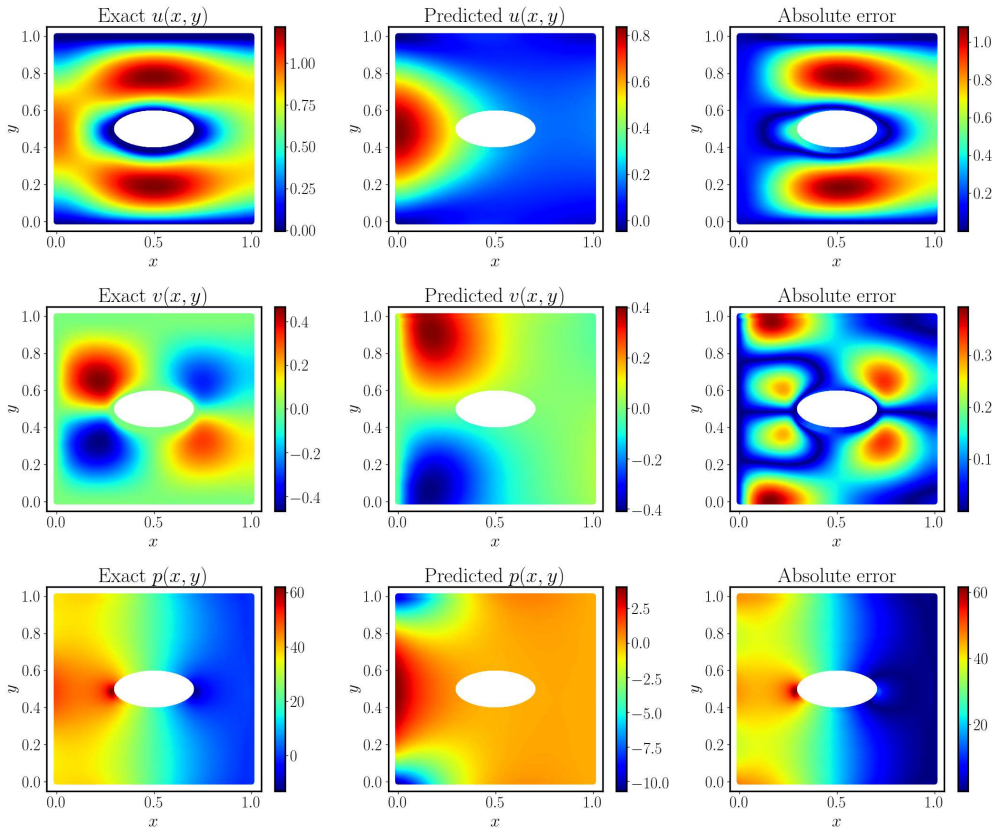

Figure 12: *Stokes equation:* Exact solution versus the prediction of a trained conventional physics-informed DeepONet for a representative example in the test data-set. The resulting relative L^2 errors are $74.32\%, 82.64\%, 99.63\%$ for u, v, p, respectively. The trained model completely fails to predict the correct solution.

function during training. This is accomplished by performing a rigorous analysis of the training dynamics of DeepONets via Neural Tangent Kernel (NTK) theory. In addition, we design a novel DeepONet architecture which better preserves the propagation of inputs signals through the network's layers. Our empirical evidence suggests that this architecture is consistently better than conventional DeepONets in terms of approximating abstract non-linear operators. Taken together, our developments provide new insights into the training of DeepONets and physics-informed DeepONets and yield significant improvement in their predictive accuracy by a factor of 10-50x across a range of benchmarks involving learning the solution operator of parametric PDEs with inherently different dynamics.

Despite some promising results reported here, there are numerous open questions worth further investigation. From a theoretical standpoint, in contrast to conventional DeepONets guaranteed by universal approximation theorem of operators [4, 1], little is known about the approximation properties of the physics-informed DeepONets trained by minimizing a loss function where PDE constraints are encoded. Given a series of established a-priori error estimates for physics-informed neural networks (PINNs) [41, 42], in a parallel thrust, it would be interesting to provide rigorous mathematical justifications for physics-informed DeepONets. From a methodological standpoint, there is still plenty of room for improving the performance of physics-informed DeepONets. For one thing, the proposed training algorithm is based on a simple NTK analysis of full-batch gradient descent. The effect of mini-batch gradient descent with momentum on the training dynamics remains poorly understood. From an applications standpoint, physics-informed DeepONets can be employed as a fast and differentiable surrogate for tackling general PDE-constrained optimization problems that routinely arise in science and engineering (e.g., design and control optimization problems), which generally require numerous evaluations of costly forward solvers or experiments.

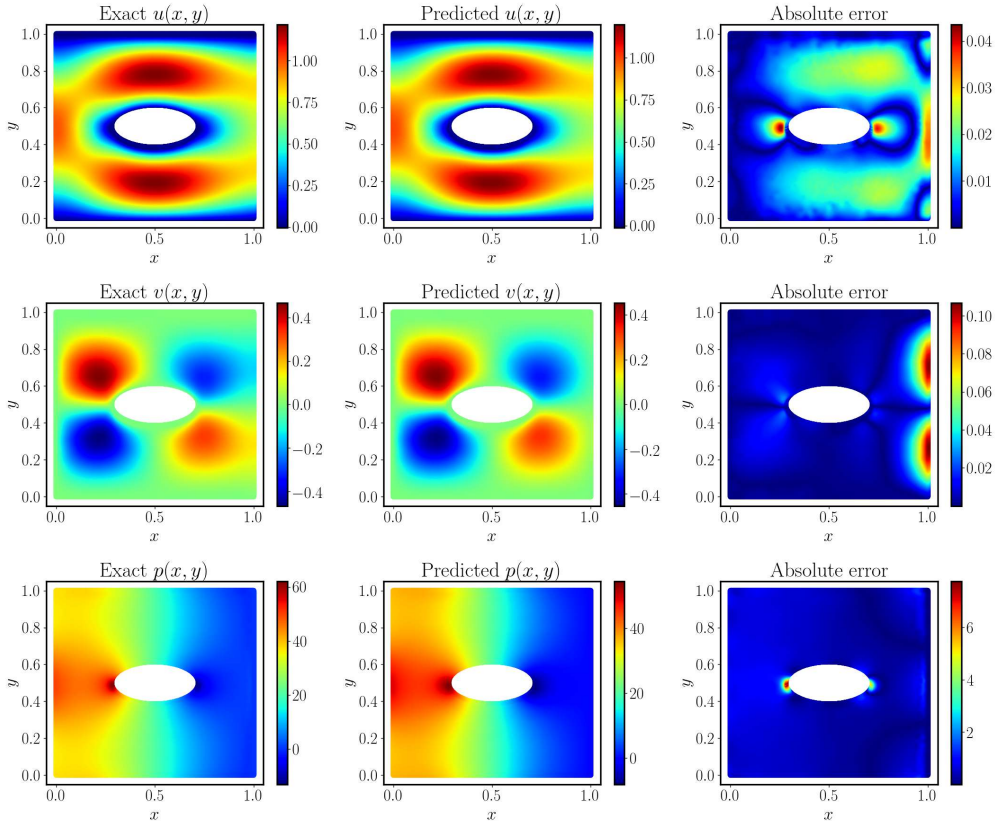

Figure 13: *Stokes equation:* Exact solution versus the prediction of a physics-informed DeepONet represented by modified DeepONet architecture and trained using Algorithm 1 for the same example as in Figure 12. The resulting relative L^2 errors are $1.88\%, 5.74\%, 2.41\%$ for u, v, p, respectively.

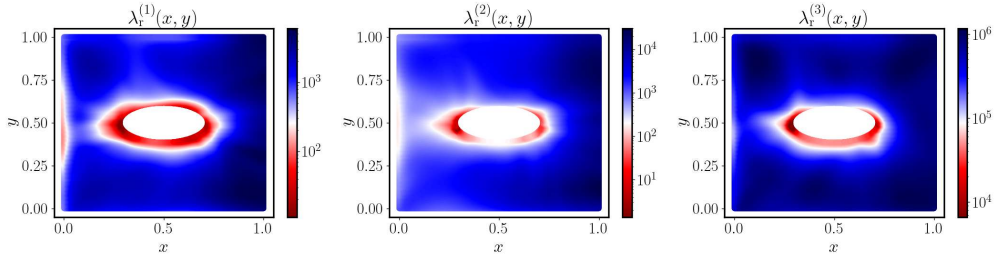

Figure 14: *Stokes equation:* Weight distribution defined in equation (4.39) for the same input sample as in Figure 12.

Acknowledgments

This work received support from DOE grant DE-SC0019116, AFOSR grant FA9550-20-1-0060, and DOE-ARPA grant DE-AR0001201. We would also like to thank the developers of the software that enabled our research, including JAX [43], Matplotlib [44], and NumPy [45].

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

SW and PP conceptualized the research and designed the numerical studies. SW implemented the methods and conducted the numerical experiments. HW assisted with the numerical studies. PP provided funding and supervised all aspects of this work. SW and PP wrote the manuscript.

Data and materials availability

All methods needed to evaluate the conclusions in the paper are present in the paper and/or the Appendix. All code and data accompanying this manuscript will be made publicly available at `https://github.com/PredictiveIntelligenceLab/ImprovedDeepONets`.

References

[1] Samuel Lanthaler, Siddhartha Mishra, and George Em Karniadakis. Error estimates for DeepONet: A deep learning framework in infinite dimensions. *arXiv preprint arXiv:2102.09618*, 2021.

[2] Nikola Kovachki, Samuel Lanthaler, and Siddhartha Mishra. On universal approximation and error bounds for fourier neural operators. *arXiv preprint arXiv:2107.07562*, 2021.

[3] Annan Yu, Chloé Becquey, Diana Halikias, Matthew Esmaili Mallory, and Alex Townsend. Arbitrary-depth universal approximation theorems for operator neural networks. *arXiv preprint arXiv:2109.11354*, 2021.

[4] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[5] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

[6] Houman Owhadi. Do ideas have shape? plato's theory of forms as the continuous limit of artificial neural networks. *arXiv preprint arXiv:2008.03920*, 2020.

[7] Hachem Kadri, Emmanuel Duflos, Philippe Preux, Stéphane Canu, Alain Rakotomamonjy, and Julien Audiffren. Operator-valued kernels for learning from functional response data. *Journal of Machine Learning Research*, 17(20):1–54, 2016.

[8] Sifan Wang, Hanwen Wang, and Paris Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. *Science Advances*, 7(40):eabi8605, 2021.

[9] Sifan Wang and Paris Perdikaris. Long-time integration of parametric evolution equations with physics-informed DeepONets. *arXiv preprint arXiv:2106.05384*, 2021.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[13] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016.

[14] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[15] P Clark Di Leoni, Lu Lu, Charles Meneveau, George Karniadakis, and Tamer A Zaki. DeepONet prediction of linear instability waves in high-speed boundary layers. *arXiv preprint arXiv:2105.08697*, 2021.

[16] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[17] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[18] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why PINNs fail to train: A neural tangent kernel perspective. *arXiv preprint arXiv:2007.14527*, 2020.

[19] Levi McClenny and Ulisses Braga-Neto. Self-adaptive physics-informed neural networks using a soft attention mechanism. *arXiv preprint arXiv:2009.04544*, 2020.

[20] Sifan Wang and Paris Perdikaris. Deep learning of free boundary and stefan problems. *Journal of Computational Physics*, 428:109914, 2021.

[21] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

[22] Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.

[23] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International Conference on Machine Learning*, pages 242–252. PMLR, 2019.

[24] Yuan Cao, Zhiying Fang, Yue Wu, Ding-Xuan Zhou, and Quanquan Gu. Towards understanding the spectral bias of deep learning. *arXiv preprint arXiv:1912.01198*, 2019.

[25] Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, and Zheng Ma. Frequency principle: Fourier analysis sheds light on deep neural networks. *arXiv preprint arXiv:1901.06523*, 2019.

[26] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310, 2019.

[27] Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in neural information processing systems*, pages 8572–8583, 2019.

[28] Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks. *arXiv preprint arXiv:2012.10047*, 2020.

[29] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[30] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[31] Shengze Cai, Zhicheng Wang, Lu Lu, Tamer A Zaki, and George Em Karniadakis. Deepm&mnet: Inferring the electroconvection multiphysics fields based on operator approximation by neural networks. *arXiv preprint arXiv:2009.12935*, 2020.

[32] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] Stanislav Fort, Gintare Karolina Dziugaite, Mansheej Paul, Sepideh Kharaghani, Daniel M Roy, and Surya Ganguli. Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the neural tangent kernel. *arXiv preprint arXiv:2010.15110*, 2020.

[35] Guillaume Leclerc and Aleksander Madry. The two regimes of deep network training. *arXiv preprint arXiv:2002.10376*, 2020.

[36] Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

[37] Yibo Yang and Paris Perdikaris. Physics-informed deep generative models. *arXiv preprint arXiv:1812.03511*, 2018.

[38] Tobin A Driscoll, Nicholas Hale, and Lloyd N Trefethen. Chebfun guide, 2014.

[39] Steven M Cox and Paul C Matthews. Exponential time differencing for stiff systems. *Journal of Computational Physics*, 176(2):430–455, 2002.

[40] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[41] Yeonjong Shin, Jérôme Darbon, and George Em Karniadakis. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. 2020.

[42] Siddhartha Mishra and Roberto Molinaro. Estimates on the generalization error of physics informed neural networks (pinns) for approximating pdes. *arXiv preprint arXiv:2006.16144*, 2020.

[43] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[44] John D Hunter. Matplotlib: A 2D graphics environment. *IEEE Annals of the History of Computing*, 9(03):90–95, 2007.

[45] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

A Nomenclature

Table 8 summarizes the main symbols and notation used in this work.

Notation	Description
$\boldsymbol{u}(\cdot)$	an input function
$\boldsymbol{s}(\cdot)$	a solution to a parametric PDE
G	an operator
$G_{\boldsymbol{\theta}}$	an unstacked DeepONet representation of the operator G
$\boldsymbol{\theta}$	all trainable parameters of a DeepONet
$\{\boldsymbol{x}_i\}_{i=1}^m$	m sensor points where input functions $\boldsymbol{u}(\boldsymbol{x})$ are evaluated
$[u(\boldsymbol{x}_1), u(\boldsymbol{x}_2), \dots, u(\boldsymbol{x}_m)]$	an input of the branch net, representing the input function u
$\boldsymbol{y}$	an input of the trunk net, a point in the domain of $G(u)$
N	number of input samples in the training data-set
m	number of locations for evaluating the input functions u
P	number of locations for evaluating the output functions $G(u)$
Q	number of collocation points for evaluating the PDE residual
PDE	Partial differential equation
GRF	a Gaussian random field
SDF	a signed distance function
MLP	Multi-layer perceptron
l	length scale of a Gaussian random field
k	output scale of a Gaussian random field
$\boldsymbol{K}$	Neural Tangent Kernel of an MLP
$\boldsymbol{H}$	Neural Tangent Kernel of a physics-informed DeepONet

Table 8: *Nomenclature*: Summary of the main symbols and notation used in this work.

B Hyper-parameter settings

Case	Input function space	m	P	Q	# u Train	# u Test	Iterations
Anti-derivative	GRF($l = 0.2, k \in [10^{-2}, 10^2]$)	10^2	1	-	10^4	10^3	4×10^4
Advection equation	GRF($l = 0.2$)	10^2	10^2	2×10^3	10^3	10^2	3×10^5
Burger's equation	$\mathcal{N}\left(0, 25^2(-\Delta + 5^2 I)^{-4}\right)$	10^2	10^2	2×10^3	10^3	10^2	2×10^5
Stokes equation	Ellipse	10^2	10^2	2×10^3	10^3	10^2	2×10^5

Table 9: Default hyper-parameter settings for each benchmark employed in this work (unless otherwise stated).

Case	Architecture	Trunk depth	Trunk width	Branch depth	Branch width
Advection equation	MLP	7	100	7	100
	Modified MLP	7	100	7	100
	Modified DeepONet	7	100	7	100
Burger's equation	MLP	7	100	7	100
	Modified MLP	7	100	7	100
	Modified DeepONet	7	100	7	100
Stokes equation	MLP	7	100	7	100
	Modified MLP	7	100	7	100
	Modified DeepONet	7	100	7	100

Table 10: Physics-informed DeepONet architectures for each benchmark employed in this work (unless otherwise stated).

C Computational cost

Training: Table C summarizes the computational cost (hours) of training physics-informed DeepONet models with different network architectures and weighting schemes. The size of different models as well as network architectures are listed Table 10, respectively. All networks are trained using a single NVIDIA RTX A6000 graphics card.

Case	Architecture	Weighting scheme	Training time (hours)
Advection equation	MLP	Fixed weights	1.98
		NTK weights	4.63
	Modified MLP	Fixed weights	2.88
		NTK weights	7.65
	Modified DeepONet	Fixed weights	3.25
		NTK weights	7.51
Burger's equation	MLP	Fixed weights	2.21
		NTK weights	7.00
	Modified MLP	Fixed weights	3.05
		NTK weights	9.82
	Modified DeepONet	Fixed weights	3.88
		NTK weights	9.49
Stokes equation	MLP	Fixed weights	2.36
		NTK weights	14.38
	Modified MLP	Fixed weights	4.67
		NTK weights	20.86
	Modified DeepONet	Fixed weights	5.60
		NTK weights	20.58

Table 11: Computational cost (hours) for training physics-informed DeepONet models across the different becnhmarks and architectures employed in this work. Reported timings are obtained on a single NVIDIA RTX A6000 graphics card.

D Proof of Lemma 3.2

Proof. First we observe that $\boldsymbol{H}(\boldsymbol{\theta})$ is a Gram matrix. Indeed, we define

$$v_i = T^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i)), \quad i = 1.2, \ldots, N^*. \tag{D.1}$$

Then by the definition of $\boldsymbol{H}(\boldsymbol{\theta})$, we have

$$\boldsymbol{H}_{ij}(\boldsymbol{\theta}) = \langle v_i, v_j \rangle \tag{D.2}$$

(a) By the definition of a Gram matrix.

(b) Let $\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty = \langle v_i, v_j \rangle$ for some i, j and $\langle v_k, v_k \rangle = \max_{1 \le k \le N^*} \boldsymbol{H}_{kk}(\boldsymbol{\theta})$/ for some k. Then we have

$$\max_{1 \le k \le N^*} \boldsymbol{H}_{kk}(\boldsymbol{\theta}) = \langle v_k, v_k \rangle \le \|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty = \langle v_i, v_j \rangle \le \|v_i\| \|v_j\| \le \|v_k\|^2 = \langle v_k, v_k \rangle \tag{D.3}$$

Therefore, we have

$$\|\boldsymbol{H}(\boldsymbol{\theta})\|_\infty = \max_{1 \le k \le N^*} \boldsymbol{H}_{kk}(\boldsymbol{\theta}) \tag{D.4}$$

□

E Proof of Lemma 3.4

Proof. Recall the definition of the loss function 3.6,

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N^*}\sum_{i=1}^{N^*}\left|T^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))\right|^2 \tag{E.1}$$

Now we consider the corresponding gradient flow

$$\frac{d\boldsymbol{\theta}}{dt} = -\nabla\mathcal{L}(\boldsymbol{\theta}) \tag{E.2}$$

$$= -\frac{4}{N^*}\sum_{k=1}^{N^*}\left(T^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))\right)\frac{dT^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))}{d\boldsymbol{\theta}} \tag{E.3}$$

For $1 \le j \le N^*$, note that

$$\frac{dT^{(j)}(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j))}{dt} \tag{E.4}$$

$$= \frac{dT^{(j)}(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j))}{d\boldsymbol{\theta}}\cdot\frac{d\boldsymbol{\theta}}{dt} \tag{E.5}$$

$$= \frac{dT^{(j)}(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j))}{d\boldsymbol{\theta}}\cdot\left[-\frac{4}{N^*}\sum_{i=1}^{N^*}\left(T^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))\right)\frac{dT^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))}{d\boldsymbol{\theta}}\right] \tag{E.6}$$

$$= -\frac{4}{N^*}\sum_{i=1}^{N^*}\left(T^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))\right)\left\langle\frac{dT^{(i)}(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i))}{d\boldsymbol{\theta}}, \frac{dT^{(j)}(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j))}{d\boldsymbol{\theta}}\right\rangle \tag{E.7}$$

By Definition 3.1, 3.3, we conclude that

$$\frac{d\boldsymbol{T}\left(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y})\right)}{dt} = -\frac{4}{N^*}\boldsymbol{K}(\boldsymbol{\theta})\cdot\boldsymbol{T}(\boldsymbol{U}(\boldsymbol{X}), G_{\boldsymbol{\theta}(t)}(\boldsymbol{U})(\boldsymbol{Y}))) \tag{E.8}$$

where $\boldsymbol{K}$ is a $N^*\times N^*$ matrix whose entries are given by

$$\boldsymbol{K}_{ij}(\boldsymbol{\theta}) = \left\langle\frac{dT^{(i)}\left(u^{(i)}(\boldsymbol{x}_i), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(\boldsymbol{y}_i)\right)}{d\boldsymbol{\theta}}, \frac{dT^{(j)}\left(u^{(j)}(\boldsymbol{x}_j), G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(j)})(\boldsymbol{y}_j)\right)}{d\boldsymbol{\theta}}\right\rangle \tag{E.9}$$

□

F Antiderivative

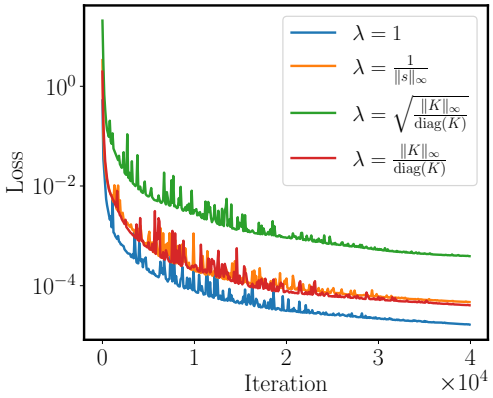

Figure 15: *Anti-derivative operator:* Training loss convergence of a DeepONet models using different loss schemes for 4×10^4 40,000 iterations of gradient descent using the Adam optimizer. Here we remark that all losses are weighted, and, therefore, the magnitude of these losses is not informative.

G Advection equation

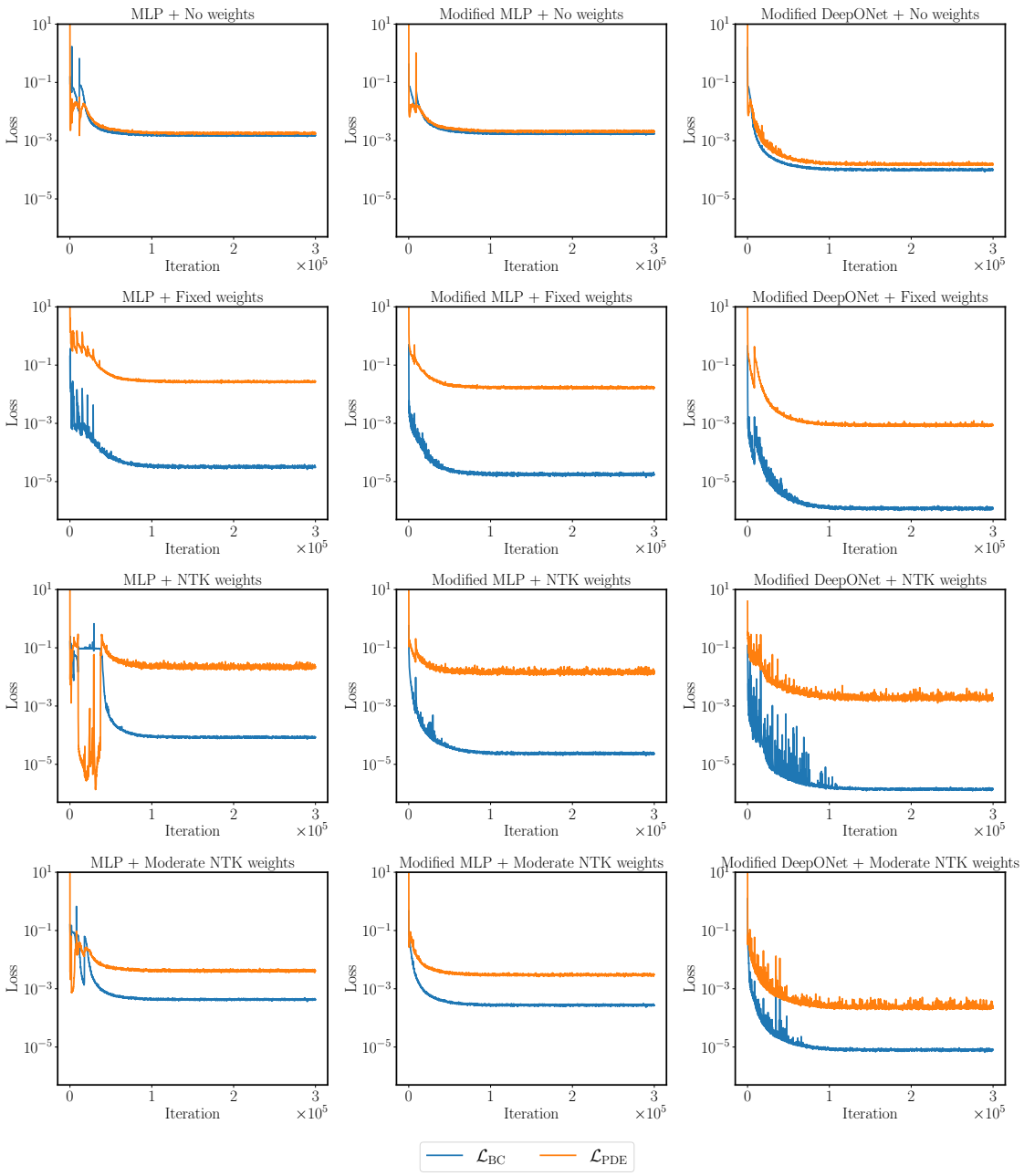

Figure 16: *Advection equation:* Training loss convergence of a DeepONet models using different DeepONet architectures and weighting schemes for 3×10^5 iterations of gradient descent using the Adam optimizer. Here we remark that all losses are unweighted.

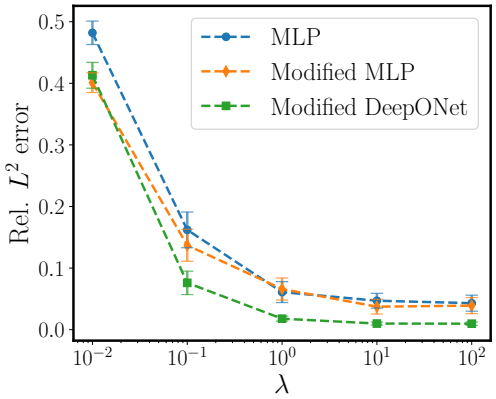

Figure 17: *Advection equation:* Relative L^2 error of physics-informed DeepONets represented by different architectures and trained using different fixed weights $\lambda_{\text{bc}} = \lambda_{\text{ic}} = \lambda \in [10^{-2}, 10^2]$, averaged over the test data-set.

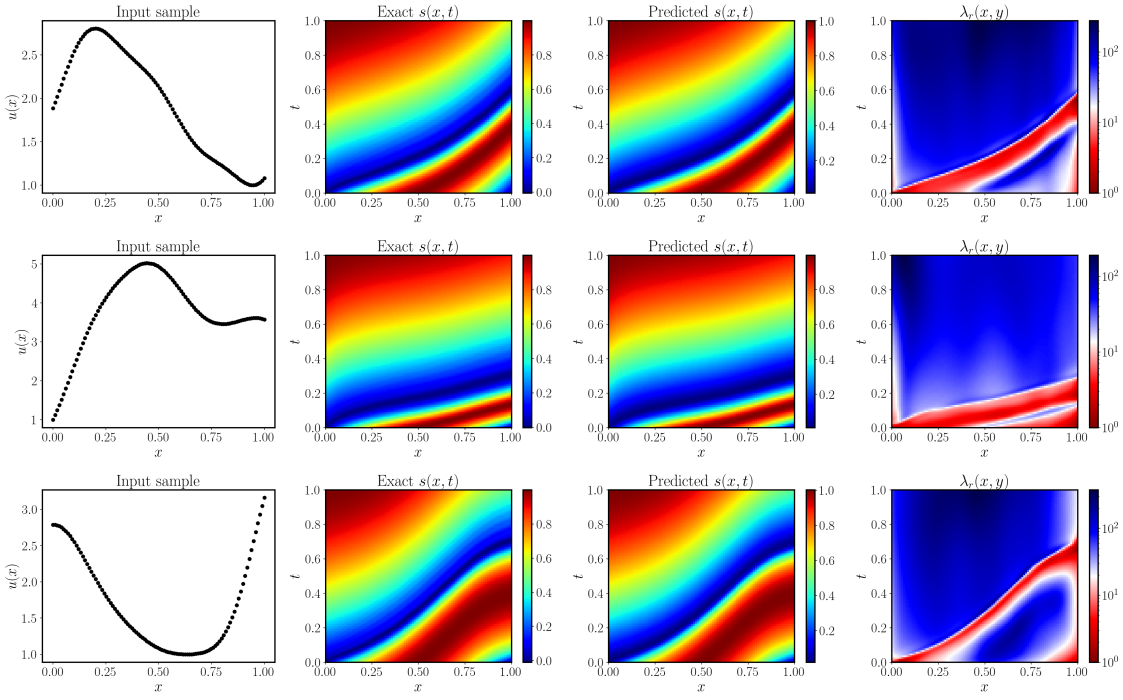

Figure 18: *Advection equation:* Predicted solution of the best trained physics-informed DeepONet for three different examples in the test data-set.

H Burger's equation

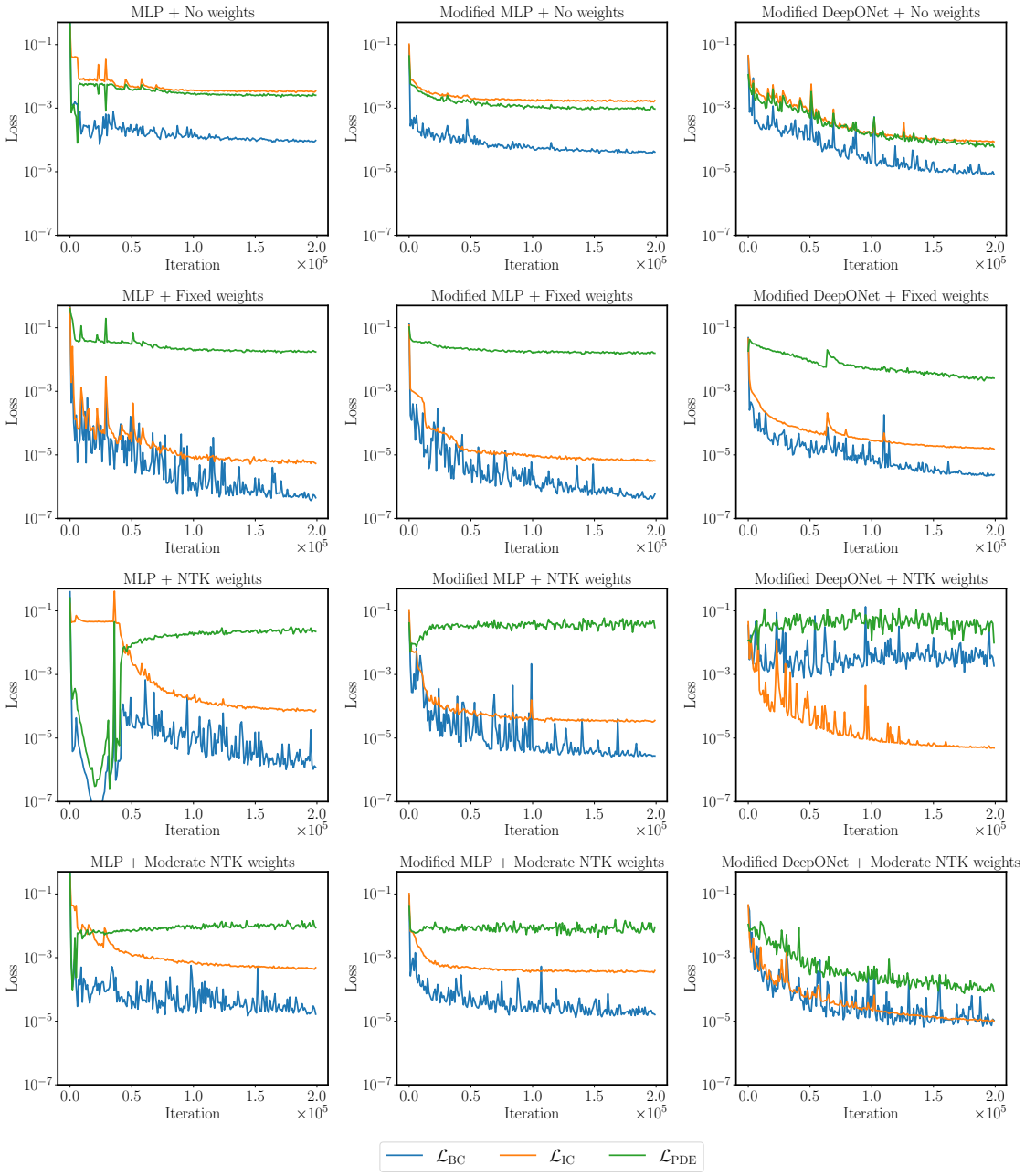

Figure 19: *Burger's equation:* Training loss convergence of a DeepONet models using different DeepONet architectures and weighting schemes for 2×10^5 iterations of gradient descent using the Adam optimizer. Here we remark that all losses are unweighted.

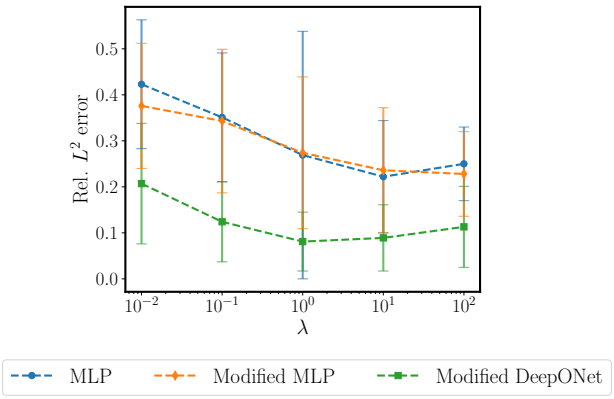

Figure 20: *Burger's equation:* Relative L^2 error of physics-informed DeepONets represented by different architectures and trained using different fixed weights $\lambda_{\text{bc}} = \lambda_{\text{ic}} = \lambda \in [10^{-2}, 10^2]$, averaged over the test data-set.

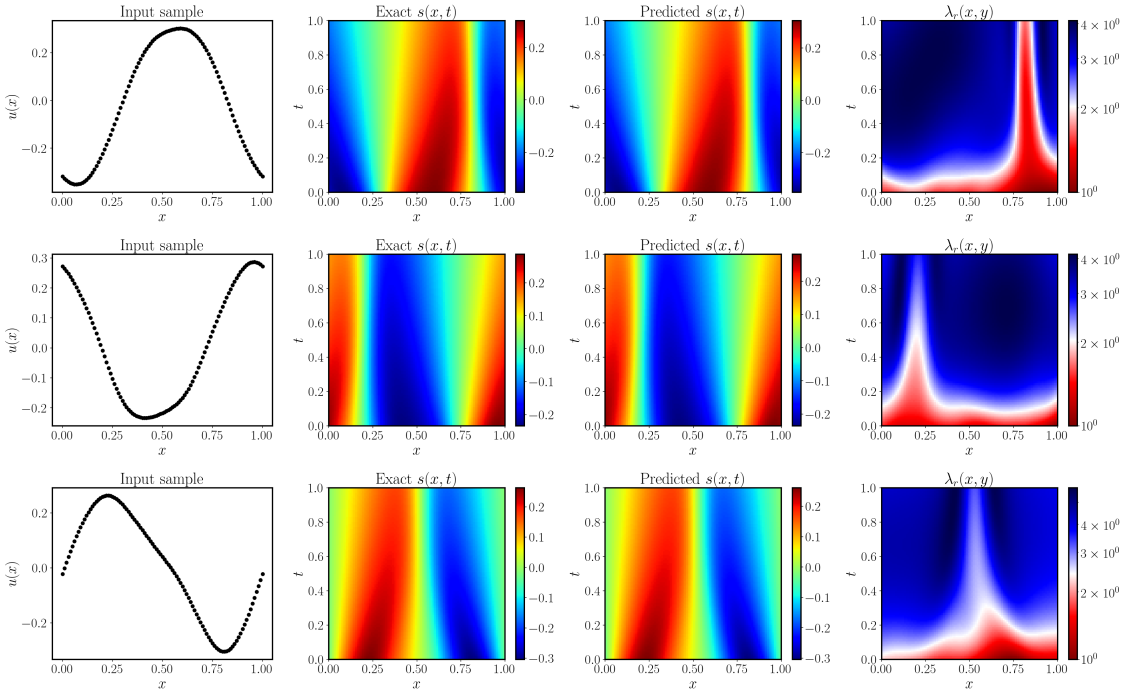

Figure 21: *Burger's equation ($\nu = 0.01$):* Predicted solution of the best trained physics-informed DeepONet for three different examples in the test data-set.

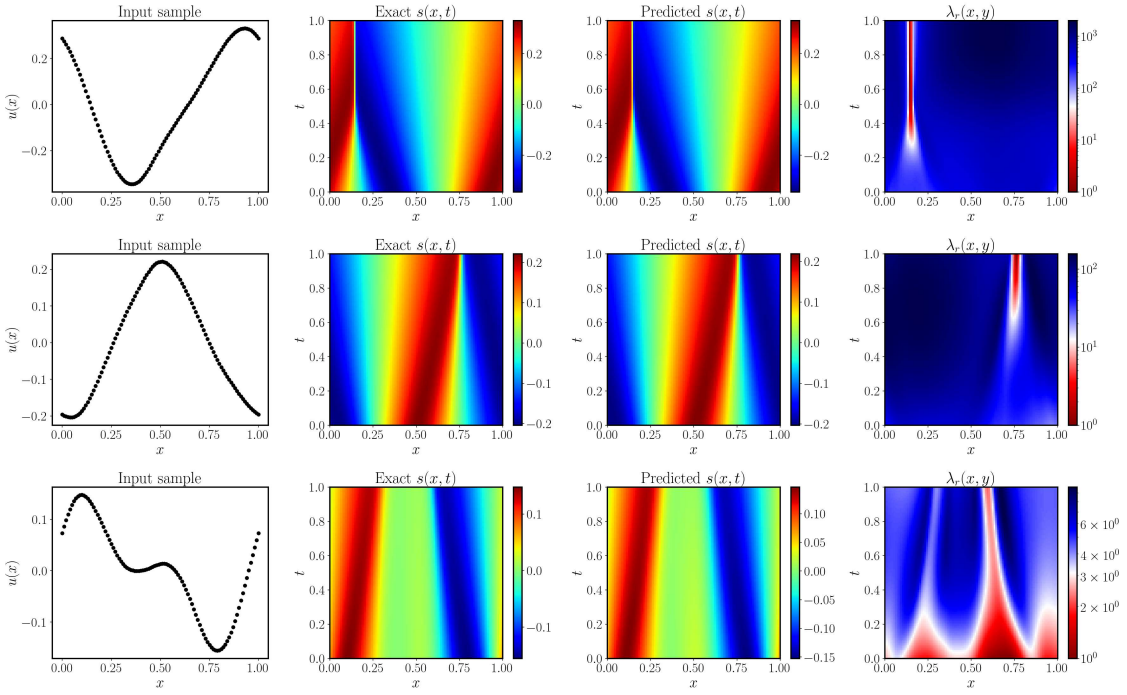

Figure 22: *Burger's equation ($\nu = 0.001$):* Predicted solution of the best trained physics-informed DeepONet for three different examples in the test data-set.

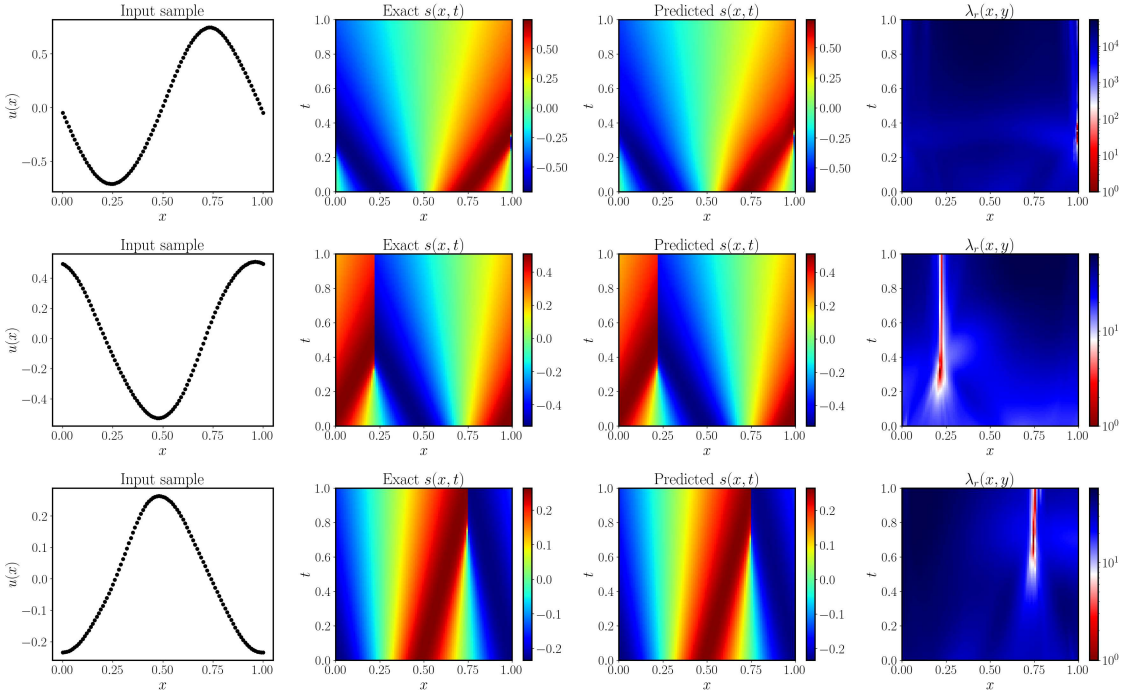

Figure 23: *Burger's equation ($\nu = 0.0001$):* Predicted solution of the best trained physics-informed DeepONet for three different examples in the test data-set.

I Stokes flow

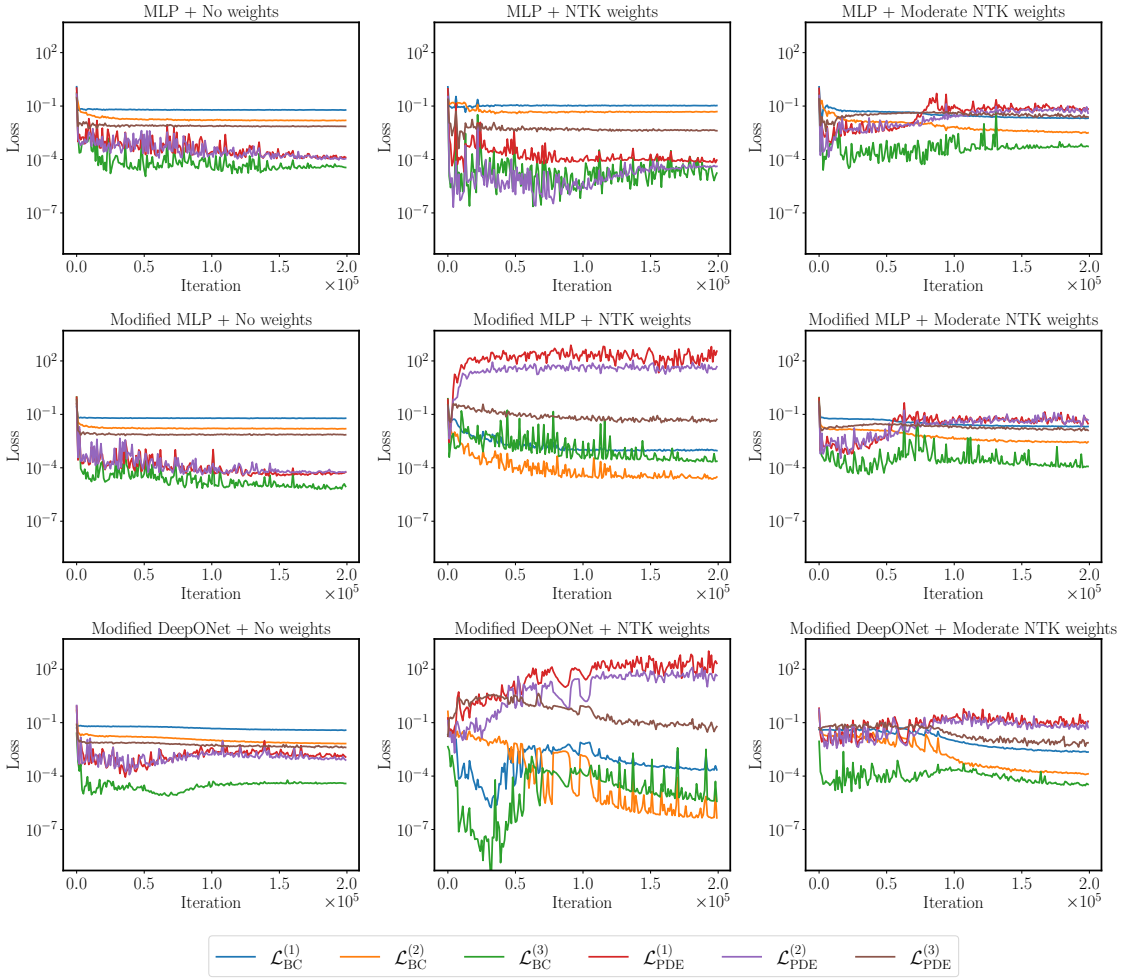

Figure 24: *Stokes equation:* Training loss convergence of a DeepONet models using different DeepONet architectures and weighting schemes for 2×10^5 iterations of gradient descent using the Adam optimizer. Here we remark that all losses are unweighted.

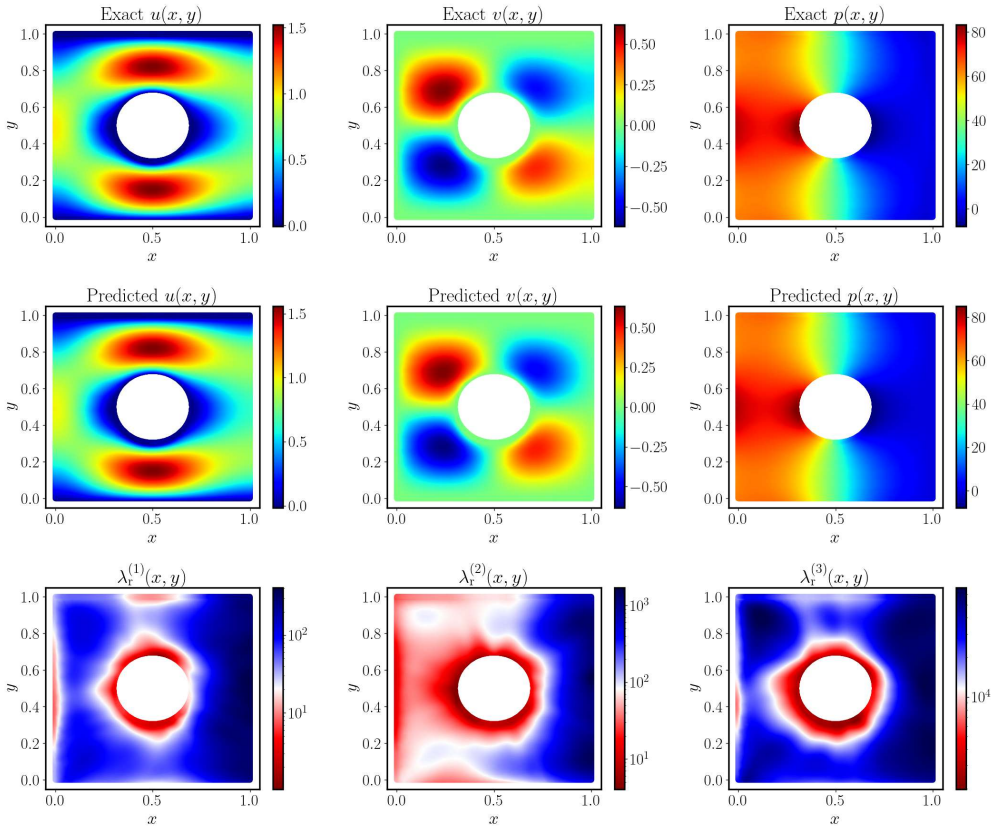

Figure 25: *Stokes equation:* Predicted solution of the best trained physics-informed DeepONet for one example in the test data-set.

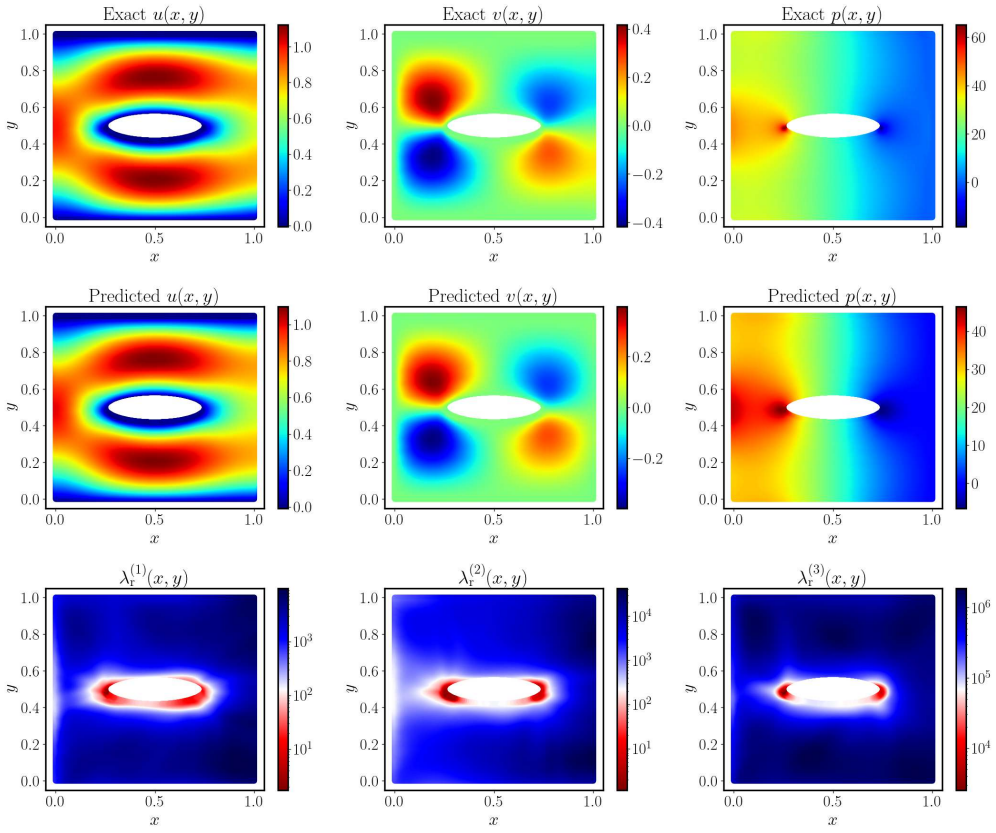

Figure 26: *Stokes equation:* Predicted solution of the best trained physics-informed DeepONet for one example in the test data-set.

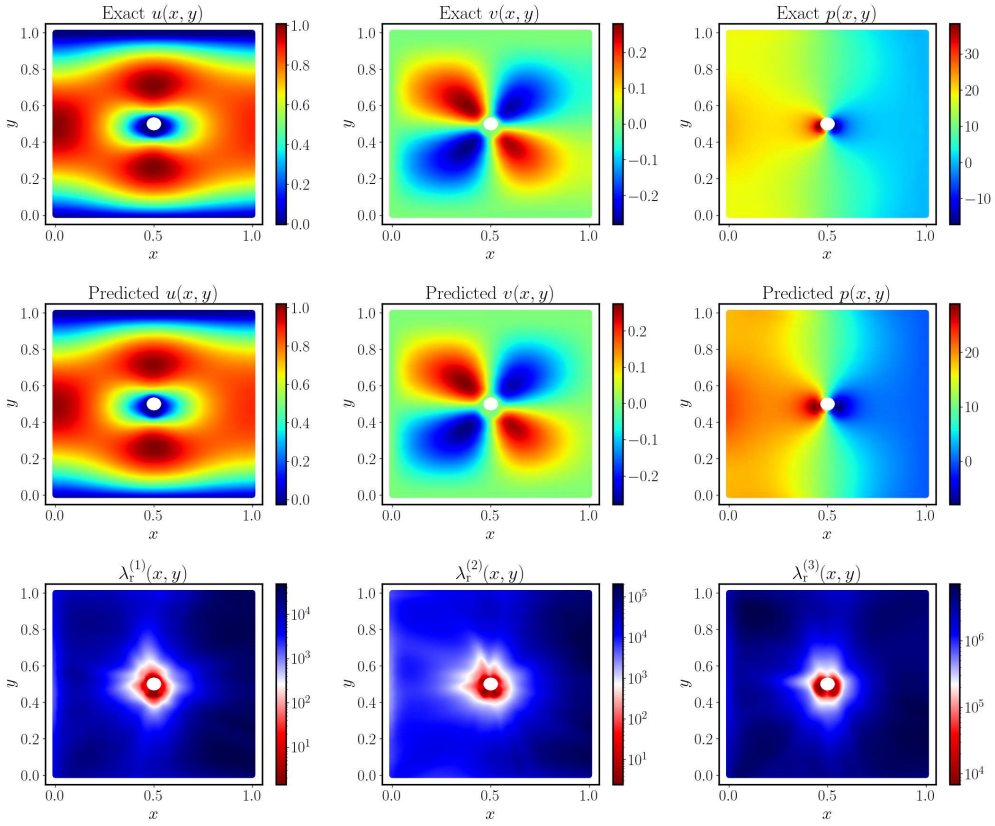

Figure 27: *Stokes equation:* Predicted solution of the best trained physics-informed DeepONet for one example in the test data-set.