Journal of Computational Physics 505 (2024) 112904

Contents lists available at ScienceDirect

Journal of Computational Physics

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

Zero coordinate shift: Whetted automatic differentiation for physics-informed operator learning

Kuangdai Leng [a], Mallikarjun Shankar [b], Jeyan Thiyagalingam [a]

[a] Scientific Computing Department, STFC, Rutherford Appleton Laboratory, Didcot, OX11 0QX, UK
[b] Oak Ridge National Laboratory, PO Box 2008, Oak Ridge, TN 37831-6012, United States of America

ARTICLE INFO

Keywords:
Deep learning
Physics-informed
Partial differential equations
Automatic differentiation

ABSTRACT

Automatic differentiation (AD) is a critical step in physics-informed machine learning, required for computing the high-order derivatives of network output w.r.t. coordinates of collocation points. In this paper, we present a novel and lightweight algorithm to conduct AD for physics-informed operator learning, which we call the trick of Zero Coordinate Shift (ZCS). Instead of making all sampled coordinates as leaf variables, ZCS introduces only one scalar-valued leaf variable for each spatial or temporal dimension, simplifying the wanted derivatives from "many-roots-many-leaves" to "one-root-many-leaves" whereby reverse-mode AD becomes directly utilisable. It has led to an outstanding performance leap by avoiding the duplication of the computational graph along the dimension of functions (physical parameters). ZCS is easy to implement with current deep learning libraries; our own implementation is achieved by extending the DeepXDE package. We carry out a comprehensive benchmark analysis and several case studies, training physics-informed DeepONets to solve partial differential equations (PDEs) without data. The results show that ZCS has persistently reduced GPU memory consumption and wall time for training by an order of magnitude, and such reduction factor scales with the number of functions. As a low-level optimisation technique, ZCS imposes no restrictions on data, physics (PDE) or network architecture and does not compromise training results from any aspect.

1. Introduction

Physics-informed machine learning offers a versatile framework for the emulation and inversion of partial differential equations (PDEs) in physics [1,2], extensible to other types of differential equations such as stochastic differential equations [3], integro-differential equations [4], and fractional differential equations [5]. This deep learning-based paradigm has originated from the physics-informed neural networks (PINNs) [6–9], targeted at learning a map $x \mapsto u$, with x being the coordinates of the collocation points sampled from the domain and u the solution field. An innate limitation of PINNs is the absence of generalisability over the variation of physical parameters, such as material properties, external sources, and initial and boundary conditions. Such a limitation has motivated the physics-informed neural operators (PINOs), such as the deep operator networks (DeepONets) [10] and the physics-informed Fourier neural operators (FNOs) [11],[1] which learn a map $p(x) \mapsto u(x)$, with $p(x)$ being any physical parameter of interest.

E-mail address: kuangdai.leng@stfc.ac.uk (K. Leng).

[1] In this paper, we use PINOs to refer to any neural networks for physics-informed operator learning, not specifically to the physics-informed FNOs [11] also called PINOs by the authors.

https://doi.org/10.1016/j.jcp.2024.112904
Received 13 November 2023; Received in revised form 21 February 2024; Accepted 27 February 2024

Available online 2 March 2024

In brief, PINNs learn a map between two vector spaces and PINOs a map between two function spaces; in many contexts, a PINO degenerates to a PINN given an invariant $p(x)$.

The term "physics-informed" in PINNs and PINOs has a specific meaning: embedding the target PDE and its initial and boundary conditions into the loss function. Such embedded information is supposed to aid a neural network in understanding the underlying physics (rather than just fitting data) whereby to achieve a lower generalisation error with less training data. To compute the loss functions associated with a PDE and any of its non-Dirichlet boundary conditions, one needs the high-order derivatives of the network output w.r.t. the coordinates. For example, bending of an elastic plate and the stream function for incompressible 2-D fluids both involve a fourth-order PDE. Besides, the gradient of the PDE itself may serve as an even stronger regularisation [12], which further increments the required orders of derivatives. A powerful tool for derivative computation is automatic differentiation (AD), a pivotal technique behind today's deep learning [13–16] and one of the major facilitators of physics-informed machine learning [1,2,9].

Technical advancements have been thriving since the establishment of modern PINNs [8,9], e.g., multi-scale and multi-frequency models [17,18], domain decomposition techniques [19,20], adaptive activation functions [21] and data sampling [22,23], and embedding physics as hard constraints [24–26]. Despite such continuing efforts, we feel that less attention has been paid to AD, especially in view of that the coordinate derivatives in physics-informed machine learning differ from the well-known backpropagation (i.e., calculating the gradient of the loss function w.r.t. network weights) in many ways. A comprehensive survey of related work will be provided in the next section.

In this paper, we present a novel algorithm to reshape AD for operator learning, which can reduce both GPU memory consumption and wall time for training by an order of magnitude. Our algorithm, which we refer to as Zero Coordinate Shift (ZCS), inventively introduces one scalar-valued leaf variable for each dimension, simplifying the wanted coordinate derivatives from "many-roots-many-leaves" to "one-root-many-leaves" whereby the most powerful reverse-mode AD can be exploited to the maximum. Our algorithm lies at a lower level than the neural network (i.e., agnostic to the forward pass), imposing no restrictions on data, point sampling or network architecture. Furthermore, the implementation of ZCS is straightforward; our own implementation is by extending the DeepXDE package [27] with a very few overridden methods.

The remainder of this paper is organised as follows. In Section 2, we provide a more insightful review of the related work for further justification of our motivation. In Section 3, we will describe the theory and implementation of ZCS. We report our experiments in Section 4, including a benchmark analysis exhibiting the scaling behaviours of ZCS and a few PDE operators learned without data. Identified limitations are then described in Section 5 before the paper being concluded in Section 6.

2. Related work

In this section, we summarise the existing studies related to computing the coordinate derivatives in physics-informed machine learning. For clarity, we divide them into three categories, as discussed below.

2.1. Finite discretisation

A number of approaches can be available to (partially) replace AD under certain circumstances. Finite difference (FD), for instance, is a popular choice when the collocation points reside on a structured gird, such as for the convolution-based PhyGeoNet [28] and PhyCRNet [29]. A structured gird is however not always available, e.g., when the geometry of the domain is irregular or the sensors for data collection are randomly located. A few methods have been proposed to overcome this difficulty. Given an unstructured point cloud, the can-PINNs [30] make FD utilisable by adding many extra points in the forward pass to create small local grids anchored by the original points. In a similar spirit, the neural bootstrapping method (NBM) [31,32] creates a finite-volume cell around each point so that i) complex geometry can be realised by preconditioning the network output, and ii) the accuracy and convergence of the surrogate model can be improved via hp-refinements of the cells. Besides, the DT-PINNs [33] use FD in a mesh-less manner via a direct interpolation among the nearest neighbours of a point, whose implementation relies on sparse matrix-vector product; this is similar to pooling in graph neural networks [34]. It is evident that, to handle an unstructured point cloud without AD, these methods have necessitated different kinds of overheads, and such overheads will increase with the wanted derivative order. For example, the can-PINNs and NBM feed many extra points to the forward pass, the number of which would normally grow by $O(P^D)$, with P being the derivative order and D the dimensionality of the problem. The above-mentioned studies have been targeted at or tested on second-order PDEs. A thorough and fair benchmark between these discretised methods and AD-based PINNs would be valuable.

2.2. Analytical differentiation

Being both error-free and the fastest, analytical differentiation should be the optimal choice if available. A good example is using the fast Fourier transform along the periodic spatial dimensions [11], which requires both solution periodicity and a structured grid. The recently proposed SC-PINNs [35] can replace AD with polynomial differentiation based on some closed-form parameterisation of coordinate mapping, justified by several forward and inverse problems with analytical solutions; whether the proposed closed-form parameterisation can be sufficiently expressive and robust for applications awaits further justification. Two mentionable works, albeit less relevant to AD, are the linearly constrained neural networks [26] and the exact imposition of Neumann and Robin boundary conditions [25]; they bypass some of the derivatives by enforcing the physical constraints through architecture design, which may help enhancing the smoothness of the loss function as compared to appending the PDE as a soft regularisation [36]. To encapsulate, analytical differentiation usually entails rather strong prerequisites (mostly from physics), but it is worth exploring if the target problem falls within the right scope.

2.3. Enhanced AD

This category is directly linked to our work. Recently, Cho et al. [37] proposed the Separable PINNs or SPINNs that, similar to this work, feature the reduction of leaf variables in AD. The SPINNs use separation of variables to reduce the leaves, conceptually close to the classic "separation of variables" in PDE textbooks. As the authors proclaim, for a D-dimensional domain with each dimension discretised by N points, the number of leaves can be reduced from N^D to ND whereby forward-mode AD can be utilised (as the number of roots will become much larger than that of the leaves). This is also similar to the Low-Rank Adaptation (LoRA) [38] for fine-tuning large language models, i.e., a matrix M_{ij} being approached by the outer product of two vectors $a_i b_j$. In short, the SPINNs reduce the number of leaves to ND by paying two prices: the assumption of separation of variables (which can weaken the expressiveness of the learned function) and a mesh grid (N^D) imposed upon the output and the PDE fields. In contrast, our ZCS algorithm can further reduce the number of leaves to D without such assumptions (as ZCS is agnostic to network architecture and point sampling). Furthermore, as shown in Section 3.2, a vanilla PINN can take full advantage of reverse-mode AD through a simple summation of roots, so the baseline of the SPINNs seems not indubitably clear to us.

For ordinary differential equations (ODEs), a few studies have featured the utilisation and generalisation of forward-mode AD [39–42]. On one hand, we will show that ZCS can maximise the performance of forward-mode AD by diminishing the number of leaf variables to a very few (D). On the other hand, we push one step ahead to reduce the number of root variables to one so that reverse-mode AD becomes directly utilisable. The motivation is clear and simple: the vast majority of deep learning tasks rely only upon reverse-mode AD for backpropagation, so its development and optimisation (both at a software and hardware level) have been prioritised over forward-mode AD. More details are given in Section 3.1.

3. Method

3.1. Understanding AD

AD is a big topic in deep learning. Here we only summarise some of the key concepts that are essential for understanding this paper. We refer the readers to [13–16] for in-depth reading. We note that, though AD has been extensively used for deep learning nowadays, it is nowhere near perfect but with active research going on, as addressed in the cited studies.

In a nutshell, AD is semi-symbolic calculation: a *computational graph* is built based on the chain rule whereby the gradients of *the root nodes* w.r.t. *the leaf nodes* can be exactly evaluated. There are two strategies in AD: forward and reverse modes, depending on in which direction the computational graph is constructed. The forward and reverse modes are most efficient (in-graph vectorised) respectively for the situations of many-roots-one-leaf and one-root-many-leaves (i.e., respectively for computing the Jacobian-vector product and vector-Jacobian product). Clearly, most deep learning tasks only use reverse-mode AD because they involve one scalar-valued loss function (the only root) and millions of network weights (many leaves). Consequently, the development of forward-mode AD lags behind. An inherent difficulty of forward-mode AD is nesting [40,41,43], as required for the second- and higher-order derivatives, whose computational cost increases exponentially with the order (nesting depth). At the time of writing, nested forward-mode AD has been supported by JAX [44] but not by PyTorch [45] and TensorFlow [46].

For the above reasons, we target our algorithm at reverse-mode AD but keep its potential for exploiting forward-mode AD in the future. For brevity, we use the notations $\partial_1 1$, $\partial_1 \infty$, $\partial_\infty 1$, and $\partial_\infty \infty$ to represent a derivative being respectively one-root-one-leaf, many-roots-one-leaf, one-root-many-leaves and many-roots-many-leaves. In summary, *the state-of-the-art reverse-mode AD requires the target derivative to be $\partial_1 1$ or $\partial_\infty 1$ to unleash its power*. A large number of roots will then necessitate an explicit for-loop or data vectorisation, both having intensive impacts on memory and time efficiency, as detailed in the context of PINOs in Section 3.2.

3.2. AD for PINNs and PINOs

For better readability, we start with the simpler case of PINNs. The forward pass of PINNs can be formulated as,

$$u_j = f_\theta\left(x_j\right), \quad j = 1, 2, \cdots, N, \tag{1}$$

where x_j's are the coordinates of the collocation points sampled from the domain (with N points in total), u_j the output field at x_j, and θ the weights of the neural network. For simplicity, the above formulation is one-dimensional, which does not affect the generality of our theory for the high-dimensional cases (where x_j becomes a vector). To obtain the PDE-induced loss function, we need to compute the derivatives $\frac{\partial u_j}{\partial x_j}$, $\frac{\partial^2 u_j}{\partial x_j^2}$, $\cdots$.[2] Since f_θ is a pointwise mapping, reverse-mode AD can be efficiently performed by using the sum of u_j as the scalar-valued root (namely, by filling the "vector" in the vector-Jacobian product with ones):

$$\frac{\partial u_j}{\partial x_j} = \frac{\partial \sum_j u_j}{\partial x_j}. \tag{2}$$

The higher-order derivatives are computed in the same manner, taking the lower-order results as the input. Equation (2) indicates that PINNs have made the most of reverse-mode AD (which makes us uncertain about the baseline of the SPINNs [37]).

[2] Einstein summation convention is *not* adopted throughout this paper.

The formulation becomes more complex for PINOs, whose forward pass can be formulated as

$$u_{ij} = f_\theta\left(p_i, x_j\right), \quad i = 1, 2, \cdots, M;\ j = 1, 2, \cdots, N, \tag{3}$$

where p_i's represent the physical parameters of interest, such as material properties, external loads, initial and boundary conditions, and observations of sensors. Here, we have M sets of physical parameters in total. The generalisability over this parameter dimension distinguishes PINOs from PINNs. Clearly, a PINO degenerates to a PINN when $M = 1$, which learns a single function $x \mapsto u$. Therefore, the dimension of physical parameters is also known as the dimension of functions.

Our goal is to compute $\frac{\partial u_{ij}}{\partial x_j}$, which is $\partial_\infty\infty$ and non-pointwise. Without an all-in-one algorithm available so far, two workaround approaches have been used. The first one is through an explicit for-loop over the i-dimension (e.g., the "aligned" operator learning in DeepXDE):

$$\frac{\partial u_{ij}}{\partial x_j} = \frac{\partial \sum_j u_{ij}}{\partial x_j}, \quad \text{for } i = 1, 2, \cdots, M. \tag{4}$$

With i viewed as a constant within the loop, $\sum_j u_{ij}$ becomes scalar-valued to allow for reverse-mode AD; in other words, the PINO is tackled as M separate PINNs one after another. The second approach is data vectorisation (e.g., the "unaligned" operator learning in DeepXDE), which upscales eq. (3) into the following pointwise form:

$$u_{ij} = f_\theta\left(\hat{p}_{ij}, \hat{x}_{ij}\right), \quad \text{where } \hat{p}_{ij} = p_i,\ \hat{x}_{ij} = x_j, \tag{5}$$

so that reverse-mode AD can be enabled using $\sum_{ij} u_{ij}$ as the scalar-valued root.

Both these two approaches, however, will significantly impair the performance of training. The former is slowed down by the function loop whereas the latter suffers unscalable memory due to massive data duplication ($2MN$ times). Furthermore, computing $\frac{\partial u_{ij}}{\partial x_j}$ is not our final goal; what we eventually want is the gradient of the PDE-induced loss function w.r.t. the network weights θ, or backpropagation. Let $\mathcal{G}_1$ denote the computation graph of backpropagation for the corresponding PINN, i.e., eq. (3) with $M = 1$. The function-loop approach will make M duplicates of $\mathcal{G}_1$ that are connected at the root end, and the data-vectorisation approach will enlarge $\mathcal{G}_1$ by M times at the leaf end (because the coordinates are duplicated by M times in eq. (5)). Therefore, both these approaches will make backpropagation highly memory demanding and slow. In summary, the full potential of reverse-mode AD is yet to be exploited for PINOs in the form of eq. (3).

3.3. Zero coordinate shift

Our algorithm starts from introducing a dummy variable z into eq. (3), which defines the following associated field v_{ij}:

$$v_{ij} := f_\theta\left(p_i, x_j + z\right). \tag{6}$$

Clearly, $v_{ij} = u_{ij}$ when $z = 0$. The following equivalence can then be established, bearing the central idea of this paper:

$$\frac{\partial u_{ij}}{\partial x_j} = \left.\frac{\partial v_{ij}}{\partial z}\right|_{z=0}, \tag{7}$$

as proved by

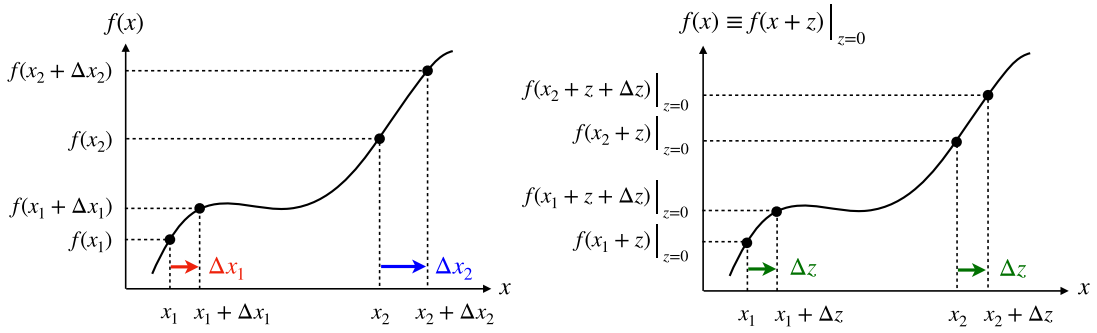
$$\begin{aligned}\frac{\partial u_{ij}}{\partial x_j} &= \frac{\partial f_\theta\left(p_i, x_j\right)}{\partial x_j} = \left.\frac{\partial f_\theta\left(p_i, x\right)}{\partial x}\right|_{x=x_j} = \left.\frac{\partial f_\theta\left(p_i, x+z\right)}{\partial x}\right|_{\substack{x=x_j\\z=0}} \\ &\overset{*}{=} \left.\frac{\partial f_\theta\left(p_i, x+z\right)}{\partial z}\right|_{\substack{x=x_j\\z=0}} \overset{\dagger}{=} \left.\frac{\partial f_\theta\left(p_i, x_j+z\right)}{\partial z}\right|_{z=0} = \left.\frac{\partial v_{ij}}{\partial z}\right|_{z=0}.\end{aligned} \tag{8}$$

In the above proof, the step marked by $*$ is based on that x and z are symmetric in f_θ, and the step marked by † exchanges the order of derivative ($\partial/\partial z$) and evaluation ($x = x_j$), thanks to that f_θ is continuously differentiable w.r.t. x everywhere. The other steps involve no subtlety. Equation (7) is significant because *it simplifies the wanted derivative from $\partial_\infty\infty$ to $\partial_1\infty$, with z being the only leaf*. Geometrically, z can be interpreted as a zero-valued shift (translation) applied to all the coordinates, and hence the naming of the algorithm as Zero Coordinate Shift (ZCS). Fig. 1 is provided to facilitate the understanding of eq. (7) from the perspective of limits.

Being $\partial_1\infty$, eq. (7) has been prepared for forward-mode AD. However, given the current immaturity of forward-mode AD, we prefer to take one step further to manipulate it into $\partial_\infty 1$. For this purpose, we introduce another arbitrarily-valued dummy variable a_{ij}, and define

$$\omega := \sum_{ij} a_{ij} v_{ij}, \quad \text{so that} \quad v_{ij} = \frac{\partial \omega}{\partial a_{ij}}. \tag{9}$$

Note that ω is a scalar. Inserting eq. (9) into (7), we obtain



(a) Taking both x_1 and x_2 as leaf variables (b) Taking only z as a leaf variable

Fig. 1. Understanding ZCS via limits. In (a), $\frac{\partial f(x_1)}{\partial x_1}$ and $\frac{\partial f(x_2)}{\partial x_2}$ are approached individually by taking Δx_1 and Δx_2 as independent infinitesimal increments, corresponding to taking x_1 and x_2 as independent leaf variables for AD. In (b), Δz is the only infinitesimal increment, associated with a zero-valued dummy variable z, and $\frac{\partial f(x_1)}{\partial x_1}$ and $\frac{\partial f(x_2)}{\partial x_2}$ are respectively equal to $\frac{\partial f(x_1+z)}{\partial z}\Big|_{z=0}$ and $\frac{\partial f(x_2+z)}{\partial z}\Big|_{z=0}$, meaning that z can be the only leaf variable for AD.

$$\underbrace{\frac{\partial u_{ij}}{\partial x_j}}_{\partial_\infty \infty} = \underbrace{\frac{\partial}{\partial a_{ij}}}_{\partial_\infty 1} \underbrace{\frac{\partial \omega}{\partial z}\Big|_{z=0}}_{\partial_1 1}. \tag{10}$$

As annotated in eq. (10), *the wanted $\partial_\infty \infty$ derivative is eventually factorised into a $\partial_1 1$ and a $\partial_\infty 1$ derivative, both of which can be efficiently computed using reverse-mode AD, loop- and duplication-free.*

Concerning higher-order derivatives and non-linear terms in a PDE, the following useful properties can be shown:

$$\underbrace{\frac{\partial^n u_{ij}}{\partial x_j^n}}_{n\times\partial_\infty \infty} = \underbrace{\frac{\partial}{\partial a_{ij}}}_{\partial_\infty 1} \underbrace{\frac{\partial^n \omega}{\partial z^n}\Big|_{z=0}}_{n\times\partial_1 1}, \tag{11}$$

and

$$\underbrace{\frac{\partial^m u_{ij}}{\partial x_j^m}\frac{\partial^n u_{ij}}{\partial x_j^n}}_{(m+n)\times\partial_\infty \infty} = \frac{1}{2} \underbrace{\frac{\partial^2}{\partial a_{ij}^2}}_{2\times\partial_\infty 1} \underbrace{\left(\frac{\partial^m \omega}{\partial z^m}\frac{\partial^n \omega}{\partial z^n}\right)\Big|_{z=0}}_{(m+n)\times\partial_1 1}. \tag{12}$$

The proofs are given in Appendix A. Equations (11) and (12) suggest that the number of $\partial_\infty 1$ ADs w.r.t. a_{ij} can be reduced by collecting terms with the same multiplicative power (not differential order) in a PDE. In particular, for a linear PDE, only one $\partial_\infty 1$ AD is required. For example, consider a PDE, $g = u_x + u_y + u_{xy} + u_x u_y + u_{xx} u_{yy} = 0$, and let z_x and z_y be the ZCS scalars respectively for the x and y dimensions; the following process computes each term separately using eq. (11):

$$\begin{aligned} g_{ij} = \Bigg[& \frac{\partial}{\partial a_{ij}}\frac{\partial \omega}{\partial z_x} + \frac{\partial}{\partial a_{ij}}\frac{\partial \omega}{\partial z_y} + \frac{\partial}{\partial a_{ij}}\frac{\partial^2 \omega}{\partial z_x \partial z_y} \\ & + \left(\frac{\partial}{\partial a_{ij}}\frac{\partial \omega}{\partial z_x}\right)\left(\frac{\partial}{\partial a_{ij}}\frac{\partial \omega}{\partial z_y}\right) + \left(\frac{\partial}{\partial a_{ij}}\frac{\partial^2 \omega}{\partial z_x^2}\right)\left(\frac{\partial}{\partial a_{ij}}\frac{\partial^2 \omega}{\partial z_y^2}\right)\Bigg]\Bigg|_{\substack{z_x=0 \\ z_y=0}}; \end{aligned} \tag{13}$$

alternatively, one can combine the linear and the non-linear terms to reduce the number of $\partial\infty 1$ ADs, using eq. (12) for the non-linear terms:

$$g_{ij} = \Bigg[\underbrace{\frac{\partial}{\partial a_{ij}}}_{\partial_\infty 1} \underbrace{\left(\frac{\partial \omega}{\partial z_x} + \frac{\partial \omega}{\partial z_y} + \frac{\partial^2 \omega}{\partial z_x \partial z_y}\right)}_{\partial_1 1\text{'s}} + \frac{1}{2} \underbrace{\frac{\partial^2}{\partial a_{ij}^2}}_{2\times\partial_\infty 1} \underbrace{\left(\frac{\partial \omega}{\partial z_x}\frac{\partial \omega}{\partial z_y} + \frac{\partial^2 \omega}{\partial z_x^2}\frac{\partial^2 \omega}{\partial z_y^2}\right)}_{\partial_1 1\text{'s}} \Bigg]\Bigg|_{\substack{z_x=0 \\ z_t=0}}. \tag{14}$$

So far, we have addressed how to compute the PDE field by our two-step AD algorithm. Accelerating the computation of the PDE, however, is not the biggest reason behind a huge performance improvement. As stated in Section 3.2, the two workaround approaches, function loop and data vectorisation, will both enlarge the computational graph of backpropagation by M times, but ZCS will not. As indicated by eq. (7), the scalar-valued leaf variable z is shared not only by the N points but also by the M functions. Therefore, the graph for $M > 1$ can remain as large as that for $M = 1$ (or the corresponding PINN). This will be demonstrated by our experiments in Section 4.2. *Diminishing the physical size of the computational graph is our largest source of memory and time savings.*

3.4. Implementation

Our ZCS algorithm is easy to implement using high-level APIs from current deep learning libraries. Here we provide a complete example in Algorithm 1 to facilitate understanding, aimed for computing the Laplacian of the network output. The loop-based algorithm is also provided for reference. It is visible that ZCS involves neither a for-loop nor data duplication and feeds only two scalar-valued leaf variables to the neural network.

Algorithm 1 Computing Laplacian $u_{xx} + u_{yy}$ for PINOs by reverse-mode AD.

Input:
- $\mathbf{p} \in \mathbb{R}^M \times \mathbb{R}^Q$ ▷ Physical parameters
- $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ ▷ Coordinates
- $f_\theta : \left(\mathbb{R}^M \times \mathbb{R}^Q, \mathbb{R}^N \times \mathbb{R}^2\right) \to \mathbb{R}^M \times \mathbb{R}^N$ ▷ Neural network

Output: $\mathbf{g} \in \mathbb{R}^M \times \mathbb{R}^N$ ▷ Laplacian of network output

Baseline: loop-based

1: $(\mathbf{x}, \mathbf{y})$.requires_grad ← True ▷ Make $\mathbf{x}$ and $\mathbf{y}$ leaf variables for AD
2: $\mathbf{u} \leftarrow f_\theta(\mathbf{p}, \{\mathbf{x}, \mathbf{y}\}^{\mathrm{T}})$ ▷ Feed forward
3: **for** $i \leftarrow 1$ to M **do** ▷ Parameter loop (slow)
4: $(\mathbf{q}, \mathbf{r}) \leftarrow \dfrac{\partial \sum_j u_{ij}}{\partial (\mathbf{x}, \mathbf{y})}$ ▷ $\partial_\infty 1$ AD: $\mathbf{q} \in \mathbb{R}^N$ for u_x; $\mathbf{r} \in \mathbb{R}^N$ for u_y
5: $\mathbf{s} \leftarrow \dfrac{\partial \sum_j q_j}{\partial \mathbf{x}}, \mathbf{t} \leftarrow \dfrac{\partial \sum_j r_j}{\partial \mathbf{y}}$ ▷ $\partial_\infty 1$ AD: $\mathbf{s} \in \mathbb{R}^N$ for u_{xx}; $\mathbf{t} \in \mathbb{R}^N$ for u_{yy}
6: $\mathbf{g}_i = \mathbf{s} + \mathbf{t}$ ▷ $u_{xx} + u_{yy}$ for parameter $\mathbf{p}_i$
7: **end for**

ZCS (ours)

1: $z_x \leftarrow 0$, $z_y \leftarrow 0$ ▷ Create ZCS scalars z_x and z_y
2: $a_{ij} \leftarrow 1$, for $i = 1, 2, \cdots, M$; $j = 1, 2, \cdots, N$ ▷ Create dummy variable a_{ij}
3: $(z_x, z_y, \mathbf{a})$.requires_grad ← True ▷ Make them leaf variables for AD
4: $x_j \leftarrow x_j + z_x$, $y_j \leftarrow y_j + z_y$, for $j = 1, 2, \cdots, N$ ▷ Apply ZCS to coordinates
5: $\mathbf{u} \leftarrow f_\theta(\mathbf{p}, \{\mathbf{x}, \mathbf{y}\}^{\mathrm{T}})$ ▷ Feed forward (only z_x and z_y being leaves)
6: $\omega \leftarrow \sum_{ij} a_{ij} u_{ij}$ ▷ The scalar-valued root
7: $q \leftarrow \dfrac{\partial \omega}{\partial z_x}, r \leftarrow \dfrac{\partial \omega}{\partial z_y}$ ▷ $\partial_1 1$ AD: q for u_x; r for u_y
8: $s \leftarrow \dfrac{\partial q}{\partial z_x}, t \leftarrow \dfrac{\partial r}{\partial z_y}$ ▷ $\partial_1 1$ AD: s for u_{xx}; t for u_{yy}
9: $g \leftarrow s + t$ ▷ Laplacian of ω
10: $\mathbf{g} = \dfrac{\partial g}{\partial \mathbf{a}}$ ▷ $\partial_\infty 1$ AD: Laplacian of $\mathbf{u}$

▷ Note that steps 7 and 8 are scalar-to-scalar derivatives

4. Experiments

We report our experiments in this section. We implement our ZCS algorithm by extending the DeepXDE package [27] with a PyTorch backend. The results are similar for TensorFlow and JAX. We choose to extend DeepXDE for two reasons: to start from a well-established, state-of-the-art baseline, and to demonstrate how easily ZCS can be integrated to an existing model or framework. However, we note that ZCS works not just for DeepONets but *any operators in the form of eq.* (3), including PINNs as a special case (when $M = 1$).

For each problem setup, the following three models using different AD strategies will be compared:

- `FuncLoop`: the "aligned" DeepONets using an explicit for-loop over the function dimension (i.e., the dimension of physical parameters) for AD, formulated by eq. (4) and implemented as the `PDEOperatorCartesianProd` class in DeepXDE;
- `DataVect`: the "unaligned" DeepONets using data vectorisation for AD, formulated by eq. (5) and implemented as the `PDEOperator` class in DeepXDE; for fair comparison, we have generalised this class for batch support along the function dimension;
- `ZCS`: DeepONets equipped with ZCS, formulated by eq. (10) and implemented as the `PDEOperatorCartesianProdZCS` class in our extended DeepXDE.

Because ZCS does not affect the resultant model, *the metrics of merit we consider are GPU memory consumption and wall time for training*. All the experiments can be found at https://github.com/stfc-sciml/ZeroCoordinateShift.

4.1. Scaling analysis

We first carry out a systematic benchmark analysis for understanding the scaling behaviours of the three compared methods. We consider the following high-order linear PDE in 2-D:

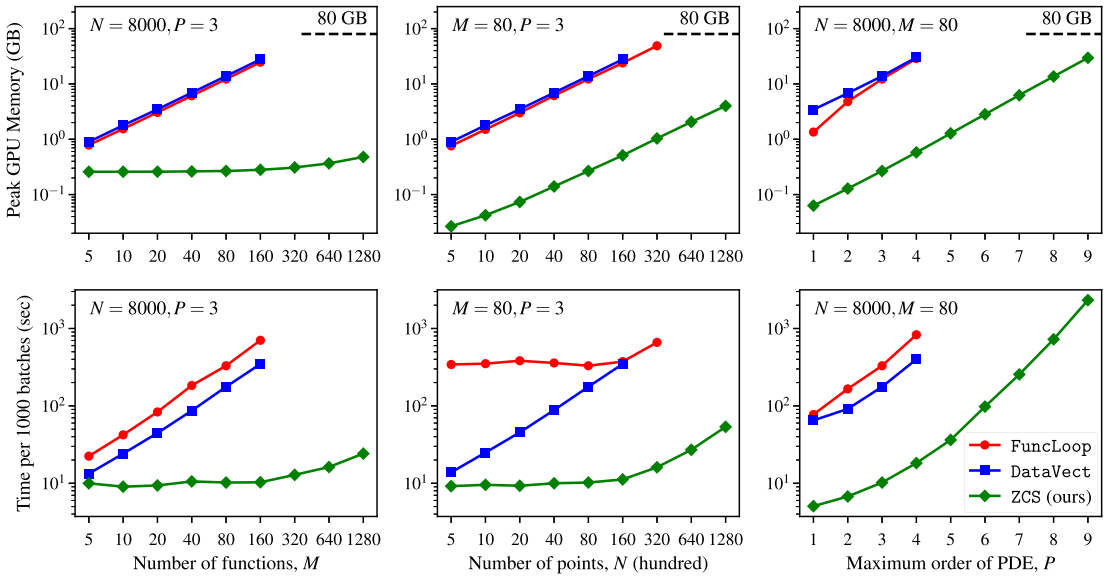


Fig. 2. Peak GPU memory and wall time for training DeepONets with different AD strategies. The PDE is given by eq. (15), with a maximum differential order of P, and the function and point numbers M and N are defined in eq. (3). In the three columns from left to right, we vary respectively M, N and P while fixing the other two. The measurements are taken on a Nvidia-A100 GPU with 80 GB memory.

$$\sum_{k=0}^{P} \left(\frac{\partial}{\partial x} + \frac{\partial}{\partial y} \right)^k u = 0. \tag{15}$$

We investigate three parameters that define the problem scale, which have the greatest impact on memory and time efficiency: i) the number of functions M in eq. (3), ii) the number of collocation points N in eq. (3), and iii) the maximum differential order P in eq. (15). The tested DeepONet has a branch net with four fully-connected layers respectively of size 50 (number of features in each physical parameter), 128, 128 and 128, and a trunk net with four fully-connected layers respectively of size 2 (number of dimensions), 128, 128 and 128.

The GPU memory and time measurements are shown in Fig. 2. On the whole, it is clearly shown that ZCS has simultaneously reduced memory and time by an order of magnitude across the tested ranges (except for the extremely small problems). This also implies that the absolute savings will rapidly increase with the scale of the problem. Next, we look into the influence of M, N and P individually.

- *Number of functions* M: The first column of Fig. 2 indicates that the memory and time for both `FuncLoop` and `DataVect` scale with M; `FuncLoop` is slightly less memory demanding while `DataVect` is about twice faster. Such scaling verifies that these two approaches will expand the computational graph by M times. In contrast, both the memory and time for `ZCS` increase extremely slowly with M, almost remaining constant when $M \leq 160$, thanks to that ZCS can maintain the graph size at that for $M = 1$; as M continues to increase ($M \geq 320$), the forward pass of the branch net becomes more and more dominant (as N is fixed), causing the memory and time to start increasing slowly. Note that M will be the batch size for applications, so it is unlikely to be very large for a desirable level of trajectory noise in stochastic gradient descent.
- *Number of points* N: As shown in the second column of Fig. 2, the memory usage of `ZCS` scales with N across the whole range (i.e., no plateau at the smaller N's), as the two ZCS scalars have been added to all the N coordinates. This is the main difference from the scaling w.r.t. M. Besides, both `FuncLoop` and `ZCS` exhibit a plateau of wall time at the smaller N's, as backpropagation is governed by the branch net within this range; `DataVect` does not show such a plateau because the physical parameters are duplicated by N times to form the input of the branch net.
- *Maximum differential order* P: The last column of Fig. 2 suggests that P has the strongest impact on both memory and time efficiency (note that the horizontal axis in this column is in linear scale). This is because the higher-order derivatives recursively expand the computational graph. Such undesirable scalability cannot be remedied by ZCS. Nevertheless, `ZCS` has managed to push P to nine on a single GPU, considering a decently large M and N. Forward-mode AD, in theory, is impossible for such a high order. In view of such scalability, the FO-PINNs [47], which recommend decomposing a high-order PDE into a system of first-order PDEs, seem a sensible suggestion.

In Section 4.2, we will break down the memory and time into different stages, from which the above scaling behaviours can be even better understood.

4.2. Operator learning

In this section, we train DeepONets to learn four PDE operators. For training, we only use the physics-based loss functions, i.e., the PDE and its initial and boundary conditions; true solutions are used only for validation. For each problem, we train five models with different weight initialisations to obtain the mean measurements unless otherwise stated.

The first PDE is the one-component reaction–diffusion equation:

$$\begin{aligned} u_t - Du_{xx} + ku^2 - f(x) = 0, \quad & x \in (0,1), t \in (0,1); \\ u(x,0) = 0, \quad & x \in (0,1); \\ u(0,t) = u(1,t) = 0, \quad & t \in (0,1), \end{aligned} \tag{16}$$

where the constants are set as $D = k = 0.01$. We learn an operator mapping from the time-independent source term $f(x)$ to the solution $u(x,t)$. The training data contain $1,000$ $f(x)$'s sampled from a Gaussian process, learned with $M = 50$ (batch size) and $N = 1,000$. This small-scale problem is presented by DeepXDE as a demonstration.

In the second problem, we consider the following Burgers' equation:

$$\begin{aligned} u_t + uu_x - \nu u_{xx} = 0, \quad & x \in (0,1), t \in (0,1); \\ u(x,0) = u_0(x), \quad & x \in (0,1); \\ u(0,t) = u(1,t), \quad & t \in (0,1), \end{aligned} \tag{17}$$

where the viscosity ν is set at 0.01. The learned operator maps from the initial condition $u_0(x)$ to the solution $u(x,t)$. The data come from the physics-informed FNOs [11], containing 1000 $u_0(x)$'s sampled from a Gaussian process. We choose $M = 50$ and $N = 12800$ for this problem. Therefore, the scale of this problem is significantly larger than that of the previous one in terms of N.

Our third problem considers bending of a square Kirchhoff-Love plate, governed by the following forth-order Germain-Lagrange equation:

$$\begin{aligned} \frac{\partial^4 u}{\partial x^4} + \frac{2\partial^4 u}{\partial x^2 \partial y^2} + \frac{\partial^4 u}{\partial y^4} = \frac{q}{D}, \quad & x \in (0,1), y \in (0,1); \\ u(x,0) = u(x,1) = 0, \quad & x \in (0,1); \\ u(0,y) = u(1,y) = 0, \quad & y \in (0,1), \end{aligned} \tag{18}$$

where the flexural rigidity D is set at 0.01. We learn an operator mapping from the source term $q(x,y)$ to the solution $u(x,y)$. We assume $q(x,y)$ to have the following bi-trigonometric form:

$$q(x,y) = \sum_{r=1}^{R} \sum_{s=1}^{S} c_{rs} \sin(r\pi x) \sin(s\pi y), \tag{19}$$

whereby an analytical solution exists for validation. We sample 1080 sets of the coefficients c_{rs} from $\mathcal{N}(0,1)$ assuming $R = S = 10$, so the number of input features for the branch net is 10^2. We use $M = 36$ and $N = 10000$ for training. The problem scale grows even larger in terms of P.

Our last problem describes the 2-D Stokes flow in a square box with a moving lid. Different from the previous ones, this problem features vector-valued network output: the horizontal and vertical velocities (u and v) and the pressure (p):

$$\begin{aligned} \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial p}{\partial x} = 0, \quad & x \in (0,1), y \in (0,1); \\ \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial p}{\partial y} = 0, \quad & x \in (0,1), y \in (0,1); \\ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \quad & x \in (0,1), y \in (0,1); \\ u(x,1) = u_1(x), v(x,1) = 0, \quad & x \in (0,1); \\ u(x,0) = v(x,0) = p(x,0) = 0, \quad & x \in (0,1); \\ u(0,y) = v(0,y) = 0, \quad & y \in (0,1); \\ u(1,y) = v(1,y) = 0, \quad & y \in (0,1), \end{aligned} \tag{20}$$

where the dynamic viscosity μ is set at 0.01. We learn an operator mapping from the lid velocity $u_1(x)$ to the solution $\{u, v, p\}(x,y)$, with 1000 $u_1(x)$'s sampled from a Gaussian process. Note that the zero-pressure boundary condition on the bottom ($p(x,0) = 0$) is added only to fix the constant part of p. We use $M = 50$ and $N = 5000$ for training.

The measurements and accuracy for these problems are summarised in Table 1, along with some training details in the caption. The general conclusions are similar to those from Section 4.1, that is, `ZCS` turns out to be one-order-of-magnitude more efficient in both memory and time, except for the reaction-diffusion problem (comparing the wall time of `DataVect` and `ZCS`) whose scale is

Table 1
GPU memory consumption and wall time for training DeepONets to learn PDE operators. Some of the columns are clarified as follows: "Graph" for the memory occupied by the computational graph of backpropagation, "Inputs" for the time used to prepare input tensors, and "Loss (PDE)" for the time used to compute the physics-based loss functions (mostly the PDE field). Training is purely physics-based (without a data loss). The measurements are taken on a Nvidia-A100 GPU with 80 GB memory. For reaction-diffusion, the measurements and errors are obtained after training for 10000 batches. For Burgers, the measurements are based on short runs with 1000 batches, but the errors are obtained from long runs with 10^5 batches (ignoring different model initialisations for `FuncLoop` and `DataVect` to save resources). For Kirchhoff-Love and Stokes, `DataVect` cannot be trained for insufficient memory; for `FuncLoop` and `ZCS`, we obtain the measurements from short runs with 200 batches and the errors from long runs with 5×10^4 batches (ignoring different model initialisations for `FuncLoop` to save resources).

Problem	Scale	Method	GPU memory (GB)		Time per 1000 batches (sec)					Relative error
			Graph	Peak	Inputs	Forward	Loss (PDE)	Backprop	Total	

Reaction-diffusion	$M = 50$
	$N = 1000$
	$P = 2$
		`FuncLoop`
		`DataVect`
		`ZCS` (ours)
			0.96
			0.97
			0.02
				0.98
				1.46
				0.05
					0
					5
					1
						1
						3
						1
							81
							4
							4
								99
								12
								4
									181
									24
									10
										8.3±2.0%
										9.5±2.5%
										8.2±2.0%

Burgers	$M = 50$
	$N = 12800$
	$P = 2$
		`FuncLoop`
		`DataVect`
		`ZCS` (ours)
			7.84
			7.73
			0.20
				7.91
				11.40
				0.36
					1
					95
					1
						2
						20
						2
							140
							24
							7
								173
								82
								5
									316
									221
									15
										7.5%
										7.2%
										7.1±0.5%

Kirchhoff-Love	$M = 36$
	$N = 10000$
	$P = 4$
		`FuncLoop`
		`DataVect`
		`ZCS` (ours)
			77.57
			–
			2.36
				77.57
				–
				3.30
					1
					–
					1
						6
						–
						6
							1765
							–
							77
								2309
								–
								60
									4081
									–
									144
										27.3%
										–
										26.9±0.5%

Stokes	$M = 50$
	$N = 5000$
	$P = 2$
		`FuncLoop`
		`DataVect`
		`ZCS` (ours)
			74.44
			–
			1.99
				78.58
				–
				3.30
					1
					–
					1
						4
						–
						3
							2036
							–
							82
								2212
								–
								60
									4253
									–
									147
										10.3%
										–
										10.4±0.6%

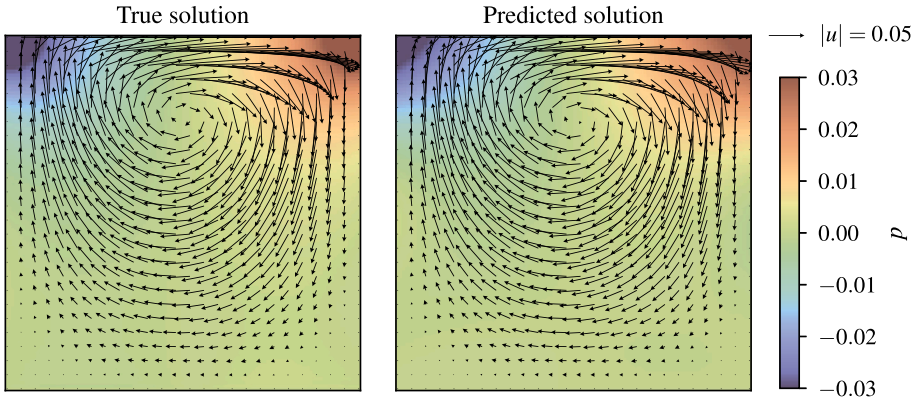


Fig. 3. True and predicted solutions of the Stokes flow in a square box with a moving lid. The PDEs and boundary conditions are given by eq. (20), with the source term $u_1(x) = x(1 - x)$. The details of model training are given in Table 1. The true solution is computed using FreeFEM++ [48].

impractically small. We manage to factor out the memory occupied by the computational graph, which reveals that memory saving by `ZCS` stems from a diminished graph size. The slimmed-down graph then leads to a huge reduction of wall time for both PDE calculation and backpropagation. We emphasise that, for all the problems, the graph sizes of `ZCS` are roughly M times smaller than those of `FuncLoop` and `DataVect`. As a side note, for `DataVect`, one can see a clear gap between the graph and peak memories as well as greater portions of wall time for preparing input tensors (excluding our code for batch sampling) and forward pass, both caused by the massively enlarged tensors (both input and intermediate) due to data vectorisation.

Accuracy-wise, we do not report extremely low errors for two reasons: i) training is purely physics-based and, ii) we do not run very long jobs due to resource limitation. However, the close errors for each problem should be sufficiently convincing that ZCS does not affect the training results (except some randomness from floating-point errors). A pair of true and predicted solutions to the Stokes problem is shown in Fig. 3. Our concluding remark from these experiments is that, *before ZCS, data vectorisation and function loop are better suited respectively to smaller- and larger-scale problems; ZCS emerges as a replacement for both, with an outstanding superiority across all problem scales.*

5. Limitations

The only limitation we have identified is that ZCS cannot improve the training efficiency of network architectures built upon a structured grid, such as convolutional neural networks (CNNs) [28,29] and FNOs [11,49]. First, we emphasise that AD (with or without ZCS) is available for a grid-based model if and only if it is translation invariant. Let us write the forward pass as (assuming a 2-D domain and omitting the physical parameters)

$$\mathbf{u} = f_\theta(\mathbf{x}, \mathbf{y}), \tag{21}$$

where $\mathbf{u} = \{u_{I,J}\}$ is the output image, and $\mathbf{x} = \{x_{I,J}\}$ and $\mathbf{y} = \{y_{I,J}\}$ are the position encoding images, with I and J respectively denoting the pixel indices along x and y. Translation invariance then requires

$$\begin{aligned} u_{I+1,J}(\mathbf{x}, \mathbf{y}) &= u_{I,J}(\mathbf{x} + \Delta x, \mathbf{y}), \\ u_{I,J+1}(\mathbf{x}, \mathbf{y}) &= u_{I,J}(\mathbf{x}, \mathbf{y} + \Delta y), \end{aligned} \tag{22}$$

with Δx and Δy being the grid intervals. Clearly, without translation invariance, the position embeddings $\mathbf{x}$ and $\mathbf{y}$ cannot be interpreted as the coordinates bearing the output field. CNNs and FNOs both satisfy this condition except at the near-edge pixels due to the issue of padding. Excluding such pixels, one can still apply AD to calculate the PDE filed and optionally employ ZCS to boost the performance of AD (i.e., feeding two scalars z_x and z_y instead of the whole $\mathbf{x}$ and $\mathbf{y}$ as leaf variables). In fact, we have implemented and verified ZCS for CNNs and FNOs.

Nevertheless, even boosted by ZCS, AD remains more memory demanding and slower than finite difference and analytical differentiation (the fast Fourier transform [11] for FNOs), as both of them lightly enlarge the computational graph as compared to their non-physics-informed counterparts; after all, this is one of the major motivations for using a structured grid. Precisely speaking, what has been discussed in this section is not a limitation of ZCS but a downside of any pointwise operators (i.e., eq. (3)) not taking advantage of structured data where available. In turn, not restricting point sampling makes pointwise operators more flexible.

6. Conclusions

We have presented a novel algorithm to conduct automatic differentiation (AD) for physics-informed operator learning. We show that a physics-informed neural operator in the form of eq. (3), such as a DeepONet, cannot directly utilise AD to compute the derivatives (first or higher orders) of the network output w.r.t. the coordinates of collocation points, owing to the presence of the dimension of physical parameters (i.e., the dimension of functions). The current workaround approaches have significantly undermined the memory and time efficiency of training. Based on simple calculus, we reformulate the wanted derivatives as ones w.r.t. a zero-valued dummy scalar, or eq. (7), simplifying them from "many-roots-many-leaves" to "many-roots-one-leaf". Further, by introducing another arbitrarily-valued dummy tensor, we eventually simplify the derivatives to "one-root-many-leaves", or eq. (10), which can then exploit the most powerful reverse-mode AD. Based on the geometric interpretation of the zero-valued scalar, we call our algorithm Zero Coordinate Shift (ZCS). ZCS is a low-level optimisation technique, independent of data, physics (PDE), point sampling and network architecture, and does not affect training results.

We implement ZCS by a light extension of the DeepXDE package. Based on this implementation, we carry out several experiments, comparing our algorithm to the two workaround approaches: loop over functions and data vectorisation. The results show that ZCS has persistently reduced GPU memory consumption and wall time for training DeepONets by an order of magnitude, with this reduction factor scaling with the number of functions. As verified by the memory and time measurements, such outstanding improvements have originated from the avoidance of duplicating the computation graph of backpropagation along the dimension of functions. Our code and experiments can be found at https://github.com/stfc-sciml/ZeroCoordinateShift.

CRediT authorship contribution statement

Kuangdai Leng: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Mallikarjun Shankar:** Resources. **Jeyan Thiyagalingam:** Funding acquisition, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Kuangdai Leng reports financial support was provided by Engineering and Physical Sciences Research Council (EPSRC). Mallikarjun Shankar reports financial support was provided by Department of Energy (DOE). If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

We thank the two reviewers of this paper for their constructive suggestions. We thank Lu Lu for supporting us on building ZCS into DeepXDE. This work is supported by the EPSRC grant, Blueprinting for AI for Science at Exascale (BASE-II, EP/X019918/1), and by the International Science Partnerships Fund (ISPF), most specifically through the AI for Realistic Science (AIRS) programme in

collaboration with the Department of Energy (DOE) supported by the Oak Ridge Leadership Computing Facility (OLCF) under DOE Contract No. DE-AC05-00OR22725.

Appendix A. Additional proofs

Equation (11) is proved as follows:

$$\begin{aligned}\frac{\partial^n u_{ij}}{\partial x_j^n} &= \frac{\partial^n f_\theta\left(p_i, x_j\right)}{\partial x_j^n} = \left.\frac{\partial^n f_\theta\left(p_i, x\right)}{\partial x^n}\right|_{x=x_j} = \left.\frac{\partial^n f_\theta\left(p_i, x+z\right)}{\partial x^n}\right|_{\substack{x=x_j\\z=0}} \\ &= \left.\frac{\partial^n f_\theta\left(p_i, x+z\right)}{\partial z^n}\right|_{\substack{x=x_j\\z=0}} \overset{\dagger}{=} \left.\frac{\partial^n f_\theta\left(p_i, x_j+z\right)}{\partial z^n}\right|_{z=0} = \left.\frac{\partial^n v_{ij}}{\partial z^n}\right|_{z=0} = \left.\frac{\partial}{\partial a_{ij}}\frac{\partial^n \omega}{\partial z^n}\right|_{z=0}.\end{aligned} \tag{A.1}$$

Here the step marked by † further requires that f_θ should have C^n continuity w.r.t. x. This condition is met by the commonly-used activation functions in PINNs and PINOs, such as tanh, gelu and softplus, which are of C^∞.

The proof of eq. (12) reads (omitting $|_{z=0}$ everywhere for clarity):

$$\begin{aligned}2\frac{\partial^m u_{ij}}{\partial x_j^m}\frac{\partial^n u_{ij}}{\partial x_j^n} &\overset{*}{=} 2\frac{\partial}{\partial a_{ij}}\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial}{\partial a_{ij}}\frac{\partial^n \omega}{\partial z^n} \\ &\overset{\dagger}{=} \frac{\partial}{\partial a_{ij}}\left(\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial}{\partial a_{ij}}\frac{\partial^n \omega}{\partial z^n}\right) - \cancel{\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial^2}{\partial a_{ij}^2}\frac{\partial^n \omega}{\partial z^n}} + \\ &\quad \frac{\partial}{\partial a_{ij}}\left(\frac{\partial}{\partial a_{ij}}\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial^n \omega}{\partial z^n}\right) - \cancel{\frac{\partial^2}{\partial a_{ij}^2}\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial^n \omega}{\partial z^n}} \\ &= \frac{\partial}{\partial a_{ij}}\left(\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial}{\partial a_{ij}}\frac{\partial^n \omega}{\partial z^n} + \frac{\partial}{\partial a_{ij}}\frac{\partial^m \omega}{\partial z^m} \times \frac{\partial^n \omega}{\partial z^n}\right) \\ &= \frac{\partial^2}{\partial a_{ij}^2}\left(\frac{\partial^m \omega}{\partial z^m}\frac{\partial^n \omega}{\partial z^n}\right).\end{aligned} \tag{A.2}$$

The step marked by $*$ uses eq. (11), and the one marked by † first split the l.h.s. into two identical copies and then apply the product rule to each one. The crossed-out terms are based on that $\frac{\partial^2 \omega}{\partial a_{ij}^2} = 0$.

References

[1] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (6) (2021) 422–440.
[2] S. Cuomo, V.S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, F. Piccialli, Scientific machine learning through physics-informed neural networks: where we are and what's next, J. Sci. Comput. 92 (2022) 88.
[3] L. Yang, D. Zhang, G.E. Karniadakis, Physics-informed generative adversarial networks for stochastic differential equations, SIAM J. Sci. Comput. 42 (1) (2020) A292–A317.
[4] L. Yuan, Y.-Q. Ni, X.-Y. Deng, S. Hao, A-PINN: auxiliary physics informed neural networks for forward and inverse problems of nonlinear integro-differential equations, J. Comput. Phys. 462 (2022) 111260.
[5] G. Pang, L. Lu, G.E. Karniadakis, fPINNs: fractional physics-informed neural networks, SIAM J. Sci. Comput. 41 (4) (2019) A2603–A2626.
[6] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Netw. 9 (5) (1998) 987–1000.
[7] M. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations, Commun. Numer. Methods Eng. 10 (3) (1994) 195–201.
[8] J. Berg, K. Nyström, A unified deep artificial neural network approach to partial differential equations in complex geometries, Neurocomputing 317 (2018) 28–41.
[9] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.
[10] L. Lu, P. Jin, G. Pang, Z. Zhang, G.E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, Nat. Mach. Intell. 3 (3) (2021) 218–229.
[11] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, K. Azizzadenesheli, A. Anandkumar, Physics-informed neural operator for learning partial differential equations, arXiv preprint, arXiv:2111.03794, 2021.
[12] J. Yu, L. Lu, X. Meng, G.E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse pde problems, Comput. Methods Appl. Mech. Eng. 393 (2022) 114823.
[13] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: a survey, J. Mach. Learn. Res. 18 (2018) 1–43.
[14] C.C. Margossian, A review of automatic differentiation and its efficient implementation, Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 9 (4) (2019) e1305.
[15] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, L. Zhang, Fuzzing automatic differentiation in deep-learning libraries, arXiv preprint, arXiv:2302.04351, 2023.
[16] D. Johnson, T. Maxfield, Y. Jin, R. Fedkiw, Software-based automatic differentiation is flawed, arXiv preprint, arXiv:2305.03863, 2023.
[17] Z. Liu, W. Cai, Z.-Q. J. Xu, Multi-scale deep neural network (MscaleDNN) for solving Poisson-Boltzmann equation in complex domains, Commun. Comput. Phys. 28 (5) (2020) 1970–2001.
[18] W. Cai, X. Li, L. Liu, A phase shift deep neural network for high frequency approximation and wave problems, SIAM J. Sci. Comput. 42 (5) (2020) A3285–A3312.
[19] B. Moseley, A. Markham, T. Nissen-Meyer, Finite basis physics-informed neural networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, arXiv preprint, arXiv:2107.07871, 2021.

[20] E. Kharazmi, Z. Zhang, G.E. Karniadakis, hp-vpinns: variational physics-informed neural networks with domain decomposition, Comput. Methods Appl. Mech. Eng. 374 (2021) 113547.
[21] A.D. Jagtap, K. Kawaguchi, G. Em Karniadakis, Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks, Proc. R. Soc. A 476 (2239) (2020) 20200334.
[22] Z. Mao, A.D. Jagtap, G.E. Karniadakis, Physics-informed neural networks for high-speed flows, Comput. Methods Appl. Mech. Eng. 360 (2020) 112789.
[23] J. Yu, L. Lu, X. Meng, G.E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems, Comput. Methods Appl. Mech. Eng. 393 (2022) 114823.
[24] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, J. Comput. Phys. 435 (2021) 110242.
[25] N. Sukumar, A. Srivastava, Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks, Comput. Methods Appl. Mech. Eng. 389 (2022) 114333.
[26] J. Hendriks, C. Jidling, A. Wills, T. Schön, Linearly constrained neural networks, arXiv preprint, arXiv:2002.01600, 2020.
[27] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, Deepxde: a deep learning library for solving differential equations, SIAM Rev. 63 (1) (2021) 208–228.
[28] H. Gao, L. Sun, J.-X. Wang, PhyGeoNet: physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state pdes on irregular domain, J. Comput. Phys. 428 (2021) 110079.
[29] P. Ren, C. Rao, Y. Liu, J.-X. Wang, H. Sun, PhyCRNet: physics-informed convolutional-recurrent network for solving spatiotemporal pdes, Comput. Methods Appl. Mech. Eng. 389 (2022) 114399.
[30] P.-H. Chiu, J.C. Wong, C. Ooi, M.H. Dao, Y.-S. Ong, CAN-PINN: a fast physics-informed neural network based on coupled-automatic–numerical differentiation method, Comput. Methods Appl. Mech. Eng. 395 (2022) 114909.
[31] P. Mistani, S. Pakravan, R. Ilango, S. Choudhry, F. Gibou, Neuro-symbolic partial differential equation solver, arXiv preprint, arXiv:2210.14907, 2022.
[32] P.A. Mistani, S. Pakravan, R. Ilango, F. Gibou, JAX-DIPS: neural bootstrapping of finite discretization methods and application to elliptic problems with discontinuities, J. Comput. Phys. 493 (2023) 112480.
[33] R. Sharma, V. Shankar, Accelerated training of physics-informed neural networks (pinns) using meshless discretizations, Adv. Neural Inf. Process. Syst. 35 (2022) 1034–1046.
[34] D. Grattarola, D. Zambon, F.M. Bianchi, C. Alippi, Understanding pooling in graph neural networks, IEEE Trans. Neural Netw. Learn. Syst. 35 (2) (2022) 2708–2718.
[35] J.E.S. Cardona, M. Hecht, Replacing automatic differentiation by Sobolev cubatures fastens physics informed neural nets and strengthens their approximation power, arXiv preprint, arXiv:2211.15443, 2022.
[36] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, M.W. Mahoney, Characterizing possible failure modes in physics-informed neural networks, Adv. Neural Inf. Process. Syst. 34 (2021) 26548–26560.
[37] J. Cho, S. Nam, H. Yang, S.-B. Yun, Y. Hong, E. Park, Separable physics-informed neural networks, arXiv preprint, arXiv:2306.15969, 2023.
[38] E.J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen Lora, Low-rank adaptation of large language models, arXiv preprint, arXiv:2106.09685, 2021.
[39] A. Griewank, D. Griffiths, G. Watson, ODE Solving via Automatic Differentiation and Rational Prediction, Techn. Univ., Rektor, 1995.
[40] J. Bettencourt, M.J. Johnson, D. Duvenaud, Taylor-mode automatic differentiation for higher-order derivatives in JAX, in: Program Transformations for ML Workshop at NeurIPS 2019, 2019.
[41] J. Kelly, J. Bettencourt, M.J. Johnson, D.K. Duvenaud, Learning differential equations that are easy to solve, Adv. Neural Inf. Process. Syst. 33 (2020) 4370–4380.
[42] M.J. Woodward, Y. Tian, C. Hyett, C. Fryer, D. Livescu, M. Stepanov, M. Chertkov, Physics informed machine learning of SPH: Machine learning Lagrangian turbulence, 2021.
[43] J.M. Siskind, B.A. Pearlmutter, Nesting forward-mode AD in a functional framework, High.-Order Symb. Comput. 21 (4) (2008) 361–376.
[44] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs, 2018.
[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An Imperative Style, High-Performance Deep Learning Library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.
[46] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.
[47] R.J. Gladstone, M.A. Nabian, H. Meidani, FO-PINNs: a first-order formulation for physics informed neural networks, arXiv preprint, arXiv:2210.14320, 2022.
[48] F. Hecht, New development in freefem++, J. Numer. Math. 20 (3–4) (2012) 251–265.
[49] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar, Fourier neural operator for parametric partial differential equations, arXiv preprint, arXiv:2010.08895, 2020.