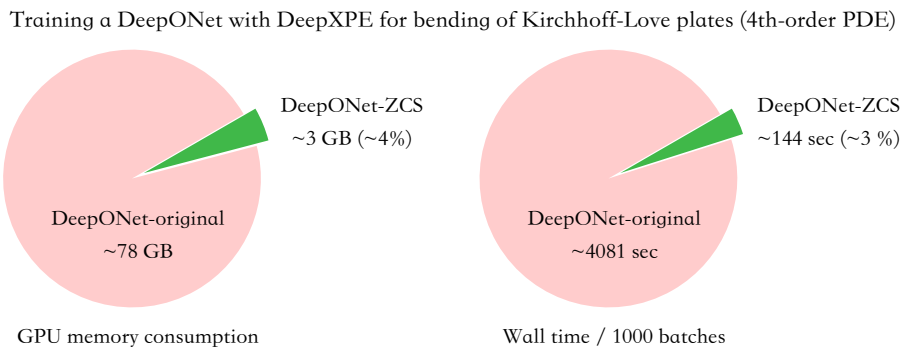
Graphical Abstract

Zero Coordinate Shift: Whetted Automatic Differentiation for Physics-informed Operator Learning

Kuangdai Leng, Mallikarjun Shankar, Jeyan Thiyagalingam



Highlights

Zero Coordinate Shift: Whetted Automatic Differentiation for Physics-informed Operator Learning

Kuangdai Leng, Mallikarjun Shankar, Jeyan Thiyagalingam

- We present a novel and lightweight algorithm to conduct automatic differentiation w.r.t. coordinates for physics-informed operator learning.
- Being at a low level, our algorithm neither affects training results nor imposes restrictions on data, physics (PDE) or network architecture.
- Our algorithm can persistently reduce GPU memory consumption and wall time for training physics-informed DeepONets by more than 90%; this percentage increases with problem scale.

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, US*

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. In this paper, we present a novel and lightweight algorithm to conduct the AD for physics-informed operator learning, as we call the trick of Zero Coordinate Shift (ZCS). Instead of making all sampled coordinates leaf variables, ZCS introduces only one scalar-valued leaf variable for each spatial or temporal dimension, leading to a game-changing performance leap by simplifying the wanted derivatives from "many-roots-many-leaves" to "one-root-many-leaves". ZCS is easy to implement with current deep learning libraries; our own implementation is 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 brought down GPU memory consumption and wall time for training by an order of magnitude, with the savings increasing with problem scale (i.e., number of functions, number of points and order of PDE). As a low-level optimisation, ZCS entails no restrictions on data, physics (PDEs) or network architecture and does not compromise training results from any aspect.

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

1. Introduction

Physics-informed machine learning offers a versatile framework for the emulation and inversion of partial differential equations (PDEs) from all subjects 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], targeted at learning the map $x \mapsto u$, with x being the coordinates of the collocation points sampled from the domain and u the target 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 DeepONets [7] and physics-informed Fourier neural operators

(FNOs) [8][1], which learn the map $p(x) \mapsto u(x)$, with $p(x)$ being any physical parameter of interest. 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 for 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 the PDE and any of its non-Dirichlet boundary conditions, one needs the high-order derivatives of the network output w.r.t. the coordinates. Besides, the gradient of the PDE itself may serve as an even stronger regularisation [9], which further increment the required orders of derivatives. The universal way for derivative computation is automatic differentiation (AD), a pivotal technique behind deep learning [10, 11, 12] and a major motivation of physics-informed machine learning [6, 1, 2].

Since the establishment of PINNs [6], related technical advancements have been sprouting, e.g., multi-scale and multi-frequency models [13, 14], domain decomposition techniques [15, 16], adaptive activation functions [17] and data sampling [18, 19], and embedding physics as hard constraints [20, 21, 22]. In light of such continuing efforts, it seems that less attention has been paid to AD, which, however, can bear a wellspring of impetus since the coordinate derivatives differ from 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 new algorithm to reshape AD for training PINOs (so naturally applicable to PINNs), which can reduce both GPU memory consumption and wall time by an order of magnitude. Our algorithm, as named the 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 neural networks (i.e., agnostic to the forward pass), entailing no changes to data structure, point sampling or network architecture. Its implementation is straightforward; our own implementation is by extending the DeepXDE package [23] by overriding a few methods.

The remainder of this paper is organised as follows. In the next section, we give a more insightful review over the related work for further justification of our motivation. In Section 3, we will describe the theory and implementation of our algorithm, after a brief recap of AD in deep learning. Next, our experiments are reported in Section 4, including a benchmark analysis exhibiting the scaling behaviour of our algorithm and three PDE operators learned without data. Identified limitations are then described in Section 5 before the paper being concluded by Section 6. Einstein summation convention is not adopted throughout this paper.

[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 [8] as also called PINOs by the authors.

2. Related work

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

2.1. Finite difference

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 [24] and PhyCRNet [25]. For an unstructured point cloud, the can-PINNs [26] make FD utilisable by adding many extra points in the forward pass to create small local grids anchored by the original ones; besides, the DT-PINNs [27] realise FD in a mesh-less manner via a direct interpolation among the nearest neighbours of a point, similar to global pooling in graph neural networks [28], whose implementation relies on sparse matrix-vector product. Apart from that a mesh grid for FD can be a strong prerequisite or induce a heavyweight overhead, the major caveat of FD is a generalisation error exponentially increasing with the order of derivatives, rooted from the well-understood Runge's phenomenon in polynomial interpolation and Gibbs phenomenon in trigonometric interpolation. In the cited studies, only second-order PDEs have been examined.

2.2. Analytical differentiation

Being both error-free and the fastest, analytical differentiation should be one's optimal choice if available. A good example is using the fast Fourier transform along the periodic spatial dimensions [8], which requires both periodicity and a structured grid. Besides, the SC-PINNs proposed in a recent study [29] 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 less relevant but mentionable works are the linearly constrained neural networks [22] and the exact imposition of Neumann and Robin boundary conditions [21], which bypass some of the derivatives by enforcing the physical constraints through architecture design. To encapsulate, analytical differentiation usually entails rather strong prerequisites (mostly from physics), but it is worth exploring if the target problem falls within the scope.

2.3. Enhanced AD

This category is directly linked to our work. Recently, Cho *et al.* [30] propose the Separable PINNs or SPINNs, which deserve detailing because we share one notion in common: reducing the amount of leaf variables in AD. The SPINNs use separation of variables to reduce the leaves, similar to the classic "separation of variables" in the 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). Such an idea is also similar to the Low-Rank Adaptation (LoRA) [31] 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, 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 large mesh grid (N^D) imposed upon the output and the PDE field. In comparison, our algorithm ZCS further reduces the number of leaves to D paying neither of the prices (since ZCS is entirely agnostic to network architecture and point sampling). Besides, as we will show in Section 3.2, PINNs can take full advantage of reverse-mode AD through a simple summation of roots, so the starting point of 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 [32, 33, 34, 35]. 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. On the other hand, we push one step ahead to exploit reverse-mode AD. The reason 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 large topic in deep learning. Here we summarise some of the key concepts that are essential for understanding the motivation of our algorithm. We refer the readers to [10, 11, 12] for in-depth reading.

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 numerically 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 (inherently vectorised) respectively for the situations of many-roots-one-leaf and one-root-many-leaves (as derived from the Jacobian-vector product and vector-Jacobian product). Nearly all deep learning tasks use reverse-mode AD because they normally involve one loss function (the root) and millions of network weights (the leaves). Consequently, the development of forward-mode AD lags behind. An inherent difficulty of forward-mode AD is nesting [36, 33, 34], as required for the second- and higher-order derivatives, whose computational cost increases exponentially with the order. At the time of writing, nested forward-mode AD has been supported by JAX [37] but not by PyTorch [38] and TensorFlow [39].

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 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, our 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 $\frac{\partial u_j}{\partial x_j}$, $\frac{\partial^2 u_j}{\partial x_j^2}$, $\cdots$. Because 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, the "vector" in the vector-Jacobian product being full of one):

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

The higher-order derivatives are computed in the same way, taking the lower-order results as the input. In brief, PINNs have made the most of reverse-mode AD. The simple fact given by eq. (2) makes us uncertain about the motivation of the SPINNs [30] for using forward-mode AD – the number of roots can always be one for PINNs.

The situation becomes more complicated 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 at some 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. No 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 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, can significantly impair the performance of training. The former is slacked up by the for-loop while the latter suffers unscalable memory due to massive data duplication ($2MN$ times). Furthermore, computing $\frac{\partial u_{ij}}{\partial x_j}$ is not the end of the game; what we eventually want is the gradient of the PDE-induced loss function w.r.t. the network weights θ. Both the two approaches will make the computational graph exceedingly large for backpropagation, as will be shown in our experiments. In short, the power of reverse-mode AD remains underused for 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}:

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

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

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

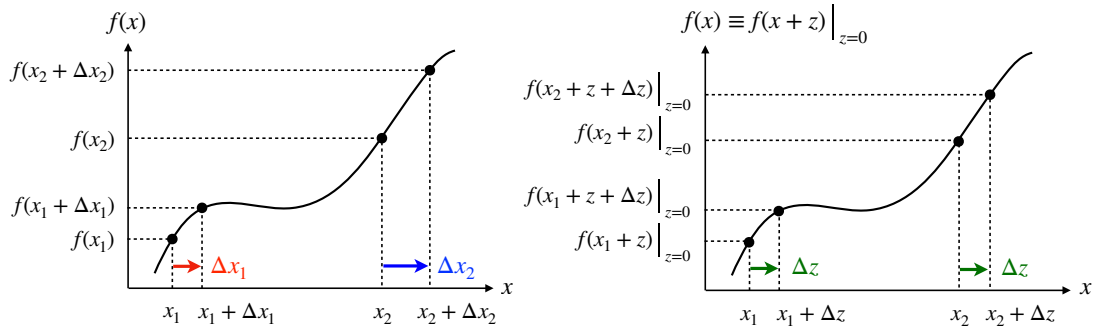
In the above proof, the step marked by $*$ is based on that x and z are symmetric in f_θ, and the step marked by $\dagger$ exchanges the order of derivative $(\partial/\partial z)$ and evaluation $(x = x_j)$, thanks to that f_θ is continuously differentiable w.r.t. x everywhere. The other steps involve 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 this is why we call our algorithm Zero Coordinate Shift (ZCS). Figure 1 is provided to facilitate the understanding of eq. 7 from the perspective of limits.

Being $\partial_1\infty$, eq. 7 has been made ready for forward-mode AD. However, given that forward-mode AD is currently immature, 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

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



(a) Taking both x_1 and x_2 as leaf variables

(b) Taking only z as a leaf variable

Figure 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, equivalent to taking x_1 and x_2 as independent leaf variables for AD. In (b), Δz is the only 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.

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:

$$\frac{\partial^n u_{ij}}{\partial x_j^n} = \frac{\partial}{\partial a_{ij}} \frac{\partial^n \omega}{\partial z^n}\bigg|_{z=0}, \tag{11}$$

and

$$\frac{\partial^m u_{ij}}{\partial x_j^m} \frac{\partial^n u_{ij}}{\partial x_j^n} = \frac{1}{2} \frac{\partial^2}{\partial a_{ij}^2} \left(\frac{\partial^m \omega}{\partial z^m} \frac{\partial^n \omega}{\partial z^n} \right)\bigg|_{z=0}. \tag{12}$$

The proofs are given in Appendix A. Equations (11) and (12) suggests 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 collect the linear and the non-linear terms to reduce the number

of $\partial_\infty 1$ ADs, using eq. (12) for the non-linear terms:

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

So far, we have addressed how to compute the PDE by our two-step AD algorithm. Accelerating the computation of the PDE, however, is not the biggest reason behind a huge performance leap. For PINOs, both θ and x must be made leaf variables, and the physical size of the computational graph for backpropagation (i.e., computing the gradient of the loss function w.r.t. θ) roughly scales with both the sizes of θ and x. In the previous PINOs, the size of x is usually large, depending on the number of sampled points, whereas ZCS can reduce this size to the number of dimensions. For instance, in a 3-D spatiotemporal domain, a modest 100^3-point sampling will feed 3×100^3 leaf scalars to the neural network; ZCS can bring this number all the way down to three. *Minimising the physical size of the computational graph is our largest source of memory and time savings*, as will be demonstrated in our experiments.

3.4. Implementation

Our ZCS algorithm is easy to implement using high-level APIs from any 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.

4. Experiments

In this section, we report our experiments. We implement our ZCS algorithm by extending the DeepXDE package [23] with PyTorch backend. We choose to extend DeepXDE for two reasons: to start from a solid state-of-the-art baseline and to show 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;

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: $\quad (\mathbf{q}, \mathbf{r}) \leftarrow \dfrac{\partial \sum_j u_{ij}}{\partial (\mathbf{x}, \mathbf{y})}$ ▷ $\partial_\infty 1$ AD: $\mathbf{q} \in \mathbb{R}^N$ for u_x; $\mathbf{r} \in \mathbb{R}^N$ for u_y

5: $\quad \mathbf{s} \leftarrow \dfrac{\partial \sum_j q_j}{\partial \mathbf{x}}, \mathbf{t} \leftarrow \dfrac{\partial \sum_j r_j}{\partial \mathbf{y}}$ ▷ $\partial_\infty 1$ AD: $\mathbf{s} \in \mathbb{R}^N$ for u_{xx}; $\mathbf{t} \in \mathbb{R}^N$ for u_{yy}

6: $\quad \mathbf{g}_i = \mathbf{s} + \mathbf{t}$ ▷ $u_{xx} + u_{yy}$ with parameter $\mathbf{p}_i$

7: **end for**

ZCS (ours)

1: $z_x \leftarrow 0, z_y \leftarrow 0$ ▷ Create ZCS scalars

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 7~9 are scalar-to-scalar derivatives

- `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 we concern here are GPU memory consumption and wall time for training*. All the experiments can be found at `https://github.com/stfc-sciml/ZeroCoordinateShift`, reproducible with one click.

4.1. Scaling analysis

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

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

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 Figure 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 smallest problems when $M \leq 10$). This also implies that the absolute savings will increase fast with the problem scale. Next, we look into the influence of of M, N and P individually.

- *Number of functions M* The first column of Figure 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. The reason why `FuncLoop` cannot be *much* less memory demanding than `DataVect` is that most of the GPU memory is consumed by the computational graph for backpropagation, which aggregates (instead of being overwritten) throughout the function loop. In contrast, both memory and time for `ZCS` increase extremely slowly with M, almost remaining constant when $M \leq 160$, thanks to that the branch net does not see the two ZCS scalars (z_x and z_y). Note that M will be the batch size for applications, 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 Figure 2, the models' scaling behaviours w.r.t. N are only slightly different from those w.r.t. M. First, the memory usage of `ZCS` scales with N across the whole range (i.e., no plateau at the small N's), as the two ZCS scalars are deep rooted in the computational graph of the trunk net. Second, both `FuncLoop` and `ZCS` exhibit a plateau of wall time at the small 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.

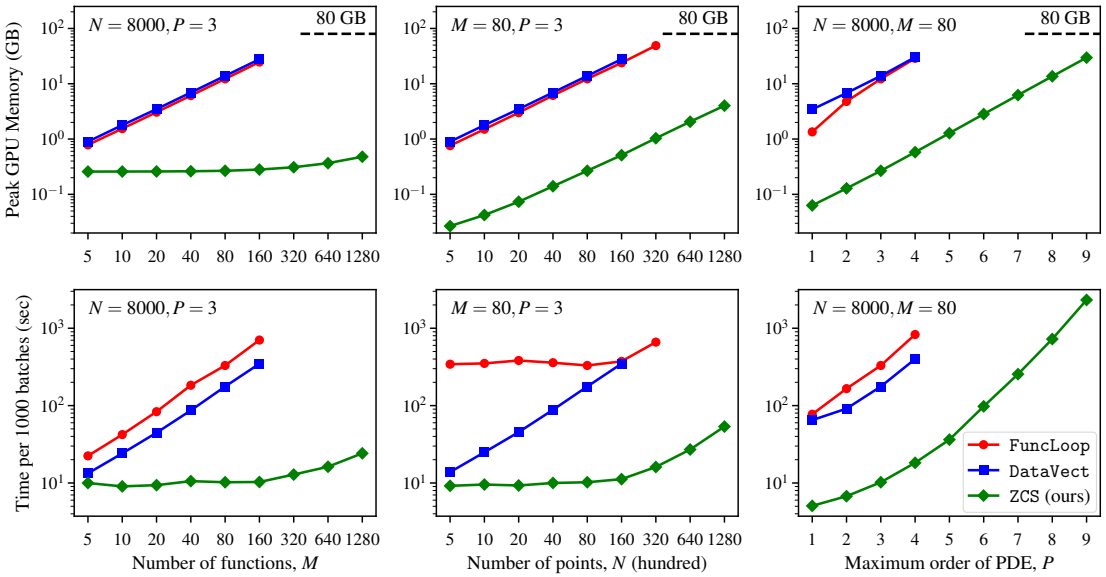


Figure 2: GPU memory consumption and wall time for training DeepONets with different AD strategies. The PDE is given by eq. (15), with a maximum differential order of P, and the function and point numbers M and N are defined in eq. (3). In the three columns from left to right, we vary respectively M,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

- *Maximum differential order P* The last column of Figure 2 suggests that P has the strongest impact on both memory and time (note that the horizontal axis shared by these two plots 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 high orders. In view of such scalability, the FO-PINNs [40], 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 memory and time into different stages, from which the above scaling behaviours can be better understood.

4.2. Operator learning

In this section, we train DeepONets to learn a few 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.

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

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

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 1000 $f(x)$'s sampled from a Gaussian process, learned with $M = 50$ (batch size) and $N = 1000$. 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 [8], 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 larger than that of the previous one in terms of N.

The bending of a square Kirchhoff-Love plate is employed as the third problem, 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 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.

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 impractically small. We manage to isolate the memory occupied by the computational graph, which reveals that the 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. As a side note, for `DataVect`, one can see a clear gap between the graph and peak memories as well as a fast increase 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.

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`	0.96	0.98	0	1	81	99	181	8.3±2.0%
		`DataVect`	0.97	1.46	5	3	4	12	24	9.5±2.5%
		`ZCS` (ours)	0.02	0.05	1	1	4	4	10	8.2±2.0%
Burgers	$M = 50$ $N = 12800$ $P = 2$	`FuncLoop`	7.84	7.91	1	2	140	173	316	7.5%
		`DataVect`	7.73	11.40	95	20	24	82	221	7.2%
		`ZCS` (ours)	0.20	0.36	1	2	7	5	15	7.1±0.5%
Kirchhoff-Love	$M = 36$ $N = 10000$ $P = 4$	`FuncLoop`	77.57	77.57	1	6	1765	2309	4081	27.3%
		`DataVect`	–	–	–	–	–	–	–	–
		`ZCS` (ours)	2.36	3.30	1	6	77	60	144	26.9±0.5%

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, "Inputs" for the time used to prepare input tensors, and "Loss (PDE)" for the time used to compute the physics-based losses (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. For Kirchhoff-Love, `DataVect` cannot be trained for insufficient memory (an error raised from CUDA showing an allocation of 190 GB in need); 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. Due to resource limitation, the different weight initialisations are skipped in the long runs of `FuncLoop` and `DataVect`.

Accuracy-wise, we do not obtain 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 reasonably low errors reported in Table 1 are sufficient to show that ZCS does not affect the training results (except for some randomness from stochastic AD). The 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) [24, 25] and FNOs [41, 8]. 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)

$$\mathbf{u} = f_\theta(\mathbf{x}, \mathbf{y}), \tag{20}$$

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

$$\begin{aligned} u_{I+1,J}(\mathbf{x}, \mathbf{y}) &= u_{I,J}(\mathbf{x} + \Delta x, \mathbf{y}), \\ u_{I,J+1}(\mathbf{x}, \mathbf{y}) &= u_{I,J}(\mathbf{x}, \mathbf{y} + \Delta y), \end{aligned} \tag{21}$$

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 some near-boundary pixels due to the issue of padding. Excluding such pixels, one can apply AD to calculate the PDE filed and optionally employ ZCS to boost the performance of AD (i.e., feeding two scalars 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 notably more memory demanding and slower than FD and analytical differentiation (such as fast Fourier transform [8]), as both of them lightly enlarge the computational graph – this is a key motivation for using a structured grid. Simply put, what has been discussed in this section is not a limitation of ZCS itself but a downside of any pointwise PINOs (i.e., eq. 3) as they do not take special advantage of structured data when available. In turn, not restricting point sampling makes pointwise PINOs 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 (PINO) 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 physical parameter dimension. 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 around AD, agnostic to data, point sampling and network architecture. It does not affect training results and is easy to implement with any deep learning libraries.

We implement ZCS by extending the DeepXDE package. Based on this implementation, we carry out several experiments, comparing our algorithm to the existing workaround approaches: loop over functions (physical parameters) and data vectorisation. The results show that ZCS has persistently reduced the GPU memory consumption and wall time for training DeepONets by an order of magnitude. The savings grow with the problem scale. The memory and time measurements reveal that such a performance leap has rooted from a diminish computational graph for backpropagation, thanks to that ZCS brings down the number of leaf variables from thousands or millions to a very few (i.e., the number of dimensions). Our code and experiments can be found at `https://github.com/stfc-sciml/ZeroCoordinateShift`.

Acknowledgements

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) laboratories in the US.

References

[1] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nature Reviews Physics 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, Journal of Scientific Computing 92 (2022) 88.

[3] L. Yang, D. Zhang, G. E. Karniadakis, Physics-informed generative adversarial networks for stochastic differential equations, SIAM Journal on Scientific Computing 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, Journal of Computational Physics 462 (2022) 111260.

[5] G. Pang, L. Lu, G. E. Karniadakis, fPINNs: Fractional physics-informed neural networks, SIAM Journal on Scientific Computing 41 (4) (2019) A2603–A2626.

[6] 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, Journal of Computational physics 378 (2019) 686–707.

[7] L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, Nature machine intelligence 3 (3) (2021) 218–229.

[8] 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).

[9] J. Yu, L. Lu, X. Meng, G. E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse pde problems, Computer Methods in Applied Mechanics and Engineering 393 (2022) 114823.

[10] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, Journal of Marchine Learning Research 18 (2018) 1–43.

[11] C. C. Margossian, A review of automatic differentiation and its efficient implementation, Wiley interdisciplinary reviews: data mining and knowledge discovery 9 (4) (2019) e1305.

[12] 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).

[13] Ziqi, Liu, , 9594, , Z. Liu, Wei, Cai, , 9595, , W. Cai, Zhi-Qin, J. Xu, , 9596, , Z.-Q. J. Xu, Multi-scale deep neural network (mscalednn) for solving poisson-boltzmann equation in complex domains, Communications in Computational Physics 28 (5) (2020) 1970–2001.

[14] W. Cai, X. Li, L. Liu, A phase shift deep neural network for high frequency approximation and wave problems, SIAM Journal on Scientific Computing 42 (5) (2020) A3285–A3312.

[15] 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).

[16] E. Kharazmi, Z. Zhang, G. E. Karniadakis, hp-vpinns: Variational physics-informed neural networks with domain decomposition, Computer Methods in Applied Mechanics and Engineering 374 (2021) 113547.

[17] A. D. Jagtap, K. Kawaguchi, G. Em Karniadakis, Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks, Proceedings of the Royal Society A 476 (2239) (2020) 20200334.

[18] Z. Mao, A. D. Jagtap, G. E. Karniadakis, Physics-informed neural networks for high-speed flows, Computer Methods in Applied Mechanics and Engineering 360 (2020) 112789.

[19] J. Yu, L. Lu, X. Meng, G. E. Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse pde problems, Computer Methods in Applied Mechanics and Engineering 393 (2022) 114823.

[20] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, Journal of Computational Physics 435 (2021) 110242.

[21] N. Sukumar, A. Srivastava, Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks, Computer Methods in Applied Mechanics and Engineering 389 (2022) 114333.

[22] J. Hendriks, C. Jidling, A. Wills, T. Schön, Linearly constrained neural networks, arXiv preprint arXiv:2002.01600 (2020).

[23] L. Lu, X. Meng, Z. Mao, G. E. Karniadakis, Deepxde: A deep learning library for solving differential equations, SIAM Review 63 (1) (2021) 208–228.

[24] H. Gao, L. Sun, J.-X. Wang, PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state pdes on irregular domain, Journal of Computational Physics 428 (2021) 110079.

[25] P. Ren, C. Rao, Y. Liu, J.-X. Wang, H. Sun, PhyCRNet: Physics-informed convolutional-recurrent network for solving spatiotemporal pdes, Computer Methods in Applied Mechanics and Engineering 389 (2022) 114399.

[26] 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, Computer Methods in Applied Mechanics and Engineering 395 (2022) 114909.

[27] R. Sharma, V. Shankar, Accelerated training of physics-informed neural networks (pinns) using meshless discretizations, Advances in Neural Information Processing Systems 35 (2022) 1034–1046.

[28] D. Grattarola, D. Zambon, F. M. Bianchi, C. Alippi, Understanding pooling in graph neural networks, IEEE Transactions on Neural Networks and Learning Systems (2022).

[29] 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).

[30] J. Cho, S. Nam, H. Yang, S.-B. Yun, Y. Hong, E. Park, Separable physics-informed neural networks, arXiv preprint arXiv:2306.15969 (2023).

[31] 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).

[32] A. Griewank, D. Griffiths, G. Watson, ODE solving via automatic differentiation and rational prediction, Techn. Univ., Rektor, 1995.

[33] 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.

[34] J. Kelly, J. Bettencourt, M. J. Johnson, D. K. Duvenaud, Learning differential equations that are easy to solve, Advances in Neural Information Processing Systems 33 (2020) 4370–4380.

[35] 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).

[36] J. M. Siskind, B. A. Pearlmutter, Nesting forward-mode ad in a functional framework, Higher-Order and Symbolic Computation 21 (4) (2008) 361–376.

[37] 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).

[38] 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.

[39] 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).

[40] 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).

[41] 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).

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):

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

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 $\frac{\partial^2 \omega}{\partial a_{ij}^2} = 0$.