Separable PINN: Mitigating the Curse of Dimensionality in Physics-Informed Neural Networks

Junwoo Cho[1*] **Seungtae Nam**[1*] **Hyunmo Yang**[1] **Seok-Bae Yun**[2] **Youngjoon Hong**[2] **Eunbyung Park**[1,3†]

[1]Department of Artificial Intelligence [2]Department of Mathematics
[3]Department of Electrical and Computer Engineering
Sungkyunkwan University
{jwcho000, stnamjef, cms8033, sbyun01, hongyj, epark}@skku.edu

Abstract

Physics-informed neural networks (PINNs) have emerged as new data-driven PDE solvers for both forward and inverse problems. While promising, the expensive computational costs to obtain solutions often restrict their broader applicability. We demonstrate that the computations in automatic differentiation (AD) can be significantly reduced by leveraging forward-mode AD when training PINN. However, a naive application of forward-mode AD to conventional PINNs results in higher computation, losing its practical benefit. Therefore, we propose a network architecture, called separable PINN (SPINN), which can facilitate forward-mode AD for more efficient computation. SPINN operates on a per-axis basis instead of point-wise processing in conventional PINNs, decreasing the number of network forward passes. Besides, while the computation and memory costs of standard PINNs grow exponentially along with the grid resolution, that of our model is remarkably less susceptible, mitigating the curse of dimensionality. We demonstrate the effectiveness of our model in various PDE systems by significantly reducing the training run-time while achieving comparable accuracy. Project page: `https://jwcho5576.github.io/spinn/`

1 Introduction

With the vast increases in computational power and breakthroughs in machine learning, researchers have explored data-driven and learning-based methods for solving partial differential equations [1, 3, 4, 5, 9, 10, 12, 13]. Among the promising methods, physics-informed neural networks (PINNs) have recently emerged as new data-driven PDE solvers for both forward and inverse problems [11]. PINNs employ neural networks and gradient-based optimization algorithms to represent and obtain the solutions, leveraging automatic differentiation to enforce physical constraints of underlying PDEs. It has enjoyed great success in various forward and inverse problems thanks to its numerous benefits, such as flexibility in handling a wide range of forward and inverse problems, mesh-free solutions, and not requiring observational data, hence, unsupervised training.

PINNs, using neural networks and gradient-based iterative optimization algorithms, generally require a large number of iterations to converge. Furthermore, the exponential growth of the number of forward and backward propagations due to the curse of dimensionality restricts their capability in solving high-dimensional PDEs. It primarily stems from using coordinate-based MLP architectures to represent the solution function, which takes input coordinates and outputs corresponding solution

[*]Equal contribution, alphabetically ordered.
[†]Corresponding author.

DLDE Workshop in the 36th Conference on Neural Information Processing Systems (NeurIPS 2022).

quantities. For each training point, computing PDE residual loss involves multiple forward and backward propagations, and the number of training points required to obtain more accurate solutions grows substantially. This paper aims to find a solution architecture to this fundamental limitation of training PINNs.

We propose a novel PINN architecture, *separable PINN (SPINN)*, which utilizes forward-mode automatic differentiation (AD)* to significantly reduce the computational cost for solving multi-dimensional PDEs. Instead of feeding every multi-dimensional coordinate into a single MLP, e.g. $f(x_1, x_2, x_3)$, we use factorized coordinates and separated sub-networks, where each sub-network takes independent one-dimensional coordinates as input, e.g. $f_1(x_1), f_2(x_2), f_3(x_3)$. The final output is generated by a simple outer product and element-wise summation (Fig 3). The suggested architecture obviates the need to query every multi-dimensional coordinate input pair, exponentially reducing the number of network propagations to generate a solution, $\mathcal{O}(n^d) \to \mathcal{O}(nd)$, where n is the resolution of the solution for each dimension, and d is the input dimension.

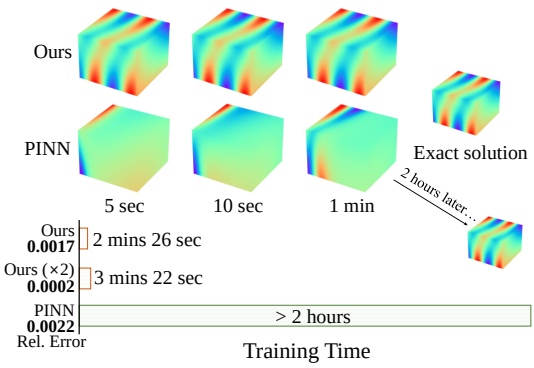


Figure 1: Training wall-clock time and relative errors compared against PINN on Klein-Gordon equation. 'Ours ($\times 2$)' denotes our model trained on $\times 2$ resolution grid. Our model significantly reduces the computational cost without any loss in accuracy.

Experimental results demonstrate that given the same number of training points, SPINN significantly reduces the computational costs both in terms of FLOPs and wall-clock training time on commodity GPUs while achieving better accuracy. Figure 1 shows the run-time of training conventional PINN and our proposed method trained on a (2+1)-dimensional Klein-Gordon equation.

2 Forward-Mode AD with Separated Functions

We demonstrate that leveraging forward mode AD and separating the function into multiple functions can significantly reduce the computational cost of computing Jacobian matrix $\mathbb{J}$. In the separated approach (Fig 2 (b)), we first sample n one-dimensional coordinates on each of d axes, which makes a total of nd points. Next, these coordinates are fed into d individual functions $f_1, \ldots, f_d$ to generate the features. We denote the number of operations of f as $\mathsf{ops}(f)$. Then, a feature merging function g is used to construct the solution of the entire n^d discretized domain. The amount of computations for AD is known to be 2 or 3 times more expensive than the forward propagation [2, 6]. With the scale constants $c_f, c_g \in [2, 3]$ we can approximate the total number of operations to compute the Jacobian matrix of a separated approach,

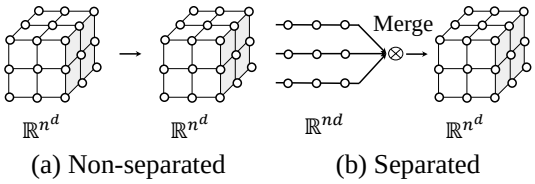


Figure 2: Non-separated vs separated. (a) $\mathbb{J}$ (Jacobian matrix) of a non-separated function approach is a $n^d \times n^d$ matrix. Computing $\mathbb{J}$ requires $\mathcal{O}(n^d)$ evaluations in both forward and reverse mode AD. (b) $\mathbb{J}$ is a $nd \times n^d$ matrix in a separated approach. $\mathcal{O}(nd)$ evaluations are required using forward-mode AD, but $\mathcal{O}(n^d)$ with reverse-mode AD.

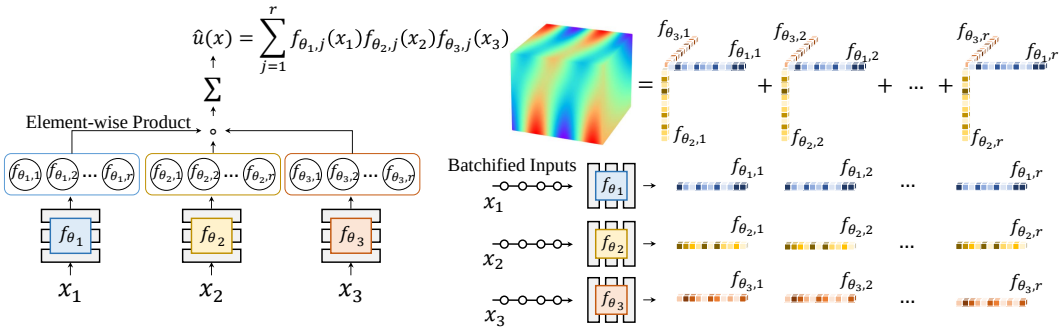
$$\mathcal{C}_{\text{sep}} = ndc_f\mathsf{ops}(f) + c_g\mathsf{ops}(g). \tag{1}$$

For a non-separated approach (Fig 2 (a)),

$$\mathcal{C}_{\text{non-sep}} = n^d c_f \mathsf{ops}(f). \tag{2}$$

If we can make $\mathsf{ops}(g)$ sufficiently small, then the ratio $\mathcal{C}_{\text{sep}}/\mathcal{C}_{\text{non-sep}}$ becomes $\frac{nd}{n^d} \ll 1$, quickly converging to 0 as d and n increases. This mitigates the curse of dimensionality for solving high-dimensional PDEs. It also has linear complexity with respect to n, implying it can obtain more accurate solutions (high-resolution) efficiently.

*a.k.a. forward accumulation mode or tangent linear mode.



(a) Architecture (b) Batchfied processing (Low-rank tensor decomposition)

Figure 3: (a) SPINN architecture in a 3-dimensional system. To solve d-dimensional PDE system, our model requires d body MLP networks. (b) Batchfied processing. SPINN learns a low-rank decomposed representation of a solution.

3 Separable PINN

Figure 3 (a) illustrates the overall SPINN architecture, parameterizing multiple separated functions with neural networks. SPINN consists of d body-networks (MLPs), each of which takes an individual 1-dimensional coordinate component as an input. Each body-network $f_{\theta_i} : \mathbb{R} \to \mathbb{R}^r$ (parameterized by θ_i) transforms an i-th input coordinate into a feature representation. We define our predicted solution function as below:

$$\hat{u}(x) = \sum_{j=1}^{r} \prod_{i=1}^{d} f_{\theta_i,j}(x_i) \tag{3}$$

where $x \in \mathbb{R}^d$ is an input coordinate, x_i denotes i-th input coordinate. f_{θ_i} is a vector-valued function and $f_{\theta_i,j}$ denotes the j-th component of the output vector. We used a tanh activation function throughout the paper. In Eq. 3, the feature merging operation is a simple product and summation which corresponds to the merging function g described in Eq. 1. Due to its simplicity, g operations are cheaper than operations in MLP layers and thus, neglectable. This makes the assumption $\mathcal{C}_{\text{sep}}/\mathcal{C}_{\text{non-sep}} \approx \frac{nd}{n^d}$ in Eq. 1 and 2 valid if we choose a such feature merging scheme.

The evaluated coordinate points of our model and conventional PINN have a distinct difference. Both are *uniformly* evaluated on a d-dimensional hypercube, but points of SPINN form a lattice-like structure, which we call as *factorizable coordinates*. This is because SPINN samples 1-dimensional coordinates from each axis and merges the features with outer product. On the other hand, non-factorizable coordinates are random points without any structure. Factorizable coordinates with our separated MLP architecture enable us to evaluate functions on a much more dense (n^d) grid, with a small number (nd) of inputs. Due to its structural points and outer products between feature vectors, SPINN's solution approximation can also be viewed as a low-rank tensor decomposition (Figure 3 (b)). Among many decomposition methods, SPINN corresponds to CP-decomposition [7] which approximates a tensor by finite summation of rank-1 tensors. While traditional methods use iterative methods such as alternating least-squares (ALS) or alternating slicewise diagonalization (ASD) [8] to directly fit the decomposed vectors, we train neural networks which *learn* a continuous representation, enabling us to take derivative w.r.t. input coordinates.

4 Experiments

We compared SPINN against vanilla PINN [11] on 3-d systems. For every experiment, the PDE loss is calculated on fixed 90^3 collocation points, which means we sampled $90 \cdot 3$ factorized coordinates for SPINN, while 90^3 points are directly fed into PINN. The number of collocation points of 90^3 was the upper limit for the baseline model when we trained with a single NVIDIA A100 GPU with 40GB of memory. However, the memory usage of our model was significantly smaller, enabling SPINN to use more collocation points (180^3 and 360^3) to get more accurate solutions. All experiments are trained with full batch for both models and reported error metrics are average relative L_2 errors computed by $\|\hat{u} - u\|^2 / \|u\|^2$, where $\hat{u}$ is the model prediction and u is the reference solution.

Table 1 shows our overall results. SPINN is significantly more computationally efficient than the baseline PINN in terms of wall-clock run-time. Klein-Gordon equation showed the largest gap, where our model achieved $57\times$ speed-up given the same input size. With the help of the method outlined in [6], we also estimated the FLOPs for evaluating the derivatives. In comparison to the baseline, SPINN requires $1,195\times$ less computations to evaluate the forward pass, first and second order derivatives. See section D in the appendix for details.

In all cases, our model finds the most accurate solution. Furthermore, while the number of collocation points grows exponentially, the memory usage and actual run-time of our model show logarithmic growth (Figure 4). SPINN is also memory-efficient because it stores a much smaller batch of tensors, which are the intermediate values (primals) while building the computational graph. We also identified from the results that we get more accurate solutions with more collocation points. This training characteristic of PINN substantiates that our method is very effective for solving multi-dimensional PDEs. Figure 5 shows the error curves with run-time and visualized solutions of Helmholtz and Klein-Gordon equation. Additional experimental details and analyses are provided in the appendix.

system (3-d)	model	relative L_2 error	memory (MB)	run-time (ms/iter.)
linear diffusion	PINN (90^3)	0.0045±0.0002	35,640	129.84
	Ours (90^3)	0.0031±0.0002	832	2.64
	Ours (180^3)	**0.0029**±0.0002	964	3.59
	Ours (360^3)	0.0031±0.0002	1,596	7.48
nonlinear diffusion	PINN (90^3)	0.0076±0.0003	35,640	139.82
	Ours (90^3)	0.0061±0.0009	864	3.01
	Ours (180^3)	**0.0047**±0.0006	1,348	4.51
	Ours (360^3)	0.0049±0.0007	5,180	11.59
Helmholtz	PINN (90^3)	0.1501±0.0272	35,640	166.06
	Ours (90^3)	0.0730±0.0369	856	2.95
	Ours (180^3)	0.0623±0.0143	964	4.16
	Ours (360^3)	**0.0514**±0.0204	2,108	9.57
Klein-Gordon	PINN (90^3)	0.0092±0.0022	35,640	166.44
	Ours (90^3)	0.0016±0.0017	888	2.93
	Ours (180^3)	**0.0009**±0.0002	1,132	4.04
	Ours (360^3)	**0.0009**±0.0003	3,164	9.13

Table 1: Overall results of 3-d systems. n^3 denotes the number of collocation points. The training run-time (milliseconds per iteration) of our model is substantially lower than the baseline while achieving higher accuracy.

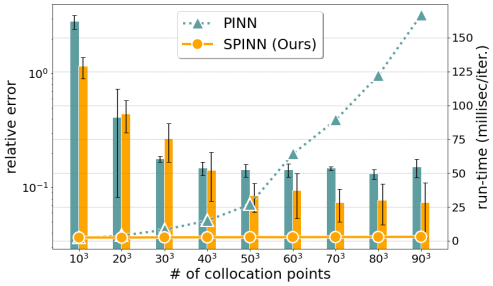


Figure 4: Our model mitigates the curse of dimensionality. It shows the relative error and run-time of the Helmholtz equation, on a varying number of collocation points. The bar graph shows the relative L_2 error (left y-axis), and the line graph shows the run-time in milliseconds (right y-axis) per iteration. We get more accurate solutions as the number of collocation points increases. The run-time of the baseline model grows exponentially with respect to the resolution, while our model keeps it almost constant.

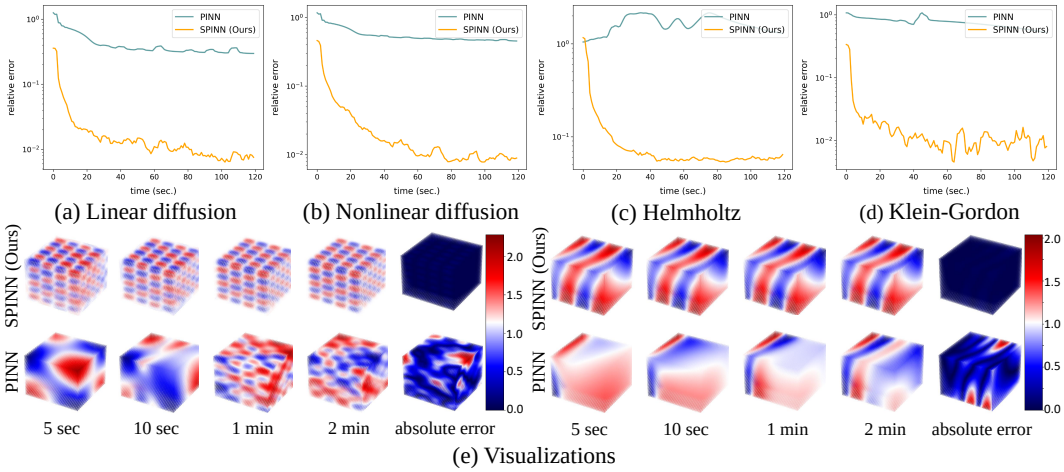


Figure 5: (a) to (d) Error curves of each experiment trained on 90^3 collocation points. The x-axis shows the actual run-time in seconds during training. (e) Visualized solutions at 4 time stamps (left: 3-d Helmholtz, right: 3-d Klein-Gordon). Absolute error is plotted at training time 2 min.

References

[1] Vsevolod I Avrutskiy. Neural networks catching up with finite differences in solving partial differential equations in higher dimensions. *Neural Computing and Applications*, 32(17):13425–13440, 2020.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[3] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.

[4] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual review of fluid mechanics*, 52:477–508, 2020.

[5] Weinan E, Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in mathematics and statistics*, 5(4):349–380, 2017.

[6] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[7] Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927.

[8] Jian-hui Jiang, Hai-long Wu, Yang Li, and Ru-qin Yu. Three-way data resolution by alternating slice-wise diagonalization (asd) method. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 14(1):15–36, 2000.

[9] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[10] Maziar Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research*, 19(1):932–955, 2018.

[11] Maziar Raissi, Paris Perdikaris, and George 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*, 378:686–707, 2019.

[12] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[13] Jared Willard, Xiaowei Jia, Shaoming Xu, Michael Steinbach, and Vipin Kumar. Integrating scientific knowledge with machine learning for engineering and environmental systems, 2020.

A Related Works on Separated Architectures

CoordX [6] and NAM [1] have recently explored using separated MLP architectures. Unlike ours, CoordX had to use additional layers after the feature merging step, which makes their model more computationally expensive than ours. In addition, their primary purpose is to reconstruct natural signals, such as images or 3D shapes. We focus on solving PDEs and carefully devised the architecture to exploit forward-mode differentiation to efficiently compute PDE residual losses. NAM [1] suggested separated network architectures and inputs, but only for achieving interpretability of the model's prediction. TensoRF [3] also shares similarities with ours. It used CP-decomposition to factorize the output with low-rank tensors (Figure 3 (b)). However, TensoRF *directly optimizes* low-rank tensors and uses tri-linear interpolation to implement continuous functions, or neural fields [11]. On the other hand, we use coordinated-based MLPs to predict low-rank tensors, which is a critical difference since solving PDEs mandates computing partial derivatives with respect to input coordinates. FBPINNs [8] is another approach to use multiple MLPs to solve PDEs. They decompose *input domains*, and each small MLP is used to cover a particular subdomain. In contrast, we decompose *input dimensions* and solve PDEs over the entire domain cooperated by all separated MLPs. To the best of our knowledge, this is the first work to accelerate PINN training with separated MLPs leveraging forward-mode AD.

B Forward/Reverse-Mode AD

In this section, we provide some background on the two modes of AD and how Jacobian matrices are evaluated. For clarity, we will follow the notations used in [2] and [4]. Suppose our function $f : \mathbb{R}^n \to \mathbb{R}^m$ is a two-layers MLP with tanh activation. First column of Table 2 demonstrates a single forward trace of f. To obtain a $m \times n$ Jacobian matrix $\mathbb{J}_f$ in forward-mode, we compute the Jacobian-vector product (JVP),

Forward primal trace	Forward tangent trace	Backward adjoint trace
$v_0 = x$	$\dot{v}_0 = \dot{x}$	$\bar{x} = \bar{v}_0$
$v_1 = W_1 \cdot v_0$	$\dot{v}_1 = W_1 \cdot \dot{v}_0$	$\bar{v}_0 = \bar{v}_1 \cdot \frac{\partial v_1}{\partial v_0} = \bar{v}_1 \cdot W_1$
$v_2 = \tanh(v_1)$	$\dot{v}_2 = \tanh'(v_1) \circ \dot{v}_1$	$\bar{v}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = \bar{v}_2 \circ \tanh'(v_1)$
$v_3 = W_2 \cdot v_2$	$\dot{v}_3 = W_2 \cdot \dot{v}_2$	$\bar{v}_2 = \bar{v}_3 \cdot \frac{\partial v_3}{\partial v_2} = \bar{v}_3 \cdot W_2$
$y = v_3$	$\dot{y} = \dot{v}_3$	$\bar{v}_3 = \bar{y}$

Table 2: An example of forward and reverse mode AD in a two-layers tanh MLP. Here v_0 denotes the input variable, v_k the primals, $\dot{v}_k$ the tangents, $\bar{v}_k$ the adjoints, and W_1, W_2 the weight matrices. Biases are omitted for brevity.

$$\mathbb{J}_f r = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \partial x_1 / \partial x_i \\ \vdots \\ \partial x_n / \partial x_i \end{bmatrix}, \tag{4}$$

for $i \in \{1, \ldots, n\}$. The forward-mode AD is a one-phase process: while tracing primals (intermediate values), v_k, it continues to evaluate and accumulate their tangents $\dot{v}_k = \partial v_k / \partial x_i$. This is equivalent to decomposing one large JVP into a series of JVPs by the chain rule and computing them from right to left. A run of JVP with the initial tangents $\dot{v}_0$ as the first column vector of an identity matrix I_n gives the first column of $\mathbb{J}_f$. Thus, the full Jacobian can be obtained in n evaluations.

On the other hand, the reverse-mode AD computes vector-Jacobian product (VJP):

$$r^\top \mathbb{J}_f = [\partial y_j / \partial y_1 \quad \ldots \quad \partial y_j / \partial y_m] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}, \tag{5}$$

for $j \in \{1, \ldots, m\}$, which is the reverse-order operation of JVP. This is a two-phase process. The first phase corresponds to forward propagation, storing all the primals, v_k, and recording the elementary operations in the computational graph. In the second phase, the derivatives are computed by accumulating the adjoints $\bar{v}_k = \partial y_j / \partial v_k$ (third of Table 2). Since VJP builds one row of a Jacobian at a time, it takes m evaluations to obtain the full Jacobian. To sum up, the forward-mode is more efficient for a tall Jacobian ($m > n$), while the reverse-mode is better suited for a wide Jacobian ($n > m$). Please refer to [2] for a more comprehensive review.

C Training with Physics-Informed Loss

After SPINN predicts an output function with the methods described in section 3, the rest of the training procedure follows the same process used in conventional PINN training [9], except we use forward mode AD (standard back-propagation, a.k.a. reverse mode AD, used for parameter updates). With the slight abuse of notation, our predicted solution function is denoted as $\hat{u}(x, t)$, explicitly expressing time coordinates. Given an underlying PDE (or ODE) and initial, boundary conditions, SPINN is trained with a "physics-informed" loss function:

$$\min_{\theta} \mathcal{L}(\hat{u}_\theta(x,t)) = \min_{\theta} \lambda_{\text{pde}}\mathcal{L}_{\text{pde}} + \lambda_{\text{ic}}\mathcal{L}_{\text{ic}} + \lambda_{\text{bc}}\mathcal{L}_{\text{bc}}, \tag{6}$$

$$\mathcal{L}_{\text{pde}} = \int_{\Gamma}\int_{\Omega} \|\mathcal{N}_{x,t}[\hat{u}_\theta](x,t)\|^2 dxdt, \tag{7}$$

$$\mathcal{L}_{\text{ic}} = \int_{\Omega} \|\hat{u}_\theta(x,0) - u_{\text{ic}}(x)\|^2 dx, \tag{8}$$

$$\mathcal{L}_{\text{bc}} = \int_{\Gamma}\int_{\partial\Omega} \|\mathcal{B}_{x,t}[\hat{u}_\theta](x,t) - u_{\text{bc}}(x,t)\|^2 dxdt, \tag{9}$$

where Ω is an input domain, $\mathcal{N}, \mathcal{B}$ are generic differential operators and $u_{\text{ic}}, u_{\text{bc}}$ are initial, boundary conditions, respectively. λ are weighting factors for each loss term. When calculating the PDE loss ($\mathcal{L}_{pde}$) we sampled collocation points from factorized coordinates and used forward-mode AD. The remaining $\mathcal{L}_{\text{ic}}$ and $\mathcal{L}_{\text{bc}}$ are then computed with initial and boundary coordinates to regress the given conditions. By minimizing the objective loss in Eq. 6, the model output is enforced to satisfy the given equation and the initial, boundary conditions.

D FLOPs Estimation

The FLOPs for evaluating the derivatives can be systematically calculated by disassembling the computational graph into elementary operations such as additions and multiplications. Given the computational graph of forward pass for computing the primals, AD augments each elementary operation into other elementary operations. The FLOPs in the forward pass can be calculated precisely since it consists of a series of matrix multiplications and additions. We used the method described in [4] to estimate FLOPs for evaluating the derivatives. Table 3 shows the number of additions (ADDS), and multiplications (MULTS) in each evaluation process. Note that FLOPs is a summation of ADDS and MULTS by definition.

	SPINN (ours)		PINN (baseline)	
	ADDS ($\times 10^6$)	MULTS ($\times 10^6$)	ADDS ($\times 10^6$)	MULTS ($\times 10^6$)
forward pass	39	40	36,742	36,742
1st-order derivative	79	80	147,404	73,921
2nd-order derivative	159	160	221,324	148,279
MFLOPs (total)	556		664,411	

Table 3: The number of elementary operations for evaluating forward pass, first and second-order derivatives. The calculation is based on 90^3 collocation points in a 3-d system and the MLP settings described in section E for both models. We assumed that each derivative is evaluated on every coordinate axis.

E Experimental Details and Analyses

We matched the total number of learnable parameters between each model for a fair comparison. For our model, we used 3 body networks of 5 hidden layers with 50 hidden/output feature sizes each. For the baseline model, we used a single MLP of 5 hidden layers with 100 hidden feature sizes. We used Adam [5] optimizer with a learning rate of 0.001 and trained for 50,000 iterations for every experiment. All experiments were performed five times with different random seeds. All weight factors λ in the loss function in Eq. 6 are set to 1. In the following sections, we give more detailed descriptions and analyses of each experiment.

E.1 Linear/Non-linear Diffusion Equation

Diffusion equation is one of the most representative parabolic PDEs, often used for modeling the heat diffusion process. It can be written as:

$$\partial_t u = \alpha \Delta u, \qquad x \in \Omega, t \in \Gamma, \tag{10}$$

$$\partial_t u = \alpha \left(\|\nabla u\|^2 + u \Delta u \right), \qquad x \in \Omega, t \in \Gamma, \tag{11}$$

$$u(x, 0) = u_{ic}(x), \qquad x \in \Omega, \tag{12}$$

$$u(x, t) = 0, \qquad x \in \partial\Omega, t \in \Gamma, \tag{13}$$

where Eq. 10 is a linear form and Eq. 11 is a non-linear form. We used diffusivity $\alpha = 0.05$, spatial domain $\Omega \in [-1, 1]^2$, temporal domain $\Gamma \in [0, 1]$ and used superposition of three Gaussian functions for initial condition u_{ic}. For error measurement, we compared against the reference solution obtained by a widely-used PDE solver platform FEniCS [7]. Note that FEniCS is a FEM-based solver. The first and second row of Table 1 shows the numerical results of the linear and non-linear diffusion equation, respectively. With $40\times$ less memory usage, our method finds more accurate solutions, $49\times$ ($47\times$ for nonlinear case) faster than the baseline where 180^3 collocation points showed the lowest error.

E.2 Helmholtz Equation

The Helmholtz equation is a time-independent wave equation that takes the form:

$$\Delta u + k^2 u = q, \qquad x \in \Omega, \tag{14}$$

$$u(x) = 0, \qquad x \in \partial\Omega, \tag{15}$$

where the spatial domain is $\Omega \in [-1, 1]^3$. For a given source term $q = -(a_1\pi)^2 u - (a_2\pi)^2 u - (a_3\pi)^2 u + k^2 u$, we devised a manufactured solution $u = \sin(a_1\pi x_1)\sin(a_2\pi x_2)\sin(a_3\pi x_3)$, where we take $k = 1, a_1 = 3, a_2 = 3, a_3 = 2$. The results of our experiments are shown in the third row of Table 1. Due to stiffness in the gradient flow, it hinders conventional PINNs to find accurate solutions. Therefore, a learning rate annealing algorithm is suggested to mitigate such phenomenon [10]. However, without bells and whistles, SPINN successfully finds accurate solutions. On the same number of collocation points, the memory usage of our model is $41\times$ smaller, and the training speed is $56\times$ faster than the baseline. Visualized solutions are illustrated in Figure 5 (e) left.

Helmholtz equation contains three second-order derivative terms, which makes the baseline model most time-consuming (more than two hours for full training). Meanwhile, the linear diffusion equation contains one first-order derivative and two second-order derivative terms, which has fewer second-order derivatives than Helmholtz. Comparing Helmholtz's experiment against the linear diffusion experiment, the training time increased 28% ($129.84 \rightarrow 166.06$) for the baseline model, but SPINN has only increased 12% ($2.64 \rightarrow 2.95$). This is due to SPINN's significant reduction of the forward-mode AD computations, which makes our model also very efficient at solving high-order PDEs.

Figure 4 also shows the result of training Helmholtz equation to identify the error and run-time dynamics along the number of collocation points. We increased the resolution of each axis from 10 to 90 (maximum size for baseline) and recorded the relative errors and training run-times for each resolution. As expected, the relative errors of both models tend to decrease along the resolution. However, the errors of the baseline model starts to converge at about 40^3, while our model keeps finding a more accurate solution with larger points. More importantly, the training run-times of our model barely increase ($2.43 \rightarrow 2.95$) while that of the baseline model is very sensitive to the resolution ($2.13 \rightarrow 166.06$).

E.3 Klein-Gordon Equation

The Klein-Gordon equation is a non-linear hyperbolic PDE, which arises in diverse applied physics for modeling relativistic wave propagation. The inhomogeneous Klein-Gordon equation is given by

$$\partial_{tt} u - \Delta u + u^2 = f, \qquad x \in \Omega, t \in \Gamma \tag{16}$$

$$u(x, 0) = x_1 + x_2, \qquad x \in \Omega \tag{17}$$

$$u(x, t) = u_{bc}(x), \qquad x \in \partial\Omega, t \in \Gamma, \tag{18}$$

where we chose the spatial/temporal domain to be $\Omega \in [-1, 1]^2$ and $\Gamma \in [0, 10]$, respectively. For error measurement, We used a manufactured solution $u = (x_1 + x_2)\cos(t) + x_1 x_2 \sin(t)$ and f, u_{bc} are extracted from this exact solution. Fourth row of Table 1 shows the quantitative results. Again, our method finds the best solution in every resolution with much efficient memory usage ($40\times$) and training run-time ($57\times$). Note that this equation showed the largest difference in terms of run-time. Analogous to Helmholtz equation, Klein-Gordon equation contains the same number of second-order derivatives and thus, shows a similar aspect on training run-time. Visualized results are shown in Figure 5 (e) right.

References

[1] Rishabh Agarwal, Levi Melnick, Nicholas Frosst, Xuezhou Zhang, Ben Lengerich, Rich Caruana, and Geoffrey E Hinton. Neural additive models: Interpretable machine learning with neural nets. In *Advances in Neural Information Processing Systems*, volume 34, pages 4699–4711. Curran Associates, Inc., 2021.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[3] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision*, 2022.

[4] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[6] Ruofan Liang, Hongyi Sun, and Nandita Vijaykumar. Coordx: Accelerating implicit neural representation with a split MLP architecture. In *International Conference on Learning Representations*, 2022.

[7] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[8] Ben Moseley, Andrew Markham, and Tarje Nissen-Meyer. Finite basis physics-informed neural networks (fbpinns): a scalable domain decomposition approach for solving differential equations, 2021.

[9] Maziar Raissi, Paris Perdikaris, and George 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*, 378:686–707, 2019.

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

[11] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. Neural fields in visual computing and beyond. In *Computer Graphics Forum*, volume 41, pages 641–676. Wiley Online Library, 2022.