SHoP: A Deep Learning Framework for Solving High-Order Partial Differential Equations

Tingxiong Xiao[1], Runzhao Yang[1], Yuxiao Cheng[1], Jinli Suo[1,2]

[1]Department of Automation, Tsinghua University, Beijing 100084, China
[2]Institute for Brain and Cognitive Science, Tsinghua University, Beijing 100084, China
jlsuo@tsinghua.edu.cn

Abstract

Solving partial differential equations (PDEs) has been a fundamental problem in computational science and of wide applications for both scientific and engineering research. Due to its universal approximation property, neural network is widely used to approximate the solutions of PDEs. However, existing works are incapable of solving high-order PDEs due to insufficient calculation accuracy of higher-order derivatives, and the final network is a black box without explicit explanation. To address these issues, we propose a deep learning framework to solve high-order PDEs, named SHoP. Specifically, we derive the high-order derivative rule for neural network, to get the derivatives quickly and accurately; moreover, we expand the network into a Taylor series, providing an explicit solution for the PDEs. We conduct experimental validations four high-order PDEs with different dimensions, showing that we can solve high-order PDEs efficiently and accurately. The source code can be found at https://github.com/HarryPotterXTX/SHoP.git.

Introduction

Partial differential equations (PDEs) are used to describe the basic rules underlying complex processes in both scientific and engineering fields, and researchers have devoted lots of efforts to developing algorithms searching for their numerical solutions. Conventional finite difference methods become infeasible for high scale PDEs due to the difficulty in constructing mesh. With the rapid development of deep neural networks, ones began to utilize its universal approximating capability (Cybenko 1989; Hornik, Stinchcombe, and White 1989; Chen and Chen 1995; Leshno et al. 1993) to fit the solutions of PDEs, including its differential operator and constraints (the initial condition and boundary conditions), if any. Being a mesh-free approach, deep learning based solvers can circumvent the grand challenges in terms of memory and time when tackling high scale PDE.

The pioneering work utilizing neural networks to solve PDEs can date back to 1990s, when Dissanayake et al. proposed to use an MLP to find PDE's numerical solution (Dissanayake and Phan-Thien 1994) with easy implementation and high running efficiency, but with low proximity. With the rapid development in deep learning, PDE solvers based on deep neural networks are gaining momentum recently. PDE-FIND(Rudy et al. 2017) proposes a sparse regression method capable of discovering the governing partial differential equation(s) of a given system by measured time series in the spatial domain, and demonstrates its computation efficiency, robustness, and applicability on a variety of canonical problems spanning a number of scientific domains. Later, Deep Ritz Method(Yu et al. 2018) is designed to solve PDEs via approximating its analytical solution using a deep neural network, using the estimators in Ritz method to train the network and obtain the approximate solution. To solve high-dimensional PDEs, Sirignano et al. draw inspirations from Galerkin methods and proposed Deep Galerkin Method (DGM)(Sirignano and Spiliopoulos 2018), with the solution approximated by a neural network instead of a linear combination of basis functions. In 2019, Raissi et al. attempt to incorporate the principled physical constraints into learning of the deep neural networks, named Physics-informed neural networks (PINNs)(Raissi, Perdikaris, and Karniadakis 2019), targeting for reducing demanding training data, raising robustness and accelerate convergence. This approach can both search for data-driven solution and conduct data-driven discovery of partial differential equations, and fire up a series of working for further improvement. For example, DeepXDE (Lu et al. 2021b) improves its training efficiency using a new residual-based adaptive refinement (RAR) method and provides a Python library for PINNs as an educational and research tool; there are also some work(Wang, Teng, and Perdikaris 2021; Liu and Wang 2021; Xiang et al. 2021) studying the composite loss functions in the training process to accelerate the convergence or improve the final accuracy. Some other researchers (Meng et al. 2020; Moseley, Markham, and Nissen-Meyer 2021; Lyu et al. 2022) adapt the domain decomposition technology in traditional PDEs methods to realize parallel operations in time and space, or decomposes the order of derivatives to reduce the complexity and difficulty of single network learning. In the most recent years, Lu et al. design DeepONet(Lu et al. 2021a), a new network structure consisting of a branch net and a trunk net to encode the discrete input function space and output functions. Under this new architecture, one can learn various explicit operators, such as integrals and fractional Laplacians, as well as implicit operators that represent deterministic and stochastic

differential equations.

In spite of the striding progress, existing work is still at early stage and are faced with at least two challenges. Firstly, they are incapable of handling high-order PDEs, which is an important branch in PDE with wide applications. Most of above works adapt automatic differentiation module, like Autograd(Paszke et al. 2017) to calculate the derivatives under the current input. Autograd uses computation graph to record the intermediate process, and back-propagate to calculate the derivatives based on the existing computation graphs. However, as the order of derivative increase to a certain extent, an amount of calculation graphs need to be created and thus the calculation becomes intractable, in terms of both explosive growth of memory and inference time. On the other hand, as we all know, most neural networks are black box and the lacking interpretability hampers its practical applications, even with excellent performance. Designing algorithms with explicit explanations of the differentiation operators is key for pushing forward the deep-neural-network-based solvers towards real applications.

To solve the above two issues, we propose a deep learning framework to solve high-order PDEs, named SHoP, being able to solve high-order PDEs in explicit manner. Theoretically, it is proved that when the activation function is infinitely differentiable, the neural network is equivalent to its Taylor series, and when the network parameters meet certain distribution rules the Taylor series converges. Unlike the computation graph, our method gives an explicit formula for calculating the first n-order derivatives, which brings two-fold benefits. Firstly, after calculating the transformation matrix, we can get all the derivatives in just one step, more quickly and accurately than computation graph, and can greatly save memory resources. Secondly, once the network trained, we can expand the black-box network into an explicit expression of Taylor series if needed. We tested SHoP on four types of PDEs, and experimentally show that SHoP can solve the equation efficiently and accurately.

To summarize, the technical contributions are as follows:

- We propose the high-order derivative rule of neural network to calculate the derivatives quickly and accurately.
- We solve the high-order PDEs under the new derivative rule, via calculating the high-order derivatives in just one step, with higher accuracy, higher speed and less memory consumption than conventional computation graph.
- We propose to expand a neural network into Taylor series, providing an explicit explanation for the neural network fitting the PDE solution.
- We prove the equivalence between a neural network and its Taylor series, and analyze its convergence condition.

High-Order Derivatives of Neural Network for Solving PDEs

As known, we can describe the underlying solution of a PDE with a deep neural network and optimize the network parameters in a data driven manner. Mathematically, the key module of the solver is to calculate the derivatives. A neural network can be viewed as a composite function. In the following, we will present the high-order derivative rule for composite functions and the extension to deep neural networks, which allows for efficient and accurate computation of their high-order derivatives.

High-Order Derivatives of Composite Function

Considering a composite function $f(g(x))$, with $g(x)$ and $f(z)$ being n-order differentiable at $x0$ and $z0 = g(x0)$ respectively. $\frac{\partial^k g}{\partial x^k}|_{x0}$ and $\frac{\partial^k f}{\partial g^k}|_{z0}$ are the k-order derivative of $g(x)$ at $x0$ and of $f(z)$ at $z0$. According to the chain rule, we can calculate the first three terms of $f(g(x))$'s n-order derivatives as

$$\begin{cases} \frac{\partial f}{\partial x} = \frac{\partial g}{\partial x}\frac{\partial f}{\partial g}, \\ \frac{\partial^2 f}{\partial x^2} = \frac{\partial^2 g}{\partial x^2}\frac{\partial f}{\partial g} + (\frac{\partial g}{\partial x})^2\frac{\partial^2 f}{\partial g^2}, \\ \frac{\partial^3 f}{\partial x^3} = \frac{\partial^3 g}{\partial x^3}\frac{\partial f}{\partial g} + 3\frac{\partial g}{\partial x}\frac{\partial^2 g}{\partial x^2}\frac{\partial^2 f}{\partial g^2} + (\frac{\partial g}{\partial x})^3\frac{\partial^3 f}{\partial g^3}. \end{cases} \tag{1}$$

For more terms, we convert $\frac{\partial}{\partial x}\frac{\partial^i f}{\partial g^i}$ to $\frac{\partial g}{\partial x}\frac{\partial^{i+1} f}{\partial g^{i+1}}$, and $\frac{\partial^n f}{\partial x^n}$ can be calculated given $\{\frac{\partial^i f}{\partial g^i}, i = 1, \ldots, n\}$ and $\{\frac{\partial^i g}{\partial x^i}, i = 1, \ldots, n\}$. Then Eq. (1) turns into following matrix form

$$\begin{bmatrix} \frac{\partial f}{\partial x} \\ \vdots \\ \frac{\partial^n f}{\partial x^n} \end{bmatrix} = \begin{bmatrix} \frac{\partial g}{\partial x} & 0 & 0 & 0 \\ \frac{\partial^2 g}{\partial x^2} & (\frac{\partial g}{\partial x})^2 & 0 & 0 \\ \frac{\partial^3 g}{\partial x^3} & 3\frac{\partial g}{\partial x}\frac{\partial^2 g}{\partial x^2} & (\frac{\partial g}{\partial x})^3 & 0 \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \frac{\partial f}{\partial g} \\ \vdots \\ \frac{\partial^n f}{\partial g^n} \end{bmatrix}, \tag{2}$$

which can be further abbreviated as

$$\mathbf{v}^{f,x} = \mathbf{M}^{g,x}\mathbf{v}^{f,g}. \tag{3}$$

In this equation $\mathbf{v}^{f,x} \in \mathbb{R}^n$ and $\mathbf{v}^{f,g} \in \mathbb{R}^n$ are respectively the vectors composed of partial derivatives $\{\frac{\partial^i f}{\partial x^i}\}$ and $\{\frac{\partial^i f}{\partial g^i}\}$; $\mathbf{M}^{g,x} \in \mathbb{R}^{n\times n}$ is a transformation matrix composed of $\frac{\partial^i g}{\partial x^i}$ and takes a lower triangular form. So far, the calculation of $f(g(x))$'s n-order derivatives turns into the computation of $\mathbf{M}^{g,x} \in \mathbb{R}^{n\times n}$.

From Eq. (2), the ith and $i+1$th terms ($i < n$) are respectively

$$\frac{\partial^i f}{\partial x^i} = \sum_{j=1}^{n} \mathbf{M}^{g,x}_{i,j}\frac{\partial^j f}{\partial g^j}, \tag{4}$$

and

$$\frac{\partial^{i+1} f}{\partial x^{i+1}} = \sum_{j=1}^{n} \mathbf{M}^{g,x}_{i+1,j}\frac{\partial^j f}{\partial g^j}. \tag{5}$$

Taking derivatives over both sides of Eq. (4) we can get

$$\begin{aligned} \frac{\partial^{i+1} f}{\partial x^{i+1}} &= \frac{\partial}{\partial x}\frac{\partial^i f}{\partial x^i} = \sum_{j=1}^{n}\frac{\partial}{\partial x}(\mathbf{M}^{g,x}_{i,j}\frac{\partial^j f}{\partial g^j}) \\ &= \sum_{j=1}^{n}\frac{\partial \mathbf{M}^{g,x}_{i,j}}{\partial x}\frac{\partial^j f}{\partial g^j} + \sum_{j=1}^{n}\frac{\partial g}{\partial x}\mathbf{M}^{g,x}_{i,j}\frac{\partial^{j+1} f}{\partial g^{j+1}} \\ &= \sum_{j=1}^{n}\left(\frac{\partial \mathbf{M}^{g,x}_{i,j}}{\partial x} + \frac{\partial g}{\partial x}\mathbf{M}^{g,x}_{i,j-1}\right)\frac{\partial^j f}{\partial g^j} \\ &\quad - \frac{\partial g}{\partial x}\mathbf{M}^{g,x}_{i,0}\frac{\partial f}{\partial g} + \frac{\partial g}{\partial x}\mathbf{M}^{g,x}_{i,n}\frac{\partial^{n+1} f}{\partial g^{n+1}}. \end{aligned} \tag{6}$$

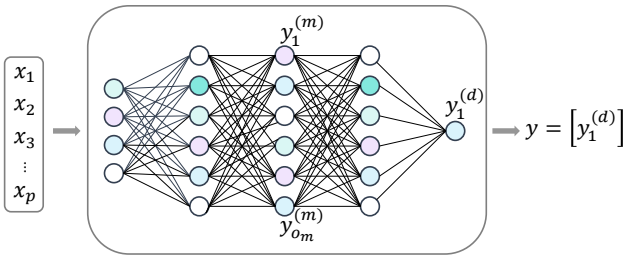

Figure 1: The structure of a single-output multilayer perceptron with d layers and o_m nodes in the mth layer.

Because $\mathbf{M}_{i,0}^{g,x} = 0$ and $\mathbf{M}_{i,n}^{g,x} = 0$ $(i < n)$, Eq. (6) can be simplified into

$$\frac{\partial^{i+1} f}{\partial x^{i+1}} = \sum_{j=1}^{n} \left(\frac{\partial \mathbf{M}_{i,j}^{g,x}}{\partial x} + \frac{\partial g}{\partial x} \mathbf{M}_{i,j-1}^{g,x} \right) \frac{\partial^j f}{\partial g^j}. \tag{7}$$

Compare Eq. (5) and Eq. (7), we can get the recurrence formula of $\mathbf{M}^{g,x}$ as

$$\begin{cases} \mathbf{M}_{1,1}^{g,x} = \frac{\partial g}{\partial x}, \\ \mathbf{M}_{i,j}^{g,x} = 0, & i < j \\ \mathbf{M}_{i+1,j}^{g,x} = \frac{\partial \mathbf{M}_{i,j}^{g,x}}{\partial x} + \frac{\partial g}{\partial x} \mathbf{M}_{i,j-1}^{g,x}, & i \geq j \end{cases} \tag{8}$$

which explicitly composes the n-order chain transformation matrix $\mathbf{M}^{g,x}$ in Eq. (3). The detailed derivation process of Eq. (8) can be found in Supplementary Materials.

High-Order Derivatives of Neural Network

Without loss of generality, we take Multilayer Perceptron (MLP) as an example, with the network structure illustrated in Fig. 1. Denoting the input as $\mathbf{x} = [x_1 \dots x_p]^T \in \mathbb{R}^p$, the network depth as d, the width of mth layer as o_m, the output of the i-th node in mth layer as $y_i^{(m)}$, the linear weighted result of input of the i-th node in mth layer as $z_i^{(m)}$, and the final output as $\mathbf{y} = \mathbf{y}^{(d)} = \left[y_1^{(d)}\right] \in \mathbb{R}$, this article aims to calculate $\mathbf{y}$'s n-order derivatives with respect to input $\mathbf{x}$. The input-output relationship of the MLP can be described explicitly, with the first layer being

$$y_i^{(1)} = \sigma(z_i^{(1)}) = \sigma(\sum_{j=1}^{p} \mathbf{W}_{i,j}^{(1)} x_j + \mathbf{b}_i^{(1)}), \tag{9}$$

and the successive layers as

$$y_i^{(m+1)} = \sigma(z_i^{(m+1)}) = \sigma(\sum_{j=1}^{o_m} \mathbf{W}_{i,j}^{(m+1)} y_j^{(m)} + \mathbf{b}_i^{(m+1)}). \tag{10}$$

Then the final output is defined as

$$\mathbf{y} = \mathbf{y}^{(d)} = \left[y_1^{(d)}\right], \tag{11}$$

in which $\mathbf{W}^{(m+1)} \in \mathbb{R}^{o_{m+1} \times o_m}$ is the weight matrix of layer $m+1$, $\mathbf{b}^{(m+1)} \in \mathbb{R}^{o_{m+1}}$ is the bias vector, $\sigma(*)$ is the nonlinear activation function. $\mathbf{y}^{(m)} = \left[y_1^{(m)}, \dots, y_{o_m}^{(m)}\right]^T \in \mathbb{R}^{o_m}$ is the output vector of mth layer.

With above denotations, we induce $\mathbf{y}$'s derivatives with respect to the input $\mathbf{x}$. First, from Eq. (11) and the definition in Eq. 3, we can get a initial partial derivative vector

$$\begin{aligned} \mathbf{v}^{\mathbf{y}, y_1^{(d)}} &= \left[\begin{array}{cccc} \frac{\partial \mathbf{y}}{\partial y_1^{(d)}} & \frac{\partial^2 \mathbf{y}}{\partial {y_1^{(d)}}^2} & \cdots & \frac{\partial^n \mathbf{y}}{\partial {y_1^{(d)}}^n} \end{array} \right]^T \\ &= \left[\begin{array}{cccc} 1 & 0 & \dots & 0 \end{array} \right]^T, \end{aligned} \tag{12}$$

Taking derivatives over both sides of Eq. (10) arrives at

$$\frac{\partial^k y_i^{(m+1)}}{\partial {y_j^{(m)}}^k} = {\mathbf{W}_{i,j}^{(m+1)}}^k \frac{\partial^k \sigma(z_i^{(m+1)})}{\partial {z_i^{(m+1)}}^k}, \tag{13}$$

which form the basic elements of matrix $\mathbf{M}^{y_i^{(m+1)}, y_j^{(m)}}$ defined in Eq. (3).

Given $\{\frac{\partial^k \mathbf{y}}{\partial {y_i^{(m+1)}}^k}\}$ and $\{\frac{\partial^k y_i^{(m+1)}}{\partial {y_j^{(m)}}^k}\}$, according to Eqns. (2)(3)(8), we can calculate $\frac{\partial^k \mathbf{y}}{\partial {y_j^{(m)}}^k}$ as

$$\mathbf{v}^{\mathbf{y}, y_j^{(m)}} = \sum_{i=1}^{o_{m+1}} \mathbf{M}^{y_i^{(m+1)}, y_j^{(m)}} \mathbf{v}^{\mathbf{y}, y_i^{(m+1)}}. \tag{14}$$

By analogy, we can get all the unmixed partial derivatives $\{\frac{\partial^k \mathbf{y}}{\partial {x_j}^k}, j = 1, \dots, p\}$ by calculating $\mathbf{v}^{\mathbf{y}, x_j}$.

For the mixed partial derivatives, we can get $\{\frac{\partial^k \mathbf{y}}{\partial {y_q^{(1)}}^k}, q = 1, \cdots, o_1\}$ and calculate them. For example,

$$\begin{aligned} \frac{\partial^2 \mathbf{y}}{\partial x_{i_1} \partial x_{i_2}} &= \sum_{q=1}^{o_1} \frac{\partial}{\partial x_{i_2}} \left(\frac{\partial y_q^{(1)}}{\partial x_{i_1}} \frac{\partial \mathbf{y}}{\partial y_q^{(1)}} \right) \\ &= \sum_{q=1}^{o_1} \left(\frac{\partial^2 y_q^{(1)}}{\partial x_{i_1} \partial x_{i_2}} \frac{\partial \mathbf{y}}{\partial y_q^{(1)}} + \frac{\partial y_q^{(1)}}{\partial x_{i_1}} \frac{\partial y_q^{(1)}}{\partial x_{i_2}} \frac{\partial^2 \mathbf{y}}{\partial {y_q^{(1)}}^2} \right) \\ &= \sum_{q=1}^{o_1} \mathbf{W}_{q,i_1}^{(1)} \mathbf{W}_{q,i_2}^{(1)} \left(\frac{\partial^2 \sigma(z_q^{(1)})}{\partial {z_q^{(1)}}^2} \frac{\partial \mathbf{y}}{\partial y_q^{(1)}} + (\frac{\partial \sigma(z_q^{(1)})}{\partial z_q^{(1)}})^2 \frac{\partial^2 \mathbf{y}}{\partial {y_q^{(1)}}^2} \right). \end{aligned} \tag{15}$$

Therefore, during the back-propagation of the neural network, we do not need to calculate the mixed partial derivatives like $\frac{\partial^k \mathbf{y}}{\partial {y_i^{(m)}}^{k-1} \partial y_j^{(m)}}$, but only need to calculate $\frac{\partial^k \mathbf{y}}{\partial {y_i^{(m)}}^k}$ instead. Further according to the chain rule, we can get all the mixed partial derivatives from $\frac{\partial^k \mathbf{y}}{\partial {y_q^{(1)}}^k}$ in one time.

For faster calculation of the partial derivatives, we convert the above formulas into matrix form, and the detailed formulas can be find in Supplementary Materials.

SHoP: A Deep Learning Framework for Solving High-Order PDEs

After calculating the partial derivatives, we can solve a PDE via designing a sampler and constructing a loss function.

The Working Flow of SHoP

Considering a PDE with p dimensions

$$\begin{cases} \mathcal{L}u(\mathbf{x}) = 0, & \mathbf{x} \in \Omega \\ u(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega \end{cases} \tag{16}$$

where $\mathbf{x} \in \Omega \subset \mathbb{R}^p$, $\partial\Omega$ is Ω's boundary, $\mathcal{L}u(\mathbf{x})$ is a combination of derivatives of $u(\mathbf{x})$ with respect to $\mathbf{x}$. We use a neural network $f(\mathbf{x};\theta)$ to approximate $u(\mathbf{x})$ with θ being the network parameters.

In terms of the sampler, before network training, we establish discrete coordinates according to Ω and $\partial\Omega$, and randomly sample the coordinate points according to the preset batch size during training.

The objective function is defined as

$$J(\mathbf{x};\theta) = \lambda\|\mathcal{L}f(\mathbf{x};\theta)\|_{\Omega}^2 + \mu\|f(\mathbf{x};\theta) - g(\mathbf{x})\|_{\partial\Omega}^2. \tag{17}$$

Here the two terms are l_2 norm fitting the partial derivatives over the defining field Ω and along the boundary $\partial\Omega$ respectively, λ and μ are hyper-parameters balancing two terms in the loss function.

In each training epoch, after applying forward propagation on the neural network, we can get $\mathbf{v}^{\mathbf{y},\mathbf{y}^{(1)}}\{\frac{\partial^k \mathbf{y}}{\partial y_q^{(1)\,k}}, q = 1,\cdots,o_1, k = 1,\cdots,n\}$ with Eqns. (12)(14), and calculate the derivatives we want with Eq. (15).

Explicit Expression of the PDE Solution

With the first n-order derivatives, we can get an explicit Taylor series to approximate the original network locally. The n-order Taylor series can be calculated as

$$\begin{aligned} f(\mathbf{x}) = f(\mathbf{x}0;\theta) + \sum_{i=1}^{p} \frac{\frac{\partial f(\mathbf{x};\theta)}{\partial \mathbf{x}_i}|_{\mathbf{x}0}}{1!}\Delta\mathbf{x}_i + \dots \\ + \sum_{i_1,\dots,i_n=1}^{p} \frac{\frac{\partial^n f(\mathbf{x};\theta)}{\partial \mathbf{x}_{i_1}\dots\partial \mathbf{x}_{i_n}}|_{\mathbf{x}0}}{n!}\Delta\mathbf{x}_{i_1}\dots\Delta\mathbf{x}_{i_n}, \end{aligned} \tag{18}$$

where $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}0$, $\frac{\partial^k f(\mathbf{x};\theta)}{\partial \mathbf{x}_{i_1}\dots\partial \mathbf{x}_{i_k}}|_{\mathbf{x}0}$ is a k-order partial derivative on the reference point $\mathbf{x}0$.

Although approximating a PDE's solution with a deep neural network is of high efficiency and accuracy, as we all know, such a black-box model lacks interpretability and hampers its practical applications. After Taylor expansion, one can retrieve the Taylor series out of the black-box explicitly, which can provide us a deeper understanding of the mapping mechanism of the learned neural network. Here we give two potential studies benefiting from such explicit expansion: (i) In a complex process with multiple input, after expanding the governing neural network into Taylor series, one can quantify the contribution of each input. Such explicit description might inspire researchers to analyze the underlying causation mechanism of the target output. (ii) We can also bridge the network parameters (mapped to the weights of Taylor series) and the domain expertise, and thus measure the reliability of the deep neural network interpretably and set proper confidence level to the network output. In other words, our expansion facilitates studying the fidelity of the neural network in a more explainable way and advancing its real applications.

Analysis of the Taylor Polynomial Convergence

When the activation function is infinitely differentiable, we can calculate all the derivatives and thus the neural network is equivalent to its Taylor series. From Eqns. (13)(15), the k-order derivatives are related to $\mathbf{W}^{(m)}_{i_1,j_1}\mathbf{W}^{(m)}_{i_2,j_2}\dots\mathbf{W}^{(m)}_{i_k,j_k}$, which is the continuous multiplication of k weights in $\mathbf{W}^{(m)}$.

$$|\frac{\partial^k \mathbf{y}}{\partial \mathbf{x}^k}| \propto |\mathbf{W}^{(m)}_{i,j}|^k. \tag{19}$$

When the parameters in $\mathbf{W}^{(m)}$ is concentrated near 0, higher-order derivatives are more likely to approach 0. When the parameters is located far from 0, higher-order derivatives may become increasingly larger due to continuous addition and multiplication, and thus the Taylor series diverge, i.e., we cannot obtain the Taylor approximate solution.

$$\lim_{\substack{|\mathbf{W}^{(m)}_{i,j}|\to 0 \\ k\to\infty}} |\frac{\partial^k \mathbf{y}}{\partial \mathbf{x}^k}| = 0, \quad \lim_{\substack{|\mathbf{W}^{(m)}_{i,j}|>1 \\ k\to\infty}} |\frac{\partial^k \mathbf{y}}{\partial \mathbf{x}^k}| = +\infty. \tag{20}$$

The above analysis tells that the parameter distribution of each layer has a great influence on the convergence of Taylor expansion. The above rules help imposing constraints on the network parameters during the network training, and can also help designing network structures with high-order Taylor approximation. More theory details can be found in Supplementary Materials.

Time Complexity Analysis of SHoP

The core algorithm of deep learning is back-propagation, and most of the deep learning frameworks adapt automatic differentiation module. Here, we analyze and compare the time complexity of computational-graph-based method and SHoP for a p-D neural network.

(i) computational-graph-based method calculates derivatives based on computational graphs whose length increase exponentially at base 2. There are p^k k-order derivatives, and the length of their computational graphs is 2^{k-1}. The time complexity is

$$T(n) = \sum_{k=1}^{n} p^k 2^{k-1} \sim \mathcal{O}((2p)^n). \tag{21}$$

(ii) SHoP obtains all the derivatives at one time, with the main calculations lie in calculating the transformation matrix $\mathbf{M}$ and conducting back propagation. $\mathbf{M}$ is a lower triangular matrix and the block matrices in k-th row need k operations, so the complexity of calculating $\mathbf{M}$ is $T(n) = \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \sim \mathcal{O}(n^3)$. For linear layers, $\mathbf{M}$ turns into a diagonal matrix and the complexity reduces to $T(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \sim \mathcal{O}(n^2)$. For mixed partial derivatives, $\mathbf{M}$ is a diagonal matrix and the size of $\mathbf{Q}_k$ is p^{k-1} times larger than $\mathbf{W}$, the complexity is about $T(n) = \sum_{k=1}^{n} p^{k-1} = \frac{1-p^n}{1-p} \sim \mathcal{O}(p^n)$. Therefore, the complexity of SHoP is

$$\mathcal{O}(n^2) < T(n) < max(\mathcal{O}(n^3), \mathcal{O}(p^n)). \tag{22}$$

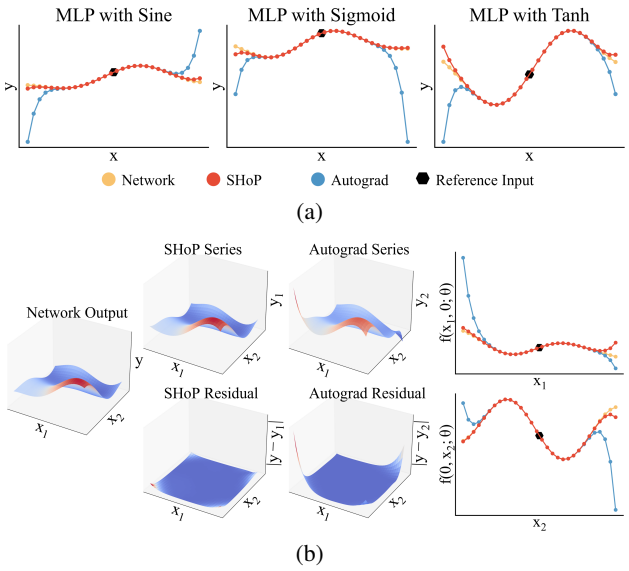

Figure 2: Approximation results of networks' Taylor expansion. (a) The approximation results on three different 1D MLPs. (b) The performance on a 2D MLP.

Experiments

Implementation Details

We use MLPs with Sine activation function(Sitzmann et al. 2020) to validate our Taylor expansion of neural networks and its capability of solving high-order PDEs. Since the expansion of the multi-output model is a direct extension of the single-output model, here we use single-output setting for easier demonstration.

We use Adamax(Kingma and Ba 2014) to optimize the parameters and conduct 1000 epochs of model optimization in most cases. The learning rate is initialized to be 5e-3 and MultiStepLR is adopted to schedule the learning rate progressively. The SHoP framework is implemented with Pytorch, and the GPU version is NVIDIA GeForce RTX 3090 on a Linux operation system. For more implementation details, please refer to the Supplementary Materials.

Performance of the High-Order Derivative Rule

In this section, we test the accuracy and efficiency of the new high-order derivative rule, and compare it with Autograd.

Accuracy of Derivatives. Fig. 2(a) shows the approximation results of SHoP and Autograd on three 1D MLPs with different activation functions. We use above two approaches to calculate the first 10-order derivatives at the reference points and use them to approximate the target MLPs. The plots show that our results (red curves) are closer to the true output (yellow curves), while Autograd (blue curves) fits well near the reference point but deviates a lot as the input moves far from the reference point. We can induce that although Autograd can calculate low-order derivatives well but is of insufficient accuracy when dealing with high-order derivatives. On the contrary, SHoP conducts one-step inference to avoid error accumulation and thus achieves high accuracy even at high-orders. Fig. 2(b) compares the approximated surfaces (middle, upper) and residues (middle, lower) of SHoP and Autograd on a 2D MLP with output shown in the left panel, and shows their results along x_1=0 and x_2=0 (right). Both plots display our superior performance and arrive at the same conclusion as in Fig. 2(a).

Running Efficiency. Tab. 1 shows the running time of our approach in parallel with that of Autograd for calculating the first n order derivatives of a p-input MLP. With the increase of input dimension and order, the running time of both methods increase but our running time is consistently shorter than Autograd by a large margin. When p=2 and n=10, we just need 0.3828s, while Autograd takes 1435.0s. Besides, when p=3 and n=8, Autograd runs out of memory because it need to create too many computation graphs, while we finish it in just 0.2618s, which prove the higher time and memory efficiency of our method.

Series Convergence. In Tab. 2, we initialized the weights of each layer following uniform distribution $\mathbf{W}_{i,j}^{(m)} \sim U(-\frac{w_0}{o_{m-1}}, \frac{w_0}{o_{m-1}})$. When $w_0 = 0.01,\ 0.1$, the higher-order derivatives are far smaller than the lower-order derivatives, and we can ignore the higher-order derivatives and the Taylor series converges. When $w_0 = 1.0$, the derivatives of different orders oscillate, and the higher-order terms cannot be ignored. When $w_0 = 10,\ 100$, Taylor series are seriously divergent. The results inspire us to impose proper constraints on the network parameters when using its Taylor series as a surrogate for either calculation or analysis. We can also induce that the neural networks' strong capability of fitting diverse functions is due to its wide Taylor series covering all convergence cases.

Effectiveness of Solving PDEs

Here we demonstrate the performance of our method on four different types of PDEs: a 1D 4th-order Harmonic oscillator system, a 2D 4th-order Biharmonic equation, a 2D 8th-order Helmholtz equation, and a 3D 4th-order Heat equation. We compared our method with PINN, and expand the black-box results to four explicit Taylor polynomials. The coefficient of determination R^2 is used as the evaluation metric for the quality of PDE solutions. The detailed PDE conditions, initial conditions, and boundary conditions can be found in Supplementary Materials.

1D Function. We consider a 1D 4th-order Harmonic oscillator system

$$\begin{cases} u_{tttt} + 2u_{tt} + u = 0, & t \in [0, 2\pi], \\ u(0) = 0, u_t(0) = 1, u_{tt}(0) = 0. \end{cases} \tag{23}$$

The initial conditions indicate that the initial position of the harmonic oscillator is the balance point, the initial speed is 1, and the initial acceleration is 0. The hyper-parameters $\lambda = 5$, $\mu = 1$, and the results are shown in Fig. 3.

The results show that SHoP's solution $u_2(t)$ is closer to the true solution $u(t)$, and we expand the final black-box

n		2	3	4	5	6	7	8	9	10
p=1	SHoP	0.0303	0.0397	0.0526	0.0619	0.0935	0.1343	0.1801	0.2319	0.3480
	Autograd	0.0450	0.0474	0.0556	0.0706	0.1092	0.2151	0.4524	1.3157	4.1036
p=2	SHoP	0.0321	0.0408	0.0538	0.0715	0.1043	0.1356	0.1922	0.2982	0.3828
	Autograd	0.0485	0.0532	0.0756	0.1958	0.8930	5.1107	32.028	204.34	1435.0
p=3	SHoP	0.0322	0.0418	0.0567	0.0720	0.1080	0.1623	0.2618	0.5235	1.2327
	Autograd	0.0514	0.1102	0.4226	2.6690	24.417	236.75	OOM	OOM	OOM

Table 1: Comparison of the running time between SHoP and Autograd. Here p denotes the dimension of input, and n is the order of derivatives, and "OOM" means out of memory.

w_0	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
0.010	6.16e-03	4.77e-05	3.80e-07	2.51e-09	2.57e-11	1.44e-13	1.81e-15	8.79e-18	1.32e-19
0.100	3.74e-02	6.87e-03	1.88e-04	5.60e-05	1.93e-06	4.79e-07	1.97e-08	4.22e-09	1.91e-10
1.000	0.67e+00	0.36e+00	0.37e+00	0.28e+00	0.25e+00	0.48e+00	0.06e+00	1.22e+00	0.62e+00
10.00	4.58e+01	1.02e+02	7.31e+03	3.68e+04	1.72e+06	1.69e+07	5.20e+08	1.05e+10	1.96e+11
100.0	6.29e+03	3.90e+05	7.86e+08	2.54e+11	2.97e+14	2.93e+17	1.57e+20	4.96e+23	2.25e+26

Table 2: Convergence of Taylor series for a three-layer MLP under various parameter distributions. Here w_0 determines the distribution of network parameters $U(-\frac{w_0}{o_{m-1}}, \frac{w_0}{o_{m-1}})$, and the scores in each cell is $|\frac{\partial^n f}{\partial x^n} / \frac{\partial f}{\partial x}|$, indicating the convergence.

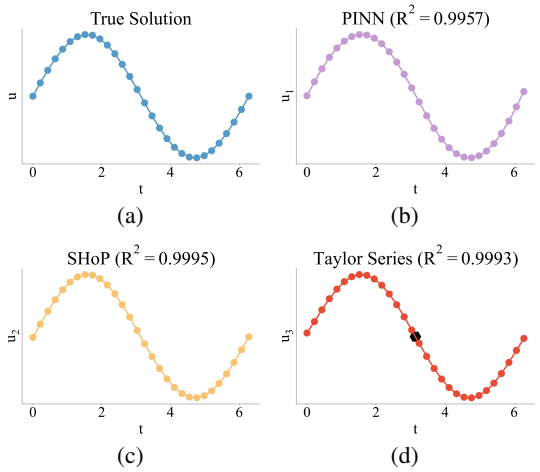

Figure 3: Performance on 1D 4th-order Harmonic oscillator system. (a) The true solution u in interval [0, 2π]. (b) PINN's solution u_1. (c) SHoP's solution u_2. (d) The Taylor polynomial u_3 for the network on the reference input $t = \pi$.

network into a 7-order Taylor polynomial $u_3(t)$, providing an explicit explanation for the final solution. Given the network's output at point π and its corresponding first 7-order derivatives, the Taylor polynomial can be described as

$$\begin{aligned} u_3(t) &= 0.0059 - \frac{0.9943}{1!}\Delta_\pi + \frac{0.0097}{2!}\Delta_\pi^2 + \frac{0.9986}{3!}\Delta_\pi^3 \\ &- \frac{0.0235}{4!}\Delta_\pi^4 - \frac{1.0089}{5!}\Delta_\pi^5 + \frac{0.0680}{6!}\Delta_\pi^6 + \frac{1.0894}{7!}\Delta_\pi^7 \\ &\approx -\frac{1}{1!}\Delta_\pi + \frac{1}{3!}\Delta_\pi^3 - \frac{1}{5!}\Delta_\pi^5 + \frac{1}{7!}\Delta_\pi^7 \end{aligned} \tag{24}$$

where $\Delta_\pi = t - \pi$. One can easily see that $u_3(t)$ match well with the Taylor series of the true solution $u = sin(t)$, which again validates the accuracy of SHoP's solution.

2D Function. We solve a 2D fourth-order PDE and a 2D eighth-order PDE using SHoP. The first one is a Biharmonic equation defined over $(x_1, x_2) \in [0, \pi]^2$, with PDE condition

$$\nabla^4 u = 4sin(x_1 + x_2), \tag{25}$$

where ∇^4 is the fourth power of the del operator and the square of the Laplacian operator ∇^2 (or Δ). Fig. 4 shows the performance of PINN and SHoP on this PDE, with SHoP's solution closer to the ground truth version. We also expand the network into a 2D 10-order Taylor polynomial on an reference input $(x_1, x_2) = (0.5\pi,\ 0.5\pi)$, and plot its output in Fig. 4(d). The plot shows that the Taylor polynomial can actually provide a good approximate expression, making this neural network more transparent and interpretable.

The second one is a Helmholtz equation defined over $(x_1, x_2) \in [0, 1]^2$, and its PDE condition is

$$\Delta^4 u + u = 17e^{-x_1 - x_2}. \tag{26}$$

The results are shown in Fig. 5. In line with the conclusion for the Biharmonic equation, SHoP has been shown to be an effective tool for obtaining approximate solutions to PDEs, and it only takes 834 seconds with SHoP, while PINN required 1.18e5 seconds.

3D Function. Further, we use SHoP to solve the 4th-order PDE of a heat equation defined as

$$u_t - \nabla^4 u = \pi^2 sin(\pi x_1) sin(\pi x_2)(cos(\pi t) - 4\pi^2 sin(\pi t)), \tag{27}$$

where ∇^4 is the square of the Laplacian operator w.r.t. x_1 and x_2. Fig. 6(a) is the true PDE solution, and Fig. 6(b)(c) show the solutions of PINN and SHoP. The coefficient of determination of SHoP is 0.9998, whereas that of PINN is 0.9962, indicating that SHoP achieves higher accuracy.

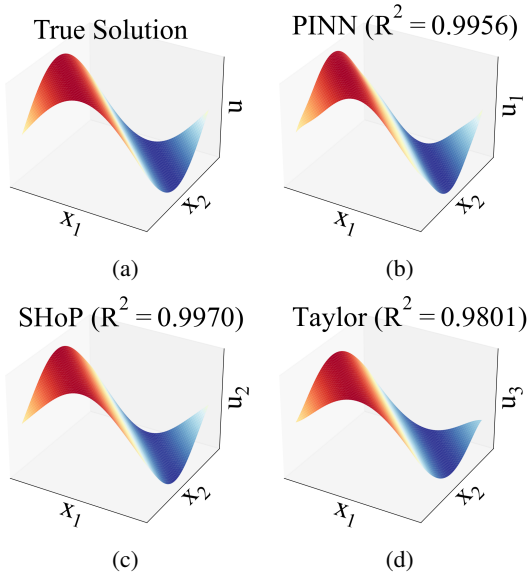

Figure 4: Performance on 2D 4th-order Biharmonic equation. (a) The true solution u in interval $[0, \pi] \times [0, \pi]$. (b) PINN's solution u_1. (c) SHoP's solution u_2. (d) Taylor polynomial u_3 on $(x_1, x_2) = (0.5\pi, 0.5\pi)$.

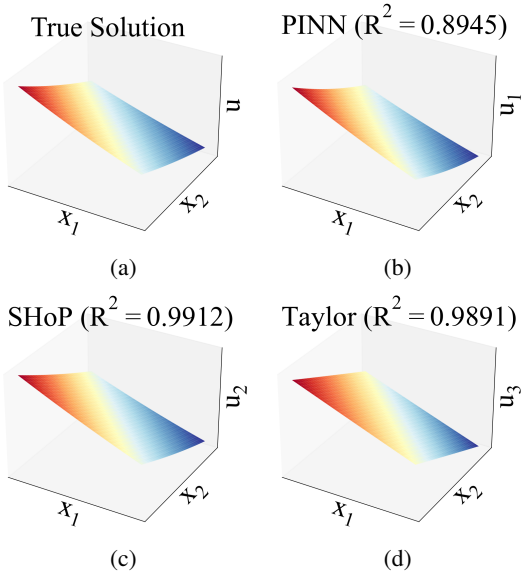

Figure 5: Performance on 2D 8th-order Helmholtz equation. (a) The true solution u in interval $[\frac{1}{3}, \frac{2}{3}] \times [\frac{1}{3}, \frac{2}{3}]$. (b) PINN's solution u_1. (c) SHoP's solution u_2. (d) Taylor polynomial u_3 on $(x_1, x_2) = (0.5, 0.5)$.

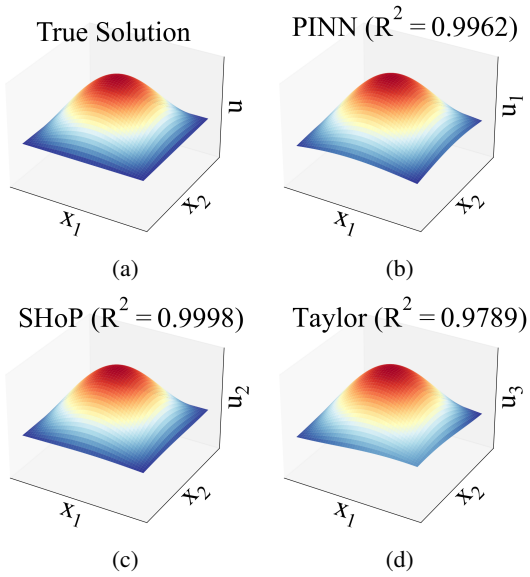

Figure 6: Performance on 3D 4th-order Heat equation. (a) The true solution u in interval $x_1, x_2 \in [0, 1] \times [0, 1]$ at $t = 0.5$. (b) PINN's solution u_1. (c) SHoP's solution u_2. (d) Taylor polynomial u_3 on $(t, x_1, x_2) = (0.5, 0.5, 0.5)$.

Conclusions

Aiming at solving high-order PDEs effectively, we derive the high-order derivative rule of neural network for quick and accurate derivative calculation, adopt it to develop a neural-network-based PDE solver, and expand the final black-box neural network into an explicit Taylor polynomial. The convergence condition of the Taylor series is analyzed experimentally validated as well.

SHoP has built a simple and general framework to obtain the approximate solution of PDEs quickly. Comprehensive experiments are conducted to verify the high approximation accuracy of Taylor series to the target neural network, and the high efficiency in calculating partial derivatives. We also validate the high performance of SHoP on multiple high-order PDEs, from 1D to 3D. Moreover, SHoP provides an interpretable understanding of the learned black-box neural network, and can also be potentially used to specify the function parameters if given the form of the latent PDE solution.

Limitations. The derivation of the high-order derivative rule is based on the assumption that all components are differentiable. For networks including components such as ReLU, or LeakyReLU, both SHoP and computational-graph-based method can only obtain their first-order derivative. This article focuses on deriving the most basic MLP. It is worth noting that similar high-order derivative formulas can be derived for more complex architectures such as convolutional neural networks, recurrent neural networks, or transformer with the ideas presented in this paper.

Future Work. In the future, in addition to raising the accuracy further, we would like to apply SHoP to some different topics, e.g., explaining the working mechanism of neural networks, developing high-order optimization algorithms. We can get the derivatives between any nodes of a network, which might inspire lightweight network design. Interpreting and simplifying a network describing the physical field or industrial controller can also be considered.

Acknowledgments

This work is jointly funded by Ministry of Science and Technology of China (Grant No. 2020AAA0108202), National Natural Science Foundation of China (Grant No. 61931012) and Beijing Municipal Natural Science Foundation (Grant No. Z200021).

References

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

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4): 303–314.

Dissanayake, M.; and Phan-Thien, N. 1994. Neural-network-based approximations for solving partial differential equations. *communications in Numerical Methods in Engineering*, 10(3): 195–201.

Hornik, K.; Stinchcombe, M.; and White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5): 359–366.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Leshno, M.; Lin, V. Y.; Pinkus, A.; and Schocken, S. 1993. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6): 861–867.

Liu, D.; and Wang, Y. 2021. A Dual-Dimer method for training physics-constrained neural networks with minimax architecture. *Neural Networks*, 136: 112–125.

Lu, L.; Jin, P.; Pang, G.; Zhang, Z.; and Karniadakis, G. E. 2021a. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3): 218–229.

Lu, L.; Meng, X.; Mao, Z.; and Karniadakis, G. E. 2021b. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1): 208–228.

Lyu, L.; Zhang, Z.; Chen, M.; and Chen, J. 2022. MIM: A deep mixed residual method for solving high-order partial differential equations. *Journal of Computational Physics*, 452: 110930.

Meng, X.; Li, Z.; Zhang, D.; and Karniadakis, G. E. 2020. PPINN: Parareal physics-informed neural network for time-dependent PDEs. *Computer Methods in Applied Mechanics and Engineering*, 370: 113250.

Moseley, B.; Markham, A.; and Nissen-Meyer, T. 2021. Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations. *arXiv preprint arXiv:2107.07871*.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in pytorch. *Advances in Neural Information Processing Systems*.

Raissi, M.; Perdikaris, P.; and Karniadakis, G. E. 2019. 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.

Rudy, S. H.; Brunton, S. L.; Proctor, J. L.; and Kutz, J. N. 2017. Data-driven discovery of partial differential equations. *Science Advances*, 3(4): e1602614.

Sirignano, J.; and Spiliopoulos, K. 2018. DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375: 1339–1364.

Sitzmann, V.; Martel, J.; Bergman, A.; Lindell, D.; and Wetzstein, G. 2020. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33: 7462–7473.

Wang, S.; Teng, Y.; and Perdikaris, P. 2021. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5): A3055–A3081.

Xiang, Z.; Peng, W.; Zheng, X.; Zhao, X.; and Yao, W. 2021. Self-adaptive loss balanced physics-informed neural networks for the incompressible navier-stokes equations. *arXiv preprint arXiv:2104.06217*.

Yu, B.; et al. 2018. The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1): 1–12.