Numer. Math. Theor. Meth. Appl.
doi: 10.4208/nmtma.OA-2021-0035

Vol. **15**, No. 1, pp. 42-67
February 2022

# A Local Deep Learning Method for Solving High Order Partial Differential Equations

Jiang Yang[1,2,3] and Quanhui Zhu[2,*]

[1] *International Center of Mathematics, Southern University of Science and Technology, Shenzhen 518055, China*
[2] *Department of Mathematics, Southern University of Science and Technology, Shenzhen 518055, China*
[3] *Guangdong Provincial Key Laboratory of Computational Science and Material Design, Southern University of Science and Technology, Shenzhen 518055, China*

Received 8 March 2021; Accepted (in revised version) 12 August 2021

**Abstract.** At present, deep learning based methods are being employed to resolve the computational challenges of high-dimensional partial differential equations (PDEs). But the computation of the high order derivatives of neural networks is costly, and high order derivatives lack robustness for training purposes. We propose a novel approach to solving PDEs with high order derivatives by simultaneously approximating the function value and derivatives. We introduce intermediate variables to rewrite the PDEs into a system of low order differential equations as what is done in the local discontinuous Galerkin method. The intermediate variables and the solutions to the PDEs are simultaneously approximated by a multi-output deep neural network. By taking the residual of the system as a loss function, we can optimize the network parameters to approximate the solution. The whole process relies on low order derivatives. Numerous numerical examples are carried out to demonstrate that our local deep learning is efficient, robust, flexible, and is particularly well-suited for high-dimensional PDEs with high order derivatives.

**AMS subject classifications**: 35Q68

**Key words**: Deep learning, deep neural network, high order PDEs, reduction of order, deep Galerkin method.

## 1. Introduction

Partial differential equations (PDEs) play a significant role in the fields of physics, chemistry, biology, engineering, finance, and others. Classical numerical methods focus

[*]Corresponding author. *Email addresses:* `yangj7@sustech.edu.cn` (J. Yang), `11849393@mail.sustech.edu.cn` (Q. Zhu)

on designing efficient, accurate, and stable numerical schemes. Within the context of high-dimensional problems, however, the curse of dimensionality renders classical numerical methods impractical. As a result, many mathematicians have introduced neural networks into PDEs precisely because multilayer feedforward networks are proven to be universal approximators for the PDEs [15,16]. More specifically, once the network structure is determined, any order derivatives of the neural network can be obtained analytically. Coupled with the automatic differentiation technique, neural networks can be applied to solve PDEs [2]. Depending upon different purposes, neural networks can be used to approximate the solution function, represent the solution solver, and even invert the equations.

In this paper, we consider the deep learning method as a means to solve the following $k$-th order initial boundary value problem (IBVP):

$$\begin{cases} u_t = \mathcal{L}(u), & x \in \Omega, \quad t \in [0,T], \\ u(x,0) = u_0(x), & x \in \Omega, \\ \mathcal{B}u = \mathbf{g}, & x \in \partial\Omega, \quad t \in [0,T], \end{cases} \tag{1.1}$$

where $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}_+$, $\mathcal{L}(u) = F(x,t,u,Du,\cdots,D^k u)$, $F$ and $\mathbf{g}$ are linear or nonlinear functions, $\mathcal{B}$ is the boundary condition operator, and the $p$-th order derivative operator $D^p$ consists of

$$\partial_{x_1}^{\alpha_1}\partial_{x_2}^{\alpha_2}\cdots\partial_{x_d}^{\alpha_d}u \quad \text{with} \quad \sum \alpha_i = p, \quad \alpha_i \in \mathbb{N}.$$

The neural network function $\varphi(x,t;\theta) : \mathbb{R}^{d+1} \times \Theta^M \mapsto \mathbb{R}^m$ is defined as follows:

$$\begin{aligned} &\varphi(x,t;\theta) = \mathcal{N}_{\text{out}} \circ \mathcal{N}_L \circ \cdots \circ \mathcal{N}_1 \circ \mathcal{N}_{\text{in}}(x,t), \\ &\mathcal{N}_{\text{in}}(x,t) = \sigma_{\text{in}}(\alpha x + \beta t + b), \quad \alpha \in \mathbb{R}^{n\times d}, \quad \beta, b \in \mathbb{R}^n, \\ &\mathcal{N}_{\text{out}}(y) = \sigma_{\text{out}}(\gamma y + c), \qquad\quad \gamma \in \mathbb{R}^{m\times n}, \quad c \in \mathbb{R}^m, \end{aligned} \tag{1.2}$$

where $d$ is the dimension of $x$, $m$ is the dimension of the output, $n$ is the width of the hidden layers, $L$ is the number of the hidden layers (i.e., the network's depth) and $\mathcal{N}_i : \mathbb{R}^n \mapsto \mathbb{R}^n$ is the structure of the hidden layers. $\sigma_{\text{in}}$ usually is the same nonpolynomial activation function as the hidden layers and $\sigma_{\text{out}}$ is set as an equivalent function in most cases, i.e., $\sigma_{\text{out}}(x) = x$. For specific examples, a proper output transformation $\sigma_{\text{out}}$ should be determined. Our goal is to find a suitable neural network $\varphi(x,t;\theta)$ to approximate a solution $u(x,t)$ to the problem (1.1).

[7] gives an overview of the progress that has been made in linking computational mathematics and machine learning. In most of existing literatures, the loss function is determined by either the PDEs or an equivalent formulation. For instance, the parabolic PDE is reformulated as a backward stochastic differential equation in [10,11,31], where the loss function is given by the solution of the backward stochastic differential equation, and the training process is shown to be a deep reinforcement learning process. In [28,29], the solution is approximated by a neural network. The proposal of a mesh-free algorithm makes high-dimensional calculations feasible. [21,30] provides

multi-scale deep neural network methods which separate different frequencies of the loss function and approximate them by the corresponding neural networks. [37] considers variational problems, and the loss function is defined as a weak formulation. To deal with the essential boundary conditions, [20] resorts to Nitsche's variational formulation. Further, [32] introduces an adversarial network as a test function in variational problems; this is particularly suitable for high-dimensional PDEs defined in irregular domains.

We consider using deep learning methods to solve PDEs with high order derivatives. The cost of computing high order derivatives for neural networks is prohibitive. Addressing the issue of deep learning methods, [29] proposes a Monte Carlo method to approximate second order derivatives. In [32, 37], the variational form reduces the order of derivatives through the integration by parts. [11] proposes a derivative free method for parabolic PDEs by solving the equivalent BSDE problem. But, for the higher order derivatives of neural networks, there are still numerous computational challenges. High order derivatives limits the choices of network structure, influences the robustness in training, and are expansive to be computed.

The local discontinuous Galerkin (LDG) method introduces new variables and rewrites the problem (1.1) as a system of first order differential equations [5, 34–36]. Then, the method is obtained by discretizing the system with the discontinuous Galerkin method. The reduction of order technique in LDG inspires us to use a similar technique to compute high order derivatives in deep learning. To this end, we first rewrite the PDEs to a system of low order differential equations. A neural network with multiple outputs is then used to approximate the solution and intermediate variables. Taking the $L_2$ residual of the system as the loss function, we can optimize the neural network to approximate the solution of (1.1). Unlike the classical deep learning methods, our approach avoids calculating the high order derivatives. As consequence, it is more efficient and stable.

Our paper is organized as follows. In Section 2, we briefly introduce the deep learning method for solving PDEs and illustrate the difficulties in computing the high order derivatives of the neural networks. After rewriting the problem as a system of low order equations, the local deep Galerkin method and the local deep Ritz method are proposed in Section 3. The advantages of this method are provided in Section 4, and the corresponding numerical experiments are presented in Section 5. Several concluding remarks are given in the final section.

## 2. Preliminaries

In this section, we present a deep learning based method for solving PDEs and show the main difficulties in computing the high order derivatives of neural networks.

### 2.1. Deep learning based method for solving PDEs

A deep neural network defined as (1.2) is used to approximate the solution of (1.1).

Substituting the neural network into (1.1), we have

$$\begin{cases} \varphi_t = \mathcal{L}(\varphi), & x \in \Omega, \quad t \in [0,T], \\ \varphi(x,0) = u_0(x), & x \in \Omega, \\ \mathcal{B}\varphi = \mathbf{g}, & x \in \partial\Omega, \quad t \in [0,T]. \end{cases} \tag{2.1}$$

Instead of solving the equation step by step under a given initial value, the neural network parameters should be optimized to satisfy the dynamic system, the initial value condition and the boundary condition. The loss function $J(\varphi)$ is defined by the $L_2$ norm mostly [6, 13, 18, 19, 22, 28, 29], i.e.,

$$J(\varphi) = w_e \|\varphi_t - \mathcal{L}(\varphi)\|^2_{\Omega_T} + w_i \|\varphi(\cdot,0) - u_0(\cdot)\|^2_{\Omega} + w_b \|\mathcal{B}\varphi - \mathbf{g}\|^2_{\partial\Omega_T}, \tag{2.2}$$

where $\Omega_T = \Omega \times [0,T]$, $\partial\Omega_T = \partial\Omega \times [0,T]$ and $w$ controls the contribution of each term. Denote

$$J(\varphi) := w_e J_e(\varphi) + w_i J_i(\varphi) + w_b J_b(\varphi),$$

which represents the equation loss, the initial condition loss, and the boundary condition loss, respectively. The learning algorithm is described in Algorithm 2.1.

The selection of the active function, initialization method, random sampling distribution, and optimization method will affect the approximation of the network. Incorrect settings will result in either the failure of the neural network to converge or a very slow rate of convergence.

## 2.2. High order derivatives of the deep neural network in PDEs

High order derivatives of the neural network rarely appear in classic deep learning problems. But, to solve high order PDEs, we have to calculate the value of high order derivatives. Consider a fully connected neural network $\varphi(x,\theta) : \mathbb{R}^d \times \Theta^M \mapsto \mathbb{R}$ similar to Fig. 1, in which there are $L$ hidden layers and $n$ neurons per layer.

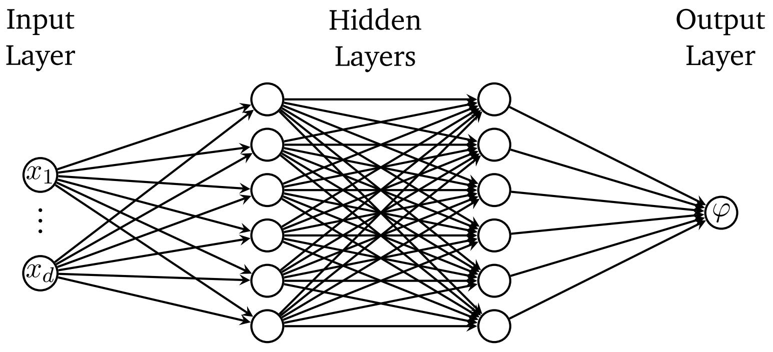


Figure 1: The structure of the fully connected neural network.

**Algorithm 2.1**

1: Build up a neural network $\varphi(x,t;\theta)$, which determines active functions, the hidden layer structure, and the network's depth and width. The inputs are space $x$ and time $t$ and the output is the function value at $(x,t)$, which approximates the solution of (1.1). $\theta$ is the trainable parameters in the neural network.

2: Obtain random samples

$$\mathcal{D}=\big\{(x_k^e,t_k^e)\big\}_{k=1}^{N_e}\cup\big\{(x_k^i,0)\big\}_{k=1}^{N_e}\cup\big\{(x_k^b,t_k^b)\big\}_{k=1}^{N_b}$$

from within the domain $\Omega_T$, $\partial\Omega_T$ and $\{0\}\times\Omega$, respectively. $N_e,N_i,N_b$ are the number of sampling nodes in different domains. Random sampling makes high-dimensional calculations feasible. The mesh-free property is one of the most critical differences between deep learning methods and classical numerical methods.

3: Solve the optimization problem on the given sampling set $\mathcal{D}$

$$\min_\theta J(\varphi)|_{\mathcal{D}}.$$

The discrete form of the loss function is given as

$$\begin{aligned}
J(\varphi)|_{\mathcal{D}} &= w_eJ_e(\varphi)|_{\mathcal{D}_1}+w_iJ_i(\varphi)|_{\mathcal{D}_1}+w_bJ_b(\varphi)|_{\mathcal{D}_3},\\
J_e(\varphi)|_{\mathcal{D}_1} &= \frac{1}{N_e}\sum_{k=1}^{N_e}\Big(\varphi_t\big(x_k^e,t_k^e\big)-\mathcal{L}(\varphi)\big(x_k^e,t_k^e\big)\Big)^2,\\
J_i(\varphi)|_{\mathcal{D}_2} &= \frac{1}{N_i}\sum_{k=1}^{N_e}\Big(\varphi\big(x_k^i,0\big)-u_0\big(x_k^i\big)\Big)^2,\\
J_b(\varphi)|_{\mathcal{D}_3} &= \frac{1}{N_b}\sum_{k=1}^{N_b}\Big(\mathcal{B}\varphi\big(x_k^b,t_k^b\big)-\mathbf{g}\big(x_k^b,t_k^b\big)\Big)^2.
\end{aligned}\tag{2.3}$$

Finding the optimal parameters for a fixed width neural network is difficult since the optimization problem is nonconvex. The Adam optimizer is a popular choice for deep learning [17].

4: Repeat steps 2 and 3 until the result converges.

The layer structure is

$$\mathcal{N}_i(x)=\sigma\big(W_i\mathcal{N}_{i-1}(x)+b_i\big),\quad W_i\in\mathbb{R}^{n\times n},\quad b_i\in\mathbb{R},\quad i=2,\ldots,L.\tag{2.4}$$

Based on the chain rule, the computational cost of $k$-th order derivative of $\varphi$ is about $\mathcal{O}(L^kn^{2k})$, and $D_x^k\varphi$ costs about $\mathcal{O}(L^kd^kn^{2k})$. Although the automatic differentiation technique provides some convenience in practical problems, the exponential growth of the order $k$ is unacceptable for solving a specific high order PDE.

Putting aside the problem of computational efficiency, there are still inherent challenges to using deep neural networks to solve PDEs with high order derivatives. The gradient vanishing and exploding problems of neural networks have been discussed for many years [3, 9, 12, 14, 27]. And each probably makes an appearance when PDEs are being solved. For simplicity, we consider when $n = d = 1$ and $\mathcal{N}_{\text{out}}(x) = \mathcal{N}_{\text{in}}(x) = x$, i.e.,

$$\varphi(x) = \prod_{i=1}^{L} \sigma(\mathcal{N}_i). \tag{2.5}$$

Then, we have the following first order derivative:

$$\frac{\partial \varphi}{\partial x} = \frac{\partial \varphi}{\partial \mathcal{N}_L} \cdot \prod_{i=2}^{L} \frac{\partial \mathcal{N}_i}{\partial \mathcal{N}_{i-1}} \cdot \frac{\partial \mathcal{N}_1}{\partial x} = \prod_{i=1}^{L} W_i \sigma'(\mathcal{N}_i). \tag{2.6}$$

Given the active function $\sigma(x) = \tanh(x)$, we have $\sigma'(x) = 1 - \tanh^2(x) < 1$, except the point $x = 0$. With a normal initialization $|W| < 1$, $\varphi_x(x;\theta) \sim W^L \sigma'(x)^L$ will become small and this leads to the gradient vanishing problem. Similarly, if the active function satisfies $|W\sigma'(x)| > 1$, it will result in the gradient exploding problem.

There are many mature deep learning techniques for resolving these problems in classification, regression, and so on. But, for deep learning methods in PDEs, things are different. Consider a $k$-th order ordinary differential equation

$$F\big(x, u, u', u'', \cdots, u^{(k)}\big) = 0. \tag{2.7}$$

We use the above neural network to approximate the solution $u(x)$. Substituting (2.5) into (2.7) and denoting $\sigma_i^{(k)} = \sigma^{(k)}(\mathcal{N}_i), k \in \mathbb{N}$, we have

$$F\left(x, \prod_{i=1}^{L} \sigma_i, \prod_{i=1}^{L} W_i \sigma_i', \cdots, \prod_{i=1}^{L} W_i^k \sigma_i^{(k)}\right) = 0. \tag{2.8}$$

The issue is different from that of the classical deep learning problems. Our concern cannot just be limited to whether $|W\sigma'|$ is exploding or vanishing. The high order derivatives of the active function and the high order powers of parameters bring new difficulties. Taking $y := \sigma(x) = \tanh(x)$ as an example, we have

$$\begin{aligned}
&y' = 1 - y^2, && y' \in (0, 1], \\
&y'' = -2y\big(1 - y^2\big), && y'' \in \left[-4\sqrt{3}/9, 4\sqrt{3}/9\right], \\
&y^{(3)} = \big(6y^2 - 2\big)\big(1 - y^2\big), && y^{(3)} \in [-2, 2/3], \\
&y^{(4)} = 8y\big(2 - 3y^2\big)\big(1 - y^2\big), && y^{(4)} \in (-4.086, 4.086).
\end{aligned} \tag{2.9}$$

The derivative value can be greater than 1 and the different order derivatives are controlled by the different scales. Fig. 2 gives an example of gradient exploding problems

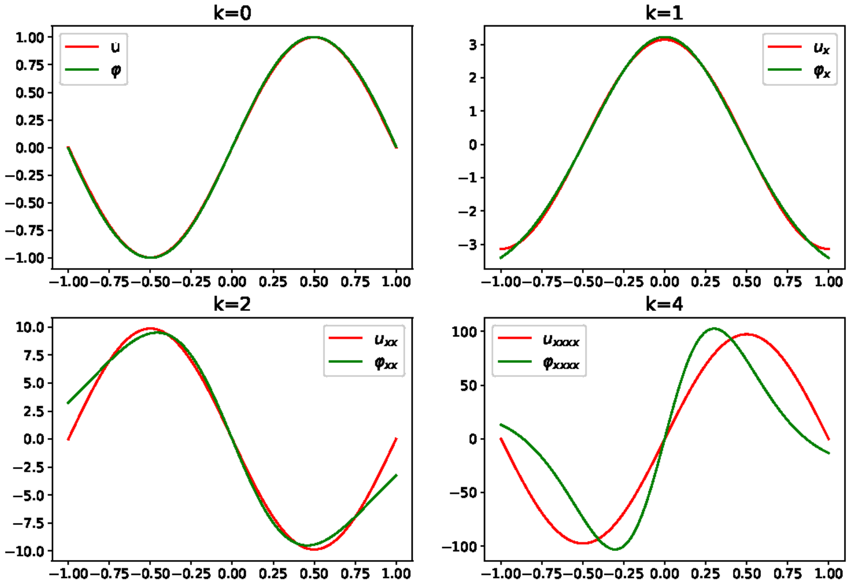


Figure 2: Different order derivatives of the neural network $D^k\varphi$ compared to the target function $D^k u$ in one dimension.

of high order derivatives in using a $L=3, n=32$ neural network $\varphi$ to approximate the function $u(x)=\sin(\pi x)$. When order $k=4$, the gap between two derivatives is about $\mathcal{O}(10)$, even though $\varphi(x)$ approximates $u(x)$ well.

We use the following terms to represent the different order derivatives of a neural network with $L$ hidden layers:

$$\varphi^{(k)}(x;\theta)\sim\left(\sigma^{(k)}(x)\theta^k\right)^L. \tag{2.10}$$

The gradient of the parameters is given as

$$\nabla_\theta\varphi^{(k)}(x;\theta)\sim\left(x\sigma^{(k+1)}(x)\theta^k+k\sigma^{(k)}(x)\theta^{k-1}\right)^L. \tag{2.11}$$

The restriction on $\sigma'(x)\theta$ cannot restrict higher order derivatives. High order derivatives become more sensitive and unstable. It seems that different order derivatives are learned on different scales. When different order derivatives are in same equation, optimizing the loss is difficult (see also Section 5.3). The neural networks lack the efficiency and robustness of computing high order derivatives.

## 3. Methodology

In this section, we propose employing local deep learning methods (LDLM) to overcome the derivative calculation problem of neural networks by using a multi-output neural network and a loss function of the equivalent system.

### 3.1. The system of PDEs

Consider a $k$-th order IBVP

$$\begin{cases} u_t = F\big(u, Du, D^2u, \cdots, D^k u\big), & x \in \Omega, \quad t \in [0,T], \\ u(x,0) = u_0(x), & x \in \Omega, \\ \mathcal{B}u = \mathbf{g}, & x \in \partial\Omega, \quad t \in [0,T]. \end{cases} \tag{3.1}$$

Similar to the first part of the local discontinuous Galerkin method, we introduce the intermediate variables $\{v_i\}_{i=1}^k$, where $v_i \in \mathbb{R}^{d^i}, i = 1, \ldots, k$. Then the PDE can be rewritten as the following system:

$$\begin{cases} u_t = F(u, v_1, v_2, \cdots, v_k), & x \in \Omega, \quad t \in [0,T], \\ v_1 = Du, & x \in \Omega, \quad t \in [0,T], \\ v_{i+1} = Dv_i, \quad i = 1, \ldots, k-1, & x \in \Omega, \quad t \in [0,T], \\ u(x,0) = u_0(x), & x \in \Omega, \\ \mathcal{B}u = \mathbf{g}, & x \in \partial\Omega, \quad t \in [0,T]. \end{cases} \tag{3.2}$$

Only the first order derivatives are included in the system. Similarly, we can build a system of the second order PDEs.

$$\begin{cases} u_t = F\big(u, Du, \cdots, w_{[\frac{k}{2}]}, Dw_{[\frac{k}{2}]}\big), & x \in \Omega, \quad t \in [0,T], \\ w_1 = D^2 u, & x \in \Omega, \quad t \in [0,T], \\ w_{i+1} = D^2 w_i, \quad i = 1, \ldots, [\frac{k}{2}], & x \in \Omega, \quad t \in [0,T], \\ u(x,0) = u_0(x), & x \in \Omega, \\ \mathcal{B}u = \mathbf{g}, & x \in \partial\Omega, \quad t \in [0,T]. \end{cases} \tag{3.3}$$

Some specific examples are given in Table 1.

Table 1: The system form of serval classical equations.

| Equation | Origin Form | First Order System | Second Order System |
|---|---|---|---|
| Heat | $u_t = \Delta u$ | $\begin{cases} u_t = \nabla \cdot v, \\ v = \nabla u. \end{cases}$ | $\begin{cases} u_t = v, \\ v = \Delta u. \end{cases}$ |
| Allen-Cahn | $u_t = \epsilon \Delta u + f(u)$ | $\begin{cases} u_t = \epsilon \nabla \cdot v + f(u), \\ v = \nabla u. \end{cases}$ | $\begin{cases} u_t = \epsilon v + f(u), \\ v = \Delta u. \end{cases}$ |
| Cahn-Hilliard | $u_t = -\Delta\big(\epsilon \Delta u + f(u)\big)$ | $\begin{cases} u_t = -\nabla \cdot v, \\ v = \nabla \phi, \\ \phi = \epsilon \nabla \cdot w + f(u), \\ w = \nabla u. \end{cases}$ | $\begin{cases} u_t = -\Delta v, \\ v = \epsilon \Delta u + f(u). \end{cases}$ |
| KdV | $u_t + 6uu_x + u_{xxx} = 0$ | $\begin{cases} u_t + 6uv + w_x = 0, \\ v - u_x = 0, \\ w - v_x = 0. \end{cases}$ | $\begin{cases} u_t + 6uu_x + v_x = 0, \\ v - u_{xx} = 0. \end{cases}$ |

It is intuitive that the introduction of intermediate variables can effectively reduce the order of the derivatives.

### 3.2. Multi-output neural network

For approximating the intermediate variables and solution, a multi-output neural network $\varphi(x,t;\theta):\mathbb{R}^{d+1}\times\Theta^M\mapsto\mathbb{R}^m$ is needed. The 1-D coupled neural network is described in Fig. 3.

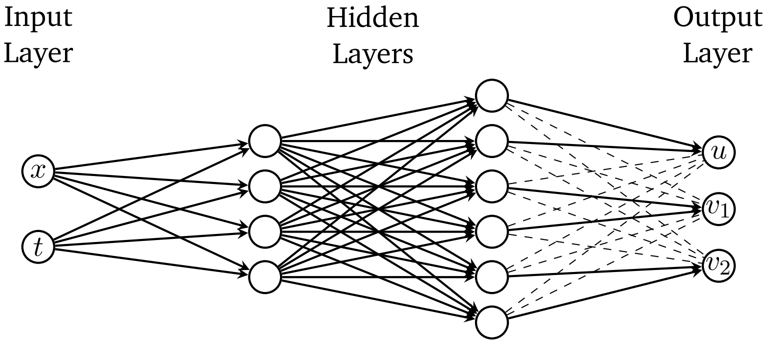


Figure 3: The multi-output neural network for approximating all of the intermediate variables.

In contrast to previous methods, all necessary intermediate variables are included in the output. For greater accuracy, the output layer can be a series of hidden layers. The final active function of each output is usually uniquely determined by the problem. When the number of intermediate variables increases, we only need to change the width of the output layer, and this is much cheaper than computing derivatives.

For high-dimensional problems, a decoupled neural network can better distinguish derivatives and the solution, as is shown in Fig. 4. After a certain number of hidden layer operations, the input is transformed into a series of intermediate states. The states are then separated by some independent hidden layers to calculate the different variables. For example, $(p_1,\cdots,p_d)\approx\nabla u$ and $q\approx\Delta u$. The decoupled network is not always better than the fully connected one. But it provides more flexible dependency on $u$ of the derivatives. The depth of the two types of hidden layers depends on how closely you need the different order derivative to be connected.

### 3.3. Local deep Galerkin method

With the system and network structure defined, we can propose local deep learning methods to solve the PDEs with high order derivatives.

Taking the residual of the system (3.1), the modified loss function is given as fol-

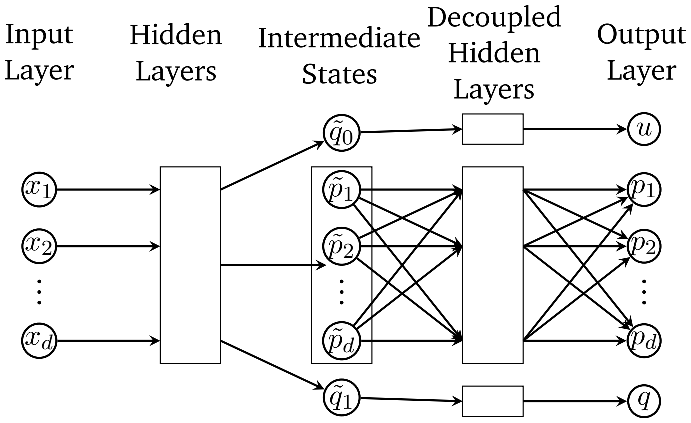


Figure 4: A decoupled neural network structure for solving high-dimensional PDEs with multiple outputs.

lows:

$$\begin{aligned} J_e(\varphi) &= \big\|(\varphi_1)_t - F(\varphi_1,\varphi_2,\cdots,\varphi_k,D\varphi_k)\big\|_{\Omega_T}^2 + \sum_{i=1}^{k-1}\big\|\varphi_{i+1} - D\varphi\big\|_{\Omega_T}^2, \\ J_i(\varphi) &= \big\|\varphi_1(\cdot,0) - u_0(\cdot)\big\|_{\Omega}^2, \end{aligned} \tag{3.4}$$

and the boundary condition is also expressed by the intermediate variables. Taking heat equation as an example, we define $\varphi = (\varphi_1,\cdots,\varphi_{d+1})$ and different boundary conditions are given as following:

- Dirichlet boundary condition: $u = g \longrightarrow J_b(\varphi) = \|\varphi_1 - g\|_{\partial\Omega}^2$.
- Neumann boundary condition: $\frac{\partial u}{\partial \mathbf{n}} = g \longrightarrow J_b(\varphi) = \frac{1}{d}\sum_{i=1}^d \|\varphi_{i+1} - g\|_{\partial\Omega_i}^2$, where $\partial\Omega_i$ means the $i$-th variable $x_i$ lies on the boundary.
- Periodic boundary condition: Let

  $$\Omega = [a_1,b_1]\times\cdots\times[a_d,b_d],$$

  then the loss function is

  $$J_b(\varphi) = \sum_{i=1}^d \big\|\varphi(\mathbf{x}_i^l) - \varphi(\mathbf{x}_i^r)\big\|_{\partial\Omega_i}^2, \tag{3.5}$$

  where $\mathbf{x}_i^l$ means the $i$-th variable of $\mathbf{x}$ is on the left boundary, i.e., $(\mathbf{x}_i^l)_i = a_i$ and similarly $(\mathbf{x}_i^r)_i = b_i$.

The local deep Galerkin method, which follows deep Galerkin method [29], is summarized in Algorithm 3.1. This method treats the solution and its derivatives or other necessary intermediate variables as unknown functions while simultaneously learning their values. These restrictions cause a certain increase in calculations, but this is still far less expansive than calculating the derivatives.

**Algorithm 3.1** Local deep Galerkin method (LDGM)

```
Build up the neural network $\vec{\varphi}(x,t;\theta)$. Determine the hidden layer structure $\mathcal{N}$, the layer width $n$, the dimension of output $m$ and the active functions $\sigma$ according to the given PDEs.
Initialize the parameters $\theta = \theta^0$, sampling times $s_1$, the number of samples $N_e, N_i, N_b$ in $\Omega_T, \Omega, \partial\Omega_T$, optimization steps $s_2$ and the learning rate $\gamma$.
for $i = 0 : s_1$ do
    Obtain random sampling points $\{(x_e,t_e)\}_{N_e}, \{(x_i,0)\}_{N_i}, \{(x_b,t_b)\}_{N_b}$.
    Set $\theta^{i,0} = \theta^i$.
    for $j = 0 : s_2$ do
        Calculate the loss function $J(\varphi(x,t;\theta^{i,j}))$ at sampling points.
        Optimize the parameters $\theta$
            $$\theta^{i,j+1} = \theta^{i,j} - \gamma\nabla_\theta J(\theta^{i,j}).$$
    end for
    Set $\theta^{i+1} = \theta^{i,s_2+1}$.
end for
```

## 3.4. Local deep Ritz method

We can also combine the technique discussed above with the deep Ritz method [31]. Consider the following bi-Laplacian equation:

$$\begin{cases} \Delta^2 u = f, & x \in \Omega, \\ u(x) = 0, & x \in \partial\Omega, \\ \dfrac{\partial u}{\partial \mathbf{n}} = 0, & x \in \partial\Omega, \end{cases} \tag{3.6}$$

the weak formulation of which is

$$J(u) = \int_\Omega \left( \frac{1}{2}\big(\Delta u(x)\big)^2 - f(x)u(x) \right) dx, \tag{3.7}$$

where $u \in H$ and $H$ is the set of trial functions. Using a neural network $\varphi$, the loss function in the deep Ritz method is defined as

$$\begin{aligned} J(\varphi) = & \int_\Omega \left( \frac{1}{2}\big(\Delta_x \varphi(x;\theta)\big)^2 - f(x)\varphi(x;\theta) \right) dx \\ & + \lambda \int_{\partial\Omega} \left( \varphi(x;\theta)^2 + \left( \frac{\partial \varphi(x;\theta)}{\partial \mathbf{n}} \right)^2 \right) dx. \end{aligned} \tag{3.8}$$

We use a multi-output neural network $\varphi(x;\theta) : \mathbb{R}^d \times \Theta^M \mapsto \mathbb{R}^{d+1}$, and reformulate the loss function as follows:

$$\begin{aligned}\hat{J}(\varphi) = &\int_\Omega \left(\frac{1}{2}\big(\nabla_x \cdot \mathbf{q}(x;\theta)\big)^2 - f(x)p(x;\theta) + \|\nabla_x p(x;\theta) - \mathbf{q}(x;\theta)\|^2\right) dx \\ &+ \lambda \int_{\partial\Omega} \big(p(x;\theta)^2 + (\mathbf{q}(x;\theta)\cdot\mathbf{n})^2\big) dx, \end{aligned} \tag{3.9}$$

where $p = \varphi_1, \mathbf{q} = (\varphi_2, \cdots, \varphi_{d+1})$. A local deep Ritz method can then be used to solve the variational problem.

Notice that we can solve a fourth order problem with a $d+1$-dimensional output neural network. For high-dimensional problems which are the kind of variational problems, the local deep learning method can also be applied.

## 4. Advantages of LDLM

In this section, we illustrate the advantages of using the local deep learning method to solve differential equations with high order derivatives. The intuitive performance of some numerical tests will be shown in Section 5.

### 4.1. Reduction of calculations

The computational complexity of computing a $k$-th order derivative is $\mathcal{O}(L^k n^{2k})$, which is delineated in Section 2.2. In the LDLM, a $k$-th order derivative becomes $k-1$ restrictions and $k$ first order derivatives. As the restrictions are much cheaper than computing derivatives, the total cost is about $\mathcal{O}(Lkn^2)$. The linear growth with respect to the order $k$ is especially suitable for solving high order PDEs.

### 4.2. Improving robustness for solving complex differential equations

The robustness of local deep learning methods is made manifest in the solving of complex nonlinear differential equations, which contain different order derivatives and multiple scales, like the Cahn-Hilliard equation (5.2). In Section 2.2, we see that the scale of a $k$-th order derivative is about

$$u^{(k)}(x;\theta) \sim \big(\sigma^{(k)}(x)\theta^k\big)^L. \tag{4.1}$$

Considering the $p$-th order derivative and the $q$-th order derivative in one equation, the formulation contains

$$\theta^{pL}\left(\big(\sigma^{(p)}(x)\big)^L + \big(\sigma^{(q)}(x)\big)^L \theta^{(q-p)L}\right). \tag{4.2}$$

Assuming $\sigma^{(p)}(x) \sim \sigma^{(q)}(x)$, the term $\theta^{(q-p)L}$ is sensitive if $p \neq q$ and $L \gg 1$. The initialization of the parameters $\theta$ will seriously impact the performance of a neural network. This will cause the information of one of the derivatives to be neglected during the training process when $|(q-p)L| \gg 1$.

Additionally, high order derivative terms are often accompanied by small coefficients, like the viscosity $\mu$ in the Navier-Stokes equation and the interface width $\epsilon$ in the Cahn-Hilliard equation. In traditional numerical methods, high order numerical schemes or multi-scale analysis methods can overcome the equation's parameter sensitivity. But small coefficients, coupled with the different order derivatives of the neural network, can cause great difficulties in optimization. In other words, the neural network may not converge due to the sensitivity of the small coefficients.

The modified loss function of the system only includes

$$\sigma^L(x) + \big(\sigma'(x)\big)^L \theta^L. \tag{4.3}$$

It is easier to assume $\sigma(x) \sim \sigma'(x)$, and the parameter scale can be balanced by $\sigma(x) \sim \sigma'(x)\theta$. In training, different order derivatives only affect adjacent ones, which leads to a more robust result.

## 4.3. Weaker restrictions on activation functions

Following the above, local deep learning methods place less of restriction on active functions. Different active functions have different uses in neural networks. For example, the ReLU active function allows us to circumvent the gradient vanishing problem and the hyperbolic tangent active function can provide smoothness. Choosing a suitable active function for a given task is an open hyperparameter learning problem.

For PDEs containing the $k$-th order derivative, the neural network $\varphi$ should, at least, belong to $C^{k+1}(\Omega)$ from (4.1), where $C(\Omega)$ is the collection of continuous functions on $\Omega$ and $C^k(\Omega) := \{f | f^{(k)} \in C(\Omega)\}$. Combining this with (4.2), the active function should satisfy

i. $\sigma(x) \in C^{k+1}(\Omega)$;

ii. $\sigma^{(p)}(x) \sim \sigma^{(q)}(x),\ \forall x \in \Omega,\ 1 \leq p, q \leq k+1.$

Condition (i) can be satisfied with smooth nonpolynomial active functions, like the hyperbolic tangent and sigmoid. But a common problem is that these active functions inevitably lead to gradient vanishing and exploding problems. Other popular active functions, like ReLU and ReCU, do not meet the condition. Condition (ii) is much more stringent. Among the common elementary functions, only the exponential function satisfies this condition, but it is not usually used as an active function.

For local deep learning methods, these conditions are weakened as

iii. $\sigma(x) \in C^2(\Omega)$;

iv. $\sigma(x) \sim \sigma'(x),\ \forall x \in \Omega.$

If we approximate the weak solution of the equation, condition (iii) can be further weakened as $\sigma \in C^1(\Omega)$ and the weak derivative of $\sigma'$ exists. Then, some active functions with nonexistent second order derivatives can be used to train the neural network. Condition (iv) means that $\sigma'$ should be bounded in $\Omega$.

The weaker restriction on the active functions provides more choices in local deep learning methods.

### 4.4. More flexible choices of network structures

Deep neural networks usually have a large number of hidden layers. As mentioned above, the high order derivatives of deep neural networks not only increase the size of the calculation, but also destabilize the training process. So local deep learning methods, which have successfully avoided the calculation of high order derivatives, can be better combined with deep neural networks.

In addition, we can also use more complex neural network models. For example, one advantage of the residual layer

$$\mathcal{N}(x) = \sigma\big(W_2\sigma(W_1x + b_1) + b_2\big) + x, \tag{4.4}$$

is that it avoids the gradient vanishing problem. This is because the linear term $x$ keeps an additive constant gradient in the first order derivatives, i.e.,

$$\frac{\partial \mathcal{N}}{\partial z} = \big(F'(x) + 1\big)\frac{\partial x}{\partial z}, \tag{4.5}$$

where $F(x) = \sigma(W_2\sigma(W_1x + b_1) + b_2)$. But for high order derivatives, it is not usually effective. For example,

$$\frac{\partial^3 \mathcal{N}}{\partial z^3} = F'''(x)\left(\frac{\partial x}{\partial z}\right)^3 + 3F''(x)\frac{\partial x}{\partial z}\frac{\partial^2 x}{\partial z^2} + F'(x)\frac{\partial^3 x}{\partial z^3} + \frac{\partial^3 x}{\partial z^3} \tag{4.6}$$

it can always avoid the gradient vanishing problem, but it cannot grasp the contribution of various components to the high order derivatives well, which, in turn, will lead to an inaccurate calculation of the derivatives. For other complex network structures, like the long short term memory layer, the calculation of high order derivatives relies on a series of complex composite functions which greatly increases complexity of the computation. For local deep learning methods, which are similar to classical deep learning problems, many existing tools, methods and network structures can be directly migrated to solve high order differential equations.

## 5. Numerical examples

### 5.1. Setup

In this section, we use the local deep Galerkin method (LDGM) to compute a series of examples including high-dimensional linear and nonlinear differential equations with high order derivatives. The accuracy of the solution $\varphi(x;\theta)$ is measured by the relative $L_2$ error $\|\varphi - u^*\|_2/\|u^*\|_2$ where $u^*$ is the exact solution and

$$\|u\|_2^2 = \int_\Omega u^2 dx.$$

If the exact solution is not analytical, we will compare the solution to the reference solution obtained by the finite difference method. The base solution for comparison is obtained by the deep Galerkin method (DGM) [29].

The numerical implementation of the algorithm is based on TensorFlow, which is a widely-used open-source software library in machine learning [1]. The automatic differentiation is included in function tf.gradient$(y, x)$, which returns

$$\sum_{i=1}^{m}\left(\frac{y_i}{x_1},\cdots,\frac{y_i}{x_d}\right),$$

rather than the Jacobi matrix. In all numerical experiments, a fully connected feedforward network is chosen as the network structure. Unless otherwise noted, the neural network is configured to be a coupled network as Fig. 1 with $3$ hidden layers and $50$ neurons per hidden layer. Parameters are initialized by the Xavier Initializer (also known as the Glorot Uniform Initializer), which is used to avoid the gradient vanishing and exploding problems [8]. Most active functions are selected as $\tanh(x)$ for smoothness and final active functions are determined by practical problems. In optimization, we set the learning rate $r = 0.001$, sampling times $s_1 = 1000$, optimization steps $s_2 = 5$ as a default and use the Adam optimizer. Sampling settings are $N_e = 200, N_i = 50, N_b = 50$, which contain a total of $300$ nodes per suboptimization problem, and the uniform distribution is used for sampling. The weights of loss functions are chosen equal $w_e = w_i = w_b = 1$, unless there is a singularity on the boundary or initial condition.

Notations of the experiments and algorithm parameters are summarized in Table 2 for quick reference.

Table 2: The list of parameters.

| Notation | Stands for ... |
|---|---|
| $\varphi(x,t;\theta)$ | Neural network of input $(x,t)$ with trainable parameters $\theta$ |
| $d$ | Dimension of $\Omega \subset \mathbb{R}^d$ |
| $L$ | Number of hidden layers |
| $m$ | Dimension of output |
| $n$ | Hidden layer width |
| $\sigma$ | Active functions in neural network |
| $J_e, J_i, J_b$ | The loss of the equation, the initial value and the boundary condition |
| $w_e, w_i, w_b$ | The weights of the losses |
| $N_e, N_i, N_b$ | Number of sampling nodes on the domain $\Omega_T$,$\Omega \times \{0\}$ and $\partial\Omega_T$ |
| $r$ | Learning rate of network parameter $\theta$ |
| $s_1$ | Sampling times on the whole training process |
| $s_2$ | Optimization steps of per sampling stage |

## 5.2. Fourth order PDE

In the first case, we show that while there is no obvious difference between the LDGM and the DGM in terms of accuracy, the LDGM can greatly speed up the calculation.

We consider a simple model for a vibrating elastic beam first [26]

$$u_t = -u_{xxxx}, \quad x \in [0, 2\pi], \quad t \in [0, 1]. \tag{5.1}$$

With the Dirichlet boundary conditions $u(x,t) = 0, u_{xx}(x,t) = 0, x \in \partial\Omega$ and the initial condition $u_0(x) = \sin x$, the exact solution is given as $u(x,t) = e^{-t}\sin x$. It is costly to calculate the fourth order derivative in deep neural networks while we get high order derivatives directly from the multi-output neural network.

From Fig. 5, we find that under the same iteration step, the LDGM is trained much faster than the DGM, while the error is slightly different. In the third picture, we can conclude that the LDGM approaches the solution of the PDEs containing high order derivatives faster than the DGM. It saves a lot of time in training, and the advantage will be magnified as the network's depth increases.

Another interesting aspect of this study is that the oscillation amplitude of the LDGM is smaller than that of the DGM. Although not sufficient, we realize that the local deep Galerkin method is more stable for PDEs with high order derivatives.

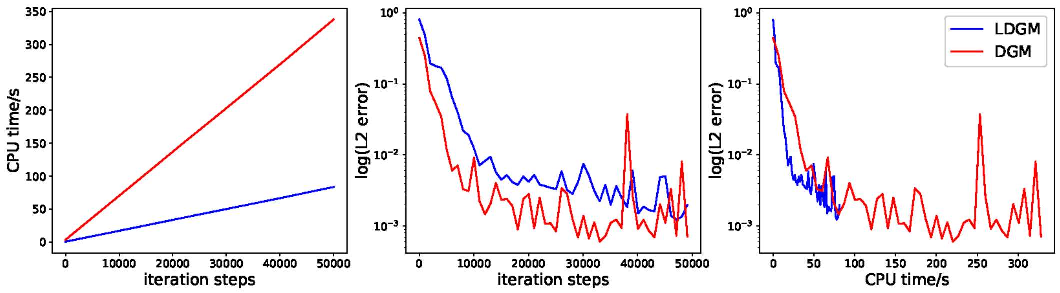


Figure 5: The training processes of DGM and LDGM with the learning rate $r = 10^{-4}$ and $s_1 \times s_2 = 50000$ training steps. The red line is the deep Galerkin method and the blue line is the local deep Galerkin method. The left figure shows the time spent between two methods under the same iteration step. The middle figure shows the logarithmic $L_2$ error with respect to the iteration step. The right figure shows how fast the $L_2$ error drops.

## 5.3. Cahn-Hilliard equation

The Cahn-Hilliard (CH) equation is a popular mathematical physical equation used to describe the process of phase separation. When we use the deep learning method to solve the Cahn-Hilliard equation, it fails when $\epsilon$ is small. This is why we propose the local method to strengthen the robustness in training.

The 1-D CH equation can be given as

$$u_t + \epsilon u_{xxxx} + f(u)_{xx} = 0, \quad x \in [0, 2\pi], \quad t \in [0, 1], \tag{5.2}$$

where $f(u) = u - u^3$. Given the initial condition $u_0(x) = \cos x$ and the zero Neumann boundary condition, we define the following loss function:

$$\begin{aligned}
J_e(\varphi) &= \|(\varphi_1)_t - (\varphi_4)_x\|^2_{\Omega_T} + \|\varphi_3 + \epsilon(\varphi_2)_x + f(\varphi_1)\|^2_{\Omega_T} \\
&\quad + \|\varphi_2 - (\varphi_1)_x\|^2_{\Omega_T} + \|\varphi_4 - (\varphi_3)_x\|^2_{\Omega_T}, \\
J_i(\varphi) &= \|\varphi_1 - \cos(x)\|^2_{\Omega}, \\
J_b(\varphi) &= \|\varphi_2\|^2_{\partial\Omega_T} + \|\varphi_4\|^2_{\partial\Omega_T}, \\
J(\varphi) &= J_e(\varphi) + w_i J_i(\varphi) + w_b J_b(\varphi).
\end{aligned} \tag{5.3}$$

Here, the multi-output neural network is

$$\varphi(x,t;\theta) = \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \end{pmatrix} \approx \begin{pmatrix} u \\ u_x \\ \phi \\ \phi_x \end{pmatrix}, \tag{5.4}$$

where $\phi = -\epsilon u_{xx} - f(u)$. For the DGM, classical $L_2$ loss is used. The reference solution is given by a pseudo-spectral method with $129$ spectral nodes, and the numerical scheme is

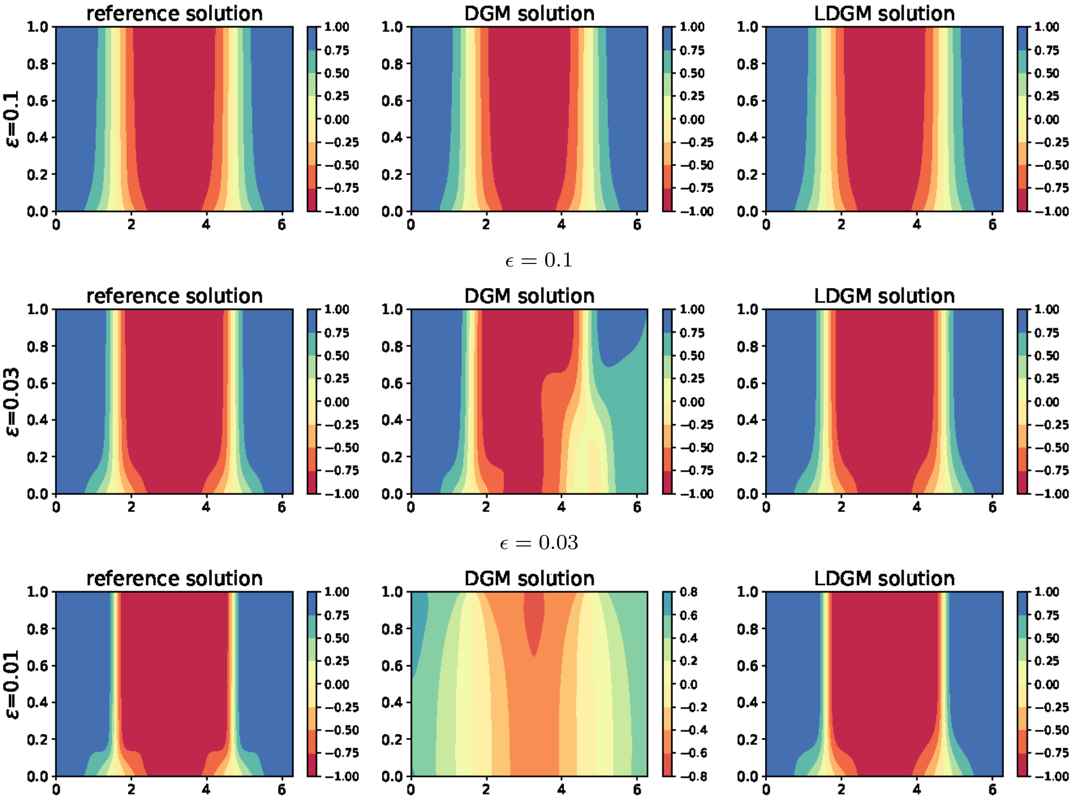
$$\frac{\hat{u}_k^{n+1} - \hat{u}_k^n}{\delta t} + \epsilon(ik)^4 \hat{u}_k^{n+1} + (ik)^2 \hat{f}_k^n = 0, \tag{5.5}$$

where $f^n = u^n - (u^n)^3$, $\delta t = 0.01$ and the FFT solver is used here. We show numerical results in Fig. 6 with sampling stages $s_1 = 5000$. In addition, we know that the penalty coefficients $w_i$ and $w_b$ will influence the training process [4, 25]. Thus we test the performance of the LDGM under different penalty coefficients and the results are given in Table 3.

When $\epsilon \sim \mathcal{O}(1)$ or there is a source term $g(x,t,\epsilon)$, the DGM can be guided to approximate the true solution quickly. But when $\epsilon$ becomes small, the DGM fails while the LDGM is still able to capture the interface. For the DGM, it costs about $s_1 = 40000$ sampling stages to get a reasonably accurate solution when $\epsilon = 0.02$ and it costs about $s_1 = 100000$ sampling stages when $\epsilon = 0.01$ in experiments.

Table 3: The relative L2 error of DGM and LDGM with different penalty coefficients.

| | | | | |
|---|---|---|---|---|
| $\epsilon = 0.1$ | | | | |
| $w_i(w_b)$ | 0.1 | 1 | 10 | 100 |
| DGM | 3.43% | 1.20% | 9.61% | 3.41% |
| LDGM | 2.04 % | 0.80% | 2.03% | 7.05% |
| $\epsilon = 0.01$ | | | | |
| $w_i(w_b)$ | 0.1 | 1 | 10 | 100 |
| DGM | 70.84% | 60.57% | 54.57% | 51.92% |
| LDGM | 6.05% | 2.65% | 13.55% | 32.56% |



$\epsilon = 0.1$

$\epsilon = 0.03$

$\epsilon = 0.01$

Figure 6: The solutions to the Cahn-Hilliard equation are obtained by the pseudo-spectral method, the DGM and the LDGM from left to right with different $\epsilon = 0.1, 0.03, 0.01$. When $\epsilon$ is small, the DGM fails to approximate the solution under given sampling stages $s_1 = 5000$.

The following reasons together result in this failure:

a. The Xavier initializer gives Var$(\theta) = 1/n$ and $E(\theta) = 0$. So the fourth order term $\epsilon u_{xxxx}$ modeled by $\epsilon(\theta^4\sigma^4)^L$ leads to a gradient vanishing problem at the beginning.

b. The number of sampling nodes is insufficient to capture the interface. In each suboptimization problem, only about $\epsilon N_e/2\pi$ nodes are around the interface. It follows that the parameters are updated slowly.

c. It is a non-convex optimization, and the learning rate needs to be small, which causes the parameters to stay in an incorrect interval for a long time.

From Table 3, we realize that in spite of changing penalty coefficients is able to improve the training effect, there still exists essential difficulties in the DGM for solving the Cahn-Hilliard equation. [33] proposes an adaptive strategy which has improved the accuracy of PINN in solving the Allen-Cahn and Cahn-Hilliard equations while the

LDGM provides another feasible way to solve the Cahn-Hilliard equation. Since the order of derivatives has been reduced in the LDGM, this algorithm is less affected by the gradient vanishing problem caused by initialization, as well as the influence of $\epsilon$. In general, the robustness of the LDGM is better than that of the DGM for solving the Cahn-Hilliard equation.

### 5.4. Modified KdV equation

In this test, we show that when the neural network becomes deeper, the parameter scale difference between different order derivatives becomes more obvious. Consider the following modified Korteweg-de Vries equation:

$$\begin{cases} u_t - 6u^2 u_x + u_{xxx} = 0, & x \in [-2,2], \quad t \in [0,1], \\ u(2,t) = \tanh(2t+1), \quad u(-2,t) = \tanh(2t-3), & t \in [0,1], \\ u(x,0) = \tanh(x-1), & x \in [-2,2]. \end{cases} \tag{5.6}$$

The kink solution of problem (5.6) is $u(x,t) = \tanh(x+2t-1)$. Set up the neural network

$$\varphi(x,t;\theta) = \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{pmatrix} \approx \begin{pmatrix} u \\ u_x \\ u_{xx} \end{pmatrix}, \tag{5.7}$$

and the loss function is given as follows:

$$\begin{aligned} J_e(\varphi) &= \left\| (\varphi_1)_t - 6\varphi_1^2 \varphi_2 + (\varphi_3)_x \right\|_{\Omega_T}^2 + \sum_{i=1}^{2} \left\| (\varphi_i)_x - \varphi_{i+1} \right\|_{\Omega_T}^2, \\ J_i(\varphi) &= \left\| \varphi_1 - \tanh(x-1) \right\|_{\Omega}^2, \\ J_b(\varphi) &= \left\| \varphi_1(-2,t) - \tanh(2t-3) \right\|_{[0,1]}^2 + \left\| \varphi_1(2,t) - \tanh(2t+1) \right\|_{[0,1]}^2. \end{aligned} \tag{5.8}$$

Under the default settings in Fig. 7, the only proven superiority of the LDGM is that it costs less time when solving the problem. We then compare the performance of different network settings in solving the modified KdV equation. The relative $L_2$ errors after 25000 steps training are shown in Table 4.

Table 4: Relative L2 errors under various network coefficients.

| $n = 10$,various $L$ | 3 | 6 | 9 | 12 | 24 | 48 |
|---|---|---|---|---|---|---|
| DGM | 0.132% | 0.099% | 0.155% | 0.171% | 0.579% | 84.22% |
| LDGM | 0.196% | 0.164% | 0.165% | 0.142% | 0.098% | 0.243% |
| $L = 3$,various $n$ | 10 | 20 | 40 | 80 | 160 | 320 |
| DGM | 0.132% | 0.059% | 0.060% | 0.209% | 0.027% | 1.694% |
| LDGM | 0.196% | 0.304% | 0.174% | 0.210% | 0.191% | 0.342% |

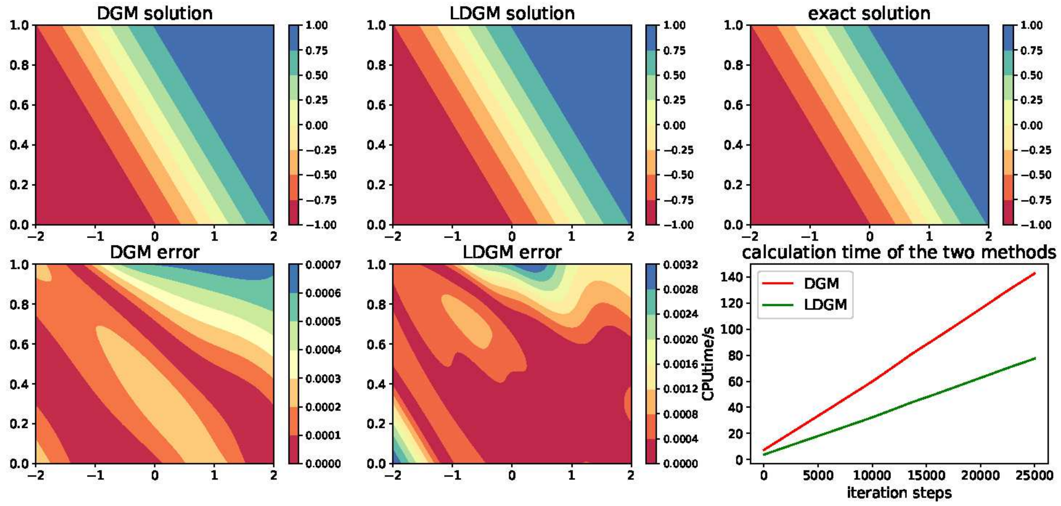


Figure 7: Deep learning solutions of the modified KdV equation obtained by the DGM and the LDGM. Under the default settings, the LDGM is faster than the DGM.

Table 4 shows that the LDGM is less affected by network settings, while the DGM is more sensitive. When $L = 48, n = 10$, the DGM fails to solve the KdV equation under the default settings. One reason of the failure is that the active function $\tanh(x)$ can cause gradient vanishing problem. But under the continuity assumption of the DGM, the active function of the neural network does not have many choices. In the LDGM, it only requires $\sigma(x) \in \mathbb{C}^2$. Here the exponential linear units

$$\sigma(x) = \begin{cases} x, & x > 0, \\ \alpha(e^x - 1), & \text{otherwise} \end{cases} \tag{5.9}$$

can be used to further reduce errors and avoid gradient vanishing problem. When $L = 64$, we repeat the experiment $100$ times and record whether the relative $L_2$ error is less than $1\%$ after $10000$ iteration steps. The success rates of the DGM and the LDGM are $30\%$ and $89\%$, respectively. This means that when the network is deep, the DGM is almost ineffective, while the LDGM provides more flexible choices in terms of network structures and active functions.

## 5.5. High-dimensional heat equation

Innumerable previous studies have shown that deep learning methods have distinct advantages in solving high-dimensional problems. In this example, we show that the LDGM inherits these advantages. Consider the general heat equation

$$\begin{aligned} &u_t - \Delta u = f(x,t), && x \in \Omega = [0,1]^d, && t \in [0,1], \\ &u(x,t) = g(x,t), && x \in \partial\Omega, && t \in [0,1], \end{aligned} \tag{5.10}$$

where

$$f(x,t)=2d(t+1)+\sum_{i=1}^{d}x_i(1-x_i),\quad g(x,t)=\sum_{i=1}^{d}x_i(1-x_i)(t+1).$$

Given the initial condition

$$u_0(x)=\sum_{i=1}^{d}x_i(1-x_i),$$

problem (5.10) has a classical solution

$$u^*(x,t)=\sum_{i=1}^{d}x_i(1-x_i)(t+1).$$

Define the multi-output neural network as follows:

$$\varphi(x,t;\theta)=\begin{pmatrix}\varphi_1\\ \varphi_2\\ \vdots\\ \varphi_{d+1}\end{pmatrix}\approx\begin{pmatrix}u\\ u_{x_1}\\ \vdots\\ u_{x_d}\end{pmatrix}. \tag{5.11}$$

The loss function has the form of

$$\begin{aligned}&J_e(\varphi)=\left\|(\varphi_1)_t-\sum_{i=1}^{d}(\varphi_{i+1})_{x_i}-f\right\|_{\Omega_T}^2+\sum_{i=1}^{d}\|(\varphi_1)_{x_i}-\varphi_{i+1}\|_{\Omega_T}^2,\\ &J_i(\varphi)=\|\varphi_1-u_0\|_{\Omega}^2,\quad J_b(\varphi)=\|\varphi_1-g\|_{\partial\Omega_T}^2.\end{aligned} \tag{5.12}$$

Notice that we only need to output the first order derivatives of all dimensions for computing the second order derivatives, which greatly saves calculation and storage space. The same strategy applies to higher order derivatives.

Setting $d=5$, $L=3$, $n=50$, $r=0.001$ and using $50000$ iteration steps ($s_1=10000$ and $s_2=5$), the solutions are given in Fig. 8.

Both the LDGM and the DGM need to learn more details in optimization. The error is caused by the limited approximation ability of such a neural network and training set. The default settings is not sufficient to cover the entire region $\Omega^d$. For a more precise solution, adding hidden layers, expanding the network's width and increasing the number of sampling nodes are all viable options. Fig. 9 uses the error curve to provide a more intuitive comparison. High-dimensional second order derivatives calculated by the automatic differentiation in the loss are harder to be optimized. Sometimes, an adaptive piecewise learning rate like $r\sim 10^{-[\log(k)]}$ is chosen, where $k$ is the iteration step. It always works but it can be costly. In the LDGM, the error drops quickly, which means that the LDGM retains its advantages in solving high-dimensional problems.

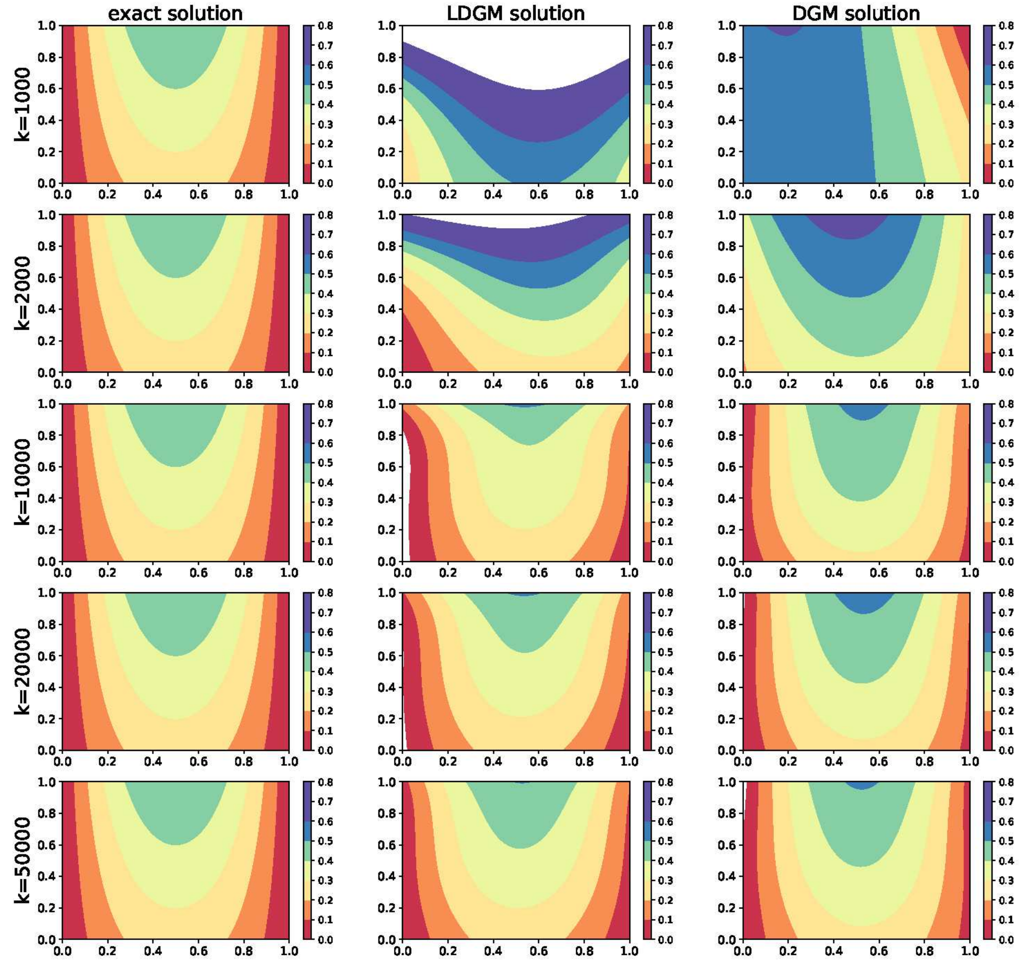


Figure 8: The training process of the high-dimensional heat equation obtained by DGM and LDGM with respect to dimensions $d = 5$. The picture shows the exact solution, the solution of DGM and the solution of LDGM from left to right. From top to bottom, it shows the solutions when the iteration steps $k = 1000, 2000, 10000, 20000, 50000$. For display purposes, the images show the slices of $x_2, x_3, \cdots, x_d = 0$. The abscissa is $x_1$ and the ordinate is $t$.

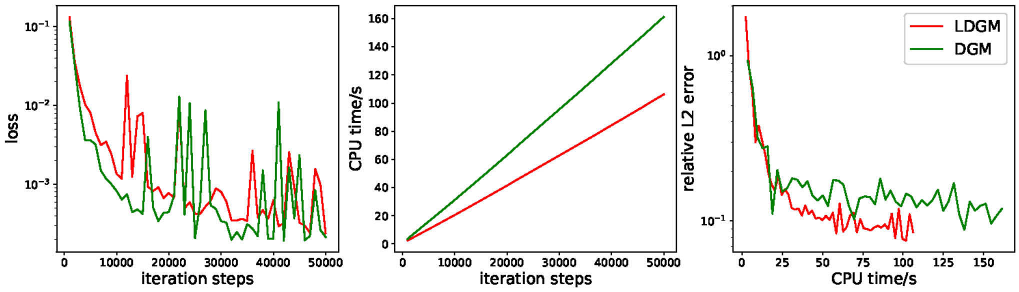


Figure 9: The iteration curves of the DGM and the LDGM for the 5-D heat equation. From left to right: loss vs. iteration steps, relative $L_2$ error vs. iteration steps, and relative $L_2$ error vs. calculation time, respectively.

Table 5: The relative L2 error of DGM and LDGM in different dimensions.

| $d$ | 2 | 10 | 20 | 50 | 100 |
|---|---|---|---|---|---|
| DGM | 0.070% | 0.082% | 0.121% | 0.178% | 0.282% |
| LDGM | 0.081% | 0.072% | 0.093% | 0.130% | 0.134% |

In addition, giving the exact solution

$$u(x,t) = e^{-t}\sum_{i=1}^{d}\sin x$$

and using $25000$ iteration steps, we compare the performance of LDGM in different dimensions. As the dimension changing, we show the relative L2 error of DGM and LDGM in Table 5. It implies that compared to the DGM, LDGM is less affected by the dimensional growth.

## 6. Concluding remarks and declaration

In this paper, we list the difficulties associated with computing high order derivatives of neural networks for solving PDEs. Calculating high order derivatives is costly and can cause a parameter scaling problem. In addition, complex calculations limit our choices of network structures and active functions. We propose a local deep learning method to overcome these problems. We consider the derivatives of the solution as intermediate variables and rewrite the original problem as a system of low order PDEs. The loss function takes the residual of the equivalent system. With a multi-output neural network, the local deep learning method is established. We demonstrate the performance of the local deep Galerkin method on a variety of PDEs, including high-dimensional problems, phase field problems and high order PDEs. In all numerical tests, the local deep Galerkin method is shown to be both stable and highly efficient.

**Declaration:** We enclose this paper by a declaration to show the originality of our work. Our project started more than one and half a years ago. About finishing the work by the end of 2020, we found the work [24] on arxiv post in June 2020. The exactly same technique is adopt to solve high-order PDEs by deep learning. They called it as deep mixed residual method, in contrast called as local deep learning method in our paper. Next we try to show the independence and originality of our work by shortly presenting how we propose the so-called local deep leaning method and the difference between our work with [24]. First, our motivation of this study was to use deep learning methods to solve phase field equations. While we successfully solved the Allen-Cahn equation, we failed to solve the Cahn-Hilliard equation, even though they are both gradient flow problems associating with same free energy. We later discovered that our failure to solve the CH equation was due to the high order derivative and small $\epsilon$ causing a parameter scaling problem. Thus the LDLM was proposed. Secondly, [24] focuses more on how the technique results in more accurate solutions as well as more accurate derivatives and the authors intend to enforcing exact boundary and initial conditions [23]. But we focus more on how this technique reduces the computations, improves the robustness for high order differential equations, and overcomes the gradient vanishing or exploding problem in high order PDEs.

## Acknowledgements

This work is partially supported by the National Natural Science Foundation of China/Hong Kong RRC Joint Research Scheme (NSFC/RGC 11961160718), and the fund of the Guangdong Provincial Key Laboratory of Computational Science and Material Design (No. 2019B030301001). The work of J. Yang is supported by the National Science Foundation of China (NSFC-11871264) and the Guangdong Basic and Applied Basic Research Foundation (2018A0303130123).

## References

[1] M. Abadi, et al., *Tensorflow: A system for large-scale machine learning*, In: 12th USENIX symposium on operating systems design and implementation (OSDI 16), (2016), 265–283.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: A survey*, J. Mach. Learn. Res. 18 (2017), 5595–5637.

[3] Y. Bengio, P. Simard, and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult*, IEEE Trans. Neural Netw. Learn. Syst. 5 (1994), 157–166.

[4] J. Chen, R. Du, and K. Wu, *A comparison study of deep Galerkin method and deep Ritz method for elliptic problems with different boundary conditions*, Commun. Math. Res. 36 (2020), 354–376.

[5] B. Cockburn and C.-W. Shu, *The local discontinuous Galerkin method for time-dependent convection-diffusion systems*, SIAM J. Numer. Anal. 35 (1998), 2440–2463.

[6] T. Dockhorn, *A discussion on solving partial differential equations using neural networks*, arXiv:1904.07200, 2019.

[7] W. E, *Machine learning and computational mathematics*, Commun. Comput. Phys. 28 (2020), 1639–1670.

[8] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, In: Proceedings of the thirteenth international conference on artificial intelligence and statistics, (2010), 249–256.

[9] R. Grosse, *Lecture 15: Exploding and vanishing gradients*, University of Toronto Computer Science, 2017.

[10] J. Han, J. Arnulf, and E. Weinan, *Solving high-dimensional partial differential equations using deep learning*, Proceedings of the National Academy of Sciences, (2018), 201718942.

[11] J. Han, M. Nica, and A. R. Stinchcombe, *A derivative-free method for solving elliptic partial differential equations with deep neural networks*, J. Comput. Phys. 419 (2020), 109672.

[12] B. Hanin, *Which neural net architectures give rise to exploding and vanishing gradients?*, In: Advances in Neural Information Processing Systems, (2018), 582–591.

[13] M. HAYATI AND B. KARAMI, *Feedforward neural network for solving partial differential equations*, J. Appl. Sci. 7 (2007), 2812–2817.

[14] S. HOCHREITER, *The vanishing gradient problem during learning recurrent neural nets and problem solutions*, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6 (1998), 107–116.

[15] K. HORNIK, *Approximation capabilities of multilayer feedforward networks*, Neural Netw. 4 (1991), 251–257.

[16] K. HORNIK, M. STINCHCOMBE, AND H. WHITE, *Multilayer feedforward networks are universal approximators*, Neural Netw. 2 (1989), 359–366.

[17] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, arXiv:1412.6980, 2014.

[18] I. E. LAGARIS, A. LIKAS, AND D. I. FOTIADIS, *Artificial neural networks for solving ordinary and partial differential equations*, IEEE Trans. Neural Netw. Learn. Syst. 9 (1998), 987–1000.

[19] I. E. LAGARIS, A. C. LIKAS, AND D. G. PAPAGEORGIOU, *Neural-network methods for boundary value problems with irregular boundaries*, IEEE Trans. Neural Netw. Learn. Syst. 11 (2000), 1041–1049.

[20] Y. LIAO AND P. MING, *Deep Nitsche method: Deep Ritz method with essential boundary conditions*, Commun. Comput. Phys. 29 (2021), 1365–1384.

[21] Z. LIU, W. CAI, AND Z.-Q. JOHN XU, *Multi-scale deep neural network (mscalednn) for solving Poisson-Boltzmann equation in complex domains*, Commun. Comput. Phys. 28 (2020), 1970–2001.

[22] L. LU, X. MENG, Z. MAO, AND G. KARNIADAKIS, *Deepxde: A deep learning library for solving differential equations*, SIAM Review 63 (2021), 208–228.

[23] L. LYU, K. WU, R. DU, AND J. CHEN, *Enforcing exact boundary and initial conditions in the deep mixed residual method*, arXiv:2008.01491, 2020.

[24] L. LYU, Z. ZHANG, M. CHEN, AND J. CHEN, *Mim: A deep mixed residual method for solving high-order partial differential equations*, arXiv:2006.04146, 2020.

[25] J. MÜLLER AND M. ZEINHOFER, *Error estimates for the variational training of neural networks with boundary penalty*, arXiv:2103.01007, 2021.

[26] F. I. NIORDSON, *On the optimal design of a vibrating beam*, Quart. Appl. Math. 23 (1965), 47–53.

[27] R. PASCANU, T. MIKOLOV, AND Y. BENGIO, *On the difficulty of training recurrent neural networks*, In: International conference on machine learning, 2013, 1310–1318.

[28] M. RAISSI, P. PERDIKARIS, AND G. KARNIADAKIS, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, J. Comput. Phys. 378 (2019), 686–707.

[29] J. SIRIGNANO AND K. SPILIOPOULOS, *Dgm: A deep learning algorithm for solving partial differential equations*, J. Comput. Phys. 375 (2017), 1339–1364.

[30] B. WANG, W. ZHANG, AND W. CAI, *Multi-scale deep neural network (mscalednn) methods for oscillatory Stokes flows in complex domains*, arXiv:2009.12729, 2020.

[31] E. WEINAN, J. HAN, AND A. JENTZEN, *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations*, Commun. Math. Stat. 5 (2017), 349–380.

[32] E. WEINAN AND T. YU, *The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*, Commun. Math. Stat. 6 (2017), 1–12.

[33] C. L. WIGHT AND J. ZHAO, *Solving Allen-Cahn and Cahn-Hilliard equations using the adaptive physics informed neural networks*, Commun. Comput. Phys. 29 (2021), 930–954.

[34] Y. Xu and C.-W. Shu, *Local discontinuous Galerkin methods for high-order time-dependent partial differential equations*, Commun. Comput. Phys. 7 (2010), 1.

[35] J. Yan and C.-W. Shu, *A local discontinuous Galerkin method for KDV type equations*, SIAM J. Numer. Anal. 40 (2002), 769–791.

[36] J. Yan and C.-W. Shu, *Local discontinuous Galerkin methods for partial differential equations with higher order derivatives,* J. Sci. Comput. 17 (2002), 27–47.

[37] Y. Zang, G. Bao, X. Ye, and H. Zhou, *Weak adversarial networks for high-dimensional partial differential equations*, J. Comput. Phys. 411 (2020), 109409.