Numerical approximation based on deep convolutional neural network for high-dimensional fully nonlinear merged PDEs and 2BSDEs

Xu Xiao1, Wenlin Qiu1, Omid Nikan2

1Key Laboratory of Computing and Stochastic Mathematics (Ministry of Education), School of Mathematics and Statistics, Hunan Normal University, Changsha, Hunan, China
2School of Mathematics and Computer Science, Iran University of Science and Technology, Tehran, Iran

Correspondence
Wenlin Qiu, Key Laboratory of Computing and Stochastic Mathematics (Ministry of Education), School of Mathematics and Statistics, Hunan Normal University, Changsha, Hunan 410081, China. Email: qwllkx12379@163.com

Funding information
Postgraduate Scientific Research Innovation Project of Hunan Province, Grant/Award Number: CX20220454

INTRODUCTION

Nonlinear partial differential equations (NPDEs) play a key role in a large number of models, from finance to physics [1–4]. Objects such as wave functions related to quantum physical systems, value functions that depict the fair prices of financial derivatives in pricing models, or value functions which depict the expected maximum utility in portfolio optimization problems that are usually presented as the solutions of NPDEs.

Roughly speaking, the nonlinearity in PDEs used in financial engineering above is derived from the trade mix (the trade mix and utility of hedging financial derivatives claims in the case of the derivatives pricing problem must be maximized in the case of the portfolio optimization problem). Bender et al. [5] and Gobet et al. [6] adopted derivative pricing models with distinguishing lending rates. Crépey et al. [7] considered derivative pricing models incorporating the default risk of the issuer of the financial derivative. Bayraktar and Young [8] proposed the models for the pricing of financial derivatives on untradable underlyings and analyzed, for example, financial derivatives on the temperature or mortality-dependent
financial derivatives. Amadori [9] considered the models incorporating that the trading strategy affects the price processes though the demand and supply.

The resulting PDEs from these models are usually high dimensional, since the associated trading portfolio often involves a whole basket of financial assets (see [5, 7]). These high-dimensional NPDEs are often exceedingly difficult to be solved approximately. Furthermore, due to the practical relevance of the aforementioned PDEs, there is a strong demand in the financial engineering industry to approximation solutions to such high-dimensional nonlinear parabolic PDEs.

There are lots of numerical approaches for solving parabolic NPDEs approximatively in the literature, from which some of these methods are deterministic approximations, while others are stochastic approximations that depend on appropriate probabilistic representations of the corresponding PDE solutions, for example, probabilistic representations in view of backward stochastic differential equations (BSDEs) (see [10, 11]), probabilistic representations in view of second-order backward stochastic differential equations (2BSDEs) (see [12]), probabilistic representations in view of branching diffusions (see [13]), and probabilistic representations in view of extensions of the classical Feynman–Kac formula (see [14]). Then, we can refer to some articles specifically, for example, deterministic approximation approaches for PDEs (see [15, 16]), probabilistic approximation approaches for PDEs based on time discretizations of BSDEs (see [5, 6, 17–36]), probabilistic approximation approaches for PDEs in view of suitable deep learning approximations for BSDEs (see [37, 38]), probabilistic approximation approaches for BSDEs in view of Wiener chaos expansions (see [39]), probabilistic approximation approaches for BSDEs in view of sparse grid approximations (see [40]), probabilistic approximation approaches for PDEs based on branching diffusion representations (see [13, 41]), and probabilistic approximation approaches for PDEs in view of time discretization of 2BSDEs (see [12, 42–44]).

However, most of the above approximation techniques are only applicable when the dimension $d$ of PDEs/BSDEs is quite small or only when there are strict constraints on the parameters or the type of PDEs considered (e.g., small nonlinearities, small terminal/initial conditions, and the semilinear structure of PDEs). Therefore, to yield the numerical solutions of high-dimensional nonlinear PDEs, this is still an exceedingly difficult task, and there are only a few cases where practical algorithms for high-dimensional PDEs can be considered (see [13, 37, 38, 45]). Especially, to our knowledge, few practical algorithms for high-dimensional fully nonlinear parabolic PDEs currently exist in the scientific literature.

In recent years, deep learning-based methods have been developed to solving these high-dimensional PDEs; see, for example, [37, 38, 46–51]. The fundamental principle underlying the deep learning approach is to represent certain functions within the PDE using neural networks. Subsequently, an approximate solution to the PDE is obtained by minimizing the PDE residual [51]. Since it has the advantage of automatic differentiation and could break the curse of dimensionality [52], the method based on deep learning has received extensive attention. Among these approaches, some scholars choose to employ the variational formulation of PDEs and minimize the associated energy functional [46, 48]. In [49], a strong form of the PDE is introduced, coining the term physics-informed neural networks (PINNs). In this form, direct employment of automatic differentiation can circumvent issues related to truncation errors and numerical quadrature errors associated with variational formulations. In addition, some can only be applied to particular types of problems, like parabolic PDEs [37, 50].

This paper intends to solve this difficulty and present new results, that is, we solve the fully nonlinear merged PDEs and 2BSDEs with a new algorithm. Regarding the proposed problem, Beck et al. [50] first consider that by utilizing some properties from Peng’s nonlinear expectation in high-dimensional space (see [53]). The proposed algorithm uses a connection between PDEs and 2BSDEs (see Cheridito et al. [12]) to yield a merged formulation of PDEs and 2BSDEs, whose approximated solutions can be obtained via combining time discretizations with a neural network (NN) based on deep learning (see [15, 16, 37, 38, 54–57]). Loosely speaking, the merged formulation allows us to establish the original partial differential problem as a learning problem. The random loss function for the deep neural network in our method can be given by the error between the prescribed terminal condition of 2BSDEs and the neural network in view of forward time discretization of 2BSDEs. In fact, a corresponding deep learning approximation algorithm for semilinear-type PDEs in view of forward BSDEs has been recently considered in [37, 38]. A crucial distinction between [37, 38] and our work is that herein we depend on the connection between fully nonlinear PDEs and 2BSDEs given in [12], while [37, 38] depend on the almost classical combination between PDEs and BSDEs (see [10, 11]). Besides, although Beck et al. [50] have considered the merged construction of fully nonlinear PDEs and 2BSDEs, there is still room for improvement. Under the limitation of computer memory, since they only consider linear neural networks, they can only calculate general high-dimensional nonlinear parabolic problems and cannot calculate higher dimensional problems (e.g., more than 200 dimensions), and further, the approximated error can also be reduced in terms of computational accuracy. These inspired us to carry out the following research.
The main contributions of this work are as follows: (i) We improve the method of Beck et al. [50] in order to further improve the accuracy of the solution. We apply multiscale fusion technology [58–60] to the original neural network model, that is, use different scales to spatially discretize it, and finally, use the merged results. This paper currently uses four scales for fusion, (ii) we also generalize the approach in [50] so that higher dimensional models can be solved. The method of [50] is to spatially discretize the time-discrete data in the form of vectors. We first arrange the time-discrete data into a matrix and then use the convolutional neural networks [61, 62] for spatial discretizations. From the experimental results, the dimension of the solution is further expanded, and the time spent is also shorter. At present, we mainly enumerate numerical experiments in 256 and 400 dimensions, (iii) we mainly solve three practical high-dimensional examples, which possess the significant physical background, namely, the Allen–Cahn (AC), the Hamilton–Jacobi–Bellman (HJB), and the Black–Scholes–Barenblatt (BSB) equations. The numerical results can demonstrate the effectiveness of the proposed approximation method and (iv) the proposed strategy considers advanced optimization algorithms, that is, Adaptive Moment Estimation (Adam) optimizer and stochastic gradient descent (SGD)-type optimization.

Following these ideas, the organizational structure of this work is as follows. Section 2 introduces the merged construction of PDEs and 2BSDEs in [45, 50]. Section 3 presents the forward temporal discretizations of the merged PDEs–2BSDEs system, spatial discretizations based on multiscale deep learning fusion and convolutional neural network, respectively, and corresponding optimization algorithms. Section 4 reports some experiments for numerical solutions of the merged PDEs–2BSDEs system, concretely, containing the high-dimensional AC, BSB, and HJB equations. Finally, Section 5 summarizes the concluding remarks.

2 | MERGED PDES–2BSDES SYSTEM

This section mainly intends to obtain a merged PDEs–2BSDEs system. The merged PDEs–2BSDEs system was proposed by the work of Beck et al. [50], which we briefly review here. First, we introduce the fully nonlinear second-order PDEs. Besides, Table 1 summarizes some notions and notations used in this paper.

2.1 | Fully nonlinear parabolic second-order PDEs

Let $d \in \mathbb{Z}^+$, $0 < T < \infty$, $u = (u(t, x))_{0 \leq t \leq T, x \in \mathbb{R}^d} \in C^{1,3} \left( [0, T] \times \mathbb{R}^d, \mathbb{R} \right)$. A general form of the fully nonlinear second-order parabolic PDEs could be described by
where $F$ is a fully nonlinear function (no specific type) that $F : [0, T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d} \rightarrow \mathbb{R}$ with terminal condition

$$u(T, x) = \hat{g}(x), \quad x \in \mathbb{R}^d. \tag{2}$$

It is important to emphasize that (2) is a terminal condition, different from the initial conditions commonly used in the context of PDEs literature. Hence, Equation (1) is formulated as a terminal value problem. On the one hand, the terminal value problem seems to be more naturally associated with BSDEs (see Section 2.2), and on the other hand, the terminal value problem naturally appears in financial engineering applications such as the BSB equation in derivatives pricing (see Section 4.2). In addition, our method does not have a limit regarding the boundary conditions; see [50] for similar cases.

Obviously, terminal value problems can be transformed into initial value problems and vice versa. Furthermore, the initial value problem can be written as follows: Assume $\hat{F} : [0, T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d} \rightarrow \mathbb{R}$ and $V : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R}$ are the functions such that $V(t, x) = u(T - t, x)$ and

$$\hat{F}(t, x, y, z, \rho) = -F(T - t, x, y, z, \rho), \tag{3}$$

for all $(t, x, y, z, \rho) \in [0, T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^{d \times d}$. Then, we get that $V : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R}$ is a continuous function, such that $V(0, x) = \hat{g}(x), \quad \forall \{0, T\} \times \mathbb{R} \in C^{1,3} \left( (0, T] \times \mathbb{R}^d, \mathbb{R} \right)$ and

$$\frac{\partial V}{\partial t}(t, x) = \hat{F}(t, x, V(t, x), (\nabla_x V)(t, x), (\text{Hess}_x V)(t, x)), \tag{4}$$

for all $(t, x) \in (0, T] \times \mathbb{R}^d$. Based on the above discussion, in the following numerical examples, we only consider the terminal problem.

In addition, we employ the deep learning method approximation $u(0, x)$. In other words, the goal of this paper is to seek the approximate solution $u$ of the PDE (1) at time $t = 0$. For $u(t, x), t \in \{1, \cdots, N\}$, it can be computed recursively by using the deep learning approach (more details, see Section 3.2 or 3.3).

### 2.2 Combination between fully nonlinear parabolic second-order PDEs and 2BSDEs

The proposed deep learning method depend on a combination between fully nonlinear second-order PDEs and 2BSDEs (see the following Lemma 1), from which, Itô’s lemma and some suitable assumptions are employed (see [50]).

**Lemma 1** ([50, Lemma 3.1]). Let $d \in \mathbb{Z}^+$, $0 < T < \infty$, $u \in C^{1,3} \left( (0, T] \times \mathbb{R}^d, \mathbb{R} \right)$, $\nabla_x u \in C^{1,2} \left( (0, T] \times \mathbb{R}^d, \mathbb{R}^d \right)$, $\text{Hess}_u \in C^{1,1} \left( (0, T] \times \mathbb{R}^d, \mathbb{R}^{d \times d} \right)$, and $u$ is the solution of the PDE (1) with terminal condition (2). Then, assume that $(\Omega, \mathcal{F}, \mathbb{P})$ is a probability space, that $\mathcal{W} = (\mathcal{W}^{(1)}, \ldots, \mathcal{W}^{(d)}) : [0, T] \times \Omega \rightarrow \mathbb{R}^d$ is a standard Brownian motion on $(\Omega, \mathcal{F}, \mathbb{P})$, that $\mathcal{F} = (\mathcal{F}_t)_{t \in [0, T]}$ is the normal filtration on $(\Omega, \mathcal{F}, \mathbb{P})$ generated via $\mathcal{W}$, that $\xi : \Omega \rightarrow \mathbb{R}^d$ is a $\mathcal{F}_0$-B$(\mathbb{R}^d)$-measurable function, and that $\mathcal{X} = (\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(d)}) : [0, T] \times \Omega \rightarrow \mathbb{R}^d$ is an $\mathbb{F}$-adapted stochastic process, with continuous sample paths such that for all $0 \leq t \leq T$, it holds $\mathbb{P}$-a.s. that

$$\mathcal{X}_t = \xi + \int_0^t \mu(\mathcal{X}_s) \, ds + \int_0^t \sigma(\mathcal{X}_s) \, d\mathcal{W}_s, \tag{5}$$

where $\mu : \mathbb{R}^d \rightarrow \mathbb{R}^d$, $\sigma : \mathbb{R}^d \rightarrow \mathbb{R}^{d \times d}$ are continuous functions. And define the function $\mathcal{L} \sigma : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that

$$\mathcal{L} \sigma(t, x) = \left( \frac{\partial \sigma}{\partial t} \right) (t, x) + \frac{1}{2} \text{Trace} (\sigma(\mathcal{X})(\sigma(\mathcal{X}))^* (\text{Hess}_x \sigma)(t, x)), \quad (t, x) \in [0, T] \times \mathbb{R}^d, \tag{6}$$

for all $\sigma \in C^{1,3} \left( (0, T] \times \mathbb{R}^d, \mathbb{R} \right)$. And let $\mathcal{Y} : [0, T] \times \Omega \rightarrow \mathbb{R}$, $\mathcal{Z} = (\mathcal{Z}^{(1)}, \ldots, \mathcal{Z}^{(d)}) : [0, T] \times \Omega \rightarrow \mathbb{R}^d$, $\Gamma = (\Gamma^{(i)}_{(j)})_{i,j \in [1, \ldots, d]} : [0, T] \times \Omega \rightarrow \mathbb{R}^{d \times d}$, and let $\mathcal{A} = (\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(d)}) : [0, T] \times \Omega \rightarrow \mathbb{R}^d$ be the stochastic processes, such that

$$\mathcal{Y}_t = u(t, \mathcal{X}_t), \quad \mathcal{Z}_t = (\nabla_x u)(t, \mathcal{X}_t), \quad \Gamma_t = (\text{Hess}_x u)(t, \mathcal{X}_t), \quad \mathcal{A}^{(i)}_t = \left( \mathcal{L} \left( \frac{\partial u}{\partial \mathcal{X}^i} \right) \right)(t, \mathcal{X}_t), \tag{7}$$

where $u(t, x)$, $\mathcal{X}_t$, $\mathcal{Y}_t$, $\mathcal{Z}_t$, $\mathcal{A}_t$, $\mathcal{A}^{(i)}_t$ are deterministic functions.
for all $0 \leq t \leq T$ and $i \in \{1, 2, \ldots, d\}$. Then, we obtain that $\mathcal{Y}, \mathcal{Z}, \Gamma, A$ are $\mathbb{F}$-adapted stochastic processes, with continuous sample paths which satisfy that for all $0 \leq t \leq T$, it holds $\mathbb{P}$-a.s. that

$$
\mathcal{Y}_t = \mathcal{g}(\mathcal{X}_T) - \int_0^T \left( F(s, \mathcal{X}_s, \mathcal{Y}_s, \mathcal{Z}_s, \Gamma_s) + \frac{1}{2} \text{Trace} \left( \sigma(\mathcal{X}_s) \sigma(\mathcal{X}_s)^\top \Gamma_s \right) \right) ds - \int_0^T \langle \mathcal{Z}_s, d\mathcal{W}_s \rangle_{\mathbb{R}^d}
$$

and

$$
\mathcal{Z}_t = \mathcal{Z}_0 + \int_0^T \mathcal{A}_s ds + \int_0^T \Gamma_s d\mathcal{W}_s.
$$

Lemma 1 shows that under certain conditions, the existence of solution to the PDE (1) implies the existence of a solution to the 2BSDE (8), (9), and $\mathcal{Y}_t = u(t, \mathcal{X}_t)$. This provides a stochastic representation for solutions of fully nonlinear parabolic PDEs.

### 2.3 Merged construction of PDEs and 2BSDEs

In what follows, we present a merged construction for PDE (1) and 2BSDE (8) and (9) system. Let the hypotheses in Lemma 1 be satisfied and use the same notations as Lemma 1. Then, one can easily see that for $0 \leq \delta_1, \delta_2 \leq T$,

$$
\mathcal{X}_{\delta_2} = \mathcal{X}_{\delta_1} + \int_{\delta_1}^{\delta_2} \mu(\mathcal{X}_s) ds + \int_{\delta_1}^{\delta_2} \sigma(\mathcal{X}_s) d\mathcal{W}_s.
$$

Substituting the third and fourth equations from (7) into (8) and (9) yields, for all $\delta_1 < \delta_2$, we have

$$
\mathcal{Y}_{\delta_2} = \mathcal{Y}_{\delta_1} + \int_{\delta_1}^{\delta_2} \langle \mathcal{Z}_s, d\mathcal{W}_s \rangle_{\mathbb{R}^d} + \int_{\delta_1}^{\delta_2} \left( F(s, \mathcal{X}_s, \mathcal{Y}_s, \mathcal{Z}_s, (\text{Hess}_x u)(s, \mathcal{X}_s)) + \frac{1}{2} \text{Trace} \left( \sigma(\mathcal{X}_s) \sigma(\mathcal{X}_s)^\top (\text{Hess}_x u)(s, \mathcal{X}_s) \right) \right) ds
$$

and

$$
\mathcal{Z}_{\delta_2} = \mathcal{Z}_{\delta_1} + \int_{\delta_1}^{\delta_2} (\mathcal{L}(\nabla_x u))(s, \mathcal{X}_s) ds + \int_{\delta_1}^{\delta_2} (\text{Hess}_x u)(s, \mathcal{X}_s) d\mathcal{W}_s.
$$

### 3 APPROXIMATION OF THE MERGED PDES–2BSDES SYSTEM

#### 3.1 Forward-discretizations of the merged PDEs–2BSDES system

Now, we describe a forward discretization of the merged PDEs–2BSDES system (11) and (12). Let us consider positive integer $N \geq 1$ with $t_0, t_1, \ldots, t_N \in [0, T]$, such that

$$
0 = t_0 < t_1 < t_2 < \ldots < t_N = T,
$$

from which the max mesh size $\tau := \max_{0 \leq j \leq N-1} (t_{j+1} - t_j)$ is sufficiently small and we define $t_j = t_j - t_{j-1}$ for $1 \leq j \leq N$.

Notice that, for sufficiently large $N \in \mathbb{Z}^+$, (5), (7), and (10)–(12) indicate that for all $n \in \{0, 1, \ldots, N-1\}$, it holds that

$$
\mathcal{X}_{t_n} = \mathcal{X}_0 = \xi, \quad \mathcal{Y}_{t_n} = \mathcal{Y}_0 = u(0, \xi), \quad \mathcal{Z}_{t_n} = \mathcal{Z}_0 = (\nabla_x u)(0, \xi),
$$

$$
\mathcal{X}_{t_{n+1}} \approx \mathcal{X}_{t_n} + \mu(\mathcal{X}_{t_n}) t_{n+1} + \sigma(\mathcal{X}_{t_n})(\mathcal{X}_{t_{n+1}} - \mathcal{X}_{t_n}),
$$

$$
\mathcal{Y}_{t_{n+1}} \approx \mathcal{Y}_{t_n} + \left[ F(t_n, \mathcal{X}_{t_n}, \mathcal{Y}_{t_n}, \mathcal{Z}_{t_n}, (\text{Hess}_x u)(t_n, \mathcal{X}_{t_n})) + \frac{1}{2} \text{Trace}(\sigma(\mathcal{X}_{t_n})\sigma(\mathcal{X}_{t_n})^\top(\text{Hess}_x u)(t_n, \mathcal{X}_{t_n})) \right] t_{n+1} + \langle \mathcal{Z}_{t_n}, \mathcal{X}_{t_{n+1}} - \mathcal{X}_{t_n} \rangle_{\mathbb{R}^d},
$$

$$
\mathcal{Z}_{t_{n+1}} \approx \mathcal{Z}_{t_n} + \int_{t_n}^{t_{n+1}} (\mathcal{L}(\nabla_x u))(s, \mathcal{X}_s) ds + \int_{t_n}^{t_{n+1}} (\text{Hess}_x u)(s, \mathcal{X}_s) d\mathcal{W}_s,
$$

$$
\mathcal{W}_{t_{n+1}} \approx \mathcal{W}_{t_n} + \int_{t_n}^{t_{n+1}} (\text{Hess}_x u)(s, \mathcal{X}_s) d\mathcal{W}_s.
$$
Spatial discretizations based on multiscale deep learning fusion

In the following, for all \(0 \leq n \leq N - 1\) and \(x \in \mathbb{R}^d\), we select suitable approximations for functions \((\text{Hess}_x u)(t_n, x) \in \mathbb{R}^{d \times d}\) and \((\mathcal{L}(\mathcal{V}_x u))(t_n, x) \in \mathbb{R}^d\) given in (15) and (16). It is worth noting that we don’t consider approximating the functions \(u(t_n, x) \in \mathbb{R}^d\) or \((\mathcal{V}_x u)(t_n, x) \in \mathbb{R}^d\). In fact, for the functions \((t_n, x), (\mathcal{V}_x u)(t_n, x)\), it can be computed recursively by using (15) and (16) and the approximations for \((\text{Hess}_x u)(t_n, x)\), \((\mathcal{L}(\mathcal{V}_x u))(t_n, x)\).

Precisely, we assume that \(\nu \in \mathbb{N} \cap [d + 1, \infty)\) for every \(\theta \in \mathbb{R}^\nu\). Assume \(G_n^\theta : \mathbb{R}^d \to \mathbb{R}^{d \times d}\) and \(A_n^\theta : \mathbb{R}^d \to \mathbb{R}^d\) are continuous functions, and then, for every \(\theta = (\theta_1, \theta_2, \ldots, \theta_d) \in \mathbb{R}^\nu\), assume \(Y_n^\theta : \{0, 1, \ldots, N\} \times \Omega \to \mathbb{R}\) and \(Z_n^\theta : \{0, 1, \ldots, N\} \times \Omega \to \mathbb{R}\) are stochastic processes, such that \(Y_n^\theta = \theta_1, Z_n^\theta = (\theta_2, \theta_3, \ldots, \theta_{d+1})\).

\[
Y_{n+1}^\theta = Y_n^\theta + \left(Z_n^\theta, X_{n+1} - X_n\right) \in \mathbb{R}^d + \left(F(t_n, X_n, Y_n^\theta, Z_n^\theta, G_n^\theta(X_n), + \frac{1}{2}\text{Trace}(G_n^\theta(X_n)))\right) \tau_{n+1}. \tag{17}
\]

and

\[
Z_{n+1}^\theta = Z_n^\theta + A_n^\theta(X_n) \tau_{n+1} + G_n^\theta(X_n) \left(X_{n+1} - X_n\right). \tag{18}
\]

for \(0 \leq n \leq N - 1\). In [50], \(\nu\) as the number of parameters in the neural network, and for all favorable \(\theta \in \mathbb{R}^\nu\), \(x \in \mathbb{R}^d\), and \(0 \leq n \leq N - 1\), the suitable approximations as \(Y_n^\theta \approx Y_n, Z_n^\theta \approx Z_n, G_n^\theta(x) \approx (\text{Hess}_x u)(t_n, x),\) and \(A_n^\theta(x) \approx (\mathcal{L}(\mathcal{V}_x u))(t_n, x)\). Especially, we regard \(\theta_1\) and \((\theta_2, \theta_3, \ldots, \theta_{d+1})\) as the suitable approximations of \(u(0, \xi)\) and \((\mathcal{V}_x u)(0, \xi)\) with \(u(0, \xi) \in \mathbb{R}\) and \((\mathcal{V}_x u)(0, \xi) \in \mathbb{R}^d\).

It is not difficult to find that each \(n \in \{0, \ldots, N - 1\}\), we need to consider a suitable neural network to approximate function \((\text{Hess}_x u)(t_n, x), (\mathcal{L}(\mathcal{V}_x u))(t_n, x)\). If \(n\) is large, the number of neural networks will be large, and the calculation will be cumbersome. For this problem, we use the same neural network for different time \(n\). In other words, we employ \(G_n^\theta(x) \approx (\text{Hess}_x u)(t_n, x), A_n^\theta(x) \approx (\mathcal{L}(\mathcal{V}_x u))(t_n, x)\) for all \(n \in \{0, \ldots, N - 1\}\). However, if the same neural network is used to approximate functions at different times, the error will be large.

Therefore, we consider the multiscale deep learning networks to approximate the functions \((\text{Hess}_x u)(t_n, x), (\mathcal{L}(\mathcal{V}_x u))(t_n, x)\) respectively. The multiscale neural network extracts data features from different levels or scales, enhancing its approximation to the functions \((\text{Hess}_x u)(t_n, x), (\mathcal{L}(\mathcal{V}_x u))(t_n, x)\). More formally, let \(G_n^\rho(x)\) and \(A_n^\rho(x)\) as deep neural networks. In particular, \(d_i\) represents the scale of the neural network, and four scales are selected here. In addition, the parameters of our network only depend on different scales, independent of time \(n\).

Assume \(\nu \geq \left(2 \sum_{i=1}^{d} d_i + d + 1\right)(d + 1) + \sum_{i=1}^{d} (2d_i + d^2 + d)(d_i + 1)\). Supposing for all \(\theta = (\theta_1, \ldots, \theta_d) \in \mathbb{R}^\nu, x \in \mathbb{R}^d\), we have

\[
G_n^\rho(x) = \begin{pmatrix}
\theta_{d+1} & \theta_{d+3} & \ldots & \theta_{2d+1} \\
\theta_{2d+1} & \theta_{2d+3} & \ldots & \theta_{3d+1} \\
\vdots & \vdots & \ddots & \vdots \\
\theta_{d^2+1} & \theta_{d^2+3} & \ldots & \theta_{d^2+2d+1}
\end{pmatrix} \in \mathbb{R}^{d \times d}
\]

and

\[
A_n^\rho(x) = \begin{pmatrix}
\theta & \theta_{d+2} & \ldots & \theta_{2d+2} \\
\theta_{d+2} & \theta & \ldots & \theta_{3d+2} \\
\vdots & \vdots & \ddots & \vdots \\
\theta_{d^2+2} & \theta_{d^2+2d} & \ldots & \theta_{d^2+2d+2}
\end{pmatrix} \in \mathbb{R}^d.
\]

With all \(k \in \mathbb{N}\), we let \(R_k : \mathbb{R}^k \to \mathbb{R}^k\) be the activation function (ReLU), such that

\[
R_k(x) = (Ax \{x_1, 0\}, \ldots, Ax \{x_k, 0\}). \tag{19}
\]

for every \(x = (x_1, \ldots, x_k) \in \mathbb{R}^k\). For every \(\theta = (\theta_1, \ldots, \theta_d) \in \mathbb{R}^\nu, v \in \mathbb{N}_0, k, l \in \mathbb{N}\), and \(v + k(l + 1) \leq \nu\), assume \(M_k^\rho_v : \mathbb{R}^l \to \mathbb{R}^l\) is the affine linear function, such that
for all $x = (x_1, \ldots, x_i)$. For every $\theta \in \mathbb{R}^v$, $\{d_i | i \in \{1, 2, 3, 4\}, d_0 = 0\}$, $d^i = \sum_{i=1}^{4} d_i$, $d^{2i} = \sum_{i=1}^{4} d_i^2 + d_i$, $d^{3i} = \sum_{i=1}^{4} (d_i + d)(d_i + 1)$, and $x \in \mathbb{R}^{d_i}$, we assume that

$$A_{d_i}^\theta = M_{o,R_{d_i}}^{\theta,(d^2 + d + 1)(d+1) + d^2 + \sum_{i=1}^{4} (d_i + 1)} \circ R_{d_i} \circ M_{d_i,d_i}^{\theta,(d_i + d + 1)}(\sum_{i=1}^{4} d_i)(d+1),$$

and that

$$G_{d_i}^\theta = M_{o,R_{d_i}}^{\theta,(d^2 + d + 1)(d+1) + d^2 + \sum_{i=1}^{4} d_i^2 + d_i}(\sum_{i=1}^{4} d_i + d + 1)^{(d+1)} + d^2 \circ R_{d_i} \circ M_{d_i,d_i}^{\theta,(d_i + d + 1)}(\sum_{i=1}^{4} d_i + d + 1)^{(d+1)} + d^2. \tag{22}$$

The crucial difference between our approximate function (21) and (22) from [50] is that we are considering the same approximation function (21) and (22) in the time domain. The proposed method reduces the amount of computation considering multiple approximate functions in [50]. In addition, we employ the multiscale fusion techniques to reduce the error of the same approximation function. On the other hand, the proposed method also ensures the accuracy of the obtained approximated solution, which is even higher than the method in [50].

**Remark 1.** In this remark, we illustrate the multiscale deep learning fusion and the specific choice of the $v \in \mathbb{N}$ in the above.

(i) Multiscale fusion is mainly reflected in function $A_{d_i}^\theta$ and $G_{d_i}^\theta$. We use deep neural networks of different scales to obtain $A_{d_i}^\theta$ and $G_{d_i}^\theta$, then fuse them to get the final result. In fact, multiscale fusion is to obtain more information in neural network training, thereby improving training results. In addition, if it is assumed that the scales selected each time are the same, the function will be $A_{d_i}^\theta, G_{d_i}^\theta$ ($d = d_1 = d_2 = d_3 = d_4$). This means that we approximate four times with the function $A_{d_i}^\theta, G_{d_i}^\theta$, which is equivalent to a weighted average of multiple experiments. From a probabilistic point of view, the results of multiple experiments are often more accurate and stable than the results of a single experiment.

(ii) For the specific choice of the $v$, the choice of $v$ is mainly divided into three parts. On the one hand, it is employed to approximate the variables we need, which includes the real number $u(0, \xi) \in \mathbb{R}$, the $(1 \times d)$ matrix $(\nabla_x u)(0, \xi)$, the $(d \times d)$ matrix $G_{d_i}^\theta$ and $(d \times 1)$ vector $A_{d_i}^\theta$. So, we have $v \geq (d + 1)(d + 1)$. On the other hand, the remaining two parts are related to neural networks, the first part is about $A_{d_i}^\theta$, and the last part is about $G_{d_i}^\theta$.

(iii) For the $A_{d_i}^\theta$, in each of the employed $d_i$ neural network, we use $d_i(d + 1)$ components of $\theta$ to describe the affine linear function from the $d$-dimensional first layer (input layer) to the $d_i$-dimensional second layer (includes a $d_i \times d$ matrix and a $d_i$ vector, see 20). Next, we use $d_i(d + 1)$ to describe the $d_i$-dimensional second layer to the $d_i$-dimensional third layer. Finally, the $d(d_i + 1)$ is used in the $d_i$-dimensional third layer to the $d_i$-dimensional fourth layer (output layer). For the $G_{d_i}^\theta$, the few layers are basically the same as $A_{d_i}^\theta$, the only difference is that the $d^2(d_i + 1)$ is used in the $d_i$-dimensional third layer to the $d_i$-dimensional fourth layer. Therefore, combining the above analysis, we have

$$v \geq (d + 1)(d + 1) + 2 \sum_{i=1}^{4} d_i(d + 1) + 2 \sum_{i=1}^{4} d_i(d_i + 1) + \sum_{i=1}^{4} d(d_i + 1) + \sum_{i=1}^{4} d_i(d_i + 1)$$

$$= \left(2 \sum_{i=1}^{4} d_i + d + 1\right)(d + 1) + \sum_{i=1}^{4} (2d_i + d^2 + d)(d_i + 1).$$
(iv) We also depict the sketch of the architecture of multiscale deep learning fusion; see Figure 1. In Figure 1, when \( t = t_0 \), we first give the initial values \( \mathcal{X}_0 \), \( (\mathcal{L}(\nabla_x u))(t_0, \mathcal{X}_0) \), \( (\text{Hess}_x u)(t_0, \mathcal{X}_0) \). Then, use the initial value to calculate the variables of \( t = t_i \) in turn \( (1 \leq i \leq N - 1) \), which \( h^{\text{fusion}}_d \) represents the \( H \) layer of the neural network at the \( d \) scale. As can be seen from the figure, each \( \mathcal{X}_i \) is trained by neural networks of four scales, and the \( h^{\text{fusion}} \) is obtained by the weighted average method. Specifically, \( h^{\text{fusion}} = \sum w_i h^{\text{fusion}}_d \), the \( w_i \) are weight functions with \( \sum w_i = 1 \). Note that for each time \( t = t_i \), we use the same neural network, which is continuously updated as time changes. In addition, \( (\mathcal{L}(\nabla_x u))(t_i, \mathcal{X}_i) \) and \( (\text{Hess}_x u)(t_i, \mathcal{X}_i) \) are approximated separately using two networks. In Figure 1, it is not subdivided for the sake of brevity.

### 3.3 Spatial discretizations based on convolutional neural network

Here, with \( 0 \leq n \leq N - 1 \) and \( x \in \mathbb{R}^d \), based on convolutional neural network, we still choose the suitable approximations for functions \( (\text{Hess}_x u)(t_n, x) \in \mathbb{R}^{d \times d} \) and \( (\mathcal{L}(\nabla_x u))(t_n, x) \in \mathbb{R}^d \). Then, let \( v \in \mathbb{N} \cap (d + 1, \infty) \) and \( \theta \) is assumed as Section 3.2 with \( 0 \leq n \leq N \).

Suppose that \( \tilde{G}_n^\theta : \mathbb{R}^d \rightarrow \mathbb{R}^{d \times d} \) and \( \tilde{A}_n^\theta : \mathbb{R}^d \rightarrow \mathbb{R}^d \) are continuous functions. For every \( \theta = (\theta_1, \theta_2, \ldots, \theta_d) \in \mathbb{R}^d \), assume \( \mathcal{Y}^\theta \) and \( \mathcal{Z}^\theta \) are denoted as before, which satisfy \( \mathcal{Y}^\theta_0 = \mathcal{X}_0 \), \( \mathcal{Z}^\theta_0 = (\theta_2, \theta_3, \ldots, \theta_{d+1}) \),

\[
\mathcal{Y}^\theta_{n+1} = \mathcal{Y}^\theta_n + \left( \mathcal{Z}^\theta_n, \mathcal{X}^\theta_n \right), \quad \mathcal{Z}^\theta_{n+1} = \mathcal{Z}^\theta_n + \mathcal{A}^\theta_n (\mathcal{X}_n) \tau_{n+1} + \mathcal{G}^\theta_n (\mathcal{X}_n) (\mathcal{X}^\theta_n - \mathcal{X}_n),
\]

for \( 0 \leq n \leq N - 1 \). Then, we can choose suitable approximations that \( \mathcal{Y}^\theta_n \approx \mathcal{Y}_n \), \( \mathcal{Z}^\theta_n \approx \mathcal{Z}_n \), \( \tilde{G}^\theta_n (x) \approx (\text{Hess}_x u)(t_n, x) \), and \( \tilde{A}^\theta_n (x) \approx (\mathcal{L}(\nabla_x u))(t_n, x) \), in view of convolutional neural network. In addition, we consider \( \tilde{\theta}_1 \) and \( (\theta_2, \theta_3, \ldots, \theta_{d+1}) \) as the affable approximations of \( u(0, \xi) \) and \( (\nabla_x u)(0, \xi) \). Also, we can choose functions \( \tilde{G}^\theta_n \) and \( \tilde{A}^\theta_n \) as deep convolutional neural networks with \( 0 \leq n \leq N - 1 \).
Similarly, as in Section 3.2, we use the same neural network for \( \tilde{A}_n^n, \tilde{G}_n^n, \forall n \). The difference is that we introduce the number of convolution kernels \( c \). The number of convolution kernels \( c \) is introduced to increase the expressiveness and learning ability of the neural network to better capture various data features. Not only that, the data can also be extended to higher dimensions through convolution operations. Therefore, we use the new notation \( \tilde{A}_n^n, \tilde{G}_n^n, 1 \leq i \leq c, i \in \mathbb{N} \). Suppose \( \nu \geq [(4c + 4)d + d^2 + 1](d + 1) \) and for every \( \theta = (\theta_1, \ldots, \theta_c) \in \mathbb{R}^c, x \in \mathbb{R}^d \), we yield that \( \tilde{G}_n^n(x) = \tilde{G}_0^n(x) \) and \( \tilde{A}_n^n(x) = A_n^n(x) \). Assume \( k \in \mathbb{N} \), and we let the activation function (ReLU) \( R_k(x) \) be given in (19) for every \( x = (x_1, \ldots, x_k) \in \mathbb{R}^k \). For every \( \theta = (\theta_1, \ldots, \theta_c) \in \mathbb{R}^c, \nu \in \mathbb{N}_0, k \in \mathbb{N} \), and \( \nu + k(l + 1) \leq \nu \), suppose that \( M_{k, l}^{\nu, \nu} : \mathbb{R}^l \rightarrow \mathbb{R}^k \) satisfies that

\[
M_{k, l}^{\nu, \nu}(z) = P \otimes z + Q.
\]

where the notation \( \otimes \) represents the convolution rule, the matrix

\[
z = \begin{pmatrix}
x_1 & x_{\sqrt{k} + 1} & \cdots & x_{k - \sqrt{k} + 1} \\
x_2 & x_{\sqrt{k} + 2} & \cdots & x_{k - \sqrt{k} + 2} \\
x_3 & x_{\sqrt{k} + 3} & \cdots & x_{k - \sqrt{k} + 3} \\
\vdots & \vdots & \ddots & \vdots \\
x_{\sqrt{k}} & x_{2\sqrt{k}} & \cdots & x_k
\end{pmatrix},
\]

and \( P, Q \) are presented in (20).

For all \( \theta \in \mathbb{R}^c, 1 \leq i \leq c, \) and \( x \in \mathbb{R}^d \), suppose that

\[
\tilde{A}_n^n = M_{\nu, \nu}^{\nu, [(2c+1)d+1](d+1)} \circ \text{Re} \left( R_d \circ M_{\nu, \nu}^{\nu, [(2c+1)d+1](d+1)} \circ R_d \circ M_{\nu, \nu}^{\nu, [(2c+1)d+1](d+1)} \right),
\]

and that

\[
\tilde{G}_n^n = M_{\nu, \nu}^{\nu, [(4c+4)d+1](d+1)} \circ \text{Re} \left( M_{\nu, \nu}^{\nu, [(4c+4)d+1](d+1)} \circ R_d \circ M_{\nu, \nu}^{\nu, [(4c+4)d+1](d+1)} \right),
\]

in which \( \text{Re}() \) denotes the operation to pull the matrix \( z \) back into the vector \( x \).

The main role of the approximate function (26) and (27) is that we are considering the higher dimensional case. In [50], when the dimension exceeds a certain limit (e.g., \( d > 200 \)), the algorithm requires a lot of calculation and memory (cannot run on our computer). Therefore, we use a neural network approximation in the time domain to reduce the amount of computation and memory. On the other hand, we use convolutional networks to expand the dimensionality of the data.

**Remark 2.** In this remark, we describe some details in convolutional neural networks.

(i) We used three convolutional layers and one linear layer. In the convolution layer, we use a convolution kernel of \( 3 \times 3 \), and the stride and padding are both set to 1 by default. Therefore, the matrix size does not change after each convolution. In the first two convolutional layers, we set the number of channels to 32, and in the last convolutional layer, set the number of channels to 1. For the linear layer, we first pull the output of the convolutional layer into vector, then employ the linear transformation in Section 3.1.

(ii) For the specific choice of the \( \nu \), the basic calculation idea is consistent with Section 3.1. In first stage, we have \( \nu \geq (d + 1)(d + 1) \) the same as Section 3.1. In second stage, the first two convolutional layers are \( 2c \cdot d(d + 1) \), the final convolutional layer is \( 1 \cdot d(d + 1) \), and the linear layer is \( d(d + 1) \) for \( \tilde{A}^\theta \). In third stage, for \( \tilde{G}^\theta \), except that the linear layer is \( d^2(d + 1) \), the others are the same as \( \tilde{A}^\theta \). We give a specific calculation formula here. For more specific information, please refer to Section 3.1.

\[
v \geq (d + 1)(d + 1) + 2c \cdot d(d + 1) + 2c \cdot d(d + 1) + 2 \cdot d(d + 1) + d(d + 1) + d^2(d + 1)
\]

\[
= [(4c + 4)d + d^2 + 1](d + 1).
\]

(iii) Figure 2 depicts the rough schematic diagram of convolutional neural network. In fact, other processing processes are similar to Figure 1. For simplicity, we only draw the process of the convolutional neural network here. As seen in Figure 2, \( x \) has to undergo a “reshape” operation to become \( z \) before it can be input into the network. As can be seen from the figure, \( z \) is subjected to a “conv” operation to obtain matrix \( H^{\text{conv}1} \) of multiple channels. For
The rough schematic diagram of convolutional neural network. [Colour figure can be viewed at wileyonlinelibrary.com]

brevity, only four channels are drawn on the graph, there should actually be 32 channels. Note that, in the last layer of convolution $\mathbf{H}^{final}$, we turn the multiple channels back into a single channel. In addition, the “reshape+FC” operation means that the matrix is first converted into vector by the “reshape” operation. Then, “FC” is used to perform the operation. Here, “FC” is the linear transformation in Section 3.2.

3.4 Optimization algorithms

Here, we give the proposed optimization algorithms. First, we present the following lemma (see [50, Framework 3.2]).

**Lemma 2** ([50]). Let $T, N, d, \theta, \phi, \zeta, \nu$ be defined as before. Let $F : [0, T] \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^{d^d} \rightarrow \mathbb{R}$ and $\hat{g} : \mathbb{R}^d \rightarrow \mathbb{R}$ be functions and $(\Omega, F, \mathbb{P}, (\mathcal{F}_t)_{t \in \{0, T\}})$ be defined as before. Assume for every $\theta \in \mathbb{R}^\nu$, let $U^\theta : \mathbb{R}^d \rightarrow \mathbb{R}$ and $Z^\theta : \mathbb{R}^d \rightarrow \mathbb{R}^d$ be functions and for every $m \in \mathbb{N}_0$, $j \in \mathbb{N}$, let $X^{m,j} : \{0, 1, \ldots, N\} \times \Omega \rightarrow \mathbb{R}^d$ be a stochastic process such that $\lambda^{m,j}_0 = \varphi^{m,j}$ and

$$X^{m,j}_{n+1} = H(t_n, t_{n+1}, \lambda^{m,j}_n, W^{m,j}_{i_{n+1}} - W^{m,j}_{i_n}),$$

for all $0 \leq n \leq N - 1$. Then, for every $\theta \in \mathbb{R}^\nu$, $j \in \mathbb{N}, s \in \mathbb{R}^\nu, n \in \{0, 1, \ldots, N - 1\}$, assume $G^{\theta,j,s}_n : (\mathbb{R}^d)^{N_0} \rightarrow \mathbb{R}^{d^d}$ and $A^{\theta,j,s}_n : (\mathbb{R}^d)^{N_0} \rightarrow \mathbb{R}^d$ are functions. Besides, for every $\theta \in \mathbb{R}^\nu$, $m \in \mathbb{N}_0$, $j \in \mathbb{N}, s \in \mathbb{R}^\nu$, we suppose that $Y^{\theta,m,j,s} : \{0, 1, \ldots, N\} \times \Omega \rightarrow \mathbb{R}$ and $Z^{\theta,m,j,s} : \{0, 1, \ldots, N\} \times \Omega \rightarrow \mathbb{R}^d$ are stochastic processes such that

$$Y^{\theta,m,j,s}_0 = u^\theta(\varphi^{m,j}), \quad Z^{\theta,m,j,s}_0 = z^\theta(\varphi^{m,j}),$$

and

$$Y^{\theta,m,j,s}_{n+1} = Y^{\theta,m,j,s}_n + \tau_{n+1} \left[ \frac{1}{2} \text{Trace}(\sigma(\lambda^{m,j}_n)\sigma(\lambda^{m,j}_n)\cdot G^{\theta,j,s}_n((\lambda^{m,j}_n)_{i \in \mathbb{N}})) + F(t_n, \lambda^{m,j}_n, Y^{\theta,m,j,s}_n, Z^{\theta,m,j,s}_n, G^{\theta,j,s}_n((\lambda^{m,j}_n)_{i \in \mathbb{N}})) \right] + \left( Z^{\theta,m,j,s}_n, \lambda^{m,j}_{n+1} - \lambda^{m,j}_n \right) R^d,$$

and that

$$Z^{\theta,m,j,s}_{n+1} = Z^{\theta,m,j,s}_n + A^{\theta,j,s}_n \left( (\lambda^{m,j}_n)_{i \in \mathbb{N}} \right) \tau_{n+1} + G^{\theta,j,s}_n \left( (\lambda^{m,j}_n)_{i \in \mathbb{N}} \right) (\lambda^{m,j}_{n+1} - \lambda^{m,j}_n).$$

Assume $(J_m)_{m \in \mathbb{N}_0} \subseteq \mathbb{N}$ is a sequence. For every $m \in \mathbb{N}_0$, $s \in \mathbb{R}^\nu$, we let $\hat{\phi}^{m,s} : \mathbb{R}^\nu \times \Omega \rightarrow \mathbb{R}$ be the function, such that

$$\hat{\phi}^{m,s}(\theta, \omega) = \frac{1}{J_m} \sum_{j=1}^{J_m} \left| Y^{\theta,m,j,s}_N(\omega) - \hat{g}(\lambda^{m,j}_N(\omega)) \right|^2,$$

for all $(\theta, \omega) \in \mathbb{R}^\nu \times \Omega$. Then, for every $m \in \mathbb{N}_0$, $s \in \mathbb{R}^\nu$, suppose $\hat{\phi}^{m,s} : \mathbb{R}^\nu \times \Omega \rightarrow \mathbb{R}^\nu$ is a function which satisfies for all $\omega \in \Omega, \theta \in \{\zeta \in \mathbb{R}^\nu : \hat{\phi}^{m,s}(\zeta, \omega) : \mathbb{R}^\nu \rightarrow \mathbb{R} \text{ is differentiable at } \zeta\}$ that
and suppose that \( S : \mathbb{R}^x \times \mathbb{R}^y \times (\mathbb{R}^d)^{[0,1,\ldots,N-1]} \to \mathbb{R}^x \) is a function, and for every \( m \in \mathbb{N}_0 \), we let \( \tilde{\psi}_m : \mathbb{R}^\omega \to \mathbb{R}^y \) and \( \Phi_m : \mathbb{R}^\omega \times \mathbb{R}^y \to \mathbb{R}^\omega \) be functions. For all \( m \in \mathbb{N}_0 \), we let \( \Theta : \mathbb{N}_0 \times \Omega \to \mathbb{R}^\omega, S : \mathbb{N}_0 \times \Omega \to \mathbb{R}^\omega, \) and \( \Xi : \mathbb{N}_0 \times \Omega \to \mathbb{R}^\omega \) be stochastic processes, which satisfy that

\[
S_{m+1} = S\left(S_m, \Theta_m, \left(\chi_{n,i}^{m,i}\right)_{(n,i) \in \{0,1,\ldots,N-1\} \times \mathbb{N}} \right),
\]

and that

\[
\tilde{\Xi}_{m+1} = \tilde{\Phi}_m \left(\tilde{\Xi}_m, \Phi_{m,S_{m+1}}(\Theta_m)\right), \quad \Theta_{m+1} = \Theta_m - \tilde{\psi}_m \left(\tilde{\Xi}_{m+1}\right).
\]

Below, we present several special choices for functions \( \tilde{\psi}_m, \tilde{\Phi}_m, m \in \mathbb{N} \), given in (30). Based on that, we present the following optimization algorithms.

(i) **SGD method.** Provided the setting in Lemma 2, let notations \((\tilde{\gamma}_m)_{m \in \mathbb{N}} \subseteq (0,\infty)\), and suppose that \( m \in \mathbb{N}, x \in \mathbb{R}^\omega, (\varphi_j)_{j \in \mathbb{N}} \in (\mathbb{R}^\rho)^\mathbb{N}\) that

\[
\Phi = \varphi, \quad \tilde{\psi}_m \left(x, (\varphi_j)_{j \in \mathbb{N}}\right) = \varphi_1, \quad \tilde{\psi}_m(x) = \tilde{\gamma}_m x,
\]

and then, it holds that

\[
\Theta_m = \Theta_{m-1} - \tilde{\gamma}_m \Phi^{m-1}(\Theta_{m-1}),
\]

for all \( m \in \mathbb{N} \).

(ii) **Adam with mini-batches** [63]. Here, we use Adam optimizer with the deep learning 2BSDE solver. Provided the setting in Lemma 2, suppose that \( \rho = 2\rho \), and assume \( \text{Pow}_r : \mathbb{R}^\rho \to \mathbb{R}^\rho, 0 < \tilde{r} < \infty \) is the functions satisfying that

\[
\text{Pow}_r(x) = \left(|x_1|^r, \ldots, |x_p|^r\right),
\]

for all \( 0 < \tilde{r} < \infty \) and \( x = (x_1, \ldots, x_p) \in \mathbb{R}^p \).

Let \( 0 < \varepsilon < \infty, (\tilde{\gamma}_m)_{m \in \mathbb{N}} \subseteq (0,\infty), (J_m)_{m \in \mathbb{N}_0} \subseteq \mathbb{N} \), and \( 0 < \hat{\alpha}, \hat{\beta} < 1 \), and assume that \( \hat{\mathbb{M}}, \hat{\hat{\mathbb{M}}} : \mathbb{N}_0 \times \Omega \to \mathbb{R}^\rho \) are the stochastic processes which satisfy for all \( m \in \mathbb{N}_0 \) that \( \hat{\mathbb{X}}_m = \left(\hat{\mathbb{M}}_m, \hat{\hat{\mathbb{M}}}_m\right) \), and suppose that

\[
\tilde{\psi}_m \left(x, y, (\varphi_j)_{j \in \mathbb{N}}\right) = \left[\hat{\alpha} x + (1 - \hat{\alpha}) \left(\frac{1}{J_m} \sum_{j=1}^{J_m} \varphi_j\right), \hat{\beta} y + (1 - \hat{\beta}) \text{Pow}_2 \left(\frac{1}{J_m} \sum_{j=1}^{J_m} \varphi_j\right)\right],
\]

and

\[
\tilde{\psi}_m(x, y) = \left[\varepsilon + \text{Pow}_{2\tilde{r}}(y)\right]^{-1} \tilde{\gamma}_m x,
\]

for all \( m \in \mathbb{N}, x, y \in \mathbb{R}^p, (\varphi_j)_{j \in \mathbb{N}} \in (\mathbb{R}^\rho)^\mathbb{N} \). Then, for all \( m \in \mathbb{N} \), we have

\[
\hat{\hat{\mathbb{M}}}_m = \hat{\hat{\gamma}} \hat{\hat{\mathbb{M}}}_{m-1} + (1 - \hat{\gamma}) \text{Pow}_2 \left(\frac{1}{J_m} \sum_{j=1}^{J_m} \Phi^{m-1,j}(\Theta_{m-1})\right),
\]

and the final update formula is

\[
\Theta_m = \Theta_{m-1} - \left[\varepsilon + \text{Pow}_{2\tilde{r}}(\hat{\mathbb{M}}_m)\right]^{-1} \tilde{\gamma}_m \hat{\mathbb{M}}_m.
\]
This section employs the multiscale deep learning fusion and CNNs to approximately solve several stochastic PDEs, which mainly include the AC, HJB, and BSB equations. Specifically, in Section 4.1, we first employ multiscale deep learning to solve the 20-dimensional AC equation and compare with the method of Beck et al. [50] and use the CNNs to obtain numerical solutions of the higher dimensional AC equation. Then, the numerical experiments in 256 and 400 dimensions are given, respectively. Sections 4.2 and 4.3 also deal with the HJB and BSB equations, respectively, and the only difference is that when using multiscale deep learning method, we utilize the case 100 dimensions to replace that of 20 dimensions. All of the numerical experiments have been performed in Python 3.8 using TensorFlow 2.4, on NVIDIA Tesla P100 GPU (16 GB memory). The simulation codes of proposed method are available on the GitHub page.1

4.1 High-dimensional AC equation

This subsection discusses the approximate solution of the high-dimensional AC equation with a cubic nonlinearity (see 33). Next, the following two examples show that the approximated calculation of Allen equations of different dimensions from multiscale deep learning fusion and convolutional neural networks, respectively.

Example 1 (Multiscale deep learning fusion). Assuming the notations \( T = \frac{3}{10}, \tilde{\gamma} = \frac{1}{1000}, d = 20, \tilde{d} \in \{20, 30, 40, 50\} \), \( N = 20 \), assume for all \( \omega \in \Omega \) that \( \eta = \{0,0, \ldots, 0\} \in \mathbb{R}^d, t \in [0, T), x, z \in \mathbb{R}^d, y \in \mathbb{R}, S \in \mathbb{R}^{d \times d}, t_s = \frac{t}{N}, \hat{g}(x) = \left[2 + \frac{2}{3}||x||^2_{\mathbb{R}^d}\right]^{-1}, \) and
\[
f(t, x, y, z, S) = -\frac{1}{2} \text{Trace}(S) - y + y^3,
\]
(31)
and suppose that \( u : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R} \) is an at most polynomially growing continuous function, such that \( u(T, x) = \hat{g}(x), u|_{[0, T) \times \mathbb{R}^d} \in C^{1,3}([0, T) \times \mathbb{R}^d, \mathbb{R}) \), and
\[
\frac{\partial u}{\partial t}(t, x) = f(t, x, u(t, x), (\nabla_x u)(t, x), (\text{Hess}_x u)(t, x)),
\]
(32)
for all \( (t, x) \in [0, T) \times \mathbb{R}^d \). The solution \( u : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R} \) of (32) such that \( u(T, x) = \left[2 + \frac{2}{3}||x||^2_{\mathbb{R}^d}\right]^{-1}, \) and
\[
\frac{\partial u}{\partial t}(t, x) + \frac{1}{2} (\triangle_x u)(t, x) + u(t, x) - [u(t, x)]^3 = 0,
\]
(33)
for all \( (t, x) \in [0, T) \times \mathbb{R}^d \).

1https://github.com/xiaoxu1996/Deep-PDEs.
TABLE 2  Numerical simulations of the 20-dimensional AC equation.

| Method         | Training steps | $\mu_0 u_0$  | $\sigma_0 u_0$ | $\mu_{L1}$  | $\sigma_{L1}$ | Mean of the loss function | Runtime (s) |
|----------------|----------------|-------------|--------------|-------------|--------------|--------------------------|-------------|
| Beck et al. [50] | 0              | -0.04958    | 0.57116      | 1.88360     | 1.10466      | 0.47839                  | 6           |
|                | 1000           | 0.19091     | 0.14298      | 0.51528     | 0.30760      | 0.02459                  | 14          |
|                | 2000           | 0.26892     | 0.04361      | 0.15655     | 0.11004      | 0.01089                  | 23          |
|                | 3000           | 0.29646     | 0.01359      | 0.04874     | 0.03397      | 0.00724                  | 31          |
|                | 4000           | 0.30252     | 0.00584      | 0.02369     | 0.01444      | 0.01550                  | 40          |
|                | 5000           | 0.30584     | 0.00288      | 0.01243     | 0.00487      | 0.00662                  | 49          |
| Our results    | 0              | -0.02988    | 0.58509      | 1.78238     | 1.27133      | 0.35253                  | 2           |
|                | 1000           | 0.20342     | 0.15110      | 0.48308     | 0.35003      | 0.01850                  | 3           |
|                | 2000           | 0.27478     | 0.04546      | 0.14750     | 0.10976      | 0.00412                  | 5           |
|                | 3000           | 0.29954     | 0.01301      | 0.03965     | 0.03319      | 0.00139                  | 6           |
|                | 4000           | 0.30582     | 0.00393      | 0.01328     | 0.00881      | 0.00120                  | 7           |
|                | 5000           | 0.30852     | 0.00123      | 0.00363     | 0.00184      | 0.00232                  | 9           |

FIGURE 3  Relative $L_1$ approximation error and the mean of the empirical loss function of the 20-dimensional AC equation. [Colour figure can be viewed at wileyonlinelibrary.com]

Table 2 displays different methods to approximatively calculate the mean and standard deviation of $u^{\Theta_m}$ (i.e., $\mu_{\Theta_m}$ and $\sigma_{\Theta_m}$), the mean and standard deviation of corresponding $L_1$-approximation error associated to $u^{\Theta_m}$ (i.e., $\mu_{L1}$ and $\sigma_{L1}$), and the runtime in seconds needed to calculate one realization of $u^{\Theta_m}$ against $m \in \{0, 1000, 2000, 3000, 4000, 5000 \}$ based on 10 independent runs. In addition, Figure 3 depicts approximations of the mean of the relative $L_1$-approximation error and approximations of the mean of the loss function associated to $u^{\Theta_m}$ against $m \in \{0, 1, 2, \ldots, 5000 \}$ based on 10 independent realizations. In the approximative calculations of the relative $L_1$-approximation error, the value $u(0, \xi)$ of the solution $u$ of the (33) has been replaced by the value 0.30879 which, in turn, has been calculated through the branching diffusion method [37]. In particular, the relative $L_1$-approximation error is calculated as $|u^{\Theta_m} - 0.30879|$.

It is not difficult to see from Table 2 that the approximate solution obtained by our method has higher accuracy, and the running time is also greatly reduced. To more intuitively compare with the existing methods, we draw Figure 3. Regarding the relative $L_1$-approximation error in Figure 3, the proposed method is almost consistent with the method in [50] when the number of training steps is small. However, as the number of training steps increases, the proposed method has a smaller relative $L_1$-approximation error, which means that our method is more accurate and effective. In addition, we purposely magnify the relative $L_1$-approximation error from steps 4000 to 5000 to the lower part of the figure. From the enlarged picture, it can be clearly seen that our relative $L_1$-approximation error is already less than 0.01. At the same time, the right side of Figure 3 shows the trend of the loss function. As shown, our loss function value is smaller.

**Example 2** (Convolutional neural networks). We still utilize certain basic settings from Example 1, and the only thing that needs to be modified is the dimension of the data. Here, set $d = 256$ or $d = 400$.

Table 3 extracts approximate solutions of $u^{\Theta_m}$ in different dimensions by convolutional neural networks. The difference with Example 1 is that the number of iteration steps $m \in \{0, 2000, 4000, 6000, 8000, 10,000 \}$. And in Figure 4, $m \in \{0, 1, 2, \ldots, 10,000 \}$. Besides, the 256-dimension and 400-dimension values $u(0, \xi)$ of the solution $u$ of the (33) have...
TABLE 3  Numerical simulations of the large-dimensional AC equation.

| Dimension | Training steps | $\mu_{R^{\theta}}$ | $\sigma_{R^{\theta}}$ | $\mu_{L^1}$ | $\sigma_{L^1}$ | Mean of the loss function | Runtime (s) |
|-----------|----------------|--------------------|--------------------|-------------|----------------|--------------------------|-------------|
| $d = 256$ | 0              | -0.15151           | 0.57393            | 12.7026     | 7.15709        | 0.74728                  | 2           |
|           | 2000           | 0.03103            | 0.03149            | 0.73467     | 0.31448        | 0.02349                  | 4           |
|           | 4000           | 0.04045            | 0.00364            | 0.06574     | 0.06365        | 0.00616                  | 7           |
|           | 6000           | 0.04217            | 0.00131            | 0.02855     | 0.02033        | 0.00087                  | 10          |
|           | 8000           | 0.04139            | 0.000042           | 0.00797     | 0.00723        | 0.00010                  | 12          |
|           | 10000          | 0.04155            | 0.000011           | 0.00227     | 0.00158        | 0.00003                  | 15          |
| $d = 400$ | 0              | 0.08637            | 0.46341            | 14.6610     | 9.05968        | 0.32806                  | 2           |
|           | 2000           | 0.02730            | 0.01902            | 0.59862     | 0.36606        | 0.04361                  | 4           |
|           | 4000           | 0.02499            | 0.00474            | 0.16069     | 0.10427        | 0.00721                  | 7           |
|           | 6000           | 0.02685            | 0.00161            | 0.05114     | 0.03197        | 0.00239                  | 10          |
|           | 8000           | 0.02698            | 0.00082            | 0.02202     | 0.02099        | 0.00028                  | 13          |
|           | 10000          | 0.02729            | 0.00022            | 0.00850     | 0.00619        | 0.00004                  | 15          |

FIGURE 4  Relative $L_1$ approximation error and the mean of the empirical loss function of the large-dimensional AC equation. [Colour figure can be viewed at wileyonlinelibrary.com]

been replaced by the value 0.041531 and 0.027106, respectively, which is also calculated through the Branching diffusion method [37]. Hence, the different dimensional relative $L_1$-approximation error is calculated as $\frac{|u_{R^{\theta}} - 0.041531|}{0.041531}$, $\frac{|u_{R^{\theta}} - 0.027106|}{0.027106}$, respectively.

In Table 3 and Figure 4, no matter whether the dimension of the equation is 256 or 400, as the number of iteration steps increases, the relative $L_1$-approximation error of the approximate solution decreases gradually, and the loss function also tends to decrease in general. This shows that it is numerically feasible for us to use convolutional neural networks to approximately solve higher dimensional stochastic PDEs.

4.2 | High-dimensional BSB equation

This subsection presents the calculation of the high-dimensional BSB equation (see [64] and 37). Similarly, we employ two examples to show that.

Example 3 (Multiscale deep learning fusion). Suppose that $T = 1, d = 100, d \in \{75, 100, 50, 125\}, N = 20, \epsilon = 10^{-8}, \hat{r} = \frac{1}{20},$ and assume for all $\omega \in \Omega$ that $\xi(\omega) = \left(1, \frac{1}{2}, 1, \frac{1}{2}, \ldots, 1, \frac{1}{2}\right) \in \mathbb{R}^d$. Set

$$
\tilde{\gamma}_m = 1.0 \cdot \left(\frac{1}{2}\right)^{\lfloor m/200 \rfloor}.
$$

(34)

Here, $\lfloor \cdot \rfloor$ represents taking the integer of $m/200$. By setting $\sigma_{\text{max}} = \frac{4}{10}, \sigma_{\text{min}} = \frac{1}{10}, \sigma_{c} = \frac{4}{10},$ let us define the function $
\tilde{\sigma} : \mathbb{R} \to \mathbb{R}$ as

$$
\tilde{\sigma}(x) = \begin{cases} 
\sigma_{\text{max}}, & x \geq 0, \\
\sigma_{\text{min}}, & x < 0,
\end{cases}
$$

(35)
for all \( x \in \mathbb{R} \). Assuming for all \( s, t \in [0, T], x = (x_1, \ldots, x_d), w = (w_1, \ldots, w_d), z = (z_1, \ldots, z_d) \in \mathbb{R}^d, y \in \mathbb{R}, S = (S_{ij})_{(i,j)\in[1,\ldots,d]^2} \in \mathbb{R}^{bd}, \) we have that \( \sigma(x) = \sigma(x) \text{diag}(x_1, \ldots, x_d), H(s, t, x, w) = x + \sigma(x)w, \hat{g}(x) = ||x||^2_{L^d}, \) and that

\[
f(t, x, y, z, S) = -\frac{1}{2} \sum_{i=1}^{d} |x_i|^2 \bar{\sigma}(S_{ii})^2 S_{ii} + \hat{r}(y - \langle x, z \rangle_{\mathbb{R}^d}).
\]  

(36)

The solution \( u : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R} \) such that \( u(T, x) = ||x||^2_{L^d} \) and

\[
\frac{\partial u}{\partial t}(t, x) + \frac{1}{2} \sum_{i=1}^{d} |x_i|^2 \bar{\sigma} \left( \frac{\partial^2 u}{\partial x_i^2}(t, x) \right)^2 \frac{\partial^2 u}{\partial x_i^2}(t, x) = \hat{r}(u(t, x) - \langle x, (\nabla_x u)(t, x) \rangle_{\mathbb{R}^d}).
\]  

(37)

for all \((t, x) \in [0, T) \times \mathbb{R}^d\).

Table 4 lists different methods to approximatively calculate the mean and standard deviation of \( u^\Theta_m \), the mean and standard deviation of corresponding \( L_1 \)-approximation error associated to \( u^\Theta_m \), and the runtime in seconds, needed to calculate one realization of \( u^\Theta_m \) against \( m \in \{0, 100, 200, 300, 400\} \) based on 10 independent runs. In addition, Figure 5 depicts approximations of the mean of the relative \( L_1 \)-approximation error and approximations of the mean of the loss function associated to \( u^\Theta_m \) against \( m \in \{0, 1, 2, \ldots, 400\} \) based on 10 independent realizations. In the approximative calculations of the relative \( L_1 \)-approximation error, the value \( u \left( 0, \left(1, \frac{1}{2}, 1, \frac{1}{2}, \ldots, 1, \frac{1}{2} \right) \right) \) of the solution \( u \) of (37) has been replaced by the value 77.1049, in turn, which has been calculated by means of Lemma 3 (more details, see [50]). The relative \( L_1 \)-approximation error is \( \frac{|u^\Theta_m - 77.1049|}{77.1049} \).

**Lemma 3.** Suppose that \( 0 < c, \sigma_{\text{max}}, r, T < \infty, 0 < \sigma_{\text{min}} < \sigma_{\text{max}}, d \in \mathbb{N} \), and assume \( \bar{\sigma} : \mathbb{R} \rightarrow \mathbb{R} \) is the function, such that

\[
\bar{\sigma}(x) = \begin{cases} 
\sigma_{\text{max}}, & x \geq 0, \\
\sigma_{\text{min}}, & x < 0,
\end{cases}
\]  

(38)

for all \( x \in \mathbb{R} \), and we let \( \hat{g} : \mathbb{R}^d \rightarrow \mathbb{R} \) and \( u : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R} \) be the functions, such that \( \hat{g}(x) = c||x||^2_{\mathbb{R}^d} = c \sum_{i=1}^{d} |x_i|^2 \) and

\[
u(t, x) = \exp([r + |\sigma_{\text{max}}|^2(T - t)])\hat{g}(x),
\]  

(39)

for all \( t \in [0, T], x = (x_1, \ldots, x_d) \in \mathbb{R}^d \). Then, we have for all \( t \in [0, T], x = (x_1, \ldots, x_d) \in \mathbb{R}^d \) that \( u \in C^\infty([0, T] \times \mathbb{R}^d, \mathbb{R}), u(T, x) = \hat{g}(x) \), and

\[
\frac{\partial u}{\partial t}(t, x) + \frac{1}{2} \sum_{i=1}^{d} |x_i|^2 \bar{\sigma} \left( \frac{\partial^2 u}{\partial x_i^2}(t, x) \right)^2 \frac{\partial^2 u}{\partial x_i^2}(t, x) = \hat{r}(u(t, x) - \langle x, (\nabla_x u)(t, x) \rangle_{\mathbb{R}^d}).
\]  

(40)

Looking at Table 4 as a whole, we observe that the approximate solution obtained by our method has higher accuracy. However, unlike Example 1, our runtime will be a bit more. Similarly, we paint Figure 5 for comparing with the existing methods. It is evident from Figure 5 that when the number of iteration steps exceeds 200, the proposed method already stratifies with the method of Beck et al. [50]. And from the partially enlarged picture, Beck et al. [50] method differs from us by one coordinate scale in terms of the relative \( L^1 \)-approximation error and loss function value. These all demonstrate and illustrate the effectiveness of our method.

**Example 4** (Convolutional neural networks). Herein, most of our settings are the same as Example 3. Based on this point, what needs to be modified is the dimension of the data and the learning rate. Firstly, we set \( d = 256 \) or \( d = 400 \), and the learning rate is

\[
\bar{\gamma}_m = 2.0 \cdot \left( \frac{1}{2} \right)^{[m/500]}
\]  

(41)

Table 5 reports approximate solutions of \( u^\Theta_m \) in different dimensions by convolutional neural networks. The difference with Example 3 is that the number of iteration steps here \( m \in \{0, 200, 400, 600, 800, 1000\} \). In addition, in Figure 6, \( m \in \{0, 1, 2, \ldots, 1000\} \). In addition, the 256-dimension and 400-dimension value \( u \left( 0, \left(1, \frac{1}{2}, 1, \frac{1}{2}, \ldots, 1, \frac{1}{2} \right) \right) \) of the solution \( u \)
TABLE 4  Numerical simulations of the 100-dimensional BSB equation.

| Method            | Training steps | $\mu_{\theta_{\text{true}}}$ | $\sigma_{\theta_{\text{true}}}$ | $\mu_{\overline{L}_1}^{\text{true}}$ | $\sigma_{\overline{L}_1}^{\text{true}}$ | Mean of the loss function | Runtime (s) |
|-------------------|----------------|------------------------------|-------------------------------|--------------------------------|--------------------------------|---------------------------|-------------|
| Beck et al. [50]  | 0              | 0.3940                      | 0.2253                        | 0.99489                         | 0.00292                        | 5355.51                   | 23          |
|                   | 100            | 55.9301                     | 1.9195                        | 0.27462                         | 0.02489                        | 540.55                    | 27          |
|                   | 200            | 73.4561                     | 0.9547                        | 0.04732                         | 0.01238                        | 149.26                    | 31          |
|                   | 300            | 75.7877                     | 0.5027                        | 0.01708                         | 0.00652                        | 90.979                    | 36          |
|                   | 400            | 76.7701                     | 0.3009                        | 0.00491                         | 0.00316                        | 63.846                    | 40          |
| Our results       | 0              | 0.5517                      | 0.2378                        | 0.99285                         | 0.00308                        | 5411.35                   | 21          |
|                   | 100            | 57.0542                     | 0.4246                        | 0.26004                         | 0.00551                        | 226.89                    | 29          |
|                   | 200            | 75.2420                     | 0.1476                        | 0.02416                         | 0.00191                        | 8.619                     | 36          |
|                   | 300            | 76.8373                     | 0.0515                        | 0.00347                         | 0.00067                        | 4.866                     | 44          |
|                   | 400            | 77.1226                     | 0.0302                        | 0.00039                         | 0.00024                        | 4.882                     | 52          |

FIGURE 5  Relative $L_1$ approximation error and the mean of the empirical loss function of the 100-dimensional BSB equation. [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 5  Numerical simulations of the large-dimensional BSB equation.

| Dimension | Training steps | $\mu_{\theta_{\text{true}}}$ | $\sigma_{\theta_{\text{true}}}$ | $\mu_{\overline{L}_1}^{\text{true}}$ | $\sigma_{\overline{L}_1}^{\text{true}}$ | Mean of the loss function | Runtime (s) |
|-----------|----------------|------------------------------|-------------------------------|--------------------------------|--------------------------------|---------------------------|-------------|
| $d = 256$ | 0              | 0.4901                      | 0.2948                        | 0.99752                         | 0.00149                        | 35095                     | 4           |
|           | 200            | 164.3867                    | 0.6826                        | 0.16719                         | 0.00346                        | 345.59                    | 34          |
|           | 400            | 190.3597                    | 0.2866                        | 0.03561                         | 0.00145                        | 26.737                    | 63          |
|           | 600            | 194.5438                    | 0.1614                        | 0.01441                         | 0.00082                        | 17.643                    | 92          |
|           | 800            | 196.7375                    | 0.1106                        | 0.00330                         | 0.00056                        | 15.022                    | 122         |
|           | 1000           | 197.3413                    | 0.0793                        | 0.00041                         | 0.00023                        | 14.395                    | 151         |
| $d = 400$ | 0              | 0.5218                      | 0.2603                        | 0.99831                         | 0.00084                        | 86234                     | 6           |
|           | 200            | 170.3224                    | 1.6187                        | 0.44776                         | 0.00525                        | 3696.6                    | 77          |
|           | 400            | 271.3132                    | 1.1839                        | 0.12031                         | 0.00384                        | 347.57                    | 148         |
|           | 600            | 298.8917                    | 0.6204                        | 0.03089                         | 0.00201                        | 42.596                    | 219         |
|           | 800            | 305.8975                    | 0.2641                        | 0.00818                         | 0.00086                        | 29.087                    | 291         |
|           | 1000           | 308.5768                    | 0.1068                        | 0.00051                         | 0.00035                        | 23.190                    | 362         |

of (37) have been replaced via the value 197.3885 and 308.4195, respectively. It also can be computed by means of Lemma 3. And the different dimensions relative $L_1$-approximation error is $|\theta_{\text{true}}^{256} - 197.3885|$ and $|\theta_{\text{true}}^{400} - 308.4195|$, respectively.

In Table 5, it can be seen that from 256 dimensions to 400 dimensions, the running time using convolutional neural networks increases exponentially. This is mainly because as the dimension increases, the memory overhead increases. However, the accuracy of the approximated solution did not change much. This demonstrates that convolutional neural networks can extend approximated solutions to higher dimensions without losing accuracy. Also, Figure 6 can show this more intuitively.
4.3 | High-dimensional HJB equation

This subsection approximatively calculates the solution of a high-dimensional HJB equation with a nonlinearity that is quadratic in the gradient (see [37]). In the following, we present two examples to show the related calculation.

Example 5 (Multiscale deep learning fusion). We suppose \( d = 100, \tilde{d} \in \{50, 75, 100, 125\}, T = 1, N = 20, \epsilon = 10^{-8}, \) and suppose for all \( \omega \in \Omega \) that \( \xi(\omega) = \emptyset \in \mathbb{R}^d \). Then, assume for all \( m \in \mathbb{N}, s, t \in [0, T], x, w, z \in \mathbb{R}^d, y \in \mathbb{R}, S \in \mathbb{R}^{d \times d} \) that \( \sigma(x) = \sqrt{2} \text{Id}_{\mathbb{R}^d}, H(s, t, x, w) = x + \sqrt{2}w, \tilde{\gamma}(x) = \ln \left( \frac{1}{2} \left[ 1 + ||x||^2_{\mathbb{R}^d} \right] \right), f(t, x, y, z) = -\text{Trace}(S) - ||z||^2_{\mathbb{R}^d}, \) and

\[
\tilde{\gamma}_m = \frac{1}{100} \cdot \left( \frac{1}{5} \right)^{\lfloor m/1000 \rfloor}.
\]

The solution \( u : [0, T) \times \mathbb{R}^d \rightarrow \mathbb{R} \) of the PDE (32) satisfies for all \( (t, x) \in [0, T) \times \mathbb{R}^d \) that

\[
\frac{\partial u}{\partial t}(t, x) + (\triangle_x u) = ||\nabla_x u(t, x)||^2_{\mathbb{R}^d}.
\]

Table 6 lists different methods to approximatively calculate the mean and standard deviation of \( u^{\Theta_m} \), the mean and standard deviation of relative \( L_1 \)-approximation error associated to \( u^{\Theta_m} \), and the runtime in seconds, needed to calculate one realization of \( u^{\Theta_m} \) against \( m \in \{0, 500, 1000, 1500, 2000\} \), based on 10 independent runs. Furthermore, Figure 7 shows approximations of the mean of the relative \( L_1 \)-approximation error and approximations of the mean of the loss function associated to \( u^{\Theta_m} \) against \( m \in \{0, 1, 2, \ldots, 2000\} \) based on 10 independent realizations. For the approximative calculations of the relative \( L_1 \)-approximation error, the value \( u(0, \xi) \) of the solution \( u \) of (43) has been substituted by the value 4.5901, conversely, which was calculated by the means of in [37, Lemma 4.2] and the classical Monte Carlo method [37].

It can be clearly observed from Table 6 and Figure 7 that the approximated solution obtained via our method has higher accuracy. Figure 7 shows the curve slope of the relative \( L_1 \)-approximation error and the loss function change with our method at 1000 steps, which is mainly caused by the change of the learning rate (see 42). Likewise, we also place the local comparison from steps 1500 to 2000 at the bottom of this figure.

Example 6 (Convolutional neural networks). Herein, certain basic settings from Example 5 are still used, and the only thing that needs to be changed is the dimension of the data. Below, set \( d = 256 \) or \( d = 400 \). Noting the learning rate, we adjusted the learning rate with a fixed number of steps instead of exponential decay. The specific formula is

\[
\tilde{\gamma}_m = \begin{cases} 
0.01, & m < 1000, \\
0.005, & m \geq 1000.
\end{cases}
\]

Table 7 and Figure 8 display approximated solutions of \( u^{\Theta_m} \) in different dimensions by convolutional neural networks. Besides, the 256-dimension and 400-dimension values \( u(0, \xi) \) of the solution \( u \) of (31) have been replaced by the value
TABLE 6  Numerical simulations of the 100-dimensional HJB equation.

| Method           | Training steps | $\mu_{\theta^m}$ | $\sigma_{\theta^m}$ | $\mu_{L^1_{\text{emp}}}$ | $\sigma_{L^1_{\text{emp}}}$ | Mean of the loss function | Runtime (s) |
|------------------|----------------|------------------|---------------------|--------------------------|-----------------------------|--------------------------|--------------|
| Beck et al. [50] | 0              | 0.4328           | 0.0620              | 0.90571                  | 0.01351                     | 1065.5                   | 17           |
|                  | 500            | 2.5108           | 0.0555              | 0.45300                  | 0.01208                     | 37.574                   | 33           |
|                  | 1000           | 3.7526           | 0.0432              | 0.22168                  | 0.00942                     | 11.839                   | 49           |
|                  | 1500           | 4.2555           | 0.0293              | 0.03587                  | 0.00639                     | 5.105                    | 65           |
|                  | 2000           | 4.6101           | 0.0258              | 0.00673                  | 0.00232                     | 2.783                    | 81           |
| Our results      | 0              | 0.2294           | 0.0940              | 0.95001                  | 0.02047                     | 23.32                    | 18           |
|                  | 500            | 3.7223           | 0.0603              | 0.18907                  | 0.01313                     | 0.834                    | 42           |
|                  | 1000           | 4.5465           | 0.0097              | 0.00951                  | 0.00212                     | 0.025                    | 67           |
|                  | 1500           | 4.5762           | 0.0052              | 0.00304                  | 0.00113                     | 0.022                    | 91           |
|                  | 2000           | 4.5924           | 0.0021              | 0.00063                  | 0.00024                     | 0.019                    | 115          |

FIGURE 7  Relative $L_1$ approximation error and the mean of the empirical loss function of the 100-dimensional HJB equation. [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 7  Numerical simulations of the large-dimensional HJB equation.

| Dimension | Training steps | $\mu_{\theta^m}$ | $\sigma_{\theta^m}$ | $\mu_{L^1_{\text{emp}}}$ | $\sigma_{L^1_{\text{emp}}}$ | Mean of the loss function | Runtime (s) |
|-----------|----------------|------------------|---------------------|--------------------------|-----------------------------|--------------------------|--------------|
| $d = 256$ | 0              | 0.5348           | 0.2753              | 0.90346                  | 0.04970                     | 35.86                    | 1            |
|           | 500            | 4.2221           | 0.1976              | 0.23779                  | 0.03567                     | 1.974                    | 3            |
|           | 1000           | 5.3966           | 0.0502              | 0.02576                  | 0.00907                     | 0.037                    | 4            |
|           | 1500           | 5.5126           | 0.0148              | 0.00481                  | 0.00267                     | 0.010                    | 6            |
|           | 2000           | 5.5399           | 0.0025              | 0.00042                  | 0.00019                     | 0.008                    | 7            |
| $d = 400$ | 0              | 0.5902           | 0.2538              | 0.90143                  | 0.04239                     | 48.68                    | 2            |
|           | 500            | 4.3794           | 0.1916              | 0.26935                  | 0.03200                     | 2.853                    | 3            |
|           | 1000           | 5.7611           | 0.0596              | 0.03784                  | 0.00995                     | 0.076                    | 4            |
|           | 1500           | 5.9330           | 0.0207              | 0.00913                  | 0.00346                     | 0.010                    | 6            |
|           | 2000           | 5.9818           | 0.0042              | 0.00099                  | 0.00071                     | 0.006                    | 7            |

5.5393 and 5.9877, respectively, which also can be calculated through the classical Monte Carlo method [37]. Thus, the different dimensions relative $L_1$-approximation error is $\frac{|\mu_{\theta^m} - 5.5393|}{5.5393}$ and $\frac{|\mu_{\theta^m} - 5.9877|}{5.9877}$, respectively.

Comparing Tables 6 and 7, one can find the fact that the running time of using convolutional neural network is faster than using linear neural network. Generally speaking, higher dimensional problems require more memory and take longer to compute. While in Table 7, it only takes 7 s to calculate the 400-dimensional HJB equation. In addition, from the relative $L_1$-approximation error and loss function in Figure 8, the accuracy of the convolutional neural network is almost the same as that of the linear neural network. This shows that convolutional neural networks are more suitable than linear neural networks for the HJB equation.
This paper developed numerical approximation for high-dimensional fully nonlinear merged PDEs and 2BSDEs based on the deep CNN technique. First, the forward discretization was employed in the time direction, and then, two approximation approaches were adopted in the space direction by the multiscale deep learning fusion and the convolutional neural networks, from which the former is more accurate and efficient than the method of Beck et al. [50]; the latter can use matrix arrangement to calculate higher dimensional fully nonlinear PDEs, such as $d = 400$. These were reflected in the numerical experiments. Unfortunately, despite the computational improvement, we are temporarily unable to obtain theoretical results of the proposed methods, which will be further considered by us in the future. Following the results, a future study will try to apply a temporal second-order approximation combined with a regularized convolutional neural network [65] for solving high-dimensional fully nonlinear merged PDEs–2BSDEs system, based on the stochastic pooling.

**AUTHOR CONTRIBUTIONS**

Xu Xiao: Conceptualization; methodology; software; data curation; writing—original draft; investigation. Wenlin Qiu: Conceptualization; investigation; writing—review and editing; writing—original draft; methodology. Omid Nikan: Writing—original draft; writing—review and editing; investigation.

**ACKNOWLEDGEMENTS**

The authors are very grateful to the associate editor and the anonymous referees for their careful reading and valuable suggestions, which have notably improved the quality of this paper. This work was supported by Postgraduate Scientific Research Innovation Project of Hunan Province (No. CX20220454).

**CONFLICT OF INTEREST STATEMENT**

The authors declare that there are no conflicts of interest regarding the publication of this paper.

**ORCID**

Xu Xiao https://orcid.org/0000-0001-9825-8672

Wenlin Qiu https://orcid.org/0000-0003-1144-5315

Omid Nikan https://orcid.org/0000-0003-3041-8726

**REFERENCES**

1. P. Kumar and V. Panwar, *Wavelet neural network based controller design for non-affine nonlinear systems*, J. Math. Comput. Sci. **24** (2022), 49–58.
2. E. Ngondiep, A novel three-level time-split approach for solving two-dimensional nonlinear unsteady convection-diffusion-reaction equation, J. Math. Comput. Sci. 26 (2022), no. 3, 222–248.

3. T. Guo, O. Nikan, W. Qiu, and D. Xu, Localized meshless approaches based on theta method and BDF2 for nonlinear Sobolev equation arising from fluid dynamics, Commun. Nonlinear Sci. Numer. Simul. 117 (2023), 106989.

4. A. H. Ibrahim, K. Muangchoob, N. S. Mohamedc, and A. B. Abubakard, Derivative-free SMR conjugate gradient method for constraint nonlinear equations, J. Math. Comput. Sci. 24 (2022), no. 2, 147–164.

5. C. Bender, N. Schweizer, and J. Zhuo, A primal–dual algorithm for BSDEs, Math. Finance 27 (2017), no. 3, 866–901.

6. E. Gobet, J.-P. Lemor, and X. Warin, A regression-based Monte Carlo method to solve backward stochastic differential equations, Ann. Appl. Probab. 2005 (2005), 2172–2202.

7. S. Crépey, R. Gerboud, Z. Grbac, and N. Ngor, Counterparty risk and funding: the four wings of the TVA, Int. J. Theor. Appl. Finance 16 (2013), no. 02, 1350006.

8. E. Bayraktar and V. Young, Pricing options in incomplete equity markets via the instantaneous Sharpe ratio, Ann. Finance 4 (2008), 399–429.

9. A. L. Amadori, Nonlinear integro-differential evolution problems arising in option pricing: a viscosity solutions approach, Differ. Integr. Equations 16 (2003), no. 7, 787–811.

10. E. Pardoux and S. Peng, Adapted solution of a backward stochastic differential equation, Syst. Control lett. 14 (1990), no. 1, 55–61.

11. E. Pardoux and S. Tang, Forward-backward stochastic differential equations and quasilinear parabolic PDEs, Probab. Theory Related Fields 114 (1999), 123–150.

12. P. Cheridito, H. M. Soner, N. Touzi, and N. Victoir, Second-order backward stochastic differential equations and fully nonlinear parabolic PDEs, Comm. Pure Appl. Math. 60 (2007), no. 7, 1081–1110.

13. P. Henry-Labordere, X. Tan, and N. Touzi, A numerical algorithm for a class of BSDEs via the branching process, Stoch. Process. Appl. 124 (2014), no. 2, 1112–1140.

14. B. Øksendal, Stochastic differential equations: an introduction with applications, Springer Science & Business Media, 2013.

15. H. Lee and I. S. Kang, Neural algorithm for solving differential equations, J. Comput. Phys. 91 (1990), no. 1, 110–131.

16. P. Ramuhalli, L. Udpa, and S. S. Udpa, Finite-element neural networks for solving differential equations, IEEE Trans. Neur. Netw. 16 (2005), no. 6, 1381–1392.

17. C. Bender and R. Denk, A forward scheme for backward SDEs, Stoch. Process. Appl. 117 (2007), no. 12, 1793–1812.

18. B. Bouchard and N. Touzi, Discrete-time approximation and Monte-Carlo simulation of backward stochastic differential equations, Stoch. Process. Appl. 111 (2004), no. 2, 175–206.

19. J.-F. Chassagneux, Linear multistep schemes for BSDEs, SIAM J. Numer. Anal. 52 (2014), no. 6, 2815–2836.

20. J.-F. Chassagneux and A. Richou, Numerical simulation of quadratic BSDEs, Ann. Appl. Probab. 26 (2016), no. 1, 262–304.

21. J.-F. Chassagneux and A. Richou, Numerical stability analysis of the Euler scheme for BSDEs, SIAM J. Numer. Anal. 53 (2015), no. 2, 1172–1193.

22. J.-F. Chassagneux and A. Richou, Numerical simulation of quadratic BSDEs, Ann. Appl. Probab. 2016 (2016), 262–304.

23. D. Crisan and K. Manolarakis, Solving backward stochastic differential equations using the cubature method: application to nonlinear pricing, SIAM J. Numer. Anal. 3 (2012), no. 1, 534–571.

24. D. Crisan and K. Manolarakis, Second order discretization of backward SDEs and simulation with the cubature method, Ann. Appl. Probab. 2014 (2014), 652–678.

25. D. Crisan, K. Manolarakis, and N. Touzi, On the Monte Carlo simulation of BSDEs: an improvement on the Malliavin weights, Stoch. Process. Appl 120 (2010), no. 7, 1133–1158.

26. J. Douglas Jr, J. Ma, and P. Protter, Numerical methods for forward-backward stochastic differential equations, Ann. Appl. Probab. 6 (1996), no. 3, 940–968.

27. E. Gobet and C. Labart, Solving BSDE with adaptive control variate, SIAM J. Numer. Anal. 48 (2010), no. 1, 257–277.

28. E. Gobet and J.-P. Lemor, Numerical simulation of BSDEs using empirical regression methods: theory and practice, 2008. arXiv preprint arXiv:0806.4447, 1–17.

29. E. Gobet, J. G. López-Salas, P. Turkedjiev, and C. Vázquez, Stratified regression Monte-Carlo scheme for semilinear PDEs and BSDEs with large scale parallelization on GPUs, SIAM J. Sci. Comput. 38 (2016), no. 6, C652–C677.

30. E. Gobet and P. Turkedjiev, Linear regression MDP scheme for discrete backward stochastic differential equations under general conditions, Math. Comput. 85 (2016), no. 299, 1359–1391.

31. T. P. Huijskens, M. J. Ruijter, and C. W. Oosterlee, Efficient numerical Fourier methods for coupled forward–backward SDEs, J. Comput. Appl. Math. 296 (2016), 593–612.

32. J. Ma, P. Protter, J. San Martin, and S. Torres, Numerical method for backward stochastic differential equations, Ann. Appl. Probab. 2002 (2002), 302–316.

33. J. Ma and J. Yong, Forward-backward stochastic differential equations and their applications, Springer Science & Business Media, 1999.

34. G. N. Milstein and M. V. Tretyakov, Numerical algorithms for forward-backward stochastic differential equations, SIAM J. Sci. Comput. 28 (2006), no. 2, 561–582.

35. G. N. Milstein and M. V. Tretyakov, Discretization of forward–backward stochastic differential equations and related quasi-linear parabolic equations, IMA J. Numer. Anal. 27 (2007), no. 1, 24–44.

36. H. Pham, Feynman-Kac representation of fully nonlinear PDEs and applications, Acta Math. Vietnam 40 (2015), 255–269.

37. 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), no. 4, 349–380.
38. J. Han, A. Jentzen, and E. Weinan, Solving high-dimensional partial differential equations using deep learning, Proc. Natl. Acad. Sci. 115 (2018), no. 34, 8505–8510.
39. P. Briand and C. Labart, Simulation of BSDEs by Wiener chaos expansion, Ann. Appl. Probab. 24 (2014), no. 3, 1129–1171.
40. Y. Fu, W. Zhao, and T. Zhou, Efficient spectral sparse grid approximations for solving multi-dimensional forward backward SDEs, Discrete Contin. Dyn. Syst. Ser. B 22 (2017), no. 9, 3439–3458.
41. D. Chang, H. Liu, and J. Xiong, A branching particle system approximation for a class of FBSDEs, Probab. Uncertain. Quant. Risk 1 (2016), no. 1, 1–34.
42. B. Bouchard, R. Elie, and N. Touzi, Discrete-time approximation of BSDEs and probabilistic schemes for fully nonlinear PDEs, Adv. Financial Model. 8 (2009), 91–124.
43. W. Guo, J. Zhang, and J. Zhuo, A monotone scheme for high-dimensional fully nonlinear PDEs, Ann. Appl. Probab. 25 (2015), no. 3, 1540–1580.
44. T. Kong, W. Zhao, and T. Zhou, Probabilistic high order numerical schemes for fully nonlinear parabolic PDEs, Commun. Comput. Phys. 18 (2015), no. 5, 1482–1503.
45. J. Darbon and S. Osher, Algorithms for overcoming the curse of dimensionality for certain Hamilton–Jacobi equations arising in control theory and elsewhere, Res. Math. Sci. 3 (2016), no. 1, 19.
46. B. Yu, The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (2018), no. 1, 1–12.
47. M. Raissi, Deep hidden physics models: deep learning of nonlinear partial differential equations, Mach. Learn. Res. 19 (2018), no. 1, 932–955.
48. J. Sirignano and M. Spiliopoulos, DGM: a deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018), 1339–1364.
49. M. Raissi, P. Perdikaris, and G. E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019), 686–707.
50. C. Beck, W. E, and A. Jentzen, Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations, J. Nonlinear Sci. 29 (2019), 1563–1619.
51. L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, DeepXDE: a deep learning library for solving differential equations, SIAM Rev. 63 (2021), no. 1, 208–228.
52. M. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review, Int. J. Autom. Comput. 14 (2017), no. 5, 503–519.
53. X. Xiao, W. Qiu, and O. Nikan, Numerical approximation based on deep convolutional neural network for high-dimensional fully nonlinear merged PDEs and 2BSDEs, Math. Meth. Appl. Sci. 47 (2024), 6184–6204, DOI 10.1002/mma.9915.