Journal of Computational Physics 468 (2022) 111503

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Deep neural networks based temporal-difference methods for high-dimensional parabolic partial differential equations

Shaojie Zeng [a], Yihua Cai [a], Qingsong Zou [b,*]

[a] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, 510006, China
[b] School of Computer Science and Engineering, and Guangdong Province Key Laboratory of Computational Science, Sun Yat-sen University, Guangzhou 510006, China

ARTICLE INFO

Article history:
Received 18 April 2022
Received in revised form 19 July 2022
Accepted 21 July 2022
Available online 28 July 2022

Keywords:
High-dimensional PDEs
Forward-backward stochastic differential equations
Neural network
Temporal-difference learning

ABSTRACT

Solving high-dimensional partial differential equations (PDEs) is a long-standing challenge, for which classical numerical methods suffer from the well-known *curse of dimensionality*. In this paper, we propose deep neural networks (NN) based temporal-difference (TD) learning methods for numerically solving high-dimensional parabolic PDEs. To this end, we approximate the solution of the original PDE with an NN function. To calculate this neural network function, we first transform the deterministic parabolic PDE to a forward-backward stochastic differential equations (FBSDE) system with the nonlinear Feynman-Kac formula, and then transform the forward updating process of FBSDE as a Markov reward process (MRP). On this basis, we approximate the solution of the PDE by training an NN function with a reinforcement learning technique described as below: we first discretize the temporal interval to a finite number of time steps and then at each time step, we generate many trajectories and design the loss function as the mean square of temporal difference error on all trajectories. With this loss function, we update the parameters of the NN function with the stochastic gradient descent (SGD) method. Numerical experiments show that comparing to some other existed deep learning methods, our method not only accelerates the computational speed but also improves the computational accuracy. In particular, the relative errors of our algorithm achieve the order of $O(10^{-4})$ even the dimension of the problem is as high as 100.

1. Introduction

Partial differential equations (PDEs) are widely used in physics, engineering, biology, chemistry, computer science, finance and many other fields. Classical numerical methods for PDEs, such as finite difference methods and finite element methods, suffer from the so-called *curse of dimensionality* [2] that the computational cost increases exponentially along with the dimension of the problem to be solved. The neutral network (NN) methods for PDEs differ from the classical numerical method in that it approximates the solution of a PDE by a NN function, instead of a classic piecewise polynomial. Due to strong approximation capabilities for high dimensional nonlinear functions of the deep NNs [21,22], the NN method can avoid the curse of dimensionality. Besides, the NN method has many other advantages such as it can easily handle nonlinear equations [7,31].

* Corresponding author.
E-mail address: mcszqs@mail.sysu.edu.cn (Q. Zou).

https://doi.org/10.1016/j.jcp.2022.111503
0021-9991/

The NN method for solving PDEs can be roughly classified into two categories. In the first category, the NN method solves the PDE by training an NN function directly according to the original deterministic PDE. In other words, the loss function of this method is designed according to the original deterministic PDE. For examples, the physics-informed neural networks (PINN, [30]) and its extensions (see e.g. [8,10,26,28,35–37]) use a mean-squared error of the PDE residual on sampling points as its loss function; the Deep Galerkin Method (DGM, [33]) also uses the PDE residual to design its loss function; the deep Ritz method (DRM, [13]) uses the discrete energy functional of the PDE equation as its loss function; the weak adversarial method (WAN, [39]) uses the weak formulation of the PDE to design its loss function. This kind of loss function should be appropriately modified if certain boundary conditions are involved. For instances, the deep Nitsche method (DNM, [23]) improves the DRM by incorporating the idea of Nitsche to deal with the essential boundary conditions; the penalty-free neural network method (PFNN, [32]) employs an additional neural network to incorporate boundary conditions.

In the second category, the NN method solves PDEs by training a NN function according to a forward-backward stochastic differential equations (FBSDE) system which is derived from the original deterministic PDEs. Namely, the loss function of these methods is designed according to the derived FBSDE. For instances, the Deep BSDE method [12,17] calculates the initial value of a (nonlinear) parabolic PDE by training a sequence of NNs which are used to approximate each time step's gradient of the solution of the BSDE derived from the original PDE. The forward-backward stochastic neural networks (FBSNN, [29]) method approximate the solution of the original PDE by training one neural network using the loss function indicating the square error of the BSDE on sampling trajectories. Thereafter, some variants of the Deep BSDE and FBSNN have been developed ([3,5,15,16,19,34,38]). However, the Deep BSDE and the FBSNN can update the neural network parameter only after obtaining the final results of all sampled trajectories.

In this paper, we propose a novel NN method for high-dimensional PDEs from the perspective of reinforcement learning. The basic idea is to formulate the target equation as a Markov reward process (MRP), and then use the NN based reinforcement learning to solve the MRP. To this end, we first reformulate a deterministic parabolic equation into a FBSDE system with the nonlinear Feynman-Kac formula [27], and establish the MRP according to the forward updating process of FBSDE. We treat the solution of the original equation as the value function of the MRP. Correspondingly, the solution of the parabolic equation will be approximated by iteratively training the value function of the MRP. We then approximate the solution by a neural network, and apply another neural network or the auto-differentiation module [1] to approximate the gradient of the value function. The loss function is constructed by the temporal difference error and the termination condition of the FBSDE. In addition, we generalize our method to multi-steps temporal difference method. The advantages of our algorithm are that: 1) it can be updated online, 2) the time step can be adjusted at any time, and 3) the computation cost is low. Our numerical results show that compared with some popular deep learning methods, our algorithm not only accelerates the computational speed but also improves the computational accuracy.

The rest of the paper is organized as follows. In Section 2, we present some existed neural networks methods for a FBSDE system derived from a deterministic parabolic PDE. Section 3 is our main section in which we introduce our temporal difference methods for solving FBSDEs. In Section 4, several numerical experiments are conducted to demonstrate the performance of our numerical methods. Some concluding remarks are given in Section 5.

2. Neural networks based Monte-Carlo methods

We consider the following quasi-linear parabolic partial differential equation:

$$\begin{cases} \dfrac{\partial u}{\partial t} + \dfrac{1}{2}\mathrm{Tr}(\sigma\sigma^{\mathrm{T}}\mathrm{Hess}_x u) + b\cdot\nabla u + f = 0, \\ u(T,\cdot) = g, \end{cases} \tag{2.1}$$

where $u = u(t,x)$, $x \in \mathbb{R}^d$, $0 < t < T$ is an unknown function to be solved, ∇u, $\mathrm{Hess}_x u$ are its corresponding gradient and Hessian matrix, $b = b(t,x) \in \mathbb{R}^d$, $\sigma = \sigma(t,x) \in \mathbb{R}^{d\times d}$ and $f = f\left(t,x,u,\sigma^{\mathrm{T}}\nabla u\right)$ are given vectorial-valued, matrix-valued, and nonlinear (with respect to u and ∇u) functions, respectively.

Letting $\{W_t\}_{t\in[0,T]}$ be a d-dimensional Brownian motion and $\{X_t\}_{t\in[0,T]}$ a d-dimensional stochastic process satisfying the forward equation:

$$dX_t = b(t,X_t)dt + \sigma(t,X_t)dW_t,\ X_0 = \xi \in \mathbb{R}^d. \tag{2.2}$$

We define two induced stochastic processes ([27]) by

$$Y_t = u(t,X_t), \quad Z_t = \nabla u(t,X_t). \tag{2.3}$$

Then, by the Itô formula ([6]),

$$\begin{aligned} dY_t &= \Big[\frac{\partial u}{\partial t}(t,X_t) + \nabla u(t,X_t)\cdot b(t,X_t) + \frac{1}{2}\mathrm{Tr}(\sigma(t,X_t)\sigma^{\mathrm{T}}(t,X_t)\mathrm{Hess}_x u(t,X_t))\Big]dt + [\nabla u(t,X_t)]^{\mathrm{T}}\sigma(t,X_t)dW_t \\ &= -f\left(t,X_t,u(t,X_t),\sigma^{\mathrm{T}}(t,X_t)\nabla u(t,X_t)\right)dt + [\nabla u(t,X_t)]^{\mathrm{T}}\sigma(t,X_t)dW_t. \end{aligned} \tag{2.4}$$

Therefore, Y_t and Z_t satisfy the following backward equation

$$dY_t = -f(t, X_t, Y_t, \sigma^{\mathrm{T}}(t, X_t)Z_t)dt + Z_t^{\mathrm{T}}\sigma(t, X_t)dW_t,\ Y_T = g(X_T). \tag{2.5}$$

The two equations (2.2) and (2.5) together are called as a forward-backward stochastic equation (FBSDE) system. Note that once we have many triples of processes $(X(t), Y(t), Z(t))$ which satisfy the FBSDE (2.2) and (2.5), we can obtain the function u by calculating the conditional expected value

$$u(t,x) = \mathbb{E}\left[g(X_T) - \int_t^T f(s, X_s, Y_s, Z_s)ds \Big| X_t = x \right]. \tag{2.6}$$

Therefore, the solution of (2.1) can be reduced to the solution of the FBSDE (2.2) and (2.5). In the rest of the paper, we will present how to numerically solve the FBSDE (2.2) and (2.5). For this purpose, one usually discretizes the time interval $[0, T]$ to a time sequence $0 = t_0 < t_1 < \cdots < t_N = T$ and rewrite the FBSDE as:

$$X_{t_{i+1}} = X_{t_i} + \int_{t_i}^{t_{i+1}} b(t, X_t)dt + \int_{t_i}^{t_{i+1}} \sigma(t, X_t)dW_t, \tag{2.7}$$

and

$$Y_{t_{i+1}} = Y_{t_i} - \int_{t_i}^{t_{i+1}} f\left(t, X_t, Y_t, \sigma^{\mathrm{T}}(t, X_t)Z_t\right)dt + \int_{t_i}^{t_{i+1}} Z_t^{\mathrm{T}}\sigma(t, X_t)dW_t, \tag{2.8}$$

for all $i = 0, 1, \ldots, N-1$. In practice, (2.7) and (2.8) are often approximated by Euler-Maruyama scheme as below:

$$X_{t_{i+1}} = X_{t_i} + b(t_i, X_{t_i})\Delta t_i + \sigma(t_i, X_{t_i})\Delta W_{t_i}, \tag{2.9}$$

and

$$Y_{t_{i+1}} = Y_{t_i} - f\left(t_i, X_{t_i}, u(t_i, X_{t_i}), \sigma^{\mathrm{T}}(t_i, X_{t_i})Z_{t_i}\right)\Delta t_i + Z_{t_i}^{\mathrm{T}}\sigma(t_i, X_{t_i})\Delta W_{t_i}, \tag{2.10}$$

where $\Delta t_i = t_{i+1} - t_i$ and $\Delta W_{t_i} = W_{t_{i+1}} - W_{t_i}$.

When the dimension d is very large, it is a challenge task to solve the FBSDE (2.7) and (2.8), even their simplified form (2.9) and (2.10). Very recently, the methods based on *deep neural network* (DNN) demonstrated great power in the numerical solution of the high dimensional PDEs [13,23,30]. The basic idea of NN methods is to approximate the solution u with a NN function which can be described as

$$u_\theta(z_0) = (F_l \circ \delta \circ F_{l-1} \circ \delta \cdots \circ\ \delta \circ F_1)(z_0), z_0 = (t, x) \in \mathbb{R}^{d+1}, \tag{2.11}$$

where l is the depth of network, δ is a prescribed activation function and the affine mapping F_k is defined by

$$F_k(z_{k-1}) = w_k z_{k-1} + b_k, z_{k-1} \in \mathbb{R}^{m_{k-1}},$$

with the *weights* $w_k \in \mathbb{R}^{m_k \times m_{k-1}}$ and the *bias* $b_k \in \mathbb{R}^{m_k}$ are to-be-trained parameters, where $m_k (1 \le k \le l)$ is the width of the k-th layer.

In the following, we present two NN based methods to train the network parameters. The first one is the so-called Deep BSDE introduced in [12,17]. The purpose of the Deep BSDE solver is to compute $u(0, \xi)$ for each $\xi \in \mathbb{R}^d$. For this purpose, we construct $N-1$ different neutral networks $\pi_{\theta_i} = \pi_{\theta_i}(x), i = 1, \ldots, N-1$ to approximate the functions $\nabla u(t_i, x), i = 1, \ldots N-1$, respectively. We denote the parameters of the above NNs by $\theta' = (\theta_1, \ldots, \theta_{N-1})$. In addition, we also regard the function values $y_0 = u(0, \xi), z_0' = \nabla u(0, \xi)$ as parameters to be trained. In the following, we explain how to train the parameter (y_0, z_0', θ'). First, we randomly initialize the parameters (y_0, z_0', θ'). Then we update the parameters $\theta = (y_0, z_0', \theta')$ by using the gradient descent(GD) method to minimize the loss function

$$\mathbb{L}(\theta) = \mathbb{E}[|Y_T - g(X_T)|^2], \tag{2.12}$$

which is calculated as below. For any given $\xi \in \mathbb{R}^d$, we use the equation (2.9) to generate sufficient sequences of $X_{t_i}, i = 1, \ldots N$. Then for each sequence, we correspondingly calculate the sequence $Y_{t_i}, i = 1, \ldots, N$ iteratively using

$$Y_{t_{i+1}} = Y_{t_i} - f\left(t_i, X_{t_i}, Y_{t_i}, \sigma^{\mathrm{T}}(t_i, X_{t_i})\pi_{\theta_i}(X_{t_i})\right)\Delta t_i + \pi_{\theta_i}^{\mathrm{T}}(X_{t_i})\sigma(t_i, X_{t_i})\Delta W_{t_i}, \tag{2.13}$$

till we finally get $Y_T = Y_{t_N}$. With many calculated pairs of (X_T, Y_T), we calculate the loss (2.12). The Deep BSDE solver has shown some nice convergence property for high dimensional parabolic equations. However, the number of training parameters grows along with the time steps N. In addition, the solver is only capable of calculating one value $u(0, \xi)$ for each prescribed $\xi \in \mathbb{R}^d$.

The second NN solver for the FBSDE is the so-called FBSNN ([29]). With this method, we approximate the unknown solution $u(t,x)$ by using one neural network described by (2.11). To train the parameters of u_θ, we calculate the loss function as below. For $i = 0, 1, \cdots, N-1$, let $X_{t_i}, i = 1, \dots N$ be a random sequence generated by using the equation (2.9). We compute

$$\begin{aligned} Y_{t_{i+1}} &= u_\theta(t_{i+1}, X_{t_{i+1}}), \\ Y'_{t_{i+1}} &= Y_{t_i} - f\left(t_i, X_{t_i}, Y_{t_i}, \sigma^{\mathrm{T}}(t_i, X_{t_i}) Z_{t_i}\right) \Delta t_i + Z_{t_i}^{\mathrm{T}} \sigma(t_i, X_{t_i}) \Delta W_{t_i}, \end{aligned}$$

where $Z_{t_i} = \nabla u_\theta(t_i, X_{t_i})$ is computed by applying the *automatic differential module* on u_θ. We define the loss function by

$$\mathbb{L}(\theta) = \sum_M \sum_{i=0}^{N-1} |Y'_{t_{i+1}} - Y_{t_{i+1}}|^2 + \sum_M |Y_T - g(X_T)|^2, \tag{2.14}$$

where M is the number of the underlying Brownian motion. With this loss, we can train θ with the standard gradient descent method. Obviously, the FBSNN is capable of calculating the value of u on all points $(t,x) \in [0,T] \times \mathbb{R}^d$, and the number of parameters of the FBSNN is independent of the time steps N. Numerical experiments demonstrate that the FBSNN achieves nice convergence property for high dimensional quasi-linear parabolic equations. Some improved versions of the FBSNN have been proposed in [38].

We close this section with a short description of the above two NN methods in the view of reinforcement learning. First, the discrete FBSDE system (2.9) and (2.10) can be regarded as a finite Markov Reward Process (MRP), described by a triple $\Lambda = \langle S, P, R \rangle$, where S is a non-empty set of states $s \in [0,T] \times \mathbb{R}^d$, of which one state $s = (t,x)$ can be transited to another state $s' = (t', x')$ with a probability

$$P_{ss'} = Pr(S_{t'} = s' | S_t = (t,x)), \tag{2.15}$$

where $S_t = (t, X(t))$ is the stochastic variable at the time t, and from this transition, the system obtains a *reward* defined by

$$R_{ss'} = f(t, X_t, Y_t, Z_t)\Delta t - Z_t^{\mathrm{T}} \sigma(t, X_t) \Delta W_t. \tag{2.16}$$

Secondly, since there does not exist an explicit formula to calculate the transition probability (2.15), we randomly generate many samples of state sequences $X(t_i), i = 1, 2, \dots, N$ according to the equation (2.9), and then calculate the function u by averaging the returns of all these state sequences. In this sense, both the Deep BSDE and FBSNN can be regarded as *Monte-Carlo* type methods. Note that here, as a standard terminology of the reinforcement learning, the term *Monte-Carlo* indicates that the method involves a significant random component, and more importantly, it indicates that the method is based on averaging *complete* returns, instead of *partial* returns of the state sequences.

3. ResNet based temporal-difference methods

In this core section, we improve the NN based Monte-Carlo method in the following two aspects. First, to overcome the so-called *gradient vanishing* problem, we will replace the neural network architecture FCN with the Residual Neural Network (ResNet, [18]). Secondly and more importantly, to improve the computational efficiency, we will replace the Monte-Carlo learning with the online Temporal-Difference (TD) learning.

The ResNet used in our algorithm consists of many blocks. Each block contains two affine transformations and two activation functions, see Fig. 3.1. Consequently, the function expressed by our ResNet has the form:

$$v_\theta(z) = (F_l \circ B_{l-1} \circ \cdots \circ B_4 \circ B_2)(z_0), z_0 = (t,x) \in \mathbb{R}^{d+1}, \tag{3.1}$$

where

$$B_k(z_{k-2}) = \delta(w_k \delta(w_{k-1} z_{k-2} + b_{k-1}) + b_k) + z_{k-2}, \ k = 2, 4, \cdots, l-1.$$

Here, $b_{k-1} \in \mathbb{R}^{m_{k-1}}$, $b_k \in \mathbb{R}^{m_k}$, $w_{k-1} \in \mathbb{R}^{m_{k-1} \times m_{k-2}}$ and $w_k \in \mathbb{R}^{m_k \times m_{k-1}}$ are training parameters.

Next, we introduce TD learning for the FBSDE system (2.9) and (2.10). According to the discussion in the previous section, the FBSDE system can be regarded as an MRP. To indicate the *reward* at each time step, we in particular define

$$R_{t_{i+1}} = \begin{cases} f(t_i, X_{t_i}, Y_{t_i}, Z_{t_i})\Delta t_i - Z_{t_i}^{\mathrm{T}} \sigma(t_i, X_{t_i}) \Delta W_{t_i}. & 0 \le i \le N-1 \\ g(X_T), & i = N. \end{cases} \tag{3.2}$$

And the *return*, i.e., the long-term effect of the reward, is defined as

$$G_{t_i} = \sum_{j=i+1}^{N+1} R_{t_j}, \ 0 \le i \le N. \tag{3.3}$$

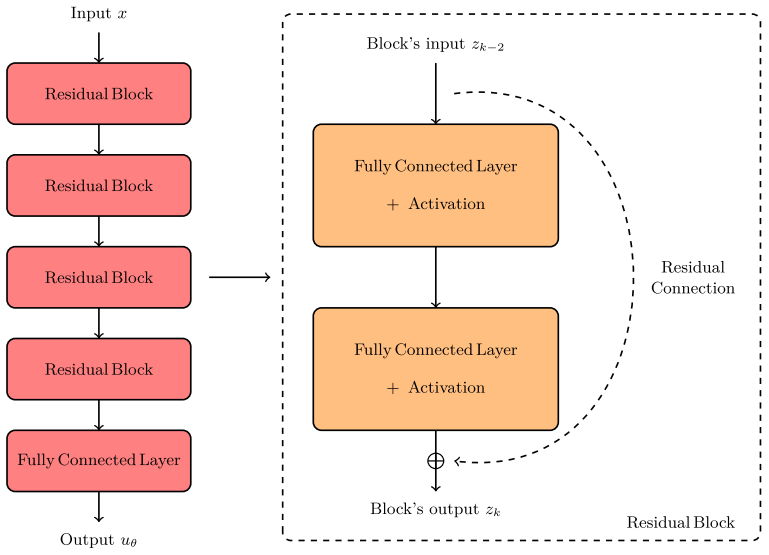

Fig. 3.1. A residual neural network with four blocks and an output linear layer.

Obviously, the return satisfies

$$G_{t_i} = R_{t_{i+1}} + G_{t_{i+1}}. \tag{3.4}$$

Moreover, we define the value function as the expected return starting at the state (t_i, x):

$$V(t_i, x) = \mathbb{E}[G_{t_i} | S_{t_i} = (t_i, x)]. \tag{3.5}$$

A fundamental property of the value function is that it satisfies the Bellman equation:

$$V(t_i, x) = \mathbb{E}[R_{t_{i+1}} + V(S_{t_{i+1}}) | S_{t_i} = (t_i, x)]. \tag{3.6}$$

Since (2.9) and (2.10) are respectively approximation of (2.7) and (2.8), the value function V is actually an approximation of u. This fact implies that we can approximate the value of the function u at the point (t_i, x) by calculating V at the state (t_i, x). Note that V is uniquely determined by the Bellman equation (3.6), theoretically we can calculate V by solving (3.6). Practically, as we have already mentioned in the previous section, it is better to calculate V by averaging the returns of many sampling trajectories starting from (t_i, x). The Monte-Carlo method is typical such a method. However, since the Monte-Carlo method updates the value of V only after all sampling sequences (trajectories) have been completed, it may suffer from the slow update speed and the low accuracy of the approximation.

The temporal-difference (TD) is also a method which averages the returns of sampling trajectories. However, different from the Monte-Carlo method, the TD method draws on the idea of dynamic programming. That is, the TD method can update V in each time step t_i, without waiting for a final outcome of each whole trajectory. The TD method uses the idea of bootstrapping: it updates the value function at the current state based on the obtained value function at other states. Precisely, with a given trajectory $\{S_{t_i}\}_{t_i \geq 0}$, we update the value function at each time t_i by

$$V(S_{t_i}) \leftarrow V(S_{t_i}) + \eta[R_{t_{i+1}} + V(S_{t_{i+1}}) - V(S_{t_i})], \tag{3.7}$$

where η is the step-size and $V(S_{t_i})$ is the current estimate of the value function. Here, $R_{t_{i+1}} + V(S_{t_{i+1}})$ is called as the *TD target* and $R_{t_{i+1}} + V(S_{t_{i+1}}) - V(S_{t_i})$ is called as the *TD error*.

Namely, we define the TD error at time t_i as

$$\mathcal{E}_i = R_{t_{i+1}} + V(S_{t_{i+1}}) - V(S_{t_i}), \quad 0 \leq i \leq N-1. \tag{3.8}$$

Note that here we use the ResNet (3.1) to present the value function, it has the form

$$V(t_i, x) = V_\theta(t_i, x) = (F_l \circ B_{l-1} \circ \cdots \circ B_4 \circ B_2)(z_0), z_0 = (t_i, x) \in \mathbb{R}^{d+1}. \tag{3.9}$$

To update V, which is determined by parameters θ, at each time step, we should construct a loss function for each time step t_i. Inspired by (3.8), we construct the first part of the loss function as below. Supposing we have generated the i-th step state $S_{t_i} = (t_i, x)$ of M (M is sufficiently large) sequences sampled from the random process X_t, we correspondingly calculate the i-th step error, which is also a random variable, which is defined by

$$Error_i(\theta) = f\left(t_i, X_{t_i}, V_\theta(S_{t_i}), \sigma_{t_i}^T \nabla V_\theta(S_{t_i})\right)\Delta t_i - \nabla V_\theta^T(S_{t_i})\sigma_{t_i}\Delta W_{t_i} + V_\theta(S_{t_{i+1}}) - V_\theta(S_{t_i}), \tag{3.10}$$

where $\sigma_{t_i} = \sigma(t_i, X_{t_i})$. Note that here, we apply the automatic differential module on $V_\theta(S_t)$ to compute $Z_t = \nabla V_\theta(S_t)$. The first part of the i-th step loss is then defined as

$$\mathbb{L}_{i,1}(\theta) = \mathbb{E}[|Error_i(\theta)|^2] = \frac{1}{M}\sum_M |Error_i(\theta)|^2. \tag{3.11}$$

The second part of our i-th step loss is related to the terminal condition $u(T,\cdot) = g$. Since at each termination state, the value network V_θ needs to meet the terminal condition, we define the termination error as:

$$\mathbb{L}_2(\theta) = \mathbb{E}[|V_\theta(S_T) - g(X_T)|^2] = \frac{1}{M}\sum_M |V_\theta(S_T) - g(X_T)|^2. \tag{3.12}$$

The third part of the i-th step loss is related to the gradient value at the termination time. We define the gradient error as

$$\mathbb{L}_3(\theta) = \mathbb{E}[|\nabla V_\theta(S_T) - \nabla g(X_T)|^2] = \frac{1}{M}\sum_M |\nabla V_\theta(S_T) - \nabla g(X_T)|^2. \tag{3.13}$$

Note that here one may argue that in the i-th ($i \le N-1$) time step, we have not generated the whole sequence S_t, so we do not have the information on the terminal state X_T and thus can not compute $\mathbb{L}_{i,2}(\theta)$. To solve this problem, we also need to use the idea of *bootstrapping*: in the initial batch-training, we may randomly generate M terminate states and in the later batch-training, we may use the terminal states of the previous batch-training. On the other hand, the error $\mathbb{L}_{i,2}(\theta)$ should be equally distributed to each time step. That is, we let $\mathbb{L}_{i,j}(\theta) = \frac{1}{N}\mathbb{L}_k(\theta), k = 2, 3$. In summary, we define the i-th step totally loss as

$$\mathbb{L}_i(\theta) = \mathbb{L}_{i,1}(\theta) + \mathbb{L}_{i,2}(\theta) + \mathbb{L}_{i,3}(\theta). \tag{3.14}$$

Given the above loss (3.14), we search the parameters θ^* such that

$$\mathbb{L}_i(\theta^*) = \min_\theta \mathbb{L}_i(\theta). \tag{3.15}$$

To this end, we use the following gradient descent iteration:

$$\theta_{i+1} = \theta_i - \alpha\nabla_\theta \mathbb{L}_i(\theta_j), \tag{3.16}$$

where α is some prescribed learning rate. We call this method as our first forward-backward stochastic equations based temporal difference method (*FBSTD1*), and it is summarized as below.

Algorithm 1: FBSTD1.

Initialization: Network parameters θ, number of time intervals N, number of walkers M, maximum number of iterations *Max_Iter*, learning rate α, a starting point $x_0 \in \mathbf{R}^d$. Let $\Delta t = T/N$, $\mathbb{L}_{i,2}(\theta) = \mathbb{L}_{i,3}(\theta) = 0$, $x_0^{(m)} = x_0, m = 1, \ldots, M$.

for $j \leftarrow 0$ **to** *Max_Iter* **do**
  $i = \mathrm{mod}(j, N)$, $t_i = t_0 + i\Delta t$.
  for $m \leftarrow 1$ **to** M **do**
    Sample $\Delta W_i^{(m)} \in \mathbb{R}^d$ from the Gauss distribution $\mathcal{N}(\mathbf{0}, \Delta t\mathbf{I})$.
    Transit the current state $s_{t_i}^{(m)} = (t_i, x_{t_i}^{(m)})$ to the next state $s_{t_{i+1}}^{(m)} = (t_{i+1}, x_{t_{i+1}}^{(m)})$ by letting $x_{t_{i+1}}^{(m)} = x_{t_i}^{(m)} + b(t_i, x_{t_i}^{(m)})\Delta t + \sigma(t_i, x_{t_i}^{(m)})\Delta W_i^{(m)}$.
    Compute $Y_{t_i}^{(m)} = V_\theta(s_{t_i}^{(m)})$, $Y_{t_{i+1}}^{(m)} = V_\theta(s_{t_{i+1}}^{(m)})$, $Z_{t_i} = \nabla_x V_\theta(s_{t_i}^{(m)})$ and the reward $R_{t_i}^{(m)}$.
  end
  Compute the loss $\mathbb{L}_{i,1}(\theta)$.
  if $i = N-1$ **then**
    Store terminal states $\{s_T^{(m)}\}_{m=1}^M = \{(T, x_N^{(m)})\}_{m=1}^M$. Compute terminal loss $\mathbb{L}_{i,2}(\theta)$ and $\mathbb{L}_{i,3}(\theta)$.
  end
  $\mathbb{L}_i(\theta) \leftarrow \mathbb{L}_{i,1}(\theta) + \mathbb{L}_{i,2}(\theta) + \mathbb{L}_{i,3}(\theta)$.
  if $\mathrm{mod}(j, 5000) = 0$ **then**
    $\alpha = \alpha/2$.
  end
  Update θ by the gradient descent iteration $\theta_{j+1} \leftarrow \theta_j - \alpha\nabla_\theta \mathbb{L}_i(\theta_j)$.
end

To avoid using the automatic differential module, one may use an additional neural network for the random process Z_t. That is, we construct two ResNet functions $V_{\theta_Y} = V_{\theta_Y}(S_t)$ and $\pi_{\theta_Z} = \pi_{\theta_Z}(S_t)$ to simulate the random processes Y_t and Z_t, respectively. In this case, the TD error at t_i is correspondingly modified to be

$$Error_i(\theta_Y, \theta_Z) = f\left(t_i, X_{t_i}, V_{\theta_Y}(S_{t_i}), \sigma_{t_i}^{\mathrm{T}} \pi_{\theta_Z}(S_{t_i})\right) \Delta t_i - \pi_{\theta_Z}^T(S_{t_i}) \sigma_{t_i} \Delta W_{t_i} + V_{\theta_Y}(S_{t_{i+1}}) - V_{\theta_Y}(S_{t_i}). \tag{3.17}$$

Similarly, we define $\mathbb{L}_{i,1}(\theta_Y, \theta_Z) = \frac{1}{M} \sum_M |Error_i(\theta_Y, \theta_Z)|^2$ and $\mathbb{L}_{i,2}(\theta_Y) = \frac{1}{N} \frac{1}{M} \sum_M |V_{\theta_Y}(S_T) - g(X_T)|^2$.

In addition, we also need to fit the gradient value at the termination time. Then we define the gradient error

$$\mathbb{L}_3(\theta_Z) = \mathbb{E}[|\pi_{\theta_Z}(S_T) - \nabla g(X_T)|^2], \tag{3.18}$$

and let $\mathbb{L}_{i,3}(\theta_Z) = \frac{1}{N} \mathbb{L}_3(\theta)$. In summary, we define the total loss at time t_i by

$$\mathbb{L}_i(\theta_Y, \theta_Z) = \mathbb{L}_{i,1}(\theta_Y, \theta_Z) + \mathbb{L}_{i,2}(\theta_Y) + \mathbb{L}_{i,3}(\theta_Z). \tag{3.19}$$

With this loss, we search the parameters θ_Y^* and θ_Z^* such that

$$\mathbb{L}_i(\theta_Y^*, \theta_Z^*) = \min_{\theta_Y, \theta_Z} \mathbb{L}_i(\theta_Y, \theta_Z). \tag{3.20}$$

To this end, we use the following iterations:

$$\theta_{Y,i+1} = \theta_{Y,i} - \alpha \nabla_{\theta_Y} \mathbb{L}_i(\theta_{Y,j}, \theta_{Z,j}),\ \theta_{Z,i+1} = \theta_{Z,i} - \beta \nabla_{\theta_Z} \mathbb{L}_i(\theta_{Y,j}, \theta_{Z,j}), \tag{3.21}$$

where α and β are the learning rates. We call this method as our second forward-backward stochastic equations based temporal difference method (*FBSTD2*) which is summarized as below.

Algorithm 2: FBSTD2.

Initialization: Network parameters θ_Y and θ_Z, number of time intervals N, number of walkers M, maximum number of iterations *Max_Iter*, learning rates α and β, a starting point $x_0 \in \mathbf{R}^d$. Let $\Delta t = T/N$, $\mathbb{L}_{i,2}(\theta_Y) = \mathbb{L}_{i,3}(\theta_Z) = 0$, $x_0^{(m)} = x_0, m = 1, \ldots, M$.

for $j \leftarrow 0$ **to** $Max\_Iter - 1$ **do**
- $i = \mathrm{mod}(j, N)$, $t_i = t_0 + i\Delta t$.
- **for** $m \leftarrow 1$ **to** M **do**
  - Sample $\Delta W_i^{(m)} \in \mathbb{R}^d$ from the Gauss distribution $\mathcal{N}(\mathbf{0}, \Delta t \mathbf{I})$.
  - Transit the current state $s_{t_i}^{(m)} = (t_i, x_{t_i}^{(m)})$ to the next state $s_{t_{i+1}}^{(m)} = (t_{i+1}, x_{t_{i+1}}^{(m)})$ by letting $x_{t_{i+1}}^{(m)} = x_{t_i}^{(m)} + b(t_i, x_{t_i}^{(m)})\Delta t + \sigma(t_i, x_{t_i}^{(m)}) \Delta W_i^{(m)}$.
  - Compute $Y_{t_i}^{(m)} = V_{\theta_Y}(s_{t_i}^{(m)})$, $Y_{t_{i+1}}^{(m)} = V_{\theta_Y}(s_{t_{i+1}}^{(m)})$, $Z_{t_i} = \pi_{\theta_Z}(s_{t_i}^{(m)})$ and $R_{t_i}^{(m)}$.
- **end**
- Compute the loss $\mathbb{L}_{i,1}(\theta_Y, \theta_Z)$.
- **if** $i = N - 1$ **then**
  - Store terminal states $\{s_T^{(m)}\}_{m=1}^M = \{(T, x_N^{(m)})\}_{m=1}^M$.
  - Compute terminal loss $\mathbb{L}_{i,2}(\theta_Y)$ and $\mathbb{L}_{i,3}(\theta_Z)$.
- **end**
- $\mathbb{L}_i(\theta_Y, \theta_Z) \leftarrow \mathbb{L}_{i,1}(\theta_Y, \theta_Z) + \mathbb{L}_{i,2}(\theta_Y) + \mathbb{L}_{i,3}(\theta_Z)$.
- **if** $\mathrm{mod}(j, 5000) = 0$ **then**
  - $\alpha = \alpha/2$; $\beta = \beta/2$.
- **end**
- Update θ_Y and θ_Z by the gradient descent iteration
  $\theta_{Y,j+1} = \theta_{Y,j} - \alpha \nabla_{\theta_Y} \mathbb{L}_i(\theta_{Y,j}, \theta_{Z,j})$, $\theta_{Z,j+1} = \theta_{Z,j} - \beta \nabla_{\theta_Z} \mathbb{L}_i(\theta_{Y,j}, \theta_{Z,j})$.

end

We remark that, in one training step of parameters updating, the FBSTD1 and FBSTD2 compute far less gradients than the Deep BSDE or the FBSNN. Consequently, the TD training is easier and often more efficient than the Deep BSDE or the FBSNN.

The above one-step TD method can be generalized to the n-step TD method, where $1 \le n \le \infty$. In n-step TD, the value function $V(S_{t_i})$ is supposed to be updated with the n-step return

$$G_{t_i:\, t_{i+n}} = R_{t_{i+1}} + R_{t_{i+2}} + \ldots + R_{t_{i+n}} + V(S_{t_{i+n}}), \tag{3.22}$$

where $0 \le i \le N - n$. Note that here, if $N - i < n$, then all the missing terms are taken as zero and the n-step return is defined to be equal to the common full return, that is $G_{t_i:t_{i+n}} = G_{t_i}$. Correspondingly, the i-th time step error for the n-step TD method is defined by:

$$\begin{aligned} \mathcal{E}_i(n) &= G_{t_i:t_{i+n}} - V(S_{t_i}) \\ &= \sum_{j=i}^{i+n-1} (f(t_j, X_{t_j}, Y_{t_j}, \sigma_{t_j}^{\mathrm{T}} Z_{t_j}) \Delta t_j - Z_{t_j}^T \sigma_{t_j} \Delta W_{t_j}) + Y_{t_{i+n}} - Y_{t_i}. \end{aligned} \tag{3.23}$$

With this approach, the one-step methods FBSTD1 and FBSTD2 can also be generalized to the n-step methods FBSTD1(n) and FBSTD2(n), respectively. In fact, for FBSTD1(n), we use one ResNet function $V_\theta(S_t)$ to approximate Y_t and apply the automatic differential module $\nabla V_\theta(S_t)$ to simulate Z_t, and we define the loss function at time t_i as

$$\mathbb{L}_i^n(\theta) = \frac{1}{M}\sum_M \left| \sum_{j=i}^{i+n-1} (f(S_{t_j}, V_\theta(S_{t_j}), \sigma_{t_j}^{\mathrm{T}} \nabla V_\theta(S_{t_j}))\Delta t_j - \nabla V_\theta^T(S_{t_j})\sigma_{t_j}\Delta W_{t_j}) + V_\theta(S_{t_{i+n}}) - V_\theta(S_{t_i}) \right|^2 + \frac{1}{N}\frac{1}{M}\sum_M |V_\theta(S_{t_j}) - g(X_T)|^2 + \frac{1}{N}\frac{1}{M}\sum_M |\nabla V_\theta(S_T) - \nabla g(X_T)|^2, \tag{3.24}$$

where M is the number of the underlying Brownian motion and N is the number of time intervals.

For FBSTD2(n), we use two different ResNets $V_{\theta_Y} = V_{\theta_Y}(S_t)$ and $\pi_{\theta_Z} = \pi_{\theta_Z}(S_t)$ to simulate Y_t and Z_t, respectively, and we define the loss function at time t_i as

$$\mathbb{L}_i^n(\theta_Y, \theta_Z) = \frac{1}{M}\sum_M \left| \sum_{j=i}^{i+n-1} (f(S_{t_j}, V_{\theta_Y}(S_{t_j}), \sigma_{t_j}^T \pi_{\theta_Z}(S_{t_j}))\Delta t_j - \pi_{\theta_Z}^T \sigma_{t_j}\Delta W_{t_j}) + V_{\theta_Y}(S_{t_{i+n}}) - V_{\theta_Y}(S_{t_i}) \right|^2 + \frac{1}{N}\frac{1}{M}\sum_M |V_{\theta_Y}(S_T) - g(X_T)|^2 + \frac{1}{N}\frac{1}{M}\sum_M |\pi_{\theta_Z}(S_T) - \nabla g(X_T)|^2. \tag{3.25}$$

We remark that both the Monte Carlo method and the one-step TD method can be regarded as special cases of n-step TD. In fact, the Monte-Carlo method is the case that $n = \infty$. In practice, there may exist a proper n_0 which is different from 1 and ∞ such that n_0-step TD is better than both the Monte-Carlo and the one-step TD method.

4. Numerical results

We will test the performance of our proposed TD methods on Black-Scholes, Hamilton-Jacobi-Bellman, Quadratically Growing, and Reaction-Diffusion equations. In all our tests, we will use a ResNet which has 2 residual blocks and 256 neurons per layer. The weights and the biases in the network are initialized with quantities randomly sampled from a uniform distribution, without any pre-training. We set the initial learning rates $\alpha = \beta = 0.005$, $d = 100$, $T = 1$ and choose $M = 512$. We use Adam optimizer [20] to update the parameters. The accuracy of a computed approximation V to the exact solution u is indicated by the *relative errors* defined by

$$RE = \frac{\|V - u\|_{L^2([0,T]\times\mathbb{R}^d)}}{\|u\|_{L^2([0,T]\times\mathbb{R}^d)}}, \quad RE_0 = \frac{|V(0, X_0) - u(0, X_0)|}{|u(0, X_0)|}.$$

All numerical experiments are implemented using Python with the library Torch on a machine equipped with an Nvidia GeForce GTX 2080 Ti GPU.

In the first three examples, we will test the performance of the *one*-step TD methods. For comparison, we will also report the numerical results computed by the Deep BSDE and the FBSNN. The numerical results demonstrate that given a fixed number of training steps, both our FBSTD1 and FBSTD2 have higher accuracy and less computational time than the Deep BSDE or the FBSNN. In addition, in the first example, we test the impact of different activation functions and NN architectures on the performance of the FBSTD. Our experiments show that the sin activation function and the ResNet often lead better performance than other choices. In the third example, we test the impact of the width a network and the number of time intervals N on the performance of a NN method. We observe that the accuracy might not be increasingly enhanced as N or the width of the NN increases. In the fourth example, we test the impact of n, the number of TD steps, on the performance of a TD method. We find that the four-step TD algorithm behaves better than both the Monte-Carlo and the one-step TD method in this example.

4.1. Black-Scholes equation

We consider

$$\begin{cases} \dfrac{\partial u}{\partial t}(t,x) = -\dfrac{1}{2}\mathrm{Tr}[0.16\mathrm{diag}(x^2)\mathrm{Hess}_x u(t,x)] + 0.05(u(t,x) - (\nabla u(t,x), x)), \\ u(T,x) = \|x\|^2, \end{cases} \tag{4.1}$$

which admits an exact solution

$$u(x,t) = \exp((0.05 + 0.4^2)(T-t))\|x\|^2.$$

This equation has been discussed in [29]. Following the idea of Section 2, the deterministic equation (4.1) can be transformed to the FBSDE

$$\begin{cases} dX_t = 0.4\mathrm{diag}(X_t)dW_t, & X_0 = \xi \in R^d, \\ dY_t = 0.05(Y_t - Z_t^{\mathrm{T}} X_t)dt + 0.4 Z_t^{\mathrm{T}}\mathrm{diag}(X_t)dW_t, & Y_T = \|X_T\|^2. \end{cases} \tag{4.2}$$

Table 4.1
Errors and computation time for (4.1).

Method	RE	RE_0	Time (s)
Deep BSDE	–	0.34%	6825
FBSNN	0.63%	0.65%	6042
FBSTD1	0.15%	0.12%	718
FBSTD2	0.19%	0.23%	748

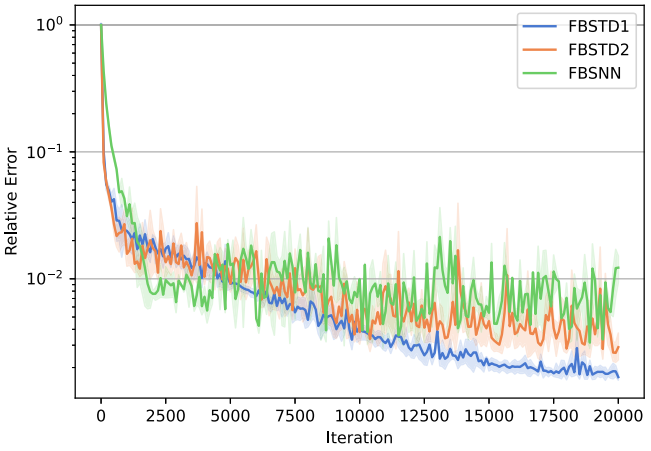

Fig. 4.1. Errors w.r.t. iteration steps for (4.1).

We use FBSTD1, FBSTD2, the Deep BSDE and the FBSNN to simulate the FBSDE (4.2). We partition the time domain [0, T] into $N = 50$ equally sized intervals. In each training step, we sample $M = 512$ trajectories according to the FSDE $dX_t = 0.4\text{diag}(X_t)dW_t$ with the starting point $\xi = (1, 0.5, ..., 1, 0.5) \in R^d$.

Presented in Table 4.1 are the relative errors of different methods after 20000 iterative steps (i.e. the parameters of the NN approximate function have been updated 20000 times). From this table, we observe that the TD methods FBSTD1 and FBSTD2 improve significantly the accuracy of the Monte-Carlo type methods Deep BSDE and FBSNN, using greatly reduced computation time. It is interesting to notice that for this example, the performance of FBSTD1 is a little bit better than FBSTD2.

Depicted in Fig. 4.1 is the dynamic change of the relative errors along with the increase of iteration steps for different methods. Again we find that the FBSTD1 behaves better than FBSNNs and FBSTD2: as the number of iteration steps increases, the relative error RE of the FBSTD1 decreases steadily, while the RE of the FBSTD2 and FBSNNs decrease too, but with significantly larger oscillation. We can also find for sufficiently large iteration steps, the RE of the FBSDT1 is less than that of the FBSTD2, and that of the FBSTD2 is much less than that of FBSNNs.

Depicted in Fig. 4.2 are the exact solution and the approximate solution computed with FBSTD1 on 5 randomly generated trajectories. The red line indicates the exact solution and the blue line indicates the approximate solution. From this Figure, we get a direct impression that the FBSTD1 approximate solution is in very good agreement with the exact solution on these trajectories. Since these 5 trajectories are randomly generated, we believe that the FBSTD1 solution can predict the exact solution very well everywhere anytime.

Depicted in Fig. 4.3 is the efficiency of different activation functions including the $\sin, \cos$, Relu [25], Leakly Relu [24], Hyperbolic tangent and Silu [11]. After running FBSTD1 with 20000 iteration steps, the relative errors corresponding to the activation function $\sin, \cos$, Relu, Leakly Relu, hyperbolic tangent and Silu are 0.15%, 0.24%, 0.61%, 0.65%, 0.71% and 0.46%, respectively. It seems that, the sin activation function works best among all activation functions. Therefore, in the rest 3 numerical examples, we will always choose the activation function to be the sin function.

Presented in Table 4.2 and Fig. 4.4 is a comparison of two different neural network architectures: the FCN and the ResNet. From Table 4.2, we observe that no matter we apply FBSTD1 or FBSTD2, the ResNet yields better accuracy than the FCN. Therefore, in the rest 3 numerical examples, we will always choose the architecture to be the ResNet.

4.2. Hamilton-Jacobi-Bellman equation

We consider the equation

$$\begin{cases} \dfrac{\partial u}{\partial t}(t,x) + \Delta u(t,x) - \lambda \|\nabla u(t,x)\|^2 = 0, \\ u(T,x) = \ln\left(\dfrac{1+\|x\|^2}{2}\right), \end{cases} \tag{4.3}$$

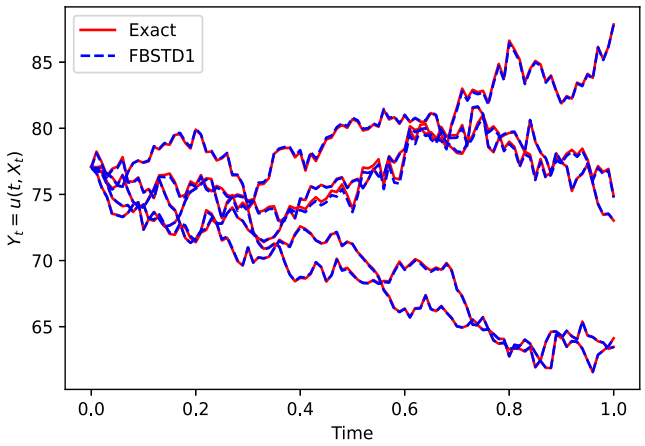

Fig. 4.2. The fit of FBSTD1 solution to the exact solution of (4.1). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

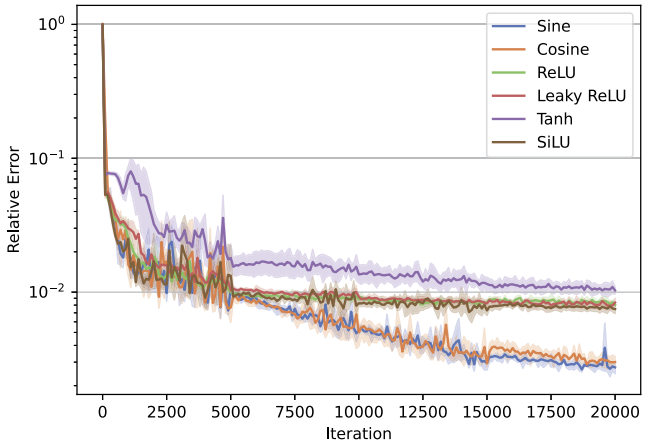

Fig. 4.3. Errors of FBSTD1 (different activation functions) w.r.t. iteration steps.

Table 4.2
Errors by different NN architectures.

Network architecture	RE	RE_0
FBSTD1-FCN	0.46%	0.27%
FBSTD2-FCN	0.26%	0.11%
FBSTD1-ResNet	0.15%	0.12%
FBSTD2-ResNet	0.19%	0.23%

which admits the exact solution

$$u(t,x) = -\frac{1}{\lambda}\ln\left(\mathbb{E}\left[\exp(-\lambda g(x+\sqrt{2}W_{T-t}))\right]\right),$$

where $\|\nabla u(t,x)\| = \sqrt{\sum_{i=1}^{d}\left(\frac{\partial u}{\partial x_i}(t,x)\right)^2}$ and for simplicity, here we choose $\lambda = 1$. This equation has been discussed in [4], [9], and [12]. Following Section 2, we transform (4.3) to the FBSDE

$$\begin{cases} dX_t = \sqrt{2}dW_t, & X_0 = \xi \in R^d, \\ dY_t = \|Z_t\|^2 dt + \sqrt{2}Z_t^{\mathrm{T}} dW_t, & Y_T = g(X_T). \end{cases} \tag{4.4}$$

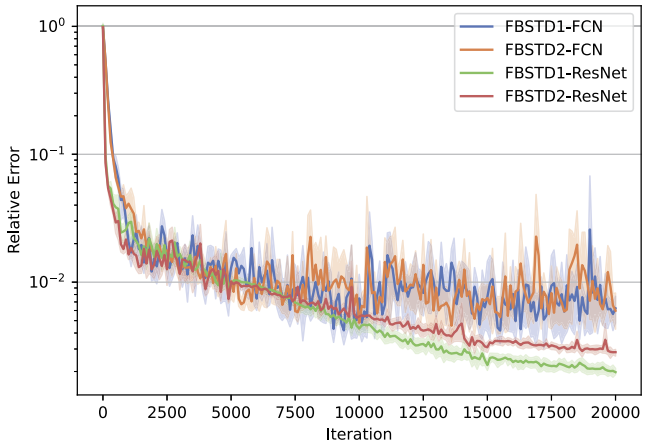

Fig. 4.4. Errors of different architectures w.r.t. iteration steps.

Table 4.3
Errors and computation time for (4.3).

Method	RE	RE_0	Time (s)
Deep BSDE	–	0.45%	8241
FBSNN	0.49%	0.18%	7680
FBSTD1	0.53%	0.29%	514
FBSTD2	0.31%	0.06%	624

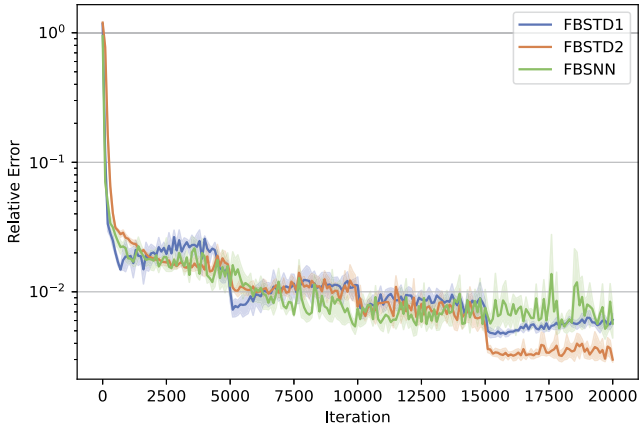

Fig. 4.5. Errors w.r.t. iteration steps for (4.3).

We solve (4.4) using FBSTD1, FBSTD2, Deep BSDE and FBSNN, with $N = 50$ and the starting point $\xi = (0, 0, ..., 0, 0) \in R^d$. The relative errors RE, RE_0 and the computational time after 20000 iterations are presented in Table 4.3, from which we find that the FBSTD2 yields better accuracy than the other three methods.

The dynamic change of relative errors with respect to the increase of the iteration steps for FBSTD1, FBSTD2 and FBSNN is shown in Fig. 4.5. We find that the FBSTD2 behaves better than the other two methods when the number of iteration steps is sufficiently large. In addition, we present the average relative errors of 100 randomly generated trajectories in Fig. 4.6. It seems that for relatively small time t, FBSTD2 has smaller RE than FBSTD1 and FBSNN, which implies that FBSTD2 has better prediction than the other two methods. Again, we find that the computational time for the TD methods FBSTD1 and FBSTD2 is significantly less than that for the Monte-Carlo method FBSNN.

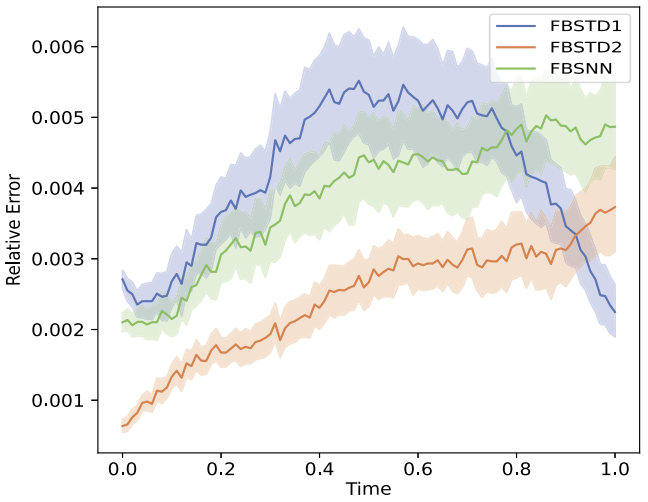

Fig. 4.6. Average errors of 100 random trajectories for (4.3).

Table 4.4
Errors and computation time for (4.5).

Method	RE	RE_0	Time (s)
Deep BSDE	–	0.47%	7637
FBSNN	0.42%	0.34%	6902
FBSTD1	0.33%	0.56%	458
FBSTD2	0.54%	0.31%	454

4.3. Quadratically growing equation

We consider the high-dimensional equation

$$\begin{cases} \dfrac{\partial u}{\partial t}(t,x) + \|\nabla u(t,x)\|^2 + \dfrac{1}{2}\Delta u(t,x) = h(t,x,u,\nabla u), \\ u(T,x) = \sin\left(\left[\dfrac{1}{d}\|x\|^2\right]^\alpha\right), \end{cases} \tag{4.5}$$

where the right-hand side function

$$h(t,x,u,\nabla u) = 2\|x\|^2 d^{-2}\left\{\alpha^2 l^{2\alpha-2}[2\cos^2 l^\alpha - \sin l^\alpha] + \alpha(\alpha-1)l^{\alpha-2}\cos(l^\alpha)\right\},$$

with $l = l(t,x) = T - t + \frac{1}{d}\|x\|^2$. This equation admits the exact solution

$$u(t,x) = \sin\left(\left[T - t + \frac{1}{d}\|x\|^2\right]^\alpha\right),$$

and it has been discussed in [12,14]. For simplicity, here we let the parameter $\alpha = 0.4$. This equation can be transformed to the FBSDE

$$\begin{cases} dX_t = dW_t, & X_0 = \xi, \\ dY_t = (h(t,X_t,Y_t,Z_t) - \|Z_t\|^2)dt + Z_t^{\mathrm{T}} dW_t, & Y_T = g(X_T). \end{cases} \tag{4.6}$$

We choose the starting point $\xi = (0,0,\ldots,0,0) \in R^d$ and partition the time domain [0, T] into $N = 30$ equally spaced intervals. We apply our one-step TD algorithm, the Deep BSDE and the FBSNN to this problem. We present the relative errors of different methods in Table 4.4 and depict the dynamic changes of relative errors versus iteration number in Fig. 4.7. It can be seen that the training trends of FBSTD1 and FBSTD2 are more stable than the other methods. In addition, compared to other 3 methods, the FBSTD1 achieves higher accuracy and has the fastest convergence speed.

We present the exact solution and the approximate solution computed with FBSTD1 on 5 randomly generated trajectories in Fig. 4.8. The red line indicates the exact solution and the blue line indicates the approximate solution. From this figure, we observe that the FBSTD1 approximate solution is in good agreement with the exact solution on these trajectories, which implies that the FBSTD1 solution can predict the exact solution.

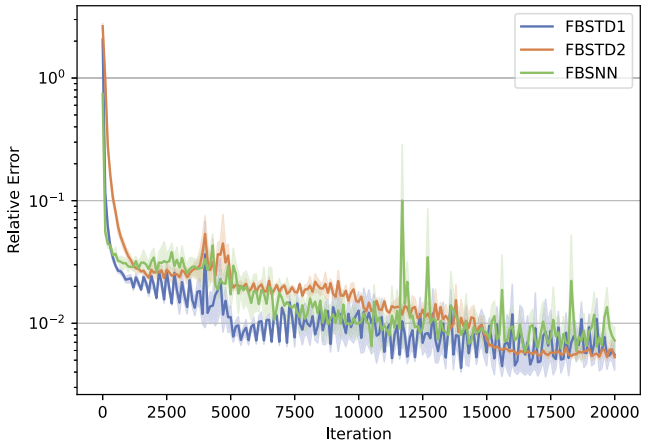

Fig. 4.7. Errors w.r.t. iteration steps for (4.5).

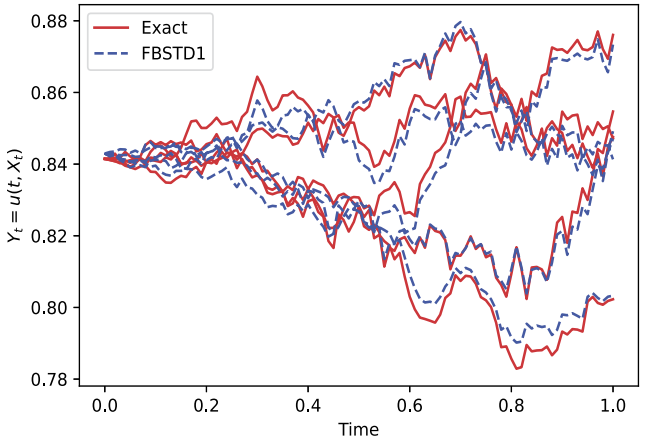

Fig. 4.8. The fit of FBSTD1 solution to exact solutions for (4.5).

Table 4.5
Errors of FBSTD1 with different width and N for (4.5).

Width	20000 iterations			30000 iterations			40000 iterations		
	$N = 20$	$N = 40$	$N = 80$	$N = 20$	$N = 40$	$N = 80$	$N = 20$	$N = 40$	$N = 80$
128	1.54%	1.59%	2.42%	1.11%	0.97%	2.08%	1.05%	0.93%	2.04%
256	1.45%	2.18%	2.13%	0.56%	1.93%	1.23%	0.45%	1.78%	1.06%
512	1.33%	2.22%	1.53%	0.48%	1.15%	0.69%	0.43%	0.92%	0.59%
1024	4.51%	1.84%	2.22%	2.11%	0.75%	2.01%	2.09%	0.68%	1.96%

We conduct some numerical experiments to demonstrate the impact of the width of the NN and the number of time intervals N. With fixed block numbers 2, we report the REs in Table 4.5. We observe that the accuracy of the FBSTD1 solution does not monotone improved as N goes up from 20 to 80. Similarly, the accuracy might also decrease as the width of the network goes up from 128 to 512, which is in contradict with our direct impression that the accuracy will be in a declining trend when the width constantly rises. The reason for this phenomenon might be that too many parameters lead overfitting of the training samples.

4.4. Reaction-diffusion equation

We consider the equation:

$$\begin{cases} \dfrac{\partial u}{\partial t}(t,x) + \dfrac{1}{2}\Delta u(t,x) - \min\left\{1, \left[u(t,x) - k - 1 - q(x)\exp\left(\dfrac{t-T}{2d}\right)\right]^{\lambda}\right\} = 0, \\ u(T,x) = k + 1 + q(x), \end{cases} \tag{4.7}$$

Table 4.6
Errors and computation time for (4.7).

Method	RE	RE_0	Time (s)
Deep BSDE	–	0.065%	7982
FBSNN	0.12%	0.872%	6791
FBSTD1 (4)	0.07%	0.004%	812
FBSTD2 (4)	0.08%	0.002%	804

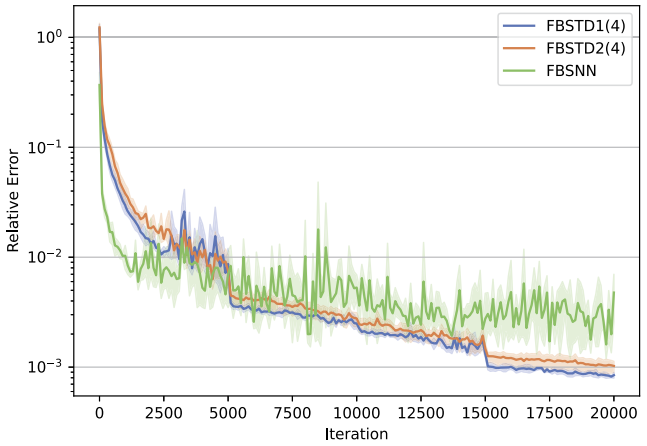

Fig. 4.9. Errors w.r.t. iteration steps for (4.7).

Table 4.7
Errors with different n for (4.7).

n	1	2	3	4	5
FBSTD1(n)	0.24%	0.11%	0.09%	0.07%	0.17%
FBSTD2(n)	0.15%	0.09%	0.09%	0.08%	0.16%

which admits the exact solution

$$u(t,x) = k + 1 + q(x)\exp\left(\frac{t-T}{2d}\right),$$

where $q(x) = \cos\left(\frac{1}{d}\sum_{i=1}^{d} x_i\right)$, $k = 0.5$ and $\lambda = 3$. We transform (4.7) to the FBSDE

$$\begin{cases} dX_t = dW_t, & X_0 = \xi \in R^d, \\ dY_t = f(t, X_t, Y_t, Z_t)dt + Z_t^{\mathrm{T}} dW_t, & Y_T = g(X_T), \end{cases} \tag{4.8}$$

where $f(t, X_t, Y_t, Z_t) = \min\left\{1, \left[Y_t - k - 1 - q(X_{i,t})\exp\left(\frac{t-T}{2d}\right)\right]^3\right\}$ and $X_{i,t}$ is the i-th component of X_t. We choose the start point $\xi = (0, 0, ..., 0, 0) \in R^d$ and partition the time domain [0, T] into $N = 50$ equally spaced intervals.

We apply our four-step TD algorithm, the Deep BSDE and the FBSNN to this problem. The numerical results are shown in Table 4.6 and the dynamic change of relative errors versus iteration number is depicted in Fig. 4.9. It can be observed that as the number of iteration steps increases, the RE of the FBSTD1(4) and FBSTD2(4) decrease steadily, while the RE of the FBSNN decreases too, but with relatively larger oscillation. Moreover, the performance of FBSTD1(4) is a little bit better than FBSTD2(4). Not surprisingly, FBSTD1(4) and FBSTD2(4) cost less time than other methods.

We believe that a proper selected n may lead better numerical results than those obtained by the MC method or the one-step TD method. In the following, we test the performance of FBSTD1 and FBSTD2 with different n. We report the numerical results in Table 4.7 from $n = 1$ to $n = 5$. It illustrates that as n goes up, the accuracy of the predicted solution is not increasingly enhanced. The proposed FBSTD1 and FBSTD2 methods reach the minimum error value at $n = 4$.

5. Concluding remarks

In this paper, we propose a TD based DL method, the so-called FBSTD, for solving high-dimensional PDEs. The key idea of the FBSTD is to train a neural network function in each time step by using the TD learning method. To this end, we first

need to transform the original PDE into an Markov reward process (MRP). Then we use a neural network to approximate the original solution, and another neural network or the auto-differentiation module to approximate the gradient of the solution. The associated NN function(s) are trained in each time step by using the loss function designed with the temporal-difference error of one-step or multi-steps. Numerical examples demonstrate that the FBSTD has higher accuracy and less computational cost than some other FBSDE solvers. Our ongoing research subjects include to generalize our method to the solution of non time-evolution equations on a bounded domain.

CRediT authorship contribution statement

Shaojie Zeng: Methodology, Programming, Writing – Original Draft, Review and Editing. **Yihua Cai:** Methodology, Programming, Visualization. **Qingsong Zou:** Conceptualization, Writing – Original Draft, Review and Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

The research was supported in part by the National Natural Science Foundation of China under grant 12071496, by the Key-Area Research and Development Program of Guangdong Province grant No. 2021B0101190003, by the Guangdong Basic and Applied Basic Research Foundation under grant 2022A1515012106, and by the Guangdong Provincial Key Laboratory of Computational Science at Sun Yat-sen University grant 2020B1212060032.

References

[1] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: a survey, J. Mach. Learn. Res. 18 (153) (2018) 1–43.
[2] R.E. Bellman, Dynamic Programming. Rand Corporation Research Study, Princeton University Press, 1967.
[3] C. Beck, S. Becker, P. Cheridito, et al., Deep splitting method for parabolic PDEs, SIAM J. Sci. Comput. 43 (5) (2021) A3135–A3154.
[4] J.-F. Chassagneux, A. Richou, Numerical simulation of quadratic BSDEs, Ann. Appl. Probab. 26 (1) (2016) 262–304.
[5] Q. Chan-Wai-Nam, J. Mikael, X. Warin, Machine learning for semi linear PDEs, J. Sci. Comput. 79 (3) (2019) 16671712.
[6] P. Cheridito, H.M. Soner, N. Touzi, N. Victoir, Second-order backward stochastic differential equations and fully nonlinear parabolic PDEs, Commun. Pure Appl. Math. 60 (2007) 1081–1110.
[7] T. Chen, H. Chen, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Trans. Neural Netw. 4 (6) (1993) 910–918.
[8] Z. Chen, Y. Liu, H. Sun, Physics informed learning of governing equations from scarce data, Nat. Commun. 12 (2021) 6136.
[9] L. Debnath, Nonlinear Partial Differential Equations for Scientists and Engineers, 3rd edition, Birkhauser/Springer, New York, 2012.
[10] V. Dwivedi, B. Srinivasan, Physics informed extreme learning machine (PLELM) - a rapid method for the numerical solution of partial differential equations, Neurocomputing 391 (2020) 96–118.
[11] S. Elfwinga, E. Uchibeab, K. Doyab, Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, Neural Netw. 107 (2018) 3–11.
[12] W. E, J. Han, A. Jentzen, Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations, Commun. Math. Stat. 5 (4) (2017) 349380.
[13] W. E, B. Yu, The Deep Ritz Method: a deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (2018) 1–12.
[14] E. Gobet, P. Turkedjiev, Linear regression MDP scheme for discrete backward stochastic differential equations under general conditions, Math. Comput. 85 (299) (2016) 1359–1391.
[15] B. Güler, A. Laignelet, P. Parpas, Towards robust and stable deep learning algorithms for forward backward stochastic differential equations, arXiv preprint, arXiv:1910.11623, 2019.
[16] M. Germain, H. Pham, X. Warin, Deep backward multistep schemes for nonlinear PDEs and approximation error analysis, arXiv preprint, arXiv:2006.01496, 2020.
[17] J. Han, A. Jentzen, W. E, Solving high-dimensional partial differential equations using deep learning, Proc. Natl. Acad. Sci. USA 115 (34) (2018) 85058510.
[18] K. He, X. Zhang, S. Ren, et al., Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[19] C. Huré, H. Pham, X. Warin, Deep backward schemes for high-dimensional nonlinear PDEs, Math. Comput. 89 (324) (2020) 15471579.
[20] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv:1412.6980, 2014.
[21] H. Kurt, Approximation capabilities of multilayer feedforward networks, Neural Netw. 4 (2) (1991) 251–257.
[22] H. Kurt, T. Maxwell, W. Halbert, Multilayer feedforward networks are universal approximators (PDF), Neural Netw. 2 (1989) 359–366.
[23] Y. Liao, P. Wang, Deep Nitsche method: Deep Ritz method with essential boundary conditions, Commun. Comput. Phys. 29 (2021) 1365–1384.
[24] A.L. Maas, A.Y. Hannun, A.Y. Ng, Rectifier nonlinearities improve neural network acoustic models, Int. Conf. Mach. Learn. 30 (1) (2013).
[25] V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, Int. Conf. Mach. Learn. 807 (814) (2010).
[26] G. Pang, M. D'Elia, M. Parks, G.E. Karniadakis npinns, Nonlocal physics-informed neural networks for a parametrized nonlocal universal Laplacian operator. Algorithms and applications, J. Comput. Phys. 422 (1) (2020) 109760.

[27] E. Pardoux, S. Peng, Backward stochastic differential equations and quasilinear parabolic partial differential equations, in: Stochastic Partial Differential Equations and Their Applications, Springer, Berlin, Heidelberg, 1992, pp. 200–217.
[28] A.A. Ramabathiran, P. Ramachandran, SPINN: sparse, physics based, and partially interpretable neural networks for PDEs, J. Comput. Phys. 445 (15) (2021) 110600.
[29] M. Raissi, Forward-backward stochastic neural networks: deep learning of high-dimensional partial differential equations, arXiv:1804.07010, 2018.
[30] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.
[31] F. Rossi, B. Conan-Guez, Functional multi-layer perceptron: a non-linear tool for functional data analysis, Neural Netw. 18 (1) (2005) 45–60.
[32] H. Sheng, C. Yang, PFNN: a penalty-free neural network method for solving a class of second-order boundary-value problem on complex geometries, J. Comput. Phys. 428 (2021) 110085.
[33] J. Sirignano, K. Spiliopoulosb, DGM: a deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018) 1339–1364.
[34] A. Takahashi, Y. Tsuchida, T. Yamada, A new efficient approximation scheme for solving high dimensional semi-linear PDEs: control variate method for Deep BSDE solver, J. Comput. Phys. 454 (1) (2022).
[35] L. Yang, D. Zhang, Physics informed generative adversarial networks for stochastic differential equations, SIAM J. Sci. Comput. 42 (1) (2020) A292–A317.
[36] L. Yang, X. Meng, G.E. Karniadakis, BPINNs: Bayesian physics informed neural networks for forward and inverse PDE problems with noisy data, J. Comput. Phys. 425 (2021) 109913.
[37] Y. Yang, P. Perdikaris, Adversarial uncertainty quantification in physics informed neural net works, J. Comput. Phys. 394 (2019) 136152.
[38] W. Zhang, W. Cai, FBSDE based neural network algorithms for high-dimensional quasilinear parabolic PDEs, J. Comput. Phys. (2021).
[39] Y. Zang, G. Bao, X. Ye, H. Zhou, Weak adversarial networks for high-dimensional partial differential equations, J. Comput. Phys. 411 (2020) 109409.