Quantitative Finance, 2021
Vol. 21, No. 1, 45–67, https://doi.org/10.1080/14697688.2020.1788219

Deep neural network framework based on backward stochastic differential equations for pricing and hedging American options in high dimensions

YANGANG CHEN *† and JUSTIN W. L. WAN ‡

†Department of Applied Mathematics, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada
‡David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada

(*Received 1 April 2019; accepted 19 June 2020; published online 8 September 2020*)

We propose a deep neural network framework for computing prices and deltas of American options in high dimensions. The architecture of the framework is a sequence of neural networks, where each network learns the difference of the price functions between adjacent timesteps. We introduce the least squares residual of the associated backward stochastic differential equation as the loss function. Our proposed framework yields prices and deltas for the entire spacetime, not only at a given point (e.g. $t = 0$). The computational cost of the proposed approach is quadratic in dimension, which addresses the curse of dimensionality issue that state-of-the-art approaches suffer. Our numerical simulations demonstrate these contributions, and show that the proposed neural network framework outperforms state-of-the-art approaches in high dimensions.

Keywords: American options; Delta hedging; Neural network; Stochastic differential equations

1. Introduction

American options are among the most common derivatives in financial markets. In practical applications of hedging, we are required to compute not only an American option price, but also the derivatives of a price with respect to the underlying asset prices, called American option delta (Hull 2003). Numerous approaches have been proposed for solving American option problems, such as binomial trees (Hull 2003), numerically solving partial differential equations (PDEs) with free boundary conditions or with penalty terms (Forsyth and Vetzal 2002, Achdou and Pironneau 2005, Duffy 2006), regression-based methods (Tsitsiklis and Van Roy 1999, Longstaff and Schwartz 2001, Kohler 2010), stochastic mesh methods (Broadie and Glasserman 2004), etc. When the dimension of an American option, i.e. the number of underlying assets, is greater than 3, numerical solution of PDEs is infeasible, as the complexity grows exponentially with the dimension. When the dimension d is moderate (e.g. $d \leq 20$), the regression-based Longstaff-Schwartz method (Longstaff and Schwartz 2001) is widely considered as the state-of-the-art approach for computing option prices. In addition, one can combine the Longstaff-Schwartz method with the methods proposed in Broadie and Glasserman (1996), Bouchard and Warin (2012) and Thom (2009) to compute corresponding option deltas. We note that these approaches only compute option prices and deltas at a given point (e.g. $t = 0$).† However, we emphasize that price and delta at a given point are insufficient for a complete delta hedging process, which requires computing prices and deltas for the entire spacetime (see Hull 2003, He *et al.* 2006, Kennedy *et al.* 2009, for explanations and concrete examples). Furthermore, for the

*Corresponding author. Email: y493chen@uwaterloo.ca

† Although one may consider using the Longstaff-Schwartz regressed values as an estimate of the spacetime prices, figure 1 in Bouchard and Warin (2012) shows that using such regressed values as the spacetime solution is inaccurate. Alternatively, one may consider applying the Longstaff-Schwartz method repeatedly on all the spacetime points, where *every* point requires $M \to \infty$ samples. However, this is expensive.

Longstaff-Schwartz method, a set of χth degree polynomials is normally used as the basis for regression, which leads to χth degree complexity (rather than exponential complexity). However, χ is required to go to infinity for convergence (Longstaff and Schwartz 2001, Stentoft 2004), which still results in a high complexity.

In this paper, we propose a deep neural network framework for solving high-dimensional American option problems. The major contributions of the proposed neural network framework are summarized as follows:

- Our proposed approach can evaluate both option prices and option deltas for the entire spacetime, not only at a given point, which further enables a complete hedging simulation.
- We propose a novel neural network architecture and incorporate the domain knowledge of American options into our network, such that our formulation yields accurate prices and deltas. Assuming that there are N discrete timesteps, we design a sequence of N recursively-defined feedforward neural networks,† where each network extracts the difference between the price functions of adjacent timesteps. This architecture provides a good initial state that is close to the exact solution even before the training starts, which makes a critical contribution to the accuracy of our formulation. The domain knowledge we leverage includes smoothing the payoff at $t = T$, and adding the payoff and the previous continuation price as features, etc. These additional techniques further improve the accuracy of the proposed formulation.
- We introduce the least squares residual of the associated backward stochastic differential equation (BSDE) as the loss function of neural networks. BSDE couples prices and deltas in one single equation, and thus evaluates both prices and deltas accurately.
- The computational cost of the proposed neural network framework grows quadratically with the dimension d, in contrast to exponential growth as in the Longstaff-Schwartz method. In particular, our approach outperforms the Longstaff-Schwartz method when $d \geq 20$, in the sense that our proposed approach solves American option prices and deltas in as high as 200 dimension, while the Longstaff-Schwartz method fails to solve the problems due to the out-of-memory error and the worse-than-quadratic cost.

We note that this paper is not the only neural network framework for American option problems. Early research of neural networks in American options can be found in Kohler *et al.* (2010) and Haugh and Kogan (2004). They consider using one-hidden-layer (shallow) feedforward neural networks for option pricing. However, the highest dimension considered in their numerical simulations is 10. Very recently, several types of deep neural network approaches were proposed in Sirignano and Spiliopoulos (2018), E *et al.* (2017), Beck *et al.* (2017), Han *et al.* (2018), Fujii *et al.* (2017), Huré *et al.* (2019) and Becker *et al.* (2019a, 2019b). They suggest that increasing the depth of neural networks is important in pushing the solutions to higher dimensions. Similar to these approaches, our proposed framework is also a deep neural network approach. However, we emphasize that there are a few key differences between our proposed approach and the other deep neural network approaches.

- *Different computed quantities:* Our approach computes American option prices and deltas for the entire spacetime. The approach in Sirignano and Spiliopoulos (2018) computes prices but not deltas. The approaches in E *et al.* (2017), Beck *et al.* (2017) and Han *et al.* (2018) only consider European option prices, noting that European options are easier to price than American options. Although Fujii *et al.* (2017) extends their methods to American options, the authors only compute the price at a given point. In particular, we emphasize that only our paper discusses and simulates hedging options, which is beyond merely pricing options.
- *Different network architectures:* Our network architecture is a chain of recursively-defined networks that learn the difference of the price functions between adjacent timesteps, which yields accurate computed prices and deltas; Sirignano and Spiliopoulos (2018) uses a long short-term neural network that learns the price function itself; E *et al.* (2017), Beck *et al.* (2017), Han *et al.* (2018) and Fujii *et al.* (2017) consider a chain of isolated, independent feedforward networks.
- *Different loss functions:* The approach in Sirignano and Spiliopoulos (2018) defines the loss function by the residual of the Hamilton-Jacobi-Bellman partial differential equation emerging from the Black-Scholes theory. It involves computing the Hessian of the output price function, which is expensive in both time and memory, and is difficult to implement. Our framework uses the residual of one single BSDE as the loss function, which avoids computing the Hessian. The approaches in E *et al.* (2017), Beck *et al.* (2017), Han *et al.* (2018) and Fujii *et al.* (2017) involve the integral form of multiple BSDEs, which is redundant for option pricing. In addition, their BSDEs are not used as loss functions.

The paper is organized as follows. Section 2 defines the American option problems. Section 3 introduces the BSDE formation and the least squares residual loss function. Section 4 describes the architecture of the proposed neural network model. Section 5 discusses the techniques that improve the accuracy of the framework. Section 6 summarizes the algorithm. Section 7 analyzes the computational cost. In Section 8, we present numerical solutions of option prices and deltas to illustrate the advantage of our deep neural network framework. Section 9 concludes the paper.

† Here the proposed 'recursively-defined' feedforward network is not the same as the Recurrent Neural Network (RNN) in the literature, which will be explained in Section 4.1.

2. American options

In this paper, we use capital and lowercase letters to distinguish random and deterministic variables respectively. Suppose an American option contains a basket of d underlying assets. Let $\vec{S} = (S_1, \dots, S_d)^T \in \mathbb{R}^d$ be the prices of the underlying assets. Let $t \in [0, T]$ be the time up to the expiry T. Let r be the interest rate. Let δ_i and σ_i $(i = 1, \dots, d)$ be the dividend and volatility of each underlying asset. Let $\rho \in \mathbb{R}^{d\times d}$ be a correlation matrix. Define d correlated random variables $\mathrm{d}W_i(t) = \sum_{j=1}^d L_{ij}\phi_j(t)\sqrt{\mathrm{d}t}$, where $\phi_i(t) \sim \mathcal{N}(0,1)$ are independent standard normal random variables, and L is the Cholesky factorization of the correlation matrix, i.e. $\rho = LL^T$. Given an initial state $\vec{s}^0 \in \mathbb{R}^d$, the prices of the underlying assets $\vec{S}$ evolve under the following stochastic differential equations (SDEs):

$$\mathrm{d}S_i(t) = (r - \delta_i)S_i(t)\,\mathrm{d}t + \sigma_i S_i(t)\,\mathrm{d}W_i(t), \quad i = 1, \dots, d, \\ \vec{S}(0) = \vec{s}^0. \tag{1}$$

Let $f(\vec{s})$ be the payoff function of the option at the state $\vec{s}$, which usually takes the form of

$$f(\vec{s}) = \max(g(\vec{s}), 0). \tag{2}$$

Let $c(\vec{s}, t)$ be the continuation price, i.e. the discounted option payoff provided that the option is not exercised at time t and state $\vec{s}$:

$$c(\vec{s}, t) = \max_{\tau\in[t,T]} \mathbb{E}\left[\mathrm{e}^{-r(\tau-t)} f(\vec{S}(\tau)) \mid \vec{S}(t) = \vec{s}\right], \tag{3}$$

where τ is the stopping time. Then the American option price $v(\vec{s}, t)$ is defined as

$$v(\vec{s}, t) = \max\left[c(\vec{s}, t), f(\vec{s})\right] \\ = \begin{cases} c(\vec{s}, t), & \text{if } c(\vec{s}, t) > f(\vec{s}), \\ & \quad \text{i.e. the option is continued at } (\vec{s}, t), \\ f(\vec{s}), & \text{if } c(\vec{s}, t) \le f(\vec{s}), \\ & \quad \text{i.e. the option is exercised at } (\vec{s}, t). \end{cases} \tag{4}$$

In practical application of hedging, we are also interested in the first derivative of the American option price,

$$\vec{\nabla} v(\vec{s}, t) \equiv \left(\frac{\partial v}{\partial s_1}(\vec{s}, t), \dots \frac{\partial v}{\partial s_d}(\vec{s}, t)\right)^T.$$

This is called the 'delta' of the American option. The objective of this paper is to solve for both the option price $v(\vec{s}, t)$ and the option delta $\vec{\nabla} v(\vec{s}, t)$ on the entire spacetime.

3. Backward stochastic differential equation (BSDE) formulation

3.1. *BSDE formulation*

Our approach is to first convert the American option problem into a backward stochastic differential equation (BSDE) using the following theorem:

THEOREM 3.1 (BSDE formulation) *Assume that an American option is not exercised at time* $[t, t + \mathrm{d}t]$. *Then the continuation price of an American option at time t satisfies the following BSDE*:

$$\mathrm{d}c(\vec{S}, t) = rc(\vec{S}, t)\,\mathrm{d}t + \sum_{i=1}^d \sigma_i S_i(t) \frac{\partial c}{\partial s_i}(\vec{S}, t)\,\mathrm{d}W_i(t), \tag{5}$$

where $\vec{S}$ *satisfies the SDE* (1), *and* r, σ_i *and* $\mathrm{d}W_i(t)$ *are the same as in* (1).

Proof We refer interested readers to the proof in El Karoui *et al.* (1997) and Leentvaar (2008), which uses Ito's lemma. ■

The significance of the BSDE formulation (5) is two-fold. One is that it correlates the price $c(\vec{s}, t)$ with the delta $\vec{\nabla} c(\vec{s}, t)$. If the price is solved correctly, then (5) simultaneously yields the correct delta. A simultaneously correct evaluation of the price and the delta is essential for performing a complete hedging process. The other significance is that the BSDE formulation allows a less expensive and more manageable neural network approach. In fact, other than the BSDE formulation, American option problems can also be formulated as a Hamilton-Jacobi-Bellman partial differential equation (PDE) based on the Black-Scholes theory. Sirignano and Spiliopoulos (2018) considers a neural network approach for solving the PDE, which involves computing Hessian tensors. Unfortunately, a Hessian tensor is an $O(Md^2)$ tensor, where M is the number of samples for a neural network. When d is high, a Hessian tensor can be expensive to compute and store. In addition, given a neural network, the automatic differentiation of a Hessian is nearly impossible to derive, which makes it difficult to implement using existing deep learning libraries. However, unlike the PDE formulation, the BSDE formulation (5) does not contain a Hessian, which avoids the computation and storage of Hessian tensors. Instead, it only requires computing price tensors of size $O(M)$ and delta tensors of size $O(Md)$. In addition, delta tensors can be easily evaluated by the built-in automatic differentiation of Tensorflow (Abadi *et al.* 2016), i.e. 'tf.gradients'.

In this paper, we use an Euler timestepping Monte Carlo method to simulate the SDEs (1) and the BSDE (5). Let $m = 1, \dots, M$ be the indices of simulation paths, $n = 0, \dots, N$ be the indices of discrete timesteps from 0 to T, $\Delta t = T/N$, $t^n = n\Delta t$ be the timesteps, and $(\Delta W_i)_m^n = \sum_{j=1}^d L_{ij}(\phi_j)_m^n\sqrt{\Delta t}$. We discretize (1) as

$$(S_i)_m^0 = s_i^0, \quad i = 1, \dots, d; \tag{6}$$

$$(S_i)_m^{n+1} = (1 + (r - \delta_i)\Delta t)(S_i)_m^n + \sigma_i (S_i)_m^n (\Delta W_i)_m^n, \\ n = 0, \dots, N-1,\ i = 1, \dots, d. \tag{7}$$

We also discretize (5) as

$$c(\vec{S}_m^{n+1}, t^{n+1}) \\ = (1 + r\Delta t)c(\vec{S}_m^n, t^n) \\ + \sum_{i=1}^d \sigma_i (S_i)_m^n \frac{\partial c}{\partial s_i}(\vec{S}_m^n, t^n)(\Delta W_i)_m^n, \quad n = N-1, \dots, 0. \tag{8}$$

Theorem 3.1 assumes that an American option is not exercised at time $[t, t + \mathrm{d}t]$. More generally, if we allow the option to be exercised at any time after t, then we can replace $c(\vec{S}_m^{n+1}, t^{n+1})$ on the left hand side of (8) by $v(\vec{S}_m^{n+1}, t^{n+1})$. In addition, we add the expiry condition $v(\vec{s}, T) = f(\vec{s})$ into the discretization. This yields a complete discretized system for the BSDE:

$$v(\vec{S}_m^N, t^N) = f(\vec{S}_m^N), \quad n = N. \tag{9}$$

$$\text{Solve}(1 + r\Delta t)c(\vec{S}_m^n, t^n) + \sum_{i=1}^{d} \sigma_i(S_i)_m^n \frac{\partial c}{\partial s_i}(\vec{S}_m^n, t^n)(\Delta W_i)_m^n,$$
$$= v(\vec{S}_m^{n+1}, t^{n+1}) \quad \text{for } c(\vec{S}_m^n, t^n), \tag{10}$$

and then compute $v(\vec{S}_m^n, t^n)$

$$= \max\left[c(\vec{S}_m^n, t^n), f(\vec{S}_m^n)\right], \quad n = N - 1, \ldots, 0. \tag{11}$$

To sketch the idea of solving the discretized BSDE, let (6)–(7) generate samples of underlying asset prices $\{\vec{S}_m^n\}$ for all n's and m's. Then one starts with $n = N$, computes the expiry condition (9), and then performs backward timestepping from $n = N - 1$ to $n = 0$ using (10)–(11) iteratively, which yields $\{v(\vec{S}_m^n, t^n)\}$ for all n's and m's. Eventually, at $n = 0$, noting that $\vec{S}_m^0 = \vec{s}^0$ by (6), we obtain the option price $v(\vec{s}^0, 0)$ and the option delta $\vec{\nabla}v(\vec{s}^0, 0)$.

3.2. *Least squares solution for the discretized BSDE*

Consider only the nth timestep t^n, and introduce a short notation for the corresponding price and delta functions as $v^n(\vec{s}) \equiv v(\vec{s}, t^n)$ and $\vec{\nabla}v^n(\vec{s}) \equiv \vec{\nabla}v(\vec{s}, t^n)$. Solving (10) requires finding a d-dimensional function $c^n(\vec{s})$ where both the function $c^n(\vec{s})$ itself and its derivative $\vec{\nabla}c^n(\vec{s})$ satisfy (10). This is challenging, especially when d is large.

In this paper, we consider finding an approximation of the continuous price function. We let the approximation satisfy (10) in a least squares sense. More specifically, define the residual of (10) as the difference between the left and right hand sides:

$$\mathcal{R}[c^n]_m \equiv (1 + r\Delta t)c^n(\vec{S}_m^n) + \sum_{i=1}^{d} \sigma_i(S_i)_m^n \frac{\partial c^n}{\partial s_i}(\vec{S}_m^n)(\Delta W_i)_m^n$$
$$- v^{n+1}(\vec{S}_m^{n+1}), \quad m = 1, \ldots, M. \tag{12}$$

Then our goal is to find an approximation y^n to the actual continuation function c^n that minimizes the least squares residual:

$$c^n \approx (y^n)^* \equiv \arg\min_{y^n} \left(\sum_{m=1}^{M} \mathcal{R}[y^n]_m^2\right). \tag{13}$$

4. Neural network formulation

Finding the optimal approximate function in the least squares sense (13) is non-trivial. One approach is to use a parameterized function to represent the approximate function y^n. Then the optimization problem in terms of function space is converted to the optimization problem in terms of parameter space, which is more manageable.

One well-known example of the parameterized approach is the Longstaff-Schwartz method (Longstaff and Schwartz 2001). More specifically, the continuation price is approximated by a χth degree polynomial. We note that unlike our approach, their objective is not to minimize the least squares residual of the BSDE (12), but to minimize the least squares difference between the discounted payoffs and the parameterized polynomials. In practical implementation of the Longstaff-Schwartz method, we let $\chi \ll d$, which means that the number of the polynomial basis is $\binom{d+\chi}{d} \approx (1/\chi!)d^\chi$. However, convergence of the Longstaff-Schwartz method to the exact American option prices requires the number of the basis tending to infinity, i.e. $\chi \to \infty$ (Longstaff and Schwartz 2001, Stentoft 2004), which results in a high computational cost. In addition, a pre-defined, static polynomial basis may not be the optimal choice for American options.

4.1. *Sequence of neural networks*

Our approach is to use neural networks to represent the approximate continuation price function y^n. A neural network is a nonlinear parameterization where the basis is dynamic, i.e. the optimal basis is learned during the training process (Goodfellow *et al.* 2016). The main advantage of neural network formulation is that the complexity does not grow exponentially with the dimension d.

The architecture of neural network determines the proximity between the global minimum of the loss function and the true underlying price function, the landscape of the loss function, and the level of difficulty for optimization algorithms to find the global minimum. These directly impact the accuracy of the approximate price function. There exist many neural network architectures, such as feedforward, convolutional, or recurrent networks. We refer interested readers to Goodfellow *et al.* (2016) for a review of these standard network architectures. However, these standard networks are not designed for solving American option problems.

In this paper, we propose a sequence of N networks $\{y^n(\vec{s}; \Omega^n) \mid n = N - 1, \ldots, 1, 0\}$, where Ω^n is the trainable parameter set of the nth network. Each individual network $y^n(\vec{s}; \Omega^n)$ approximates the price function at the nth timestep $c^n(\vec{s})$. The design of each individual network is motivated by the fact that the approximate function of the nth timestep, $y^n(\vec{s}; \Omega^n)$, should differ from $y^{n+1}(\vec{s}; \Omega^{n+1})$ by a function of magnitude $O(\Delta t)$. Mathematically, it means that

$$y^N(\vec{s}) = f(\vec{s}), \quad n = N; \tag{14}$$

$$y^n(\vec{s}; \Omega^n) = y^{n+1}(\vec{s}; \Omega^{n+1}) + \Delta t \cdot \mathcal{F}(\vec{s}; \Omega^n),$$
$$n = N - 1, \ldots, 0; \tag{15}$$

where $\mathcal{F}(\vec{s}; \Omega^n)$ is the difference between the approximate functions at the two adjacent timesteps, or the 'residual' that we aim to find. We note that the sequence of networks (15) is defined in a recursive sense. In addition, the sequence of networks is backward in time, i.e. the timestep n decreases from

$N-1$ to 0. Hence, in this paper, we use the 'previous', 'current' and 'next' timesteps to refer to the $(n+1)$th, nth and $(n-1)$th timesteps, respectively.

Regarding each residual network $\mathcal{F}(\vec{s};\Omega^n)$, we parameterize it by an L-layer feedforward network with batch normalizations. In the following part, we drop the timestep index n temporarily, and use superscript with square brackets for the layer index $l=0,\ldots,L$. Let the dimensions of the layers be $\{d^{[l]} \mid l=0,\ldots,L\}$. Let the input of the neural network be $\vec{x}^{[0]}=\vec{s}\in\mathbb{R}^{d^{[0]}}$, where the input dimension is $d^{[0]}=d$. Then we construct an L-layer feedforward neural network as follows:

- For the hidden layers, $l=1,\ldots,L$:

  $$\text{linear transformation: } \vec{z}^{[l]}=\mathbf{W}^{[l]}\cdot\vec{x}^{[l-1]}, \tag{16}$$

  batch normalization:

  $$\vec{h}^{[l]}=\mathrm{bnorm}(\vec{z}^{[l]};\vec{\beta}^{[l]},\vec{\gamma}^{[l]},\vec{\mu}^{[l]},\vec{\sigma}^{[l]}), \tag{17}$$

  $$\text{rectified linear unit activation: } \vec{x}^{[l]}=\max(\vec{h}^{[l]},0), \tag{18}$$

  where

  $$\mathrm{bnorm}(\vec{x};\vec{\beta},\vec{\gamma},\vec{\mu},\vec{\sigma})\equiv\vec{\gamma}\cdot\frac{\vec{x}-\vec{\mu}}{\vec{\sigma}}+\vec{\beta} \tag{19}$$

  is the batch normalization operator, $\vec{x}^{[l]},\vec{z}^{[l]},\vec{h}^{[l]}\in\mathbb{R}^{d^{[l]}}$ are hidden layer variables, $\mathbf{W}^{[l]}\in\mathbb{R}^{d^{[l]}\times d^{[l-1]}}$ are trainable weights, $\vec{\mu}^{[l]},\vec{\sigma}^{[l]}\in\mathbb{R}^{d^{[l]}}$ are moving averages of batch means and standard deviations, and $\vec{\gamma}^{[l]},\vec{\beta}^{[l]}\in\mathbb{R}^{d^{[l]}}$ are trainable scales and offsets. The operations in (17)–(19) are evaluated elementwise. For instance, (18) means $x_i^{[l]}=\max(h_i^{[l]},0)$ for all $i=1,\ldots,d^{[l]}$.
- For the output layer:

  $$\mathcal{F}(\vec{s};\Omega^n)=\vec{\omega}\cdot\vec{x}^{[L]}+b, \tag{20}$$

  where $\vec{\omega}\in\mathbb{R}^{d^{[L]}}$, $b\in\mathbb{R}$ are trainable weight and bias.

In addition, we propose adding a scaling parameter α^n to each neural network (15) and revise it as

$$y^n(\vec{s};\Omega^n)=\alpha^n\left[y^{n+1}(\vec{s};\Omega^{n+1})+\Delta t\cdot\mathcal{F}(\vec{s};\Omega^n)\right], \quad n=N-1,\ldots,0. \tag{21}$$

We let α^n be trainable, or equivalently, $\alpha^n\in\Omega^n$. α^n is initialized as 1 before training, and is close to 1 during and after training. Introducing the trainable parameter α^n expands the function space the neural network can represent. A neural network with a larger function space is less likely to underfit, and thus more likely to have an accurate training result (Goodfellow *et al.* 2016).

We remark that our proposed recursive architecture (21) is different from the other architectures in the literature, particularly Sirignano and Spiliopoulos (2018), where one single neural network is used to represent the spacetime price function. The justification of our choice of this recursive architecture is that it is critical to the accuracy of the resulting prices and deltas. We note that the true price functions $c^{n+1}(\vec{s})$ and $c^n(\vec{s})$ differ by a function of magnitude $O(\Delta t)$. In (21), if we let $y^{n+1}(\vec{s};\Omega^{n+1})\approx c^{n+1}(\vec{s})$ and $\alpha^n\approx 1$, then regardless of the value of $\mathcal{F}(\vec{s};\Omega^n)$, $y^n(\vec{s};\Omega^n)$ will only differ from the true price function $c^n(\vec{s})$ by a magnitude of $O(\Delta t)$. Hence, before training starts, $y^n(\vec{s};\Omega^n)$ is already a good approximation of $c^n(\vec{s})$. This makes it more likely for the training to find the optimal solution that (almost) equals $c^n(\vec{s})$.

4.2. Feature selection

Feature selection, i.e. choosing the correct input features based on domain knowledge, has a great impact on the accuracy of neural network models (Goodfellow *et al.* 2016). Naively one can simply set the input as the underlying asset prices $\vec{x}^{[0]}=\vec{s}$. In this paper, we consider adding two new features.

One new feature is the payoff function. It is suggested in Kohler (2010) and Firth (2005) that including the payoff in the nonlinear basis can improve the accuracy of the regression-based algorithms. In this paper, we consider using $g(\vec{s})$ in (2) as an input feature. The reason of using $g(\vec{s})$ rather than $f(\vec{s})$ is that the maximum operator in (2) is irreversible. In other words, $f(\vec{s})$ can be computed by $g(\vec{s})$ but not conversely. Hence, using $g(\vec{s})$ as the input contains more information than $f(\vec{s})$. The additional maximum operator in (2) can be learned by the activation function (18) in the network.

The other new feature we consider adding is the output price function from the previous timestep, i.e. $y^{n+1}(\vec{s};\Omega^{n+1})$ in (21). The intuition is that the solution at the nth step should look similar to the solution at the $(n+1)$th step. We note that this feature is similar but not exactly the same as the payoff function, which makes it useful as an additional feature. More specifically, when $n\approx N$, y^{n+1} is approximately the same as but slightly smoother than the payoff function; when $n\ll N$, y^{n+1} can be very different from the payoff function.

The accuracy of neural network models can be further improved by input normalization (Sola and Sevilla 1997). Effectively, we can combine the implementation of feature selection and input normalization by adding the following 'input layer' (denoted as $l=0$) before the hidden layer $l=1$:

$$\text{feature concatenation: } \vec{z}^{[0]}=\left(\vec{s},g(\vec{s}),y^{n+1}(\vec{s};\Omega^{n+1})\right)^T\in\mathbb{R}^{d^{[0]}}, \tag{22}$$

$$\text{input normalization: } \vec{x}^{[0]}=\mathrm{bnorm}(\vec{z}^{[0]};\vec{\beta}^{[0]},\vec{\gamma}^{[0]},\vec{\mu}^{[0]},\vec{\sigma}^{[0]}), \tag{23}$$

where the input dimension is changed to $d^{[0]}=d+2$ after the concatenation. We note that $\vec{\mu}^{[0]}$ and $\vec{\sigma}^{[0]}$ can be pre-computed from the entire training dataset, unlike $\vec{\mu}^{[l]}$ and $\vec{\sigma}^{[l]}$ in the hidden layers that are computed by moving averages of training batches.

To summarize Sections 4.1–4.2, the architecture of the proposed neural network framework is defined by (14) and (21), where the residual network at each timestep $\mathcal{F}(\vec{s};\Omega^n)$ is defined by the input layer (22)–(23), the hidden layers (16)–(18) and the output layer (20). The trainable parameters of the

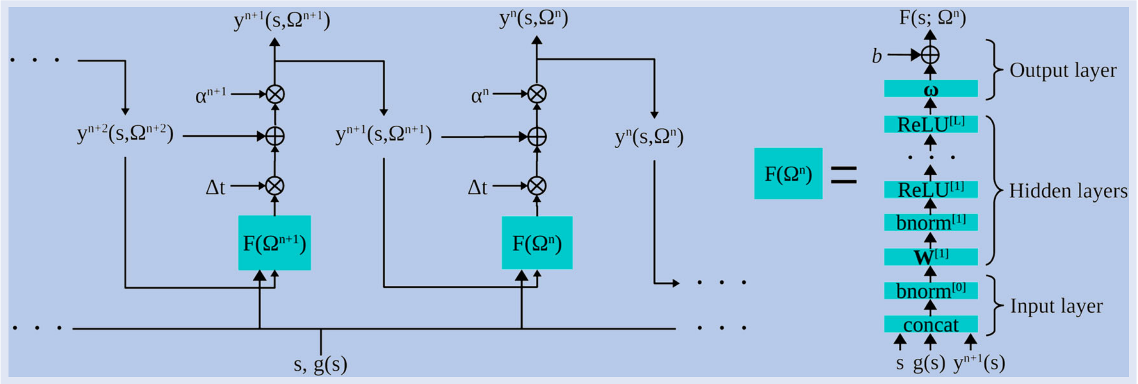


Figure 1. The architecture of the proposed neural network framework defined by (14) and (21), where the residual network at each timestep $\mathcal{F}(\vec{s};\Omega^n)$ is defined by the input layer (22)–(23), the hidden layers (16)–(18) and the output layer (20). The symbols $\otimes$ and $\oplus$ represent multiplication and addition, respectively.

neural network framework are $\{\Omega^n \mid n = N-1,\ldots,0\}$, where

$$\begin{aligned}\Omega^n \equiv \{&(\mathbf{W}^{[l]})^n, (\vec{\gamma}^{[l]})^n, (\vec{\beta}^{[l]})^n, (\vec{\gamma}^{[0]})^n,\\ &(\vec{\beta}^{[0]})^n, \vec{\omega}^n, b^n, \alpha^n \mid L = 1,\ldots,L\}.\end{aligned} \tag{24}$$

Figure 1 illustrates the architecture of the proposed neural network framework.

After the completion of this paper, we realized that, after introducing y^{n+1} as an additional feature, our architecture becomes a *novel variation* of the Residual Neural Networks (ResNet) proposed in He *et al.* (2016). The commonality between our architecture and ResNet is the pattern of the 'shortcut connections'. Mathematically, it means that, for each building block (in our case, the network at each timestep), the output (in our case, y^n) is the sum of the input feature (in our case, y^{n+1}) and its propagation through the 'residual' neural network (in our case, $\mathcal{F}(\vec{s}, g(\vec{s}), y^{n+1}(\vec{s}))$). ResNet is well-known for its high accuracy. As explained in He *et al.* (2016), this is because the 'shortcut connections' make it easier for the input information to propagate through the deep neural network, and, compared with optimizing the full mapping, it is easier to optimize the 'residual' mapping as it is close to zero. This provides another insight on how our network architecture achieves accuracy. As a side remark, despite the similarity, our architecture is more sophisticated than ResNet due to the input feature $(\vec{s}, g(\vec{s}))$ in each residual network $\mathcal{F}$ and the scaling parameter α^n.

4.3. *More efficient neural network sequence*

Sections 4.1–4.2 have explained that the main advantage of our recursive architecture is the accuracy. However, the recursive architecture (21) is expensive when N is large. More specifically, consider the 0th timestep, and consider using the sequence of the neural networks to compute the value of $y^0(\vec{s})$. By applying the recursive relation (21), we have

$$y^0(\vec{s}) = y^N(\vec{s}) + \Delta t \cdot \sum_{\nu=1}^{N} \mathcal{F}(\vec{s}; \Omega^{N-\nu}), \tag{25}$$

where for simplicity we set $\alpha^n = 1$ for all timesteps. Equation (25) shows that the computation of $y^0(\vec{s})$ requires going through N feedforward networks.

Here we propose a modified neural network architecture to reduce the computational cost. In Section 4.1, we motivated the recursive relation (21) based on the fact that the outputs of the two adjacent timesteps, $y^n(\vec{s})$ and $y^{n+1}(\vec{s})$, should differ by a function of magnitude $O(\Delta t)$. In fact, we can generalize this relation to any two timesteps n and $n+j$ where $j \ll N$. That is, the outputs $y^n(\vec{s})$ and $y^{n+j}(\vec{s})$ should differ by a function of magnitude $O(\Delta t)$. Similar to (21), we formulate this idea into the following recursive relation:

$$y^n(\vec{s};\Omega^n) = \alpha^n \left[y^{n+j}(\vec{s};\Omega^{n+j}) + j\Delta t \cdot \mathcal{F}(\vec{s};\Omega^n) \right]. \tag{26}$$

This generalization allows us to recur the feedforward networks at every few timesteps, rather than at every single timestep, and thus reduces the computational cost.

To be more precise, if we recur the feedforward networks at every J timesteps ($J \ll N$), then we modify the sequence of the neural networks (21) as follows:

$$\begin{aligned} &y^n(\vec{s};\Omega^n) = \alpha^n \left[y^{n+\eta}(\vec{s};\Omega^{n+\eta}) + \eta\Delta t \cdot \mathcal{F}(\vec{s};\Omega^n) \right],\\ &\text{where } \eta \equiv [(N-n-1) \bmod J] + 1, \quad n = N-1,\ldots,0. \end{aligned} \tag{27}$$

Equivalently, we can enumerate (27) as

$$\begin{aligned}
\text{at the } (n-1)\text{th step:}\quad & y^{n-1}(\vec{s};\Omega^{n-1})\\
& = \alpha^{n-1}\left[y^n(\vec{s};\Omega^n)\right.\\
& \quad \left. + \Delta t \cdot \mathcal{F}(\vec{s};\Omega^{n-1})\right],\\
& \vdots\\
\text{at the } (n-j)\text{th step:}\quad & y^{n-j}(\vec{s};\Omega^{n-j})\\
& = \alpha^{n-j}\left[y^n(\vec{s};\Omega^n)\right.\\
& \quad \left. + j\Delta t \cdot \mathcal{F}(\vec{s};\Omega^{n-j})\right],\\
& \vdots\\
\text{at the } (n-J)\text{th step:}\quad & y^{n-J}(\vec{s};\Omega^{n-J})\\
& = \alpha^{n-J}\left[y^n(\vec{s};\Omega^n)\right.\\
& \quad \left. + J\Delta t \cdot \mathcal{F}(\vec{s};\Omega^{n-J})\right],
\end{aligned} \tag{28}$$

where $1 \le j \le J$ and $n = N, N-J, N-2J, \ldots$. We remark that (21) is simply a special case of (27) with $J = 1$. Figure 2 illustrates the modified architecture with $J = 3$. Readers can generalize the idea of figure 2 to any $J \ll N$.

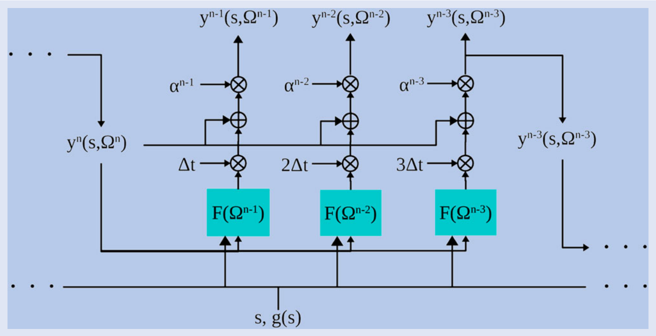


Figure 2. The modified architecture of the proposed neural network framework defined by (14) and (27), where $J = 3$. Similar to figure 1, the residual network at each timestep $\mathcal{F}(\vec{s};\Omega^n)$ is defined by the input layer (22)–(23), the hidden layers (16)–(18) and the output layer (20).

Regarding the choice of J, smaller J yields more precise trained y^n with higher computational cost; larger J is computationally cheaper but the trained y^n is less precise. In our numerical simulations, we choose $N = 100$ and $J = 4$.

To give an example of how the modified architecture reduces the computational cost, let us reconsider evaluating $y^0(\vec{s})$. By applying the recursive relation (27), we have

$$y^0(\vec{s}) = y^N(\vec{s}) + J\Delta t \cdot \sum_{\nu=1}^{\lfloor N/J \rfloor} \mathcal{F}(\vec{s};\Omega^{N-\nu J}) + (N \bmod J)\Delta t \cdot \mathcal{F}(\vec{s};\Omega^0), \tag{29}$$

where for simplicity we set $\alpha^n = 1$ for all timesteps. Compared to (25), using (29) to compute $y^0(\vec{s})$ only requires going through $\lceil N/J \rceil$ feedforward networks. In other words, the computation is J times cheaper.

4.4. *Computation of the derivatives*

Using the BSDE formulation (12)–(13) requires the computation of the derivatives $\vec{\nabla} y^n(\vec{s})$. This can be evaluated by the built-in automatic differentiation of Tensorflow (Abadi *et al.* 2016), i.e. 'tf.gradients'.

4.5. *Training the neural network*

Consider training the network at the nth timestep for solving (12)–(13). The training inputs are

$$\{\vec{S}^n_m, \Delta\vec{W}^n_m, v^{n+1}(\vec{S}^{n+1}_m), g(\vec{S}^n_m), y^{n+\eta}(\vec{S}^n_m;(\Omega^{n+\eta})^*), \vec{\nabla} y^{n+\eta}(\vec{S}^n_m;(\Omega^{n+\eta})^*) \mid \forall\, m\}, \tag{30}$$

where the first three inputs are the required inputs of (12), the last three inputs are the features introduced in Section 4.2, $y^{n+\eta}$ is defined in (27) and $(\Omega^{n+\eta})^*$ is the trained parameters from the previous timestep $n+\eta$. The training output is $\{y^n(\vec{S}^n_m;\ \Omega^n), \vec{\nabla} y^n(\vec{S}^n_m;\ \Omega^n) \mid \forall\, m\}$. The loss function of the network is given by (12)–(13), i.e. the least squares BSDE residual, which we rewrite as a function of the trainable parameters Ω^n:

$$\mathcal{L}[\Omega^n] \equiv \sum_{m=1}^{M} \Bigg[(1 + r\Delta t) y^n(\vec{S}^n_m;\Omega^n) + \sum_{i=1}^{d} \sigma_i(S_i)^n_m \frac{\partial y^n}{\partial s_i}(\vec{S}^n_m;\Omega^n)(\Delta W_i)^n_m - v^{n+1}(\vec{S}^{n+1}_m) \Bigg]^2. \tag{31}$$

We consider using the popular Adam optimizer (Kingma and Ba 2014) to minimize the loss function (31), which yields the set of optimal trainable parameters

$$(\Omega^n)^* \equiv \arg\min_{\Omega^n} \mathcal{L}[\Omega^n]. \tag{32}$$

Then, using the trained neural network, we can compute the estimated option price $y^n(\vec{s};(\Omega^n)^*)$ and delta $\vec{\nabla} y^n(\vec{s};(\Omega^n)^*)$. In addition, we use the estimated option price to determine the exercise boundary as

$$\xi^n(\vec{s}) = \begin{cases} \text{continued}, & \text{if } y^n(\vec{s};(\Omega^n)^*) > f(\vec{s}), \\ \text{exercised}, & \text{otherwise.} \end{cases} \tag{33}$$

In order to ensure the accuracy of training, we follow suggested good practices in the deep learning community (Goodfellow *et al.* 2016). For instance, mini-batch optimization is used; the learning rate of the Adam optimizer is decayed to ensure convergence; gradient clipping is applied to avoid exploding gradients. In particular, we let the number of training steps be 600. At the sth training step $(0 \le s \le 600)$, we let the moving average rate for $\vec{\mu}^{[l]}$ and $\vec{\sigma}^{[l]}$ in (17) be $\frac{1}{0.99}(0.01^{\max(\min(s/350,1),0)} - 0.01)$, and let the learning rate for the Adam optimizer be $0.01 \times 0.001^{\max(\min((s-150)/350,1),0)}$.

5. Improving the accuracy of the formulation

Sections 3–4 describe the foundation of our deep neural network formulation. In general, achieving precision is a major challenge for developing deep neural networks. Sections 4.1–4.2 have explained that our neural network architecture, i.e. using the sequence of recursive networks to provide a good initial guess and introducing $g(\vec{s})$ and $y^{n+1}(\vec{s};\Omega^{n+1})$ as additional features, is critical to the accuracy of the resulting prices and deltas. In this section, we propose multiple techniques that further improve the accuracy, including smoothing payoff functions, defining training input v^{n+1} in (30) carefully, weight reuse, network ensemble, and special formula for the price and delta at $t = 0$.

5.1. *Smoothing payoff functions*

We note that most of the payoff functions in practical applications have the form of (2), which is not differentiable at $g(\vec{s}) = 0$. In other words, $y^N(\vec{s})$ in (14) is not differentiable. However, $y^{N-1}(\vec{s};\Omega^{N-1})$ as an approximation of the continuation price function is differentiable. Consequentially, the left and right hand sides of (21) at $n = N-1$, i.e. $y^{N-1}(\vec{s};\Omega^{N-1}) =$

$\alpha^{N-1}[y^N(\vec{s}) + \Delta t \cdot \mathcal{F}(\vec{s}; \Omega^{N-1})]$, are inconsistent in terms of differentiability. Such inconsistency makes it difficult to learn an accurate $\mathcal{F}(\vec{s}; \Omega^{N-1})$, which negatively affects the accuracy of the trained $y^{N-1}(\vec{s}; \Omega^{N-1})$, and furthermore, the accuracy of the trained $y^n(\vec{s}; \Omega^n)$ in the subsequent timesteps. In this paper, we propose smoothing the function $y^N(\vec{s})$ in (14) as follows:

$$y^N(\vec{s}) = f_\kappa(\vec{s}) \equiv \frac{1}{\kappa} \ln\left(1 + \mathrm{e}^{\kappa g(\vec{s})}\right), \tag{34}$$

where κ is a user-defined parameter. The operations in (34) are evaluated element-wise. $f_\kappa(\vec{s})$ converges to $f(\vec{s})$ when $\kappa \to \infty$, and is a good approximation of $f(\vec{s})$ when κ is large. The significance of (34) is that $f_\kappa(\vec{s})$ is differentiable, which makes it easier to train an accurate $\mathcal{F}(\vec{s}; \Omega^{N-1})$. In practice, we choose $\kappa = 2/\Delta t$. We note that smoothing payoff functions is a standard technique in the literature of binomial trees for option pricing (Heston and Zhou 2000). However, to the best of our knowledge, this paper is the first to propose smoothing payoff functions among the literature of neural networks for option pricing.

5.2. *The training input 'v'*

Consider the nth timestep. The definition of $v^{n+1}(\vec{S}_m^{n+1})$ in the training input (30) turns out to play a significant role in the accuracy of the trained continuation price y^n. More specifically, if the training input $v^{n+1}(\vec{S}_m^{n+1})$ is incorrectly defined, which means that we feed incorrect values to the right hand side of (10), then the trained network y^n would not represent the correct c^n.

Finding the correct definition of $v^{n+1}(\vec{S}_m^{n+1})$ turns out to be non-trivial. One natural way of defining $v^{n+1}(\vec{S}_m^{n+1})$ is to use the output prices of the trained network. More specifically, suppose $y^{n+1}(\vec{s}; (\Omega^{n+1})^*)$ is already trained. Then

$$v^{n+1}(\vec{S}_m^{n+1}) = \begin{cases} y^{n+1}(\vec{S}_m^{n+1}; (\Omega^{n+1})^*), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{continued}, \\ f(\vec{S}_m^{n+1}), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{exercised}, \end{cases} \tag{35}$$

where ξ^{n+1} is defined in (33). However, in practice, due to the finite number of samples and training steps, training error in the network y^{n+1} is inevitable, which means that $v^{n+1}(\vec{S}_m^{n+1})$ might contain error after applying (35). Consequentially, the error of the training input $v^{n+1}(\vec{S}_m^{n+1})$ will propagate into y^n after training the nth network, and propagate into $v^n(\vec{S}_m^n)$ after applying (35) again, and propagate into y^{n-1}, $v^{n-1}(\vec{S}_m^{n-1})$, $y^{n-2}, \ldots$, after further backward timestepping. In other words, (35) is not robust against the accumulation of training errors over timesteps and may result in bias.

In fact, such bias can be quantified. Assume that $\{\vec{S}_m^\nu \mid 0 \le \nu \le n, \forall\, m\}$ are given/fixed. Let $\{\vec{S}_m^{n+1} \mid \forall\, m\}$ be another set generated under (7). Consider taking the conditional expectation of (10):

$$(1 + r\Delta t)c^n(\vec{S}_m^n) + \sum_{i=1}^{d} \sigma_i(S_i)_m^n \frac{\partial c^n}{\partial s_i}(\vec{S}_m^n)\mathbb{E}[(\Delta W_i)_m^n \mid \vec{S}_m^n]$$
$$= \mathbb{E}[v^{n+1}(\vec{S}_m^{n+1}) \mid \vec{S}_m^n].$$

We note that under the assumption, $\{\vec{S}_m^n\}$, $\{c^n(\vec{S}_m^n)\}$ and $\{(\partial c^n/\partial s_i)(\vec{S}_m^n)\}$ in this equation are not random variables, and the only random variables are $\{\vec{S}_m^{n+1}\}$ and $\{(\Delta W_i)_m^n\}$. Since $\mathbb{E}[(\Delta W_i)_m^n \mid \vec{S}_m^n] = 0$ and $1 + r\Delta t \approx \mathrm{e}^{r\Delta t}$, we have

$$c^n(\vec{S}_m^n) = \mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1}) \mid \vec{S}_m^n]. \tag{36}$$

Equation (36) indicates that if $v^{n+1}(\vec{S}_m^{n+1})$ is correctly evaluated, then $\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})]$ should match the true underlying continuation function $c^n(\vec{S}_m^n)$. After a few timesteps, if $\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})]$ deviates from $c^n(\vec{S}_m^n)$, then it indicates an accumulation of training errors from the previous timesteps.

Figure 3 shows a concrete example of the bias. Consider a simulation of a one-dimensional American option, where $T = 0.5$, $N = 100$ and the true continuation function c^n can be computed by finite difference methods. Consider using (35) to define the training input $v^{n+1}(\vec{S}_m^{n+1})$ at every timestep. As shown in the top left plot of figure 3, when the simulation proceeds to $n = 33$, there is a clear deviation of $\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})]$ (blue line)† from $c^n(\vec{S}_m^n)$ (black line) around $\vec{S}_m^n = 80$.

In fact, we can use the relation (36) to avoid the bias caused by the definition (35). More specifically, let $\vec{S}_m^{n+1}$ be a continued point. Then $v^{n+1}(\vec{S}_m^{n+1}) = c^{n+1}(\vec{S}_m^{n+1}) = \mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+2}(\vec{S}_m^{n+2})]$. This motivates us to redefine the training input $v^{n+1}(\vec{S}_m^{n+1})$ as follows:

$$v^{n+1}(\vec{S}_m^{n+1}) = \begin{cases} \mathrm{e}^{-r\Delta t}\, v^{n+2}(\vec{S}_m^{n+2}), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{continued}, \\ f(\vec{S}_m^{n+1}), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{exercised}. \end{cases} \tag{37}$$

We note that (37) is actually the 'discounted payoffs' used in Longstaff and Schwartz (2001). They use (37) as the target prices for regression.

The top right plot of figure 3 considers again the same simulation, where the definition of $v^{n+1}(\vec{S}_m^{n+1})$ is changed to (37). The deviation of $\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})]$ (blue line) from $c^n(\vec{S}_m^n)$ (black line) around $\vec{S}_m^n = 80$ disappears. The blue and black lines agree well with each other. This shows that using the definition (37) does not introduce bias as does the definition (35). However, the noisy red dots show that using the definition (37) results in a big variance of $\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})$. This poses a risk for the model to fit the noise, which may still result in an inaccurate trained y^n.

† To assess $\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})]$, we start with a fixed set of $\{\vec{S}_m^n\}$. For each point of $\vec{S}_m^n$, we generate multiple $\vec{S}_m^{n+1}$'s by (7), denoted as $\{\vec{S}_{m;m'}^{n+1} | m' = 1, \ldots, M'\}$; compute $\{v(\vec{S}_{m;m'}^{n+1})\}$; and then compute the imperial average:

$$\mathbb{E}[\mathrm{e}^{-r\Delta t} v^{n+1}(\vec{S}_m^{n+1})] \approx \mathrm{e}^{-r\Delta t} \frac{1}{M'} \sum_{m'} v(\vec{S}_{m;m'}^{n+1}).$$

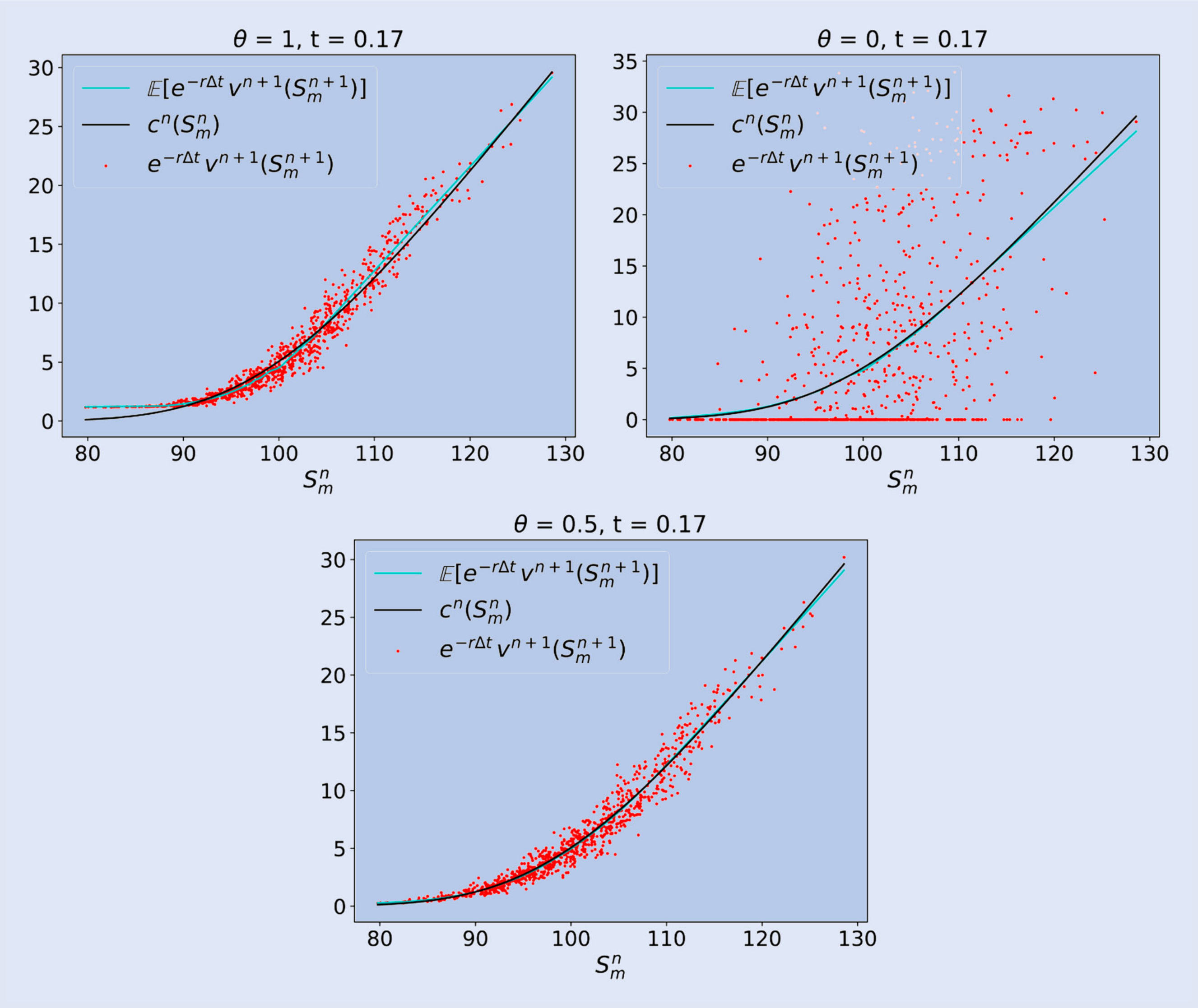


Figure 3. The values of $c^n(\vec{S}_m^n)$ (black line), $\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})$ (red dots) and $\mathbb{E}[\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})]$ (blue line) under different definitions of $v^{n+1}(\vec{S}_m^{n+1})$. (Top left) The values under the definition of (35), which shows a bias, i.e. deviation of $\mathbb{E}[\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})]$ from $c^n(\vec{S}_m^n)$, especially near $\vec{S}_m^n = 80$; (Top right) The values under the definition of (37), which shows a variance, i.e. large noise of $\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})$; (Bottom) The values under the definition of (38) with $\theta = 0.5$, where both bias and variance are reduced.

In this paper, we define $v^{n+1}(\vec{S}_m^{n+1})$ as the linear combination of the two definitions (35) and (37):

$$v^{n+1}(\vec{S}_m^{n+1}) = \begin{cases} \theta\, y^{n+1}(\vec{S}_m^{n+1}; (\Omega^{n+1})^*) \\ \quad +(1-\theta)\,\mathrm{e}^{-r\Delta t}v^{n+2}(\vec{S}_m^{n+2}), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{continued}, \\ f(\vec{S}_m^{n+1}), & \text{if } \xi^{n+1}(\vec{S}_m^{n+1}) \\ & \quad = \text{exercised}, \end{cases} \tag{38}$$

where $\theta \in [0, 1]$ is a user-defined hyperparameter. This linear combination mitigates both the bias caused by the definition (35) and the variance caused by the definition (37). That is, the resulting $v^{n+1}(\vec{S}_m^{n+1})$ would accumulate less training error over multiple timesteps, and meanwhile contain less noise. The bottom plot in figure 3 considers the same simulation, where the definition of $v^{n+1}(\vec{S}_m^{n+1})$ is (38) with $\theta = 0.5$. We observe almost no deviation of $\mathbb{E}[\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})]$ (blue line) from $c^n(\vec{S}_m^n)$ (black line), and a small variance of $\mathrm{e}^{-r\Delta t}v^{n+1}(\vec{S}_m^{n+1})$ (red dots), as expected. Hence, the definition (38) can improve the accuracy of the trained networks.

5.3. *Weight reuse*

The trainable parameters Ω^n need to be initialized for each individual network from $n = N-1$ to $n = 0$. Starting from the network at $n = N-1$, we initialize $(\vec{\beta}^{[l]})^{N-1}$ and b^{N-1} by zeros; $(\vec{\gamma}^{[l]})^{N-1}$ and α^{N-1} by ones; and $(\mathbf{W}^{[l]})^{N-1}$ and $\vec{\omega}^{N-1}$ by uniformly distributed random numbers in $(-1/\sqrt{d^{[l]}+d^{[l-1]}}, 1/\sqrt{d^{[l]}+d^{[l-1]}})$, as suggested in Goodfellow *et al.* (2016). Move on to the consecutive networks at $n < N-1$. One can use the same idea to initialize their trainable parameters. However, we notice that when Δt is sufficiently small, the networks at the nth and $(n+1)$th

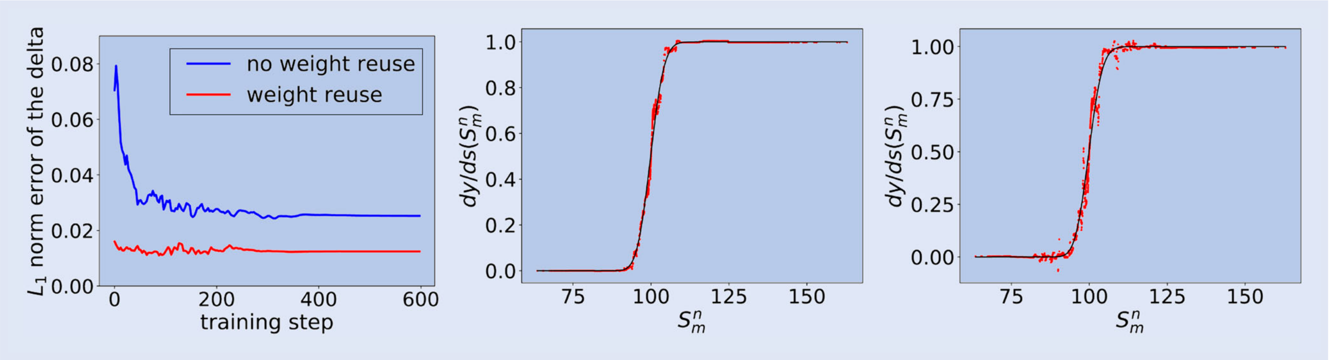


Figure 4. Example of the computed deltas with or without weight reuse. (Left) The L_1 norm error of the computed delta over 600 training steps. Blue: the error with no weight reuse. Red: the error with weight reuse. (Middle) The computed delta with weight reuse after 600 training steps. Black line: the exact delta computed by finite difference. Red dots: the sample values of the delta obtained from the network y^n. (Right) The computed delta without weigh reuse after 600 training steps.

timesteps should be close. In other words, their optimal trainable parameters should be close, i.e. $(\Omega^{n+1})^* \approx (\Omega^n)^*$. We can take advantage of this fact and use the values of the trained parameters $(\Omega^{n+1})^*$ as the initial values of the corresponding trainable parameters Ω^n. Such 'weight reuse' provides a good initial guess before the training starts at the nth timestep. Hence, the training results will be more accurate.

Figure 4 demonstrates a concrete example on how weight reuse improves the training accuracy. Consider again a simulation of a one-dimensional American option with $T = 0.5$, $N = 50$. Consider a particular timestep $n = 47$. We computed the delta $(\mathrm{d}y^n/\mathrm{d}s)(S_m^n)$ of 180000 sample points. The first plot shows the evolution of the L_1 norm error of the computed delta over 600 training steps. The error with weight reuse (red line) is significantly lower than the error without weight reuse (blue line). The second plot shows that after 600 training steps, the computed delta with weight reuse (red dots) agrees with the exact delta (black line). As a comparison, the third plot shows that after 600 training steps, the computed delta without weight reuse (red dots) still has a large fluctuation and does not match the exact delta (black line) well.

5.4. *Ensemble of neural networks*

It is well-known that ensemble learning, which is a combination of the multiple machine learning models, usually outperforms individual models (Goodfellow *et al.* 2016). Inspired by this, we consider 'ensemble of neural networks'.

To describe the details, at each timestep (e.g. the nth timestep), we construct C networks $\{y^n(\vec{s};\, \Omega_c^n) \,|\, c = 1, \ldots, C\}$ instead of one network. All the C networks have the same architecture as defined in Sections 4.1–4.3. The difference is that their trainable parameters $\{\Omega_c^n \,|\, c = 1, \ldots, C\}$ are initialized by different set of numbers. Then the C networks are trained by different input data. To do this, we generate CM input samples (30) with $m = 1, \ldots, CM$, split them into C copies, and then use each copy of the input samples to train each of the C networks. Consequentially, the trained results of the C networks are independent of each other, i.e. $\{(\Omega_c^n)^* \,|\, c = 1, \ldots, C\}$ are distinct from each other. Then after training, we compute the averages across the ensemble:

$$y^n(\vec{s}) = \frac{1}{C}\sum_{c=1}^{C} y^n(\vec{s};\, (\Omega_c^n)^*),$$
$$\vec{\nabla} y^n(\vec{s}) = \frac{1}{C}\sum_{c=1}^{C} \vec{\nabla} y^n(\vec{s};\, (\Omega_c^n)^*), \tag{39}$$

for the prices and deltas, respectively. Eventually, we use the ensemble-average prices to determine the exercise boundary at the nth timestep by (33) before proceeding to the $(n-1)$th timestep.

Such ensemble technique yields more precise prices, deltas and thus more precise exercise boundaries. We note that the computation across different ensembles can be parallelized. In practice, we find that $C = 3$ is a good choice, in the sense that the accuracy is improved compared with $C = 1$ without dramatically increasing computational cost.

5.5. *Price and delta at t = 0*

Our neural network formulation yields prices and deltas for the entire spacetime domain. In practical applications, the price and the delta at $t = 0$, $v(\vec{s}^0, 0)$ and $\vec{\nabla} v(\vec{s}^0, 0)$, are of particular interest. We can extract their values from the trained neural network at $t = 0$. Here we discuss how to further improve the accuracy of their values.

Our approach is to use the expectation values of the Monte Carlo paths, subject to the exercise boundary computed by our neural network formulation. More specifically, given the mth path, the trained neural networks determine its stopping time, denoted as τ_m. Then the price at $t = 0$ can be computed by the mean of the discounted payoffs:

$$v(\vec{s}^0, 0) = \frac{1}{CM}\sum_{m=1}^{CM} \mathrm{e}^{-r\tau_m} f(\vec{S}_m(\tau_m)). \tag{40}$$

Regarding the delta at $t = 0$, we can use the method in Thom (2009), which is an adaptation of 'pathwise derivative method' (Broadie and Glasserman 1996) to American options:

$$\frac{\partial v}{\partial s_i}(\vec{s}^0,0)=\frac{1}{CM}\sum_{m=1}^{CM}\left(\mathrm{e}^{-r\tau_m}\sum_{j=1}^{d}\frac{\partial f}{\partial s_j}(\vec{S}_m(\tau_m))\frac{\partial (S_j)_m}{\partial (s_i)^0}\right). \tag{41}$$

When the underlying asset prices evolve under (1), we have

$$\frac{\partial (S_j)_m}{\partial (s_i)^0}=\frac{(S_j)_m}{(s_i)^0}\delta_{ij}.$$

We note that the pathwise derivative approach may not be applicable if $\partial(S_j)_m/\partial(s_i)^0$ is not evaluable (e.g. the underlying asset prices do not evolve under (1)) or if the payoff function is not differentiable. For such non-applicable cases, we can still obtain the deltas from our trained neural network at $t=0$.

Using (40)–(41) to compute the price and the delta at $t=0$ is also observed in other Monte Carlo style pricing approaches, including the Longstaff-Schwartz algorithm. However, we emphasize that our approach differs from the others. More specifically, (40)–(41) are not computable unless combined with an algorithm that can determine the exercise boundary on the entire spacetime. In this paper, our neural network framework is used to determine the exercise boundary before applying (40)–(41). In Section 8, we will demonstrate that our neural network formulation yields a more accurate exercise boundary, and thus more accurate prices and deltas at $t=0$, compared to the Longstaff-Schwartz algorithm.

6. Final algorithm

The final version of the proposed algorithm is summarized in Algorithm 1. We note that in Algorithm 1, we store $\{y^n(\vec{S}^n_m), \vec{\nabla}y^n(\vec{S}^n_m)\,|\,\forall\, n, \forall\, m\}$ on the entire spacetime (i.e. for all m's and n's). The reason is that we are interested in a complete delta hedging simulation, which requires sample values of both prices and deltas for the entire spacetime. The implementation of Algorithm 1 uses an overwriting strategy for more efficient memory. We note, however, that if an algorithm user does not need sample values from the entire spacetime, then only the storage of the training outputs $\{y^n(\vec{S}^n_m), \vec{\nabla}y^n(\vec{S}^n_m)\,|\,\forall\, m\}$ and the training inputs $\{y^{n+\eta}(\vec{S}^n_m), \vec{\nabla}y^{n+\eta}(\vec{S}^n_m)\,|\,\forall\, m\}$ at the current timestep (i.e. for all m's and for a given n) is necessary.

7. Computational cost

In this section, we analyze the computational cost of the proposed algorithm, and make a comparison with the Longstaff-Schwartz algorithm. For the Longstaff-Schwartz algorithm, consider degree-χ monomial basis

$$\varphi_\chi(\vec{s}) \equiv \{s_1^{a_1}s_2^{a_2}\cdots s_d^{a_d}\,|\,a_1+a_2+\cdots+a_d\le\chi\}, \tag{42}$$

as proposed in Longstaff and Schwartz (2001) and Kohler (2010). In practice, we choose $\chi \ll d$. Then the number of the monomial basis is $\binom{d+\chi}{d}\approx(1/\chi!)d^\chi$.

Algorithm 1 Neural network pricing and hedging under BSDE formulation

```
Parameters
    $C$: the number of networks in network ensemble
    $M$: the number of samples per ensemble
    $N$: the number of timesteps
    $J$: the number of timesteps between the network recurrence

Initialize the underlying asset prices $\{\vec{S}^0_m \equiv \vec{s}^0\,|\,\forall\, m(\text{i.e.}m=1,\ldots,CM)\}$.
for $n=1,\ldots,N$ do
  Use (6)–(7) to generate $CM$ Monte Carlo trajectories of the underlying asset prices $\{\vec{S}^n_m\,|\,\forall\, m\}$.
end for

Use (34) to compute the expiry option prices and option deltas
    $\{Y^\nu_m = y^N(\vec{S}^\nu_m)\,|\,0\le\nu\le N, \forall\ \ m\}$ and $\{\vec{Z}^\nu_m = \vec{\nabla}y^N(\vec{S}^\nu_m)\,|\,0\le\nu\le N, \forall\, m\}$.
Initialize $\{v^N(\vec{S}^N_m)\,|\,\forall\, m\}$ by (9).

for $n=N-1,\ldots,0$ do
  for $c=1,\ldots,C$ do
    Initialize the neural network $y^n(\vec{s};\,\Omega^n_c)$ defined by (27), where the input layer is (22)–(23), the hidden layers are (16)–(18) and the output layer is (20).
    Training: minimize the least squares residual (31)–(32), using the training input (30).
    Result: the trained neural network $y^n(\vec{s};\,(\Omega^n_c)^*)$.
  end for

  if $(N-n) \bmod J = 0$ then
    Ensemble evaluation (all future timesteps): overwrite the option prices and deltas
      $\{Y^\nu_m = \frac{1}{C}\sum_{c=1}^{C} y^n(\vec{S}^\nu_m;(\Omega^n_c)^*)\,|\,0\le\nu\le n, \forall\, m\}$,
      $\{\vec{Z}^\nu_m = \frac{1}{C}\sum_{c=1}^{C} \vec{\nabla}y^n(\vec{S}^\nu_m;(\Omega^n_c)^*)\,|\,0\le\nu\le n, \forall\, m\}$.
  else
    Ensemble evaluation (current timestep): overwrite the option prices and deltas
      $\{Y^n_m = \frac{1}{C}\sum_{c=1}^{C} y^n(\vec{S}^n_m;(\Omega^n_c)^*)\,|\,\forall\, m\}$,
      $\{\vec{Z}^n_m = \frac{1}{C}\sum_{c=1}^{C} \vec{\nabla}y^n(\vec{S}^n_m;(\Omega^n_c)^*)\,|\,\forall\, m\}$.
  end if

  Determine whether $\vec{S}^n_m$ is continued or exercised using (33) for all $m$'s.
  Update $\{v^n(\vec{S}^n_m)\,|\,\forall\, m\}$ by (38).
end for

Result: samples of option price and delta functions on the entire spacetime
    $\{Y^n_m \leftarrow \max(Y^n_m, f(\vec{S}^n_m))\,|\,\forall\ \ n, \forall\ \ m\}$ and $\{\vec{Z}^n_m\,|\,\forall\, n, \forall\, m\}$.
Optional result: Recompute the option price and the option delta at $t=0$ using (40) and (41).
```

7.1. Memory

The proposed algorithm requires storing

- the underlying asset prices $\{\vec{S}_m^n \mid \forall\ n, \forall\ m\}$ on the entire spacetime, requiring NMd floating point numbers;
- the training outputs $\{y^n(\vec{S}_m^n), \vec{\nabla} y^n(\vec{S}_m^n) \mid \forall\ m\}$ and the training inputs $\{y^{n+\eta}(\vec{S}_m^n), \vec{\nabla} y^{n+\eta}(\vec{S}_m^n) \mid \forall\ m\}$ at the current timestep, requiring $2(M + Md)$ floating point numbers.

Hence, the entire process requires a total memory of $NMd + 2(M + Md) \approx NMd$ floating point numbers. As a comparison, the Longstaff-Schwartz method requires storing $\{\vec{S}_m^n \mid \forall\ n, \forall\ m\}$ on the entire spacetime and storing $\{\varphi_\chi(\vec{S}_m^n), y^n(\vec{S}_m^n) \mid \forall\ m\}$ at the current timestep. This requires a total memory of $NMd + M \cdot (1/\chi!)d^\chi + M \approx NMd + (1/\chi!)Md^\chi$ floating point numbers. We remind readers that convergence of the Longstaff-Schwartz method to the exact American option prices requires $\chi \to \infty$. As a result, the proposed neural network method is more memory efficient than the Longstaff-Schwartz method.

7.2. Time

Consider a given timestep n. The computational time is dominated by two stages:

- Stage 1: Computing the training inputs (30), in particular, $\{y^{n+\eta}(\vec{S}_m^n; (\Omega^{n+\eta})^*), \vec{\nabla} y^{n+\eta}(\vec{S}_m^n; (\Omega^{n+\eta})^*) \mid \forall\ m\}$, using the trained networks $\{(\Omega^\nu)^* \mid \nu \geq n + \eta\}$.
- Stage 2: Training, using the training inputs (30).

To derive the computational time of each stage, denote the maximal width of the L-layer neural network $\mathcal{F}$ as $d_{\max} \equiv \max_{l=0,\ldots,L} d^{[l]}$. We note that matrix multiplication is the dominant operation in (16)–(18). Hence, for each stage, the computational time *per neural network* is given by $c_1 MLd_{\max}^2$ and $c_2 MLd_{\max}^2$, where c_1 and c_2 are constants. Typically $c_1 \ll c_2$, because Stage 1 only involves computing the outputs of neural networks, while Stage 2 involves training. This seems to suggest that Stage 2 dominates Stage 1. However, we note that Stage 2 involves only one single network (i.e. the nth network), while Stage 1 involves multiple networks from the previous timesteps. More specifically, following the same analysis as (29), one can show that the computation of the training input $y^{n+\eta}(\vec{S}_m^n; (\Omega^{n+\eta})^*)$, given by

$$y^{n+\eta}(\vec{s}) = y^N(\vec{s}) + J\Delta t \cdot \sum_{\nu=1}^{(N-n-\eta)/J} \mathcal{F}(\vec{s}; \Omega^{N-\nu J}), \tag{43}$$

requires going through $(N - n - \eta)/J \approx (N - n)/J$ feedforward networks. As a result, the actual computational time for Stage 1 is $c_1 MLd_{\max}^2 \cdot (N - n)/J$.

Furthermore, if we consider all the N timesteps, then the total computational time is

$$\begin{aligned} &\text{Stage 1: } \sum_{n=0}^{N} c_1 MLd_{\max}^2 \cdot \frac{N-n}{J} = \frac{c_1 N^2}{2J} MLd_{\max}^2, \\ &\text{Stage 2: } \sum_{n=0}^{N} c_2 MLd_{\max}^2 = c_2 NMLd_{\max}^2. \end{aligned} \tag{44}$$

Equation (44) suggests that when N is large, Stage 1 is dominant. However, we can significantly reduce the computational time of Stage 1 by increasing J, as discussed in Section 4.3. In our numerical simulation, we chose $d_{\max} = d + 5$. Then the total computational time of the proposed algorithm is approximately $(c_1 N/2J + c_2)NMLd^2$, which is quadratic in the dimension d.

Regarding the Longstaff-Schwartz method, if we assume that the standard normal equation or QR factorization is used for solving regression problems, then the computational time is $O(NM((1/\chi!)d^\chi)^2) = O(NMd^{2\chi})$, which is worse-than-quadratic in d. Hence, the proposed neural network method is asymptotically more efficient than the Longstaff-Schwartz method in high dimensions.

8. Numerical results

In this section, we solve the American option problem (1)–(4) using our neural network described in Algorithm 1. We compute the price $v(\vec{s}^0, 0)$ and the delta $\vec{\nabla} v(\vec{s}^0, 0)$ at $t = 0$ for given $\vec{s}^0 = (s_1^0, \ldots, s_d^0)$ where $s_1^0 = \cdots = s_d^0 = 0.9K, K$ or $1.1K$. We also compute the prices $v(\vec{s}, t)$ and the deltas $\vec{\nabla} v(\vec{s}, t)$ for sample paths of $(\vec{s}, t)$ spread over the entire spacetime.

In our experiments, we set the strike price $K = 100$, the number of the timesteps $N = 100$, the number of timesteps between the network recurrence $J = 4$, the smoothing parameter in (34) $\kappa = 2/\Delta t$, the coefficient in (38) $\theta = 0.5$. At each timestep, we train an ensemble of $C = 3$ neural networks, where each neural network has a depth of $L = 7$ and a uniform width of $d^{[l]} = d + 5$ across all the hidden layers. We let the number of samples per network be $M = 240{,}000$ (or the total number of samples be $CM = 720{,}000$), and let the batch size and the number of training steps be 400 and 600 respectively. Each numerical experiment is implemented on one Cedar† base-GPU node, which contains 4 NVIDIA P100-PCIE-12GB GPUs, 24 CPUs and 128 GB memory.

We compare the numerical results computed by our proposed method with those computed by the finite difference method, the Longstaff-Schwartz method and the deep neural network method proposed in Sirignano and Spiliopoulos (2018). For the Longstaff-Schwartz method, we choose degree-χ monomial basis (42) with $\chi = 4$. Finite difference solutions with very fine grids are used as exact solutions. We note that this is feasible only if $d \leq 3$. In addition, we remark that the comparison is not made with the other methods referenced in the introduction, such as E *et al.* (2017), Beck *et*

† Cedar is a Compute Canada cluster. For more details, see https://docs.computecanada.ca/wiki/Cedar and https://docs.computecanada.ca/wiki/Using_GPUs_with_Slurm.

al. (2017) and Han *et al.* (2018). This is because their methods solve European option problems but do not discuss the more challenging American option problems, even though their methods also belong to the category of neural networks for option pricing based on BSDEs.

We note that when finite difference solutions are available, we can evaluate the absolute and percent errors of computed prices and deltas. More specifically, denote the finite difference solutions as v_{exact}. Then the percent errors of the price and the delta at $t = 0$ are

$$\frac{|v(\vec{s}^0, 0) - v_{exact}(\vec{s}^0, 0)|}{|v_{exact}(\vec{s}^0, 0)|} \times 100\%,$$
$$\frac{\|\vec{\nabla} v(\vec{s}^0, 0) - \vec{\nabla} v_{exact}(\vec{s}^0, 0)\|_{L_2}}{\|\vec{\nabla} v_{exact}(\vec{s}^0, 0)\|_{L_2}} \times 100\%; \qquad (45)$$

and the percent errors of the spacetime price and the spacetime delta are

$$\frac{\sum_{m,n} |v(\vec{S}^n_m, t^n) - v_{exact}(\vec{S}^n_m, t^n)|}{\sum_{m,n} |v_{exact}(\vec{S}^n_m, t^n)|} \times 100\%,$$
$$\frac{\sum_{m,n} \|\vec{\nabla} v(\vec{S}^n_m, t^n) - \vec{\nabla} v_{exact}(\vec{S}^n_m, t^n)\|_{L_2}}{\sum_{m,n} \|\vec{\nabla} v_{exact}(\vec{S}^n_m, t^n)\|_{L_2}} \times 100\%. \qquad (46)$$

In addition, we can evaluate the quality of the computed exercise boundaries. More specifically, each sample point $(\vec{S}^n_m, t^n)$ is classified as 'exercised' or 'continued' by either the proposed algorithm or other algorithms that we compare with. Meanwhile, the true 'exercised' or 'continued' class of each sample point can be determined by the finite difference method. Let 'exercised' class be the positive class, and denote the numbers of true positive, true negative, false positive and false negative samples as TP, TN, FP, FN, respectively. Then the quality of the exercise boundaries can be evaluated by the f1-score:

$$\text{f1} - \text{score} \equiv \frac{2TP}{2TP + FP + FN}. \qquad (47)$$

The best (or worst) case of the f1-score is 1 (or 0), respectively. We note that another common metric to evaluate the quality of classification problems is the accuracy. Since in all our experiments, the positive class is skewed (around 3–17%), the f1-score would be a better metric than the accuracy (see Murphy 2012, for explanations).

8.1. Multi-dimensional geometric average options

Consider a d-dimensional 'geometric average' American call option, where $\rho_{ij} = \rho$ for $i \neq j$, $\sigma_i = \sigma$ for all i's, and the payoff function is given by $f(\vec{s}) = \max[(\prod_{i=1}^d s_i)^{1/d} - K, 0]$. Although such options are rarely seen in practical applications, they have semi-analytical solutions for benchmarking the performance of our algorithm in high dimensions. More specifically, it is shown in Glasserman (2004) and Sirignano and Spiliopoulos (2018) that such a d-dimensional option can be reduced to a one-dimensional American call option in the variable $s' = (\prod_{i=1}^d s_i)^{1/d}$, where the effective volatility is $\sigma' = \sqrt{(1 + (d-1)\rho)/d}\sigma$ and the effective drift is $r - \delta + \frac{1}{2}(\sigma'^2 - \sigma^2)$. Hence, by solving the equivalent one-dimensional option using finite difference methods, one can compute the d-dimensional option prices and (sometimes) deltas† accurately.

In the following Experiments 1–5, we consider the geometric average option in Section 4.3 of Sirignano and Spiliopoulos (2018), where $\rho_{i,j} = 0.75$, $\sigma = 0.25$, $r = 0$, $\delta = 0.02$, $T = 2$.

Experiment 1 Comparison between our proposed method and the Longstaff-Schwartz method. First we compare the computed prices at $t = 0$; see table 1. Each sub-table includes: the exact prices computed by the Crank-Nicolson finite difference method with 1000 timesteps and 16,385 space grid points, the prices and the corresponding percent errors computed by our proposed method, and the prices and the corresponding percent errors computed by the Longstaff-Schwartz method. For the proposed method, the computed prices are accurate up to 2 decimal places; the percent errors are bounded by 0.34%, and remain approximately the same as the dimension increases. As a comparison, for the Longstaff-Schwartz method, the percent errors deteriorate from 1% to 9% as the dimension increases from 7 to 20. If we keep increasing the dimension towards 100, the Longstaff-Schwartz method encounters an out-of-memory error, because, at $d = 100$, it requires storing $\binom{d+\chi}{d}CM = 3.3 \times 10^{12}$ floating point numbers, or around 23 TB of memory.

The Longstaff-Schwartz algorithm combined with the approaches in Thom (2009) and Broadie and Glasserman (1996) can be used to compute the deltas at $t = 0$. Table 2 compares the deltas at $t = 0$ computed by our proposed approach with the ones computed by the Longstaff-Schwartz algorithm. For the Longstaff-Schwartz algorithm, as the dimension increases from 7 to 20, the percent errors of the deltas worsen from 1.6% to 12.7%; as the dimension continues to increase towards 100, an out-of-memory error occurs. However, for our proposed method, the computed deltas are accurate up to 3 decimal places; the percent errors do not increase with the dimension and stay below 1.7%.

Furthermore, we compare the exercise boundaries computed by the proposed neural network approach with the ones computed by the Longstaff-Schwartz approach. Table 3 evaluates the f1-score of the exercise boundary classification, as defined in (47). For the proposed method, the f1-score remains around 0.95–0.98 as the dimension increases from 7 to 100. For the Longstaff-Schwartz algorithm, the f1-score drops from 0.78 to 0.42 as the dimension increases from 7 to 20. This illustrates a more precise exercise boundary determined by our proposed algorithm.

Figure 5 visualizes the exercise boundaries computed by both algorithms. In order to visualize this, we start with $(\vec{s}^0, t^0) = (1.1K, 0)$ and use the SDE (6)–(7) to generate sample points on the entire spacetime, i.e. $\{(\vec{S}^n_m, t^n) \mid n = 0, \ldots, N; m = 1, \ldots, M\}$; we classify each sample point using

† We note that solving the equivalent one-dimensional option is not sufficient for computing the d-dimensional delta except at the symmetric points $s_1 = \cdots = s_d$. Interested readers can verify this by straightforward algebra.

Table 1. Multi-dimensional geometric average call options: Computed prices at $t = 0$, i.e. $v(\vec{s}^0, 0)$. OOM means 'out-of-memory'.

		Proposed method		Longstaff-Schwartz	
s_i^0	Exact price $v(\vec{s}^0, 0)$	Computed price $v(\vec{s}^0, 0)$	Percent error	Computed price $v(\vec{s}^0, 0)$	Percent error
(i) 7-dimensional geometric average call option					
90	5.9021	5.8822	0.34%	5.8440	0.98%
100	10.2591	10.2286	0.30%	10.1736	0.83%
110	15.9878	15.9738	0.09%	15.8991	0.55%
(ii) 13-dimensional geometric average call option					
90	5.7684	5.7719	0.06%	5.5962	3.0%
100	10.0984	10.1148	0.16%	9.9336	1.6%
110	15.8200	15.8259	0.04%	15.6070	1.4%
(iii) 20-dimensional geometric average call option					
90	5.7137	5.7105	0.06%	5.2023	9.0%
100	10.0326	10.0180	0.15%	9.5964	4.4%
110	15.7513	15.7425	0.06%	15.2622	3.1%
(iv) 100-dimensional geometric average call option					
90	5.6322	5.6154	0.30%	OOM	OOM
100	9.9345	9.9187	0.16%	OOM	OOM
110	15.6491	15.6219	0.17%	OOM	OOM

Table 2. Multi-dimensional geometric average call options: Computed deltas at $t = 0$, i.e. $\vec{\nabla} v(\vec{s}^0, 0)$.

		Proposed method		Longstaff-Schwartz
s_i^0	Exact delta $\vec{\nabla} v(\vec{s}^0, 0)$	Computed delta $\vec{\nabla} v(\vec{s}^0, 0)$	Percent error	Percent error
(i) 7-dimensional geometric average call option				
90	(0.0523, ..., 0.0523)	(0.0516, ..., 0.0516)	1.2%	1.2%
100	(0.0722, ..., 0.0722)	(0.0710, ..., 0.0710)	1.7%	1.6%
110	(0.0912, ..., 0.0912)	(0.0901, ..., 0.0901)	1.2%	1.4%
(ii) 13-dimensional geometric average call option				
90	(0.0279, ..., 0.0279)	(0.0277, ..., 0.0277)	0.76%	5.4%
100	(0.0387, ..., 0.0387)	(0.0384, ..., 0.0384)	0.83%	3.7%
110	(0.0492, ..., 0.0492)	(0.0486, ..., 0.0486)	1.1%	2.6%
(iii) 20-dimensional geometric average call option				
90	(0.0180, ..., 0.0180)	(0.0179, ..., 0.0179)	0.70%	12.7%
100	(0.0251, ..., 0.0251)	(0.0248, ..., 0.0248)	1.2%	8.3%
110	(0.0320, ..., 0.0320)	(0.0316, ..., 0.0316)	1.2%	6.8%
(iv) 100-dimensional geometric average call option				
90	(0.00359, ..., 0.00359)	(0.00357, ..., 0.00357)	0.58%	OOM
100	(0.00502, ..., 0.00502)	(0.00495, ..., 0.00495)	1.3%	OOM
110	(0.00639, ..., 0.00639)	(0.00631, ..., 0.00631)	1.3%	OOM

Note: Note that all the reported deltas in the table are length-d vectors where all the elements are the same. The column 'Longstaff-Schwartz' is the Longstaff-Schwartz method combined with Thom (2009) and Broadie and Glasserman (1996). OOM means 'out-of-memory'.

either our proposed method, i.e. (33), or the Longstaff-Schwartz method; then we project these $(d + 1)$-dimensional points onto the 2-dimensional points $\{(s'^n_m, t^n)\}$, where $s'^n_m = (\prod_{i=1}^d (S_i)^n_m)^{1/d}$ is the geometric average of the underlying asset prices $\vec{S}^n_m$. We use bold dark blue to mark the sample points that should be exercised but are misclassified as continued, and bold dark red to mark the ones that should be continued but are misclassified as exercised. The plots show that the proposed neural network approach (top left and bottom left) has fewer misclassified sample points than the Longstaff-Schwartz approach (top right and bottom right). In other words, the proposed neural network approach yields more precise exercise boundaries.

Experiment 2 Confidence intervals by the proposed method. We repeat the experiments of computing the prices and deltas at $t = 0$ (tables 1–2) for 9 times. tables 4–5 report the mean values of the computed prices and deltas, and the corresponding 95% T-statistic confidence intervals. The last columns of the tables show that, for both the prices and the deltas, the deviations from the mean values remain a constant of $\pm 0.2\%$ as the dimension increases.

Experiment 3 Evaluation of computed spacetime prices and deltas by the proposed method. Our proposed algorithm yields not only the prices and deltas at $t = 0$, but also the prices and deltas for the entire spacetime, which are directly extracted from the output of the neural networks. We emphasize that the computation of spacetime prices and deltas using the Longstaff-Schwartz method is infeasible. The reason is that using the Longstaff-Schwartz method to compute prices and deltas for the entire spacetime would require repeating the algorithm at *every* sample point, noting that the Longstaff-Schwartz method at one sample point is already non-trivial.

Table 3. Multi-dimensional geometric average call options: The f1-score of the exercise boundary classification.

	Proposed method				Longstaff-Schwartz			
s_i^0	$d = 7$	$d = 13$	$d = 20$	$d = 100$	$d = 7$	$d = 13$	$d = 20$	$d = 100$
Geometric average call option								
90	0.96	0.95	0.96	0.95	0.72	0.56	0.42	OOM
100	0.95	0.95	0.97	0.97	0.75	0.61	0.47	OOM
110	0.98	0.96	0.96	0.97	0.78	0.65	0.51	OOM

Note: OOM means ‘out-of-memory’.

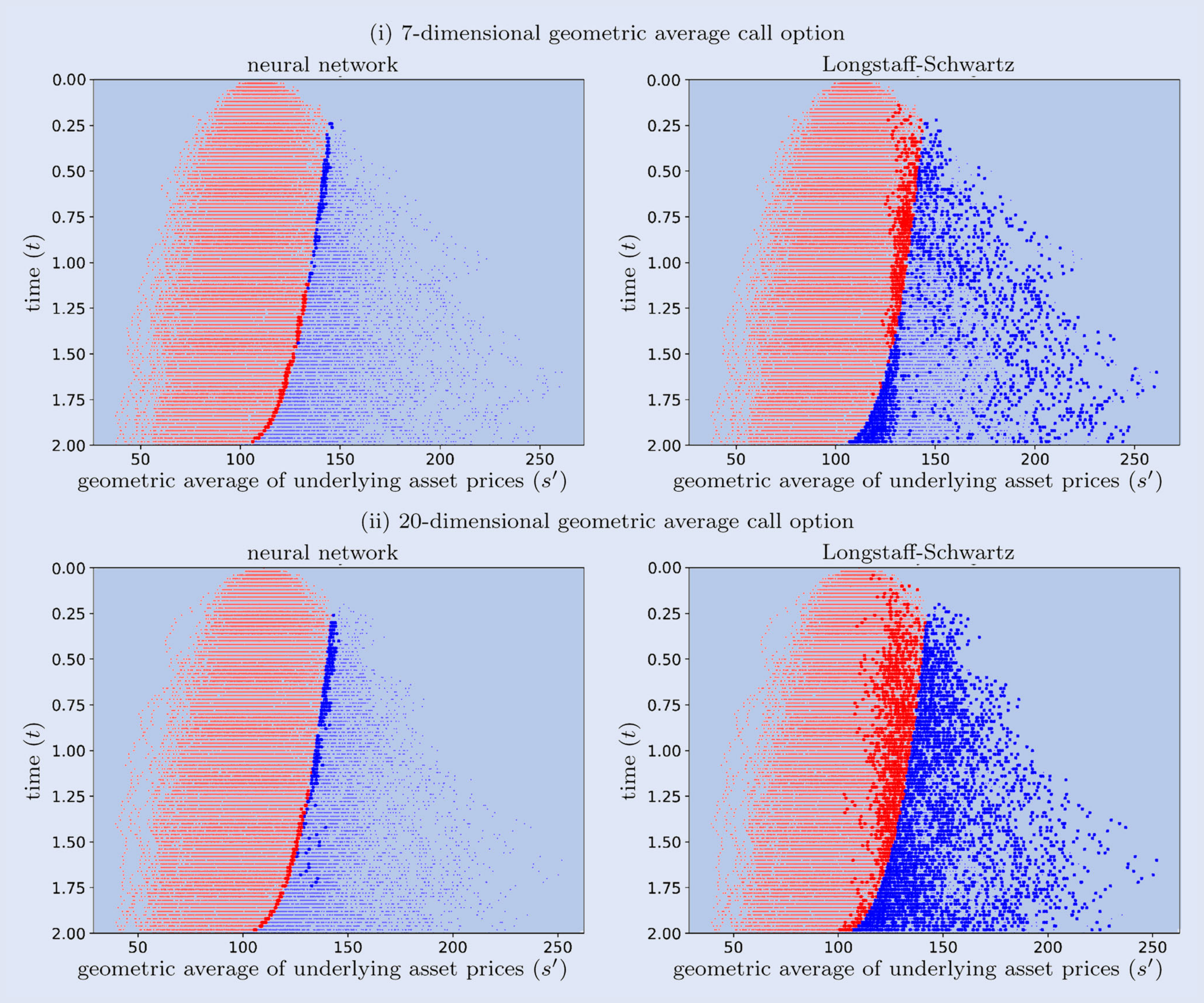


Figure 5. Multi-dimensional geometric average call options: Comparison of exercise boundaries between the proposed neural network approach (top left and bottom left) and the Longstaff-Schwartz approach (top right and bottom right). All blue points: sample points that should be exercised; all red points: sample points that should be continued; bold dark blue points: sample points that should be exercised but are misclassified as continued; bold dark red points: sample points that should be continued but are misclassified as exercised. (i) 7-dimensional geometric average call option and (ii) 20-dimensional geometric average call option.

We also remark that although one may consider using the Longstaff-Schwartz regressed values as an estimate of the spacetime prices, figure 1 in Bouchard and Warin (2012) shows that using such regressed values as the spacetime solution is inaccurate.

First we evaluate the absolute and percent errors of the spacetime price $v(\vec{s}, t)$ and the derivative $(\partial v/\partial s')(s', t)$ computed by our proposed method. Here we evaluate the errors of the derivative $(\partial v/\partial s')(s', t)$ instead of the delta $\vec{\nabla} v(\vec{s}, t)$, because the exact values of the former can be computed by finite difference method spacetime-wise, but not the latter. Table 6 shows that the absolute errors of the spacetime prices and derivatives are around 0.04–0.07 and 0.01 respectively, or in other words, the spacetime prices and derivatives are accurate up to 2 decimal places; the percent errors are less than 1.2% and 3.8%, respectively. We note that the percent

Table 4. Multi-dimensional geometric average call options: mean values and 95% T-statistic confidence intervals (CIs) of the computed prices at $t = 0$, i.e. $v(\vec{s}^0, 0)$, using the proposed neural network method.

d	Exact price $v(\vec{s}^0, 0)$	Mean of computed prices	Percent error	95% CI
Geometric average call option, $s_i^0 = 100$				
7	10.2591	10.2468	0.12%	± 0.0161 ($\pm 0.16\%$)
13	10.0984	10.0822	0.16%	± 0.0201 ($\pm 0.20\%$)
20	10.0326	10.0116	0.21%	± 0.0173 ($\pm 0.17\%$)
100	9.9345	9.9163	0.18%	± 0.0038 ($\pm 0.04\%$)

Table 5. Multi-dimensional geometric average call options: mean values of the computed deltas at $t = 0$, i.e. $\nabla v(\vec{s}^0, 0)$, using the proposed neural network method, and the corresponding 95% T-statistic confidence intervals (CIs) of the first elements of deltas, i.e. $(\partial v/\partial s_1)(\vec{s}^0, 0)$.

d	Exact delta $\nabla v(\vec{s}^0, 0)$	Mean of computed deltas	Percent error	95% CI of $\frac{\partial v}{\partial s_1}(\vec{s}^0, 0)$
Geometric average call option, $s_i^0 = 100$				
7	$(0.0722, \ldots, 0.0722)$	$(0.0717, \ldots, 0.0717)$	0.67%	$\pm 1.8 \times 10^{-4}$ ($\pm 0.25\%$)
13	$(0.0387, \ldots, 0.0387)$	$(0.0384, \ldots, 0.0384)$	0.70%	$\pm 7.3 \times 10^{-5}$ ($\pm 0.19\%$)
20	$(0.0251, \ldots, 0.0251)$	$(0.0249, \ldots, 0.0249)$	0.78%	$\pm 4.2 \times 10^{-5}$ ($\pm 0.17\%$)
100	$(0.00502, \ldots, 0.00502)$	$(0.00498, \ldots, 0.00498)$	0.76%	$\pm 8.9 \times 10^{-6}$ ($\pm 0.18\%$)

Table 6. Multi-dimensional geometric average call options: Spacetime prices and deltas (in terms of absolute and percent errors) computed by our proposed method.

	Spacetime price $v(\vec{s}, t)$		Spacetime derivative $\frac{\partial v}{\partial s'}(s', t)$	
s_i^0	Absolute error	Percent error	Absolute error	Percent error
(i) 7-dimensional geometric average call option				
90	0.0688	1.2%	0.0102	3.3%
100	0.0545	0.54%	0.0102	2.3%
110	0.0450	0.29%	0.0092	1.6%
(ii) 13-dimensional geometric average call option				
90	0.0540	0.94%	0.0101	3.3%
100	0.0475	0.48%	0.0106	2.4%
110	0.0465	0.30%	0.0093	1.6%
(iii) 20-dimensional geometric average call option				
90	0.0567	1.00%	0.0115	3.7%
100	0.0455	0.46%	0.0111	2.5%
110	0.0397	0.26%	0.0090	1.6%
(iv) 100-dimensional geometric average call option				
90	0.0534	0.96%	0.0117	3.8%
100	0.0458	0.47%	0.0107	2.4%
110	0.0480	0.31%	0.0099	1.7%

errors of the spacetime prices and deltas (table 6) are slightly larger than the percent errors of the prices and deltas at $t = 0$ (tables 1–2). This is expected, as the values at $t = 0$ are computed by the improved approach described in Section 5.5.

To visualize the spacetime solutions, we consider the 100-dimensional case, select three time slices $t = 0.5, 1.0, 1.5$, and project the 100-dimensional sample points of $v(\vec{s}, t)$ and $\vec{\nabla} v(\vec{s}, t)$ to 1-dimensional points of $v(s', t)$ and $(\partial v/\partial s')(s', t)$, as shown in figure 6. The spacetime option prices and deltas computed by the proposed neural network approach (the blue/red dots) agree well with the exact solutions by finite difference methods (black lines). We note that small fluctuations exist for the computed spacetime deltas (right subplots), especially near the strike price $K = 100$. This is expected, as the deltas of the payoff functions are discontinuous at the strike price. Smoothing the payoff, as described in Section 5.1, can mitigate this issue, although it does not eliminate the fluctuations.

Experiment 4 Comparison between our proposed method and the method in Sirignano and Spiliopoulos (2018). First we compare the computed prices at $t = 0$; see table 7. Up to 200 dimension is tested. In particular, by comparing the last two columns of the table, we observe that the percent errors computed by our method are bounded by 0.17%, while the ones computed by Sirignano and Spiliopoulos (2018) are bounded by 0.22%.

Next we compare the computed spacetime prices by the two approaches. Figure 7 compares the absolute errors of the spacetime prices. To plot the figure, we start with $(\vec{s}^0, t^0) = (K, 0)$ and use the SDE (6)–(7) to generate sample points on the entire spacetime, i.e. $\{(\vec{S}_m^n, t^n) \mid n = 0, \ldots, N; m = 1, \ldots, M\}$. We compute the error at each sample point, $e(\vec{S}_m^n, t^n) \equiv |v(\vec{S}_m^n, t^n) - v_{exact}(\vec{S}_m^n, t^n)|$. Then we project $\{e(\vec{S}_m^n, t^n)\}$ from $(d + 1)$-dimensional to 2-dimensional space and get the sample points $\{e(s'^n_m, t^n)\}$, where s'^n_m is the geometric average of $\vec{S}_m^n$. From the discrete data points $\{e(s'^n_m, t^n)\}$, we use interpolation to obtain a continuous error function $e(s', t)$ and represent it by a heatmap (also known as filled contour plot), where the x and y axes are the time t and the geometric average s', and the color represents the magnitude of $e(s', t)$. The red, green and blue areas represent the areas where the samples have large, median and small errors, respectively. The white areas are the areas outside the convex hull of the sampled points, where no value of $e(s', t)$ can be interpolated from the sampled $\{e(s'^n_m, t^n)\}$. We remark that this plotting procedure is the same as Sirignano and Spiliopoulos (2018). Indeed, the right subplot of figure 7 is directly taken from Sirignano and Spiliopoulos (2018). In addition, we note that the colored areas of the left and right subplots are not exactly the same. This is because the points on (or near) the boundary of the convex hull are only sampled with

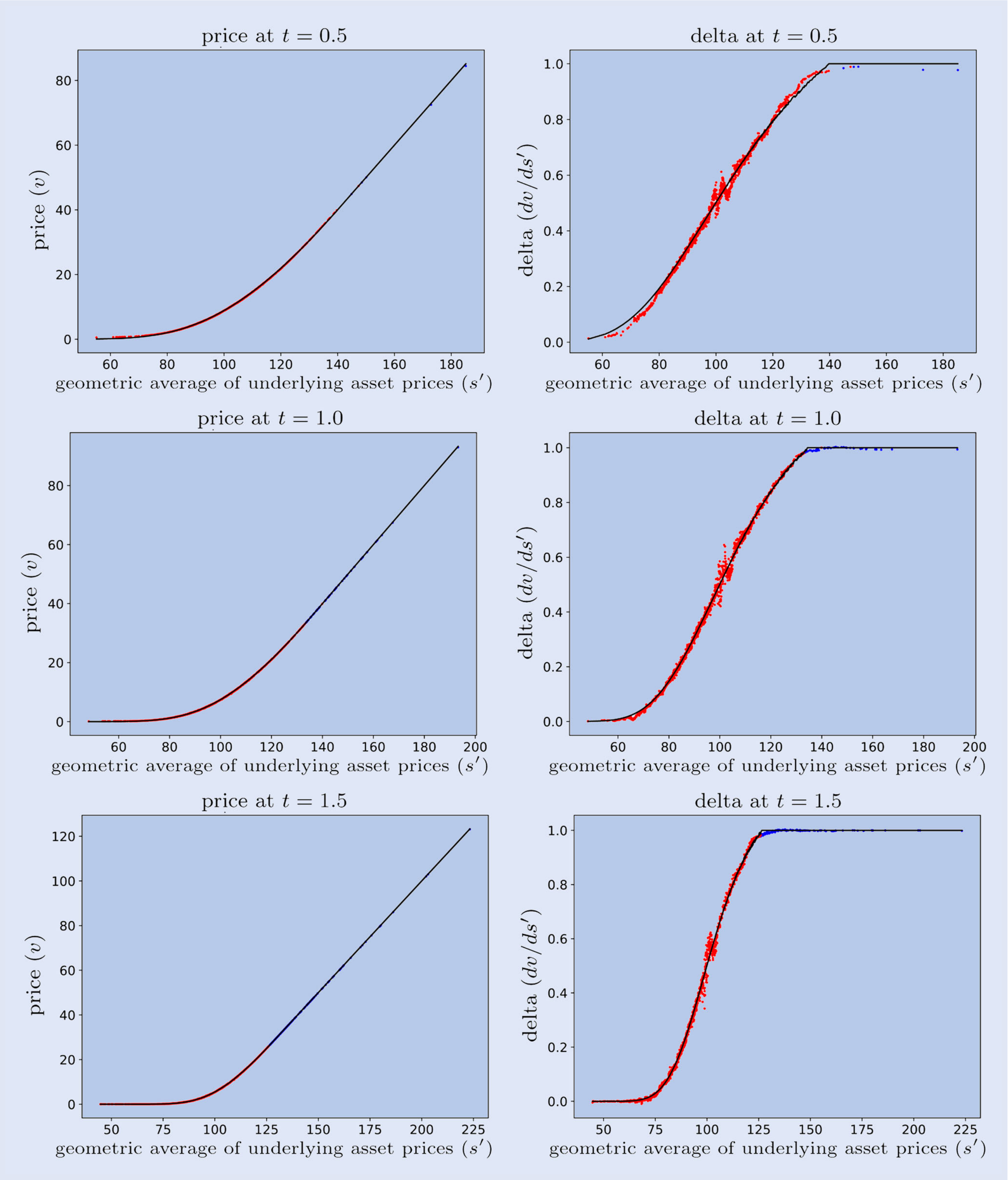


Figure 6. 100-dimensional geometric average call option: Prices (left subplots) and deltas (right subplots) computed by the proposed neural network approach at t =0.5, 1.0, 1.5. The blue/red dots are neural network output values of the exercised/continued sample points. The black lines are the exact solutions computed by finite difference methods.

a small probability and would have a large variation under the two independent stochastic sampling processes that generate the two subplots.

Figure 7(left) shows that the absolute error computed by our proposed approach is close to zero almost on the entire spacetime domain. The error is slightly larger near $(t, s'/K) \approx (0.2, 0.7)$ and bounded by 0.0072. The reason why the error is slightly larger near $t = 0$ is that our proposed approach computes the price in a backward manner, and hence the error may accumulate near $t = 0$. As a comparison, figure 7(right) shows that the error computed by Sirignano and Spiliopoulos (2018) has a larger error in most of the

Table 7. Multi-dimensional geometric average call options: Computed prices at $t = 0$, i.e. $v(\vec{s}^0, 0)$. $s_i^0 = 100$.

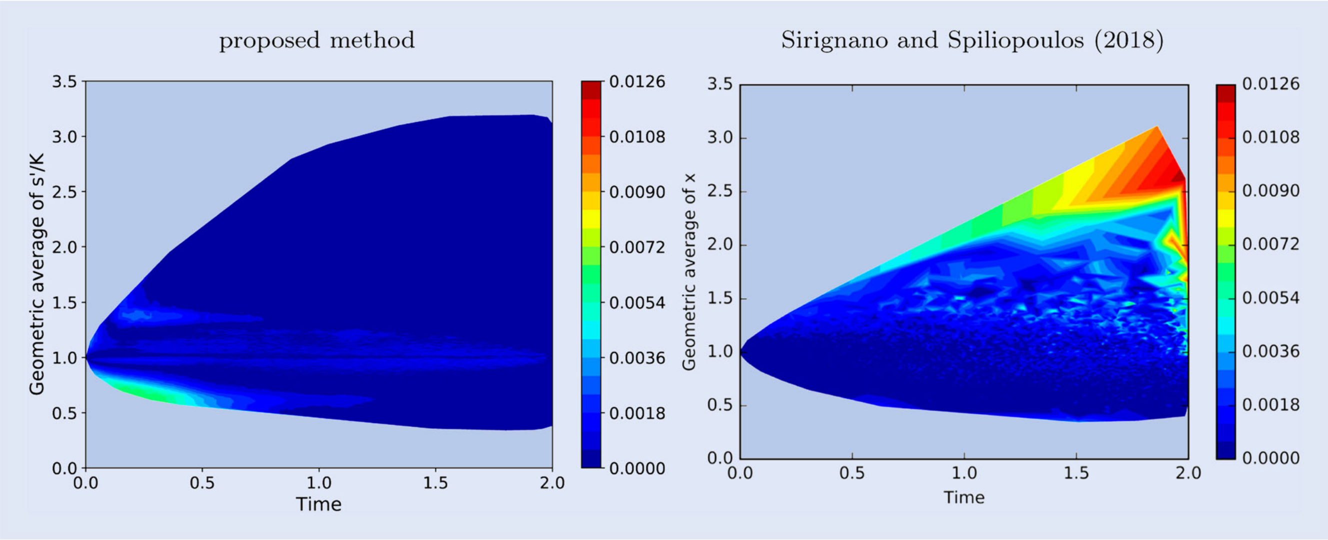
		Proposed method		Sirignano and Spiliopoulos (2018)
d	Exact price $v(\vec{s}^0, 0)$	Computed price $v(\vec{s}^0, 0)$	Percent error	Percent error
Geometric average call option, $s_i^0 = 100$				
3	10.7185	10.7368	0.17%	0.05%
20	10.0326	10.0180	0.15%	0.03%
100	9.9345	9.9187	0.16%	0.11%
200	9.9222	9.9088	0.14%	0.22%

Note: The percent errors reported in table 1 of Sirignano and Spiliopoulos (2018) are also included in the last column of this table.

spacetime domain. In particular, the error reaches 0.0126 near $(t, s'/K) \approx (2.0, 2.7)$, which is larger than the upper bound of our error, 0.0072.

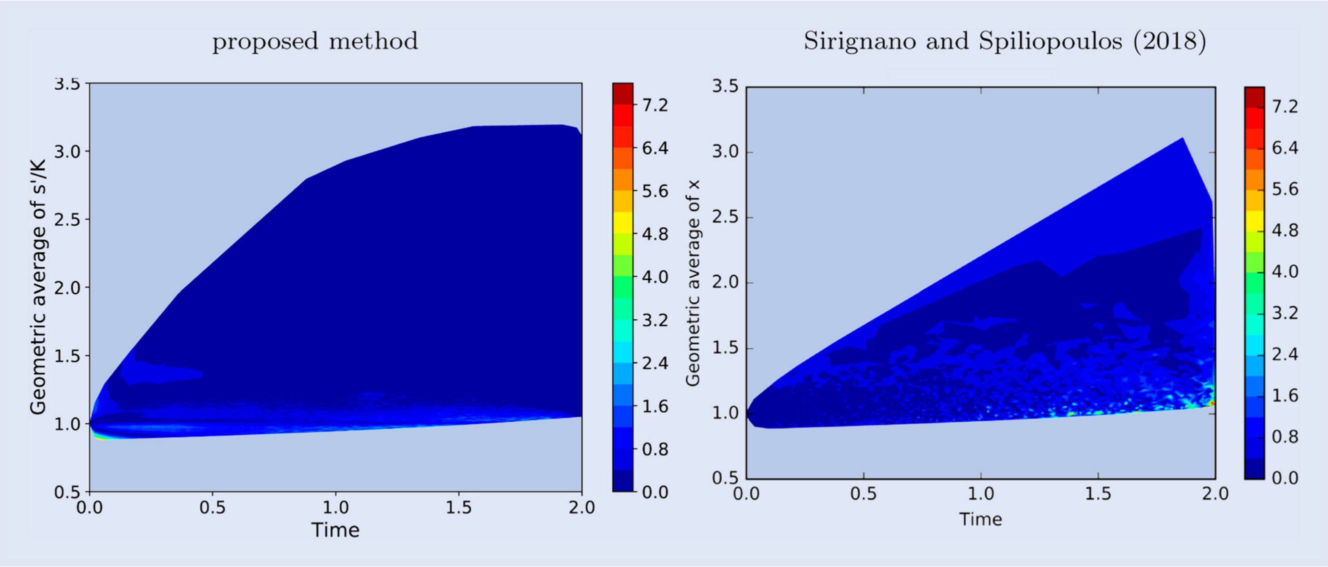
Figure 8 compares the heatmaps of the corresponding percent errors. Following Sirignano and Spiliopoulos (2018), the percent errors are only plotted for the areas where $|v_{exact}(s', t)| > 0.05$. Similar to figure 7, figure 8(left) shows that our proposed approach yields zero error almost everywhere, except that near $(t, s'/K) \approx (0.05, 0.9)$ the error reaches 5.6%. Figure 8(right) shows that the approach in Sirignano and Spiliopoulos (2018) results in a larger error, particularly near $(t, s'/K) \approx (2.0, 1.05)$, where the error reaches 7.2%.

We emphasize that Sirignano and Spiliopoulos (2018) does not compute deltas, whereas our proposed method does yield the deltas. Table 8 reports the deltas at $t = 0$ computed by our proposed method. The percent errors are bounded by 1.3%, and remain approximately the same as the dimension increases. Our approach also computes spacetime deltas, which has been discussed in Experiment 3 and is thus skipped here.


proposed method

Sirignano and Spiliopoulos (2018)

Figure 7. 20-dimensional geometric average call options: Heatmaps of the absolute errors of the computed spacetime prices. Left: absolute error computed by the proposed approach; right: absolute error computed by Sirignano and Spiliopoulos (2018).


proposed method

Sirignano and Spiliopoulos (2018)

Figure 8. 20-dimensional geometric average call options: Heatmaps of the percent errors of the computed spacetime prices. Left: percent error computed by the proposed approach; right: percent error computed by Sirignano and Spiliopoulos (2018).

Table 8. Multi-dimensional geometric average call options: Computed deltas at $t = 0$, i.e. $\vec{\nabla} v(\vec{s}^0, 0)$. $s_i^0 = 100$.

		Proposed method	
d	Exact delta $\vec{\nabla} v(\vec{s}^0, 0)$	Computed delta $\vec{\nabla} v(\vec{s}^0, 0)$	Percent error
Geometric average call option, $s_i^0 = 100$			
3	(0.1702, ..., 0.1702)	(0.1683, ..., 0.1683)	1.1%
20	(0.0251, ..., 0.0251)	(0.0248, ..., 0.0248)	1.2%
100	(0.00502, ..., 0.00502)	(0.00495, ..., 0.00495)	1.3%
200	(0.00251, ..., 0.00251)	(0.00250, ..., 0.00250)	0.53%

Experiment 5 Delta hedging. We perform delta hedging simulations over the period $[0, T]$ with our proposed method. We evaluate the quality of the approach using the distribution of the relative profit and loss (Forsyth 2017, He *et al.* 2006): Relative P&L $\equiv \mathrm{e}^{-rT}\Pi_T / v(\vec{s}^0, 0)$, where Π_T is the balance of an initially-zero hedging portfolio at the expiry T. For perfect hedging, the relative P&L should be a Dirac delta function. Due to the discretization of time, the relative P&L would be close to a normal distribution, where the mean is zero and the standard deviation is a small value depending on Δt. We emphasize that the computation of the relative P&L must use both prices and deltas for the entire spacetime. Hence, none of the existing methods referenced in this paper, except our proposed method, are designed to compute the relative P&L.

Table 10. 2-dimensional max call option: Computed prices at $t = 0$, i.e. $v(\vec{s}^0, 0)$.

		Proposed method		Longstaff-Schwartz	
s_i^0	Exact price $v(\vec{s}^0, 0)$	Computed price $v(\vec{s}^0, 0)$	Percent error	Computed price $v(\vec{s}^0, 0)$	Percent error
2-dimensional max call option					
90	4.2122	4.1992	0.31%	4.1748	0.89%
100	9.6333	9.6080	0.26%	9.5646	0.71%
110	17.3487	17.3313	0.10%	17.2751	0.42%

Table 9 shows the means and the standard deviations of the relative P&Ls for all the 720000 simulation paths, computed by our proposed method. The reported values are indeed close to zero. Figure 9 illustrates the distributions of the relative P&Ls. The resulting distributions are indeed approximately normal distributions with zero means. These results confirm the accuracy of the spacetime prices and the spacetime deltas computed by the proposed method.

8.2. *Multi-dimensional max options*

Multi-dimensional max options are common in practical applications. In this section, we report simulation results for this type of options.

Table 9. Multi-dimensional geometric average call options: Computed means and standard deviations of the relative P&Ls, subject to 100 hedging intervals.

	$d = 7$		$d = 13$		$d = 20$		$d = 100$	
s_i^0	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Geometric average call option								
90	– 0.0023	0.1788	0.0017	0.1827	– 0.0003	0.1877	– 0.0021	0.1908
100	– 0.0016	0.1159	0.0021	0.1170	– 0.0007	0.1184	– 0.0010	0.1184
110	– 0.0001	0.0757	0.0013	0.0755	0.0005	0.0751	– 0.0009	0.0763

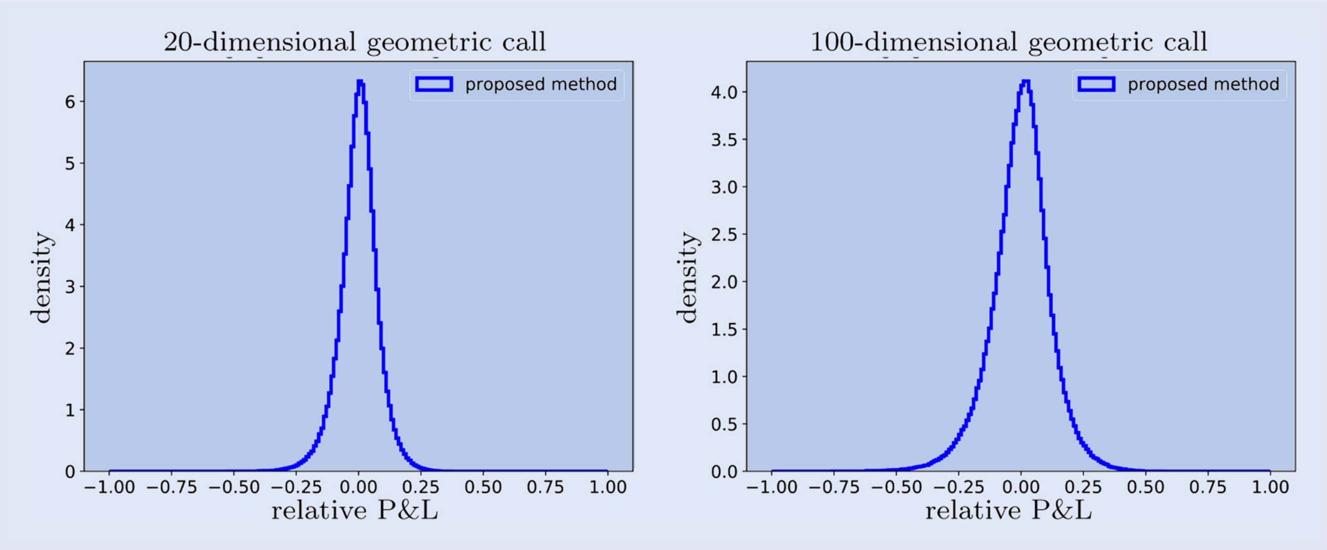


Figure 9. Multi-dimensional geometric call options: Distributions of the relative P&Ls computed by the proposed neural network approach, subject to 100 hedging intervals.

Table 11. 2-dimensional max call option: Computed deltas at $t = 0$, i.e. $\vec{\nabla} v(\vec{s}^0, 0)$.

		Proposed method		Longstaff-Schwartz
s_i^0	Exact delta $\vec{\nabla} v(\vec{s}^0, 0)$	Computed delta $\vec{\nabla} v(\vec{s}^0, 0)$	Percent error	Percent error
2-dimensional max call option				
90	(0.2062, 0.2062)	(0.2025, 0.2019)	1.9%	5.2%
100	(0.3338, 0.3338)	(0.3300, 0.3324)	0.84%	4.4%
110	(0.4304, 0.4304)	(0.4252, 0.4277)	0.96%	3.3%

Note: 'Longstaff-Schwartz' is the Longstaff-Schwartz method combined with Thom (2009) and Broadie and Glasserman (1996).

Table 12. 2-dimensional max call option: Spacetime prices and deltas (in terms of absolute and percent errors) computed by our proposed method.

	Spacetime price $v(\vec{s}, t)$		Spacetime delta $\vec{\nabla} v(\vec{s}, t)$	
s_i^0	Absolute error	Percent error	Absolute error	Percent error
2-dimensional max call option				
90	0.0563	1.3%	0.0155	4.9%
100	0.0828	0.85%	0.0180	3.4%
110	0.0678	0.39%	0.0207	3.0%

Table 13. 2-dimensional max call option: The f1-score of the exercise boundary classification.

s_i^0	Proposed method	Longstaff-Schwartz
2-dimensional max call option		
90	0.93	0.74
100	0.95	0.76
110	0.94	0.79

Experiment 6 2-dimensional max call option. Consider the 2-dimensional max call option from table 3 of Broadie and Glasserman (1997), where the payoff function is $f(\vec{s}) = \max[\max(s_1, s_2) - K, 0]$, and the parameters are $\rho = 0.3$, $\sigma = 0.2$, $r = 0.05$, $\delta = 0.1$, $T = 1$. The reason to consider this example is that the exact prices and deltas are available spacetime-wise. More specifically, we approximate the exact prices and deltas by the Crank-Nicolson finite difference method with 1000 timesteps and 2049 × 2049 space grid points. Hence, we can again benchmark the values computed by our approach with the exact ones.

Using our proposed method, the percent errors of the computed prices at $t = 0$ are less than 0.31% (table 10); the percent errors of the computed deltas at $t = 0$ are less than 1.9% (table 11). These errors are smaller than the corresponding ones computed by the Longstaff-Schwartz method. In addition, the percent errors of the computed spacetime prices and deltas are less than 1.3% and 4.9% (table 12).

Here we also compare the exercise boundary computed by the proposed approaches with the one computed by the Longstaff-Schwartz method. Table 13 shows that the f1-scores computed by our proposed method are around 0.94, higher than the ones computed by the Longstaff-Schwartz algorithm (around 0.76). Figure 10 plots the exercise boundaries at the time slices $t = 0.75$ and 0.5. Similar to figure 5, here the misclassified sample points are highlighted by dark cross markers, and we observe again that the proposed neural network approach has fewer misclassified points than the Longstaff-Schwartz method. Both table 13 and figure 10 illustrate a more accurate exercise boundary determined by our proposed method than by the Longstaff-Schwartz method.

In addition, we compute the relative P&Ls by the finite difference method† and compare them with the values computed by our approach. table 14 and figure 11 show the means, standard deviations and the distributions of the relative P&Ls computed by the proposed approach versus by finite difference methods. The results computed by the proposed approach are similar to the ones computed by finite difference methods. This again verifies the accuracy of the spacetime prices and deltas computed by our proposed algorithm.

Experiment 7 5-dimensional max call option. We study the 5-dimensional max call option from table 3.5 of Firth (2005), where $\rho_{i,j} = 0$, $\sigma = 0.2$, $r = 0.05$, $\delta = 0.1$, $T = 3$. We note that unlike the previous experiments, here the exact solutions are not available. Table 15 reports the option prices and deltas at $t = 0$ computed by the proposed method. The table also includes the Longstaff-Schwartz prices reported in Firth (2005). The prices given by the proposed algorithm and the Longstaff-Schwartz method differ by 10^{-2}. We note that the Longstaff-Schwartz method is a low-biased method due to its sub-optimal computed exercise boundary, as explained in Longstaff and Schwartz (2001) and Firth (2005). The proposed algorithm gives slightly higher prices.

9. Conclusion

We propose a neural network framework for high-dimensional American option problems. Our algorithm minimizes the residual of the backward stochastic differential equation that couples both prices and deltas. The neural network is designed to learn the differences between the price functions of the adjacent timesteps. We improve the algorithm by various

† We note that even though finite difference methods yield nearly exact spacetime prices and deltas, due to the finite number of hedging intervals, the resulting relative P&Ls are not a Dirac delta distribution.

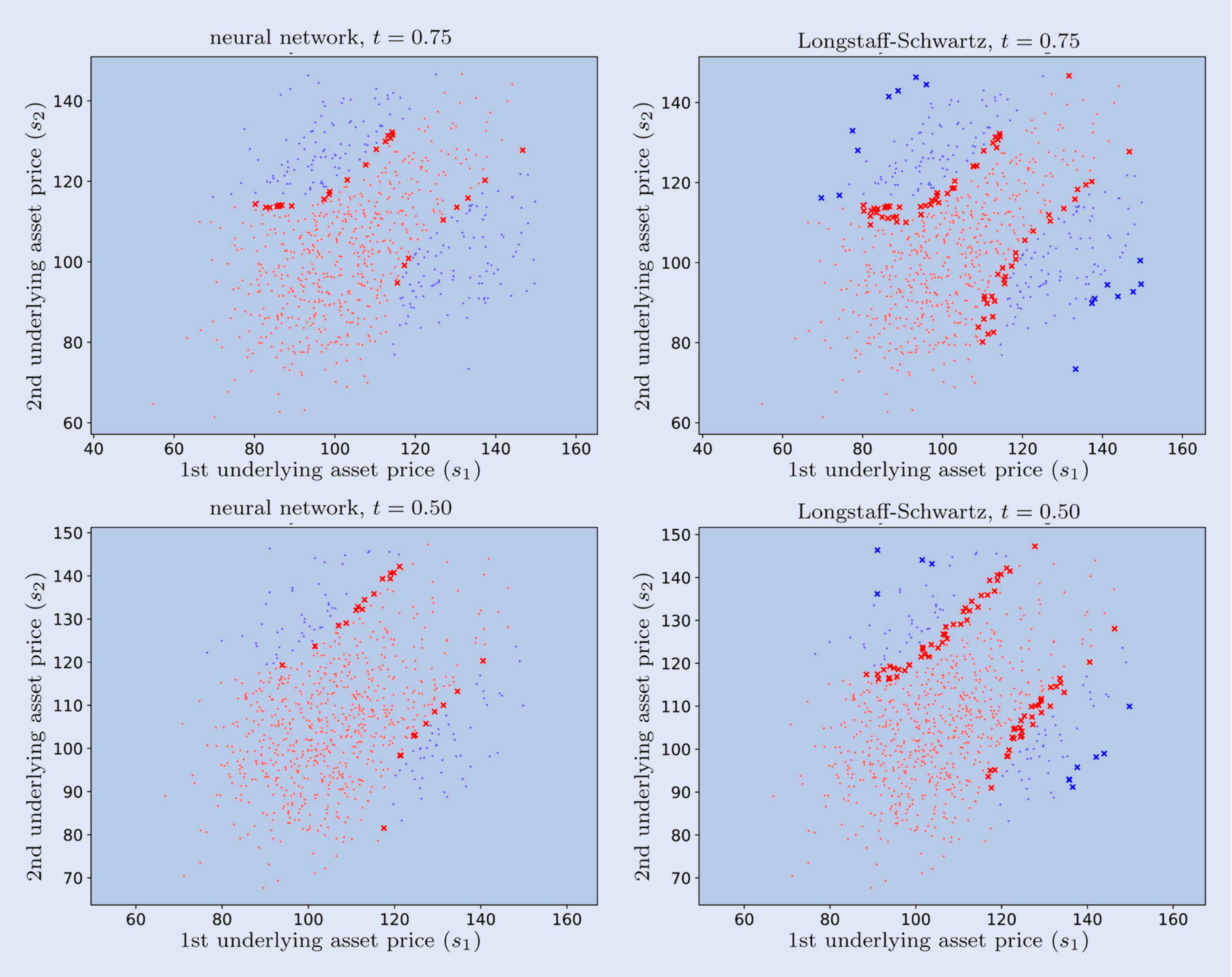


Figure 10. 2-dimensional max call option: Comparison of exercise boundaries between the proposed neural network approach (top left and bottom left) and the Longstaff-Schwartz approach (top right and bottom right). Note that only the time slices of t =0.75 and 0.5 are plotted. All blue points: sample points that should be exercised; all red points: sample points that should be continued; bold dark blue points: sample points that should be exercised but are misclassified as continued; bold dark red points: sample points that should be continued but are misclassified as exercised.

Table 14. 2-dimensional max call option: Means and standard deviations of the relative P&Ls by finite difference versus by the proposed method, subject to 100 hedging intervals.

	Finite difference		Proposed method	
s_i^0	Mean	Std	Mean	Std
2-dimensional max call option				
90	0.0025	0.1683	0.0022	0.1932
100	0.0014	0.0894	0.0016	0.0990
110	0.0011	0.0544	0.0016	0.0614

techniques, including feature selection, weight reuse, ensemble learning, redefining training input ‘v’, etc. The proposed algorithm yields not only the prices and deltas at $t = 0$, but also the prices and deltas for the entire spacetime. The cost of the proposed algorithm grows quadratically with the dimension d, which mitigates the curse of dimensionality. In particular, our algorithm outperforms the Longstaff-Schwartz algorithm when $d \geq 20$.

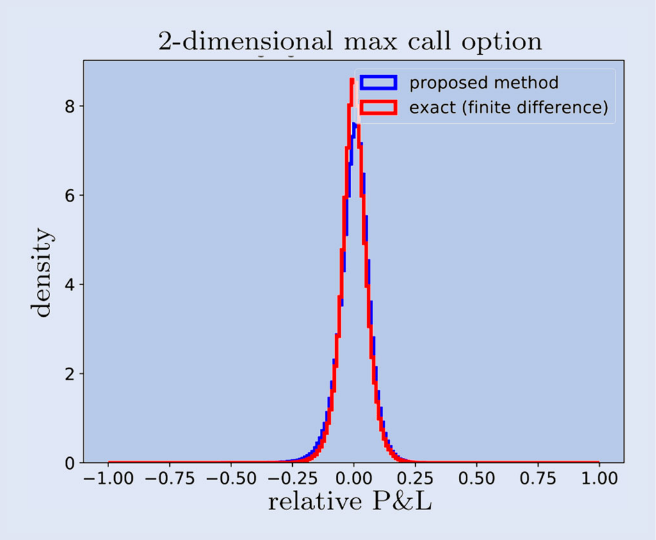


Figure 11. 2-dimensional max call option: Comparison of the distributions of the relative P&Ls computed by the proposed neural network approach (blue) versus by finite difference method (red), subject to 100 hedging intervals.

Table 15. 5-dimensional max call option: Computed prices and deltas at $t = 0$, i.e. $v(\vec{s}^0, 0)$ and $\vec{\nabla} v(\vec{s}^0, 0)$.

	Computed price $v(\vec{s}^0, 0)$		
s_i^0	Proposed method	Longstaff-Schwartz	Computed delta $\vec{\nabla} v(\vec{s}^0, 0)$ by proposed method
5-dimensional max call option			
90	16.8896	16.76	(0.1728, 0.1732, 0.1747, 0.1754, 0.1738)
100	26.4876	26.28	(0.2017, 0.2004, 0.1998, 0.2071, 0.2041)
110	37.0996	36.89	(0.2157, 0.2198, 0.2190, 0.2149, 0.2202)

Note: The column ‘Longstaff-Schwartz’ is the Longstaff-Schwartz prices reported in Firth (2005).

We note that the main drawback of the proposed algorithm is that the computational cost is quadratic (rather than linear) in the number of the timesteps N, even though a mitigation is proposed in Section 4.3. A potential future work is to re-design the architecture of the neural network in order to improve this drawback.

A version of the code is provided at https://github.com/yangangchen/Amer_Op_Neural_Net. This version of code is written for CPU machines (rather than GPU machines). Interested readers are recommended to optimize the code based on their GPU machines, noting that the way to optimize the code can vary significantly from one GPU machine to another.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This work was supported by Natural Sciences and Engineering Research Council of Canada.

ORCID

Yangang Chen http://orcid.org/0000-0002-5101-7596
Justin W. L. Wan http://orcid.org/0000-0001-8367-6337

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X. and Google Brain, Tensorflow: A system for large-scale machine learning. In *Proceedings of the OSDI*, Savannah, GA, USA, Vol. 16, pp. 265–283, 2016.

Achdou, Y. and Pironneau, O., Computational methods for option pricing. In *Frontiers in Applied Mathematics*, Vol. 30, 2005 (Society for Industrial and Applied Mathematics (SIAM): Philadelphia, PA).

Beck, C., E, W. and Jentzen, A., Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *J. Nonlinear Sci.*, 2019, **29**, 1563–1619.

Becker, S., Cheridito, P. and Jentzen, A., Deep optimal stopping. *J. Mach. Learn. Res.*, 2019a, **20**, 1–25.

Becker, S., Cheridito, P., Jentzen, A. and Welti, T., Solving high-dimensional optimal stopping problems using deep learning. Preprint, 2019b. arXiv:1908.01602.

Bouchard, B. and Warin, X., Monte-Carlo valuation of American options: Facts and new algorithms to improve existing methods. In *Numerical Methods in Finance*, Vol. 12 of Springer proc. math., pp. 215–255, 2012 (Springer: Heidelberg).

Broadie, M. and Glasserman, P., Estimating security price derivatives using simulation. *Manage. Sci.*, 1996, **42**, 269–285.

Broadie, M. and Glasserman, P., Pricing American-style securities using simulation. *J. Econom. Dynam. Control*, 1997, **21**, 1323–1352. Computational financial modelling.

Broadie, M. and Glasserman, P., A stochastic mesh method for pricing high-dimensional American options. *J. Comput. Finance*, 2004, **7**, 35–72.

Duffy, D.J., *Finite Difference Methods in Financial Engineering*, Wiley finance series, 2006 (John Wiley & Sons, Ltd.: Chichester). A partial differential equation approach, With 1 CD-ROM (Windows, Macintosh and UNIX).

E, W., Han, J. and Jentzen, A., Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Commun. Math. Stat.*, 2017, **5**, 349–380.

El Karoui, N., Peng, S. and Quenez, M.C., Backward stochastic differential equations in finance. *Math. Finance*, 1997, **7**, 1–71.

Firth, N.P., High dimensional American options. PhD Thesis, University of Oxford, 2005.

Forsyth, P.A. and Vetzal, K.R., Quadratic convergence for valuing American options using a penalty method. *SIAM J. Sci. Comput.*, 2002, **23**, 2095–2122.

Forsyth, P., An introduction to computational finance without agonizing pain, 2017.

Fujii, M., Takahashi, A. and Takahashi, M., Asymptotic expansion as prior knowledge in deep learning method for high dimensional BSDEs. Preprint, 2017. arXiv:1710.07030.

Glasserman, P., *Monte Carlo Methods in Financial Engineering*, Applications of mathematics (New York) Vol. 53, 2004 (Springer-Verlag: New York). Stochastic Modelling and Applied Probability.

Goodfellow, I., Bengio, Y. and Courville, A., *Deep Learning*, Adaptive computation and machine learning, 2016 (MIT Press: Cambridge, MA).

Han, J., Jentzen, A. and E, W., Solving high-dimensional partial differential equations using deep learning. *Proc. Natl. Acad. Sci. USA*, 2018, **115**, 8505–8510.

Haugh, M.B. and Kogan, L., Pricing American options: A duality approach. *Oper. Res.*, 2004, **52**, 258–270.

He, C., Kennedy, J.S., Coleman, T.F., Forsyth, P.A., Li, Y. and Vetzal, K.R., Calibration and hedging under jump diffusion. *Rev. Deriv. Res.*, 2006, **9**, 1–35.

He, K., Zhang, X., Ren, S. and Sun, J., Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, USA, pp. 770–778, 2016.

Heston, S. and Zhou, G., On the rate of convergence of discrete-time contingent claims. *Math. Finance*, 2000, **10**, 53–75.

Hull, J.C., *Options Futures and Other Derivatives*, 2003 (Pearson/Prentice Hall: Upper Saddle River, NJ).

Huré, C., Pham, H. and Warin, X., Some machine learning schemes for high-dimensional nonlinear PDEs. Preprint, 2019. arXiv:1902.01599.

Kennedy, J.S., Forsyth, P.A. and Vetzal, K.R., Dynamic hedging under jump diffusion with transaction costs. *Oper. Res.*, 2009, **57**, 541–559.

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

Kohler, M., A review on regression-based Monte Carlo methods for pricing American options. In *Recent Developments in Applied Probability and Statistics*, pp. 37–58, 2010 (Springer: Berlin, Heidelberg).

Kohler, M., Krzyżak, A. and Todorovic, N., Pricing of high-dimensional American options by neural networks. *Math. Finance*, 2010, **20**, 383–410.

Leentvaar, C.C.W., Pricing multi-asset options with sparse grids, 2008.

Longstaff, F.A. and Schwartz, E.S., Valuing American options by simulation: A simple least-squares approach. *Rev. Financ. Stud.*, 2001, **14**, 113–147.

Murphy, K.P., *Machine Learning: A Probabilistic Perspective*, 2012 (MIT Press: Cambridge, MA).

Sirignano, J. and Spiliopoulos, K., DGM: A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.*, 2018, **375**, 1339–1364.

Sola, J. and Sevilla, J., Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Trans. Nucl. Sci.*, 1997, **44**, 1464–1468.

Stentoft, L., Convergence of the least squares Monte Carlo approach to American option valuation. *Manage. Sci.*, 2004, **50**, 1193–1203.

Thom, H., Longstaff Schwartz pricing of Bermudan options and their Greeks, 2009.

Tsitsiklis, J.N. and Van Roy, B., Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives. *IEEE Trans. Automat. Control*, 1999, **44**, 1840–1851.