Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations

Maziar Raissi

Division of Applied Mathematics, Brown University, Providence, RI, 02912, USA

Abstract

Classical numerical methods for solving partial differential equations suffer from the curse dimensionality mainly due to their reliance on meticulously generated spatio-temporal grids. Inspired by modern deep learning based techniques for solving forward and inverse problems associated with partial differential equations, we circumvent the tyranny of numerical discretization by devising an algorithm that is scalable to high-dimensions. In particular, we approximate the unknown solution by a deep neural network which essentially enables us to benefit from the merits of automatic differentiation. To train the aforementioned neural network we leverage the well-known connection between high-dimensional partial differential equations and forward-backward stochastic differential equations. In fact, independent realizations of a standard Brownian motion will act as training data. We test the effectiveness of our approach for a couple of benchmark problems spanning a number of scientific domains including Black-Scholes-Barenblatt and Hamilton-Jacobi-Bellman equations, both in 100-dimensions.

Keywords: forward-backward stochastic differential equations, Black-Scholes equations, Hamilton-Jacobi-Bellman equations, stochastic control, deep learning, automatic differentiation

1. Introduction

Since their introduction [1, 2], backward stochastic differential equations have found many applications in areas like stochastic control, theoretical

economics, and mathematical finance. They have received considerable attention in the literature and interesting connections to partial differential equations have been obtained (see e.g., [3] and the references therein). The key feature of backward stochastic differential equations is the random terminal condition that the solution is required to satisfy. These equations are referred to as forward-backward stochastic differential equations, if the randomness in the terminal condition is coming from the state of a forward stochastic differential equation. The solution to a forward-backward stochastic differential equation can be written as a deterministic function of time and the state process. Under suitable regularity assumptions, this function can be shown to be the solution of a parabolic partial differential equation [3]. A forward-backward stochastic differential equation is called uncoupled if the solution of the backward equation does not enter the dynamics of the forward equation and coupled if it does. The corresponding parabolic partial differential equation is semi-linear in case the forward-backward stochastic differential equation is uncoupled and quasi-linear if it is coupled.

In this work, we approximate the aforementioned deterministic function of time and space by a deep neural network. This choice is inspired by modern techniques for solving forward and inverse problems associated with partial differential equations, where the unknown solution is approximated either by a neural network [4–6] or a Gaussian process [7–10]. Moreover, putting a prior on the solution is fully justified by the similar approach pursued in the past century by classical methods of solving partial differential equations such as finite elements, finite differences, or spectral methods, where one would expand the unknown solution in terms of an appropriate set of basis functions. However, the classical methods suffer from the curse of dimensionality mainly due to their reliance on spatio-temporal grids. In contrast, modern techniques avoid the tyranny of mesh generation, and consequently the curse of dimensionality, by approximating the unknown solution with a neural network or a Gaussian process. Moreover, unlike the state of the art deep learning based algorithms for solving high-dimensional partial differential equations [11–13], our algorithm (upon a single round of training) results in a solution function that can be evaluated anywhere in the space-time domain, not just at the initial point.

2. Problem Setup and Solution methodology

We consider coupled forward-backward stochastic differential equations of the general form

$$\begin{aligned} &dX_t = \mu(t, X_t, Y_t, Z_t)dt + \sigma(t, X_t, Y_t)dW_t, \quad t \in [0, T], \\ &X_0 = \xi, \\ &dY_t = \varphi(t, X_t, Y_t, Z_t)dt + Z_t'\sigma(t, X_t, Y_t)dW_t, \quad t \in [0, T), \\ &Y_T = g(X_T), \end{aligned} \tag{1}$$

where W_t is a vector-valued Brownian motion. A solution to these equations consists of the stochastic processes X_t, Y_t, and Z_t. It is shown in [14] and [15] (see also [3, 16, 17]) that coupled forward-backward stochastic differential equations (1) are related to quasi-linear partial differential equations of the form

$$u_t = f(t, x, u, Du, D^2u), \tag{2}$$

with terminal condition $u(T, x) = g(x)$, where $u(t, x)$ is the unknown solution and

$$f(t, x, y, z, \gamma) = \varphi(t, x, y, z) - \mu(t, x, y, z)'z - \frac{1}{2}\mathrm{Tr}[\sigma(t, x, y)\sigma(t, x, y)'\gamma]. \tag{3}$$

Here, Du and D^2u denote the gradient vector and the Hessian matrix of u, respectively. In particular, it follows directly from Ito's formula (see e.g., [3]) that solutions of equations (1) and (2) are related according to

$$Y_t = u(t, X_t), \text{ and } Z_t = Du(t, X_t). \tag{4}$$

Inspired by recent developments in *physics-informed deep learning* [4, 5] and *deep hidden physics models* [6], we proceed by approximating the unknown solution $u(t, x)$ by a deep neural network. We obtain the required gradient vector $Du(t, x)$ by applying the chain rule for differentiating compositions of functions using automatic differentiation [18]. It is worth emphasizing that automatic differentiation is different from, and in several respects superior to, numerical or symbolic differentiation; two commonly encountered techniques of computing derivatives. In its most basic description [18], automatic differentiation relies on the fact that all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known. Combining the derivatives of the constituent operations through

the chain rule gives the derivative of the overall composition. This allows accurate evaluation of derivatives at machine precision with ideal asymptotic efficiency and only a small constant factor of overhead. In particular, to compute the derivatives involved in equation (4) we rely on Tensorflow [19] which is a popular and relatively well documented open source software library for automatic differentiation and deep learning computations.

Parameters of the neural network representing $u(t,x)$ can be learned by minimizing the following loss function given explicitly in equation (6) obtained from discretizing the forward-backward stochastic differential equation (1) using the standard Euler-Maruyama scheme. To be more specific, let us apply the Euler-Maruyama scheme to the set of equations (1) and obtain

$$\begin{aligned} X^{n+1} &\approx X^n + \mu(t^n, X^n, Y^n, Z^n)\Delta t^n + \sigma(t^n, X^n, Y^n)\Delta W^n, \\ Y^{n+1} &\approx Y^n + \varphi(t^n, X^n, Y^n, Z^n)\Delta t^n + (Z^n)'\sigma(t^n, X^n, Y^n)\Delta W^n, \end{aligned} \tag{5}$$

for $n = 0, 1, \ldots, N-1$, where $\Delta t^n := t^{n+1} - t^n = T/N$ and $\Delta W^n \sim \mathcal{N}(0, \Delta t^n)$ is a random variable with mean 0 and standard deviation $\sqrt{\Delta t^n}$. The loss function is then given by

$$\sum_{m=1}^{M} \sum_{n=0}^{N-1} |Y_m^{n+1} - Y_m^n - \Phi_m^n \Delta t^n - (Z_m^n)'\Sigma_m^n \Delta W_m^n|^2 + \sum_{m=1}^{M} |Y_m^N - g(X_m^N)|^2, \tag{6}$$

which corresponds to M different realizations of the underlying Brownian motion. Here, $\Phi_m^n := \varphi(t^n, X_m^n, Y_m^n, Z_m^n)$ and $\Sigma_m^n := \sigma(t^n, X_m^n, Y_m^n)$. The subscript m corresponds to the m-th realization of the underlying Brownian motion while the superscript n corresponds to time t^n. It is worth recalling from equations (4) that $Y_m^n = u(t^n, X_m^n)$ and $Z_m^n = Du(t^n, X_m^n)$, and consequently the loss (6) is a function of the parameters of the neural network $u(t,x)$. Furthermore, from equation (5) we have

$$X_m^{n+1} = X_m^n + \mu(t^n, X_m^n, Y_m^n, Z_m^n)\Delta t^n + \sigma(t_m^n, X_m^n, Y_m^n)\Delta W_m^n,$$

and $X_m^0 = \xi$ for every m.

3. Related Work[1]

In [12, 13], the authors consider uncoupled forward-backward stochastic differential equations of the form

$$\begin{aligned} &dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \quad t \in [0, T], \\ &X_0 = \xi, \\ &dY_t = \varphi(t, X_t, Y_t, Z_t)dt + Z_t'\sigma(t, X_t)dW_t, \quad t \in [0, T), \\ &Y_T = g(X_T), \end{aligned} \tag{7}$$

which are subcases of the coupled equations (1) studied in the current work. The above equations are related to the semilinear and parabolic class of partial differential equations

$$u_t = \varphi(t, x, Du, D^2u) - \mu(t, x)'Du - \frac{1}{2}\mathrm{Tr}[\sigma(t, x)\sigma(t, x)'D^2u]. \tag{8}$$

The authors of [12, 13] then devise an algorithm to compute $Y_0 = u(0, X_0) = u(0, \xi)$ by treating Y_0 and $Z_0 = Du(0, \xi)$ as parameters in their model. Then, they employ the Euler-Maruyama scheme to discretize equations (7). Their next step is to approximate the functions $Du(t^n, x)$ for $n = 1, \ldots, N-1$ at time steps t^n by $N-1$ different neural networks. This enables them to approximate $Z^n = Du(t^n, X^n)$ by evaluating the corresponding neural network at time t^n at the spatial point X^n. Moreover, no neural networks are employed in [12, 13] to approximate the functions $u(t^n, x)$. In fact $Y^n = u(t^n, X^n)$ is computed by time marching using the Euler-Maruyama scheme used to discretize equations (7). Their loss function is then given by

$$\sum_{m=1}^{M} |Y_m^N - g(X_m^N)|^2, \tag{9}$$

which tries to match the terminal condition. The total set of parameters are consequently given by Y_0, Z_0, and the parameters of the $N-1$ neural networks used to approximate the gradients. There are a couple of major drawbacks associated with the method advocated in [12, 13].

The first and the most obvious drawback is that the number of param-

[1] This section can be skipped in the first read.

eters involved in their model grows with the number of points N used to discretized time. This is prohibitive specially in cases where one would need to perform long time integration (i.e., when the final time T is large) or in cases where it is a requirement to employ smaller time step size Δt in order to increase the accuracy of the Euler-Maruyama scheme. The second major drawback is that the method as outlined in [12, 13] is designed in such a way that it is only capable of approximating $Y_0 = u(0, X_0) = u(0, \xi)$. This means that in order to obtain an approximation to $Y_t = u(t, X_t)$ at a later time $t > 0$, they will have to retrain their algorithm. The third drawback is that assuming Y_0 and Z_0 to act as parameters of the models in addition to approximating the gradients by $N - 1$ distinct (not sharing any parameters) neural networks seems a little bit ad-hoc.

In contrast, the method proposed in the current work circumvents all of the drawbacks mentioned above by placing a neural network directly on the object of interest, the unknown solution $u(t, x)$. This choice is justified by the similar well-established approach taken by the classical methods of solving partial differential equations, such as finite elements, finite differences, or spectral methods, where one would expand the unknown solution $u(t, x)$ in terms of an appropriate set of basis functions. In addition, modern methods for solving forward and inverse problems associated with partial differential equations approximate the unknown solution $u(t, x)$ by either a neural network [4–6, 20] or a Gaussian process [7–10, 21–23]. The classical methods suffer from the curse of dimensionality mainly due to their reliance on spatio-temporal grids. Here, inspired by the aforementioned modern techniques, we avoid the curse of dimensionality by approximating $u(t, x)$ with a neural network. It should be highlighted that the number of parameters of the neural network we use to approximate $u(t, x)$ is independent of the number of the number of points N needed to discretized time (see equation (5)). Moreover, upon a single round of training, the neural network representing $u(t, x)$ can be evaluated anywhere in the space-time domain, not just at the initial point $u(0, X_0)$. Furthermore, we compute the required gradients $Du(t, x)$ by differentiating the neural network representing $u(t, x)$ using automatic differentiation. Consequently, the networks $Du(t, x)$ and $u(t, x)$ share the same set of parameters. This is fully justified by the theoretical connection (see equation (4)) between solutions of forward-backward stochastic differential equations and their associated partial differential equations. A major advantage of the approach pursued in the current work is the reduction in the

number of parameters employed by our model, which helps the algorithm generalize better during test time and consequently mitigate the well-known over-fitting problem.

In [11], a follow-up work on [12, 13], the authors extend their framework to fully-nonlinear second-order partial differential equations of the general form

$$u_t = f(t, x, u, Du, D^2u), \tag{10}$$

with terminal condition $u(T, x) = g(x)$. Here, let X_t denote a high-dimensional stochastic process satisfying the forward stochastic differential equation

$$\begin{aligned} &dX_t = \mu(X_t)dt + \sigma(X_t)dW_t, \\ &X_0 = \xi, \end{aligned} \tag{11}$$

where $\mu(X_t)$ is the drift vector and $\sigma(X_t)$ is the diffusion matrix. It then follows directly from Ito's formula [3] that the processes

$$\begin{aligned} &Y_t := u(t, X_t), \\ &Z_t := Du(t, X_t), \\ &\Gamma_t := D^2u(t, X_t), \\ &A_t := \mathcal{L}Du(t, X_t) := Du_t(t, X_t) + \tfrac{1}{2}D\mathrm{Tr}[D^2u(t, X_t)\sigma(X_t)\sigma(X_t)^T], \end{aligned} \tag{12}$$

solve the second-order backward stochastic differential equation

$$\begin{aligned} &dY_t = f(t, X_t, Y_t, Z_t, \Gamma_t)dt + \tfrac{1}{2}\mathrm{Tr}[\Gamma_t\sigma(X_t)\sigma(X_t)^T]dt + Z_t^T dX_t, \\ &dZ_t = A_t dt + \Gamma_t dX_t, \\ &Y_T = g(X_T). \end{aligned} \tag{13}$$

Similar to their prior works [12, 13], the authors then devise an algorithm to compute $Y_0 = u(0, X_0) = u(0, \xi)$ by treating Y_0, $Z_0 = Du(0, \xi)$, $\Gamma_0 = D^2u(0, \xi)$, and $A_0 = \mathcal{L}Du(0, \xi)$ as parameters of their model. Then, they proceed by discretizing equations (13) by the Euler-Maruyama scheme. Their next step is to approximate the functions $D^2u(t^n, x)$ and $\mathcal{L}Du(t^n, x)$ for $n = 1, \ldots, N-1$, corresponding to each time step t^n, by $2(N-1)$ distinct neural networks. This enables them to approximate $\Gamma^n = D^2u(t^n, X^n)$ and $A^n = D^2u(t^n, X^n)$ by evaluating the corresponding neural networks at X^n. Moreover, no neural networks are employed in [11] to approximate the functions $u(t^n, x)$ and $Du(t^n, x)$. In fact $Y^n = u(t^n, X^n)$ and $Z^n = Du(t^n, X^n)$

are computed by time marching using the Euler-Maruyama scheme applied to equations (13). Their loss function is then given by (9) which tries to match the terminal condition. The total set of parameters are consequently given by Y_0, Z_0, Γ_0, A_0, and the parameters of the $2(N-1)$ neural networks used to approximate the functions $D^2u(t^n, x)$ and $\mathcal{L}Du(t^n, x)$. This framework, being a descendant of [12, 13], also suffers from the drawbacks listed above. It should be emphasized that, although not pursued here, the framework proposed in the current work can be straightforwardly extended to solve the second-order backward stochastic differential equations (13). The key (see e.g., [3]) is to leverage the fundamental relationships (12).

4. Results

The proposed framework provides a universal treatment of coupled forward-backward stochastic differential equations of fundamentally different nature and their corresponding high-dimensional partial differential equations. This generality will be demonstrated by applying the algorithm to a wide range of canonical problems spanning a number of scientific domains including a 100-dimensional Black-Scholes-Barenblatt equation and a 100-dimensional Hamilton-Jacobi-Bellman equation. These examples are motivated by the pioneering works [11–13]. All data and codes used in this manuscript will be publicly available on GitHub at https://github.com/maziarraissi/FBSNNs.

4.1. Black-Scholes-Barenblatt Equation in 100D

Let us start with the following forward-backward stochastic differential equations

$$\begin{aligned} &dX_t = \sigma \mathrm{diag}(X_t) dW_t, \quad t \in [0, T], \\ &X_0 = \xi, \\ &dY_t = r(Y_t - Z_t' X_t)dt + \sigma Z_t' \mathrm{diag}(X_t) dW_t, \quad t \in [0, T), \\ &Y_T = g(X_T), \end{aligned} \tag{14}$$

where $T = 1$, $\sigma = 0.4$, $r = 0.05$, $\xi = (1, 0.5, 1, 0.5, \ldots, 1, 0.5) \in \mathbb{R}^{100}$, and $g(x) = \|x\|^2$. The above equations are related to the Black-Scholes-Barenblatt equation

$$u_t = -\frac{1}{2}\mathrm{Tr}[\sigma^2 \mathrm{diag}(X_t^2) D^2 u] + r(u - (Du)'x), \tag{15}$$

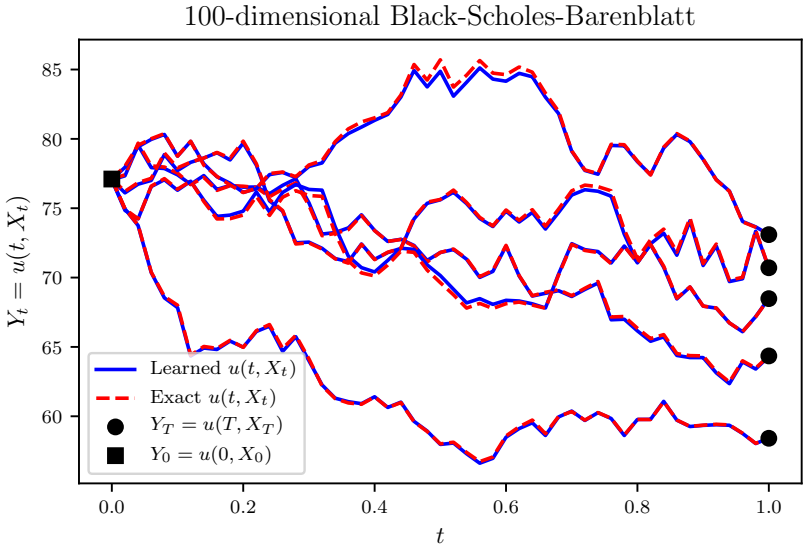

Figure 1: *Black-Scholes-Barenblatt Equation in 100D:* Evaluations of the learned solution $Y_t = u(t, X_t)$ at representative realizations of the underlying high-dimensional process X_t. It should be highlighted that the state of the art algorithms [11–13] can only approximate $Y_0 = u(0, X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$.

with terminal condition $u(T, x) = g(x)$. This equation admits the explicit solution

$$u(t, x) = \exp\left((r + \sigma^2)(T - t)\right) g(x), \tag{16}$$

which can be used to test the accuracy of the proposed algorithm. We approximate the unknown solution $u(t, x)$ by a 5-layer deep neural network with 256 neurons per hidden layer. Furthermore, we partition the time domain $[0, T]$ into $N = 50$ equally spaced intervals (see equations (5)). Upon minimizing the loss function (6), using the Adam optimizer [24] with mini-batches of size 100 (i.e., 100 realizations of the underlying Brownian motion), we obtain the results reported in figure 1. In this figure, we are evaluating the learned solution $Y_t = u(t, X_t)$ at representative realizations (not seen during training) of the underlying high-dimensional process X_t. Unlike the state of the art algorithms [11–13], which can only approximate $Y_0 = u(0, X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$, our algorithm is capable of approximating the entire solution function $u(t, x)$ in a single round of training as demonstrated in figure 1. Figures such as this one are absent in [11–13], by design.

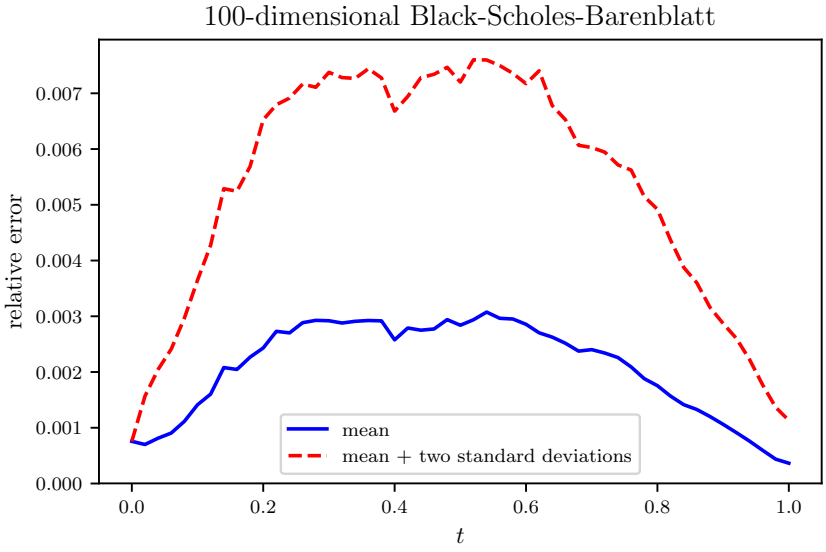

Figure 2: *Black-Scholes-Barenblatt Equation in 100D:* Mean and mean plus two standard deviations of the relative errors between model predictions and the exact solution computed based on 100 realizations of the underlying Brownian motion.

To further scrutinize the performance of our algorithm, in figure 2 we report the mean and mean plus two standard deviations of the relative errors between model predictions and the exact solution computed based on 100 independent realizations of the underlying Brownian motion. It is worth noting that in figure 1 we were plotting 5 representative examples of the 100 realizations used to generate figure 2. The results reported in figures 1 and 2 are obtained after 2×10^4, 3×10^4, 3×10^4, and 2×10^4 consecutive iterations of the Adam optimizer with learning rates of 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}, respectively. The total number of iterations is therefore given by 10^5. Every 10 iterations of the optimizer takes about 0.88 seconds on a single NVIDIA Titan X GPU card. In each iteration of the Adam optimizer we are using 100 different realizations of the underlying Brownian motion. Consequently, the total number of Brownian motion trajectories observed by the algorithm is given by 10^7. It is worth highlighting that the algorithm converges to the exact value $Y_0 = u(0, X_0)$ in the first few hundred iterations of the Adam optimizer. For instance after only 500 steps of training, the algorithms achieves an accuracy of around 2.3×10^{-3} in terms of relative

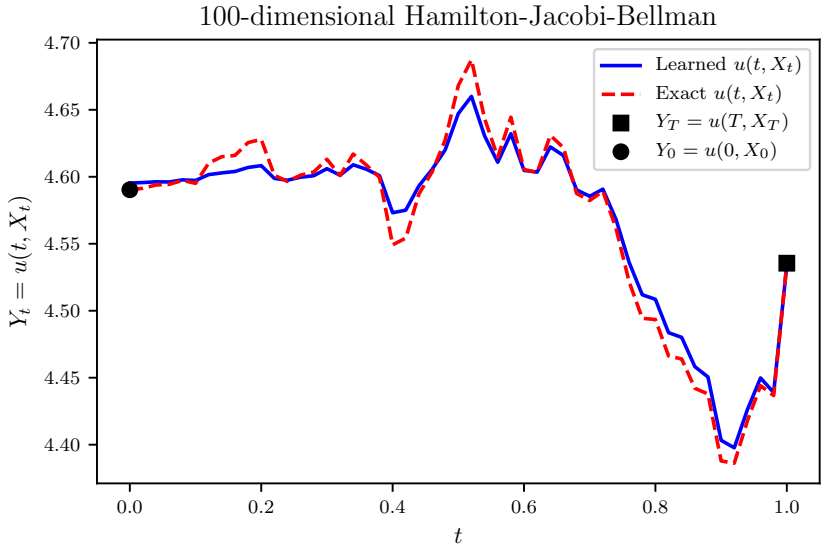

Figure 3: *Hamilton-Jacobi-Bellman Equation in 100D:* Evaluation of the learned solution $Y_t = u(t, X_t)$ at a representative realization of the underlying high-dimensional process X_t. It should be highlighted that the state of the art algorithms [11–13] can only approximate $Y_0 = u(0, X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$.

error. This is comparable to the results reported in [11–13], both in terms of accuracy and the speed of the algorithm. However, to obtain more accurate estimates for $Y_t = u(t, X_t)$ at later times $t > 0$ we need to train the algorithm using more iterations of the Adam optimizer.

4.2. Hamilton-Jacobi-Bellman Equation in 100D

Let us now consider the following forward-backward stochastic differential equations

$$\begin{aligned} &dX_t = \sigma dW_t, \quad t \in [0, T], \\ &X_0 = \xi, \\ &dY_t = \|Z_t\|^2 dt + \sigma Z_t' dW_t, \quad t \in [0, T), \\ &Y_T = g(X_T), \end{aligned} \tag{17}$$

where $T = 1$, $\sigma = \sqrt{2}$, $\xi = (0, 0, \ldots, 0) \in \mathbb{R}^{100}$, and $g(x) = \ln\left(0.5\left(1 + \|x\|^2\right)\right)$. The above equations are related to the Hamilton-Jacobi-Bellman equation

$$u_t = -\mathrm{Tr}[D^2 u] + \|Du\|^2, \tag{18}$$

with terminal condition $u(T,x) = g(x)$. This equation admits the explicit solution

$$u(t,x) = -\ln\left(\mathbb{E}\left[\exp\left(-g(x+\sqrt{2}W_{T-t})\right)\right]\right), \tag{19}$$

which can be used to test the accuracy of the proposed algorithm. In fact, due to the presence of the expectation operator $\mathbb{E}$ in equation (19), we can only approximately compute the exact solution. To be precise, we use 10^5 Monte-Carlo samples to approximate the exact solution (19) and use the result as ground truth. We represent the unknown solution $u(t,x)$ by a 5-layer deep neural network with 256 neurons per hidden layer. Furthermore, we partition the time domain $[0,T]$ into $N = 50$ equally spaced intervals (see equations (5)). Upon minimizing the loss function (6), using the Adam optimizer [24] with mini-batches of size 100 (i.e., 100 realizations of the underlying Brownian motion), we obtain the results reported in figure 3. In this figure, we are evaluating the learned solution $Y_t = u(t,X_t)$ at a representative realization (not seen during training) of the underlying high-dimensional process X_t. It is worth noting that computing the exact solution (19) to this problem is prohibitively costly due to the need for the aforementioned Monte-Carlo sampling strategy. That is why we are depicting only a single realization of the solution trajectories in figure 3. Unlike the state of the art algorithms [11–13], which can only approximate $Y_0 = u(0,X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$, our algorithm is capable of approximating the entire solution function $u(t,x)$ in a single round of training as demonstrated in figure 3.

To further investigate the performance of our algorithm, in figure 4 we report the relative error between model prediction and the exact solution computed for the same realization of the underlying Brownian motion as the one used in figure 3. The results reported in figures 3 and 4 are obtained after 2×10^4, 3×10^4, 3×10^4, and 2×10^4 consecutive iterations of the Adam optimizer with learning rates of 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}, respectively. The total number of iterations is therefore given by 10^5. Every 10 iterations of the optimizer takes about 0.79 seconds on a single NVIDIA Titan X GPU card. In each iteration of the Adam optimizer we are using 100 different realizations of the underlying Brownian motion. Consequently, the total number of Brownian motion trajectories observed by the algorithm is given by 10^7. It is worth highlighting that the algorithm converges to the exact value $Y_0 = u(0,X_0)$ in the first few hundred iterations of the Adam optimizer. For

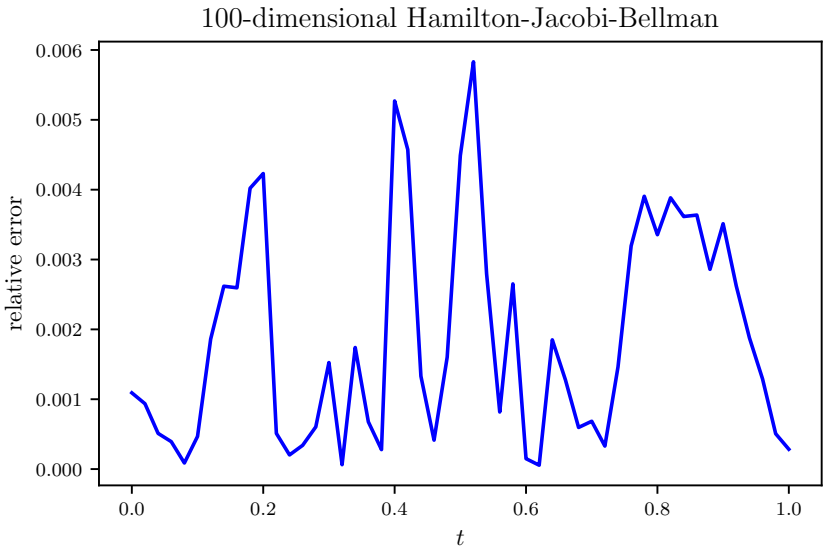

Figure 4: *Hamilton-Jacobi-Bellman Equation in 100D:* The relative error between model prediction and the exact solution computed based on a single realization of the underlying Brownian motion.

instance after only 100 steps of training, the algorithms achieves an accuracy of around 7.3×10^{-3} in terms of relative error. This is comparable to the results reported in [11–13], both in terms of accuracy and the speed of the algorithm. However, to obtain more accurate estimates for $Y_t = u(t, X_t)$ at later times $t > 0$ we need to train the algorithm using more iterations of the Adam optimizer.

4.3. Allen-Cahn Equation in 20D

Let us consider the following forward-backward stochastic differential equations

$$\begin{aligned} &dX_t = dW_t, \quad t \in [0, T], \\ &X_0 = \xi, \\ &dY_t = (-Y_t + Y_t^3)dt + Z_t' dW_t, \quad t \in [0, T), \\ &Y_T = g(X_T), \end{aligned} \tag{20}$$

where $T = 0.3$, $\xi = (0, 0, \ldots, 0) \in \mathbb{R}^{20}$, and $g(x) = (2 + 0.4\|x\|^2)^{-1}$. The above equations are related to the Allen-Cahn equation

$$u_t = -0.5\mathrm{Tr}[D^2 u] - u + u^3, \tag{21}$$

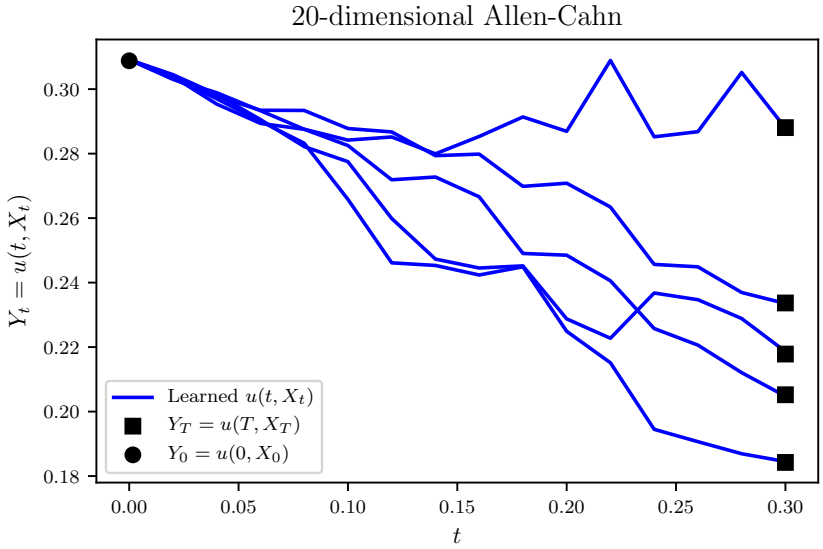

Figure 5: *Allen-Cahn Equation in 20D:* Evaluation of the learned solution $Y_t = u(t, X_t)$ at representative realizations of the underlying high-dimensional process X_t. It should be highlighted that the state of the art algorithms [11–13] can only approximate $Y_0 = u(0, X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$.

with terminal condition $u(T, x) = g(x)$. We represent the unknown solution $u(t, x)$ by a 5-layer deep neural network with 256 neurons per hidden layer. Furthermore, we partition the time domain $[0, T]$ into $N = 15$ equally spaced intervals (see equations (5)). Upon minimizing the loss function (6), using the Adam optimizer [24] with mini-batches of size 100 (i.e., 100 realizations of the underlying Brownian motion), we obtain the results reported in figure 5. In this figure, we are evaluating the learned solution $Y_t = u(t, X_t)$ at five representative realizations (not seen during training) of the underlying high-dimensional process X_t. Unlike the state of the art algorithms [11–13], which can only approximate $Y_0 = u(0, X_0) = 0.30879$ at time 0 and at the initial spatial point $X_0 = \xi$, our algorithm is capable of approximating the entire solution function $u(t, x)$ in a single round of training as demonstrated in figure 5.

5. Summary and Discussion

In this work, we put forth a deep learning approach for solving coupled forward-backward stochastic differential equations and their corresponding

high-dimensional partial differential equations. The resulting methodology showcases a series of promising results for a diverse collection of benchmark problems. As deep learning technology is continuing to grow rapidly both in terms of methodological, algorithmic, and infrastructural developments, we believe that this is a timely contribution that can benefit practitioners across a wide range of scientific domains. Specific applications that can readily enjoy these benefits include, but are not limited to, stochastic control, theoretical economics, and mathematical finance.

In terms of future work, one could straightforwardly extend the proposed framework in the current work to solve second-order backward stochastic differential equations (13). The key (see e.g., [3]) is to leverage the fundamental relationships (12) between second-order backward stochastic differential equations and fully-nonlinear second-order partial differential equations. Moreover, our method can be used to solve stochastic control problems, where in general, to obtain a candidate for an optimal control, one needs to solve a coupled forward-backward stochastic differential equation (1), where the backward components influence the dynamics of the forward component.

Acknowledgements

This work received support by the DARPA EQUiPS grant N66001-15-2-4055 and the AFOSR grant FA9550-17-1-0013.

References

[1] J.-M. Bismut, Conjugate convex functions in optimal stochastic control, Journal of Mathematical Analysis and Applications 44 (1973) 384–404.

[2] E. Pardoux, S. Peng, Adapted solution of a backward stochastic differential equation, Systems & Control Letters 14 (1990) 55–61.

[3] P. Cheridito, H. M. Soner, N. Touzi, N. Victoir, Second-order backward stochastic differential equations and fully nonlinear parabolic pdes, Communications on Pure and Applied Mathematics 60 (2007) 1081–1110.

[4] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations, arXiv preprint arXiv:1711.10566 (2017).

[5] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, arXiv preprint arXiv:1711.10561 (2017).

[6] M. Raissi, Deep hidden physics models: Deep learning of nonlinear partial differential equations, arXiv preprint arXiv:1801.06637 (2018).

[7] M. Raissi, P. Perdikaris, G. E. Karniadakis, Numerical gaussian processes for time-dependent and nonlinear partial differential equations, SIAM Journal on Scientific Computing 40 (2018) A172–A198.

[8] M. Raissi, G. E. Karniadakis, Hidden physics models: Machine learning of nonlinear partial differential equations, Journal of Computational Physics 357 (2018) 125–141.

[9] M. Raissi, P. Perdikaris, G. E. Karniadakis, Inferring solutions of differential equations using noisy multi-fidelity data, Journal of Computational Physics 335 (2017) 736–746.

[10] M. Raissi, P. Perdikaris, G. E. Karniadakis, Machine learning of linear differential equations using gaussian processes, Journal of Computational Physics 348 (2017) 683–693.

[11] C. Beck, A. Jentzen, et al., Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations, arXiv preprint arXiv:1709.05963 (2017).

[12] E. Weinan, J. Han, A. Jentzen, Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations, Communications in Mathematics and Statistics 5 (2017) 349–380.

[13] J. Han, A. Jentzen, et al., Overcoming the curse of dimensionality: Solving high-dimensional partial differential equations using deep learning, arXiv preprint arXiv:1707.02568 (2017).

[14] F. Antonelli, Backward-forward stochastic differential equations, The Annals of Applied Probability (1993) 777–793.

[15] J. Ma, P. Protter, J. Yong, Solving forward-backward stochastic differential equations explicitlya four step scheme, Probability theory and related fields 98 (1994) 339–359.

[16] E. Pardoux, S. Tang, Forward-backward stochastic differential equations and quasilinear parabolic pdes, Probability Theory and Related Fields 114 (1999) 123–150.

[17] F. Delarue, S. Menozzi, A forward-backward stochastic algorithm for quasi-linear pdes, The Annals of Applied Probability (2006) 140–184.

[18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, arXiv preprint arXiv:1502.05767 (2015).

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, arXiv preprint arXiv:1603.04467 (2016).

[20] M. Raissi, P. Perdikaris, G. E. Karniadakis, Multistep neural networks for data-driven discovery of nonlinear dynamical systems, arXiv preprint arXiv:1801.01236 (2018).

[21] M. Raissi, Parametric gaussian process regression for big data, arXiv preprint arXiv:1704.03144 (2017).

[22] P. Perdikaris, M. Raissi, A. Damianou, N. Lawrence, G. E. Karniadakis, Nonlinear information fusion algorithms for data-efficient multi-fidelity modelling, Proc. R. Soc. A 473 (2017) 20160751.

[23] M. Raissi, G. Karniadakis, Deep multi-fidelity Gaussian processes, arXiv preprint arXiv:1604.07484 (2016).

[24] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).