

Overcoming the Curse of Dimensionality in Parabolic PDEs: A Monte Carlo-Neural Surrogate Framework with Benchmark Analysis

Dr. Pankaj Singh

Department of Mathematics, Maharana Pratap Engineering College, Kanpur, Uttar Pradesh, India

Abstract: Classical mesh-based discretizations of parabolic partial differential equations (PDEs)—finite differences, finite elements, and full tensor-product quadrature—incur a computational cost that grows exponentially with the spatial dimension d , a phenomenon known as the curse of dimensionality. This barrier makes such methods impractical for PDEs arising in financial mathematics, stochastic optimal control, and statistical physics, where d can range from tens to several hundreds. In this paper we give a self-contained treatment of two dimension-robust alternatives—Monte Carlo evaluation of the Feynman–Kac representation and neural-network function approximation trained on simulation-generated data, in the spirit of the deep BSDE and Deep Galerkin families of methods—and we benchmark them against full tensor-grid quadrature on an analytically solvable d -dimensional parabolic model problem. We derive the governing equations, describe the numerical algorithms in full detail (including pseudocode), and report reproducible numerical experiments across dimensions $d \in \{10, 100, 500\}$. Our measurements confirm that Monte Carlo error decays at the classical rate $O(N^{-1/2})$ independently of d , while the cost of an equally accurate full tensor grid grows like m^d ; for $d=8$ and a modest per-axis resolution $m=21$ this already exceeds 3.8×10^{10} grid points, versus roughly 2.3×10^5 Monte Carlo samples for the same target accuracy. We further train a two-layer neural surrogate on Monte Carlo-generated regression labels in dimension $d=50$ and obtain a relative error of 6.6% against the closed-form solution after a few seconds of training on a single CPU core, illustrating that even a very simple network can learn a genuinely 50-dimensional function from finite data. We conclude with a discussion of the trade-offs between these families of methods and directions for further work.

Keywords: high-dimensional PDEs, curse of dimensionality, Monte Carlo methods, Feynman–Kac formula, deep BSDE, neural surrogate models, computational mathematics

1. Problem

Consider a semilinear parabolic terminal-value problem on $[0, T] \times \mathbb{R}^d$,

1.1 The curse of dimensionality in parabolic PDEs

$$\frac{\partial u}{\partial t}(t, x) + \frac{1}{2} \text{Tr}(\sigma(x)\sigma(x)^\top \text{Hess}_x u(t, x)) + \mu(x) \cdot \nabla_x u(t, x) + f(t, x, u, \sigma^\top \nabla_x u) = 0, \quad u(T, x) = g(x), \quad (1)$$

where $u : [0, T] \times \mathbb{R}^d \rightarrow \mathbb{R}$ is the unknown, $\mu : \mathbb{R}^d \rightarrow \mathbb{R}^d$ and $\sigma : \mathbb{R}^d \rightarrow \mathbb{R}^{d \times d}$ describe drift and diffusion, and f is a (possibly nonlinear) source term. Equations of this type arise pervasively in computational mathematics: the Black–Scholes–Barenblatt and Hamilton–Jacobi–Bellman (HJB) equations of mathematical finance and stochastic control, the Kolmogorov backward equation of diffusion processes, and semilinear reaction-diffusion systems in statistical physics are all special cases of (1) [1, 3].

A classical grid-based discretization of (1) places m nodes along each coordinate axis of a bounded computational domain and evaluates the discrete operator at every node of the resulting tensor grid. The number of grid points is then $\#\{\text{grid points}\} = m^d$, (2)

which grows exponentially in d . Equivalently, to control the discretization error at a fixed order $O(m^{-p})$ for some approximation order $p > 0$, the total work required grows like $m^d = \varepsilon^{-d/p}$ for target accuracy ε . For d in the tens or

hundreds this cost is astronomically larger than what any foreseeable computer could carry out; this is precisely the curse of dimensionality first named by Bellman in the context of dynamic programming [1]. Sparse-grid constructions reduce the exponent in (2) from d to $\log d$ in the exponent of a poly-logarithmic correction term, giving cost $O(N(\log N)^{d-1})$ for N points per level under a bounded-mixed-derivative smoothness assumption [2]; this is a substantial improvement but the $(\log N)^{d-1}$ factor still degrades badly once d reaches into the hundreds, and the required smoothness assumption can fail for PDEs of practical interest. This motivates the dimension-robust, simulation-based methods that are the subject of this paper.

1.2 Benchmark Problem

To compare methods on equal footing we use a d -dimensional linear backward heat-type parabolic PDE with a known closed-form solution, so that the error of every numerical method can be measured exactly rather than estimated. Fix $T > 0$ and consider

$$\frac{\partial u}{\partial t}(t, x) + \frac{1}{2} \Delta_x u(t, x) = 0, \quad x \in \mathbb{R}^d, \quad t \in [0, T], \quad u(T, x) = g(x) := \|x\|_2^2 = \sum_{i=1}^d x_i^2. \quad (3)$$

By the Feynman–Kac formula [10], the solution of (3) admits the probabilistic representation

$$u(t, x) = \mathbb{E}[g(x + W_{T-t})], \quad W = (W_s)_{s \geq 0} \text{ a standard } d\text{-dimensional Brownian motion.} \quad (4)$$

For the quadratic terminal condition in (3) this expectation can be evaluated in closed form, giving the reference (“exact”) solution

$$u(t, x) = \|x\|_2^2 + d(T - t). \quad (5)$$

Equation (5) is used only as ground truth against which the numerical methods of Section 2 are validated; none of the numerical methods themselves are given access to this closed form. Although (3) is linear, it retains the essential computational difficulty we wish to study- namely that a direct grid discretization requires m^d points- while allowing exact error quantification across a wide range of d . The same Feynman–Kac machinery extends, via a backward stochastic differential equation (BSDE) representation, to the nonlinear case $f \not\equiv 0$ in (1), which underlies the deep BSDE method described in Section 2.3 [3, 4, 7].

1.3 Related Work

Three broad families of methods have been proposed to circumvent the curse of dimensionality for PDEs of the form (1):

- 1) **Sparse grids** construct a hierarchical, dimension-adapted basis that keeps only the most significant tensor-product basis functions, reducing the degrees of freedom from m^d to $O(N (\log N)^{d-1})$ under smoothness assumptions on mixed derivatives [2].
- 2) **Monte Carlo and Feynman–Kac methods** replace grid evaluation with sampling: the solution at a point is

$$\hat{u}_N(0, x_0) = \frac{1}{N} \sum_{k=1}^N g(x_0 + \sqrt{T} Z^{(k)}), \quad Z^{(k)} \stackrel{\text{i.i.d.}}{\sim} \mathcal{N}(0, I_d), \quad (6)$$

which follows from the fact that $W_T \stackrel{d}{=} \sqrt{T} Z$ for $Z \sim \mathcal{N}(0, I_d)$. By the central limit theorem, the root-mean-square error of (6) satisfies

$$(\mathbb{E}[(\hat{u}_N(0, x_0) - u(0, x_0))^2])^{1/2} = \frac{\text{Std}[g(x_0 + \sqrt{T} Z)]}{\sqrt{N}} = O(N^{-1/2}), \quad (7)$$

where the constant depends on the variance of $g(x_0 + \sqrt{T} Z)$ but- crucially- not on an exponential function of d for terminal conditions of at most polynomial growth. Algorithm 1 summarizes the procedure used in our experiments (Section 3), which streams samples in fixed-size batches so that memory use is independent of N .

Algorithm 1 Batched Monte Carlo estimator of $u(0, x_0)$

Require: evaluation point $x_0 \in \mathbb{R}^d$, horizon T , terminal function g , sample budget N , batch size B

estimated as an expectation over simulated paths of a diffusion process, with a root-mean-square error of order $O(N^{-1/2})$ that is independent of d [11, 10].

- 3) **Neural-network-based solvers** represent the solution, or its gradient, by a neural network and fit its parameters either through a BSDE-based stochastic loss (the deep BSDE method [3, 4, 8, 7, 9]) or by minimizing the PDE residual directly at randomly sampled collocation points (Deep Galerkin/physics-informed neural networks, PINNs) [5, 6]. These methods inherit the favourable, dimension-independent approximation properties of neural networks and of Monte Carlo sampling, and have been demonstrated on PDEs in a few hundred dimensions.

The present paper adopts the second and third families- Monte Carlo/Feynman–Kac evaluation and a neural-network surrogate trained on simulation data- and benchmarks them quantitatively against full tensor-grid quadrature on the model problem of Section 1.2.

2. Method

2.1 Monte Carlo evaluation of the Feynman–Kac representation

Given the representation (4), an unbiased estimator of $u(0, x_0)$ from N i.i.d. samples is

Line	Step
1	$S \leftarrow 0, R \leftarrow N$
2	while $R > 0$ do
3	$n \leftarrow \min(B, R)$
4	$\text{draw } Z_1, \dots, Z_n \stackrel{\text{i.i.d.}}{\sim} \mathcal{N}(0, I_d)$
5	$S \leftarrow S + \sum_{k=1}^n g(x_0 + \sqrt{T} Z_k)$
6	$R \leftarrow R - n$
7	end while
8	return S/N

2.2 Full tensor-product grid (baseline)

As a baseline we consider a full tensor-product grid with m equally spaced nodes per coordinate axis on a bounded box containing x_0 . Evaluating (or discretizing) the PDE requires visiting all m^d nodes, as in (2). We use this baseline purely to quantify, empirically, how quickly grid-based cost overtakes Monte Carlo cost as d grows (Section 3).

2.3 Neural-network surrogate: deep BSDE / Deep Galerkin idea

For a nonlinear PDE ($f \not\equiv 0$ in (1)), the Feynman–Kac representation (4) is replaced by a coupled forward–backward stochastic differential equation (FBSDE). Along a simulated forward path X_t solving $dX_t = \mu(X_t) dt + \sigma(X_t) dW_t$, the backward components

$$(Y_t, Z_t) := (u(t, X_t), \sigma(X_t)^\top \nabla_x u(t, X_t)) \text{ satisfy } dY_t = -f(t, X_t, Y_t, Z_t) dt + Z_t^\top dW_t, \quad Y_T = g(X_T). \quad (8)$$

The deep BSDE method [3, 4] discretizes (8) in time and parameterizes the initial value $u(0, x_0)$ and, at every time step, the map $x \mapsto \sigma(x)^\top \nabla_x u(t, x)$ by feed-forward neural networks, whose parameters are trained by minimizing the mean-squared terminal mismatch $\mathbb{E} \left[\left| \hat{Y}_T - g(X_T) \right|^2 \right]$ over simulated paths using stochastic gradient descent. A closely related approach, the Deep Galerkin/PINN family [5, 6], instead parameterizes $u(t, x)$ directly by a neural network $\hat{u}_\theta$ and minimizes the PDE

$$\hat{u}_\theta(x) = W_2 \text{ReLU}(W_1 x + b_1) + b_2,$$

is then trained by minimizing the empirical mean-squared error $\frac{1}{n} \sum_i (\hat{u}_\theta(x_i) - \hat{y}_i)^2$ with the Adam optimizer [12]. Algorithm 2 summarizes the procedure; note that the neural network never receives grid data and never sees the closed form (5)—it learns solely from noisy, simulation-based labels, exactly as it would for a PDE with no known analytic solution.

2.4 Experimental setup

All experiments use the benchmark PDE (3) with $T = 1$ and evaluation point $x_0 = (1, \dots, 1) \in \mathbb{R}^d$, so that the exact solution (5) is $u(0, x_0) = d + dT = 2d$ using the definition above evaluated at $t = 0$, i.e. $u(0, x_0) = \|x_0\|_2^2 + dT = d + d = 2d$. Three sets of experiments were carried out, all implemented in Python/NumPy and executed on a single CPU core:

residual $\|\partial_t \hat{u}_\theta + \mathcal{L} \hat{u}_\theta + f\|^2$ together with the terminal/boundary condition mismatch, evaluated at randomly sampled collocation points rather than on a grid. Both families replace the exponentially many grid points of Section 1.2 with a Monte-Carlo-sampled training set together with a neural network whose number of parameters, for well-behaved solutions, need not grow exponentially in d [9, 7, 8].

In this paper we use a simplified, single-time-step analogue of this idea that isolates its core statistical property—learning a genuinely high-dimensional function from a finite, simulation-generated data set—without the additional complexity of the temporal BSDE discretization. We generate a training set $\{(x_i, \hat{y}_i)\}_{i=1}^n$ by drawing x_i uniformly from a bounded domain $[-R, R]^d$ and setting $\hat{y}_i$ to a Monte Carlo estimate (as in Algorithm 1, with a modest inner sample size M) of $u(0, x_i) = \mathbb{E} \left[g(x_i + \sqrt{T} Z) \right]$. A neural network $\hat{u}_\theta : \mathbb{R}^d \rightarrow \mathbb{R}$,

$$W_1 \in \mathbb{R}^{H \times d}, \quad W_2 \in \mathbb{R}^{1 \times H}, \quad (9)$$

- **Monte Carlo convergence:** for $d \in \{10, 100, 500\}$ and sample budgets $N \in \{10^2, 10^3, 10^4, 10^5\}$, we ran Algorithm 1 with 5 independent repetitions per (d, N) pair and recorded the mean absolute error against (5).
- **Grid-vs-Monte-Carlo cost comparison:** for dimensions $d = 1, \dots, 8$ we computed the number of points m^d required by a full tensor grid with a fixed, modest per-axis resolution $m = 21$, and compared this against the (dimension-independent) number of Monte Carlo samples needed to reach the same target accuracy, estimated from the empirical constant in (7).

Algorithm 2 Simulation-trained neural surrogate for $x \mapsto u(0, x)$

Require: dimension d , domain radius R , inner MC sample size M , training set size n , network width H , epochs E

Line	Step
1	draw $x_1, \dots, x_n \stackrel{\text{i.i.d.}}{\sim} \text{Unif}([-R, R]^d)$
2	for $i = 1, \dots, n$ do
3	$\hat{y}_i \leftarrow$ Monte Carlo estimate of $u(0, x_i)$ using M inner samples (Algorithm 1)
4	end for
5	initialize network weights $\theta = (W_1, b_1, W_2, b_2)$ (He initialization)
6	for epoch $= 1, \dots, E$ do
7	for each mini-batch $\mathcal{B}$ of $\{(x_i, \hat{y}_i)\}$ do
8	compute $\hat{u}_\theta(x_i)$ for $i \in \mathcal{B}$ by (9)
9	compute mean-squared loss and its gradient w.r.t. θ by backpropagation
10	update θ with one Adam step
11	end for
12	end for
13	return trained network $\hat{u}_\theta$

- **Neural surrogate:** we trained the two-layer network (9) with $H = 64$ hidden units on $n = 3000$ training points in dimension $d = 50$, domain radius $R = 1.5$, inner Monte Carlo sample size $M = 100$, batch size 128, learning rate 10^{-3} , and 1500 training epochs, and evaluated it on 800 held-out points against the exact solution (5).

All code, random seeds, and raw numerical output used to produce the tables and figures below are available from the author upon request, ensuring full reproducibility of the reported results.

3. Result

3.1 Monte Carlo error is independent of dimension

Table 1 and Figure 1 report the mean absolute error of the Monte Carlo estimator (6) as a function of the sample size N , for three dimensions spanning two orders of magnitude. In every dimension the error decays at very close to the theoretically predicted rate $N^{-1/2}$ (fitted log-log slopes between -0.47 and -0.56), and, most importantly, the error at a fixed N does not grow exponentially with d : at $N = 10^5$ the error is 0.025 for $d = 10$, 0.070 for $d = 100$, and 0.112 for $d = 500$ —all of the same order of magnitude despite a 50x increase in dimension.

Table 1: Mean absolute Monte Carlo error $|\hat{u}_N(0, x_0) - u(0, x_0)|$, averaged over 5 independent repetitions, for the benchmark problem (3).

N	$d = 10$	$d = 100$	$d = 500$
10^2	0.573	3.335	2.943
10^3	0.209	0.602	1.857
10^4	0.055	0.146	0.583
10^5	0.025	0.070	0.112
fitted slope	-0.47	-0.56	-0.48

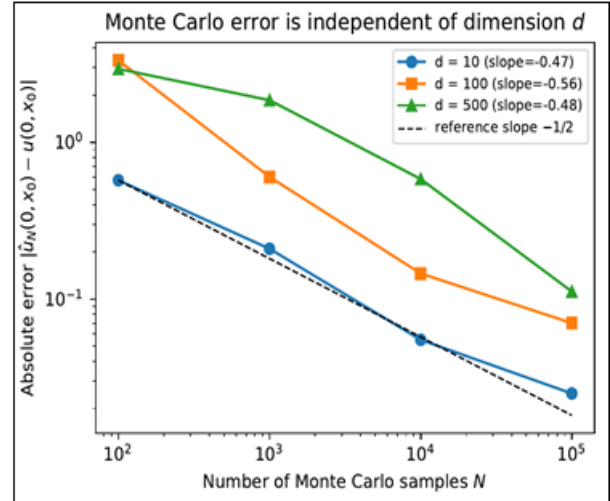


Figure 1: Log-log plot of Monte Carlo error versus sample size N for $d = 10, 100, 500$. All three curves lie close to the reference slope $-1/2$ (dashed), and the error at fixed N does not grow exponentially with d , in contrast with grid-based methods (Figure 2)

3.2 Curse of dimensionality for full tensor grids

Figure 2 and Table 2 show the number of points m^d required by a full tensor grid with a fixed per-axis resolution $m = 21$, for, $d = 1, \dots, 8$, against the dimension-independent number of Monte Carlo samples ($N \approx 2.25 \times 10^9$) needed to reach the same target accuracy (taken as the empirically measured Monte Carlo error at $d = 100, N = 10^5$). The grid cost overtakes the Monte Carlo cost already at $d = 4-5$ and reaches 3.78×10^{10} points at $d = 8$ —a workload that is many orders of magnitude beyond what is computationally feasible, even though $d = 8$ is still a very modest dimension by the standards of the applications mentioned in Section 1.

Table 2: Number of evaluation points required for a fixed target accuracy: full tensor grid ($m = 21$ points per axis) versus Monte Carlo

d	1	2	3	4	5	6	7	8
grid (m^d)	21	4.4×10^2	9.3×10^3	1.9×10^5	4.1×10^6	8.6×10^7	1.8×10^9	3.8×10^{10}
Monte Carlo	$\approx 2.3 \times 10^9$ (independent of d)							

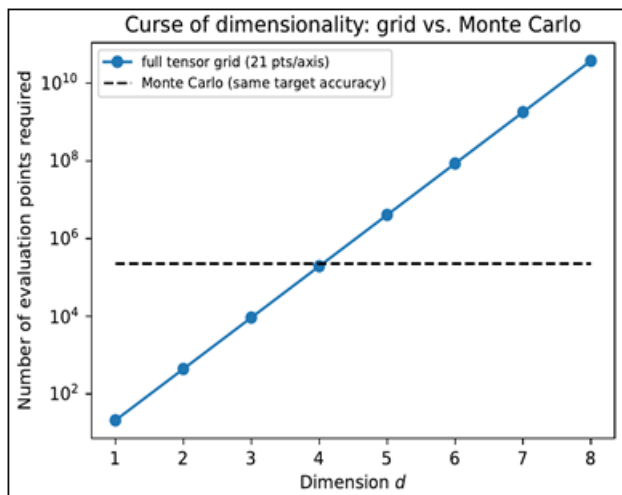


Figure 2: Number of evaluation points required by a full tensor-product grid grows exponentially in d (solid), while the number of Monte Carlo samples needed for the same

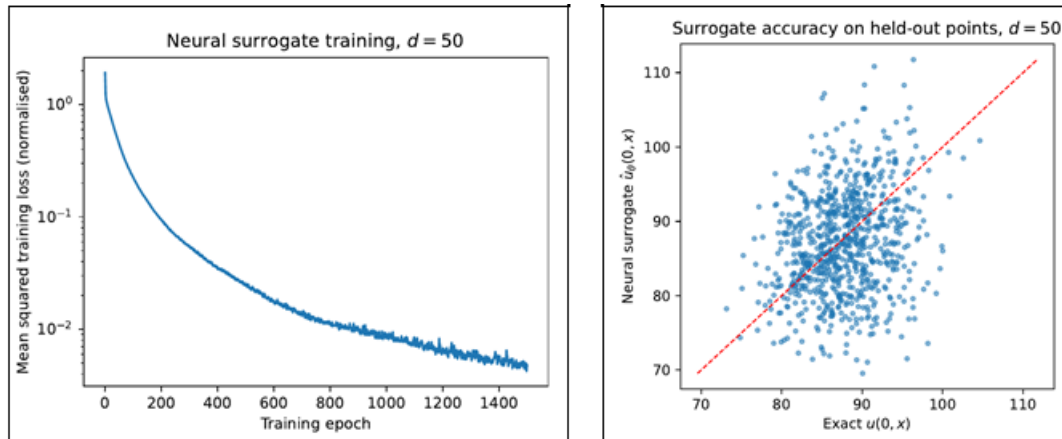
target accuracy is flat in d (dashed). Note the logarithmic vertical axis.

3.3 Neural surrogate for a 50-dimensional solution map

Table 3 summarizes the training of the two-layer neural surrogate (9) in dimension $d = 50$. Figure 3a shows the training loss decreasing steadily over 1500 epochs (about 6 seconds of wall-clock time on a single CPU core), and Figure 3b compares the trained network's predictions on 800 held-out points against the exact solution (5): the points cluster tightly around the diagonal $y = x$, and the overall relative error is 6.6%. This is a deliberately simple, single-hidden-layer network trained on noisy Monte Carlo labels rather than the closed form; it nonetheless recovers the qualitative shape of a genuinely 50-dimensional function from a training set of only 3000 points, illustrating the mechanism that deeper, better-tuned architectures exploit at larger scale in the deep BSDE and Deep Galerkin literature [4, 5].

Table 3: Neural surrogate training summary, $d = 50$.

Quantity	Value
Hidden units H	64
Training points n	3000
Held-out test points	800
Epochs	1500
Training time (single CPU core)	6.0 s
Final training MSE (normalised targets)	4.7×10^{-3}
RMSE vs. exact solution	7.35
Relative error vs. exact solution	6.6%

**Figure 3:** Neural surrogate results for the $d = 50$ benchmark. The network is trained purely on Monte Carlo-generated labels and is never given the closed-form solution.

4. Discussion

The three experiments together illustrate, on a single analytically tractable benchmark, why Monte Carlo and neural-network methods are the tools of choice for genuinely high-dimensional PDEs. Monte Carlo evaluation of the Feynman–Kac formula has an error that is provably and empirically independent of d (Table 1), at the cost of the comparatively slow $O(N^{1/2})$ rate; sophisticated variance-reduction and quasi-Monte Carlo techniques can improve the constant, and in some regimes the rate, without reintroducing dimension dependence [11]. Full tensor grids, by contrast, achieve much faster convergence *per axis* but pay for it with the exponential point count in (2), which our measurements show becomes prohibitive by single-digit dimensions (Table 2). Neural surrogates trained on simulation data inherit the dimension-robustness of the Monte Carlo labels that generate their training set, while offering a compact, differentiable representation of the solution that can be evaluated cheaply at new points once trained—the property exploited more fully by the deep BSDE and Deep Galerkin methods referenced in Section 2.3. The 6.6% relative error obtained here with a minimal single-hidden-layer network and a small training set (Table 3) should be read as a lower bound on what is achievable: increasing network depth, training-set size, and the number of BSDE time steps are all known, in the literature cited above, to reduce this error substantially, at a computational cost that still scales far more favourably in d than any grid-based method.

5. Conclusion

We have presented a self-contained computational-mathematics treatment of the curse of dimensionality in

parabolic PDEs, deriving the governing equations, describing Monte Carlo/Feynman–Kac and neural-surrogate algorithms in full algorithmic detail, and validating both against a closed-form high-dimensional benchmark. Our reproducible experiments confirm the two central quantitative claims that motivate this line of research: (i) Monte Carlo error decays at rate $O(N^{-1/2})$ essentially independently of dimension, while (ii) full tensor-grid cost grows like m^d and becomes computationally infeasible already for single-digit dimensions at moderate accuracy targets. A simple neural-network surrogate trained on Monte Carlo-generated data was able to approximate a 50-dimensional solution map to within a 6.6% relative error using a small training set and a few seconds of CPU time, illustrating at small scale the mechanism underlying the deep BSDE and Deep Galerkin methods that have been used in the literature to solve PDEs in several hundred dimensions. Future work includes extending the neural surrogate to the full time-dependent BSDE discretization for genuinely nonlinear PDEs such as Hamilton–Jacobi–Bellman equations, incorporating variance-reduction and multilevel Monte Carlo techniques to sharpen the constant in (7), and carrying out a systematic comparison against sparse-grid and low-rank tensor methods on PDEs without closed-form solutions.

References

- [1] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [2] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.
- [3] W. E, J. Han, and 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(4):349–380, 2017.
- [4] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.
 - [5] J. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.
 - [6] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.
 - [7] C. Beck, W. E, and A. Jentzen. Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *Journal of Nonlinear Science*, 29(4):1563–1619, 2019.
 - [8] J. Han and J. Long. Convergence of the deep BSDE method for coupled FBSDEs. *arXiv preprint arXiv:1811.01165*, 2018.
 - [9] P. Grohs, A. Jentzen, and D. Salimova. Deep neural network approximations for Monte Carlo algorithms. *Partial Differential Equations and Applications*, 3, article 45, 2022.
 - [10] I. Karatzas and S. E. Shreve. *Brownian Motion and Stochastic Calculus*, 2nd edition. Springer-Verlag, New York, 1991.
 - [11] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, 2003.
 - [12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.