Table of Contents

Deep Neural Networks for Solving High Dimensional PDEs in Quantitative Finance

Preface

Using Physics-Informed Neural Networks (PINN) to solve partial differential equations (PDEs) has become a hot and cutting-edge topic in numerous disciplines such as mathematics, physics, medicine, aerospace, and many other academic fields. This stems from the inherent high difficulty in solving PDEs, the broad knowledge system involved, and their unavoidable importance in practical applications.

In paper [3], Jian Liang points out that American option pricing based on the least square Monte Carlo method has been widely adopted in the industry. Inspired by this, I embarked on the exploration of the option pricing problem. In that study, the least squares-based deep neural network proposed by the authors aligns with the conceptual framework of the Forward-Backward Stochastic Neural Networks (FBSNN) presented by Professor Panos Parpas.

However, after implementing Monte Carlo simulation for 100-dimensional American option pricing, I observed a noteworthy phenomenon: the Monte Carlo simulation for 100 dimensions took less than 3 minutes, while the 1-dimensional Black-Scholes option pricing model based on FBSNN required over half an hour even when running on Google Colab with GPU acceleration. This prompted me to ponder: Is there a way to accelerate the solution? Are there other more efficient deep neural network-based approaches?

For this purpose, I further studied the paper by Boussange, V. in [4], which presents a high-dimensional PDE solving method based on the Julia language. I translated the algorithm from that paper into a Python implementation and conducted a comparative analysis with the FBSNN method. Therefore, this project focuses on the option pricing problem, emphasizing three methods: **Monte Carlo simulation**, **FBSNN**, and **DeepBSDE**, and based on this, carried out various comparisons and experimental attempts.

Given the limited time and my own capabilities, choosing this topic inevitably involves many areas worthy of deeper exploration—a foreseen outcome even before the project started. However, I believe pricing is the most core and fundamental aspect of CQF and quantitative finance, while PDE solving is one of the most challenging problems within it. I am very grateful to the professor for providing us the opportunity to experiment and for the guidance in this area. I hope that in the future, if I can work in quantitative finance, I will be able to explore this topic more deeply and comprehensively.

1. introduction

The numerical solution of partial differential equations (PDEs) plays a crucial role in fields such as financial engineering, physical modeling, and quantitative finance, particularly in core issues like option pricing, risk management, and derivative valuation. Traditional numerical methods, such as the finite difference method and the finite element method, perform excellently in low-dimensional cases. However, as the problem dimensionality increases, these methods face severe "curse of dimensionality" issues—computational complexity and memory requirements grow exponentially with dimensionality, making the solution of high-dimensional PDEs almost infeasible in practical applications. This "dimensionality explosion" phenomenon has long constrained the effective solution of high-dimensional financial problems.

In recent years, breakthroughs in deep learning technology have provided new paradigms for PDE solving. Physics-Informed Neural Networks (PINN) embed physical laws into the neural network structure, achieving mesh-free solving of PDEs and significantly alleviating dimensionality issues. Especially in the financial field, hybrid methods based on stochastic differential equations and deep learning, such as the Forward-Backward Stochastic Neural Network (FBSNN) and the Deep Backward Stochastic Differential Equation (DeepBSDE) method, combine Monte Carlo simulation and neural network approximation to provide feasible solutions for high-dimensional option pricing problems.

This paper focuses on the solution of the high-dimensional Black-Scholes-Barenblatt equation, systematically comparing two mainstream methods: the FBSNN framework and the DeepBSDE method. We deeply analyze the theoretical foundations, algorithmic implementations, and numerical performance of each method. In particular, this paper explores how to overcome the curse of dimensionality through neural network parameterization and the advantages of different network structures (such as NAIS-Net) in ensuring numerical stability.

The paper is structured as follows: Chapter 2 introduces the necessary theoretical framework, including the mathematical foundations of stochastic differential equations and the Feynman-Kac formula; Chapter 3 provides a detailed analysis of the FBSNN method and its implementation; Chapter 4 discusses the DeepBSDE method and its extensions; Chapter 5 presents a comprehensive comparative analysis; finally, Chapter 6 summarizes the full text and outlines future research directions. Additionally, Chapter 7 is separate, introducing the least squares Monte Carlo simulation for solving the optimal stopping time problem in American option pricing.

2 Theoretical Framework

2.1 Forward Stochastic Differential Equations

A forward stochastic differential equation (abbreviated as forward SDE or SDE) is formally expressed as:

$$\mathrm{d}X_t = \mu(t, X_t)\mathrm{d}t + \sigma(t, X_t)\mathrm{d}W_t, \quad t \in (0, T]; \quad X_0 = x_0$$

where $\mu : [0, T] \times \mathbb{R}^n \to \mathbb{R}^n$ and $\sigma : [0, T] \times \mathbb{R}^n \to \mathbb{R}^{n\times d}$ are measurable functions, called the **drift coefficient** and the **diffusion coefficient**, respectively. Equation (2.1) can be interpreted as the stochastic integral equation

$$X_t = x_0 + \int_0^t \mu(s, X_s)\mathrm{d}s + \int_0^t \sigma(s, X_s)\mathrm{d}W_s.$$

2.2 Backward stochastic differential equations

The BSDEs considered in this thesis are of the form

$$\begin{aligned} dY_t &= f(t, Y_t, Z_t)dt - Z_t dW_t, \quad t \in [0, T) \\ Y_T &= g(X_T) \end{aligned} \tag{2.3}$$

where $f : \Omega \times [0, T] \times \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$ is measurable. The integral form of (2.3) reads

$$Y_t = g(X_T) + \int_t^T f(s, Y_s, Z_s)ds - \int_t^T Z_s dW_s, \quad t \in [0, T]. \tag{2.4}$$

The BSDE (2.3) is characterized by the **generator** f and the **terminal condition** $g(X_T)$. Therefore, the pair $(f, g(X_T))$ is referred to as the parameters of the BSDE. The **parameters** are said to be **standard parameters** if:

- The terminal value $g(X_T) \in L_T^2(\mathbb{R})$,
- The stochastic process $f(\cdot, 0, 0) \in H_T^2(\mathbb{R})$,
- The function f is uniformly Lipschitz continuous in the second and third variables i.e., there exists a constant $L_f > 0$ such that for $t \in [0, T]$ and for $y_1, y_2 \in \mathbb{R}$ and $z_1, z_2 \in \mathbb{R}^d$, it holds almost surely that

$$|f(t, y_1, z_1) - f(t, y_2, z_2)| \le L_f \left(|y_1 - y_2| + |z_1 - z_2|\right).$$

2.3 Coupled FBSDE

FBSDE is a coupled system, consisting of a forward stochastic differential equation (SDE) that describes the evolution of a system's state, and a backward stochastic differential equation (BSDE) that starts from a terminal condition and evolves backward in time to describe a certain "value" or "cost."

The solution (Y_t, Z_t) of the backward equation directly influences the dynamics of the forward equation (e.g., the drift term depends on Y_t, Z_t).

$$\begin{cases} dX_t = \mu(t, X_t, Y_t, Z_t)dt + \sigma(t, X_t, Y_t, Z_t)dW_t, \\ -dY_t = f(t, X_t, Y_t, Z_t)dt - Z_t dW_t, \\ X_0 = x_0, \quad Y_T = g(X_T). \end{cases} \tag{2.5}$$

• The "coupling" of the system is reflected in:

- The drift or diffusion term of the forward SDE may depend on the solution (Y_t, Z_t) of the backward process.
- The generator f and the terminal condition of the backward BSDE also depend on the solution X_t of the forward process.

The forward and backward equations are interdependent and cannot be solved separately; they require joint iterative computation (e.g., through fixed-point iteration or synchronous approximation using neural networks).

2.4 Nonlinear Feynman-Kac Formula

The classical Feynman-Kac formula establishes the connection between linear PDEs and stochastic differential equations (SDEs). For nonlinear PDEs such as the HJB equation, we need its generalization—the nonlinear Feynman-Kac formula.

First, consider a general nonlinear parabolic partial differential equation, for example, the **Hamilton-Jacobi-Bellman equation** from stochastic optimal control theory:

$$\begin{cases} \frac{\partial u}{\partial t}(t,x) + \mathcal{L}u(t,x) + f(t,x,u(t,x),\sigma^\top \nabla u(t,x)) = 0, & (t,x) \in [0,T) \times \mathbb{R}^n \\ u(T,x) = g(x), & x \in \mathbb{R}^n \end{cases}$$

where $\mathcal{L}$ is the differential operator:

$$\mathcal{L}u = \mu(t,x) \cdot \nabla u + \frac{1}{2}\mathrm{Tr}\big(\sigma\sigma^T(t,x)\nabla^2 u\big)$$

- $u(t,x)$ is the solution of the PDE (e.g., representing the **value function** in control problems)
- f is the generator, which determines the nonlinear characteristics of the PDE
- $g(x)$ is the terminal condition

This equation describes the condition that the optimal value function (the best expected cost or reward achievable from the current system state to the future) must satisfy. Its general form includes a minimization operation over the control variable u.

This implies that we reinterpret it as an optimization problem: find an optimal control policy (e.g., how to trade assets) that minimizes a total cost (e.g., risk or tracking error). The solution to the PDE is the value function of this optimization problem.

Next is the derivation process of transforming the PDE into a forward-backward SDE system via the Feynman-Kac formula.

The **key assumption** is that the PDE has a sufficiently smooth solution $u(t,x) \in C^{1,2}$.

We "track" the solution $u(t,x)$ of this PDE along the path formed by the solution X_t of the forward SDE. Define a new process:

$$\tilde{Y_t} := u(t, X_t)$$

The goal is to find the stochastic differential of $\tilde{Y}_t$. At this point, we use **Itô's Lemma**.

Itô's Lemma states that for a smooth function $u(t, X_t)$, its stochastic differential is as follows:

$$d\tilde{Y}_t = du(t,X_t) = \left[\frac{\partial u}{\partial t}(t,X_t) + \mathcal{L}u(t,X_t)\right]dt + [\nabla u(t,X_t)]^T \sigma(t,X_t) dW_t$$

where $\mathcal{L}$ is the differential operator defined earlier.

Because $u(t,x)$ is the solution of the PDE, it satisfies:

$$\frac{\partial u}{\partial t}(t,x) + \mathcal{L}u(t,x) = -f(t,x,u(t,x),\sigma^T \nabla u(t,x))$$

Substitute the PDE into the Itô formula, replacing the term in square brackets:

$$du(t,X_t) = \left[-f(t,X_t,u(t,X_t),\sigma^T\nabla u(t,X_t))\right]dt + [\nabla u(t,X_t)]^T\sigma(t,X_t)dW_t \quad (1)$$

$$n = -f(t,X_t,u(t,X_t),\sigma^T\nabla u(t,X_t))dt + \left[\sigma^T\nabla u(t,X_t)\right]^T dW_t \quad (2)$$

Comparison with the BSDE

We now obtain the dynamics equation for $\tilde{Y}_t = u(t,X_t)$:

$$du(t,X_t) = -f(\ldots)dt + \left[\sigma^T\nabla u(t,X_t)\right]^T dW_t$$

Compare this with the dynamics equation of the **backward BSDE**:

$$-dY_t = f(t,X_t,Y_t,Z_t)dt - Z_t dW_t$$

(Note: $-dY_t$ can be rewritten as $dY_t = -f(\cdots)dt + Z_t dW_t$)

By comparing these two equations, we can make the following **key identifications**:

1. **Identification for Y_t**: If we let
$$Y_t = \tilde{Y}_t = u(t,X_t)$$
then the drift terms (dt terms) of the two equations match.
2. **Identification for Z_t**: To match the diffusion terms (dW_t terms), we must let
$$Z_t = \sigma^T(t,X_t)\nabla u(t,X_t)$$
3. **Terminal Condition**: At the terminal time T, we have
$$Y_T = u(T,X_T) = g(X_T)$$
which exactly matches the terminal condition of the BSDE.

The nonlinear Feynman-Kac formula states that the problem of solving a nonlinear parabolic partial differential equation (such as the Hamilton-Jacobi-Bellman equation) can be transformed into the equivalent problem of solving a forward-backward stochastic differential equation (FBSDE) system.

The equivalent FBSDE system corresponding to the PDE is as follows:

$$\begin{cases} dX_t = \mu(t,X_t,Y_t,Z_t)dt + \sigma(t,X_t)dW_t & (\text{forward SDE, describing state evolution}) \\ -dY_t = f(t,X_t,Y_t,Z_t)dt - Z_t dW_t & (\text{backward BSDE, describing value evolution}) \\ Y_T = g(X_T) & (\text{terminal condition, corresponding to the PDE's termin} \end{cases}$$

Deep Correspondence Between PDE and FBSDE Solutions

If the solution to the PDE is $u(t,x)$, then the associated FBSDE solution (Y_t, Z_t) satisfies the following core relationship:

$$Y_t = u(t,X_t)$$
$$Z_t = \sigma^\top(t,X_t)\nabla u(t,X_t)$$

This means:

- **Physical meaning of Y_t**: The backward process Y_t is exactly the value of the PDE solution u along the path of the forward process X_t.
- **Physical meaning of Z_t**: The backward process Z_t is the value of the gradient ∇u of the PDE solution u along the path of X_t, then linearly transformed by the diffusion coefficient σ.

- **Origin of the generator** f: The nonlinear term of the PDE itself (for example, in the HJB equation) completely determines the generator function f in the backward equation.

This theory (nonlinear Feynman-Kac formula) successfully "probabilizes" deterministic PDE problems. Therefore, solving PDEs is transformed into solving FBSDE systems.

Now, we cannot help but mention the following landmark theorem in this field.

Theorem 1 (Pardoux and Peng 1996). Let $Y : [0,T] \times \Omega \to \mathbb{R}$ and $Z : [0,T] \times \Omega \to \mathbb{R}^n$ be adapted stochastic processes with continuous sample paths which satisfy that for all $t \in [0,T]$ it holds that

$$Y_t = g(X_T) + \int_t^T f(s, X_s, Y_s, Z_s)ds - \int_t^T Z_s \cdot dW_s \qquad (2.4)$$

under some regularity assumptions [13]. Then, the solution to the forward-backward stochastic differential equation (FBSDE) is related to the nonlinear partial differential equation (PDE) given by

$$\frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\mathrm{Tr}\big[\sigma(t,x)\sigma(t,x)^\top \nabla^2 u(t,x)\big] + \nabla u(t,x)^\top \mu(t,x) + f(t,x,u(t,x),\sigma(t,x)^\top \nabla u(t,x)) = 0$$

with terminal condition

$$u(T,x) = g(x),$$

in the sense that for all $t \in [0,T]$ it holds that

$$Y_t = u(t, X_t), \quad Z_t = \sigma(t, X_t)^\top \nabla u(t, X_t).$$

Where:

- $u(t,x)$ with $t \in \mathbb{R}_+$ and $x \in X \subset \mathbb{R}^d$ is the unknown function, $\nabla u(t,x)$ is its gradient with respect to x, and $\nabla^2 u(t,x)$ is its Hessian matrix.
- $\sigma(t,x)$ is a known $d \times d$ matrix-valued function.
- Tr denotes the trace of a matrix and $^\top$ denotes the transpose.
- f is a known nonlinear function.
- g is a known terminal/boundary condition.

This extension allows us to establish a connection between the classical solution of a semilinear Partial Differential Equation (PDE) and a process-based solution to the corresponding Forward-Backward Stochastic Differential Equation (FBSDE).

2.5 The Black-Scholes-Barenblatt Equation

The Black-Scholes-Barenblatt equation can be viewed as a stochastic optimal control problem:

$$\frac{\partial u}{\partial t}+\sup_{\sigma\in[\underline{\sigma},\bar{\sigma}]}\left[\frac{1}{2}\sigma^2x^2\frac{\partial^2u}{\partial x^2}\right]+r\left(x\frac{\partial u}{\partial x}-u\right)=0$$

- The volatility σ is not assumed to be a fixed value, but rather a varying "adversary" within a range. When pricing, we consider the **worst-case scenario** (i.e., the supremum $\sup$) to ensure that the pricing is robust under any possible volatility.
- The **sup term**: Among all possible values of volatility σ, we look for the one that maximizes $\frac{1}{2}\sigma^2x^2\frac{\partial^2u}{\partial x^2}$. This is essentially an optimization problem of a linear function in σ^2.

Financial Economic Interpretation:

- Investors do not know the true volatility parameter.
- **Ambiguity Aversion**: Investors consider the worst-case scenario, leading to robust pricing.

Equation Properties and Theoretical Foundation: The Black-Scholes-Barenblatt equation is a **second-order nonlinear parabolic PDE**. The backward iterative method, which is the dual form of the Hamilton-Jacobi-Bellman equation, is perfectly suitable for solving the BSB equation. This also forms the theoretical basis for using the BSB equation to train neural networks.

Relationship with the Standard Black-Scholes Equation: When the volatility is certain (i.e., $\underline{\sigma}=\bar{\sigma}=\sigma$), the BSB equation reduces to the standard Black-Scholes equation:

$$\partial_tu+r\left(x\partial_xu-u\right)+\frac{1}{2}\sigma^2x^2\partial_{xx}u=0$$

Under this condition, there exist analytical solutions for certain terminal conditions, such as:

- **European call option**: $\varphi(x)=\max(x-K,0)$
- **European put option**: $\varphi(x)=\max(K-x,0)$

These analytical solutions will serve as **benchmark tests** for verifying the accuracy of deep neural networks.

This image clearly illustrates how the BSB equation originates from a robust pricing stochastic control problem, is expressed in the form of a nonlinear PDE, and degenerates into the classical model under specific conditions. It provides the theoretical foundation and validation benchmarks for deep learning-based numerical solution methods.

2.6 Formulation of the High-Dimensional PDE Problem

I will formulate this high-dimensional PDE problem using standard stochastic differential equation notation.

A d-dimensional stochastic process $X_t = (X_t^1, X_t^2, \ldots, X_t^d)$ satisfies the following stochastic differential equation:

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW_t$$

where:

- $W_t = (W_t^1, W_t^2, \ldots, W_t^d)$ is a d-dimensional standard Brownian motion
- Initial condition: $X_0 = (1.0, 0.5, 1.0, 0.5, \ldots)$ (repeated 50 times)

We make the following coefficient function settings:

Drift coefficient (zero vector):

$$\mu(X, t) = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{d\times 1}$$

Diffusion coefficient (diagonal matrix):

$$\sigma(X, t) = \begin{pmatrix} \sigma X^1 & 0 & \cdots & 0 \\ 0 & \sigma X^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma X^d \end{pmatrix}_{d\times d}$$

where $\sigma = 0.4$

Thus, the function $u(t, X)$ we need to solve satisfies the following **PDE**:

Main equation:

$$\frac{\partial u}{\partial t} + \mathcal{L}u + f\left(X, u, \sigma^T \nabla u\right) = 0$$

Terminal condition:

$$u(T, X) = g(X) = \sum_{i=1}^{d} (X^i)^2$$

Next, decompose using differential operators:

Infinitesimal generator $\mathcal{L}$:

$$\mathcal{L}u = \mu \cdot \nabla u + \frac{1}{2}\mathrm{Tr}\left[\sigma\sigma^T \nabla^2 u\right]$$

Since $\mu = 0$, the above simplifies to:

$$\mathcal{L}u = \frac{1}{2}\mathrm{Tr}\left[\sigma\sigma^T\nabla^2 u\right]$$

Trace term of the diffusion matrix:

$$\sigma\sigma^T = \begin{pmatrix} (\sigma X^1)^2 & 0 & \cdots & 0 \\ 0 & (\sigma X^2)^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & (\sigma X^d)^2 \end{pmatrix}$$

Therefore:

$$\mathrm{Tr}\left[\sigma\sigma^T\nabla^2 u\right] = \sum_{i=1}^{d}(\sigma X^i)^2\frac{\partial^2 u}{\partial (X^i)^2}$$

Nonlinear term:

$$f(X, u, \sigma^T\nabla u) = r\left(u - \sum_{i=1}^{d} X^i \cdot \left[\sigma^T\nabla u\right]_i\right)$$

where:

- $r = 0.05$ is the risk-free interest rate
- $\left[\sigma^T\nabla u\right]_i = \sigma X^i \frac{\partial u}{\partial X^i}$ is the i-th component of the gradient

Combining all parts, we obtain the complete PDE:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sum_{i=1}^{d}(\sigma X^i)^2\frac{\partial^2 u}{\partial (X^i)^2} + r\left(u - \sigma\sum_{i=1}^{d}(X^i)^2\frac{\partial u}{\partial X^i}\right) = 0$$

Boundary condition:

$$u(T, X) = \sum_{i=1}^{d}(X^i)^2, \quad \forall X \in \mathbb{R}^d$$

Financial interpretation of the corresponding stochastic process:

Each component X_t^i satisfies an independent geometric Brownian motion:

$$dX_t^i = \sigma X_t^i dW_t^i, \quad i = 1, \ldots, d$$

This is a zero-drift multi-asset geometric Brownian motion model, commonly used in finance to simulate high-dimensional asset price dynamics.

This PDE problem describes an option pricing problem under a 100-dimensional stochastic process, where the terminal payoff is the sum of squares of asset prices. In finance, this corresponds to a type of basket option or square option.

3. FBSNN

Based on the theoretical foundation established previously, the FBSNN framework is constructed with the following core components:

1. Forward Process: Simulate the state trajectory X_t via the Euler-Maruyama method.
2. Neural Network Approximation: Use a neural network to approximate the solution function $u(t,x)$ and its gradient.
3. Loss Function: Construct multi-constraint losses based on FBSDE discretization.
4. Optimization Algorithm: Train the neural network parameters through backpropagation.

3.1 Time Discretization and Solution Parameterization

We have discretized the time interval $[0,T]$. Based on the Euler discretization (9) for the forward SDE and the discretization (10) for the FBSDE, we have:

$$\tilde{Y}_{n+1} = \tilde{Y}_n + \varphi(t_n, \tilde{X}_n, \tilde{Y}_n, \tilde{Z}_n)\Delta t_n + \tilde{Z}_n^T \sigma(t_n, \tilde{X}_n)\Delta W_n, \qquad (3.1)$$

where $\Delta t_n = t_{n+1} - t_n$, $\Delta W_n = W_{t_{n+1}} - W_{t_n}$, and $\tilde{Y}_n \approx Y_{t_n}$, $\tilde{Z}_n \approx Z_{t_n}$.

According to the **nonlinear Feynman-Kac formula** (Theorem 2.1), the solution (Y_t, Z_t) of the FBSDE can be represented by a function $u(t,x)$ and its gradient:

$$Y_t = u(t, X_t), \quad Z_t = \sigma(t, X_t)^\top \nabla_x u(t, X_t).$$

Therefore, at the discrete time points, we can make the following substitutions:

$$\tilde{Y}_n = u(t_n, \tilde{X}_n), \quad \tilde{Z}_n = \sigma(t_n, \tilde{X}_n)^\top \nabla_x u(t_n, \tilde{X}_n).$$

Substituting these relations into the discrete equation (3.1), we obtain the core iterative equation for the solution function u:

$$\begin{aligned} u(t_{n+1}, \tilde{X}_{n+1}) = u(t_n, \tilde{X}_n) &+ \varphi\left(t_n, \tilde{X}_n, u(t_n, \tilde{X}_n), \sigma(t_n, \tilde{X}_n)^\top \nabla_x u(t_n, \tilde{X}_n)\right) \Delta t_n \\ &+ \left[\nabla_x u(t_n, \tilde{X}_n)^o p \sigma(t_n, \tilde{X}_n)\right] \sigma(t_n, \tilde{X}_n)^o p \Delta W_n, \qquad (3.2) \end{aligned}$$

where $\Delta W_n \sim \mathcal{N}(0, \Delta t_n\, I)$.

The ultimate goal of the algorithm is to compute the initial value, i.e.:

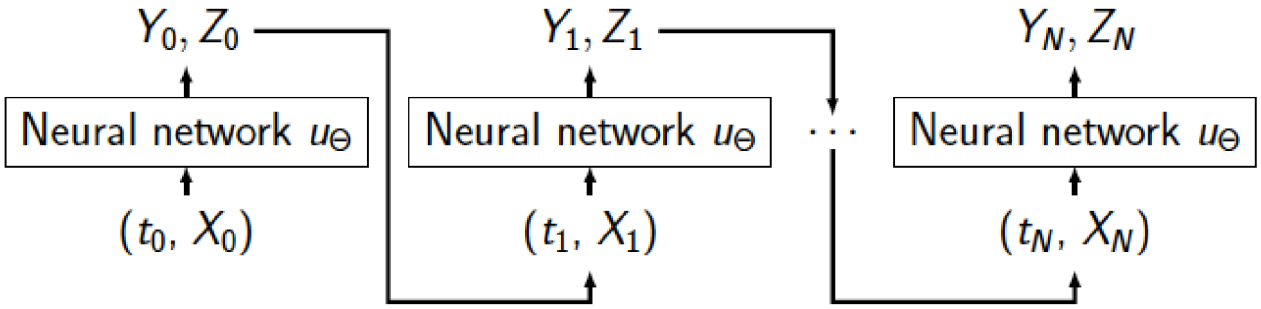
$$\tilde{Y}_0 \approx Y_0 = u(0, X_0).$$

Parameterized Implementation: In numerical computation, we use a neural network $u(t,x;\Theta)$ to approximate the theoretical solution function $u(t,x)$. Its gradient $\nabla_x u(t,x;\Theta)$ can be efficiently computed via automatic differentiation (e.g., backpropagation). Thus, the problem is transformed into finding the optimal parameters Θ such that the iterative relation (3.2) is satisfied as closely as possible.

3.2 Neural Network Approximation

The core idea is to use a neural network to parameterize the unknown gradient term Z_t (i.e., $\sigma^\top \nabla u$).

- As illustrated, at each time step t_n, the neural network f receives the current state X_n as input and outputs an approximation of Z_n.



- Θ: shared parameters through time,
- Z_n: computed using auto-differentiation.

3.2.1 Loss Function

The loss function for FBSNNs consists of three components:

1. **Path Loss**: Ensures the consistency of the discretized FBSDE.

$$\mathcal{L}_{\text{path}} = \sum_{m=1}^{M} \sum_{n=0}^{N-1} \left| Y_{n+1}^m - Y_n^m - \varphi \Delta t_n - (Z_n^m)^T \sigma \Delta W_n^m \right|^2$$

2. **Terminal Loss**: Matches the terminal condition.

$$\mathcal{L}_{\text{terminal}} = \sum_{m=1}^{M} \left| Y_N^m - g\left(X_N^m\right) \right|^2$$

3. **Terminal Gradient Constraint**: Ensures the terminal gradient matches.

$$\mathcal{L}_{\text{terminal-gradient}} = \sum_{m=1}^{M} \left\| Z_N^m - \nabla g\left(X_N^m\right) \right\|^2$$

Total Loss:

$$\mathcal{L} = \mathcal{L}_{\text{path}} + \mathcal{L}_{\text{terminal}} + \mathcal{L}_{\text{terminal-gradient}}$$

The data flow of the FBSNN is as follows:

```
Initial condition Xi → Generate time grid t and Brownian motion W → Forwa
rd propagation:
  ├─ Time step n=0: t0, X0=Xi
  ├─ Network u0, Z0 = net_u(t0, X0)
  ├─ Time step n=1:
  │   ├─ X1 = X0 + μ·Δt + σ·ΔW0
  │   ├─ Network u1, Z1 = net_u(t1, X1)
  │   └─ Calculate Y1_tilde = Y0 + φ·Δt + Z0·σ·ΔW0
  └─ ... Repeat until terminal time T

Loss calculation:
  ├─ Path loss: Σ||Yn+1 - Yn+1_tilde||²
  ├─ Terminal loss: ||YT - g(XT)||²
  └─ Terminal gradient constraint: ||ZT - ∇g(XT)||²
```

3.2.2 Training and Optimization

The training of a neural network is an optimization task. Its goal is to find the set of parameters θ that minimizes a loss function, $\mathcal{J}(\theta)$. The form of this function is problem-specific.

The minimization is typically achieved using gradient-based algorithms. A standard approach is gradient descent, or one of its variants, where the parameters are updated at each training iteration k according to the rule:

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta \mathcal{J}(\theta_k),$$

where $\alpha > 0$ is the learning rate.

3.3 NAIS-Net

3.3.1 Architecture Introduction

In the architecture of FBSNN, there are two types of network structures that can be chosen: one is the MLP, and the other is NAIS-Net. Since the former is the most basic structure for building neural networks, it will not be elaborated here. The following mainly introduces NAIS-Net.

NAIS-Net, whose full name is **"Non-Autonomous Input-Output Stable Network"**, was proposed by Marco Ciccone [8] in 2018 in a paper. It is a deep neural network structure with stable residual connections. It adopts a modular design and supports different depth configurations of 4-6 layers. The basic structure of the network follows the pattern:

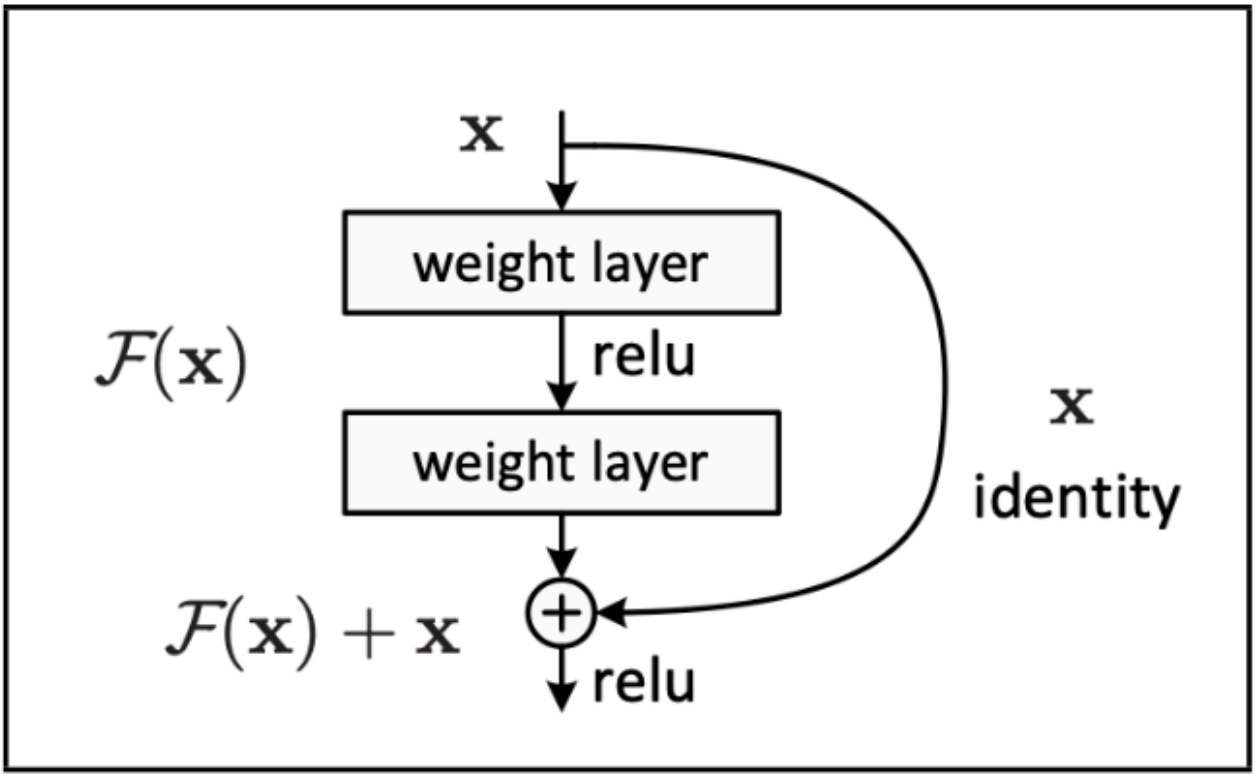
```
Input → Linear Layer 1 → Activation → [Stable Block 1] → ... → [Stable Bl
ock n] → Output Layer
```

The number of stable blocks is determined by the network depth.

The overall structure borrows the idea of building a network framework from the **"Residual Block"** in **ResNet networks**:

$$x(k+1) = x(k) + F(x(k), \theta(k))$$

1. $x(k)$: It is the input of the k -th layer (also the output of the （k−1）-th layer).
2. $F(x(k), \theta(k))$: Represents a series of **nonlinear transformations** (usually including convolution, batch normalization, and activation functions, with parameters $\theta(k)$) performed on the input $x(k)$.
3. $x(k+1)$: It is the （k+1）-th layer (i.e., the output of this residual block). It is not directly computed by F, but rather by **adding the input $x(k)$ to the result of the transformation F**.


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

This design **helps prevent the vanishing gradient problem**: In very deep networks, the gradient in backpropagation is multiplied across multiple layers, easily becoming extremely small (vanishing) or large (exploding). By introducing the identity skip connection $x(k)$, the gradient now has a direct propagation path (a "highway") that does not pass through the nonlinear transformation F, enabling effective training of deep networks.

The transformation of the **"stable block"** that also reaches the original input can be written as:

$$\text{out} = \text{project}(x) + \text{linear\_input}(u)$$

where u is the original input of the network. This design not only keeps the original information from being lost with depth but also allows the network to learn interactions between the input and deep features.

Here, the stability also relies on the key `project` method, which mainly plays the role of **spectral normalization processing** of the weight matrix of the linear layer.

Mathematical Principle

Let the weight matrix be $W \in \mathbb{R}^{m \times n}$, and the standard linear transformation be $y = Wx + b$. To achieve a stable transformation, we construct a new weight matrix:

$$A = R^T R + \epsilon I$$

where R is the normalized weight matrix, and ϵ is a small positive number to ensure positive definiteness.

The normalization process is as follows:

1. Calculate the spectral norm $||R^T R||$ of $R^T R$.
2. If $||R^T R|| > \delta$, then scale it: $R^T R \leftarrow \sqrt{\delta} \cdot \frac{R^T R}{\sqrt{||R^T R||}}$.
3. Construct $A = R^T R + \epsilon I$.

The final transformation used is:

$$y_{\text{project}} = -Ax + b$$

The negative sign design can be regarded as a form of regularization, which helps stabilize the optimization process.

Significance of Spectral Norm Control

Controlling the spectral norm of $R^T R$ is equivalent to limiting the range of singular values of the weight matrix. Let σ_i be the singular values of R, then the eigenvalues of $R^T R$ are σ_i^2. By constraining $||R^T R|| = \max \sigma_i^2 \leq \delta$, we ensure that all $\sigma_i \leq \sqrt{\delta}$. This brings two benefits:

- Prevents exponential growth of activation values (forward propagation stability).
- Prevents exponential changes in gradients (backward propagation stability).

The introduction of ϵ guarantees that the minimum eigenvalue of A is at least ϵ, avoiding matrix singularity.

3.3.2 Stability Analysis

Considering the discrete dynamical system perspective, the forward propagation of the network is regarded as a dynamical system:

$$x_{k+1} = f_k(x_k)$$

where f_k is the transformation of the k-th layer.

Theorem 2 (Network Stability): If each f_k is Lipschitz continuous, and the Lipschitz constant $L_k \leq 1$, then the entire network is stable.

Proof Idea: The change in the network output is constrained by the product of the changes in each layer. Through spectral norm control, we ensure $L_k \leq \sqrt{\delta} + \epsilon$. By choosing δ and ϵ such that $\sqrt{\delta} + \epsilon \leq 1$, stability can be guaranteed.

3.3.3 Gradient Flow Analysis

Considering backpropagation, the gradient of the loss function L with respect to the input of the l-th layer is:

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_{l+1}} \frac{\partial x_{l+1}}{\partial x_l}$$

In a standard network, $\frac{\partial x_{l+1}}{\partial x_l} = W^T$. In NAIS-Net:

$$\frac{\partial x_{l+1}}{\partial x_l} = -A^T + I$$

Since the eigenvalues of A are in the range $[\epsilon, \epsilon + \delta]$, the eigenvalues of the gradient propagation matrix are in the range $[-\epsilon - \delta + 1, -\epsilon + 1]$, which can avoid gradient explosion or vanishing.

4. DeepBSDE

4.1 Mathematical Background

This implementation is based on the nonlinear Feynman-Kac formula, which transforms the parabolic PDE:

$$\begin{aligned} \partial u/\partial t + \mathcal{L}u + f(t, x, u, \sigma^\top \nabla u) = 0 \\ u(T, x) = g(x) \end{aligned}$$

into a coupled forward-backward SDE system:

Forward SDE (State Evolution):

$$dX = \mu(t, X)dt + \sigma(t, X)dW$$

Backward BSDE (Solution Evolution):

$$du = -f(t, X, u, \sigma^\top \nabla u)dt + (\sigma^\top \nabla u)' dW$$

$$u(T, X_T) = g(X_T)$$

4.2 Network Structure Design

DeepBSDE uses several separate neural networks to approximate the solution and gradients

4.2.1 U0Network

U0Network is responsible for estimating the initial value of the PDE solution $u(0, x)$, with the following structure:

- Input layer: dimension consistent with PDE dimension d
- Hidden layers: multiple fully connected layers using ReLU activation function
- Output layer: single value output, representing the estimate of $u(0, x)$

This network is only used for initial condition estimation and does not participate in the time evolution process

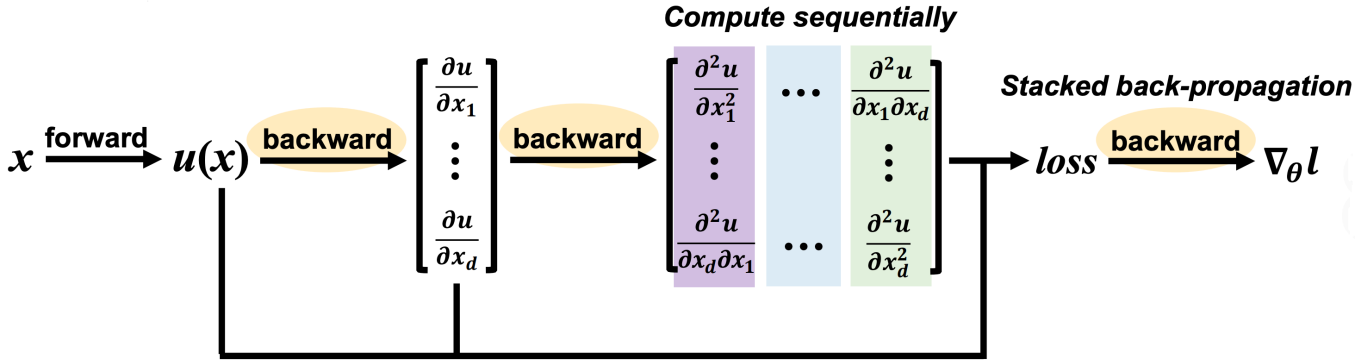
4.2.2 SigmaTGradUNetwork

Each time step is equipped with an independent SigmaTGradUNetwork for estimating the gradient term $\sigma^\top \nabla u$:

- Input layer: dimension d, receives current state X_t
- Hidden layers: deep network structure to ensure sufficient expressive capability
- Output layer: dimension d, outputs gradient estimate $\sigma^\top \nabla u$

This design allows independent optimization of gradient estimates at each time step, improving model flexibility.

However, overall, the structure of deepBSDE is consistent with the previous FBSNN network construction approach, and both can be represented by the following diagram.



4.2.3 Loss Function

The loss function of the DeepBSDE method only includes terminal condition matching error and does not include path loss:

The loss function is defined as:

$$\mathcal{L}(\theta) = \frac{1}{M}\sum_{i=1}^{M}|Y_T^i - g(X_T^i)|^2$$

where M is the number of Monte Carlo paths, Y_T^i is the terminal value estimate of the BSDE, and $g(X_T^i)$ is the terminal condition of the PDE.

Overall, **the dataflow** can be represented as follows:

```
Initial condition x0 → Backward time stepping:
  ├─ Terminal time T:
  │    └─ uT = g(XT)
  ├─ Time step T-1:
  │    ├─ Generate Brownian motion increment ΔW
  │    ├─ σT-1·∇uT-1 = SigmaTGradUNetwork(XT-1)
  │    └─ uT-1 = uT - φ·Δt + σT-1·∇uT-1·ΔW
  └─ ... Repeat to initial time 0

Loss calculation:
  └─ Only terminal loss: ||uT - g(XT)||²
```

4.3 Confidence Interval of Estimation Results

To ensure the reliability of the solution, DeepBSDE not only provides point estimates but also constructs confidence intervals for the solution through upper and lower bound estimates, providing important guarantees for the reliability of high-dimensional PDE solving.

4.3.1 Upper Bound Calculation

4.3.1.1 Duality Theory

The core of upper bound calculation in the DeepBSDE framework lies in the use of **duality theory** and **stochastic control methods**, constructing a dual problem to obtain an upper bound estimate for the solution of the original BSDE.

The previous DeepBSDE uses a backward stochastic differential equation system:

- **Forward SDE**: Describes state variable evolution: $dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t$
- **Backward BSDE**: Describes solution process evolution: $dY_t = -f(t, X_t, Y_t, Z_t)dt + Z_t dW_t$
- **Terminal condition**: $Y_T = g(X_T)$ The goal is to find a tight upper bound estimate for the initial value Y_0.

According to duality theory, there exists a dual process $\hat{Z}_t$ such that the solution to the original problem satisfies:

$$Y_0 \leq \mathbb{E}\left[g(X_T) + \int_0^T \hat{Z}_t^T dW_t\right]$$

To obtain the tightest upper bound, the optimization problem is transformed into:

$$Y_0 \leq \sup_{\hat{Z} \in \mathcal{A}} \mathbb{E}\left[g(X_T) + \int_0^T \hat{Z}_t^T dW_t\right]$$

where $\mathcal{A}$ is the set of admissible control processes.

4.3.1.2 Algorithm Implementation

The compute_upper_bound function adopts a four-stage processing flow: First step: Copy the neural network parameters (u0 and sigma_grad_u) of the solver, creating independent upper bound optimization networks u0_high and sigma_grad_u_high

Second step: Generate forward SDE trajectories Generate multiple SDE trajectories using the Euler-Maruyama discretization method:

$$X_{t_{k+1}}^i = X_{t_k}^i + \mu(t_k, X_{t_k}^i)\Delta t + \sigma(t_k, X_{t_k}^i)\Delta W_{t_k}^i$$

Third step: Start from the terminal condition and calculate the upper bound estimate backwards in time Starting from the terminal condition $U_T = g(X_T)$, iterate backward in time:

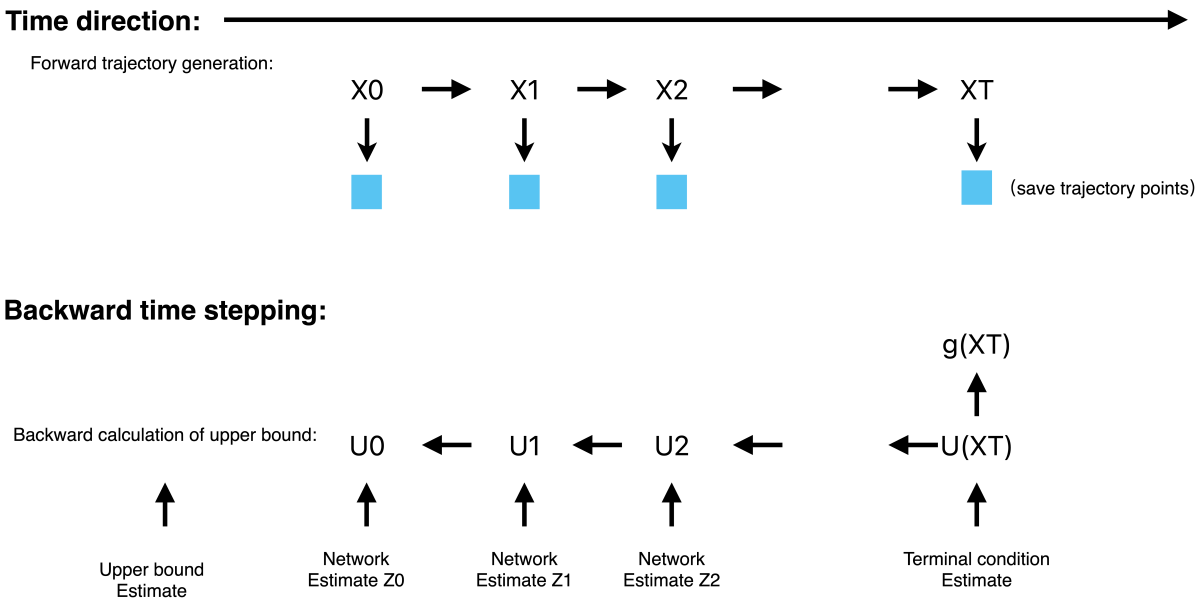
$$U_{t_k}^i = U_{t_{k+1}}^i + f(t_k, X_{t_k}^i, U_{t_{k+1}}^i, \hat{Z}_{t_k}^i)\Delta t - \hat{Z}_{t_k}^i \cdot \Delta W_{t_k}^i$$

where the gradient term $\hat{Z}_{t_k}^i = \sigma(t_k, X_{t_k}^i)^T \nabla u(t_k, X_{t_k}^i)$ is estimated by a dedicated neural network.

Loss function Set the goal to maximize the average upper bound:

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} U_0^i$$

Note: Since optimizers typically minimize the loss function, the actual optimization target is to minimize the negative upper bound: $\min -\mathcal{L}$. The overall flow chart can be represented as follows:



Upper bound optimization:

Maximize E[U0] → Train network parameters → Obtain tighter upper bound

4.3.2 Lower Bound Calculation

Unlike upper bound estimation based on stochastic control, lower bound estimation in the DeepBSDE framework transforms the nonlinear term $f(t, x, u, \sigma^\top \nabla u)$ in the PDE into a dual form through Legendre transform. The core of this process is using **Legendre transform** and **Fenchel duality theory** to convert nonlinear optimization problems into manageable convex optimization problems.

4.3.2.1 Mathematical Foundation: Legendre Transform and Duality Theory

In mathematics, Legendre transform is used to map a function from the original variable space to the dual variable space. For a convex function $h(u)$, its Legendre transform is defined as:

$$h^*(a) = \sup_u \{a \cdot u - h(u)\}$$

According to Fenchel duality theorem, the original function and dual function satisfy:

$$u \leq a^{-1}(h(u) + h^*(a))$$

This indicates that the original variable u can be bounded by the dual variable a and the control function h.

In the context of the Black-Scholes-Barenblatt equation, the generator function (i.e., the nonlinear term) is:

$$f(t, X_t, Y_t, Z_t) = -rY_t + rX_t^\top Z_t$$

where $Z_t = \sigma^\top \nabla u$. The goal is to find the lower bound of Y_0, i.e., $Y_0 \geq \text{lower bound}$.

4.3.2.2 Transformation Process of Nonlinear Term f

Step 1: Reformulate the Nonlinear Term

- At each time step, calculate the dot product $X_t^\top Z_t$ (i.e., $X_t^\top (\sigma^\top \nabla u)$), which reflects the interaction between the state variable and the gradient term.
- Generate the f matrix: $f_{\text{matrix}} = r \cdot (u_{\text{domain}} - X_t^\top Z_t)$, where u_{domain} is the predefined range of u. This essentially parameterizes the nonlinear term f as a function of u.

Step 2: Apply Legendre Transform

- For each control variable $a \in A$ (where A is the control variable domain), calculate the Legendre transform value:

  $$F(a) = \sup_{u \in u_{\text{domain}}} \{a \cdot u - f_{\text{matrix}}(u)\}$$

  Here, $f_{\text{matrix}}(u)$ is the value of f for a given u. The purpose of the transform is to find the u that maximizes $a \cdot u - f(u)$, thereby relating the dual variable a to the original function.

Step 3: Select Optimal Control Variable

- Select the optimal control a_{optimal} by maximizing the objective function:

  $$a_{\text{optimal}} = \arg\max_{a \in A} \{a \cdot u_t - F(a)\}$$

  This is equivalent to finding a tight lower bound in Fenchel duality. The optimal control a_{optimal} maximizes the combination of the original variable u_t and the dual function $F(a)$, thus ensuring the

tightness of the lower bound estimate.

4.3.2.3 Construction of Dual Form and Lower Bound Calculation

Through Legendre transform, the nonlinear term f is transformed into the function $F(a)$ of the dual variable a. Then, a dual process (I_t, Q_t) is constructed:

- I_t accumulates the effect of the optimal control: $I_{t+\Delta t} = I_t + a_{\text{optimal}} \cdot \Delta t$.
- Q_t accumulates the weighted sum of Legendre transform values: $Q_{t+\Delta t} = Q_t + e^{I_{t+\Delta t}} \cdot F(a_{\text{optimal}})$.

Finally, the lower bound estimate is:

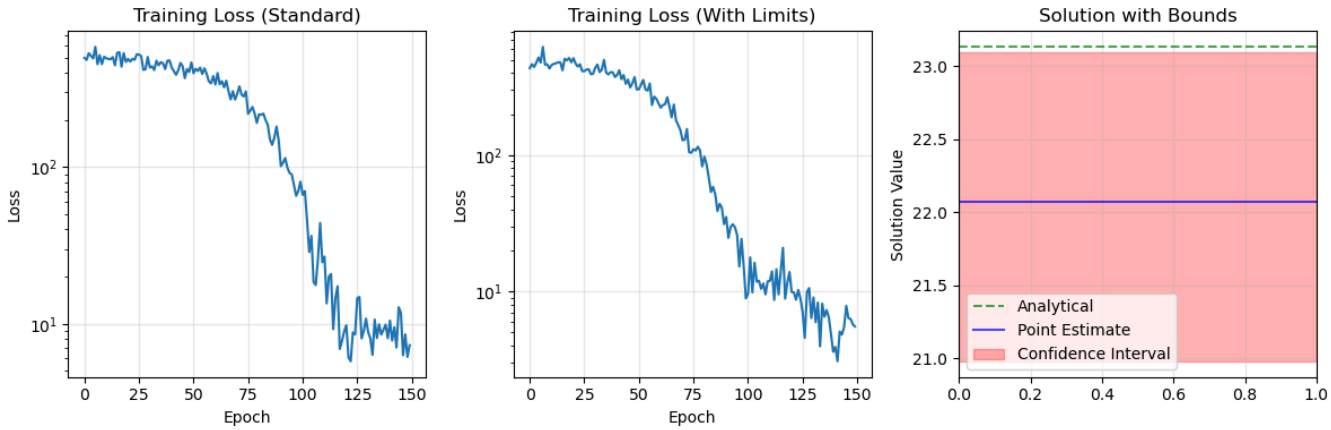
$$Y_0 \geq \mathbb{E}\left[e^{I_T} g(X_T) - Q_T\right]$$

In the discrete case, it is approximated by Monte Carlo averaging:

$$\text{lower bound} = \frac{1}{M} \sum_{i=1}^{M} \left[e^{I_T^i} g(X_T^i) - Q_T^i\right]$$

4.4 Standard DeepBSDE vs DeepBSDE with Bounds

Through testing comparison, we obtain the following results.



From the figure, it can be observed that both the standard DeepBSDE method and the DeepBSDE with Bounds (hereafter replaced by DeepBSDE Legendre, as it is named in the code implementation). show a decreasing training loss with iterations. The loss curve of the Legendre transform dual method is more stable in the later stages (final loss 5.651 vs. 10.543 for the standard method), and the upper bound optimization process exhibits steady convergence (after 10 iterations, the upper bound stabilizes at around ~23.165).

However, the Legendre transform dual method requires additional computation of upper and lower bounds after the standard DeepBSDE training is completed, which moderately increases the computational overhead.

5 Comparison between Networks

After introducing the theory of DeepBSDE, it is not difficult to see that, based on FBSNN, understanding DeepBSDE is actually not hard. Their concepts for constructing the network architecture are largely similar, but there are some minor differences in implementation, which affect the output and the applicable scopes of the two networks.

Feature	FBSNN	DeepBSDE
Time-stepping Direction	Forward + Backward Combination	Fully Backward
Loss Function Structure	Path Loss + Terminal Loss + Gradient Constraint	Terminal Loss Only
FBSDE Correspondence	Direct discretization of the forward-backward FBSDE system	Only backward SDE part discretized
Network Architecture	Single unified network approximating the value function and gradient	Multiple independent networks (one per time step)

The following table integrates training iteration counts as 250, 1000, 2500, and 5200 when, three methods (DeepBSDE Standard, DeepBSDE Legendre, FBSNN) in the 100-dimensional problem's performance comparison.

Training Iterations	Method Name	Estimate	Analytical	Error(Analytical)%	Compute Time(s)
250	DeepBSDE Standard	75.339272	77.104879	2.29	21.15
250	DeepBSDE Legendre	75.298218	77.104879	2.34	388.24
250	FBSNN	65.914825	77.104879	14.51	9.99
1000	DeepBSDE Standard	74.041328	77.104879	3.97	90.09
1000	DeepBSDE Legendre	74.577957	77.104879	3.28	540.21
1000	FBSNN	73.854149	77.104879	4.22	42.65
2500	DeepBSDE Standard	74.447006	77.104879	3.45	244.28
2500	DeepBSDE Legendre	74.753723	77.104879	3.05	687.86
2500	FBSNN	74.256027	77.104879	3.69	119.96
5200	DeepBSDE Standard	73.686493	77.104879	4.43	515.93
5200	DeepBSDE Legendre	75.214691	77.104879	2.45	950.21
5200	FBSNN	75.855530	77.104879	1.62	239.54

From multiple comparative experimental results, it can be seen that FBSNN is the method with the best overall performance, achieving an optimal balance between accuracy and computational efficiency, particularly excelling at high iteration counts. Although DeepBSDE methods show limited improvement or unstable performance as the number of iterations increases, FBSNN can significantly benefit from additional training, continuously enhancing estimation accuracy. This difference primarily stems from their network architecture designs: unlike DeepBSDE, which employs a complex network structure, FBSNN uses the same neural network at each discrete time step to approximate u(t,X), greatly reducing the number of learnable parameters, thereby improving training efficiency and stability. Of course, if the application scenario has extremely high requirements for solution speed, DeepBSDE remains a viable alternative.

6. Future Scalability of PINNs

There are two main sources of inefficiency in computing the PINN loss. The first is order-level inefficiency, which stems from the sequential computation of derivatives of different orders: it is necessary to first construct the computation graph for first-order derivatives, and then perform backpropagation on this graph to obtain second-order derivatives. The second is dimension-level inefficiency, which is a major bottleneck for high-order PDE problems. In modern deep learning frameworks like PyTorch, to implement second-order operators such as the Laplacian, it is necessary to perform backpropagation for each partial derivative sequentially on the computation graph, resulting in computational costs that are proportional to the input dimension.

These two inefficiency sources together lead to the non-parallelizability of the training process, making it particularly slow to learn high-dimensional, high-order problems with PINNs. It can be easily deduced that for a general k-th order d-dimensional PDE, the time complexity of each training iteration of PINN using backpropagation is $O(d^{k-1})$. The industry has already proposed schemes that can efficiently compute second-order derivatives without requiring backpropagation, which warrants further in-depth research.

---

7. Supplement: American Option Pricing Based on Least Squares Monte Carlo Method

> When I started exploring the pricing problem of exotic options, I suddenly realized this might take another 2 months of work, so I stopped here... However, the already implemented American option pricing using the Least Squares Monte Carlo (LSM) method will be explained below:

LSMC (Least Squares Monte Carlo) was proposed by Longstaff and Schwartz in 2001 specifically to solve the pricing challenge of American options. Since American options allow the holder to exercise at any time before expiration, their pricing involves an optimal stopping time problem. The LSM method approximates the conditional expectation at each exercisable time point by constructing a regression model, treats this conditional expectation as the continuation value of the option, and then compares it with the immediate exercise value to determine the optimal exercise strategy.

A simple flowchart to illustrate the idea:

```
Start
│
├─ 1. Initialize Parameters
│   │
│   ├─ Set financial parameters: underlying asset price S0, strike price K, risk-free rate r, volatility σ, maturity T, time steps N
│   ├─ Set simulation parameters: number of paths M, regression basis functions (e.g., polynomials)
│   └─ Initialize cash flow matrix CF (M×N matrix, initially 0)
│
├─ 2. Generate Monte Carlo Paths
│   │
│   ├─ Simulate asset price paths using geometric Brownian motion: dS = rS dt + σS dW
│   ├─ Generate M paths, each with N time points (t_0 to t_N = T)
│   └─ Store path data: S[i,j] represents the asset price of the i-th path at time t_j
│
├─ 3. Terminal Condition Handling (maturity t_N = T)
│   │
│   ├─ For each path i, calculate maturity payoff: CF[i,N] = max(S[i,N] - K, 0) (for call option)
│   └─ Mark the maturity value as the present value of the final cash flow
│
├─ 4. Backward Induction (from t_{N-1} to t_1)
│   │
│   ├─ Loop over time points j = N-1 to 1:
│   │   ||
│   │   ├─ 4.1 Identify "in-the-money paths": select paths where S[i,j] > K (for call option)
│   │   │
│   │   ├─ 4.2 Calculate immediate exercise value: X = max(S[i,j] - K, 0)
│   │   │
│   │   ├─ 4.3 Estimate continuation value: using regression analysis
│   │   │   │
│   │   │   ├─ Select basis functions (e.g., S, S^2, S^3, etc.)
│   │   │   ├─ For in-the-money paths, use S[i,j] as independent variable and discounted cash flow from next time step CF[i,j+1] * e^{-rΔt} as dependent variable
│   │   │   ├─ Perform least squares regression: E[CF_{j+1} | S_j] ≈ β₀ + β₁S + β₂S² + ...
│   │   │   └─ Obtain continuation value estimate V_cont
│   │   │
│   │   ├─ 4.4 Exercise Decision:
│   │   │   ├─ If X > V_cont, exercise early: CF[i,j] = X and clear subsequent cash flows for this path (CF[i,k]=0 for k>j)
│   │   │   └─ Otherwise, continue holding: CF[i,j] = CF[i,j+1] * e^{-rΔt}
│   │   │
│   │   └─ 4.5 Update Cash Flow Matrix
│   │
│   └─ End Time Loop
│
├─ 5. Option Value Calculation
│   │
│   ├─ Initial time t_0: discount all path cash flows to the present
│   ├─ Calculate option value: V0 = (1/M) * Σ_{i=1}^M [CF[i,1] * e^{-r t_1}]
│   └─ Output American option estimated value V0
│
└─ End
```

The implementation uses the most basic monomial basis functions for regression approximation. This method is suitable for low-dimensional problems but may lead to numerical instability in high-order regression. In future work, other basis functions (such as Laguerre polynomials, Hermite polynomials, etc.) can be tried to improve approximation accuracy and numerical stability.

References

[1] Maziar Raissi,Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations, arXiv preprint arXiv:1804.07010 (2018).

[2] Di He, Shanda Li, Wenlei Shi, Xiaotian Gao, Jia Zhang, Jiang Bian, Liwei Wang, Tie-Yan Liu,Learning Physics-Informed Neural Networks without Stacked Back-propagation, arXiv preprint arXiv:2202.09340 (2022).

[3] Jian Liang, Zhe Xu, Peter Li,Deep Learning-Based Least Square Forward-Backward Stochastic Differential Equation Solver for High-Dimensional Derivative Pricing, arXiv preprint arXiv:1907.10578 (2020).

[4] Victor Boussange, Sebastian Becker, Arnulf Jentzen, Benno Kuckuck, Loïc Pellissier,Deep learning approximations for non-local nonlinear PDEs with Neumann boundary conditions, arXiv preprint arXiv:2205.03672 (2022).

[5] Jiequn Han, Arnulf Jentzen, Weinan E,Solving high-dimensional partial differential equations using deep learning, arXiv preprint arXiv:1707.02568(2020).

[6]Li, X., Lin, Y. & Xu, W. On properties of solutions to Black–Scholes–Barenblatt equations. Adv Differ Equ 2019, 193 (2019).

[7]Andersson, Kristoffer. “Approximate stochastic control based on deep learning and forward backward stochastic differential equations.” (2019).

[8]Marco Ciccone, Marco Gallieri, Jonathan Masci, Christian Osendorfer, Faustino Gomez,NAIS-Net: Stable Deep Networks from Non-Autonomous Differential Equations, arXiv preprint arXiv:1804.07209(2018).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learn- ing for image recognition, 2015