People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Saida - Dr Moulay Tahar
Faculty of MIT
Department of Mathematics
No. Assigned by the library

Dissertation presented with a view to obtaining the diploma of

Academic Master

Discipline : MATHEMATICS
Specialty: Statistical Stochastic Analysis of Processes and Applications

by

Maroua HASSANI[1]

Under the supervision of

Dr. Imen MEKKAOUI

Theme:

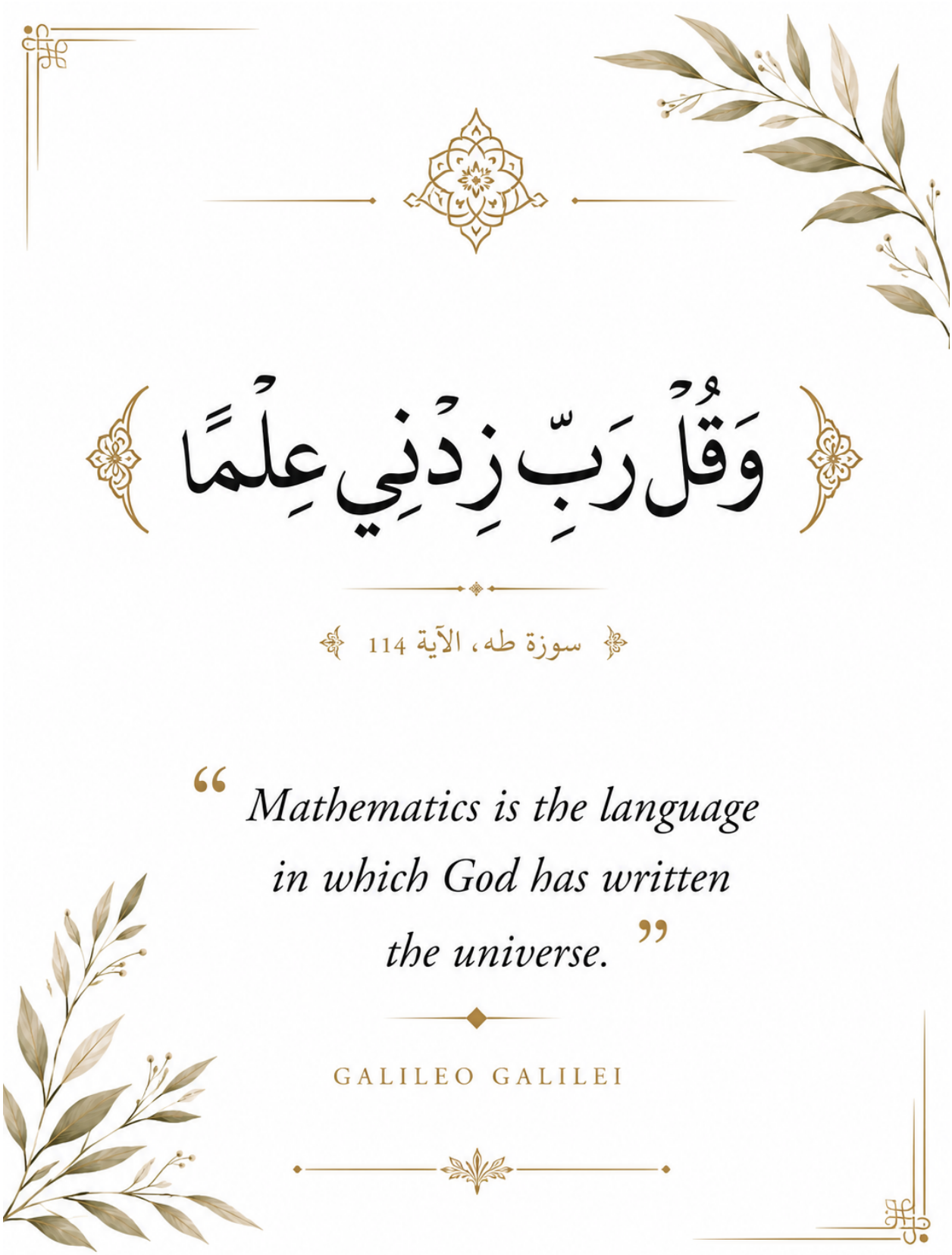
Machine Learning methods for solving Stochastic Differential Equations

Defended on June 14th, 2026 before the jury composed of

Pr. A. KANDOUCI	University of Saida Dr. Moulay Tahar	President
Dr. I. MEKKAOUI	University of Saida Dr. Moulay Tahar	Supervisor
Dr. F. MOKHTARI	University of Saida Dr. Moulay Tahar	Examiner

University year: 2025/2026

[1]e-mail: hassanimaroua25@gmail.com


وَقُلْ رَبِّ زِدْنِي عِلْمًا
سورة طه، الآية 114
“Mathematics is the language in which God has written the universe.”
GALILEO GALILEI

⁎ *Abstract* ⁎

Stochastic differential equations (SDE) play an important role in modeling random phenomena in many scientific fields, including finance, physics, and engineering. Classical numerical techniques, including Euler−Maruyama, or Monte Carlo are commonly used to estimate their responses.

However, these methods may face limitations in terms of efficiency and scalability when dealing with complex or high-dimensional problems. This work deals with machine learning methods, especially Deep Learning methods for solving stochastic differential equations and compares them with classical numerical methods. We focus on commonly used methods in the literature such as Physics Informed Neural Networks (PINNs) and Deep Forward-Backward Stochastic Neural Networks.

The results show that these techniques are promising in terms of providing accurate approximations and being also characterized by their ability to scale and efficiency in computations, especially in the context of high dimensions for nonlinear equations.

⁂ *Acknowledgements* ⁂

I thank Allah always and forever for His endless blessings.
First and foremost, I would like to express my sincere gratitude to my supervisor,
Dr. Imen MEKKAOUI
for her boundless availability, guidance, support, and valuable advice throughout this work. I would also like to thank all the teachers of the Department of Mathematics for their continuous support and encouragement.
I offer my sincerest gratitude to the esteemed members of the committee
Prof. Abdeldjabbar KANDOUCI
Dr. Fatiha MOKHTARI
who generously dedicated their time and expertise to evaluate and appraise this work.
Finally, I am deeply grateful to my family and friends for their unconditional support and motivation during my studies.

⁂ *Dedication* ⁂

Before dedicating this work to others, I would like to dedicate a few words to myself. For every effort made, every obstacle overcome, and every step taken toward this goal.

Although I know I could have done, I am proud of how far I have come and grateful for everything I have learned along the way.

I dedicate this work to myself, with pride, gratitude, and hope for what comes next. I am eternally grateful to all who have supported and nurtured my aspirations.

Foremost, I express my deepest gratitude and affection to my beloved parents. No words can truly express my gratitude for your endless love, sacrifices, patience, and unwavering support. Thank you for believing in me, encouraging me during difficult times, and providing me with the strength to continue pursuing my goals. This achievement would not have been possible without you.

To my two brothers and my two sisters,

Thank you for your encouragement, understanding, and for always being by my side. Your support has been a constant source of motivation throughout this journey.

To my family,

For your kindness, prayers, and continuous support. Thank you for sharing both the challenges and the joys of this journey with me.

With love, gratitude, and appreciation, I dedicate this work to all of you.

List of Figures

Contents

Introduction

Mathematics plays an important role in modeling of real-life phenomena. It is a crucial part of the advancement of human civilization. Far from being a mere theoretical topic, it provides realistic answers to central challenges in science, engineering, economics, etc... For example, mathematical models help to predict planetary movements, streamline commercial production, track the spread of epidemics, analyze economic markets, explore physical phenomena, etc... Mathematical equations, especially differential equations can be found in different disciplines helping to describe with accuracy the time evolution of the related natural phenomena. This unique ability to express facts in structured, quantitative models makes differential equations an important tool for tackling complex issues.

The investigation of these equations under uncertainty and random fluctuations gave rise to the development of Stochastic Differential Equations (SDEs), which are a powerful mathematical framework for modeling systems subject to randomness. Most of these equations do not have direct analytical solutions using theoretical results, and deriving closed-form solutions is usually completely inaccessible. This truth has become the cornerstone of numerical strategies for producing accurate approximations. To cope with these demanding situations, traditional approaches including the Euler method, finite differences, finite elements, and Monte Carlo simulations are undergoing visible sweeping improvements [1], [16], [21]. However, these methods often face the complexities of high-dimensional systems, nonlinear equations, or unknown coefficients.

In recent years, a new class of numerical techniques, called Machine Learning methods have emerged to deal with these complexities [14], [29]. At its core, Machine Learning (ML) functions as a specialized branch of Artificial Intelligence, prioritizing the development of algorithms that automatically extract patterns and hidden relationships from data, and generate predictions from data without needing explicit, step-by-step programming. Its impact is seen across various fields, from image recognition to scientific modeling [13], [22]. The emphasis here is on "automatic", i.e.,

machine learning is concerned with general-purpose methodologies that can be applied to many datasets, while producing something that is meaningful. There are three concepts that are at the core of machine learning: data, a model, and learning. Since machine learning is inherently data driven, data is the most important factor in machine learning. The goal is to design models that are typically related to the process that generates data, similar to the dataset model we are given. A model is said to learn from data if its performance on a given task improves after the data is taken into account. The aim is to find good models that generalize well to yet unseen data, which we may care about in the future. Learning can be understood as a learning way to automatically find patterns and structure in data by optimizing the parameters of the model.

While machine learning has achieved many successes, and software is readily available to design and train rich and flexible machine learning systems, we believe that the mathematical foundations of machine learning are important in order to understand fundamental principles upon which more complicated machine learning systems are built [13], [22]. Understanding these principles can facilitate creating new machine learning solutions, understanding and debugging existing approaches, and learning about the inherent assumptions and limitations of the methodologies we are working with.

A new class of machine learning techniques, called Deep Learning methods (DL) appeared as a dominant force in the field of numerical simulations [14], [29], due to its use of multi-layered deep neural networks (NN), which are exceptionally capable of approximating complex nonlinear problems with high accuracy [13]. The remarkable capacity of neural networks to approximate complex functions has sparked a shift toward their use in solving numerically mathematical equations such as differential equations, partial differential equations, stochastic differential equations, etc...[7], [23], [29]. What makes this approach particularly compelling is its mesh-free or network-independent nature, offering a versatile alternative to conventional methods that rely on spatial slicing or discretization such as finite difference, finite element, or spectral methods [23], [29]. Deep learning methods have proven to be very effective in dealing with large class of high-dimensional problems, and it might hold the key to solve the curse of dimensionality problem that arises when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings [24], [14], [29]. It should be emphasized that at the present time, there are no theoretical results that support such claims although the practical success of deep learning has been astonishing. However, this should not prevent researchers from trying to apply and test deep learning to other problems with more complex issues.

Among Deep Learning based methods for stochastic differential equations, we explore the concept of Physics-Informed Neural Networks (PINNs), which is a method that is trained to solve supervised learning tasks while respecting any given laws of physics described by general nonlinear partial differential equation called the Fokker-Planck equation. This equation serves as the PDE counterpart to an Itô stochastic differential equation. It describes the time evolution of the probability density function of the SDE's solution rather than the trajectory of individual particles [7], [21], [27], [29]. Another method we will focus on is Forward-Backward Stochastic Neural Networks (FBSNNs) [11], [14], which is a deep learning-based approach that can handle general high-dimensional parabolic PDEs. Developing algorithms for solving high-dimensional partial differential equations (PDEs) has been an exceedingly difficult task for a long time, due to the curse of dimensionality problem [4], [14]. To this end, the PDEs are reformulated using backward stochastic differential equations (BSDE) and the gradient of the unknown solution is approximated by neural networks. Here by using the reformulation of BSDEs, the problem of solving PDEs will be seen as a learning problem and a deep-learning framework is designed to fit naturally to that setting. It should be mentioned that even though deep learning has been a very successful tool for a number of applications, adapting it to the current setting with practical success is still a highly nontrivial task that needs more investigations and research.

The main objectives of this work are:

- To recall the mathematical background of stochastic differential equations and their role in modeling uncertain systems.
- To introduce classical numerical methods used for solving SDEs and analyze their limitations, especially in high-dimensional settings.
- To clarify the mathematical aspect of deep learning methods and how they work.
- To introduce machine learning-based approaches for solving differential equations, with a focus on Deep Learning methods such as Physics-Informed Neural Networks (PINNs) and Forward-Backward Stochastic Neural Networks (FBSNNs) methods.
- To explore how these methods can be used improve performance in terms of accuracy, scalability, and computational efficiency.

- To present a comparative evaluation between classical numerical strategies and machine learning-based approaches for solving stochastic differential equations.

This thesis is composed of four chapters. In the first chapter, we introduce the essential concepts of stochastic differential equations (SDEs) and some classical numerical techniques used to simulate solutions for SDEs. The second chapter is devoted to the definitions of machine learning and deep learning basics, concepts and necessary tools that will be used in the following chapters of this thesis. The third chapter presents the Physics-Informed Neural Network (PINNs) method for solving SDEs with explanation of the relationship between the SDEs and the corresponding Fokker-Planck partial differential equation (PDE). Numerical examples are given and discussed at the end of the chapter. In the fourth and last chapter, we deal with Forward-Backward Stochastic Neural Networks (FBSNNs) method using the well-known connection between high-dimensional partial differential equations and forward-backward stochastic differential equations. The effectiveness of this approach is tested by numerical simulations of some benchmark examples of SDEs.

Chapter 1

Reminders about Stochastic Differential Equations

In this chapter, we will discuss the stochastic differential equations and their fundamentals, as well as some classical methods for solving them. For more details, the reader is referred to the following works [1, 16, 21, 27]

1.1 Stochastic differential equations

An ordinary differential equation (ODE) is used to describe a system′s evolution deterministically, meaning the initial condition and governing dynamics fully dictate its future behavior. However, in many real-world applications, systems face uncertainties, random fluctuations, or external noise that purely deterministic models cannot capture. This limitation motivates the introduction of stochastic differential equations (SDEs)[27], which extend ODEs by incorporating random components. The SDE can be written as

$$dX_t = b(X_t, t)\, dt + \sigma(X_t, t)\, dW_t, \quad X_t \in \mathbb{R}^d$$

where

- $X_t \in \mathbb{R}^d$ is a stochastic process depending on time.
- $b(\cdot)$ is the drift function, representing the deterministic part of the dynamics.
- $\sigma(\cdot)$ is the diffusion function, controlling the intensity of randomness.
- W_t is a standard Brownian motion.

The first term governs the average trend of the system, while the second term accounts for random perturbations

From a modeling perspective, SDEs provide a flexible framework to represent complex dynamical systems influenced by noise. They have become central tools in many areas, including finance, physics, biology, and engineering. In practice, explicit analytical solutions of SDEs are rarely available, which motivates the development of numerical and data-driven methods for their approximation

1.2 Brownian motion

Brownian motion, often referred to Wiener process, is fundamental in the formulation of stochastic differential equations (SDEs). As a continuous-time stochastic process, it serves to model random fluctuations and is a standard way to represent noise within dynamical systems.
A one-dimensional Brownian motion $(W_t)_{t\geq 0}$ is characterized by the following basic properties:

- It starts at zero: $W_0 = 0$.
- It has **independent increments**: for any $0 \leq s < t$, the increment $W_t - W_s$ is independent of the past values W_u for $u \leq s$.
- Its increments are **normally distributed** with mean zero and variance proportional to the time increment:

$$W_t - W_s \sim \mathcal{N}(0, t - s)$$

Intuitively, Brownian motion can be viewed as a mathematical model of random, irregular motion. In the context of SDEs, Brownian motion provides a convenient and widely accepted way to model uncertainty and random perturbations affecting the evolution of the system.

1.3 Existence and uniqueness of solutions

The investigation of the existence and uniqueness of solutions is one of the most important contributions in stochastic differential equations, as it guarantees that a given SDE admits a well-defined and meaningful solution. this ensures that the system dynamics described by the equation are mathematically consistent and that the model does not lead to ambiguous behaviors.

Under suitable regularity conditions on the drift and diffusion coefficients, such as:

- **Linear growth condition:** There exists a constant $C > 0$ such that, for all $(t, x) \in [0, T] \times \mathbb{R}^d$,

$$|b(t, x)| + |\sigma(t, x)| \leq C\big(1 + |x|\big).$$

- **Lipschitz continuity condition:** There exists a constant $D > 0$ such that, for all $t \in [0, T]$ and for all $x, y \in \mathbb{R}^d$,

$$|b(t, x) - b(t, y)| + |\sigma(t, x) - \sigma(t, y)| \leq D|x - y|.$$

classical results ensure the existence of a unique strong solution to the SDE.

1.4 Classical Examples of SDEs

To illustrate these theoretical assumptions, we look at two classical models: the Ornstein–Uhlenbeck process, acting as a clear example of mean-reverting dynamics, and the Black–Scholes model, which is a standard application within finance.

Ornstein–Uhlenbeck Model

The Ornstein–Uhlenbeck (OU) process models mean-reverting dynamics:

$$dX_t = \theta(\mu - X_t)\, dt + \sigma\, dB_t, \quad X_0 = x_0, \tag{1.1}$$

where:

- $\theta > 0$ is the rate of mean reversion,
- μ is the long-term mean,
- $\sigma > 0$ is the volatility,
- B_t is a standard Brownian motion.

Existence and uniqueness. The coefficients

$$b(x) = \theta(\mu - x), \quad \sigma(x) = \sigma$$

satisfy the Lipschitz and linear growth conditions. Therefore, the OU SDE admits a unique adapted solution.

Exact solution. The explicit solution of the OU SDE is given by

$$X_t = \mu + (x_0 - \mu)e^{-\theta t} + \sigma \int_0^t e^{-\theta(t-s)}\, dB_s. \tag{1.2}$$

This shows explicitly how the process evolves over time, reverting toward the mean μ.

Black–Scholes Model

The Black–Scholes SDE models a stock price:

$$dS_t = \mu S_t\, dt + \sigma S_t\, dB_t, \quad S_0 = s_0, \tag{1.3}$$

where:

- μ is the drift,
- σ is the volatility,
- B_t is a standard Brownian motion.

Existence and uniqueness: The coefficients

$$b(S) = \mu S, \quad \sigma(S) = \sigma S$$

satisfy the Lipschitz and linear growth conditions. Therefore, there exists a unique adapted solution.

Exact solution: The explicit solution of the Black–Scholes SDE is

$$S_t = S_0 \exp\Big((\mu - \tfrac{1}{2}\sigma^2)t + \sigma B_t\Big). \tag{1.4}$$

We notice that while S_t is not a martingale due to the factor $e^{\mu t}$, the process is well-defined and satisfies the SDE assumptions.

1.5 Numerical Methods for SDEs

Since SDEs can rarely be solved analytically in practical scenarios, we rely on numerical methods to find approximate solutions for the process's evolution. Below, we outline some of the most standard schemes used for this purpose. The details of these methods can be found in [30].

Euler–Maruyama Method

The Euler–Maruyama (EM) scheme represents the most straightforward extension of the classical Euler method from ODEs to the stochastic realm. It serves as a fundamental numerical technique for approximating the solutions of these differential equations.
We consider the following stochastic differential equation (SDE):

$$dX_t = b(X_t, t)\, dt + \sigma(X_t, t)\, dB_t, \quad X_0 = x_0, \tag{1.5}$$

where $(B_t)_{t\geq 0}$ is a standard Brownian motion, b denotes the drift coefficient and σ the diffusion coefficient.
We introduce a time discretization:

$$0 = t_0 < t_1 < \cdots < t_N = T, \qquad \Delta t = t_{n+1} - t_n. \tag{1.6}$$

The Euler–Maruyama scheme is given by:

$$X_{n+1} = X_n + b(X_n, t_n)\, \Delta t + \sigma(X_n, t_n)\, \Delta B_n, \tag{1.7}$$

where

$$\Delta B_n = B_{t_{n+1}} - B_{t_n} \sim \mathcal{N}(0, \Delta t). \tag{1.8}$$

Remarks.

- The Euler–Maruyama method is a first-order method in time (strong error of order $O(\Delta t)$).
- It is simple to implement.
- It performs well for low-dimensional problems.

Examples

Euler–Maruyama Scheme for the OU Process: The associated numerical scheme reads:

$$X_{n+1} = X_n + \theta(\mu - X_n)\Delta t + \sigma\sqrt{\Delta t}\, Z_n, \tag{1.9}$$

where $(Z_n)_{n\geq 0}$ are independent standard normal random variables, i.e., $Z_n \sim \mathcal{N}(0, 1)$.

Euler–Maruyama Scheme for Black–Scholes: The Euler–Maruyama approximation reads:

$$S_{n+1} = S_n + \mu S_n \Delta t + \sigma S_n \Delta B_n, \tag{1.10}$$

where $\Delta B_n \sim \mathcal{N}(0, \Delta t)$.

Milstein Method

The Milstein method for stochastic differential equations (SDEs) is a numerical technique used to approximate the solution of SDEs. It is similar to the Euler-Maruyama method but incorporates an additional correction factor involving the derivative of the diffusion term.
The Milstein method improves accuracy by including an extra term from Itô's formula:

$$X_{i+1} = X_i + a(X_i, t_i)\,\Delta t_i + b(X_i, t_i)\,\Delta W_i + \frac{1}{2}\,b(X_i, t_i)\,\frac{\partial b}{\partial x}(X_i, t_i)\,\left((\Delta W_i)^2 - \Delta t_i\right). \tag{1.11}$$

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

Remarks.

- The Milstein scheme has a higher order of strong convergence than the Euler–Maruyama method.
- It is particularly useful when the diffusion coefficient $b(X_t, t)$ depends on the state variable X_t.
- The method is slightly more complex to implement since it requires the computation of the derivative $\dfrac{\partial b}{\partial x}$.

Monte Carlo Method

The Monte Carlo method for stochastic differential equations (SDEs) is a computational technique used to approximate the solution of SDEs through random sampling. It involves simulating multiple paths of the stochastic process and averaging the results to estimate the solution. Monte Carlo (MC) is often combined with Euler or Milstein methods to approximate statistical quantities, such as expectations, variances, or option prices in finance:

$$\mathbb{E}[f(X_T)] \;\approx\; \frac{1}{M}\sum_{i=1}^{M} f\left(X_T^{(i)}\right), \tag{1.12}$$

where $X_T^{(i)}$ are independent realizations of the SDE solution obtained using the Euler–Maruyama or Milstein scheme.

Remarks.

- The Monte Carlo method is straightforward to apply in high-dimensional problems.
- The accuracy of the approximation increases with the number of simulations M.
- This approach is widely used in financial applications, such as Black–Scholes option pricing.

1.6 Limitations of Classical Methods

Despite their effectiveness in low-dimensional and well-specified models, classical numerical methods for SDEs suffer from several important limitations:

- **Curse of dimensionality:** In high-dimensional systems, the computational complexity increases exponentially with the dimension, which makes classical schemes difficult to apply in practice[4], [14].
- **High computational cost:** Fine time discretization and large numbers of Monte Carlo simulations are often required to obtain accurate approximations, leading to significant computational time and memory requirements[16], [21].
- **Complex or unknown coefficients:** When the drift b and diffusion σ are highly nonlinear, irregular, or unknown, it becomes difficult to design accurate and stable numerical schemes[16], [21].
- **Limited flexibility in data-driven settings:** Classical methods rely on explicit model specifications and are not well adapted to scenarios where the dynamics must be learned directly from observed data [7], [29].

These limitations become particularly apparent when moving from simple benchmark models, such as the Ornstein–Uhlenbeck and Black–Scholes equations, to more realistic and high-dimensional stochastic systems.

Connection to Previous Examples

In models like Ornstein–Uhlenbeck and Black–Scholes, both drift (b) and diffusion (σ) meet the standard criteria of Lipschitz continuity and linear growth, which guarantees that a unique solution exists. While exact analytical solutions are available for these cases, we typically rely on numerical schemes like the Euler–Maruyama

method to approximate the trajectories of X_t or S_t in practice. However, classical approaches often struggle when dealing with high-dimensional systems or when the coefficients are too intricate to be explicitly defined. This is where Neural SDEs come in: by approximating b and σ with neural networks, they offer a more flexible framework for learning and simulating complex dynamics directly from data. By essentially replacing traditional drift and diffusion terms with learnable structures, this approach proves particularly effective in data-driven and high-dimensional settings [7], [14], [28].

Chapter 2

Machine Learning and Deep Learning basics

This chapter provides a comprehensive overview of the fundamental concepts of Machine Learning and Deep Learning. It begins by introducing the main types of machine learning models and the notion of training algorithms and their role in learning from data. It then presents the basic principles of deep learning, including neural network architectures, multilayer perceptrons, and activation functions. The chapter further covers key training components such as loss functions, backpropagation, and hyperparameters. Finally, it highlights different deep learning models and discusses the differences between Machine Learning and Deep Learning in terms of structure, capabilities, and data requirements, as well as the main challenges encountered when training deep neural networks.

2.1 Machine Learning basics

Machine learning (ML) is a subset of Artificial Intelligence (AI) that is primarily focused on enabling computers to learn from data with minimal human intervention. It can solve tasks that are infeasible or too cumbersome with "traditional" programming languages [13], [22]. In traditional programming, a human writes explicit rules for the computer to follow, but in machine learning, the computer learns the rules from examples. Machine learning uses mathematical procedures that enable computer systems to learn from data, identify patterns, and make predictions without being explicitly programmed. These Mathematical procedures are called algorithms.

Despite the variety of machine learning algorithms, the data is arguably more important than the selected algorithm. Many issues can arise with data, such as insufficient data, poor quality of data, incorrect data, missing data, irrelevant data, duplicate

data values, and so forth. Different techniques are developed in the literature to address these data-related issues.

2.1.1 The main types of Machine Learning models

Machine learning offers various ways for computers to learn from data. Each type of model has a distinct approach to how machines can understand information and make smart decisions [5], [13].

Supervised learning:

In supervised learning, the algorithm learns from labeled examples. So, humans provide input data and the correct output for each example. It's like learning with an answer key. The model tries to generalize from the provided examples to predict the correct output for new, unseen inputs.

Unsupervised learning:

In unsupervised learning, the algorithm works with data that doesn't have labels or predefined answers. Instead of being told what each example is, it studies the data to uncover patterns and groupings on its own. This helps reveal hidden structures or relationships that might not be immediately visible.

Reinforcement learning:

In reinforcement learning, the agent, like a computer program or system, interacts with its environment and learns from the outcomes of its actions. This agent receives rewards for good actions and penalties for bad ones, gradually figuring out which choices lead to the best results. It's similar to learning through trial and error, as each decision influences what it does next.

2.1.2 Notion of training algorithms in Machine Learning

A machine learning algorithm allows the system to adapt some internal parameters to the available input data so that it performs well on future unseen input data. Here we refer to this adaptation as "training a system".

The model presented by the algorithm is typically used to describe a process for generating data, similar to the dataset at hand. Therefore, good models can also be thought of as simplified versions of the real (unknown) data-generating process.

capturing aspects that are relevant for modeling the data and extracting hidden patterns from it. A good model can then be used to predict what would happen in the real world without performing real-world experiments.

The learning components of machine learning assume we are given a dataset and a suitable model. Training the model means using the available data to optimize some parameters of the model with respect to what we call a "utility or cost function" that evaluates how well the model predicts the training data and it corresponds to the optimization of some desired performance measure. However, in practice, we are interested in the model performing well on unseen data. Performing well on data that we have already seen (training data) may only mean that we found a good way to memorize the data. However, this may not generalize well to unseen data, and in practical applications, we often need to expose our machine learning system to situations that it has not encountered before[5], [13], [20].

2.1.3 Types of Machine Learning Algorithms

There are three main types of machine learning algorithms [13]:

Regression:

Is a supervised learning technique used to predict numerical quantities. An example of a regression task is predicting the value of a particular stock. Note that this task is different from predicting whether the value of a particular stock will increase or decrease tomorrow (or some other future time period). Another example of a regression task involves predicting the cost of a house in a real estate dataset. Both of these tasks are examples of regression tasks. Regression algorithms in machine learning include linear regression and generalized linear regression.

Classification:

Is also a supervised learning technique, but it's used for predicting categorical quantities. An example of a classification task is detecting the occurrence of spam, fraud, or determining the digit represented in an image. In this case, the data is already labeled, so you can compare the prediction with the label that was assigned to the given PNG. Classification algorithms in machine learning include the following list of algorithms:

- Decision Trees (a single tree)
- Random Forests (multiple trees)

- kNN (k Nearest Neighbor)
- Logistic regression
- Naïve Bayes
- SVM (Support Vector Machines)

Each of the preceding algorithms involves a model that is trained on a dataset, after which the model is used to make a prediction.

Clustering:

Is an unsupervised learning technique for grouping similar data together. Clustering algorithms put data points in different clusters without knowing the nature of the data points. After the data has been separated into different clusters, the SVM (Support Vector Machine) algorithm can be used to perform classification. Clustering algorithms in machine learning include the following:

- k-Means
- Mean Shift
- Hierarchical Cluster Analysis (HCA)
- Expectation-Maximization

2.2 Deep Learning principles

Deep learning (DL) is a specialized subfield of machine learning[13]. The term "deep" refers to the multiple layers within the neural network that deep learning algorithms use which are inspired by the human brain's network of neurons. While a simple machine learning model might consider a few features of the data, a deep learning model has many layers of simulated "neurons", allowing it to automatically learn increasingly abstract features from the raw data. Within this domain, Deep Learning (DL) utilizes multi-layered neural networks to tackle more intricate data relationships [9], [13], [22]. By learning hierarchical features, these deep networks can capture the kind of highly nonlinear patterns that simpler models might miss. At their core, Neural Networks (NNs) serve as the fundamental components of DL (Fig 2.1). These networks are made up of layers of interconnected neurons, where each unit processes an input through a mathematical transformation before passing it along. Through the training process, the network fine-tunes its parameters to approximate complex functions.

We can think of each layer as learning a higher-level representation: for example, in an image, early layers of a deep network might learn to detect edges, middle layers might detect shapes or textures, and final layers might recognize objects like faces or cars.

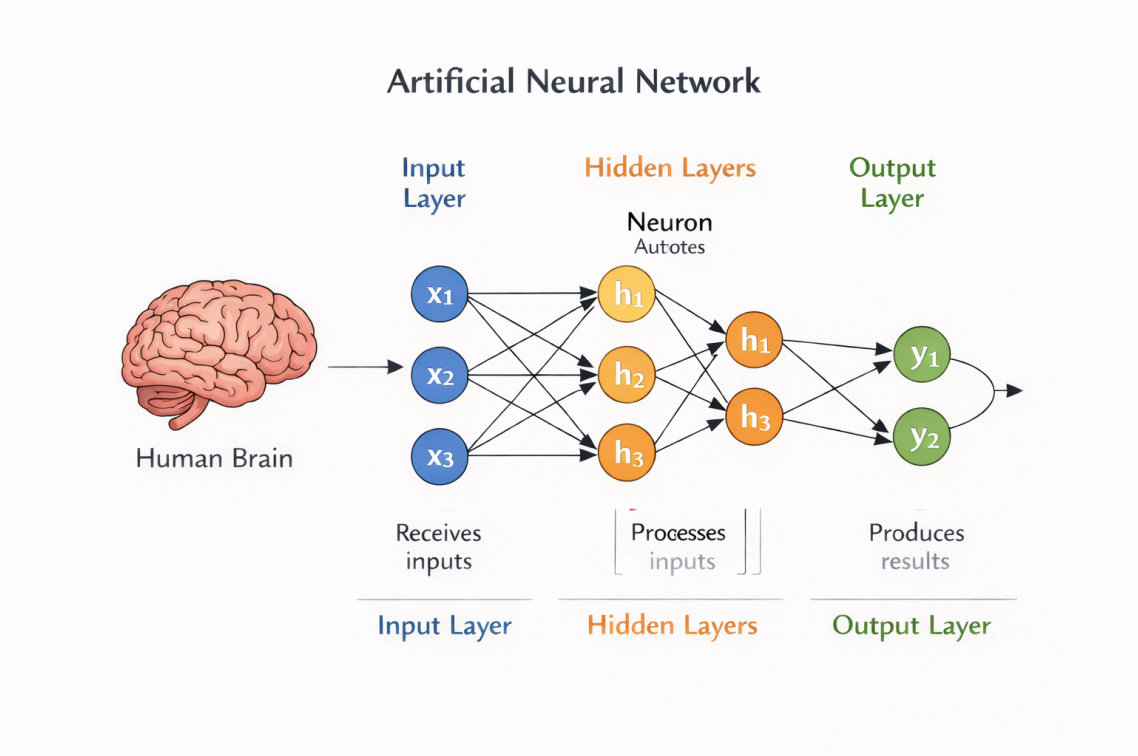


Figure 2.1: Neural network architecture

2.2.1 Types of deep learning models

Deep learning includes various types of models, each suited for specific kinds of data and tasks. Although they all use neural networks, their structures and purposes differ depending on what they are designed to handle. Some of the most common types of deep learning models are [13], [22]:

Artificial neural networks (ANNs):

ANNs are the basic form of neural networks. They consist of an input layer, an output layer, as well as one or more hidden layers, with each neuron in a layer connected to every neuron in the next ayer. They are typically the foundation for more specialized deep learning models.

Convolutional neural networks (CNNs):

CNNs use special convolutional layers to detect patterns such as edges, textures, and shapes within images. This makes them highly effective for tasks like identifying objects in photos, facial recognition, analyzing satellite imagery, and interpreting medical scans to detect conditions or abnormalities.

Recurrent neural networks (RNNs):

RNNs are built to handle sequential data where order matters. Unlike other networks, RNNs have connections that loop back, enabling them to remember previous inputs as they process new ones. This makes them ideal for tasks where understanding the context of previous words or sounds is important for accurate results.

Generative adversarial networks (GANs):

GANs involve two networks working against each other: a generator that creates new data and a discriminator that checks if the data is real or fake. Through this competition, GANs become highly effective at generating realistic images, videos, or even music.

2.2.2 Hyperparameters in Deep Learning

Deep learning involves hyperparameters, which are sort of like knobs and dials whose values are initialized by the user prior to the actual training process. For instance, the number of hidden layers and the number of neurons in hidden layers are examples of hyperparameters. we encounter many hyperparameters in deep learning models, some of which are listed below:

- Number of hidden layers
- Number of neurons in hidden layers
- Weight initialization
- An activation function
- A cost function
- An optimizer
- A learning rate

The first three hyperparameters in the preceding list are required for the initial set-up of a neural network. The fourth hyperparameter is required for forward propagation. The next three hyperparameters (i.e., the cost function, optimizer, and learning rate) are required in order to perform backward error propagation during supervised learning tasks [1]. This step calculates a set of numbers that are used to update the values of the weights in the neural network in order to improve the accuracy of the neural network.

2.2.3 Deep Learning architecture

The most important tool in Deep Learning is neural networks, which are considered computational models inspired by the structure of the human brain. They consist of interconnected units called neurons, which process information through weighted connections [13].
A neural network is prepared in layers: an input layer that receives data, one or more hidden layers that process the information, and an output layer that produces the final result (as seen in Fig. 2.1). Each neuron applies a linear transformation followed by a non-linear activation function, allowing the network to learn complex relationships from data.
Depending on the structure of the data and the problem, different neural network architectures can be used. The most common ones are Multi-Layer Perceptrons (MLP), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks

2.2.3.1 Activation Functions in Neural Networks

Activation functions are mathematical functions used within neurons to introduce nonlinearity into neural networks. Data science strives to give computers the ability to think in a way that mimics human thought as closely as possible. Activation functions are a fundamental element of neural networks, introducing nonlinearity into the model, which allows networks to learn complex patterns and relationships in data.
Many different activation functions are available, including binary, linear, and various nonlinear functions. For data scientists and machine learning engineers, the challenge lies in determining the appropriate function or sequence of functions to train the neural network. Activation functions are essential because without them, networks become equivalent linear regression models incapable of learning complex patterns [26]. The commonly used activation functions in neural networks include:

[1] The back-propagation procedure will be discussed in Section 2.2.4

- **Hyperbolic tangent (tanh):** The Tanh function is a common activation function in neural networks, where it converts any real input value to an output value in the range [-1, 1]. This function is zero-centered, making it suitable for use in the hidden layers of networks, especially in recurrent neural networks (RNNs):
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

- **The Sigmoid Function:** This function converts any real input value into an output value in the interval [0, 1]. Its key feature is that it returns values that can be interpreted as probabilities, making it suitable for use in recurrent networks such as LSTM:
$$\sigma(x) = \frac{e^x}{e^x + 1}.$$

- **Rectified Linear Unit (ReLU):** This function converts any real input value to an output value equal to the input itself if positive, and zero if negative. Its simplicity, computational speed, and ability to solve the gradient vanishing problem [2] make it an ideal choice for hidden layers in deep networks such as MLP:
$$ReLU(x) = \max(0, x).$$

2.2.3.2 Multilayer Perceptron (MLP)

The MLP is the simplest architecture of neural network. It is a feedforward network, meaning that information flows from input to output layers without feedback. Given an input vector x, each layer performs a linear transformation followed by a nonlinear activation function [13]. For example, the first hidden layer computes:

$$z_1 = W_1 x + b_1, \tag{2.1}$$

$$a_1 = \sigma(z_1), \tag{2.2}$$

where W_1 and b_1 denote the weight matrix and bias vector, and $\sigma(\cdot)$ is a nonlinear activation function. The output of this layer is then passed to the next layer:

$$z_2 = W_2 a_1 + b_2. \tag{2.3}$$

This process continues layer by layer (over all the hidden layers) until the final output is obtained. Since information flows strictly from input to output without feedback connections, the MLP is called a feedforward network. Here is a Python

[2] The vanishing gradient problem is discussed in Section 2.2.4.

code on the creation and initialization of this type of neural network:

```
import torch # torch provides tensor operations and automatic
    differentiation.
import torch.nn as nn # torch.nn contains modules for building
     neural networks from Pytorch.

class MLP(nn.Module): # The class inherits from nn.Module,
    which is the base class for all PyTorch neural networks.

     def __init__(self):
          super().__init__() # Initializing the neural network.
             self.net = nn.Sequential(
              nn.Linear(2, 64),      #2 Inputs with 64 neurons as
                   input layer
              nn.Tanh(), # Choice of the activation function for
                   each layer
              nn.Linear(64, 64), # one hidden layer with 64x64
                 neurons
              nn.Tanh(),
              nn.Linear(64, 1)      # One output in the last (
                 output) layer
          )
```

Listing 2.1: Python code for MLP initialization.

2.2.3.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are designed to process sequential data. Unlike MLPs, they have a memory mechanism that allows them to capture temporal dependencies [12]. The main idea is that each step depends on the previous one. At time step t, the hidden state is computed as:

$$h_t = \sigma\left(W_h h_{t-1} + W_x x_t + b\right) \tag{2.4}$$

where h_t denotes the hidden state at time t, h_{t-1} is the previous hidden state, x_t is the input at time t, W_h and W_x are weight matrices, b is a bias vector, and $\sigma(\cdot)$ is a nonlinear activation function. This repetitive formation allows assuming a temporal dependency, and therefore the present state is clearly dependent on the former one. Recurrent neural networks (RNNs) are particularly suitable for modeling stochastic policies, given that such methods evolve over years and depend on prior values.

It remembers information from previous time steps through its hidden state h_t. At each time step, the hidden state contains information about the previous state. This memory makes RNNs particularly useful for time series, trajectories, and such as time series, trajectories, and sequential stochastic processes.
However, Classical RNNs suffer from many limitations. Notably, at some point in training, they may suffer from the vanishing gradient problem. When back-propagated through multiple time steps, gradients can become very small, making it difficult to study the long-term dependence of a community [3].
As a consequence, classical RNNs often struggle to retain information over long time horizons and fail to capture long-term dependencies effectively. The Python code below describes how to create and initialize a RNN structure:

```python
import torch
import torch.nn as nn

class RNN(nn.Module):

    def __init__(self, input_size=2, hidden_size=64): # each time step contains 2 features. The hidden state contains 64 values.
        super().__init__()

        self.rnn = nn.RNN(    # This creates a recurrent neural network layer.
            input_size=input_size,
            hidden_size=hidden_size,
            batch_first=True
        )

        self.fc = nn.Linear(hidden_size, 1) # Creating the output layer

    def forward(self, x, y):
        out, hidden = self.rnn(x, y) # Passing through the RNN
        return self.fc(out)
```

2.2.3.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of neural network architecture designed to overcome the vanishing and exploding gradient problems [4] by introducing a gat-

[3] The problem of vanishing gradient in back-propagation is discussed in Section 2.2.4.
[4] The exploding and vanishing gradient problems are treated in Section 2.2.4.

ing mechanism. LSTMs contain specialized cells with gates that control the flow of information, allowing them to selectively remember or forget information from the input sequence. The key components of an LSTM cell are the forget gate, input gate, and output gate. These gates regulate the update of the cell state, enabling the LSTM to capture long-term dependencies effectively [17]. Below is the Python code presenting the LSTM architecture:

```
import torch
import torch.nn as nn

class LSTM(nn.Module): # Creates the LSTM network

    def __init__(self, input_size=2, hidden_size=64): # Specification of parameters
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True) # Implementation of the created LSTM
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x, y): # Describes the performed calculation when calling the LSTM model
        inputs = torch.cat([x, y], dim=-1)
        out, (hidden, cell) = self.lstm(inputs) # At each time step, calculate a hidden state.
        return self.fc(out)
```

Figure 2.2 presents the differences in architecture between the above models.

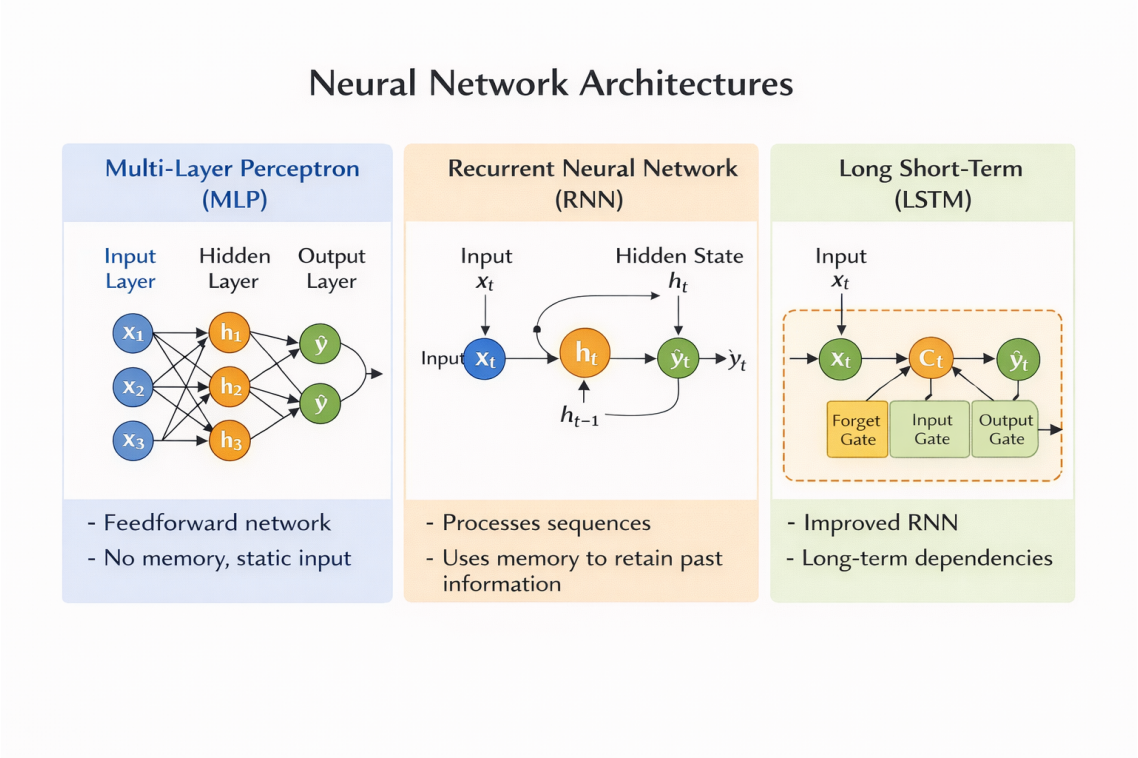


Figure 2.2: Comparison of Neural Network Architectures (MLP, RNN, and LSTM)

We can remark that neural networks can be used to approximate the solution or the coefficients of stochastic differential equations. Depending on the nature of the problem, different architectures can be employed, MLPs for static approximation, and RNNs or LSTMs for time-dependent stochastic processes.

2.2.4 Loss function and back-propagation in neural networks

Loss function in neural network:

Consider a neural network composed of n layers:

$$x \to a_1 \to a_2 \to \cdots \to a_n \to L,$$

where

- x is the input,
- a_1 is the activation of the first layer,
- a_2 is the activation of the second layer,
- $\cdots$

- a_n is the output activation (prediction),
- L is the loss function.

The activations are computed recursively as

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)},$$

$$a^{(l)} = \sigma\left(z^{(l)}\right),$$

where

- $W^{(l)}$ is the weight matrix of layer l,
- $b^{(l)}$ is the bias vector of layer l,
- $z^{(l)}$ is the pre-activation vector,
- σ is the activation function.

The input is denoted by: $a^{(0)} = x$. For example, in a three-layer network:

$$z_1 = W_1 x + b_1 \quad ; \quad a_1 = \sigma(z_1),$$

$$z_2 = W_2 a_1 + b_2 \quad ; \quad a_2 = \sigma(z_2),$$

$$z_3 = W_3 a_2 + b_3 \quad ; \quad a_3 = \sigma(z_3),$$

and

$$a_3 = \hat{y},$$

where $\hat{y}$ denotes the network prediction. The loss function is then computed as

$$L = \mathcal{L}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2,$$

in regression for example, where y is the true target.

Loss function optimization:

The aim is to reduce the loss function L with respect to the network parameters, weights W and biases b, i.e., to find the best values of the parameters such that the neural network output matches the target data as closely as possible.
To do that, we need to solve an optimization problem for L to update its parameters by using optimization algorithms requiring the calculation of the gradients:

$$\frac{\partial L}{\partial W},$$

for every weight W in the network. The same thing is done for the biases b.
Then optimization methods such as "gradient descent algorithms" are called to update the weights:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W},$$

where η is the learning rate of the gradient descent method.

Backpropagation using the chain rule:

Using the chain rule, gradients are propagated from the output layer back to the first (or input) layer. This is called the "backpropagation procedure".
The gradient can be expressed, for example, for the first layer as:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1}.$$

More generally, for a network with n layers,

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial a_{n-1}} \cdot \frac{\partial a_{n-1}}{\partial a_{n-2}} \cdots \frac{\partial a_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1}.$$

Each factor measures how a change in one layer affects the next layer. Multiplying all these terms gives the influence of the first-layer weights on the final loss.

Vanishing and exploding gradients:

The terms:

$$\frac{\partial a^{(n)}}{\partial a^{(n-1)}}, \quad \frac{\partial a^{(n-1)}}{\partial a^{(n-2)}}, \quad \dots$$

contain derivatives of activation functions and weight matrices.
If these derivatives are mostly smaller than one in magnitude, then repeated multiplication causes the gradient to decrease exponentially:

$$\left\| \frac{\partial L}{\partial W} \right\| \approx 0.$$

In this case, the weights are no longer updated, and the neural network is essentially inert.Also, debugging this problem is generally nontrivial. This phenomenon is known as the **vanishing gradient problem**.
Conversely, if these derivatives are mostly larger than one in magnitude, the gradient can grow exponentially:

$$\left\| \frac{\partial L}{\partial W} \right\| \gg 1.$$

This phenomenon is known as the **exploding gradient problem**.

Both issues make training deep neural networks difficult and motivate the use of techniques such as ReLU activations, proper weight initialization, gradient clipping, and LSTM architectures [13, 17, 6].
These problems can also arise when using the sigmoid activation function. Deep learning models usually replace the sigmoid activation function with the ReLU activation function [20, 13, 26]. ReLU is a very simple continuous function that is differentiable everywhere except at the origin. Its derivative is equal to 0 for $x < 0$ and 1 for $x > 0$. At $x = 0$, the derivative is undefined; however, in practice, a suitable value (typically 0) is assigned during back-propagation, allowing gradient-based optimization algorithms to operate effectively.

Full training cycle:

The training cycle can be summarized in the following steps:

1. Forward pass: $x \rightarrow \hat{y}$
2. Compute loss: $L(y, \hat{y})$
3. Backward pass (backpropagation) : $\dfrac{\partial L}{\partial W}$
4. Update weights: $W \leftarrow W - \eta \dfrac{\partial L}{\partial W}$
5. Repeat again the forward pass using the updated weights and so on...

2.2.5 Challenges in Deep Learning

Although deep learning is powerful and has produced impressive results in many fields, there are some important ongoing challenges that are being explored, including:

- Bias in algorithms
- Susceptibility to adversarial attacks
- Limited ability to generalize
- Lack of explainability
- Correlation but not causality

Algorithms can contain unintentional bias, and even if the bias is removed, there can be unintentional bias in data. Deep learning focuses on finding patterns in datasets, and generalizing those results is a more difficult task. There are some initiatives

that attempt to provide explainability for the outcomes of neural networks, but such work is still in its infancy. Deep learning finds patterns and can determine correlations, but it is incapable of determining causality.

2.3 Differences Between Machine Learning and Deep Learning

It's already clear that machine learning and deep learning are closely connected. In fact, deep learning is essentially a more advanced subset of machine learning. They both involve training models that make predictions or decisions based on data, with one using more complex neural networks that allow it to handle more detailed and unstructured data. The differences between deep learning and machine learning become more apparent when we directly compare them in the following categories:

Data requirements:

Machine learning models usually perform best when working with smaller amounts of data, such as a few hundred or thousand examples. They work especially well when that data is structured and clearly labeled.
In contrast to that, deep learning models require large datasets that often include thousands or millions of examples in order to reach their full potential. The reason behind this is that deep learning has many internal parameters to adjust, and without enough data, it risks memorizing training examples instead of learning general patterns.

Feature engineering:

The way the two fields handle the details within data also differs. In most traditional machine learning approaches, the user needs to decide in advance what information the model should focus on. This is the process known as feature engineering. It applies to both supervised and unsupervised machine learning, as models rely on the features provided to find patterns or make predictions.
Deep learning works differently. Because of its many layers, it can analyze raw data directly and figure out for itself what matters most. This is why deep learning models do not need feature engineering; instead, they discover and learn such things on their own as they train.

Interpretability:

Machine learning models, particularly simpler ones, are generally easier to interpret. That's because we can usually see which features influenced a prediction and how they were weighted. So, they're generally more transparent and explainable.
Deep learning models, however, are more challenging to comprehend, particularly in terms of how their layers and parameters interact to reach a decision. This lower interpretability can be a disadvantage in fields where explaining a prediction is as important as its accuracy.

Speed and resources:

Machine learning models are typically faster to train and require less computational power. They can often be developed and deployed on standard computers without specialized hardware.
Deep learning models, on the other hand, need more time and resources to train due to their complexity and the size of the data they use. Training such models usually requires powerful GPUs (graphics processing units) or cloud computing services, and they consume more memory and energy compared to simpler machine learning models.

2.4 When to Use Machine Learning vs. Deep Learning

Understanding both approaches, particularly the differences between deep learning and machine learning, is necessary when deciding which to use. Both have their strengths, and knowing when to apply each one naturally leads to better results.
While machine learning offers simplicity and clarity, deep learning unlocks powerful capabilities for more demanding tasks. Therefore, it's preferred to use machine learning when:

- We deal with problems that involve structured data with clear features.
- We need results quickly with smaller or limited datasets.
- Model explainability is important, such as in finance or healthcare.

On the other hand, deep learning might be more suitable when:

- Working with complex, unstructured data like images, audio, or natural language.

- Having large and high-quality datasets to support training.
- Maximum accuracy is needed, and interpretability is less critical to our use case.

The best thing is that with the rise of user-friendly libraries (such as scikit-learn, TensorFlow, and PyTorch) and cloud services through Microsoft Azure Machine Learning and IBM Watson, both ML and DL are becoming accessible even if the user is not an expert. So, the user can experiment and see what works best for their situation.

2.5 Advantages and limitations of Machine Learning and Deep Learning

All machine learning and deep learning models require human oversight. We present some of the main advantages and limitations of ML and DL so we can determine which team is best equipped to handle.

Advantages of Machine Learning

- Faster training times compared to deep learning: ML models like decision trees or logistic regression can be trained in minutes or hours, even on large datasets.
- Requires less computational power: Many ML algorithms can run effectively on standard CPUs (Central Processing Unit), making them accessible for smaller organizations or individual researchers.
- Works well with structured, smaller datasets: ML techniques like random forests can produce accurate results with just a few thousand data points.
- More interpretable models: Decision trees, for example, provide clear rules that can be easily interpreted and used.
- Suitable for a wide range of problems: Techniques like support vector machines can perform well even with modest amounts of training data.

Advantages of Deep Learning

- Exceptional performance on complex tasks like image and speech recognition: CNNs have achieved human-level accuracy in image classification tasks on large datasets like ImageNet.

- Ability to handle large, unstructured datasets: Recurrent neural networks (RNNs) and transformers can process and generate human-like text from a vast collection of natural language data.
- Automatic feature extraction, reducing the need for manual feature engineering: Deep networks can learn to identify relevant features in raw data, such as edges and textures in images.
- Continual improvement with more data: Unlike many ML algorithms, deep learning models are built to continuously retrain and adapt to new data inputs.

Limitations of Machine Learning

- Often requires extensive feature engineering: Domain expertise is usually needed to create effective features, which can be time-consuming and limit the model's generalizability.
- Machine learning models May struggle with complex tasks or unstructured data: Traditional ML algorithms often fall short on tasks like natural language understanding or complex image recognition.
- Performance can be limited after a certain point: Adding more data may not significantly improve model performance beyond a certain threshold.

Limitations of Deep Learning

- Risk of overfitting, especially with smaller datasets: Deep learning models with millions of parameters can easily memorize training data, leading to poor generalization on new, unseen data if not carefully managed.
- Requires large amounts of data for optimal performance: Deep learning models may need millions of examples to achieve state-of-the-art performance, which can be challenging in domains with limited data availability.
- High computational cost for training and sometimes for inference: Training large models can require significant GPU (Graphics Processing Unit) resources and take weeks or months, making it impractical for many tasks.
- the "Black box" nature of models can make them difficult to interpret: The complex interconnections in deep neural networks make it challenging to explain why a particular decision was made, which can make it difficult to understand and interpret the results.

Chapter 3

Deep Learning-based PINNs method

Classical numerical techniques for stochastic differential equations (SDEs) and partial differential equations (PDEs) often encounter a fundamental limitation known as the curse of dimensionality [14, 23, 24]. Since these methods rely on domain discretization, their computational cost increases exponentially with the dimensionality of the system, making them impractical for high-dimensional problems. In addition, the complexity of stochastic dynamics and the possible lack of explicit knowledge of system coefficients further restrict the applicability of classical numerical schemes in real-world scenarios [16, 21]. To overcome these limitations, this chapter introduces one of the promising deep learning methods, which is Physics-Informed Neural Networks (PINNs) approach for solving SDEs [9, 18]. These tools open up new possibilities for modeling systems with random temporal evolution, while classical methods often hit a wall in high-dimensional settings. We give a detailed description of the method which is based on solving the Fokker Planck equation that governs the probability density function of the stochastic process solution to the SDE. Numerical examples are then presented to illustrate the effectiveness of this method [29, 33].

3.1 Principle and motivation

To address the limitations of classical methods, deep learning based PINNs method has emerged as a powerful alternative for approximating high-dimensional nonlinear differential equation. Neural Networks (NNs) provide mesh-free approximation capabilities, making them especially suitable for high-dimensional stochastic systems where grid-based methods fail such as finite difference or finite element methods [25, 31]. These approaches focus on discretizing the computational domain into a set of grid points and solving approximate solutions on the grid, but the solution is only calculated at the grid point, and the evaluation of any other point

requires an interpolation or other reconstruction techniques. For one-dimensional (1D) problems, these methods are very effective, but in two-dimensional (2D) and higher-dimensional systems, they consume much computing resources and even have troubles to overcome the curse of dimensionality. In addition, a sparsity of the grid seriously affects the accuracy of calculation, while a dense grid will inevitably lead to a sharp increase in the amount of calculation. Another approach is to solve the stochastic differential equation and then statistically calculate its transition probability density via the so-called Monte Carlo method. Although this approach does not require interpolation, it needs a large number of sample paths and the accuracy of its solution is related to the amount of generated data. For a 2D system only, the computational cost of solving the joint probability density function (JPDF) may be too large to be obtained. Therefore, it is still a problem that needs to be solved to find an alternative technique without interpolation calculation and with rather low computational complexity. The use of artificial neural networks (ANNs) provides a promising way to solve this problem.

The key aspect of PINNs is the deep connection between stochastic differential equations (SDEs) and the Fokker-Planck partial differential equation (PDE) that gives the probability density of the stochastic process of the SDE changing a stochastic problem to a deterministic one. The idea is to integrate the structure of the corresponding Fokker-Planck PDE directly into the learning process. This approach not only overcomes the limitations of discretization-based methods but also opens new directions for solving high-dimensional stochastic problems efficiently [9, 29].

3.2 The Fokker-Planck Equation

Physics-Informed Neural Networks (PINNs) are a class of neural networks designed to solve partial differential equations (PDEs), in our case, it is the Fokker-Planck equation without requiring any observed data. The principle of this method is to train the network to approximate the desired solution using the same equation without needing external data. The PDE itself is used as a training signal. The only information needed is the PDE itself, along with initial and boundary conditions [7, 29]. The network learns to minimize the residual $R(x,t)$, which measures how much the network output violates the PDE.

The Fokker-Planck equation (also known as the forward Kolmogorov equation) is a partial differential equation (PDE) that describes the evolution of a probability density function $p(x,t)$ over time of all possible trajectories of the stochastic process X_t at position x and time t [27] solution to the SDE:

$$dX_t = \mu(X_t,t)dt + \sigma(X_t,t)dW_t, \tag{3.1}$$

starting from 0 to $T > 0$. The corresponding Fokker–Planck equation in one dimension is:

$$\frac{\partial p(x,t)}{\partial t} = -\frac{\partial}{\partial x}\big(\mu(x,t)\,p(x,t)\big) + \frac{1}{2}\frac{\partial^2}{\partial x^2}\big(\sigma^2(x,t)\,p(x,t)\big). \tag{3.2}$$

In general in n dimension, it becomes:

$$\frac{\partial p(x,t)}{\partial t} = -\sum_{i=1}^{n}\frac{\partial}{\partial x_i}(\mu_i p) + \frac{1}{2}\sum_{i,j=1}^{n}\frac{\partial}{\partial x_i x_j}\big((\sigma\sigma^T)_{i,j}p\big), \quad x \in \Omega \subset \mathbb{R}^n, t \in [0,T], \tag{3.3}$$

with initial condition $p(x,0) = p_0(x)$, and suitable boundary conditions:

$$\mathcal{B}(p,x) = 0 \quad \text{on} \quad \partial\Omega.$$

$\mathcal{B}$ is the boundary operator which could be Dirichlet, Neumann, Robin, or periodic boundary conditions [23]. In addition, $p(x,t)$ is the transition probability density function of the SDE (3.1), which must satisfy the normalization condition:

$$\int_{\Omega} p(x,t)dx = 1, \quad \forall t \in [0,T], \tag{3.4}$$

and tends to zero as x tends to infinity.

3.3 How PINNs work for the Fokker-Planck Equation

3.3.1 Step 1: Neural Network Approximation

A neural network takes (x,t) as input and outputs $\hat{p}(x,t)$, which is an approximation of the probability density function $p(x,t)$ [7], as illustrated in Fig. 3.1 that shows a typical structure of an ANN which is also called deep neural network. We describe below the numerical scheme to solve the FP equation via a deep ANN with L layers:

$$H = \big[h_1^l, h_2^l, h_3^l, \ldots, h_{n_l}^l\big] \qquad (2 \le l \le L-1)$$

is the output of the lth hidden layer, and n_l is the number of nodes in each layer.

$$W = \big[w^1, w^2, \ldots, w^l, \ldots, w^{L-1}\big]$$

are the weights of network, and w^l represents the weight from the $(l-1)$th layer to the lth layer. The size of the matrix w^l is $n_{l-1} \times n_l$.

$$\text{Bias} = \left[b^1, b^2, \ldots, b^l, \ldots, b^{L-1}\right]$$

is the bias of the layers, and b^l represents the bias of the $(l+1)$th layer with the size $n_l \times 1$. Then, we can describe a feed-forward process as

$$h_i^2 = F\left(w_i^1 \mathbf{x} + b_i^1\right), \quad \mathbf{x} = (x, t), \tag{3.5}$$

$$h_i^l = F\left(\sum_{j=1}^{n_{l-1}} w_{ij}^{l-1} h_j^{l-1} + b_i^{l-1}\right), \qquad 3 \leq l \leq L-1, \tag{3.6}$$

$$y = \sum_{i=1}^{n_{L-1}} w_i^{L-1} h_i^{L-1} + b^{L-1}, \tag{3.7}$$

where F represents the activation function (like a sigmoid, tanh, ReLU, etc.), and the output can be described as

$$y = \mathcal{F}(\mathbf{x}; \theta),$$

where θ denotes all trainable parameters of the network. $\mathcal{F}$ is the composite function which is made of the activation function, and $\theta = [W, Bias]$ is denoted as the network parameters. In general, the current methods solving the differential equations with ANN can be defined as follows: Assuming a differential equation with the initial condition and boundary condition to be:

$$\begin{cases} \mathcal{I}(p(x,0)) = 0, & x \in \Omega, \\ \mathcal{B}(p(x,t)) = g(x), & (x,t) \in \partial\Omega \times [0,T], \end{cases} \tag{3.8}$$

where $\mathcal{I}[\cdot]$ is the initial condition operator and $\mathcal{B}[\cdot]$ is the boundary operator. We set the domain data:

$$\mathcal{T}_D = \left\{\mathbf{x}_i^D \in \Omega \times [0,T]\right\}_{i=1}^{N_D}$$

and the boundary data

$$\mathcal{T}_B = \left\{\mathbf{x}_i^B \in \partial\Omega \times [0,T]\right\}_{i=1}^{N_B}$$

are discretized as input data of ANN, and the corresponding output is:

$$\hat{p} = \hat{p}(\mathbf{x}; \theta),$$

where θ are the parameters of the underlying ANN.

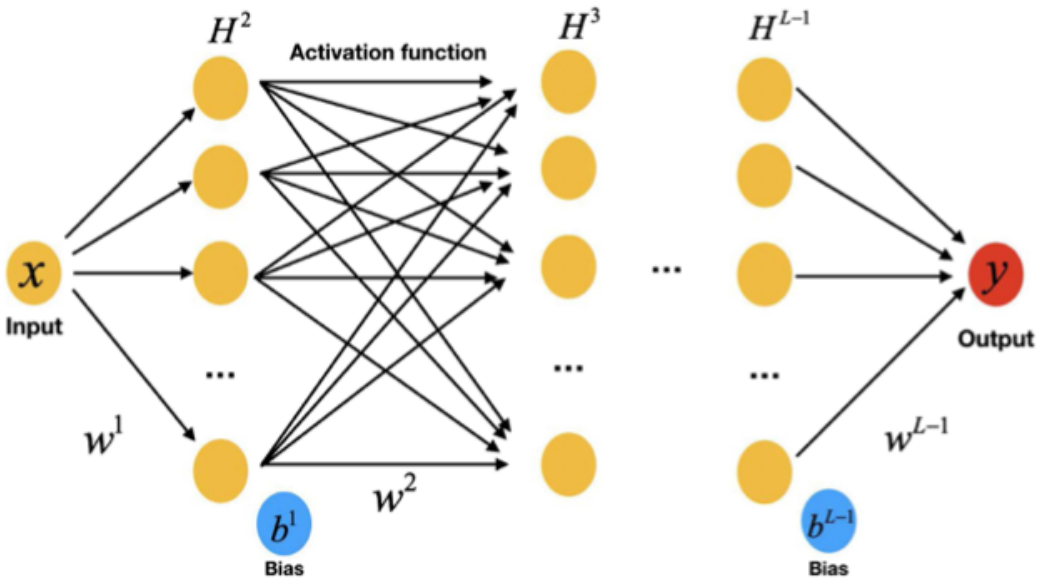


Figure 3.1: Deep neural network architecture.

3.3.2 Step 2: Automatic Differentiation

According to [23], the derivatives of the network outputs with respect to the network inputs.:

$$\frac{\partial \hat{p}}{\partial t}, \quad \frac{\partial \hat{p}}{\partial x}, \quad \frac{\partial^2 \hat{p}}{\partial x^2},$$

for the FP equation are computed using automatic differentiation (AD); also called algorithmic differentiation. In deep learning, the derivatives are evaluated using back-propagation specialized technique of AD. Considering the fact that the neural network represents a compositional function, then AD applies the chain rule repeatedly to compute the derivatives as explained in Section 2.2.4.

3.3.3 Step 3: Residual of the PDE

These derivatives are substituted into the Fokker–Planck equation to compute the residual (in 1D for example):

$$R(x,t) = \frac{\partial \hat{p}(x,t)}{\partial t} + \frac{\partial}{\partial x}\big(\mu(x,t)\,\hat{p}(x,t)\big) - \frac{1}{2}\frac{\partial^2}{\partial x^2}\big(\sigma^2(x,t)\,\hat{p}(x,t)\big).$$

3.3.4 Step 4: Training Objective

The neural network is trained such that:

$$R(x,t) \approx 0,$$

while also satisfying the initial conditions and boundary conditions of the problem:

$$\begin{cases} \mathcal{I}(p(x,0)) \approx 0, & x \in \Omega, \\ \mathcal{B}(p(x,t)) \approx g(x), & (x,t) \in \partial\Omega \times [0,T]. \end{cases} \tag{3.9}$$

3.4 The loss function in PINNs and Adam optimizer

3.4.1 Loss Function in PINNs

The loss function is a single number that measures how far the network's output is from the real solution. The smaller the loss, the closer the network is to solving the equation. The loss function is minimized with respect to the network parameters $\theta = (W, biais)$. For PINNs, for solving the Fokker-Planck equation, the loss function combines three terms (see [29]):

$$\mathcal{L} = \mathcal{L}_{PDE} + \mathcal{L}_{initial} + \mathcal{L}_{boundary} + \mathcal{L}_{normalize}$$

where:
- $\mathcal{L}_{PDE}$: measures how well the network satisfies the PDE which is related to the residual $R(x,t)$:

$$\mathcal{L}_{PDE} = \frac{1}{N_D} \sum_{i=1}^{N_D} \left| R(\mathbf{x}_i^D; \theta) \right|^2,$$

- $\mathcal{L}_{initial}$: enforces the initial condition of the PDE:

$$\mathcal{L}_{initial} = \frac{1}{N_D} \sum_{i=1}^{N_D} \left| \mathcal{I}\left(\hat{p}(x_i^D; \theta)\right) \right|^2,$$

- $\mathcal{L}_{boundary}$: enforces the boundary conditions of the PDE:

$$\mathcal{L}_{boundary} = \frac{1}{N_B} \sum_{i=1}^{N_B} \left| \mathcal{B}\left(\hat{p}(\mathbf{x}_i^B; \theta)\right) - g(\mathbf{x}_i^B) \right|^2,$$

- $\mathcal{L}_{normalize}$: to add the normalization condition as the supervision condition to

guarantee that the solution to the FP equation presents well a probability density function (3.4):

$$\mathcal{L}_{normalize} = \left| \sum_{i=1}^{N_D} \left((\Delta x)^{dim} \hat{p}(\mathbf{x}_i^B; \theta) \right) - 1 \right|^2 ,$$

Δx is the step length discretization on each dimension of the training set $\mathcal{T}_D$ and dim is the dimension of the variable x.

During training, the network adjusts its weights to minimize this loss function. The same steps discussed in Section 2.2.4 are followed here also. When the loss is very close to zero, the network output is considered to be a very good approximation to the real solution.

3.4.2 Adam Optimizer in PINNs

In Section 2.2.4, we have presented "Gradient Descent method" as an optimizer for the loss function, however, in PINNs, the most popular method used usually in the literature is "Adam (Adaptive Moment) optimizer" as proposed by [19] which is which an adaptive stepping method. Let's explain its principle:
Denote by θ_t the network parameters at iteration k, and:

$$g_k = \nabla_\theta \mathcal{L}(\theta_k)$$

is the gradient of the loss function $\mathcal{L}$ with respect to the set of parameters θ.
Adam computes the first moment estimate (the exponentially weighted average of past gradients) as:

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k, \tag{3.10}$$

where $\beta_1 \in [0, 1)$ controls the decay rate of the moving average.
The second moment estimate (the exponentially weighted average of squared gradients) is given by:

$$v_k = \beta_2 v_{k-1} + (1 - \beta_2) g_k^2, \tag{3.11}$$

where $\beta_2 \in [0, 1)$ controls the decay rate of the moving average of squared gradients. Since both m_k and v_k are initialized at zero, their values are underestimated during the first iterations. Adam therefore applies a bias-correction step to compensate for this effect by computing:

$$\hat{m}_k = \frac{m_k}{1 - \beta_1^k}, \tag{3.12}$$

$$\hat{v}_k = \frac{v_k}{1-\beta_2^k}. \tag{3.13}$$

The network parameters are then updated according to:

$$\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \varepsilon}, \tag{3.14}$$

where α is the learning rate and ε is a small positive constant introduced to avoid division by zero.
The first moment estimate m_k can be interpreted as a smoothed estimate of the gradient direction, while the second moment estimate v_t measures the magnitude of recent gradients. Consequently, Adam adapts the learning rate for each parameter individually, leading to faster and more stable convergence during training.
In practice, the default values proposed by [19] are commonly used:

$$\beta_1 = 0.9, \qquad \beta_2 = 0.999, \qquad \varepsilon = 10^{-8}.$$

These values have been shown to perform well across a wide range of deep learning applications.

3.5 Numerical experiments and comparative study

In this section, we dive into the numerical simulation and a comparison between PINNs approach for solving SDEs and Monte Carlo method (MC), and see how the MC scheme holds up against PINNs. Let's treat the following examples:

Example 1. *Consider the SDE:*

$$dX_t = f(X_t)dt + \sigma dW_t$$

with

$$f(x) = \alpha - x + \frac{x^8}{1+x^8}.$$

The associated Fokker-Planck equation is:

$$\frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}\big(f(x)p(x)\big) + \frac{\sigma^2}{2}\frac{\partial^2 p(x)}{\partial x^2}$$

with initial condition $X_0 \sim p_0(x) = p(x,0)$.
In the steady-state regime we have:

$$-\frac{d}{dx}\big(f(x)p(x)\big) + \frac{\sigma^2}{2}\frac{d^2 p(x)}{dx^2} = 0.$$

Natural conditions:

- The density p must be positive.
- $\int p(x,t)dx = 1.$
- $p(x,t) \to 0$ when $|x| \to \infty$.
- The probability flow:

$$J(x) = f(x)p(x,t) - \frac{\sigma^2}{2}\frac{\partial p}{\partial x}$$

must become zero: $J \to 0$ on the boundary of the domain in x.
All these conditions are taken into account to solve the problem. We describe in the following Python code the steps of computations:

Python Code:

```
import torch
import torch.nn as nn
import torch.autograd as autograd
import numpy as np
import numpy as np
import matplotlib.pyplot as plt

# ====== parameters ======
sigma = 0.5
alpha = 0.6
xmin, xmax = -2.0, 4.0
# ====== Network ======
class PINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 30),
            nn.Tanh(),

            nn.Linear(30, 30),
            nn.Tanh(),

            nn.Linear(30, 30),
            nn.Tanh(),
```

```
25
26              nn.Linear(30, 30),
27              nn.Tanh(),
28              nn.Linear(30, 1),
29              nn.Softplus()  # garantee p(x) >= 0
30          )
31
32      def forward(self, x):
33          return self.net(x)
34
35  model = PINN()
36
37  # ====== drift ======
38  def f(x):
39      return alpha - x + x**8 / (1 + x**8)
40  # ====== gradients ======
41  def gradients(y, x):
42      return autograd.grad(y, x, grad_outputs=torch.ones_like(y)
        ,
43                           create_graph=True)[0]
44
45  # ====== loss ======
46  def loss_function(model, x_interior, x_boundary):
47
48      x_interior.requires_grad_(True)
49
50      p = model(x_interior)
51      dp_dx = gradients(p, x_interior)
52      d2p_dx2 = gradients(dp_dx, x_interior)
53      # terme PDE
54      drift = f(x_interior)
55      fp = drift * p
56      dfp_dx = gradients(fp, x_interior)
57
58      pde = -dfp_dx + (sigma**2 / 2) * d2p_dx2
59      loss_pde = torch.mean(pde**2)
60
61      # ====== flow on the boundary ======
62      x_boundary.requires_grad_(True)
63      p_b = model(x_boundary)
64      dp_dx_b = gradients(p_b, x_boundary)
```

```
    flux_b = f(x_boundary) * p_b - (sigma**2 / 2) * dp_dx_b
    loss_bc = torch.mean(flux_b**2)
    # ====== normalization constraints ======
    x_norm = torch.linspace(xmin, xmax, 600).view(-1,1)
    p_norm = model(x_norm)
    integral = torch.trapz(p_norm.squeeze(), x_norm.squeeze())
    loss_norm = (integral - 1.0)**2

    return loss_pde + loss_bc + loss_norm
# ====== data======
N_interior = 2000
N_boundary = 2

x_interior = torch.rand(N_interior, 1)*(xmax - xmin) + xmin
x_boundary = torch.tensor([[xmin],[xmax]], dtype=torch.float32
    )

# ====== optimization ======
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
# ====== training ======
for epoch in range(10000):
    optimizer.zero_grad()
    loss = loss_function(model, x_interior, x_boundary)
    loss.backward()
    optimizer.step()
    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
# ====== results ======
x_plot = torch.linspace(xmin, xmax, 598).view(-1,1)
p_plot = model(x_plot).detach().numpy()
plt.plot(x_plot.numpy(), p_plot)
plt.title("Steady-state density (PINN)")
plt.show()
# ====== Monte Carlo ======
def simulate_sde(N=50000, T=100, dt=0.01):
    n_steps = int(T / dt)

    # initial values
    X = np.random.normal(0, 1, N)

    for _ in range(n_steps):
```

```
        dW = np.sqrt(dt) * np.random.randn(N)
        X = X + (alpha - X + X**8/(1 + X**8)) * dt + sigma *
            dW

    return X
# simulation
samples = simulate_sde()
# ====== Monte Carlo ======
hist, bins = np.histogram(samples, bins=100, density=True)
centers = 0.5 * (bins[:-1] + bins[1:])
# ====== PINN ======
x_plot = torch.linspace(xmin, xmax, 500).view(-1,1)
p_plot = model(x_plot).detach().numpy().flatten()
# ====== plot results ======
plt.figure(figsize=(8,5))
# Monte Carlo
plt.plot(centers, hist, label="Monte Carlo", linewidth=2)
# PINN
plt.plot(x_plot.numpy(), p_plot, '--', label="PINN", linewidth
    =2)
plt.title("Comparaison PINN vs Monte Carlo")
plt.xlabel("x")
plt.ylabel("density")
plt.legend()
plt.grid()
plt.show()
```

Listing 3.1: Python code for exempel 1.

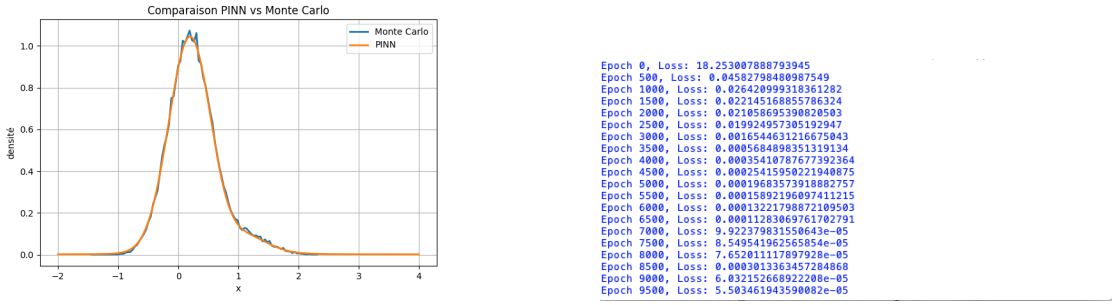
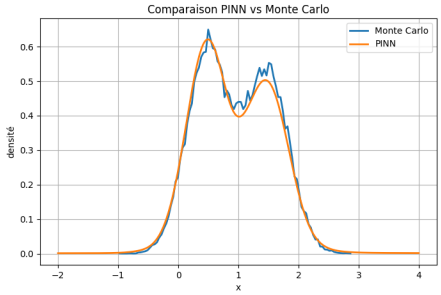
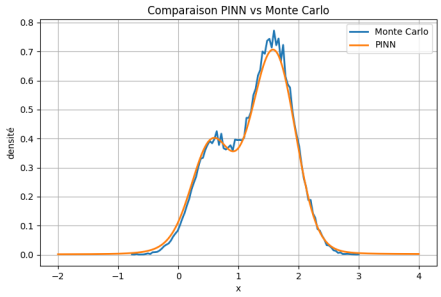


Figure 3.2: PINN Fokker-Planck equation vs Monte Carlo method with $\alpha = 0.2$




```
Epoch 0, Loss: 9.80897331237793
Epoch 500, Loss: 0.028749195858836174
Epoch 1000, Loss: 0.022586945444345474
Epoch 1500, Loss: 0.010928943753242493
Epoch 2000, Loss: 0.003679409390315413 5
Epoch 2500, Loss: 0.003169016912579536 4
Epoch 3000, Loss: 0.002469558268785476 7
Epoch 3500, Loss: 0.001076175947673618 8
Epoch 4000, Loss: 0.000520904548466205 6
Epoch 4500, Loss: 0.000340057304129004 5
Epoch 5000, Loss: 0.000176962028490379 45
Epoch 5500, Loss: 0.000128371320897713 3
Epoch 6000, Loss: 0.000070042081642895 94
Epoch 6500, Loss: 0.000010596556239761412
Epoch 7000, Loss: 6.961915642023087e-05
Epoch 7500, Loss: 6.768019375158474e-05
Epoch 8000, Loss: 0.000247146061155945 06
Epoch 8500, Loss: 6.618873885599896e-05
Epoch 9000, Loss: 4.376802098704502e-05
Epoch 9500, Loss: 3.793632276938297e-05
```

Figure 3.3: PINN Fokker-Planck equation vs Monte Carlo method with $\alpha = 0.5$




```
Epoch 0, Loss: 12.392997741699219
Epoch 500, Loss: 0.03296588361263275
Epoch 1000, Loss: 0.022783212363719 94
Epoch 1500, Loss: 0.020995561033487 32
Epoch 2000, Loss: 0.020079296082258224
Epoch 2500, Loss: 0.018076967447996 14
Epoch 3000, Loss: 0.006332297809422016
Epoch 3500, Loss: 0.005495490040630102
Epoch 4000, Loss: 0.004270217381417751
Epoch 4500, Loss: 0.001566133578307926 7
Epoch 5000, Loss: 0.000541379558853805 1
Epoch 5500, Loss: 0.000764125550631433 7
Epoch 6000, Loss: 0.000220032816287130 12
Epoch 6500, Loss: 0.000191546190762892 37
Epoch 7000, Loss: 0.000154599561938084 66
Epoch 7500, Loss: 0.000126581027870997 8
Epoch 8000, Loss: 0.000111913359432946 89
Epoch 8500, Loss: 9.330041211796924e-05
Epoch 9000, Loss: 8.271639671875164e-05
Epoch 9500, Loss: 7.864940562285483e-05
```

Figure 3.4: PINN Fokker-Planck equation vs Monte Carlo method with $\alpha = 0.6$

Observation: For all considered values of α (0.2, 0.5, and 0.6), the PINN Fokker–Planck solution shows excellent agreement with the Monte Carlo approximation. The main features of the probability density are accurately captured by the PINN model. While the Monte Carlo curves exhibit small oscillations resulting from sampling noise, the PINN solutions remain smooth and stable, highlighting their ability to provide accurate approximations of stochastic dynamics.

Example 2. *Consider the following stochastic differential equation:*

$$dX_t = \alpha X_t - \beta X_t^3 + \sigma dW_t, \qquad \alpha > 0,\ \beta > 0. \tag{3.15}$$

The corresponding stationary Fokker–Planck equation associated with the SDE (3.15) *is given by*

$$-\frac{\partial}{\partial x}\left[(\alpha x - \beta x^3)\, p(x)\right] + \frac{\sigma^2}{2}\frac{\partial^2 p(x)}{\partial x^2} = 0. \tag{3.16}$$

the exact steady-state solution is [33]:

$$p_s(x) = C\, \exp\left(\frac{2\alpha x^2 - \beta x^4}{2\sigma^2}\right), \tag{3.17}$$

where C is some normalization constant.

The Python code for this example is provided below:

Python Code:

```
import torch
import torch.nn as nn
import torch.autograd as autograd
import numpy as np
import matplotlib.pyplot as plt

# ====== seed for reproducibility ======
torch.manual_seed(0)
np.random.seed(0)

# ====== parameters (change for table rows) ======
alpha = 0.5
beta = 0.5
sigma = 0.5

xmin, xmax = -2.5, 2.5

# ====== Neural Network ======
class PINN(nn.Module):
    def __init__(self):
        super(PINN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.Tanh(),

            nn.Linear(64, 64),
            nn.Tanh(),

            nn.Linear(64, 64),
            nn.Tanh(),

            nn.Linear(64, 1),
            nn.Softplus()   # p(x) >= 0
        )

    def forward(self, x):
        return self.net(x)


```

```
model = PINN()

# ====== drift ======
def f(x):
    return alpha * x - beta * x**3

# ====== gradients ======
def grad(y, x):
    return autograd.grad(
        y, x,
        grad_outputs=torch.ones_like(y),
        create_graph=True
    )[0]

# ====== loss ======
def loss_function(model, x_in, x_b):

    x_in.requires_grad_(True)

    p = model(x_in)
    dp = grad(p, x_in)
    d2p = grad(dp, x_in)

    drift = f(x_in)
    fp = drift * p
    dfp = grad(fp, x_in)

    pde = -dfp + (sigma**2 / 2) * d2p
    loss_pde = torch.mean(pde**2)

    # boundary
    x_b.requires_grad_(True)
    p_b = model(x_b)
    dp_b = grad(p_b, x_b)

    flux = f(x_b) * p_b - (sigma**2 / 2) * dp_b
    loss_bc = torch.mean(flux**2)

    # normalization
    x_n = torch.linspace(xmin, xmax, 500).view(-1,1)
    p_n = model(x_n)
```

```
    integral = torch.trapz(p_n.squeeze(), x_n.squeeze())
    loss_norm = (integral - 1)**2

    return loss_pde + loss_bc + loss_norm


# ====== data ======
N_interior = 3000

x_in = torch.rand(N_interior,1)*(xmax-xmin) + xmin
x_b = torch.tensor([[xmin],[xmax]], dtype=torch.float32)

# ====== optimizer ======
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# ====== training ======
epochs = 30000

for epoch in range(epochs):
    optimizer.zero_grad()
    loss = loss_function(model, x_in, x_b)
    loss.backward()
    optimizer.step()

    if epoch % 2000 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.3e}")

# ====== evaluation ======
x = torch.linspace(xmin, xmax, 1000).view(-1,1)
x_np = x.detach().numpy().flatten()

p_pinn = model(x).detach().numpy().flatten()

# ====== exact solution ======
p_exact = np.exp(
    (2*alpha*x_np**2 - beta*x_np**4) / (2*sigma**2)
)

p_exact /= np.trapezoid(p_exact, x_np)

# ====== error ======
```

```
L2_error = np.linalg.norm(p_pinn - p_exact) / np.linalg.norm(
    p_exact)
accL2 = 1 - L2_error

print("\n======================")
print("alpha =", alpha)
print("beta =", beta)
print("sigma =", sigma)
print("Loss value =", loss.item())
print("accL2 =", accL2)
print("Iterations =", epochs)
print("======================\n")

# ====== plot ======
plt.figure(figsize=(8,5))
plt.plot(x_np, p_exact, 'k', label="Exact")
plt.plot(x_np, p_pinn, 'r--', label="PINN")
plt.legend()
plt.grid()
plt.title("PINN vs Exact")
plt.show()
```

Listing 3.2: Python code for exempel 2.

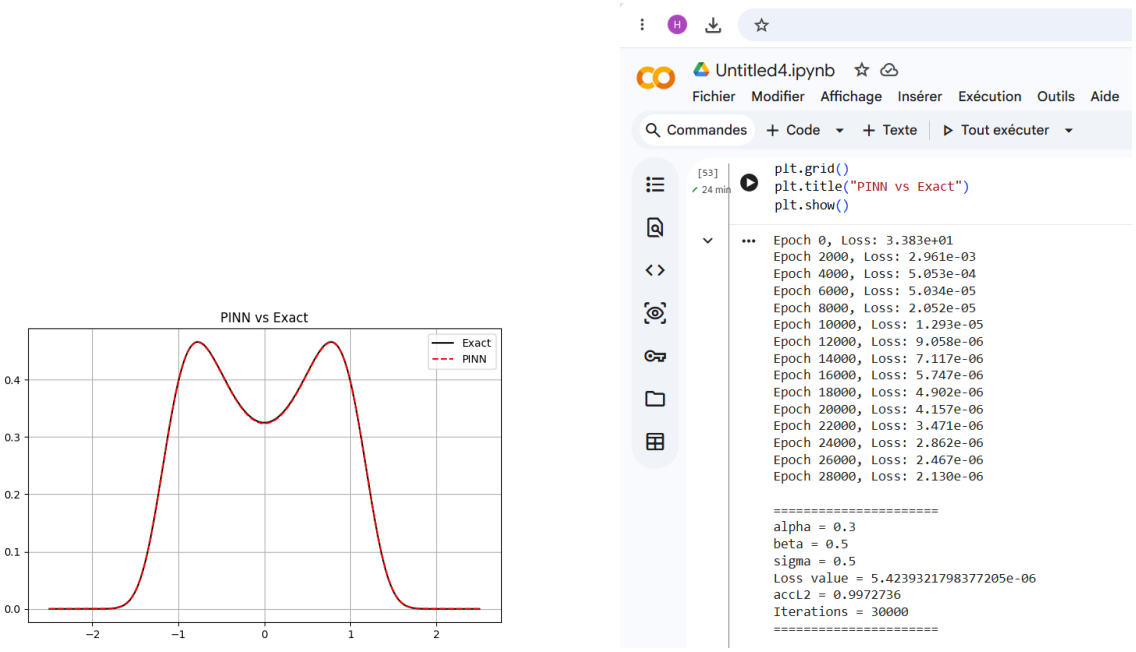
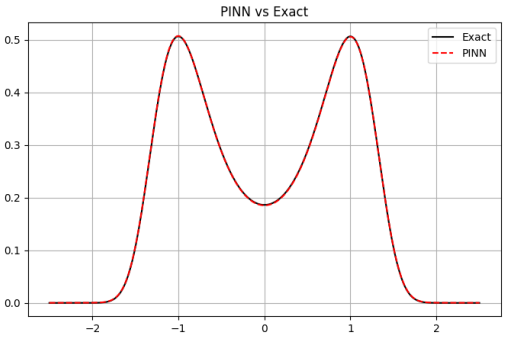


Figure 3.5: PINN Fokker-Planck equation vs Exact solution method with $\alpha = 0.3$



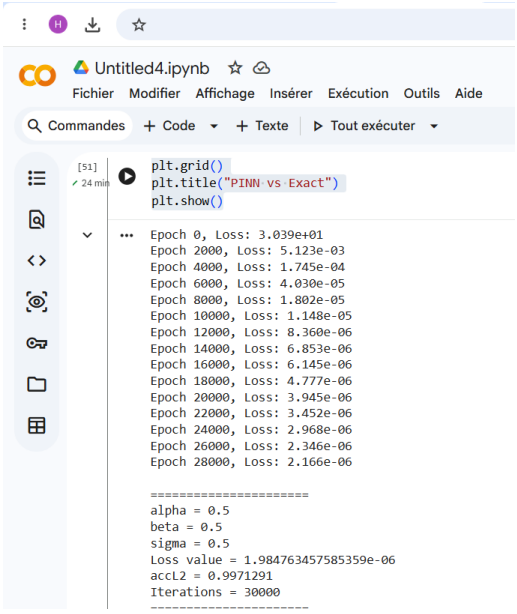
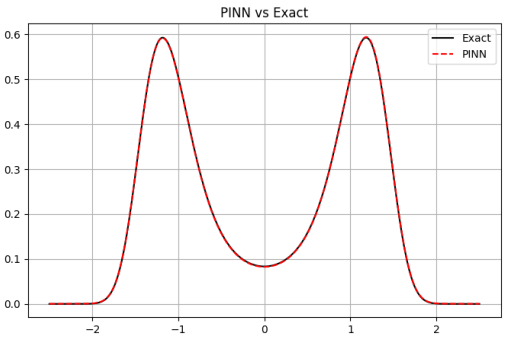


Figure 3.6: PINN Fokker-Planck equation vs Exact solution method with $\alpha = 0.5$



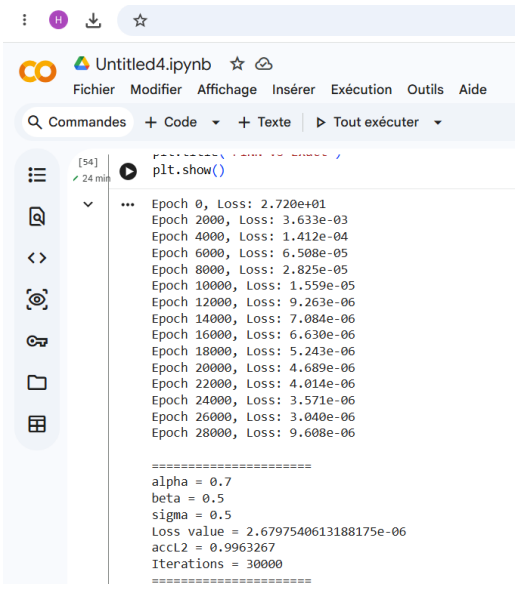


Figure 3.7: PINN Fokker-Planck equation vs Exact solution method with $\alpha = 0.7$

Observation: Figures 3.5,3.6,3.7: present the comparison between the DL-FP solution and the exact solution for different values of the parameter α. In all cases, the neural network approximation (red dashed line) closely matches the exact solution (black solid line). The excellent agreement between the two solutions demonstrates the accuracy and effectiveness of the DL-FP method in solving the Fokker–Planck equation for different parameter configurations.

Table3.1 presents the numerical results of the DL-FP method. The loss value represents the residual of the Fokker–Planck equation, while acc_{L_2} measures the relative

Table 3.1: The accuracy of the DL-FP method for equation 3.16.

Parameters	Loss value	accL2	Iterations
$\alpha = 0.3,\ \beta = 0.5,\ \sigma = 0.5$	5.42×10^{-6}	0.9972	3×10^4
$\alpha = 0.5,\ \beta = 0.5,\ \sigma = 0.5$	1.98×10^{-6}	0.9971	3×10^4
$\alpha = 0.7,\ \beta = 0.5,\ \sigma = 0.5$	2.69×10^{-6}	0.9963	3×10^4

L^2 accuracy between the PINN solution and the exact solution:

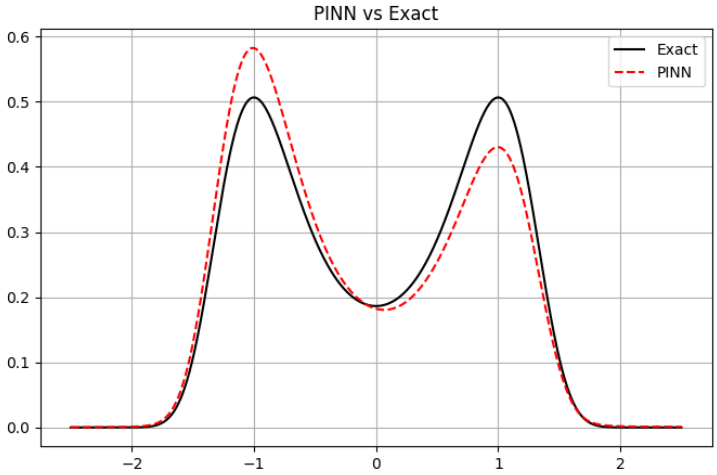
$$acc_{L_2} = 1 - \frac{\|\hat{p} - p\|_2}{\|p\|}.$$

The number of iterations is fixed at 3×10^4 for all experiments.
The results indicate that the method maintains high accuracy and stable convergence for all tested configurations. These results indicate that the DL-FP method is highly effective in solving the FP equation.

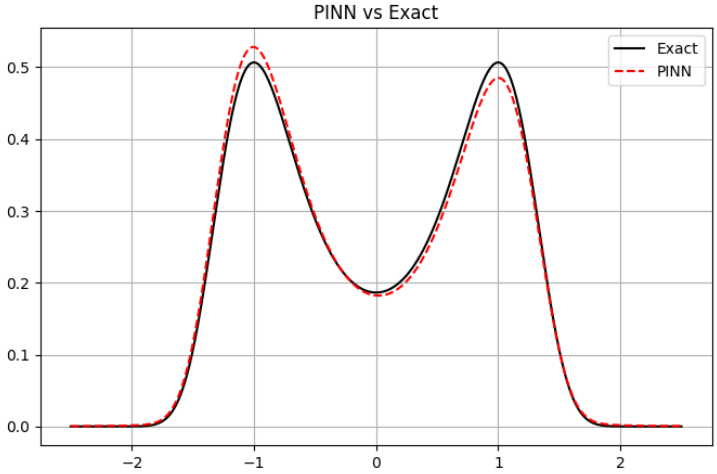
The effect of neural network structure on model performance

To further investigate the influence of neural network architecture on model performance, an additional experiment was conducted using the same problem of Example 2, with the same training procedure, optimizer, learning rate, and activation functions. The objective was to analyze how architectural modifications affect the accuracy and convergence behavior of the network.
Two separate cases were considered. In the first case, the number of hidden layers was varied while keeping the number of neurons in each layer and all other parameters unchanged. This experiment aimed to evaluate the impact of network depth on the learning process and prediction accuracy.
In the second case, the number of neurons in the hidden layers was modified while maintaining the same number of hidden layers and preserving all remaining training settings. This analysis was performed to study the effect of network width on the model's ability to approximate the target function.
The results obtained from these experiments provide valuable insights into the relationship between network architecture and model performance, highlighting the importance of selecting an appropriate number of hidden layers and neurons when designing deep neural networks.

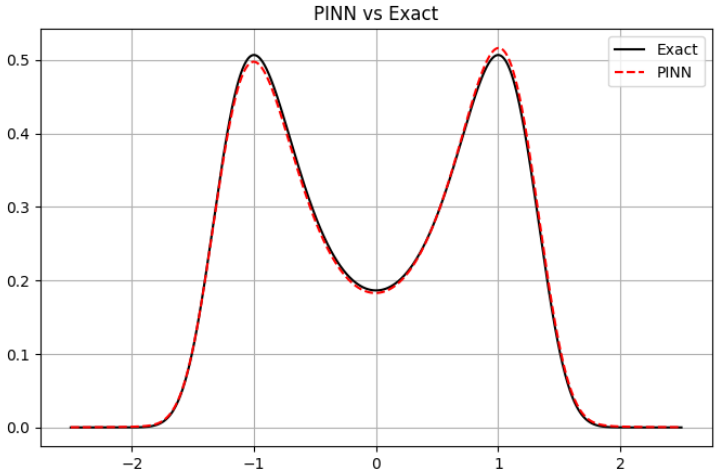
Effect of the Number of Hidden Layers



(a) Predicted Solution with 2Hidden Layers



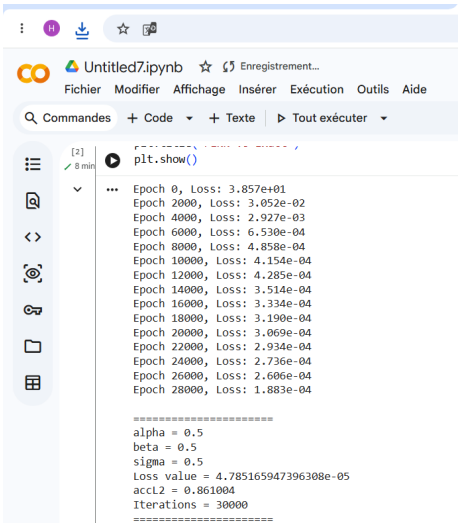
(b) Predicted Solution with 3 Hidden Layers



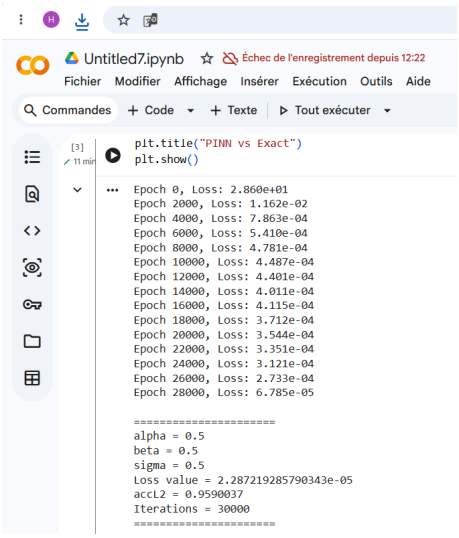
(c) Predicted Solution with 4 Hidden Layers

Figure 3.8: Comparison of the effect of hidden layer on the solution.

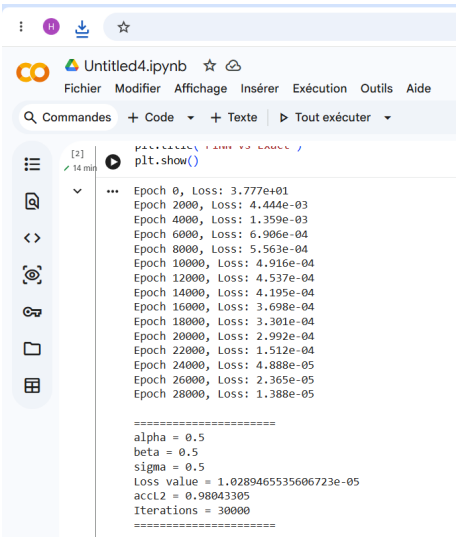
Observation: Figure 3.8 shows the evolution of the solution for different numbers of hidden layers. It can be observed that the network with 4 hidden layers fits well with the exact solution compared to the network with 2 hidden layers. However, adding more layers does not necessarily lead to further improvement and may increase the training complexity.
These results indicate that an appropriate network depth is essential for obtaining accurate solutions while maintaining computational efficiency.



(a) The accuracy of the DL-FP with 2 hidden layers



(b) The accuracy of the DL-FP with 3 hidden layers



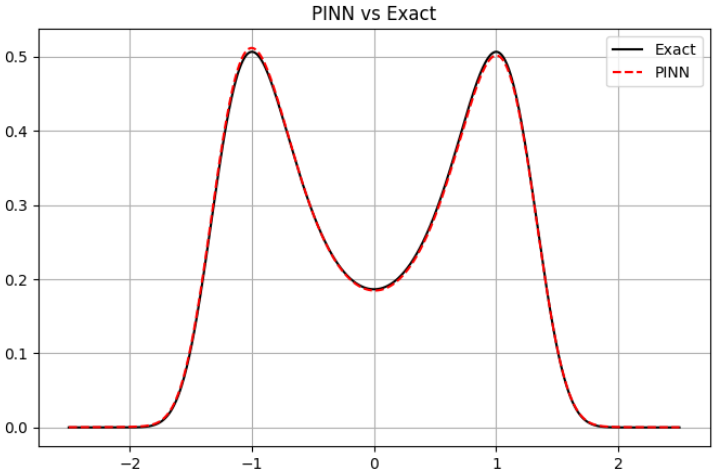
(c) The accuracy of the DL-FP with 4 hidden layers

Figure 3.9: Comparison of The accuracy of the DL-FP for different numbers of hidden layers

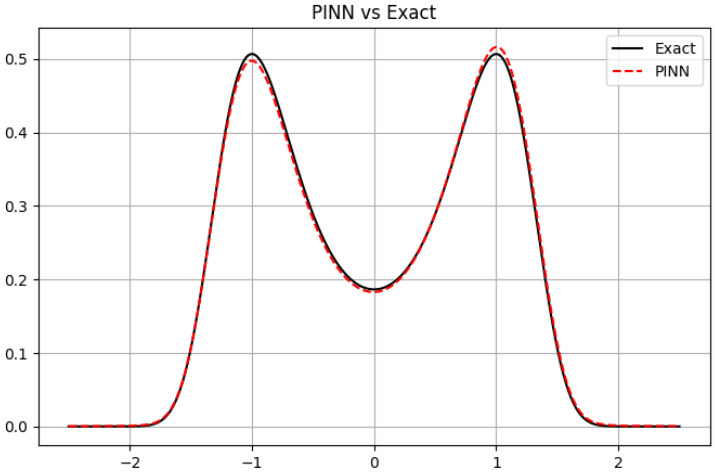
Observation: The effect of the number of hidden layers on the accuracy of the

DL-FP method is illustrated in Figure 3.9. The results indicate that the network depth plays an important role in the approximation process. Increasing the number of hidden layers improves the learning capability of the model and reduces the prediction error. However, excessive depth does not necessarily lead to significant improvements and may increase the computational cost. Therefore, an appropriate number of hidden layers should be selected to achieve a balance between accuracy and efficiency.

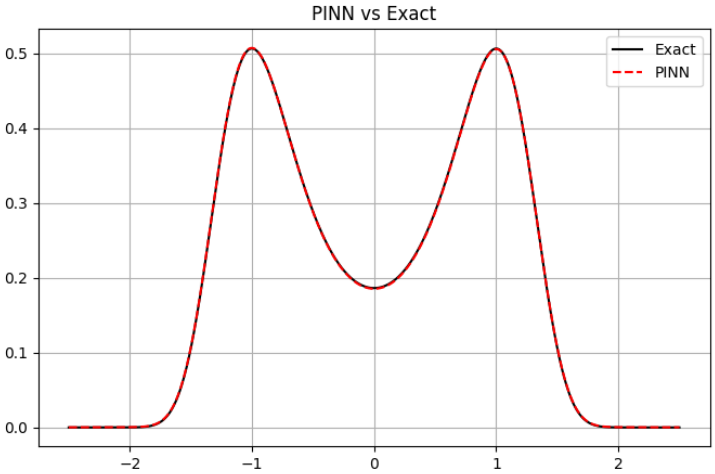
Effect of the Number of Neurons



(a) Predicted Solution with 20 Neurons



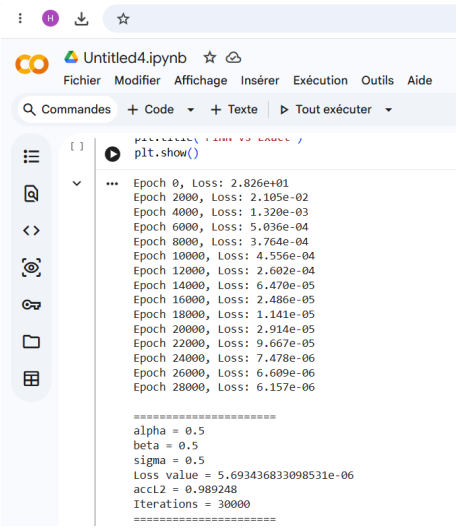
(b) Predicted Solution with 30 Neurons



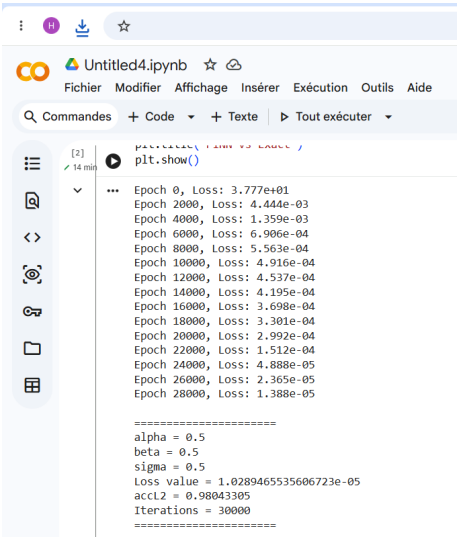
(c) Predicted Solution with 64 Neurons

Figure 3.10: Effect of the number of neurons on the neural network performance.

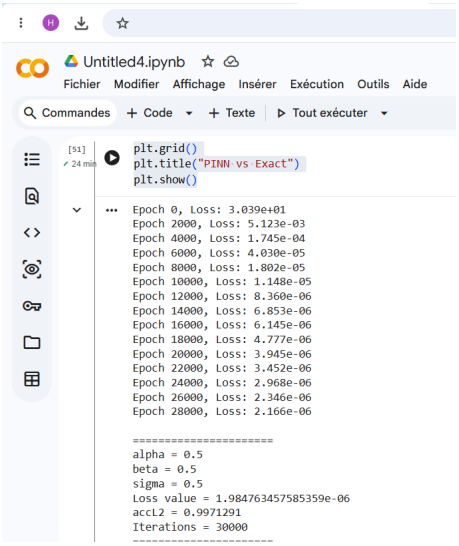
Observation: Figure 3.10 compares the performance of networks with different numbers of neurons per hidden layer. The results show that increasing the number of neurons enhances the network's representation capability and reduces the approximation error. Nevertheless, beyond a certain number of neurons, the improvement becomes marginal while the computational cost increases. Therefore, selecting an optimal number of neurons is important to balance accuracy and efficiency.



(a) The accuracy of the DL-FP with 20 Neurons



(b) The accuracy of the DL-FP with 30 Neurons



(c) The accuracy of the DL-FP with 64 Neurons

Figure 3.11: Comparison of The accuracy of the DL-FP for different numbers of neurons

Observation: Figure 3.11 present the influence of the number of neurons on the accuracy of the DL-FP method. It can be observed that increasing the number

of neurons enhances the representation capacity of the neural network, leading to more accurate approximations and lower error values. Nevertheless, beyond a certain point, the improvement becomes less significant while the model complexity increases. Consequently, the number of neurons should be chosen carefully to obtain accurate results without unnecessary computational effort.

Remark 1. *The codes execution took from 15 minutes to 25 minutes to complete the full simulations, according to the complexity of the parameters in each case. They are implemented on an DELL PC, Intel Core i5 vPro 8th Gen. Processor.*

3.6 Advantages and limitations of PINNs

3.6.1 Advantages:

- **Mesh-free:** One of the main advantages of PINNs is that they don't require domain segmentation, unlike traditional methods such as finite differences or finite elements. This removes the need for the often tedious and computationally heavy task of mesh generation.
- **No data required:** The network learns solely from the PDE itself, along with initial and boundary conditions.
- **Single model for parametric PDEs:** A PDE often depends on some parameters, for example, diffusion coefficient, force term, boundary conditions, etc... So instead of training a separate neural network for each set of parameters, PINNs can train one single model that can work for many different parameter values. This capability allows for near-instant predictions across different scenarios, i.e., we don't need to solve the PDE again numerically, we just input the new parameter values into the network, and it immediately outputs an approximate solution. So predictions become very fast. This approach is performed without the overhead of retraining the model from scratch. In fact, normally, if we change parameters, we would need to retrain a model (slow and expensive), but here, one trained PINN is reused, and no retraining needed for new parameter values (within the learned range). So, instead of solving a PDE again and again for different parameters, we train one smart model that learns the whole family of solutions and can instantly predict them.
- **Continuous solution:** PINNs provide a smooth, differentiable solution at any point (x, t) in the domain, not just at grid points such as finite difference or finite element method that require performing grid interpolation after the computations.

3.6.2 Limitations:

- **Sensitivity to hyperparameters:** In PINNs, small changes in these settings can lead to bad case where the model does not converge, solution becomes physically incorrect, the PDE residual stays large, loss function training oscillates or diverges. So, different tests are needed to guarantee smooth convergence, and accurate approximation of PDE solution.

- **High training cost:** Even though inference is fast, training PINNs can be very slow in computing time because it requires many collocation points especially when the domain is irregular with complex shapes. In some cases, classical numerical solvers are faster.

- **Unbalanced loss terms:** A common task with the PINNs loss function is the interaction between the different components, the PDE residue, the initial and boundary conditions. One term can dominate others, and choosing weights is non-trivial which yields to poor balancing and so wrong solutions.

- **Error verification and convergence issues:** PINNs networks often fail to converge, especially for complex problems. There is no convergence and stability theoretical studies. Unlike traditional numerical methods, which rely on error estimates $O(\Delta t)$, PINNs networks lack a general convergence theory. This makes it difficult to check and guarantee the accuracy of their solutions.

Chapter 4

Deep Learning-based FBSNNs methods

Developing algorithms for solving high-dimensional partial differential equations (PDEs) has been an exceedingly difficult task for a long time, due to the notoriously difficult problem known as the "curse of dimensionality" [15]. To overcome this limitation, many works [14, 3, 2] introduce a deep learning-based approach that can handle general high dimensional PDEs. To this end, the PDEs are reformulated using forward-backward stochastic differential equations. The key aspect of this development is the deep connection between forward-backward stochastic differential equations (BSDEs), and partial differential equations (PDEs) through stochastic representations such as the Feynman–Kac formula [8]. This reformulation has led to the development of modern computational frameworks such as deep backward-forward stochastic neural networks (FBSNNs) method [28, 11]. This method enables the use of stochastic simulation combined with neural network approximation to compute solutions that are otherwise difficult to find using classical techniques. Neural networks provide mesh-free approximation capabilities, making them especially suitable for high-dimensional systems where grid-based methods fail. These approaches not only overcome the limitations of discretization-based methods but also open new directions for solving high-dimensional problems efficiently [28, 3]. In this chapter, we present the basic idea of this method, and how it works from time discretization to neural network approximation and loss function minimization. We introduce numerical results on the nonlinear Black–Scholes equation. We observe that this method is quite effective in terms of accuracy and cost. We discuss some advantages and limitations of the method at the end of the chapter.

4.1 The Kolmogorov PDE and Forward-Backwad SDE

Let us consider coupled forward-backward stochastic differential equations of the general form:

$$\begin{aligned} dX_t &= \mu(t, X_t, Y_t, Z_t)\, dt + \sigma(t, X_t, Y_t)\, dW_t, \qquad t \in [0, T], \\ X_0 &= \xi, \\ dY_t &= \phi(t, X_t, Y_t, Z_t)\, dt + Z_t^T \sigma(t, X_t, Y_t)\, dW_t, \qquad t \in [0, T), \\ Y_T &= g(X_T). \end{aligned} \tag{4.1}$$

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 [8] that the coupled forward-backward stochastic differential equations (4.1) are related to quasi-linear Kolmogorov partial differential equation of the form:

$$u_t = f(t, x, u, \nabla_x u, \nabla^2_{xx} u), \tag{4.2}$$

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

$$f(t, x, y, z, \gamma) = \phi(t, x, y, z) - \mu(t, x, y, z)^T z - \frac{1}{2} \operatorname{Tr}\left[\sigma(t, x, y)\sigma(t, x, y)^T \gamma\right]. \tag{4.3}$$

Here, $\nabla_x u$ and $\nabla^2_{xx} u$, denote the gradient vector and the Hessian matrix of u, respectively. $\operatorname{Tr}(\cdot)$ is the trace operator. In particular, it follows from Ito's formula [8] that solutions of the above equations are related according to:

$$Y_t = u(t, X_t), \qquad Z_t = \nabla_x u(t, X_t). \tag{4.4}$$

4.2 How Deep FBSNNs works for the Kolmogorov equation

This section present the Deep FBSNNs method introduced by [14, 28, 3] for solving high-dimensional Kolmogorov partial differential equations. The key idea is to approximate the unknown PDE solution $u(t, x)$ using a neural network and train this network by enforcing the dynamics of the associated FBSDE rather than directly enforcing the PDE residual as in Physics-Informed Neural Networks (PINNs). The principle is instead of learning only the initial value $u(0, x)$, the method learns the

entire unknown solution $u(t,x)$ over the space-time domain by using "deep stochastic neural network". We get the process Y_t from the approximation of $u(t,x)$, and the process Z_t from application of automatic differentiation on the approximation of $u(t,x)$ to get its gradient.

4.2.1 Step 1: Reformulate the PDE as an FBSDE System

Rather than immediately dealing with the Kolmogorov PDE, the basic concept here is to modify it properly in order to get a system of two forward-backward stochastic equations. To simplify, we can discuss the case of uncoupled forward-backward stochastic differential equations of the form:

$$\begin{cases} dX_t = \mu(t, X_t)\, dt + \sigma(t, X_t)\, dW_t, \\ X_0 = \xi, \\ dY_t = \phi(t, X_t, Y_t, Z_t)\, dt + Z_t^T \sigma(t, X_t)\, dW_t, \\ Y_T = g(X_T). \end{cases} \tag{4.5}$$

These equations are related to the Kolmogorov semilinear and parabolic PDE:

$$u_t = \phi(t, x, \nabla_x u, \nabla^2_{xx} u) - \mu(t,x)^T \nabla_x u - \frac{1}{2} \operatorname{Tr}\left[\sigma(t,x)\sigma(t,x)^T \nabla^2_{xx} u\right]. \tag{4.6}$$

The relation between the tow equations is:

$$Y_t = u(t, X_t), \qquad Z_t = \nabla_x u(t, X_t). \tag{4.7}$$

4.2.2 Step 2: Time Discretization of the FBSDE

This method divises an algorithm to compute Y_t and Z_t for each time $t > 0$. Euler-Maruyama scheme is employed to discretize the forward equation in (4.5) to update the process X_t:

$$X^{n+1} \approx X^n + \mu(t^n, X^n)\, \Delta t + \sigma(t^n, X^n)\, \Delta W^n, \tag{4.8}$$

for $n = 0, 1, \ldots, N-1$, where $\Delta t := t^{n+1} - t^n = T/N$ and $\Delta W^n \sim \mathcal{N}(0, \Delta t)$ is a random variable with mean 0 and standard deviation $\sqrt{\Delta t}$.
The next step is to use X^n to approximate the $u(t^n, X^n)$ for $n = 1, \ldots, N-1$ at time steps t^n by $N-1$ stochastic neural networks. We denote it $u_\theta(t^n, X^n)$. This enables to get an approximation for $Y^n \approx u_\theta(t^n, X^n)$, and for $Z^n \approx \nabla u_\theta(t^n, X^n)$ by automatic differentiation at time t^n and at the spatial point X^n. So, no additional neural network is required for Z^n. Both quantities are directly computed from the neural network.

4.2.3 Step 3: Neural Network Approximation and loss function

Parameters $\theta = (W, biais)$ of the stochastic neural network can be learned by minimizing the following loss function related to the backward equation of the process Y_t which is first discretized by Euler-Maruyama scheme:

$$Y^{n+1} \approx Y^n + \phi(t^n, X^n, Y^n, Z^n)\,\Delta t + (Z^n)^T \sigma(t^n, X^n)\,\Delta W^n. \tag{4.9}$$

Equation (4.9) is used to compute the loss as:

$$\mathcal{L}(\theta) = \sum_{m=1}^{M} \sum_{n=0}^{N-1} \left| Y_m^{n+1} - Y_m^n - \Phi_m^n\,\Delta t - (Z_m^n)^T \Sigma_m^n\,\Delta W_m^n \right|^2 + \sum_{m=1}^{M} \left| Y_m^N - g(X_m^N) \right|^2, \tag{4.10}$$

which tries to match the discretized equation (4.9) of Y_t and to force the satisfaction of the terminal condition. Here, M corresponds to different realizations of the underlying Brownian motion, and

$$\Phi_m^n := \phi(t^n, X_m^n, Y_m^n, Z_m^n) \quad \text{and} \quad \Sigma_m^n := \sigma(t^n, X_m^n)\,.$$

The subscript m corresponds to the m-th realization of the underlying Brownian motion while the superscript n corresponds to time t^n. We recall that from equation (4.7), we have:

$$Y_m^n = u_\theta(t^n, X_m^n), \qquad Z_m^n = \nabla u_\theta(t^n, X_m^n),$$

and consequently the loss in equation (4.10) is a function of the parameters of the neural network $u_\theta(t, x)$. The parameters θ are optimized using Adam method as described in the previous chapter. At each iteration:

1. Sample new Brownian trajectories.
2. Simulate the forward process X^n.
3. Compute an approximation $u_\theta(t^n, X^n)$ by stochastic neural network.
4. Compute $Y_n = u_\theta(t_n, X_n)$.
5. Compute $Z_n = \nabla_x u_\theta(t_n, X_n)$.
6. Evaluate the loss function.
7. Backpropagate the gradient of the loss with respect to the network parameters θ.

8. Minimize the loss by Adam optimizer.

9. Update θ to compute again $u_\theta(t^n, X^n)$ and so on..

4.3 Numerical simulations and comparative study

This section introduces numerical results on the nonlinear Black–Scholes equation. This example is inspired from [28]. The values of the parameters are reduced compared to those used in [28] in order to be able to run the Python code in a reasonable elapsed time. Also "PyTorch" library for deep learning neural network is used due to its simplicity and ease of implementation over "TensorFlow" used in [28]. The results suggest that this method is quite effective in terms of accuracy and cost and can be applied to a wide range of forward-backward problems.

Black-Scholes-Barenblatt Equation

Consider the nonlinear Black–Scholes–Barenblatt equation FBSDE:

$$\begin{cases} dX_t = rX_t dt + \sigma X_t dW_t, \\ X_0 = \xi, \\ dY_t = r(Y_t - Z_t X_t)dt + \sigma Z_t X_t dW_t, \\ Y_T = g(X_T), \end{cases} \tag{4.11}$$

where $T = 1$, $\sigma = 0.4$, $r = 0.05$, $\xi = 1$, $g(x) = x^2$. The equations are related to the Black-Scholes-Barenblatt equation PDE:

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\sigma^2 x^2 \partial_{xx} u + r(u - x\partial_x u) - rx\partial_x u \qquad (t, x) \in [0, T) \times \mathbb{R}, \tag{4.12}$$

subject to the terminal condition:

$$u(T, x) = x^2. \tag{4.13}$$

The exact solution is given by:

$$u(t, x) = \exp\left((3r + \sigma^2)(T - t)\right) g(x), \tag{4.14}$$

which is used to test the accuracy of the method. We approximate the unknown solution $u(t, x)$ by 5 hidden layers in the neural network with 128 neurons per hidden layer. We discretized the time domain $[0, T]$ into $N = 50$ equally spaced intervals. To minimize the loss function, we called the Adam optimizer with mini-batches of

size 100 (i.e., 100 realizations of the underlying Brownian motion), the results are reported in figure 4.1.

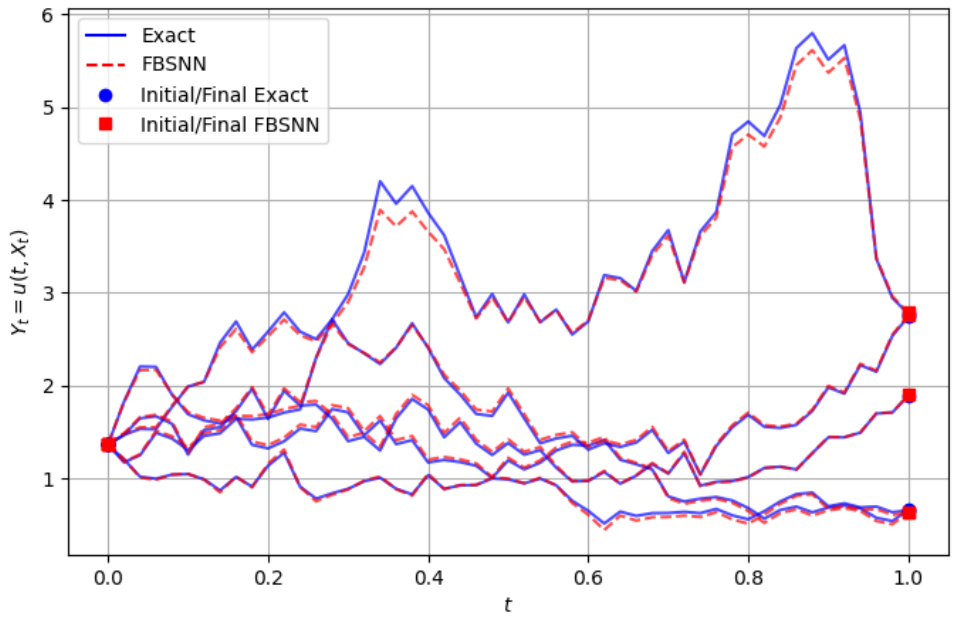


Figure 4.1: Evaluations of the learned solution $Y_t = u(t, X_t)$ at representative realizations of the underlying process X_t for different paths.

Unlike the state of the art algorithms [15, 11], which can only approximate $Y_0 = u(0, X_0)$ at time 0 and at the initial spatial point $X_0 = \xi$, this method is capable of approximating the entire solution function $u(t, x)$ in a single round of training as demonstrated in figure 4.1.

The Python code is provided below for this example:

```
import torch
import torch.nn as nn
import torch.optim as optim

# ===================================================
# Parameters
# ===================================================

D = 1
T = 1.0

N = 50                  # time steps
```

```
M = 100                # batch size

r = 0.05
sigma = 0.4

dt = T / N
sqrt_dt = dt**0.5

device = "cuda" if torch.cuda.is_available() else "cpu"

# ================================================
# Neural Network u_theta(t,x)
# ================================================

class Net(nn.Module):

    def __init__(self, D):

        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(D + 1, 128),
            nn.Tanh(),

            nn.Linear(128, 128),
            nn.Tanh(),

            nn.Linear(128, 128),
            nn.Tanh(),

            nn.Linear(128, 128),
            nn.Tanh(),

            nn.Linear(128, 128),
            nn.Tanh(),

            nn.Linear(128, 1)
        )

    def forward(self, t, x):

```

```
        inp = torch.cat([t, x], dim=1)

        return self.net(inp)


model = Net(D).to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3)

# ======================================================
# Exact solution
# ======================================================

def exact_solution(t, x):

    norm2 = torch.sum(
        x**2,
        dim=1,
        keepdim=True
    )

    return torch.exp(
        (3r + sigma**2)*(T - t)
    ) * norm2

# ======================================================
# One FBSNN loss evaluation
# ======================================================

def loss_function():

    # ------------------------------------------------
    # Initial condition
    # ------------------------------------------------

    X = torch.ones(M, D, device=device)

    loss = 0.0

    # ------------------------------------------------
    # Time stepping
```

```
    # -----------------------------------------------

    for n in range(N):

        t_n = torch.full(
            (M,1),
            n*dt,
            device=device
        )

        t_np1 = torch.full(
            (M,1),
            (n+1)*dt,
            device=device
        )

        X.requires_grad_(True)

        # -----------------------------------------------
        # Y_n = u_theta(t_n,X_n)
        # -----------------------------------------------

        Y_n = model(t_n, X)

        # -----------------------------------------------
        # Z_n = grad_x u_theta
        # -----------------------------------------------

        Z_n = torch.autograd.grad(
            Y_n.sum(),
            X,
            create_graph=True
        )[0]

        # -----------------------------------------------
        # Brownian increment
        # -----------------------------------------------

        dW = sqrt_dt * torch.randn(
            M,
            D,
```

```
            device=device
        )

        # ----------------------------------------
        # Forward SDE
        # dX = rX dt + sigma XdW
        # ----------------------------------------

        X_next = (
            X
            + r*X*dt
            + sigma*X*dW
        )

        # ----------------------------------------
        # Y_{n+1}
        # ----------------------------------------

        Y_np1 = model(t_np1, X_next)

        # ----------------------------------------
        # BSDE driver
        # f = r(Y - X^T Z)
        # ----------------------------------------

        f = r * (
            Y_n
            - torch.sum(X*Z_n,
                        dim=1,
                        keepdim=True)
        )

        # ----------------------------------------
        # sigma(X)dW term
        # ----------------------------------------

        diffusion = sigma * torch.sum(
            Z_n * X * dW,
            dim=1,
            keepdim=True
        )
```

```
177
178         # ----------------------------------------
179         # Residual
180         # ----------------------------------------
181
182         residual = (
183             Y_np1
184             - Y_n
185             - f*dt
186             - diffusion
187         )
188
189         loss += torch.mean(residual**2)
190
191         X = X_next.detach()
192
193     # ------------------------------------------------
194     # Terminal condition
195     # ------------------------------------------------
196
197     t_T = torch.full(
198         (M,1),
199         T,
200         device=device
201     )
202
203     Y_T = model(t_T, X)
204
205     g = torch.sum(
206         X**2,
207         dim=1,
208         keepdim=True
209     )
210
211     loss += torch.mean(
212         (Y_T - g)**2
213     )
214
215     return loss
216 loss_history = []
217 pred_history = []
```

```
exact_history = []

# ======================================================
# Training
# ======================================================


for it in range(2000):

    optimizer.zero_grad()

    loss = loss_function()

    loss.backward()

    optimizer.step()

    with torch.no_grad():

        X0 = torch.ones(1, D, device=device)

        t0 = torch.zeros(1, 1, device=device)

        pred = model(t0, X0).item()

        exact = exact_solution(t0, X0).item()

    loss_history.append(loss.item())
    pred_history.append(pred)
    exact_history.append(exact)

    if it % 100 == 0:

        print(
            f"Iter {it:5d} "
            f"Loss = {loss.item():.4e}"
        )



import matplotlib.pyplot as plt
```

```
259
260 plt.figure(figsize=(8,5))
261
262 for m in range(5):
263
264     times = []
265     predicted = []
266     exact = []
267
268     X = torch.ones(1, D, device=device)
269
270     for n in range(N+1):
271
272         t = n*dt
273
274         t_tensor = torch.tensor(
275             [[t]],
276             dtype=torch.float32,
277             device=device
278         )
279
280         with torch.no_grad():
281
282             pred = model(t_tensor, X)
283
284             ex = exact_solution(t_tensor, X)
285
286         times.append(t)
287         predicted.append(pred.item())
288         exact.append(ex.item())
289
290         if n < N:
291
292             dW = sqrt_dt * torch.randn(
293                 1, D, device=device
294             )
295
296             X = (
297                 X
298                 + r*X*dt
299                 + sigma*X*dW
```

```
            )

    # ==========================
    # Paths
    # ==========================

    plt.plot(
        times,
        exact,
        'b-',
        alpha=0.7
    )

    plt.plot(
        times,
        predicted,
        'r--',
        alpha=0.7
    )

    # ==========================
    # Initial points
    # ==========================

    plt.plot(
        times[0],
        exact[0],
        'bo',
        markersize=6
    )

    plt.plot(
        times[0],
        predicted[0],
        'rs',
        markersize=6
    )

    # ==========================
    # Final points
    # ==========================
```

```
341
342     plt.plot(
343         times[-1],
344         exact[-1],
345         'bo',
346         markersize=6
347     )
348
349     plt.plot(
350         times[-1],
351         predicted[-1],
352         'rs',
353         markersize=6
354     )
355
356 plt.xlabel(r'$t$')
357 plt.ylabel(r'$Y_t=u(t,X_t)$')
358
359 plt.plot([], [], 'b-', label='Exact')
360 plt.plot([], [], 'r--', label='FBSNN')
361 plt.plot([], [], 'bo', label='Initial/Final Exact')
362 plt.plot([], [], 'rs', label='Initial/Final FBSNN')
363
364 plt.legend()
365 plt.grid(True)
366 plt.show()
```

Listing 4.1: Python code for Black-Scholes-Barenblatt equation.

Remark 2. *The code execution took around 25 minutes to complete the full simulations on an DELL PC, Intel Core i5 vPro 8th Gen. Processor.*

4.4 Advantages and limitations of the FBSNNs method

4.4.1 Advantages

The deep FBSNN has an important advantages:

- **Overcoming the Curse of Dimensionality:** One of the main forces of this method is it could control high dimensional partial differential equations, where classical grid-based absolute methods fail. Traditional numerical meth-

ods require a grid of N^d points, which is computationally very hard when the values of points N and dimension d are large.

- **Mesh-Free Formulation:** Unlike classical methods, Deep FBSNNs method does not need to discretize the spatial domain. Instead, it relies only on simulating the stochastic paths via the loss function minimization and stochastic neural network simulation.

- **Provides Both Solution and Sensitivities:** Another important feature is that the method does not only approximate the solution $Y_0 \approx u(0, X_0)$ and $Z_0 = \nabla_x u(0, X_0)$, but also learns the entire processes Y_t and Z_t at each time $t > 0$. This is particularly useful in applications such as financial hedging, sensitivity analysis, and stochastic optimal control.

- **No data required:** The network learns only from FBSDE and does not need real-world measurements.

4.4.2 Limitations

Despite its advantages, the Deep FBSNNs method has several important limitations that restrict its applicability in some cases.

- **Requirement of FBSDE Formulation:** The technique is based on the idea that the real partial differential equation can be reformulated as forward-backward stochastic differential equation using Feynman−Kac formula. Not all PDEs assume any such probability connection to SDEs. This technique cannot implement any equation directly by description.

- **High Computational Cost:** Training is computationally expensive. At each iteration, the rule set simulates a large range of stochastic paths and back-propagated over time. This is particularly challenging for long time horizons of PDEs.

- **Non-Convex Optimization:** The non-convexity of the loss function especially for complex problems may not guarantee the convergence to the global minimum and the method may additionally rely on initialization and hyperparameter options to converge to the nearest local minimum.

- **Sensitivity to Hyperparameters:** The performance of the method is incredibly sensitive to hyperparameter values, including learning rates, network architecture and parameters, that can additionally seriously degrade accuracy of the method.

- **Dependence on Known Forward Dynamics:** The method assumes that the coefficients of the forward SDE (drift μ and diffusion σ) are known in advance, since they are used to simulate the stochastic paths. If these dynamics are unknown, alternative approaches are required to deal with this issue.
- **Memory Consumption:** Backpropagation through time requires storing all intermediate values of $(X_{t_n}, Y_{t_n}, Z_{t_n})$ for each time step. When both the number of time steps and the number of simulated paths are large, memory usage can become very saturated.
- **Expensive training:** Training can take hours or days, especially for high-dimensional problems or large networks.
- **Limited accuracy:** The solution is approximate, whereas traditional methods offer an accuracy that we can control by minimizing the step time discretization Δt.
- **No convergence guarantees:** There is no guarantee that training will converge to a good solution, especially for complex FBSDEs.

Conclusion

In this work, we have explored the connection between stochastic differential equations (SDEs) and modern deep learning techniques, with a particular focus on Physics-Informed Neural Networks (PINNs) and Forward–Backward Stochastic Neural Networks (FBSNNs).

We began by recalling the fundamental concepts of stochastic differential equations and Brownian motion, as well as the existence and uniqueness of solutions. We also discussed classical numerical methods and their limitations, especially when dealing with high-dimensional or complex stochastic systems.

Then, we introduced the basic principles of machine learning and deep learning, highlighting different neural network architectures, training procedures, and key concepts such as activation functions, loss functions, and optimization algorithms. This section helped establish the necessary background for understanding deep learning-based approaches to differential equations.

After that, we presented Physics-Informed Neural Networks (PINNs) as a modern and efficient framework for solving partial differential equations, in particular the Fokker–Planck equation that governs the probability density function of the solution of the stochastic differential equation (SDE). We explained how neural networks can be trained by incorporating physical laws directly into the loss function through automatic differentiation, avoiding the need for traditional discretization methods.

Finally, we studied Deep FBSNNs methods, which rely on the reformulation of partial differential equations into forward–backward stochastic differential systems. This approach provides an alternative and powerful numerical tool, especially for high-dimensional problems, but also comes with challenges related to stability and computational cost.

Overall, this work shows that deep learning techniques offer a promising alternative to classical numerical methods for solving stochastic and partial differential equa-

tions. However, despite their advantages, these methods still present limitations that require further research, particularly in terms of efficiency, convergence guarantees, and interpretability.

Due to the material limitations that we encountered during the preparation of this thesis, only simple models and examples were treated. Future works may focus on testing the performance of these models (an other models available in the literature) as well as their applicability to more complex high dimensional stochastic systems in real life applications such as finance, physics, or engineering, etc...

Bibliography

[1] M. Bayram, T. Partal and G. O. Buyukoz, *Numerical Methods for Simulation of Stochastic Differential Equations*, Advances in Difference Equations, Article No. 17, 2018.

[2] C. Beck, S. Becker, P. Cheridito, et al. *Deep learning based numerical approximation algorithms for stochastic partial differential equations*, arXiv:2012.01194v2, 2025.

[3] C. Beck, S. Becker, P. Grohs, et al. *Solving the Kolmogorov PDE by Means of Deep Learning*, Journal of Scientific Computing, 88:73, 2021.

[4] R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[6] O. Campesato, Artificial intelligence, Machine Learning and Deep Learning, Mercury Learning and Information, 2020.

[7] X. Chen, L. Yang, J. Duan and G. E. Karniadakis, *Solving Inverse Stochastic Problems from Discrete Particle Observations Using the Fokker–Planck Equation and Physics-Informed Neural Networks*, arXiv preprint arXiv:2008.10653, 2020.

[8] P. Cheridito, H. M. Soner, N. Touzi, N. Victoir, *Second-order back- ward stochastic differential equations and fully nonlinear parabolic pdes*, Communications on Pure and Applied Mathematics 60:1081–1110, 2007.

[9] S. Cuomo, V. Schiano Di Cola, F. Giampaolo, G. Rozza, M. Raissi and F. Piccialli, *Scientific Machine Learning Through Physics-Informed Neural Networks: Where We Are and What's Next*, Journal of Scientific Computing, Vol. 92, Article 88, 2022.

[10] W. E, *A Proposal on Machine Learning via Dynamical Systems*, Communications in Mathematics and Statistics, Vol. 5, No. 1, pp. 1–11, 2017.

[11] W. E, J. Han, A. Jentzen. *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations*, arXiv:1706.04702v1, 2017.

[12] J. L. Elman, Finding Structure in Time, *Cognitive Science*, Vol. 14, No. 2, pp. 179–211, 1990.

[13] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016.

[14] J. Han, A. Jentzen and W. E, *Solving High-Dimensional Partial Differential Equations Using Deep Learning*, *Proceedings of the National Academy of Sciences*, Vol. 115, No. 34, pp. 8505–8510, 2018.

[15] 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.

[16] D. J. Higham, An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations, *SIAM Review*, Vol. 43, No. 3, pp. 525–546, 2001.

[17] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

[18] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang and L. Yang, Physics-Informed Machine Learning, *Nature Reviews Physics*, Vol. 3, No. 6, pp. 422–440, 2021.

[19] D.P. Kingma, J. Ba, *Adam: A Method for Stochastic Optimization*, Proceedings of the International Conference on Learning Representations (ICLR), 2015. arXiv:1412.6980

[20] S. Kim, Mathematical Foundations of Machine Learning, Department of Mathematics and Statistics, Mississippi State University Mississippi State, MS 39762 USA, 2025.

[21] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, Springer-Verlag, Berlin, 1992.

[22] Y. LeCun, Y. Bengio and G. Hinton, Deep Learning, *Nature*, Vol. 521, No. 7553, pp. 436–444, 2015.

[23] L. Lu, X. Meng, Z. Mao and G. E. Karniadakis, DeepXDE: A Deep Learning Library for Solving Differential Equations, *SIAM Review*, Vol. 63, No. 1, pp. 208–228, 2021.

[24] M.A. Nabian, H. Meidani. *A Deep Neural Network Surrogate for High-Dimensional Random Partial Differential Equations*, 2018, doi: 10.48550/arXiv.1806.02957.

[25] J. Naprstek, R. Kral, *Finite element method analysis of Fokker-Planck equation in stationary and evolutionary versions*, Adv. Eng. Softw. 72, 28–38, 2014.

[26] C. Nwankpa, W. Ijomah, A. Gachagan and S. Marshall, *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*, 2018, arXiv preprint arXiv:1811.03378.

[27] B. Oksendal, *Stochastic Differential Equations: An Introduction with Applications*, 6th ed., Springer, Universitext, 2003.

[28] M. Raissi, *Forward-Backward Stochastic Neural Networks: Deep Learning of High-Dimensional Partial Differential Equations*, arXiv preprint arXiv:1804.07010, 2018.

[29] 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*, Vol. 378, pp. 686–707, 2019.

[30] A. Saidi, *Numerical Methods for Stochastic Differential Equations*. Master's 2 Thesis, Department of Mathematics, University Dr. Tahar Moulay, Saida.

[31] B. Sepehrian, M. K. Radpoor, *Numerical solution of non-linear Fokker- Planck equation using finite differences method and the cubic spline functions.* Appl. Math. Comput. 262, 187–190, 2015.

[32] J. Sirignano and K. Spiliopoulos, DGM: A Deep Learning Algorithm for Solving Partial Differential Equations, *Journal of Computational Physics*, Vol. 375, pp. 1339–1364, 2018.

[33] Y. Xu, H. Zhang, Y. Li, et al. *Solving Fokker-Planck equation using deep learning.* Chaos, 30 (1): 013133, 2020. https://doi.org/10.1063/1.5132840