Stochastic differential equation and machine learning: Bridging Finance and Physics

Aritrajit Roy
University of Calcutta, Kolkata, India.
Soumen shaw
Department of Mathematics
Dinabandhu Andrews College, University of Calcutta, Kolkata.

February 21, 2025

Abstract

The advancement of the concept of deep learning and machine learning (ML) has led to numerous innovations in modeling complex stochastic systems. One such innovation is Neural Stochastic Differential Equations (Neural SDEs), which is a combination of classical Stochastic Differential Equations (SDEs) with neural networks to model uncertainty in dynamic systems. Neural SDEs have emerged as a powerful tool in various fields, including finance, where stochastic processes are fundamental to modeling asset prices and risk. This article aims to compare Neural SDEs with Neural Ordinary Differential Equations (Neural ODEs), focusing on their distinct features and applications, particularly in financial modeling.

1 Introduction

The neural SDEs are intertwined with the broader evolution of differential equations in the machine learning landscape. The formal exploration of ODEs in ML began with the landmark work of Chen et al. [1], who introduced neural ODEs as a continuous analog of residual neural networks. This pioneer work is allowing neural networks to learn the trajectories of dynamical systems by modeling them as ODEs rather than discrete sequences. Consequently, neural ODEs significantly reduced memory usage and improved computational efficiency in several tasks namely, image classification and time series forecasting. However, many real-world systems exhibit intrinsic randomness. ODEs are unable to capture these phenomena. These limitations enhanced the interest in extending the neural ODE framework to SDEs, which are more advanced in-terms of modeling such stochastic behaviors. Neural SDEs were first formally explored by Tzen and Raginsky (2019) [2] and laid the foundation for merging deep learning with stochastic processes, introducing a way to model systems where noise and randomness play a crucial role. Another pivotal moment came with the work of Li et al. (2020) [3], who further refined Neural SDEs, focusing on more efficient methods of solving these equations and improving their applicability in various domains, especially those involving uncertainty. Their monogram was instrumental in advancing the training of neural SDEs by addressing challenges in back-propagation through stochastic processes. Stochastic processes are foundational in financial modeling, particularly for price derivative products, assessing risk, and forecasting volatility. The evolution of mathematical finance has relied heavily on models like the Black-Scholes equation (1973), which uses SDEs to model the dynamics of asset prices under uncertainty. Building on this foundation, Heston model (1993) introduced stochastic volatility, further embedding SDEs in the financial landscape.

The development of Neural SDEs marks a significant milestone in financial modeling. Traditional SDE-based models in finance often rely on predefined parametric forms and assumptions about market dynamics. Neural SDEs offer a data-driven approach, enabling financial models to adapt the complexity as well as nonlinear behaviors of the market, which makes it more flexible. By integrating neural networks into the SDE framework, Neural SDEs can learn market dynamics directly from data, providing a powerful tool for modeling asset prices, volatility, and risk in ways that go beyond conventional methods.

A notable application of Neural SDEs in finance is in the pricing and hedging of derivatives, where the randomness inherent in financial markets can be more accurately modeled by using stochastic processes. Moreover, Neural SDEs have shown promise in portfolio optimization and risk management by capturing the nonlinear dependencies between assets and market factors. In this regard, Neural SDEs provide a novel approach to addressing key challenges in financial modeling, such as modeling tail risks and extreme market events.

Recently, Kidger et al. (2020) [4] have explored the application of Neural SDEs, particularly in financial sectors. They extend the applicability of Neural SDEs to irregular time-series data, which is common in high-frequency trading and financial market. This work highlights how Neural SDEs can be used to model complex temporal patterns in finance, providing more accurate predictions and risk assessments in dynamic and volatile markets.

As the field of ML continues to evolve, the integration of Neural SDEs into financial modeling presents exciting opportunities for further innovation. These models not only improve the understanding of market dynamics but also open up new avenues for developing more robust financial models that can better navigate the uncertainties inherent in financial systems.

Neural SDEs represent a cutting-edge intersection of machine learning, stochastic processes, and financial modeling. By combining the flexibility of neural networks with the robustness of stochastic calculus, it offers a versatile tool for modeling complex systems under uncertainty, with significant implications for finance. As research in this area continues to evolve, the applications of Neural SDEs in finance are poised to expand, providing new insights and capabilities for risk management, asset pricing, and market prediction.

2 Mathematical Preliminaries:

2.1 Stochastic Differential Equations

Stochastic differential equations (SDEs) can be seen as an extension of ordinary differential equations (ODEs) by incorporating a stochastic component to model the systems influenced by random fluctuations. In essence, an SDE is an ODE with added noise, often represented by Gaussian white noise. The general form of an SDE is:

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW_t, \tag{1}$$

where X_t represents the state variable, $\mu(X_t, t)$ denotes the drift term (deterministic part, similar as in ODE) and $\sigma(X_t, t)$ is the diffusion term, which modulates the intensity of the noise. The term dW_t represents a differential of a Wiener process (Brownian motion), which is the mathematical representation of Gaussian white noise.

2.2 Gaussian White Noise

An ordinary differential equation (ODE) can be expressed as:

$$\frac{dX_t}{dt} = \mu(X_t, t), \tag{2}$$

here the term X_t is entirely deterministic. However, in many real-world systems, uncertainty or randomness plays a critical role. The expected slope of X_t differs from the realized or experimental slope. This difference is due to Gaussian perturbation or Gaussian White Noise(GWN). Thus necessitating the addition of a stochastic term, to introduce randomness, we add a noise term in the form of Gaussian white noise. The result is an SDE of the form:

$$dX_t = \mu(X_t, t)dt + \sigma dW_t, \tag{3}$$

where W_t is a Wiener process and *sigma* represents the intensity of the noise. The Wiener process, W_t, satisfies:

$$\begin{aligned} &(i)\ W_0 = 0, \\ &(ii)\ W_t - W_s \sim N(0, t-s),\ \ for\ t \geq s \\ &(iii)\ W_t\ has\ independent\ normally\ distributed\ increments. \end{aligned} \tag{4}$$

This noise term introduces randomness into the evolution of X_t , reflecting the behavior of many physical and financial systems. In particular dX_t can represent the return of an asset for the trading period $(t, t + dt)$.

SDEs model systems where randomness or uncertainty is an inherent component of their evolution. By adding Gaussian white noise to the deterministic dynamics of an ODE, SDEs provide a powerful framework for describing real-world systems that evolve under uncertainty.

2.3 Stochastic Processes

A stochastic process is a collection of random variables $\{X_t : t \in T\}$ defined on a probability space $(\Omega, \mathcal{F}, \mathcal{P})$, where t typically represents time. The set T can be discrete or continuous, and each X_t represents the state of the process at time t. The process evolves over time, influenced by randomness or uncertainty. Mathematically, a stochastic process can be expressed as:

$$X_t : \Omega \to \mathcal{R} \tag{5}$$

where Ω is the sample space, and $\mathcal{R}$ is the state space. The process $\{X_t, t \geq 0\}$ describes how X_t evolves over time.

3 Applications

Stochastic process has been successfully applied in financial sector to capture its uncertainty.

3.1 Stock Price Modeling in Finance

One of the most important applications of stochastic processes in finance is modeling the dynamics of stock prices. A common model used is the Geometric Brownian Motion (GBM), which assumes that the stock price follows a stochastic process with both a deterministic trend and random fluctuations.

Let S_t represent the stock price at time t. The Geometric Brownian Motion is given by the following stochastic differential equation (SDE):

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{6}$$

where μ is the drift rate, representing the expected return of the stock. σ is the volatility of the stock, representing the intensity of the random fluctuations.dW_t represents the differential of a Wiener process (Brownian motion), which introduces randomness into the system. This SDE states that the change in the stock price consists of two parts:
1. A drift term, $\mu S_t dt$, which captures the deterministic part of the stock price evolution.
2. A diffusion term, $\sigma S_t dW_t$, which introduces randomness into the evolution of the stock price.
The solution to this SDE: (6) can be written as:

$$S_t = S_0 \; exp\left(\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t\right), \tag{7}$$

where S_0 is the initial stock price at $t = 0$.

3.2 Option Pricing with the Black-Scholes Model

Another famous application of the GBM model is the Black-Scholes model for option pricing. The Black-Scholes model assumes that the stock price follows a GBM as described in Equation (6). The price of a European call option, which gives the right to buy the stock at a strike price K at a future time t , can be derived using this model. The Black-Scholes formula is given by:

$$C(S_0, t) = S_0 \Phi(d_1) - K e^{-rt} \Phi(d_2), \tag{8}$$

where $d_1 = \frac{ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right) t}{\sigma\sqrt{t}}$, $d_2 = d_1 - \sigma\sqrt{t}$ and $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution, r is the risk-free interest rate, and t is the time to maturity of the option.

Stochastic processes provide a powerful framework for modeling randomness in systems that evolve over time. In finance, they are used extensively to model asset prices and risk. The Geometric Brownian Motion is one of the most widely used stochastic processes for modeling stock prices, and it forms the foundation of the Black-Scholes option pricing model. By incorporating both deterministic trends and random fluctuations, stochastic processes like GBM capture the inherent uncertainty in financial markets.

3.3 Geometric Brownian Motion Simulations

The following graph represents simulated paths of a Geometric Brownian Motion (GBM) process, generated in Python (see fig. 1).

It illustrates multiple trajectories of a GBM, showing the stochastic behavior typical in financial asset modeling. Despite its simplicity, the Geometric Brownian Motion model remains a cornerstone in quantitative finance due to its analytical tractability and intuitive appeal. It forms the basis of the famous Black-Scholes-Merton model, used to price options and other financial derivatives. The key parameters in GBM—the drift rate (representing expected return) and volatility (capturing the magnitude of price fluctuations)—are crucial for simulating realistic stock price paths. However, the model has limitations, as it assumes constant volatility and independence of returns over time, which fail to capture phenomena like volatility clustering, fat tails, and sudden market shocks observed in real-world markets. Modern advancements in financial modeling, such as stochastic volatility models and jump-diffusion processes, build on the GBM framework to address these shortcomings, offering more nuanced representations of market dynamics. Nevertheless, GBM's foundational role in financial theory ensures its continued use as a benchmark for both academic research and practical applications. A SDE can be solved symbolically using the rules of calculus but in real world use cases such as analyzing the dynamics of a stock price we often need to calculate the actual price in decimal values. Thus we need to use time-efficient Numerical algorithms (e.g Automated Differentiation) to evaluate the price as accurately as possible.

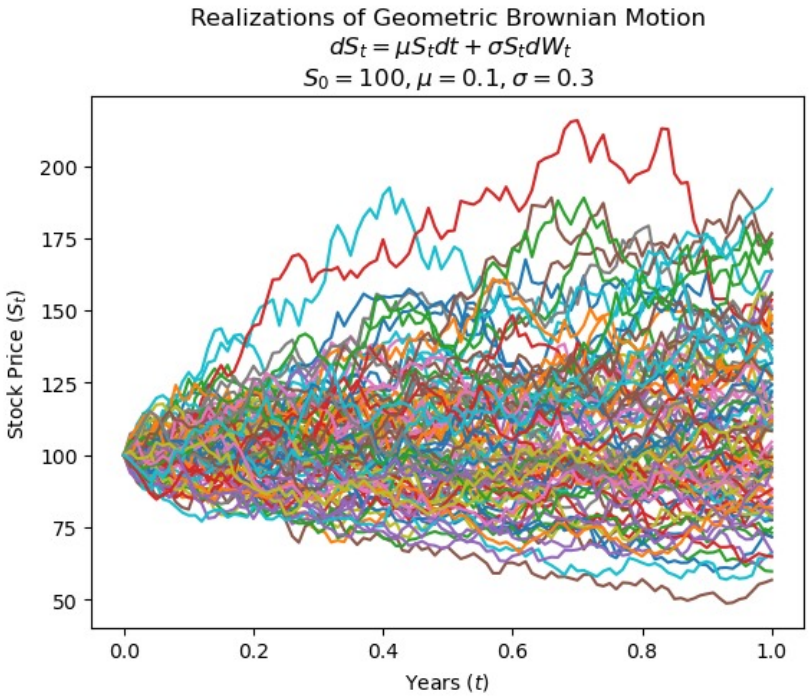

Figure 1: Geometric Brownian motion

4 Automatic Differentiation

Automatic Differentiation (AD) has a rich history, rooted in the need for accurate and efficient computation of derivatives in scientific and engineering applications. It evolved alongside advances in mathematics, numerical analysis, and computer science.

(i) Early Concepts in Derivatives (17th–19th Century): Isaac Newton and Gottfried Leibniz independently formalized calculus in the 17th century, laying the foundation for understanding derivatives. Numerical methods for approximating derivatives, like finite differences, emerged in the 19th century to address practical problems in physics and engineering.

(ii) Emergence of Computational Differentiation (1940s–1960s): World-War II and the rise of computers accelerated interest in numerical methods for solving differential equations and optimization problems. Researchers began exploring ways to compute derivatives more efficiently than finite differences, which were prone to numerical in- stability and inaccuracy.

(iii) Early days of AD uses 1960s: Wengert [5] described how to systematically record intermediate computations in a program to propagate derivatives using the chain rule. This method, known as 'Wengert tapes', introduced the idea of forward mode AD. AD remained a niche technique during this period, primarily used in optimization and control problems.

(iv) Formalization and Advancements (1970s–1980s): Researchers recognized that AD could be separated into two modes.
Forward Mode: Computes derivatives alongside function evaluation, efficient for functions with few inputs and many outputs.
Reverse Mode: Computes gradients efficiently for functions with many inputs and few outputs, pivotal for machine learning and adjoint-based methods.
In 1981, Griewank [6] published foundational work on reverse mode AD, showing its efficiency for problems like gradient-based optimization. Early software implementations of AD began to appear, but adoption was limited due to hardware and software constraints.

(v) Expansion into Scientific Computing (1990s): Increased computational power and growing interest in optimization spurred AD's adoption in fields like meteorology, computational physics, and engineering. ADIFOR (Automatic Differentiation in Fortran) and ADOL-C: Early software libraries for AD in programming languages like Fortran and C. Techniques for handling complex data structures, loops, and conditional branches in AD emerged, making it more robust and flexible.

(vi) AD and Machine Learning (2000s–2010s): Machine learning, particularly the rise of neural networks, drove significant advancements in AD. Reverse mode AD (often called back-propagation) became the backbone of gradient-based optimization in training neural networks. Libraries like Theano (2007) and TensorFlow (2015) integrated AD to compute gradients automatically, streamlining model development. PyTorch (2016) popularized dynamic computation graphs, further advancing AD's accessibility and flexibility.

(vii)Modern Era (2020s): AD is now ubiquitous in scientific computing, deep learning, and optimization. Frameworks like **JAX**, **DiffEqFlux**, and **Zygote.jl** leverage modern hardware to accelerate AD on GPUs and TPUs. Advances in mixed-mode AD (combining forward and reverse modes) and higher-order derivatives are being developed for applications in physics-informed neural networks, differential programming, and financial modeling. Research focuses on improving the efficiency, scalability, and numerical stability of AD for increasingly complex applications.

4.1 Impact of Automatic Differentiation

Automatic Differentiation (AD) stands apart from symbolic differentiation and numerical differentiation by overcoming the limitations inherent in these classical approaches. Symbolic differentiation, while exact in its mathematical results, struggles to handle complex, large-scale programs because it requires converting entire programs into mathematical expressions. This often results in inefficient, bloated computations (sometimes referred to as "expression swell") and poor performance. Numerical differentiation, on the other hand, approximates derivatives using methods like finite differences, but these approximations suffer from round-off errors due to floating-point arithmetic and truncation errors in discretization. These issues compound when higher- order derivatives are calculated, leading to increased inaccuracy and instability.

Furthermore, both symbolic and numerical methods are inefficient for computing gradients of functions with many inputs, such as in gradient-based optimization problems that arise in machine learn-

ing, scientific simulations, and control systems. Automatic differentiation addresses these challenges by using the chain rule systematically to propagate derivatives through computational graphs of the program. AD computes exact derivatives up to machine precision, avoids expression swell, and efficiently handles partial derivatives of functions with numerous inputs and outputs. These advantages make AD the method of choice for modern applications like training neural net- works, sensitivity analysis, and solving inverse problems.

AD has become indispensable across disciplines, enabling breakthroughs in fields such as: Machine Learning: Training neural networks via back-propagation. In physics, Solving partial differential equations in simulations. In finance, Computing risk sensitivities (e.g., 'Greeks') for derivative pricing. In robotics, Optimizing control strategies and learning dynamical models.

Fundamental to automatic differentiation is the decomposition of differentials provided by the chain rule of partial derivatives of composite functions. For the simple composition:

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$
$$where\ w_0 = x,\ w_1 = h(w_0),\ w_2 = g(w_1),\ w_3 = f(w_2) = y.$$

The Chain Rule gives:

$$\frac{\partial y}{\partial y} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial f(w_2)}{\partial w_2}\frac{\partial g(w_1)}{\partial w_1}\frac{\partial h(w_0)}{\partial x}$$

4.2 Types of Automatic Differentiation

Typically, automatic differentiation is implemented using two distinct modes.

• Forward accumulation (also known as bottom-up, forward mode, or tangent mode):

In forward accumulation, the chain rule is applied from the innermost function outward. Specifically:

$$\frac{\partial y}{\partial x} = \frac{\partial w_1}{\partial x}.\frac{\partial w_2}{\partial w_1}.\frac{\partial y}{\partial w_2}$$

This involves first computing $\frac{\partial w_1}{\partial x}$, then $\frac{\partial w_2}{\partial w_1}$, and finally $\frac{\partial y}{\partial w_2}$.

• Reverse accumulation(also known as top-down, reverse mode, or adjoint mode):

In contrast, reverse accumulation applies the chain rule in the opposite direction, starting from the outermost function. Specifically:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2}.\frac{\partial w_2}{\partial w_1}.\frac{\partial w_1}{\partial x}$$

Here, one computes $\frac{\partial y}{\partial w_2}$ first, followed by $\frac{\partial w_2}{\partial w_1}$, and finally $\frac{\partial w_1}{\partial x}$.

The value of the partial derivative, termed as the seed, is propagated either forward or backward. Initially, this value is considered as follows:

$$\frac{\partial x}{\partial x} = 1 \quad \text{or} \quad \frac{\partial y}{\partial y} = 1.$$

In forward accumulation, the function is evaluated, and the derivative is computed with respect to one independent variable in a single pass. For each independent variable $x_1,\ x_2,\ ...\ x_n$, a distinct pass is necessary. Consequently, during the pass for a specific variable (e.g., x_i), its derivative is to be set as follows:

$$\frac{\partial x_i}{\partial x_j} = \delta_{ij} \tag{9}$$

On the other hand, reverse accumulation requires the evaluation of partial functions to compute the partial derivatives. This approach first evaluates the function and then computes the derivatives with respect to all independent variables in an additional pass.

Forward mode is generally efficient for functions with fewer inputs and many outputs, while reverse mode is excels for functions with many inputs and fewer outputs, making it ideal for gradient-based optimization in machine learning. Back-propagation of errors in multilayer perception is a special case of reverse mode.

5 Neural Stochastic Differential Equation

Ordinary differential equation (ODE) are often used to model deterministic dynamic systems. The Adjoint sensitivity method has the ability to compute gradients of ODE solutions with constant memory complexity. In recent years, this is combined with reverse-mode automatic differentiation. This allowed ODEs with millions of parameters to be fit to data. Thus, enabling more flexible density estimation and time series models. The Adjoint Sensitivity methods was used by Chen et al. [1] in their work on Neural ODEs to compute it's gradient with respect to parameters. the generalization of ODEs equipped with instantaneous noise gives rise to SEDs which acts as a model for phenomena governed by many small and undetected interactions, such as motion of molecules in a liquid or prices in a market [3].

5.1 Adjoint Sensitivity

Adjoint sensitivity is a computational technique primarily used to calculate gradients of a scalar objective function with respect to larger number of parameters efficiently, making it crucial in the fields like optimal control, machine learning, and data assimilation.

In order to illustrate the method, let us consider a dynamical system be governed by an ODE:

$$\frac{dx}{dt} = f(x, u, t), \quad x(0) = x_0, \tag{10}$$

where:

$x \in \mathcal{R}^n$ is the state variable,

$u \in \mathcal{R}^p$ represents control inputs or parameters,

$f : \mathcal{R}^n \times \mathcal{R}^p \times \mathcal{R} \to \mathcal{R}^n$ is the nonlinear system function.

The Goal is to minimize a scalar objective $\mathcal{Z}$ (cost function) of the form:

$$\mathcal{Z}(u) = \Phi(x(T)) + \int_0^T L(x, u, t)dt, \tag{11}$$

in which:

$\Phi(x(T))$ is the terminal cost,

$L(x, u, t)$ is the running cost,

T describes the time horizon.

5.1.1 The Adjoint System

The Adjoint sensitivity method involves defining an augmented Lagrangian $\mathcal{L}$ to incorporate the system dynamics:

$$\mathcal{L}(x, u, \lambda) = \Phi(x(T)) + \int_0^T \left[L(x, u, t) + \lambda^T \left(f(x, u, t) - \frac{dx}{dt} \right) \right] dt, \tag{12}$$

where $\lambda(t) \in \mathcal{R}^n$ are the adjoint variables (Lagrangian multipliers).

By imposing the dynamics $\frac{dy}{dx} = f(x, u, t)$, the sensitivity of $\mathcal{Z}$ to u can be expressed without explicitly differentiating the entire trajectory.

5.1.2 Adjoint Gradient Computation

Theorem 5.1: Let $x(t)$ be the state trajectory obtained by solving the forward ODE, and $\lambda(t)$ be the solution of the adjoint ODE:

$$\frac{d\lambda}{dt} = -\frac{\partial f^T}{\partial x}\lambda - \frac{\partial L}{\partial x}, \quad \lambda(T) = \frac{\partial \Phi}{\partial x(T)}. \tag{13}$$

Then, the gradient of the objective function $\mathcal{Z}$ with respect to the control parameters u is given by

$$\frac{\partial \mathcal{Z}}{\partial u} = \int_0^T \left(\frac{\partial L}{\partial u} + \lambda^T \frac{\partial f}{\partial u} \right) dt. \tag{14}$$

Proof:
Differentiating $\mathcal{Z}(u)$ with respect to u we obtain,

$$\frac{\partial \mathcal{Z}}{\partial u} = \frac{\partial \Phi}{\partial x(T)} \frac{\partial x(T)}{\partial u} + \int_0^T \left(\frac{\partial L}{\partial u} + \frac{\partial L}{\partial x} \frac{\partial x}{\partial u} \right) dt. \tag{15}$$

Applying the chain rule on $x(T)$ yields,

$$\frac{\partial x(T)}{\partial u} = \int_0^T \frac{\partial f}{\partial u} dt + \int_0^T \frac{\partial f}{\partial x} \frac{\partial x}{\partial u} dt. \tag{16}$$

Now, define the adjoint variable as follows:

$$\frac{d\lambda}{dt} + \frac{\partial f^T}{\partial x} \lambda + \frac{\partial L}{\partial x} = 0, \quad \lambda(T) = \frac{\partial \Phi}{\partial x(T)}. \tag{17}$$

Substituting $\lambda(t)$ in the expression of $\frac{\partial \mathcal{Z}}{\partial u}$ we obtain

$$\frac{\partial \mathcal{Z}}{\partial u} = \int_0^T \left(\frac{\partial L}{\partial u} + \lambda^T \frac{\partial f}{\partial u} \right) dt.$$

Thus, the adjoint system efficiently computes gradients with respect to u without explicit trajectory derivatives.

Corollary 1: Time independent systems

In case of a time invariant system, where $f(x, u)$ and $L(x, u)$ do not explicitly depend on t, the adjoint ODE simplifies to:

$$\frac{d\lambda}{dt} = -\frac{\partial f^T}{\partial x} \lambda - \frac{\partial L}{\partial x}, \quad \lambda(T) = \frac{\partial \Phi}{\partial x(T)}. \tag{18}$$

5.2 Adjoint Sensitivity in Neural ODEs

In a Neural ODE, the hidden states of neural network evolve according to an ODE parameterized by the network weights θ:

$$\frac{dx(t)}{dt} = f_\theta(x(t), t), \quad x(0) = x_0, \tag{19}$$

where:
$x(t) \in \mathcal{R}^n$: state at time t,
$f_\theta(x(t), t) : \mathcal{R}^n \times \mathcal{R} \to \mathcal{R}^n$: a neural network with parameter θ,
x_0: initial input (e.g., the data point in a classification problem).
The terminal state $x(T)$ is used to compute a loss function $\mathcal{L}(\theta)$, typically defined as:

$$\mathcal{L}(\theta) = \Phi(x(T)), \tag{20}$$

where $\Phi(x(T))$ measures the error (e.g., cross-entropy loss for classification).

Theorem 5.2: Adjoint system for neural ODEs

The gradient of the loss function $\mathcal{L}(\theta) = \Phi(x(T))$ with respect to parameter θ can be computed by solving the adjoint system:

$$\frac{d\lambda}{dt} = -\lambda^T \frac{\partial f_\theta}{\partial x}, \quad \lambda(T) = \frac{\partial \Phi}{\partial x(T)}. \tag{21}$$

The gradient is then given by

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \lambda^T \frac{\partial f_\theta}{\partial \theta} dt. \tag{22}$$

Proof:
Differentiating the loss function $\mathcal{L}(\theta)$ with respect to θ:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \Phi}{\partial x(T)} \frac{\partial x(T)}{\partial \theta}. \tag{23}$$

The state $x(T)$ depends on θ implicitly through the ODE. By the chain rule, the derivative of $x(T)$ with respect to θ is:

$$\frac{\partial x(T)}{\partial \theta} = \int_0^T \left(\frac{\partial f_\theta}{\partial \theta} + \frac{\partial f_\theta}{\partial x} \frac{\partial x}{\partial \theta} \right) dt. \tag{24}$$

Introducing the adjoint variable $\lambda(t)$, which satisfies:

$$\frac{d\lambda}{dt} + \lambda^T \frac{\partial f_\theta}{\partial x} = 0, \quad \lambda(T) = \frac{\partial \Phi}{\partial x(T)}. \tag{25}$$

Now, multiplying the ODE by $\lambda(t)$ and integrating we obtain:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \lambda^T \frac{\partial f_\theta}{\partial \theta} dt.$$

5.2.1 Neural ODE with scalar dynamics

As an illustrative example, here we consider following scalar neural ODE:

$$\frac{dx(t)}{dt} = -\theta x, \quad x(0) = x_0, \tag{26}$$

Here, $f_\theta(x) = -\theta x$, and the solution is given by

$$x(t) = x_0 e^{-\theta t} \tag{27}$$

(i) Loss function:

$$\mathcal{L}(\theta) = \frac{1}{2} \left(x(t) - x_{target} \right)^2. \tag{28}$$

(ii) Adjoint ODE:

$$\frac{d\lambda}{dt} = -\lambda \frac{\partial(-\theta x)}{\partial x} = \lambda\theta, \quad \lambda(T) = \frac{\partial \mathcal{L}}{\partial x(T)} = x(T) - x_{target}. \tag{29}$$

Solving backward:

$$\lambda(t) = \lambda(T) e^{\theta(T-t)}. \tag{30}$$

(iii) Gradient computation:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \lambda(t) \frac{\partial f_\theta}{\partial \theta} \, dt = \int_0^T \lambda(t)(-x(t)) \, dt. \tag{31}$$

Substituting $x(t)$ and $\lambda(t)$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \int_0^T \left[\lambda(T) e^{\theta(T-t)} \right] \left[-x_0 e^{-\theta t} \right] dt. \\ &= -\lambda(T) x_0 \int_0^T e^{\theta T} \, dt = -x(T)\lambda(T)T. \end{aligned} \tag{32}$$

5.3 Neural Stochastic Differential Equations (Neural SDEs)

A Neural Stochastic Differential Equation (Neural SDE) is a continuous-time stochastic process where both the drift and diffusion terms are parameterized by neural networks. The general form of a Neural SDE is:

$$dX_t = f_\theta(X_t, t)dt + g_\theta(X_t, t)dW_t, \tag{33}$$

where:

- $X_t \in \mathcal{R}^d$ is the state variable,
- W_t is a Wiener process (Brownian motion),
- $f_\theta(X_t, t)$ (drift function) and $g_\theta(X_t, t)$ (diffusion function) are neural networks with trainable parameters θ.

By training these neural networks from data, the Neural SDE learns an underlying stochastic process that best describes the observed samples.

5.4 Neural SDEs as Infinite-Dimensional Generative Adversarial Networks

A Generative Adversarial Network (GAN) consists of a generator G_θ mapping a low-dimensional latent space (e.g., Gaussian noise) to a complex data distribution. Given a latent variable $Z \sim p_Z$, a traditional GAN models:

$$X = G_\theta(Z), \tag{34}$$

where G_θ is a neural network. However, this framework assumes a finite-dimensional mapping, means produces only the static samples. One can model distributions over entire stochastic processes (continuous-time data), by adopting an infinite-dimensional generalization.

5.4.1 Mathematical Formulation: Infinite-Dimensional Generalization

Instead of generating individual samples X, a Neural SDE generates entire trajectories:

$$X_t = X_0 + \int_0^t f_\theta(X_s, s)ds + \int_0^t g_\theta(X_s, s)dW_s. \tag{35}$$

This defines a probability measure over function spaces, denoting an **infinite-dimensional functional space**. The key insight is that **Neural SDEs generalize GANs from finite-dimensional vector distributions to probability measures over path spaces**.

5.4.2 Generator in Infinite-Dimensional GANs

Instead of a neural network $G_\theta : Z \mapsto X$, we now have a **Neural SDE as a stochastic process** evolving over time. The generator G_θ becomes the **SDE flow**, mapping a noise process W_t to a sample path X_t:

$$G_\theta : W_t \mapsto X_t. \tag{36}$$

5.4.3 Discriminator in Infinite-Dimensional GANs

In an infinite-dimensional GAN, the discriminator is a **functional critic** that operates on stochastic paths X_t. Common choices for infinite-dimensional critics include:

- **Wasserstein distance** over probability measures of stochastic paths,
- **Score matching**, which compares probability flow of the generated paths to real paths.

5.4.4 Training the Neural SDE GAN

The generator learns an **optimal drift-diffusion structure** such that the generated process matches the real process. The discriminator learns to identify inconsistencies in the generated stochastic paths. Training methods include:

- **Maximum Mean Discrepancy (MMD) loss**, comparing path distributions,
- **Wasserstein loss**, computing the transport cost between real and generated paths.

This **Neural SDE GAN framework** is useful for generative modeling of **financial time series, physics-based simulations, and stochastic dynamical systems**.

6 Backward Stratonovich Integrals and Itô Comparison

The integration framework for SDEs depends on the choice of calculus:

- **Itô calculus**: Non-anticipative (used in finance, stochastic control).
- **Stratonovich calculus**: Symmetric (used in physics, neural networks).

6.1 Itô Integral

The Itô integral is defined as:

$$\int_0^T H_t dW_t = \lim_{n\to\infty} \sum_{i=0}^{n-1} H_{t_i}(W_{t_{i+1}} - W_{t_i}). \tag{37}$$

Due to the **quadratic variation** of W_t, Itô's Lemma introduces an extra term:

$$dF(X_t) = F'(X_t)dX_t + \frac{1}{2}F''(X_t)g^2(X_t, t)dt. \tag{38}$$

6.2 Stratonovich Integral

The Stratonovich integral uses midpoint sampling:

$$\int_0^T H_t \circ dW_t = \lim_{n\to\infty} \sum_{i=0}^{n-1} \frac{H_{t_i} + H_{t_{i+1}}}{2}(W_{t_{i+1}} - W_{t_i}). \tag{39}$$

This makes Stratonovich calculus behave more like standard calculus, avoiding extra correction terms.

6.3 Backward Stratonovich Integral

Backward integration is used when solving backward SDEs (BSDEs). A **backward Stratonovich integral** satisfies:

$$\int_T^0 H_t \circ dW_t = \lim_{n\to\infty} \sum_{i=0}^{n-1} \frac{H_{t_i} + H_{t_{i+1}}}{2}(W_{t_{i+1}} - W_{t_i}). \tag{40}$$

This is crucial in **stochastic adjoint sensitivity analysis**, where gradients are computed in reverse time.

6.4 Computational Aspects of Neural SDEs

- **Adjoint Sensitivity**: Instead of storing all intermediate states, Neural SDEs use **stochastic adjoint methods**.
- **Efficient Training**: Solvers like `torchsde` and `Diffrax` leverage **reversible solvers** to compute gradients efficiently.
- **Memory Complexity**: The reversible trick avoids storing all time steps, making Neural SDEs practical for high-dimensional applications.
- **Parallelization**: Unlike ODEs, SDEs introduce noise-dependent gradients, requiring **stochastic backpropagation techniques**.

Adopting generalized Stratonovich method, solution of neural SDE can be visualized in the following manner (see fig.2):

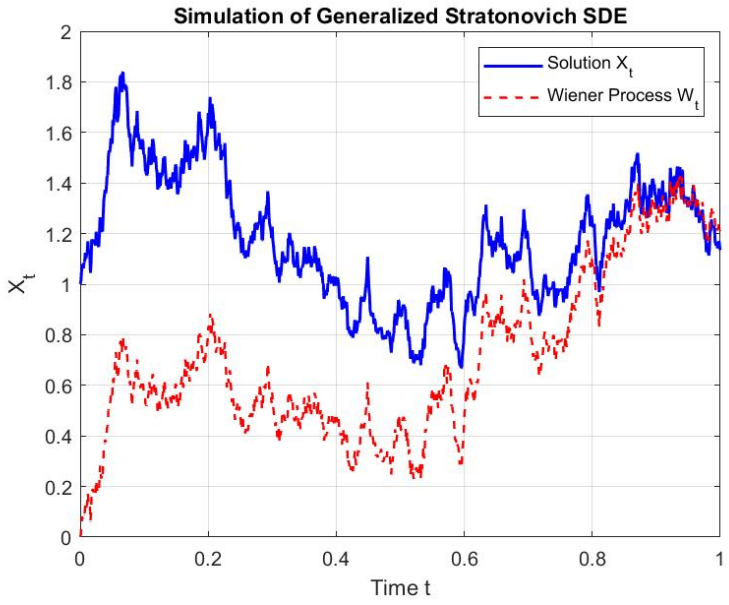

Figure 2: The solution X_t and the Winner Process W_t

7 Einstein and Bachelier's Formulations of Brownian Motion

Brownian motion is the random movement of particles suspended in a fluid, driven by collisions with molecules in thermal motion. Two landmark works independently laid the foundations for its mathematical treatment:

- **Louis Bachelier** introduced Brownian motion in the context of finance, modeling stock price movements as a stochastic process.
- **Albert Einstein** formulated Brownian motion as a physical phenomenon, using statistical mechanics to describe the random motion of microscopic particles in a fluid.

Both approaches ultimately converge on similar mathematical formulations, but they differ significantly in motivation, interpretation, and application.

7.1 Bachelier's Formulation: Brownian Motion in Finance

In 1900, Louis Bachelier's doctoral thesis [7], was the first work to model stock prices using a stochastic process. He proposed that asset prices evolve randomly, much like physical particles in Brownian motion.

7.1.1 Key Contributions

1. **Mathematical Model of Stock Prices**

$$S_t = S_0 + \sigma W_t \tag{41}$$

where:

- S_t is the stock price at time t,
- S_0 is the initial price,
- W_t is a *Wiener process* (Brownian motion),
- σ is a volatility parameter.

2. **Normal Distribution of Price Changes**

$$P(S_t \leq x) = \frac{1}{\sqrt{2\pi\sigma^2 t}} \int_{-\infty}^{x} e^{-\frac{(y-S_0)^2}{2\sigma^2 t}} dy \tag{42}$$

3. **Expectation and Martingale Property**

$$E[S_t] = S_0 \tag{43}$$

4. **Connection to the Heat Equation**

$$\frac{\partial p}{\partial t} = \frac{\sigma^2}{2} \frac{\partial^2 p}{\partial x^2} \tag{44}$$

7.1.2 Limitations

Bachelier's model allows negative prices, assumes constant volatility, and does not accurately describe real markets. Later models introduced *geometric Brownian motion* to resolve these issues.

7.2 Einstein's Formulation: Brownian Motion in Physics

In 1905, Einstein [8] provided a theoretical explanation for Brownian motion, bridging thermodynamics and statistical mechanics.

7.2.1 Key Contributions

1. **Stochastic Differential Equation for Particle Motion**

$$\langle x^2 \rangle = 2Dt \tag{45}$$

2. **Diffusion Equation**

$$\frac{\partial p}{\partial t} = D \frac{\partial^2 p}{\partial x^2} \tag{46}$$

3. **Generalization to Stochastic Processes**

$$dX_t = \sqrt{2D} dW_t \tag{47}$$

7.3 Comparison Between Bachelier and Einstein's models

8 Application of Neural SDEs to classify colors based on their diffusion patterns

8.1 The Dataset and Problem Setup Input Data:

Each sample is the diffusion pattern of a color (red, blue, green) over time on the blotting paper. The data could be captured as:

Feature	Bachelier (1900)	Einstein (1905)
Context	Finance (Stock Prices)	Physics (Molecular Motion)
Mathematical Model	Additive Brownian Motion	Diffusion Equation
Equation	$S_t = S_0 + \sigma W_t$	$\langle x^2 \rangle = 2Dt$
Assumptions	Gaussian-distributed price changes	Random collisions with molecules
Key Contribution	First stochastic model of financial markets	Experimental confirmation of atomic theory

Table 1: Comparison of Bachelier's and Einstein's Formulations

Images showing the spread pattern. Time-series data of pixel intensities or radial concentration gradients.
Features:
Spatial spread patterns (e.g., how the pigment diffuses radially or irregularly).
Temporal evolution (how the diffusion progresses over time).
Labels:
The three colors: Red, Blue, and Green.

8.1.1 Challenge:

Variability in patterns due to stochastic nature of diffusion, irregularities in paper, and different pigment properties.

8.1.2 Neural SDEs

Neural SDEs are particularly suited for this problem because: Stochasticity in the Data: Diffusion is inherently stochastic, so modelling it as an SDE captures this variability naturally.

8.1.3 Temporal Dynamics:

Neural SDEs handle time-series data effectively by learning a latent stochastic process that evolves over time. The Neural SDE model: Represents the evolution of data (e.g., pigment spread) as a stochastic process. Uses a drift term (deterministic component) and a diffusion term (stochastic component) to model dynamics. Adjusts the model parameters to maximize classification accuracy.

8.1.4 Stochastic Adjoint Algorithm

Stochastic Adjoint Algorithm Proposed by Ricky T.Q. Chen and David Duvenaud, this algorithm efficiently computes gradients for Neural SDEs, enabling training. Why it's suitable here: Handles stochasticity in training data (the variability in diffusion patterns). Reduces memory requirements compared to backpropagation through time. Scales well with large datasets, allowing efficient optimization for classifying patterns

8.1.5 Implementation Workflow

1. Data Collection: Capture and pre-process diffusion patterns from the blotting paper for each colour. Convert data into a format suitable for Neural SDEs (e.g., time-series pixel intensity or radial distribution).
2. Model Design: Define a Neural SDE with: Drift function to capture deterministic aspects of diffusion. Diffusion function to capture stochastic variability.
3. Training: Use the Stochastic Adjoint Algorithm to compute gradients for parameter updates. Train the model to minimize a loss function, such as cross-entropy, for color classification.
4. Evaluation: Test the trained Neural SDE on unseen samples to measure accuracy. Analyse misclassifications to refine the model.
5. Expected Challenges Noise in Data: Variability in blotting paper properties and experimental conditions might introduce unwanted noise. Class Overlap: If diffusion patterns for some colors are similar, the model might struggle to distinguish them. Model Complexity: Neural SDEs can be computationally intensive, especially for large datasets.

6. Extensions Explore if additional features (e.g., chemical properties of pigments) improve classification. Investigate transfer learning to generalize the model for pigments it hasn't seen before. Use this framework to study other diffusion-driven phenomena in physics, chemistry, or biology.

The movement of water molecules causing diffusion is inherently stochastic (due to Brownian motion), the overall path traced by the pigment on the blotting paper is not purely stochastic. Here's why:
1. Diffusion Process: The pigment particles spread due to diffusion, driven by random thermal motion of water molecules. This part is stochastic at a microscopic level. However, diffusion also follows a deterministic pattern described by Fick's laws of diffusion. For example:
Fick's First Law: The flux of particles is proportional to the concentration gradient.
Fick's Second Law: Over time, the concentration spreads predictably in a radial or specific pattern, depending on initial conditions.
2. Macroscopic Observation: On blotting paper, the pigment tends to spread symmetrically outward from the initial point of contact (if conditions like paper texture, water levels, and initial pigment placement are uniform). The shape is often a predictable circle or ellipse. Variations or irregularities in the blotting paper's structure (e.g., fibers, pores) or uneven water absorption could introduce slight randomness, but the overall spreading follows deterministic principles.

9 Unifying Einstein and Bachelier's models via Neural SDEs

The experiment attempts to unify these approaches by modeling **the diffusion of pigments** using **Neural SDEs**, which combine:

- The **stochastic nature of diffusion** (Einstein's perspective).
- The **mathematical structure of stochastic processes** used in finance (Bachelier's approach).

9.1 Why Neural SDEs?

1. **Bridging Finance and Physics:**
   - The pigment's diffusion **physically follows Einstein's laws**, but
   - The **stochastic adjoint method** resembles Bachelier's probabilistic framework.
2. **Learning Stochastic Dynamics:**
   - Neural SDEs learn both **deterministic trends (Fick's Laws of Diffusion)** and **random fluctuations (Brownian motion)**.
3. **Applications Beyond Color Diffusion:**
   - This framework could be applied to financial modeling, biological diffusion, and stochastic control systems.

10 Concluding remarks

Bachelier pioneered the use of Brownian motion in finance, treating it as a stochastic process for asset prices. Einstein provided a physical explanation, proving the existence of atoms via statistical mechanics. Neural SDE experiment unifies both views by analyzing diffusion patterns through a machine-learning-driven stochastic framework, applicable to both physics and finance. Including the flexibility of neural networks with the robustness of stochastic calculus, Neural SDE is a versatile tool for modeling complex systems under uncertainty, with significant implications for finance.

References

[1] T. Q. Chen, Y. Rubanova, J. Bettencourt, D. Duvenaud, Neural ordinary differential equations, CoRR abs/1806.07366 (2018). arXiv:1806.07366.
URL http://arxiv.org/abs/1806.07366

[2] B. Tzen, M. Raginsky, Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit, CoRR abs/1905.09883 (2019). arXiv:1905.09883.
URL http://arxiv.org/abs/1905.09883

[3] X. Li, T. L. Wong, R. T. Q. Chen, D. Duvenaud, Scalable gradients for stochastic differential equations, CoRR abs/2001.01328 (2020). arXiv:2001.01328.
URL http://arxiv.org/abs/2001.01328

[4] P. Kidger, J. Morrill, J. Foster, T. J. Lyons, Neural controlled differential equations for irregular time series, CoRR abs/2005.08926 (2020). arXiv:2005.08926.
URL https://arxiv.org/abs/2005.08926

[5] R. E. Wengert, A simple automatic derivative evaluation program, Commun. ACM 7 (8) (1964) 463–464. doi:10.1145/355586.364791.
URL https://doi.org/10.1145/355586.364791

[6] A. Griewank, Generalized descent for global optimization, J Optim Theory Appl 34 (1981) 11–39. doi:10.1007/BF00933356.
URL https://doi.org/10.1007/BF00933356

[7] L. Bachelier, Theory of speculation', doctoral thesis (1900), coot-ner (ed), 'random character of stock market prices', Ph.D. thesis, Massachusetts Institute of Technology (1964).

[8] A. B. Einstein, On the motion of small particles suspended in liquids at rest required by the molecular-kinetic theory of heat, Annanel der Physik 17 (1905) 549–560.
URL https://api.semanticscholar.org/CorpusID:13642744