BACHELOR'S THESIS

# Degree in Mathematics

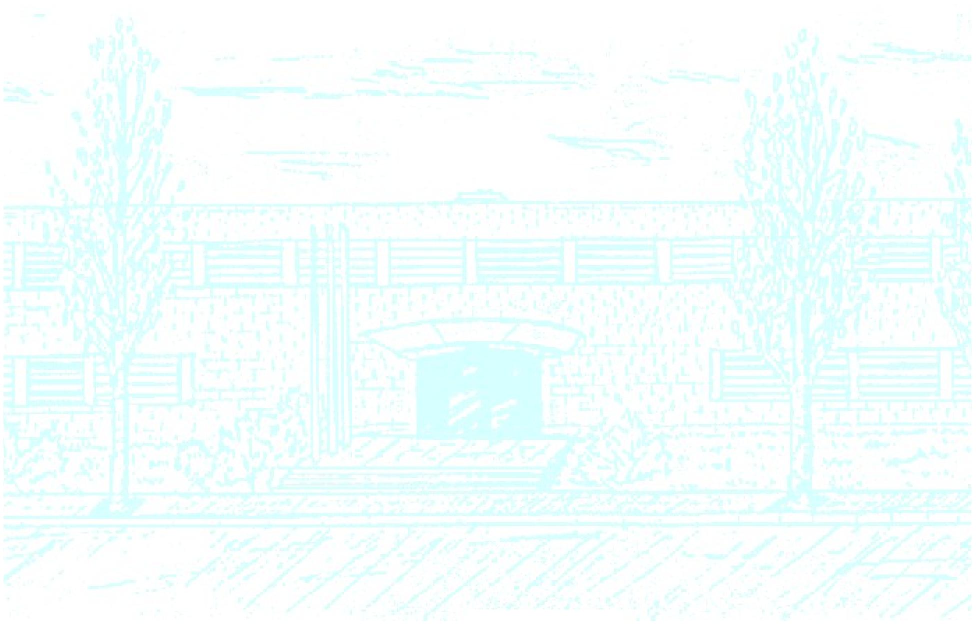
**Title:** A Deep Learning Method for Optimal Stopping Problems

**Author:** Andreu Boix Torres

**Advisor:** Argimiro Arratia Quesada

**Department:** Department of Computer Science & BGSMath & IMTech

**Academic year:** 2022-2023



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics
Bachelor's Degree Thesis

# A Deep Learning Method for Optimal Stopping Problems

**Andreu Boix Torres**

Supervised by Argimiro Arratia Quesada
Department of Computer Science & BGSMath & IMTech

May, 2023

I would like to say thank you to my family for their unconditional love and support. To my friends, for the constant words of encouragement. To Argimiro, for his inspiring vitality, interest and guidance. Also, to any who may have contributed in any way. Thank you all.

## Abstract

The main goal of this thesis is to complement the theoretical foundations and implement the deep learning model presented in the paper titled "Deep Optimal Stopping" by Becker et al, published in the Journal of Machine Learning Research in 2019. The deep learning model is specifically designed to address optimal stopping problems by learning the optimal stopping rule from Monte Carlo samples. Consequently, its versatility extends to various scenarios where the underlying randomness can be simulated effectively.

The thesis introduces the theory of Optimal Stopping problems, Deep Neural Networks and their relation. The thesis then provides a detailed analysis and implementation of the *Deep Optimal Stopping* algorithm in PyTorch using Python. The implementation is done on synthetic data to test the method with the literature. Besides, the implementation provided not only uses the Black-Scholes underlying asset model but is extended to models with jumps (Kou and Merton) and the Heston model. Then a use case with real-world financial data is shortly outlined.

Interesting applications pertaining to the valuation of financial derivatives, such as Bermudan call options, are explored, where optimal stopping plays a crucial role in the valuation process. The thesis analyzes the performance of the algorithm on these applications, comparing it to other commonly used methods for valuing financial derivatives.

In conclusion, this thesis contributes to the understanding and potential applications of the *Deep Optimal Stopping* algorithm in mathematical finance, particularly in the valuation of financial derivatives.

### Keywords

Optimal stopping problem, stochastic process, Markov process, financial asset modelling, stopping time, deep learning, neural networks, options, financial derivatives.

# Contents

# 1. Introduction

## Motivation of the thesis

The motivation for this thesis stems from the article "Deep Optimal Stopping" by Becker et al. [4] The theoretical foundations of the thesis are based on the works of Oksendal's *Stochastic Differential Equations* and G. Peskir and A. Shiryaev's *Optimal Stopping and Free-Boundary Problems* [10, 11]. These works provide a comprehensive understanding of optimal stopping problems and the underlying stochastic processes. Additionally, they support some of the theoretical details of the methodology provided by Becker et al. One of the main contributions of this thesis is the display and exploration of the detailed proofs of the different results stemming from the paper. Also, the explicit programming of the model is provided and is further generalized to other popular underlying asset models and stochastic processes, introducing jumps or non-constant volatility to capture peculiarities of determined market dynamics.

## Main goal

The main objective of this thesis is to present and expand on the theoretical details of the paper *Deep Optimal Stopping*, while simultaneously delving into the theory of optimal stopping. Another crucial aim is to implement the methodology outlined in the paper, and improve it while addressing interesting applications in mathematical finance, such as derivative valuation. Our work centers on Markov stochastic processes, which permit us to frame every optimal stopping problem within a Markovian framework, even if it necessitates integrating all pertinent past information into the state variable, which can increase the dimensionality of the problem. While it is possible to solve optimal stopping problems theoretically, numerical solutions can be challenging due to the curse of dimensionality, where higher dimensions lead to increasingly difficult and expensive computations. To address this, we employ the methodology presented by Becker et al, which is a deep learning approach for approximating the optimal stopping time, and demonstrate that it can achieve arbitrary accuracy thanks to the universal approximation theorem. Therefore, this thesis will contribute to a deeper understanding of optimal stopping theory and provide a practical method for solving optimal stopping problems in high-dimensional settings.

## Organization of the thesis

In section 2, we introduce the key definitions and concepts necessary to understand the theory of optimal stopping problems. The concepts are intended to be presented incrementally in the sense that we start with the definition of a stochastic process, concluding with the Optimal Stopping problem definition. In section 3 we address one of the main results found by Becker et al, by presenting and proving a theorem that enables us to *discretize* the optimal stopping time in terms of 0-1 stopping decisions. In section 4, we introduce the feed-forward neural networks, explain their importance and justification for use, and present the approximation of the optimal stopping time and the optimal value estimate by neural networks. It is important to note that optimal stopping times can be solved theoretically, and we present the results that grant this conclusion in Appendix A. In section 5, we tackle some applications of interest in mathematical finance and outline how to perform, to a certain extent, a market analysis for the asset or assets involved. In Appendix A we proof the theorem for the theoretical solution of the optimal stopping time problem. In Appendix B we dive into the efficient simulation details for the different underlying assets. In Appendix C we provide the complete code for the implementation of the Deep Optimal Stopping method.

# 2. Optimal Stopping theory

## 2.1 Stochastic Processes

**Definition 2.1** (Stochastic process)**.** A *stochastic process* is a collection of random variables $\{X_t : t \in T\}$ indexed by $t$, which is usually referred as the time parameter. They are defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where $\Omega$ is a sample space, $\mathcal{F}$ a $\sigma$-algebra on the sample space and $\mathbb{P}$ a probability measure on the $\sigma$-algebra $\mathcal{F}$.

**Definition 2.2.** Let $X_t$ be a stochastic process. For any $\omega \in \Omega$ fixed, as time $t$ passes, we define $\{X_t(\omega) : t \in T\}$ as a *sample path* or *trajectory* of $X_t$ corresponding to $\omega$.

**Definition 2.3** (Stochastic differential equation)**.** A *stochastic differential equation* (SDE) is a differential equation in which randomness appears in the coefficients, i.e. one or more terms are stochastic processes. Then, the solution is also a stochastic process.

We will consider SDEs of the form:

$$dX_t = b(t, X_t)dt + \sigma(t, X_t)dB_t$$

or, equivalently, in integral form:

$$X_t = X_0 + \int_0^t b(s, X_s)ds + \int_0^t \sigma(s, X_s)dB_s$$

where $B_t$ is a Brownian motion and $b$ is defined as the drift coefficient and $\sigma$ the diffusion coefficient[1]. $dB_t$ refers to the increments of the Brownian motion. We do not provide details on this class of integrals as it will not be necessary to compute them.

**Definition 2.4** (Gaussian process)**.** A *Gaussian process* is a stochastic process $\{X_t : t \in T\}$ where $(X_{t_1}, \dots, X_{t_k})$ follows a multivariate normal distribution for any choice of $t_1, \dots, t_k \in T$.

**Definition 2.5** (Brownian motion)**.** A Wiener process or Brownian motion $B_t$, $t \in \mathbb{R}$ is a continuous-time Gaussian process such that

- $B_0 = 0$
- the trajectories of $B_t$ are continuous functions
- it has independent and stationary increments $\{B_t - B_s : t < s\}$
- the increments have $\mathbb{E}[B_t - B_s] = 0$ and $\mathbb{V}\text{ar}[B_t - B_s] = |t - s|$, for $s, t \in \mathbb{R}$

An important consequence is that $B_t - B_s \sim N(0, |t - s|)$. An $n$-dimensional Brownian motion is $B_t = (B_t^{(1)}, \dots, B_t^{(n)})$ where $B_t^{(i)}$ $1 \leq i \leq n$ are independent 1-dimensional Brownian motions. On the other hand, the semi-rigorous definition and simulation of multidimensional Brownian motions can be found in *Appendix B: Simulating models*.

[1]A solution of these equations is called a *diffusion process* or *diffusion*.

## 2.2 Filtrations and martingales

**Definition 2.6** (Filtration)**.** A *filtration* on $(\Omega, \mathcal{F}, \mathbb{P})$ is an increasing family $(\mathcal{M}_t)_{t\geq 0}$ of sub-$\sigma$-algebras of $\mathcal{F}$ such that

$$\mathcal{M}_s \subseteq \mathcal{M}_t \subseteq \mathcal{F} \qquad \text{for } 0 \leq s \leq t$$

Intuitively, $\mathcal{M}_t$ is the historical information that we know up to time $t$.

**Definition 2.7.** The space $(\Omega, \mathcal{F}, (\mathcal{M}_t)_{t\geq 0}, \mathbb{P})$ is known as a *filtered probability space*.

**Definition 2.8** (Martingale)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. A *martingale* with respect to the filtration $(\mathcal{M}_t)_{t\geq 0}$ is a stochastic process $X_t$ such that

1. $X_t$ is $\mathcal{M}_t$-measurable $\forall t$
2. $\mathbb{E}[|X_t|] < \infty \ \forall t$
3. $\mathbb{E}[X_s \mid \mathcal{M}_t] = X_t \ \forall s \geq t$

In conclusion, a martingale is a stochastic process such that the conditional expected value of an observation at a future time $s$, given all the past information up to time $t$, is equal to the value of the observation at time $t$.

**Definition 2.9** (Supermartingale and submartingale)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space.

A *supermartingale* (resp., *submartingale*) with respect to the filtration $(\mathcal{M}_t)_{t\geq 0}$ is a stochastic process $X_t$ such that

1. $X_t$ is $\mathcal{M}_t$-measurable $\forall t$
2. $\mathbb{E}[|X_t|] < \infty \ \forall t$
3. $\mathbb{E}[X_s \mid \mathcal{M}_t] \leq (\geq) \ X_t \ \forall s \geq t$

## 2.3 Markov property

### 2.3.1 Conditional expectation and Markov

**Definition 2.10** (Itô diffusion)**.** A (time-homogeneous) Itô diffusion is a stochastic process $X_t(\omega) = X(t, \omega) : [0, \infty) \times \Omega \to \mathbb{R}^n$ satisfying a stochastic differential equation (SDE, from now on) of the form:

$$dX_t = b(X_t)dt + \sigma(X_t)dB_t, \quad t \geq s; \ X_s = x, \tag{1}$$

where $B_t$ is an $m$-dimensional Brownian motion and $b : \mathbb{R}^n \to \mathbb{R}^n$, $\sigma : \mathbb{R}^n \to \mathbb{R}^{n\times m}$ satisfy:

$$|b(x) - b(y)| + |\sigma(x) - \sigma(y)| \leq D|x - y|; \quad \forall x, y \in \mathbb{R}^n, \tag{2}$$

where $|\sigma^2| = \sum |\sigma_{ij}^2|$.

To see that the solution of the SDE is a time-homogeneous distribution in detail, please refer to the SDE book [10].

These stochastic processes satisfy the Markov property. Before defining the property, we will need to define the conditional expectation. For the basic properties and/or details, see [10] or [8].

**Definition 2.11** (Conditional expectation)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space and let $X : \Omega \to \mathbb{R}^n$ be a random variable such that $\mathbb{E}[|X|] < \infty$. Let $\mathcal{H} \subset \mathcal{F}$ be a sub-$\sigma$-algebra. Then, the conditional expectation of $X$ given $\mathcal{H}$, $\mathbb{E}[X|\mathcal{H}]$ is the unique function (almost surely) from $\Omega$ to $\mathbb{R}^n$ that satisfies:

(1) $\mathbb{E}[X \,|\, \mathcal{H}]$ is $\mathcal{H}$-measurable.

(2) $\int_H \mathbb{E}[X \,|\, \mathcal{H}] dP = \int_H X dP$, for all $H \in \mathcal{H}$.

The existence and uniqueness of $\mathbb{E}[X \,|\, \mathcal{H}]$ comes from the Radon-Nikodym theorem.

Also, we can define the probability laws $Q^x$ of $\{X_t\}_{t\geq 0}$. That is, $Q^x$ gives the distribution of $\{X_t\}_{t\geq 0}$ assuming $X_0 = x$. More formally, let $\mathcal{M}_\infty$ be the $\sigma$-algebra generated by $\omega \to X_t(\omega) = X_t^y(\omega)$, where $X_t^y$ means that $X_0 = y$. We then define $Q^x$ on the elements of $\mathcal{M}$:

$$Q^x[X_{t_1} \in E_1, \dots, X_{t_k} \in E_k] = \mathbb{P}(X_{t_1}^x \in E_1, \dots, X_{t_k}^x \in E_k)$$

where $E_i \subset \mathbb{R}^n$ are Borel sets for $1 \leq i \leq k$. Let $\mathcal{F}_t^{(m)}$ be the $\sigma$-algebra generated by $\{B_r : r \leq t\}$ and let $M_t$ be the $\sigma$-algebra generated by $\{X_r : r \leq t\}$.

**Definition 2.12** (Markov property)**.** Let $f$ be a bounded Borel function from $\mathbb{R}^n$ to $\mathbb{R}$. Let $X_t$ be a stochastic process. We say that $X_t$ satisfies the Markov property with respect to the family of $\sigma$-algebra $\{\mathcal{F}_t^{(m)}\}_{t\geq 0}$ if, for $t, h > 0$:

$$\mathbb{E}^x[f(X_{t+h}) \,|\, \mathcal{F}_t^{(m)}]_{(\omega)} = \mathbb{E}^{X_t(\omega)}[f(X_h)].$$

where $\mathbb{E}^x$ is the expectation with respect to the probability measure $Q^x$ and hence means $\mathbb{E}[f(X_h^y)]$. In the right hand side there is the function $\mathbb{E}^y[f(X_h)]$ evaluated at $y = X_t(\omega)$.

A stronger condition, the strong Markov property, is satisfied if the Markov condition holds even if the time $t$ is replaced by a random time $\tau(\omega)$ called stopping time or Markov time.

### 2.3.2 Markov chains

A Markov process is a stochastic process that satisfies the Markov property.

A Markov Chain is a stochastic process where the probability of moving to a new state depends only on the current state. It is characterized by a set of states and a transition probability matrix that specifies the probability of moving from one state to another. The transition probability matrix is a square matrix where each element represents the probability of moving from one state to another in a single time step. The sum of the probabilities in each row of the matrix is always one.

An example of a Markov Chain is the weather, in the following sense. We can model the weather as a Markov Chain with two states, rainy and sunny. The transition probability matrix for this Markov Chain might look like:

| | Sunny | Rainy |
|---|---|---|
| Sunny | 0.7 | 0.3 |
| Rainy | 0.4 | 0.6 |

This is interpreted as: if today it is sunny, there is a 70% chance that tomorrow it will be sunny as well and a 30% chance that it will rain. On the other hand, if today it is raining, there is a 40% chance that tomorrow it will be sunny and a 60% chance that it will be rainy tomorrow.

## 2.4 The Optimal Stopping Time problem

**Definition 2.13** (Discrete Markov time of a stochastic process)**.** Consider a stochastic process $X_t$ on a filtered space $(\Omega, \mathcal{F}, (\mathcal{M}_t)_{t\geq 0}, \mathbb{P})$. We define a *discrete Markov time* for $X_t$ as a non-negative integer-valued random variable $\tau$ satisfying that for each $t \geq 0$, the event $\{\tau = t\}$ depends only on $\{X_0, X_1, \dots, X_t\}$ and not on $\{X_{t+s} : s \geq 1\}$.

**Definition 2.14** (Stopping time)**.** A discrete stopping time is a Markov time $\tau$ such that it has probability one of being finite, i.e. $\mathbb{P}(\tau < \infty) = 1$.

Let $X = (X_n)_{n=0}^N$ be a $\mathbb{R}^d$-valued discrete time Markov process on a filtered probability space $(\Omega,\mathcal{F}, (\mathcal{F}_n)_{n=0}^N, \mathbb{P})$.

The problem we are looking to solve is given by:

$$V = \sup_{\tau\in\mathcal{T}} \mathbb{E}g(\tau, X_\tau) \tag{3}$$

where $\mathcal{T}$ is the set of all $X$-stopping times and $g : \{0, 1, \dots, N\} \times \mathbb{R}^d \to \mathbb{R}$ is a measurable function.

To make sure that (3) is well-defined and admits solution, we will assume that $g$ satisfies what we will refer to as our *integrability condition*:

$$\mathbb{E}\,|g(n, X_n)| < \infty \quad \text{for all } n \in \{0, 1, \dots, N\}.$$

If $\mathbb{E}|g(\tau, X_\tau)| = \infty$ for some $\tau \in \{0, 1, \dots, N\}$ the problem is not well-posed for $\tau$ because the reward function is not defined and thus we cannot concrete a solution for when it comes to the method of backward induction (refer to Appendix A). Also, note that $V$ is a martingale by assuming $\mathbb{E}g < \infty$ $\forall n$ or $\mathbb{E}g > -\infty$, then for simplicity we assume $\mathbb{E}|g| < \infty$.

Later on, we will derive confidence intervals of the solution $V$, and so, we will need to make a stronger assumption:

$$\mathbb{E}\left[g(n, X_n)^2\right] < \infty \quad \text{for all } n \in \{0, 1, \dots, N\}.$$

# 3. Stopping time in terms of stopping decisions

## 3.1 Representation

We will show that the decision to stop the Markov process $(X_n)_{n=0}^N$ can be made according to a sequence of functions $\{f_n(X_n)\}_{n=0}^N$, $f_n : \mathbb{R}^d \to \{0,1\}$.

Considering

$$V_n = \sup_{\tau \in \mathcal{T}_n} \mathbb{E} g(\tau, X_\tau) \tag{4}$$

where $\mathcal{T}_n$ is the set of all $X$-stopping times such that $n \leq \tau \leq N$. We observe that at time $N$, $\mathcal{T}_N = \{N\}$, and so, the time $N$ optimal stopping time is $\tau_N = N$.

If we consider the 0-1 stopping time decisions $\{f_n(X_n)\}_{n=0}^N$ with $f_N = 1$, we can write the $N$-th time optimal stopping time as a function of the 0-1 stopping decisions as $\tau_N = N \cdot f_N(X_N)$.

We aim to do likewise for $0 \leq n \leq N-1$. The reference article proposes the following equation:

$$\tau_n = \sum_{k=n}^{N} k f_k(X_k) \prod_{j=n}^{k-1} (1 - f_j(X_j)) \tag{5}$$

with $f_N \equiv 1$. We will prove that is a stopping time in $\mathcal{T}_n$. Afterwards, with Theorem 3.1, we will prove that it is also an optimal of (4).

Observe the following. Let there be $p, q$ such that $0 \leq p, q \leq N$. Also, assume, without loss of generality, that $p \leq q-1$. Then, we can derive from (5) the following equivalences:

$$\{\tau_p = q\} \iff \{f_q(X_q) \prod_{k=p}^{q-1} (1 - f_k(X_k)) = 1\} \iff$$

$$\iff f_q(X_q) = 1 \quad \text{and} \quad f_k(X_k) = 0 \quad \forall k \, \text{s.t.} \, p \leq k \leq q-1.$$

One could think what would happen if $f_i(X_i) = 1$ for more than one case. Formally, define $I = \{i \in \mathbb{N} : f_i(X_i) = 1\}$ and suppose $|I| > 1$. In that case, observe that the result would be exactly the same as for the first "success", i.e. the minimum $j \in I$, as the $(1 - f_j(X_j))$ component in (5) would map to 0 all the next successes after $j$.

Hence, notice that this is the only configuration of $f_i$ possible to get $\tau_p = q$. It is also remarkable that, up to $q$, the $X_i$ that participate are also up to $q$, and so, it is an event included in the filtration $\mathcal{F}_q$.

Therefore, $\tau_n$ is in $\mathcal{T}_n$.

## 3.2 Optimality

To show that it is optimal, let us present the following theorem.

**Theorem 3.1.** *For a given $n \in \{0, 1, \dots, N-1\}$, let $\tau_{n+1}$ be a stopping time in $\mathcal{T}_{n+1}$ of the form*

$$\tau_{n+1} = \sum_{m=n+1}^{N} m f_m(X_m) \prod_{j=n+1}^{m-1} (1 - f_j(X_j))$$

*for measurable functions $f_{n+1}, \dots, f_N : \mathbb{R}^d \to \{0,1\}$, with $f_N \equiv 1$.*

*Then, there exists a measurable function $f_n : \mathbb{R}^d \to \{0,1\}$ such that the stopping time $\tau_n \in \mathcal{T}_n$ given by (5) satisfies*

$$\mathbb{E}g(\tau_n, X_{\tau_n}) \geq V_n - (V_{n+1} - \mathbb{E}g(\tau_{n+1}, X_{\tau_{n+1}}))$$

*where $V_n$ and $V_{n+1}$ are the optimal values defined in (4).*

Before proving this theorem, let us introduce the Doob-Dynkin lemma in its conditional expectation version. For other versions and variations please refer to [13].

This lemma will be useful to prove the optimality of this "discretized" stopping time. First, let us introduce some concepts to contextualise the lemma.

**Definition 3.2** ($\sigma$-finite measure)**.** Let $(\Omega, \varepsilon)$ be a measurable space with a measure $P$. We say that this measure is $\sigma$-finite if there is a countable number of measurable sets $B_k$ with finite measure such that $\Omega = \bigcup_k B_k$.
Also, if there is a measurable space $(\Omega_Y, \mathcal{F}_Y)$ with the law $P_Y$ of a measurable $Y : \Omega \to \Omega_Y$ such that $P_Y(A) = P(Y \in A) = P(\{\omega \mid Y(\omega) \in A\}$ for $A \in \mathcal{F}$, we say that $Y$ is $\sigma$-finite if $P_Y$ is $\sigma$-finite.

Let us situate on the second case: having $(\Omega_Y, \mathcal{F}_Y)$ measurable space with $P_Y$ law of a measurable function $Y$. Also, let the $\sigma$-algebra of $Y$ be $\varepsilon_Y = \{Y^{-1}(A) : A \in \mathcal{F}_Y\}$. With this, let us announce the conditional expectation version of the Doob-Dynkin lemma.

**Lemma 3.3** (Doob-Dynkin lemma, conditional expectation version)**.** *Let $\Gamma : \Omega \to (0, \infty]$ and $Y : \Omega \to \Omega_Y$ be measurable. If $Y$ is $\sigma$-finite with $\sigma$-algebra $\varepsilon_Y$, then there exists a unique (a.e.) measurable $\phi : \Omega_Y \to \Omega_\Gamma$ such that $\mathbb{E}[\Gamma|\varepsilon_Y] = \phi(Y)$.*

*Proof for the conditional expectation Doob-Dynkin.* For the proof, we need the Radon-Nikodym theorem.

**Lemma 3.4** (Radon-Nikodym theorem)**.** *Let $(X, \Sigma)$ be a measurable space, $\mu : \Sigma \to \mathbb{R}$ a $\sigma$-finite measure ($\Sigma$ is a countable union of measurable sets with finite measure) and $\nu : \Sigma \to \mathbb{R}$ a signed $\sigma$-finite measure that is absolutely continuous with respect to $\mu$. Then, there exists $f$ a measurable function over $(X, \Sigma)$ such that $\nu(A) = \int_A f d\mu \; \forall A \in \Sigma$.*
*Also, if $\nu(A) = \int_A f d\mu$ is also $\nu(A) = \int_A g d\mu \; \forall A \in \Sigma$ then $f = g$ almost everywhere (a.e.).*

This theorem gives a way of expressing any measure $\nu$ with respect to another measure $\mu$ in the same space.

Intuitively, for example, if $f$ represented mass density and $\mu$ was the Lebesgue measure in three-dimensional space $\mathbb{R}^3$, then $\nu(A)$ would equal the total mass in a spatial region $A$.

Back to the proof of Doob-Dynkin, using the Radon-Nikodym theorem, we obtain a unique (a.e.) $\phi$ such that $\mathbb{E}[\Gamma \cap (Y \in A)]$ for all measurable $A$ is

$$\mathbb{E}[\Gamma \cap (Y \in A)] \overset{(1)}{=} \int_A \phi(Y) P_Y(dy) \overset{(2)}{=} \int_{(Y \in A)} \phi(Y) P(dw).$$

(1) Since the left-hand side defines a measure which is absolutely continuous with respect to $P_Y$ which is assumed to be $\sigma$-finite.

(2) By the general change-of-variables theorem.

This sequence of equalities gives us $\mathbb{E}[\Gamma|\varepsilon_Y] = \phi(Y)$. □

Now we have the necessary tools to prove Theorem 3.1.

*Proof of (3.1).* Let us set an arbitrary stopping time $\tau \in \mathcal{T}_n$, and defining $\varepsilon = V_{n+1} - \mathbb{E}g(\tau_{n+1}, X_{\tau_{n+1}})$.

Applying Doob-Dynkin's lemma; since $g$ is integrable, then there exists $h_n$ measurable such that:

$$h_n(X_n) = \mathbb{E}[g(\tau_{n+1}, X_{\tau_{n+1}})|X_n].$$

By (5), the following is also a measurable function of $X_{n+1}, \dots, X_N$:

$$g(\tau_{n+1}, X_{\tau_{n+1}}) = \sum_{k=n+1}^{N} g(k, X_k)\mathbb{1}_{\{\tau_{n+1}=k\}} = \sum_{k=n+1}^{N} g(k, X_k)\mathbb{1}_{\{f_k(X_k)\prod_{j=n}^{k-1}(1-f_j(X_j))=1\}}$$

It is measurable because it can be rewritten as a sum of measurable functions of $X_{n+1}, \dots, X_N$. Since $(X_n)_{n=0}^N$ is Markovian:

$$h_n(X_n) = \mathbb{E}[g(\tau_{n+1}, X_{\tau_{n+1}})|X_n] = \mathbb{E}[g(\tau_{n+1}, X_{\tau_{n+1}})|\mathcal{F}_n]$$

Now, we can assert that these sets are in $\mathcal{F}_n$:

$$D = \{g(n, X_n) \geq h_n(X_n)\} \quad \text{and} \quad E = \{\tau = n\}$$

and that $\tau_n = n\mathbb{1}_D + \tau_{n+1}\mathbb{1}_{D^c}$ is in $\mathcal{T}_n$ and $\tilde{\tau} = \tau_{n+1}\mathbb{1}_E + \tau\mathbb{1}_{E^c}$ is in $\mathcal{T}_{n+1}$. Hence,

$$\mathbb{E}g(\tau, X_{\tau_{n+1}}) = V_{n+1} - \varepsilon = \sup_{\tau\in\mathcal{T}_n} \mathbb{E}g(\tau, X_\tau) - \varepsilon \geq \mathbb{E}g(\tilde{\tau}, X_{\tilde{\tau}}) - \varepsilon.$$

And so,

$$\mathbb{E}[g(\tau, X_{\tau_{n+1}})\mathbb{1}_{E^c}] \geq \mathbb{E}[g(\tilde{\tau}, X_{\tilde{\tau}})\mathbb{1}_{E^c}] - \varepsilon = \mathbb{E}[g(\tau, X_\tau)\mathbb{1}_{E^c}] - \varepsilon. \tag{6}$$

Notice that it holds since $\mathbb{1}_{E^c} = 1 \iff \tilde{\tau} = \tau$. We then obtain the following:

$$\mathbb{E}g(\tau_n, X_{\tau_n}) \overset{(1)}{=} \mathbb{E}[g(n, X_n)\mathbb{1}_D + g(\tau_{n+1}, X_{\tau_{n+1}})\mathbb{1}_{D^c}] \overset{(2)}{=} \mathbb{E}[g(n, X_n)\mathbb{1}_D + h_n(X_n)\mathbb{1}_{D^c}] \overset{(3)}{\geq}$$
$$\overset{(3)}{\geq} \mathbb{E}[g(n, X_n)\mathbb{1}_E + h_n(X_n)\mathbb{1}_{E^c}] \overset{(4)}{=} \mathbb{E}[g(n, X_n)\mathbb{1}_E + g(\tau_{n+1}, X_{\tau_{n+1}})\mathbb{1}_{E^c}] \overset{(5)}{\geq}$$
$$\overset{(5)}{\geq} \mathbb{E}[g(n, X_n)\mathbb{1}_E + g(\tau, X_\tau)\mathbb{1}_{E^c}] - \varepsilon = \mathbb{E}g(\tau, X_\tau) - \varepsilon.$$

(1) By definition of $\tau_n$.

(2) By definition of $h_n(X_n)$.

(3) Suppose $g(n, X_n) \geq h_n(X_n)$ (if and only if $\mathbb{1}_D = 1$). Then,

$$\mathbb{E}[g(n, X_n)] \geq \mathbb{E}[g(n, X_n)\mathbb{1}_E + h_n(X_n)\mathbb{1}_{E^c}]$$

holds having either $\mathbb{1}_{E^c} = 1$ or $\mathbb{1}_E = 1$. Notice that supposing $\mathbb{1}_D = 0$, it holds with the same reasoning.

(4) By definition of $h_n(X_n)$.

(5) By (6).

Notice that $\tau$ is arbitrary, and so, in particular we have that $\mathbb{E}g(\tau_n, X_{\tau_n}) \geq V_n - \varepsilon$, proving the theorem.

To complete the proof, we have to show that $\tau_n$ found here is the same as in (5). Let us define $f_n : \mathbb{R}^d \to \{0, 1\}$ such that:

$$f_n(x) = \begin{cases} 1 & \text{if } g(n,x) \geq h_n(x) \\ 0 & \text{if } g(n,x) < h_n(x) \end{cases}$$

This way, $\mathbb{1}_D = f_n(X_n)$, and so:

$$\tau_n \overset{(1)}{=} nf_n(X_n) + \tau_{n+1}(1 - f_n(X_n)) \overset{(2)}{=} \sum_{k=n}^{N} kf_k(X_k) \prod_{j=n}^{k-1}(1 - f_j(X_j)).$$

(1) By definition of $\tau_n$.

(2) $\tau_{n+1}$ has the form the theorem preamble requires.

$\square$

*Remark* 3.5. Theorem 3.1 shows that the stopping times in terms of 0-1 stopping decision times can be used to compute $V_n$.

In fact, the inequality from the conclusion of the theorem is equivalent to:

$$\sup_{\tau \in \mathcal{T}_{n+1}} \mathbb{E}g(\tau, X_\tau) - \mathbb{E}g(\tau_{n+1}, X_{\tau_{n+1}}) \geq \sup_{\tau \in \mathcal{T}_n} \mathbb{E}g(\tau, X_\tau) - g(\tau_n, X_{\tau_N}).$$

However, we know that $\tau_N \equiv N \cdot f(X_N)$, and so by backwards induction, $\tau_n$ will provide a sufficient optimal stopping time.

*Remark* 3.6. From Theorem 3.1, the overall optimal stopping time corresponding to $V = \sup_{\tau \in \mathcal{T}} \mathbb{E}g(\tau, X_\tau)$ is given by:

$$\tau = \sum_{n=1}^{N} nf_n(X_n) \prod_{k=0}^{n-1}(1 - f_k(X_k)). \tag{7}$$

Now, we will draw upon neural networks to represent any stopping time in terms of $\{f_i\}$ stopping decisions in the form of (7), and presenting promising candidates $\{f_i\}_i$.

# 4. Neural Networks approximation

## 4.1 Feed-forward neural networks

The feed-forward neural network is one of the most popular methods to approximate a multivariate nonlinear function. Their distinctive feature is the hidden layers, in which input variables activate some nonlinear transformation function, producing a continuous signal to output nodes. Deep Neural Networks are just neural networks with more than one hidden layer.

This way, we ensemble a very efficient parallel distributed model of nonlinear statistical processes. This

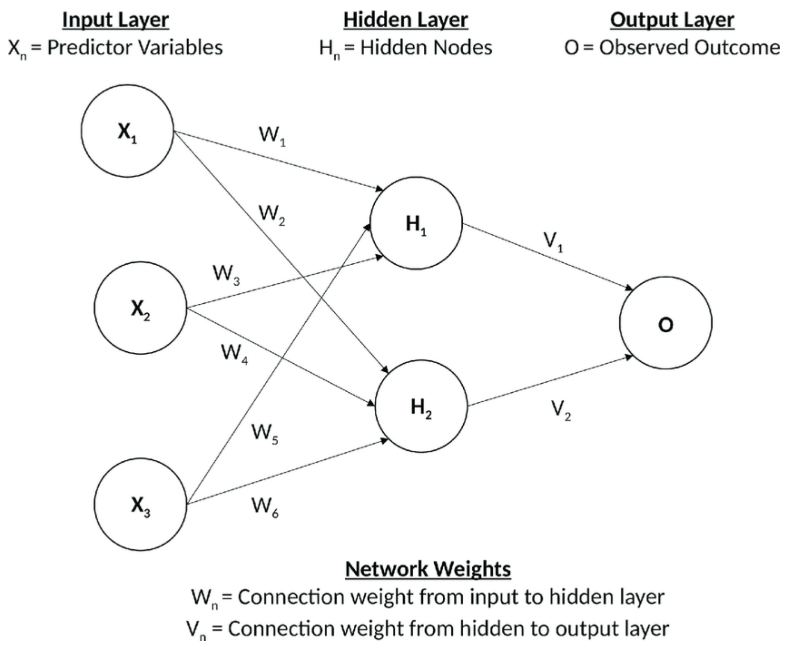


Figure 1: Example of a 3-2-1 feed forward neural network with one hidden layer. **Source**: `https://doi.org/10.1371/journal.pone.0214365.g001`

figure shows a network with three input nodes $x_i$, one output node $o$, and two hidden nodes or neurons $H_j$. Each neuron $h_j$ receives the input information and goes active, transmitting a signal to the output if a certain linear combination of the inputs $z_j = \sum_i \omega_{i,j} x_i - \alpha_j > 0$, where $\omega_{i,j}$ are the weights (notice that, for cleanliness, it is better to index the weights with two indices). The actual signal is produced by an activation function $\mathcal{L}$ applied to $z_j$, with no activation if $z_j < 0$ and, consequently, the signal is 0. When there is activation, the value of the signal is $0 < \mathcal{L}(z_j) < 1$.

Composing a linear combination of the possible values of signals transmitted by the hidden layer, plus some value $\varepsilon$ to account for the bias in the connection and another activation function $f$ for the output node, we obtain a mathematical expression for the basic feed-forward network:

$$o = f\left(\sum_j \theta_j \mathcal{L}\left(\sum_i \omega_{i,j} x_i + \alpha_j\right) + \varepsilon\right) \tag{8}$$

If we also allow direct connections from input to output nodes, then the general expression for these neural networks is:

$$o = f\left(\sum_i \phi_i x_i + \sum_j \theta_j \mathcal{L}\left(\sum_i \omega_{i,j} x_i + \alpha_j\right) + \varepsilon\right) \tag{9}$$

where $x_i$ are the input values, $o$ is the output, $\mathcal{L}$ is the activation function for each of the hidden nodes, $\alpha_j$ are the thresholds, $\omega_{i,j}$ the weights for the connection between $i$ and $j$, $\phi_i$ and $\theta_j$ are the weights for the linear combinations, $\varepsilon$ is the bias in the connections and $f$ is the activation function for the output node.

In other words, one can describe the neural network in Figure 1, for example, as the composite function $f(x) = o$, with $x = (x_1, x_2, x_3)$ and $H = (H_1, H_2)$, such that:

$$f^\theta = \mathcal{L}_2 \circ a_2 \circ \mathcal{L}_1 \circ a_1$$

where:

- $a_i$ are linear functions. $a_1 : \mathbb{R}^3 \mapsto \mathbb{R}^2$ of the form $a_1(x) = W_1 \cdot x + \alpha$, and $a_2 : \mathbb{R}^2 \mapsto \mathbb{R}^3$, of the form $a_2(H) = W_2 \cdot H + \varepsilon$, where $W_1 \in \mathbb{R}^{2\times 3}$, $W_2 \in \mathbb{R}^{3\times 2}$, $\alpha \in \mathbb{R}^2$, $\varepsilon \in \mathbb{R}$.
- $\mathcal{L}_i$ are the activation functions. $\mathcal{L}_1$ corresponding to the hidden layer and $\mathcal{L}_2$ corresponding to the output layer.
- $\theta$ in $f^\theta$ refers to the parameters of the model that include the weights and the biases $\theta = \{W_1, W_2, \alpha, \varepsilon\}$.

We can do a generalization of examples such as Figure 1 and equivalent to expression (9).

$$F^\theta = \psi \circ a_I^\theta \circ \phi_{q_{I-1}} \circ a_{I-1}^\theta \circ \cdots \circ \phi_{q_1} \circ a_1^\theta, \tag{10}$$

where

- $I, q_1, q_2, \ldots, q_{I-1}$ are positive integers that specify the depth of the network and the number of nodes in the hidden layers.
- $a_1^\theta : \mathbb{R}^d \to \mathbb{R}^{\theta q_1}, \ldots, a_{I-1}^\theta : \mathbb{R}^{q_{I-2}} \to \mathbb{R}^{q_{I-1}}$ and $a_{q_{I-1}}^\theta : \mathbb{R}^{q_{I-1}} \to \mathbb{R}$ are affine functions.
- Let $j \in \mathbb{N}$, $\phi_j : \mathbb{R}^j \to \mathbb{R}^j$ is the component-wise ReLU activation function given by $\phi_j(x_1, \ldots, x_j) = (x_1^+, \ldots, x_j^+)$.
- $\psi : \mathbb{R} \to (0,1)$ is the logistic function $\psi(z) = e^x/(1+e^x) = 1/(1+e^{-x})$.

The parameter $\theta \in \mathbb{R}^q$ of $F^\theta$ is built upon the representation of the affine functions

$$a_i^\theta(x) = A_i x + b_i, \qquad i = 1, \ldots, I.$$

The dimension of the parameter space is, then:

$$q = \begin{cases} d+1 & \text{if } I = 1 \\ 1 + q_1 + \cdots + q_{I-1} + dq_1 + \cdots + q_{I-2}q_{I-1} + q_{I-1} & \text{if } I \geq 2 \end{cases}$$

Once established the model's architecture, to predict outputs given some inputs, we need to train the model, although, to evaluate the method and prevent overfitting (that is, the model explains very well the training data but performs badly with new unseen data), the known data is first split into *train data* and *test data*. The first is used to train the model while the latter is used to evaluate it. Usually, it is a 70/30 split.The known data is usually of the form $\{(\vec{x_i}, y_i) : \vec{x_i} \in \mathbb{R}^n\}$ where $y_i$ is the real value of the predicted variable for the set $\vec{x_i}$ of observations of the explicative variables (inputs). These pairs of inputs and outputs are fed into the model in order to learn the optimal $\theta$. Optimal refers to minimizing some loss function (or maximizing a reward function), denoted $C$. Usually, it is the Mean Square Error (MSE):

$$C(\theta) = \frac{1}{2M} \sum_{m=1}^{M} \|o^\theta(x^m) - y_m\|^2,$$

although for this problem we will use a custom loss function.

The minimization of $C$ is usually done with a Gradient Descent method. The vanilla gradient descent has update steps of the form $\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta C(\theta_t)$, where $\eta$ is called *learning rate*.

We approximate $\nabla_\theta C(\theta_t)$ at each iteration using a differentiation technique called backpropagation, a method to efficiently move backwards through the neural network graph to compute each $dC/d\theta_i$. The training is completed when the gradient descent algorithm is completed, and so, $\nabla_\theta C(\theta_t) < \delta$ where $\delta$ is a pre-selected threshold close to 0. After the model is trained, we evaluate the algorithm with a new data set, the test data.

## 4.2 The universal approximation property

**Theorem 4.1** (Universal approximation property)**.** *A feed-forward network with one hidden layer of large enough width and a squashing activation function (i.e. activation functions that squash the input into a small range, for example the sigmoid, to* $(0, 1)$*) can approximate any integrable function to any accuracy.*

*Proof.* For the proof, please refer to [6]. However, we are going to provide an intuition to it, attributed to Bruno Després [3]:
*Remark* 4.2. Let $f \in \mathcal{C}^1(\mathbb{R})$. Then,

$$f(x) \overset{(1)}{=} \int_{-\infty}^{x} f'(y)dy \overset{(2)}{=} \int_{\mathbb{R}} H(x-y)f'(y)dy \overset{(3)}{\approx} \sum_{j=-J}^{J} \phi\left(\frac{x}{\varepsilon} - \frac{j\Delta x}{\varepsilon}\right) f'(j\Delta x)\Delta x$$

$$\overset{(4)}{=} \sum_{j=-J}^{J} \omega_j \cdot \phi(a_j x + b_j)$$

(1) By the fundamental theorem of calculus.

(2) $H$ is the Heaviside function: $H(z) = 0$ if $z < 0$ and $H(z) = 1$ if $z \geq 0$. Thus, when $z = x - y < 0$, that is, when $y > x$, the integral is 0, making of $H$ a sort of characteristic function.

(3) As we have seen above, the sigmoid is the smooth, continuous version of $\mathbb{1}_{[0,\infty)}(x) = H(x)$. Also, we can discretise the integral with a quadrature, with $J$ sufficiently large.

(4) Defining the weights $\omega_j := f'(j\Delta x)\Delta x$, $a_j := 1/\varepsilon$ and $b_j := j\Delta x/\varepsilon$.

Therefore, we have shown an intuition of approximating a derivable function by a composition of an activation function (the sigmoid) and an affine, linear function, which are the building blocks of a neural network.

□

A universal property theorem that will be key to prove a theorem that we will see afterwards is the following:

**Theorem 4.3** (Multilayer feedforward neural networks theorem). *Let $M$ denote the set of functions which are in $L^{\infty}_{loc}(\mathbb{R})$ and such that the closure of the set of points of discontinuity is of zero Lebesgue measure. Let $\sigma \in M$. Let*

$$\Sigma_n = span\{\sigma(\omega \cdot x + \theta) \, : \, \omega \in \mathbb{R}^n \, , \, \theta \in \mathbb{R}\}$$

*Then, $\Sigma_n$ is dense in $\mathcal{C}(\mathbb{R}^n)$ if and only if $\sigma$ is not an algebraic polynomial (almost everywhere).*

*Proof.* The detailed proof and discussion, organized in steps, can be found in [7], by Leshno et al. (1993). We can conclude, from the theorem and the overall article from Leshno et al., that for every continuous function $g \in \mathcal{C}(\mathbb{R})$ and every compact set $K \subset \mathbb{R}^n$, there is a function $f \in \Sigma_n$ such that $f$ is a good approximation to $g$ on $K$.

In other words, there exists a series of functions $f_j \in \Sigma_n$ such that, for any compact set $K \subset \mathbb{R}^n$ and for any function $g \in \mathcal{C}(\mathbb{R})$:

$$\lim_{j \to \infty} \|g - f_j\|_{L^\infty} = 0$$

□

## 4.3 Back to stopping times

The numerical method for problem (4) consists in iteratively approximating optimal stopping decisions $\{f_n\}_{n=0}^{N}$ with a sequence of neural networks $f^\theta : \mathbb{R}^d \to \{0,1\}$ with parameters $\theta_n \in \mathbb{R}^q$.

To do this, we start with the terminal stopping decision $f_N \equiv 1$ and we proceed by backward induction. In particular, we let $n \in \{0, 1, \dots, N-1\}$, and assume parameter values $\theta_{n+1}, \theta_{n+2}, \dots, \theta_N \in \mathbb{R}^q$ have been found such that $f^{\theta_N} \equiv 1$ and the stopping time

$$\tau_{n+1} = \sum_{m=n+1}^{N} m f^{\theta_m}(X_m) \prod_{j=n+1}^{m-1} (1 - f^{\theta_j}(X_j))$$

produces an expected value $\mathbb{E}g(\tau_{n+1}, X_{\tau_{n+1}})$ close to the optimum $V_{n+1}$.

Note that $f^\theta$ takes values in $\{0,1\}$, so it is not a direct gradient-based optimization method. Therefore, we introduce a feed-forward neural network $F^\theta : \mathbb{R}^d \to (0,1)$ of the form (10).

After finding the optimal $\theta_n$ by training $F^\theta$ (we will specify this methodology afterwards), we define the network $f^{\theta_n} : \mathbb{R}^d \to \{0,1\}$ as follows:

$$f^{\theta_n} = \mathbb{1}_{(0,\infty)} \circ a_I^\theta \circ \phi_{q_I - 1} \circ a_{I-1}^\theta \circ \cdots \circ \phi_{q_1} \circ a_1^\theta \tag{11}$$

*Remark* 4.4. Notice that the only difference between $F^{\theta_n}$ and $f^{\theta_n}$ is the final non-linearity. $F^{\theta_n}$ produces a stopping probability in $(0,1)$. On the other hand, the output of $f^{\theta_n}$ is a stopping decision given by 0 or 1, depending on whether $F^{\theta_n}$ takes a value below or above $1/2$. This is why we will refer to $F^{\theta_n}$ as the soft stopping decision and $f^{\theta_n}$ as the hard stopping decision.

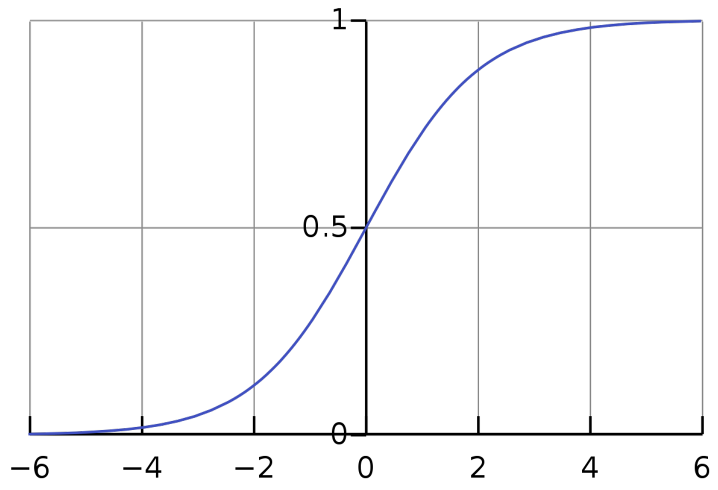


Figure 2: Sigmoid or Logistic Regression function. **Source**: Qef (talk) - Created from scratch with gnuplot, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=4310325`

Notice that the domain of the sigmoid $\psi(x) = \frac{1}{1+e^{-x}}$ is $(-\infty, \infty)$, but the range is $(0, 1)$, since it is injective and $\lim_{x\to-\infty} \psi(x) = 0$ and $\lim_{x\to\infty} \psi(x) = 1$. We can translate this by thinking of $F^{\theta_n} = \psi(x)$ where $x$ is the rest of the neural network.

In this case, we can just take limits and interpret $\mathbb{1}_{(0,\infty)}$ as the discrete counterpart of $\psi(x)$. Then $f^{\theta_n}$ can be thought of $\mathbb{1}_{(0,\infty)}(x)$ where $x$ is the rest of the neural network.

We will now prove that we can approximate the solution to problems of the form (4) using a sequence of neural networks $\{f^{\theta_n}\}$, where $f^{\theta_n}$ is of the form (11).

Observe the following. At each time step, our objective is to maximize our expected future reward. At time $n$, we know that, if we stop, $f_n(X_n) = 1$, we will receive a reward of $g(n, X_n)$. If we continue, that is, $f_n(X_n) = 0$, then, behaving optimally, we will receive a reward of $g(\tau_{n+1}, X_{\tau_{n+1}})$ by Theorem 3.1. In conclusion, $f_n$ can be intuitively thought of the probability that stopping at time $n$ is optimal .

Hence, our aim is to maximize over the set of measurable functions $f$ the following

$$\mathbb{E}\left[g(n, X_n)f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - f(X_n)\right] . \tag{12}$$

We now present a theorem that is going to show that we can replace $f$ in (12) with $f^\theta$. This allows us to maximize (12) with respect to $\theta \in \mathbb{R}^q$ of either $f^\theta$ or $F^\theta$, instead of finding the optimal function $f : \mathbb{R}^d \to \{0, 1\}$. So, the objective is to determine $\theta_n \in \mathbb{R}^q$ so that

$$\mathbb{E}\left[g(n, X_n)F^{\theta_n}(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - F^{\theta_n}(X_n)\right]$$

is close to the supremum for $\theta \in \mathbb{R}^q$.

Apart from showing that we can replace $f$ with $f^\theta$, it also shows that, for any depth $I \geq 2$, a neural network of the form (11) is flexible enough to decide on *close-to* optimal stopping decisions if it has a sufficient number of nodes.

**Theorem 4.5.** *Let $n \in \{0, 1, \dots, N-1\}$ and fix a stopping time $\tau_{n+1} \in \mathcal{T}_{n+1}$. Then, for every depth $I \geq 2$ and constant $\varepsilon > 0$, there exist positive integers $q_1, \dots, q_{I-1}$ such that*

$$\sup_{\theta \in \mathbb{R}^q} \mathbb{E}\left[g(n, X_n) f^{\theta}(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - f^{\theta}(X_n))\right] \geq$$

$$\geq \sup_{f \in \mathcal{D}} \mathbb{E}\left[g(n, X_n) f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - f(X_n))\right] - \varepsilon,$$

*where $\mathcal{D}$ is the set of all measurable functions $f : \mathbb{R}^d \to \{0, 1\}$.*

*Proof.* We fix $\varepsilon > 0$. By the integrability condition of $g$, we have $\mathbb{E}|g(n, X_n)| < \infty \; \forall n \in \{0, \dots, N\}$. Then, we can find a $\tilde{f} : \mathbb{R}^d \to \{0, 1\}$ such that $f \in \mathcal{D}$ that satisfies:

$$\mathbb{E}\left[g(n, X_n)\tilde{f}^{\theta}(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \tilde{f}^{\theta}(X_n))\right] \geq$$

$$\overset{(1)}{\geq} \sup_{f \in \mathcal{D}} \mathbb{E}\left[g(n, X_n) f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}}(1 - f(X_n))\right] - \varepsilon/4.$$

(1) Recall that the supremum is the lowest of the upper bounds. So, if we subtract any value to the supremum, then we have an upper bound $\bar{f}$ that will overcome the minimum of the upper bounds.

Next, let $\tilde{f} = \mathbb{1}_A$ for $A = \{x \in \mathbb{R}^d \mid \tilde{f}(x) = 1\}$. $A$ is a Borel set on $\mathbb{R}^d$. By the integrability condition of $g$, we define these Borel measures (measures defined on all Borel sets) on $\mathbb{R}^d$:

$$B \mapsto \mathbb{E}[|g(n, X_n)| \cdot \mathbb{1}_B(X_n)] \quad \text{and} \quad B \mapsto \mathbb{E}[|g(\tau_{n+1}, X_{\tau_{n+1}})| \cdot \mathbb{1}_B(X_n)]$$

Note that we always can find a compact set $K$ such that, in this case, $K \subseteq A$ and still

$$\mathbb{E}\left[g(n, X_n)\mathbb{1}_K(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_K(X_n))\right] \overset{(2)}{\geq}$$

$$\geq \sup_{f \in \mathcal{D}} \mathbb{E}\left[g(n, X_n) f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}}(1 - f(X_n))\right] - \varepsilon/4.$$

(2) Because of the tightness of compact sets.

We also can define a distance function $\rho_K : \mathbb{R}^d \to [0, \infty)$ by $\rho_K(x) = \inf_{y \in K} \|x - y\|_2$ and a sequence of continuous functions $k_j : \mathbb{R}^d \to [-1, 1]$, $j \in \mathbb{N}$ by

$$k_j(x) = \max\{1 - j \cdot \rho_K(x), -1\}$$

As this is the distance from $x$ to the closest point in $K$ (in the 2-norm sense), $k_j$ converges point-wise to $\lim_{j \to \infty} k_j(x) = (\mathbb{1}_K - \mathbb{1}_{K^c})(x)$, as, when $j$ grows, the only possibility for $k_j(x)$ to discard the $-1$ is to pick $1 - j \cdot \rho_K(x)$, and, if $j$ is fairly large, the only possibility is to have $\rho_K(x) = 0 \iff x \in K$.

We now apply Lebesgue's dominated convergence theorem:

**Theorem 4.6** (Lebesgue's dominated convergence theorem)**.** *Let $(f_n)_n$ be a sequence of complex-valued measurable functions on a measure space $(S, \Sigma, \mu)$. Suppose that the sequence converges point-wise to a function $f$ and is dominated by some Lebesgue integrable function $g$, i.e.:*

$$|f_n(x)| \leq g(x)$$

*for all $n$ in the index set and all points $x \in S$. Then $f$ is Lebesgue integrable and*

$$\lim_{n \to \infty} \int_S |f_n - f| d\mu = 0 \quad \textit{and} \quad \lim_{n \to \infty} \int_S f_n d\mu = \int_S f d\mu \tag{13}$$

In our case, we will choose $f_n \equiv f_j(x) = \mathbb{1}_{k_j(x)\geq 0}(x)$ and $f = \mathbb{1}_K$. Note that $\mathbb{1}_{k_j(x)\geq 0}$ converges point-wise to $\mathbb{1}_K$, since it just means keeping the $1 - j \cdot \rho_K(x)$ part in $k_j$, associated to being in $K$. So, from (13) and the fact that $\mathbb{1}_{k_j(x)}$ is monotonous (decreasing) with respect to $j$ down to $\mathbb{1}_K$:

$$\mathbb{E}\left[g(n, X_n)\mathbb{1}_{k_j(X_n)\geq 0} + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_{k_j(X_n)\geq 0})\right] \geq$$

$$\geq \mathbb{E}\left[g(n, X_n)\mathbb{1}_K(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_K(X_n))\right] - \varepsilon/4.$$

By Leshno et al. [7], since $k_j$ has only discontinuities at 0 measure points and is bounded, it can be approximated uniformly on compact sets by a function $h : \mathbb{R}^d \to \mathbb{R}$ where

$$h(x) = \sum_{i=1}^{r}(v_i^T x + c_i)^+ - \sum_{i=1}^{s}(\omega_i^T x + d_i)^+$$

for $r, s \in \mathbb{N}$, $v_1, \dots, v_r, \omega_1, \dots, \omega_s \in \mathbb{R}^d$ and $c_1, \dots, c_r, d_1, \dots, d_s \in \mathbb{R}$.

It directly follows that:

$$\mathbb{E}\left[g(n, X_n)\mathbb{1}_{h(X_n)\geq 0} + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_{h(X_n)\geq 0})\right] \geq$$

$$\geq \mathbb{E}\left[g(n, X_n)\mathbb{1}_{k_j(X_n)\geq 0} + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_{k_j(X_n)\geq 0})\right] - \varepsilon/4.$$

However, recall that $f^\theta = \mathbb{1}_{(0,\infty)} \circ a_I^\theta \circ \phi_{q_I - 1} \circ a_{I-1}^\theta \circ \cdots \circ \phi_{q_1} \circ a_1^\theta$. Therefore, we can express $\mathbb{1}_{(0,\infty)} \circ h$ as a neural network of the form $f^\theta$ for sufficiently large $q_1, q_2$, so that:

$$\mathbb{E}\left[g(n, X_n)f^\theta(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - f^\theta(X_n))\right] \geq$$

$$\geq \mathbb{E}\left[g(n, X_n)\mathbb{1}_{k_j(X_n)\geq 0} + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \mathbb{1}_{k_j(X_n)\geq 0})\right] - \varepsilon/4.$$

Following the chain of inequalities, carrying the four $\varepsilon/4$:

$$\sup_{\theta \in \mathbb{R}^q} \mathbb{E}\left[g(n, X_n)\tilde{f}^\theta(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - \tilde{f}^\theta(X_n))\right] \geq$$

$$\geq \sup_{f \in \mathcal{D}} \mathbb{E}\left[g(n, X_n)f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}}(1 - f(X_n))\right] - \varepsilon.$$

□

The following result is obtained directly from Theorem 3.1 and Theorem 4.5.

**Corollary 4.7.** *For any $\varepsilon > 0$ and stopping problem $\sup_{\tau \in \mathcal{T}} \mathbb{E}g(\tau, X_\tau)$, there exists $q_1, q_2 \in \mathbb{N}$ and neural network functions of the form (11) such that $f^{\theta_N} \equiv 1$ and the stopping time*

$$\hat{\tau} = \sum_{n=1}^{N} n f^{\theta_n}(X_n) \prod_{k=0}^{n-1}(1 - f^{\theta_k}(X_k)) \tag{14}$$

*satisfies $\mathbb{E}g(\hat{\tau}, X_{\hat{\tau}}) \geq \sup_{\tau \in \mathcal{T}} \mathbb{E}g(\tau, X_\tau) - \varepsilon$.*

In the next section, we will discuss in detail how to compute this neural network.

## 4.4 Implementation of the neural network

How to approximate $V$ in (3)? We will present an algorithm that, making use of our approximation of the optimal stopping time, we eventually get just a lower bound for $V_0 = sup_\tau \mathbb{E}[g(\tau, X_\tau)]$, so we will have to refer to a dual formulation of the optimal stopping problem.

### 4.4.1 Lower bound

Although the method is in [4], a revision of the algorithm was carried out to fill in the details not explained in the original. It was translated into a more coherent and comprehensible form and developed a version that conveyed the intended meaning more clearly.

First we will present the steps with the proper justification of them and then we will enumerate them in a compact pseudo-code algorithm. Afterwards, we will tackle applications and examples of the proposed method.

**Step 1**. Simulate $M$ independent paths of the Markov process $(X_n)_{n=0}^N$. These sample paths are denoted $(x_n^m)_{n=0}^N$ for $m = 1, \dots, M$. These $(x_n^m)$ will be our training data.

Set $f^{\theta_N}(x_N^m) \equiv 1$, $\forall m$, since we must stop at time $N$.

**Step 2**. Compute the rest of the $\{\theta_n\}_n$ recursively backwards for $n = N-1, \dots, 0$. We will do that using a loss function such that replicates (12) in the neural network, and so, at each epoch we will search for the best $\theta_n$ that maximizes (12) (or the one that minimizes the negative of (12)). We will specify this function below.

**Step 3**. Assume we are at time step $n$ (and so, we have already computed $f^{\theta_{n+1}}, \dots, f^{\theta_N}$). We can compute the $n+1$ optimal stopping time for each of the $m$ paths:

$$\hat{\tau}_{n+1}^m = \sum_{k=n+1}^{N} k f^{\theta_k}(x_n^m) \prod_{j=n+1}^{k-1} (1 - f^{\theta_j}(x_j^m)). \tag{15}$$

*Remark* 4.8. Notice that, if we stop at time $n$ along the path $m$ with probability $F^{\theta_n}$ and afterwards behave optimally with the next $f^{\theta_{n+1}}, \dots, f^{\theta_N}$, obtaining eventually the payoff of $g(\hat{\tau}_{n+1}^m)$ the realized reward for the $m$-th simulated path is:

$$r_n^m(\theta_n) = g(n, x_n^m) F^{\theta_n}(x_n^m) + g(\hat{\tau}_{n+1}^m, x_{\hat{\tau}_{n+1}^m}^m) \cdot (1 - F^{\theta_n}(x_n^m))$$

In other words, the virtual reward along the path $m$ is the probability of stopping at the time we are times the reward of stopping at this time plus the reward of following the path until we get to the optimal time.

*Remark* 4.9. Observe that this reward, for sufficiently large $M$, the averages of $r_n^m$ with respect to $m$ replicate

$$\mathbb{E}\left[g(n, X_n) f^\theta(X_n) + (1 - f^\theta(X_n)) g(\tau_{n+1}, X_{\tau_{n+1}})\right], \tag{16}$$

as we use a similar argument to that in (4.3).

**Step 4**. We can thus approximate (16) with

$$r_n^{MC} = \frac{1}{M}\sum_{m=1}^{M} r_n^m(\theta_n). \tag{17}$$

Notice that this is the Monte Carlo (MC) estimator for $\mathbb{E}_m[r_n^m]$.

Also, it is a smooth function with respect to $\theta$ almost everywhere (by definition of (17)). This is our desirable function to have optimized via a gradient descent algorithm with respect to $\theta_n$ (our loss function). After obtaining the optimal $\theta_n$, we will substitute $F^{\theta_n}$ with $f^{\theta_n}$.

We repeat this step for $n = N-1, \dots, 0$.

**Step 5**. Generate a new set of sample paths $(y_n^m)_{n=0}^N$ for $m = 1, \dots, M$, which will be our testing data.

**Step 6**. Using the $\{\theta_n\}_n$ that we found with our training data (that is, $(x_n^k)_{n=0}^N$), $k = 1, 2, \dots,$ compute backwards recursively for $n = N-1, \dots, 0$

$$\hat{\tau}_{n+1}^m = \sum_{k=n+1}^{N} k f^{\theta_k}(y_k^m) \prod_{j=n+1}^{k-1} (1 - f^{\theta_j}(y_j^m))$$

along each of the $M$ sample paths.

**Step 7**. At time $n = 0$, we finish by computing the current optimal stopping time

$$\hat{\tau}^m = \sum_{n=1}^{N} k f^{\theta_n}(y_n^m) \prod_{k=1}^{n-1}(1 - f^{\theta_k}(y_k^m)).$$

This is the estimator for the $\hat{\tau}$ of (14).

**Step 8**. The estimator of $\mathbb{E}g(\hat{\tau}, X_{\hat{\tau}})$ is given by:

$$\hat{V} = \frac{1}{M}\sum_{m=1}^{M} g(\hat{\tau}_m, y_{\hat{\tau}_m}^m).$$

*Remark* 4.10. Notice that $\hat{V}$ is the Monte Carlo estimator for $\mathbb{E}g(\hat{\tau}, X_{\hat{\tau}})$, and so, is unbiased. It is also consistent, thanks to the law of large numbers. We will see this in detail in the next section.

*Remark* 4.11. If the Markov process $X$ starts from a deterministic initial value $x_0 \in \mathbb{R}^d$, the initial stopping decision is given by a constant $f_0 \in \{0, 1\}$. To learn $f_0$ from the simulated paths of $X$, we can compare the initial reward $g(0, x_0)$ to the Monte Carlo estimate $\hat{V}$ of $\mathbb{E}g(\tau_1, X_{\tau_1})$ where $\tau_1 \in \mathcal{T}_1$ is of the form

$$\tau_1 = \sum_{n=1}^{N} n f^{\theta_n}(X_n) \prod_{j=1}^{n-1}(1 - f^{\theta_j}(X_j))$$

for $f^{\theta_N} \equiv 1$ and trained parameters $\theta_1, \dots, \theta_{N-1} \in \mathbb{R}^q$. Then, stop immediately ($f_0 = 1$) if $g(0, x_0) \geq \hat{V}$ and continue ($f_0 = 0$) otherwise. Then, the stopping time is of the form

$$\tau^{\Theta} = \begin{cases} 0 & \text{if } f_0 = 1 \\ \tau_1 & \text{if } f_0 = 0 \end{cases}$$

Notice that $\tau_1$ is equivalent to $\hat{\tau}^m$ (see (15)) specifying the drawing of the sample paths $y_n^m$ of $X$.

**Algorithm 1** Deep Optimal Stopping Time algorithm - Lower Bound

```
1: Simulate $M$ independent paths of $(X_n)_{n=0}^N$: $(x_n^m)_{n=0}^N$, training data.
2: Fix $f^{\theta_N} \equiv 1 \ \forall m$. In other words, initialize an $N+1 \times M$ matrix $MF$:
       $$\begin{cases} Mf[i][j] = 0 & \forall i = 1, \dots, N-1, \forall j = 1, \dots, M \\ Mf[N][j] = 1 & \forall j = 1, \dots, M \end{cases}$$
3: Initialize an $N+1 \times M$ matrix $M\tau$:
       $$\begin{cases} M\tau[i][j] = 0 & \forall i = 1, \dots, N-1, \forall j = 1, \dots, M \\ M\tau[N][j] = N & \forall j = 1, \dots, M \end{cases}$$
4: for $n = N-1, \dots, 0$ do
5:     Initialize a Neural Network $F^{\theta_n}$ with a suitable optimizer (a gradient descent method).
6:     We will set $r(\theta_n) := -r_n^{MC}$ as the loss function.
           $$r(\theta_n) = -\frac{1}{M}\left(\sum_{m=1}^{M} g(n, x_n^m) \cdot F^{\theta_n}(x_n^m) + g(\tau_{n+1}^m, x_n^m) \cdot (1 - F^{\theta_n}(x_n^m))\right)$$
7:     We switch from $F^{\theta_n}$ to $f^{\theta_n}$ having, for every $m$:
           $$f^{\theta_n}(x_n^m) = (F^{\theta_n}(x_n^m) > 0.5).$$
8:     Update $Mf[n][m] = (F^{\theta_n}(x_n^m) > 0.5)$ for all $m$.
9:     Update $M\tau[n][m]$ equal to the index of the maximum $Mf[i][m]$ where $i = n, \dots, N$ for all $m$.
10:    Do so chronologically, i.e. prioritise $M\tau[n][m]$ over $M\tau[n+1][m]$ if they both are $Mf[i][m] = 1$.
11: end for
12: Simulate $K_L$ independent paths of $(X_n)_{n=0}^N$: $(y_n^k)_{n=0}^N$, testing data.
13: Compute $\hat{L} := \hat{V}$ with $(y_n^k)_{n=0}^N$ instead of the ones used to construct the stopping decisions ( $(x_n^m)_{n=0}^N$).
```

As a result, we have Algorithm 1 below.

**Why is it a lower bound?**

In Algorithm 1, we have trained the stopping decisions $f^{\theta_n}$. With them, we get an approximation of the optimal stopping time, $\tau^\Theta$, which is of the form $\tau^\Theta = l(X_0, \dots, X_{N-1})$, where $l : \mathbb{R}^{dN} \to \{0, 1, \dots, N\}$ is a measurable function.

With $K_L$ independent paths $(y_n^k)_{n=0}^N$, $k = 1, 2, \dots, K_L$ we calculate

$$\hat{L} = \hat{V} = \frac{1}{K_L}\sum_{k=1}^{K_L} g(l(y_0^k, \dots, y_{N-1}^k), y^k_{l(y_0^k, \dots, y_{N-1}^k)}),$$

which is the Monte Carlo estimator for $L = V_0 = \sup_{\tau \in \mathcal{T}} \mathbb{E}g(\tau, X_\tau)$. As stated in Remark 4.10, this is unbiased and consistent.

Now by Corollary 4.7, we get the lower bound indeed:

$$\sup_{\tau \in \mathcal{T}} \mathbb{E}g(\tau, X_\tau) \leq \mathbb{E}g(\hat{\tau}, X_{\hat{\tau}}) + \epsilon.$$

Recall that the $\epsilon$ depends on the quantity of nodes of the neural networks $q_1, q_2$.

### 4.4.2 Upper bound

For the upper bound, we must refer to the Snell envelope of the reward process $(g(n, X_n))_{n=0}^N$ and the Finite Horizon theorem (see Appendix A).

The Snell envelope of the reward process is the smallest (almost surely according to $P_0$, the measure of probability) supermartingale with respect to $(\mathcal{F}_n)_{n=0}^N$ that dominates this reward process. It is given by:

$$H_n = \operatorname*{ess\,sup}_{\tau \in \mathcal{T}_n} \mathbb{E}[g(\tau, X_\tau) \,|\, \mathcal{F}_n], \quad n = 0, 1, \dots, N.$$

We will use the Doob-Meyer decomposition of submartingales:

**Claim 4.12.** *Every submartingale $X = (X_t, \mathcal{F}_t)_{t \geq 0}$ admits the decomposition:*

$$X_t = X_0 + M_t + A_t,$$

*where $M$ is a local martingale and $A = (A_t, \mathcal{F}_t)_{t \geq 0}$ is an increasing predictable locally integrable process.*

Its Doob-Meyer decomposition is:

$$H_n = H_0 + M_n^H - A_n^H,$$

where $M^H$ is the $(\mathcal{F}_n)$-martingale given by:

$$M_0^H = 0 \text{ and } M_n^H - M_{n-1}^H = H_n - \mathbb{E}[H_n \,|\, \mathcal{F}_{n-1}], \quad n = 1, \dots, N,$$

and $A^H$ is the nondecreasing $(\mathcal{F}_n)$-predictable process given by:

$$A_0^H = 0 \text{ and } A_n^H - A_{n-1}^H = H_{n-1} - \mathbb{E}[H_n \,|\, \mathcal{F}_{n-1}], \quad n = 1, \dots, N.$$

To get an estimate of an upper bound for $V_0$, we need to use another formulation of optimal stopping problems (see [5]), the dual representation of the problem:

**The Dual Problem**

The problem of valuing the optimal stopping time given a reward function $g$ is that of computing:

$$V_0 = \sup_{\tau \in \mathcal{T}} \mathbb{E} g(\tau, X_\tau).$$

We define the dual function $F(t, \pi)$ for an arbitrary adapted supermartingale $\pi_t$ as:

$$F(t, \pi) := \mathbb{E}\left[\max_{s \in [t,N] \cap \mathcal{T}} (g(s, X_s) - \pi_s)\right] + \pi_t.$$

The dual problem is to minimize the dual function at time 0 over all supermartingales $\pi_t$. Hence, the optimal value of the dual problem is $U_0$ where:

$$U_0 = \inf_\pi F(0, \pi) = \inf_\pi \mathbb{E}\left[\max_{s \in \mathcal{T}} (g(s, X_s - \pi_s)\right] + \pi_0.$$

In fact, we have the following theorem:

**Theorem 4.13** (Duality Relation)**.** *The optimal values of the primal problem* ($V_0$) *and the dual problem* ($U_0$) *are equal:*

$$V_0 = U_0.$$

*Moreover, an optimal solution of the dual problem is given by* $\pi_t^* = V_t$, *where* $V_t$ *is the value process:*

$$V_t = \sup_{\tau \in \{[t,N] \cap \mathcal{T}\}} \mathbb{E}_t[g(\tau, X_\tau)]$$

*Sketch of the proof:* For any supermartingale $\pi_t$,

$$V_0 = \sup_{\tau \in \mathcal{T}} \mathbb{E} g(\tau, X_\tau) = \sup_{\tau \in \mathcal{T}} \mathbb{E}[g(\tau, X_\tau) - \pi_t + \pi_t] \leq \sup_{\tau \in \mathcal{T}} \mathbb{E}[g(\tau, X_\tau) - \pi_t] - \pi_0 \leq \mathbb{E}[\max_{\tau \in \mathcal{T}}(g(\tau, X_\tau) - \pi_t)] + \pi_0.$$

For the first inequality, see the full proof in [5]. Taking the infimum over all supermartingales $\pi_t$ on the right hand side, $V_0 \leq U_0$.

On the other hand, $V_t$ is a supermartingale, and so:

$$U_0 \leq \mathbb{E}\left[\max_{\tau \in \mathcal{T}}(g(\tau, X_\tau) - V_t)\right] + V_0.$$

Since $V_t \geq g_t$ for all $t$, $U_0 \leq V_0$. Therefore, $V_0 = U_0$ and $\pi_t^* = V_t$. □

**Proposition 4.14.** *Let* $(\varepsilon_n)_{n=0}^N$ *be a sequence of integrable random variables on* $(\Omega, \mathcal{F}, \mathbb{P})$. *Then*

$$V_0 \geq \mathbb{E}\left[\max_{0 \leq n \leq N}(g(n, X_n) - M_n^H - \varepsilon_n)\right] + \mathbb{E}\left[\min_{0 \leq n \leq N}(A_n^H + \varepsilon_n)\right]. \tag{18}$$

*Moreover, if* $\mathbb{E}[\varepsilon_n \mid \mathcal{F}_n] = 0$ *for all* $n \in \{0, 1, \dots, N\}$, *one has*

$$V_0 \leq \mathbb{E}\left[\max_{0 \leq n \leq N}(g(n, X_n) - M_n - \varepsilon_n)\right] \tag{19}$$

*for every* $(\mathcal{F}_n)-$*martingale* $(M_n)_{n=0}^N$ *starting from* 0.

*Proof.* First,

$$\mathbb{E}\left[\max_{0 \leq n \leq N}(g(n, X_n) - M_n^H - \varepsilon_n\right] \leq \mathbb{E}\left[\max_{0 \leq n \leq N}(H_n - M_n^H - \varepsilon_n)\right] =$$
$$= \mathbb{E}\left[\max_{0 \leq n \leq N}(H_0 - A_n^H - \varepsilon_n)\right] = V_0 - \mathbb{E}\left[\min_{0 \leq n \leq N}(A_n^H + \varepsilon_n)\right],$$

which proves (18).

For the other inequality, assume $\mathbb{E}[\varepsilon_n \mid \mathcal{F}_n] = 0$ for all $n \in \{0, 1, \dots, N\}$. Let $\tau$ be an $X-$stopping time. Then:

$$\mathbb{E}\varepsilon_\tau = \mathbb{E}\left[\sum_{n=0}^N \mathbb{1}_{\{\tau = n\}} \varepsilon\right] = \mathbb{E}\left[\sum_{n=0}^N \mathbb{1}_{\{\tau = n\}} \mathbb{E}[\varepsilon \mid \mathcal{F}_n]\right] = 0.$$

Now, we can make use of the optional stopping theorem, which states that, under certain conditions, the expected value of a martingale at a stopping time is equal to its initial expected value:

**Theorem 4.15** (Optional Stopping theorem (discrete version))**.** *Let* $M = (M_n)_{0 \leq n \leq N}$ *be a discrete-time martingale and* $\tau$ *a stopping time with values in* $\{0, 1, \dots, N\} \cup \infty$, *both with respect to a filtration* $(\mathcal{F}_n)_{0 \leq n \leq N}$. *Assume that one of the conditions holds:*

(a) *There exists a constant $c$ such that $\tau \leq c$ almost surely.*

(b) *$\mathbb{E}\tau < \infty$ and there exists a constant $c$ such that $\mathbb{E}[|X_{n+1} - X_n| \,|\, \mathcal{F}_n] \leq c$ almost surely on the event $\{\tau > n\}$ for all $n \in \{0, 1, \ldots, N\}$.*

(c) *There exists a constant $c$ such that $|X_{n\wedge\tau}| \leq c$ almost surely for all $n \in \{0, 1, \ldots, N\}$. Recall that $\wedge$ denotes minimum.*

*Then, $X_t$ is an a.s. well defined random variable and $\mathbb{E}X_\tau = \mathbb{E}X_0$. Similarly, if $X_t$ is a submartingale or supermartingale and one of the above conditions holds, then $\mathbb{E}X_\tau \geq \mathbb{E}X_0$ for a submartingale and $\mathbb{E}X_\tau \leq \mathbb{E}X_0$ for a supermartingale.*

From the optional stopping theorem for supermartingales (notice that it complies with condition (a)):

$$\mathbb{E}g(\tau, X_\tau) \overset{(1)}{=} \mathbb{E}[g(\tau, X_\tau) - M_t - \varepsilon_n] \leq \mathbb{E}\left[\max_{0\leq n\leq N}(g(n, X_n) - M_n - \varepsilon_n)\right]$$

for every $(\mathcal{F}_n)-$martingale $(M_n)_{n=0}^N$ starting from 0 (that is, $M_0 = 0$). Taking the supremum on the left hand side, we have (19). □

*Notice that the right hand side of (19) provides an upper bound for $V_0$ provided that $\mathbb{E}[\varepsilon_n \,|\, \mathcal{F}_n] = 0$ for all $n$, where $\varepsilon_n$ is the sequence of integrable error terms at time $n$. By (18), the bound is tight if $M = M^H$ and $\varepsilon \equiv 0$.*

*Hence, the objective is to use our candidate optimal stopping time $\tau^\Theta$ to construct a martingale close to $M^H$.*

*Let us construct the following process $H_n^\Theta$:*

$$H_n^\Theta = \mathbb{E}\left[g(\tau_n^\Theta, X_{\tau_n^\Theta}) \,|\, \mathcal{F}_n\right], \quad n = 0, 1, \ldots, N,$$

*corresponding to*

$$\tau_n^\Theta = \sum_{m=n}^{N} m f^{\theta_m}(X_m) \prod_{j=n}^{m-1}(1 - f^{\theta_j}(X_j)), \quad n = 0, 1, \ldots, N.$$

*The idea is that the better $\tau^\Theta$ approximates an optimal stopping time, the better $H_n^\Theta$ approximates the Snell envelope $(H_n)_{n=0}^N$.*

*The martingale part of it is given by $M_0^\Theta = 0$ and*

$$M_n^\Theta - M_{n-1}^\Theta = H_n^\Theta - \mathbb{E}[H_n^\Theta \,|\, \mathcal{F}_{n-1}] = f^{\theta_n}(X_n)g(n, X_n) + (1 - f^{\theta_n}(X_n)) \cdot C_n^\Theta - C_{n-1}^\Theta,$$

*for all $n \geq 1$, where $C_n$ are the continuation values:*

$$C_n^\Theta = \mathbb{E}[g(\tau_{n+1}^\Theta, X_{\tau_{n+1}^\Theta}) \,|\, \mathcal{F}_n] \overset{(1)}{=} \mathbb{E}[g(\tau_{n+1}^\Theta, X_{\tau_{n+1}^\Theta}) \,|\, X_n], \quad n = 0, 1, \ldots, N-1.$$

*Remark* 4.16. (1) Since $X_n$ is Markov and $\tau_{n+1}^\Theta$ only depends on $(X_{n+1}, \ldots, X_{N-1})$.

- Note that $C_N^\Theta$ is not relevant since $(1 - f^{\theta_N}(X_N)) = 0$ and is actually not specified.
- Also note that $H_n^\Theta$, $M_n^\Theta$ and $C_n^\Theta$ are defined in terms of conditional expectations, and hence they are specified up to equality $\mathbb{P}$-almost sure.

**Estimation of $M^\Theta$**

*To estimate $M^\Theta$, we will generate a third set of independent realizations $(z_n^k)_{n=0}^N$, $k = 1, 2, \dots, K_U$ of $(X_n)_{n=0}^N$.*

*The novelty with respect to the other sets is that, in addition, for every $z_n^k$, we simulate $J$ continuation paths $\tilde{z}_{n+1}^{k,j}, \dots, \tilde{z}_N^{k,j}$, $j = 1, \dots, J$. These paths are conditionally independent of each other and of $z_{n+1}^k, \dots, z_N^k$, in the following sense: the tuples $(\tilde{z}_{n+1}^{k,j}, \dots, \tilde{z}_N^{k,j}), j = 1, \dots, J$ are simulated according to $p_n(z_n^k, \cdot)$ such that $p_n(X_n, B) = \mathbb{P}[(X_{n+1}, \dots, X_N) \in B \mid X_n]$ $\mathbb{P}$ almost surely for all Borel sets $B \subseteq \mathbb{R}^{(N-n)d}$. We will be generating these tuples independently across $j$ and $k$.*

*However, it is notable that the continuation paths starting from $z_n^k$ not necessarily have to be drawn independently of those from $z_m^k$ for $m \neq n$.*

*We will denote by $\tau_{n+1}^{k,j}$ the value of $\tau_{n+1}^\Theta$ along $\tilde{z}_{n+1}^{k,j}, \dots, \tilde{z}_N^{k,j}$.*

*The natural estimation of the continuation values is:*

$$C_n^k = \frac{1}{J}\sum_{j=1}^{J} g\left(\tau_{n+1}^{k,j}, \tilde{z}^{k,j}_{\tau_{n+1}^{k,j}}\right), \quad n = 0, 1, \dots, N-1,$$

*And, with them, one has an estimation of the increments $M_n^\Theta - M_{n-1}^\Theta$ along the k-th simulated path $z_0^k, \dots, z_N^k$, the so-called noisy estimates $\Delta M_n^k$:*

$$\Delta M_n^k = f^{\theta_n}(z_n^k) g(n, z_n^k) + (1 - f^{\theta_n}(z_n^k)) C_n^k - C_{n-1}^k.$$

*Consequently,*

$$M_n^k = \begin{cases} 0 & \text{if } n = 0 \\ \sum_{m=1}^{n} \Delta M_m^k & \text{if } n \geq 1 \end{cases}$$

*where $M_n^k$ can be thought of realizations of $M_n^\Theta + \varepsilon_n$ for estimation errors $\varepsilon_n$ with variances proportional to $1/J$, since the lesser paths we simulate, the more we deviate from the better estimation. Following this intuition, we interpret $\varepsilon_n$ such that $\mathbb{E}[\varepsilon_n \mid \mathcal{F}_n] = 0$ for all $n$.*

*From that point of view, we can derive*

$$\hat{U} = \frac{1}{K_U}\sum_{k=1}^{K_U} \max_{0 \leq n \leq N} \left(g(n, z_n^k) - M_n^k\right),$$

*as an unbiased estimate of the upper bound*

$$U = \mathbb{E}\left[\max_{0 \leq n \leq N} (g(n, X_n) - M_n^\Theta - \varepsilon_n)\right].$$

*By the law of large numbers, converges to $U$ for $K_U \to \infty$.*

### 4.4.3 Optimal value estimate and confidence intervals

*The point estimate of $V_0$ is the average of the estimation of the upper bound and the estimation of the lower bound, and hence:*

$$V_0 = \frac{\hat{L} + \hat{U}}{2}.$$

**Condition on the derivation of confidence intervals**

*As mentioned in the problem definition section, to derive confidence intervals, we assume $g(n, X_n)$ is square-integrable for all $n$. With this assumption, $g(\tau^\Theta, X_{\tau^\Theta})$ and $\max_{0\le n\le N}(g(n, X_n) - M_n^\Theta - \varepsilon_n)$ are as well:*

*Let us consider $g(\tau^\Theta, X_{\tau^\Theta})$. By the definition of the approximation to the optimal stopping time, we have:*

$$g(\tau^\Theta, X_{\tau^\Theta}) = \sum_{n=0}^{N} g(n, X_n) \cdot \mathbb{1}_{\tau^\Theta=n}$$

*where $\mathbb{1}_{\tau^\Theta=n}$ is the indicator function that takes value 1 if $\tau^\Theta = n$, and 0 otherwise.*

*Using this expression, we can write:*

$$|g(\tau^\Theta, X_{\tau^\Theta})|^2 \le \left(\sum_{n=0}^{N} |g(n, X_n)| \cdot \mathbb{1}_{\tau^\Theta=n}\right)^2 \overset{(1)}{\le} \sum_{n=0}^{N} |g(n, X_n)|^2 \cdot \mathbb{1}_{\tau^\Theta=n}$$

*Where in (1) we have used the Cauchy-Schwarz inequality. Taking expectations on both sides, we obtain:*

$$\mathbb{E}[|g(\tau^\Theta, X_{\tau^\Theta})|^2] \le \mathbb{E}\left[\sum_{n=0}^{N} |g(n, X_n)|^2 \cdot \mathbb{1}_{\tau^\Theta=n}\right]$$

*Using the dominated convergence theorem, we can switch the sum and the expectation, and get:*

$$\mathbb{E}[|g(\tau^\Theta, X_{\tau^\Theta})|^2] \le \sum_{n=0}^{N} \mathbb{E}[|g(n, X_n)|^2 \cdot \mathbb{1}_{\tau^\Theta=n}]$$

*Since $g(n, X_n)$ is square-integrable, we have $\mathbb{E}[|g(n, X_n)|^2] < \infty$ and thus:*

$$\mathbb{E}[|g(n, X_n)|^2 \cdot \mathbb{1}_{\tau^\Theta=n}] < \infty.$$

*Hence the finite sum is also square-integrable and therefore, $g(\tau, X_{\tau^\Theta})$ is square-integrable as well.*

*On the other hand, to prove that $\max_{0\le n\le N}(g(n, X_n) - M_n^\Theta - \varepsilon_n)$ we can reduce it to proving that $g(r, X_r) - M_r^\Theta - \varepsilon_r$ is square-integrable, supposing that the argument of the maximum is $r$.*

*Recall that $M_n^\Theta$ was defined as the linear combination of $f^{\theta_n}$ (which take values of at most 1) and $C_n^\Theta = \mathbb{E}[g(\tau_{n+1}^\Theta, X_{\tau_{n+1}^\Theta}) \mid \mathcal{F}_n]$ which are square-integrable and $g(n, X_n)$, which are square-integrable, hence, $M_n^\Theta$ are also square-integrable. Also, $\varepsilon_n$ are defined as square-integrable error terms.*

*In conclusion, we get that for a given $r$ the term $g(r, X_r) - M_r^\Theta - \varepsilon_r$ is square-integrable, and, as a consequence, the maximum (attained at time $r$).*

**Derivation of confidence intervals**

*From the central limit theorem for large $K_L$, $\hat{L}$ is approximately normally distributed with mean $L$ and variance $\hat{\sigma}_L^2/K_L$ for*

$$\hat{\sigma}_L^2 = \frac{1}{K_L - 1}\sum_{k=1}^{K_L}\left(g(l^k, y_{l^k}^k) - \hat{L}\right)^2.$$

*For every $\alpha \in (0, 1]$,*

$$\left[\hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}}, \infty\right)$$

*is a $1-\alpha/2$ confidence interval for $L$, with $z_{\alpha/2}$ the $1-\alpha/2$ quantile of the standard normal distribution.*

*Doing the same reasoning for $\hat{U}$:*

$$\hat{\sigma}_U^2 = \frac{1}{K_U - 1}\sum_{k=1}^{K_U}\left(\max_{0\le n\le N}\left(g\left(n, z_n^k\right) - M_n^k\right) - \hat{U}\right)^2,$$

*and hence*

$$\left(-\infty, \hat{U} + z_{\alpha/2}\frac{\hat{\sigma_U}}{\sqrt{K_U}}\right],$$

*is a $1-\alpha/2$ confidence interval for $U$.*

*Adding them both, one has that, for every constant $\varepsilon > 0$,*

$$\mathbb{P}\left[V_0 < \hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}} \text{ or } V_0 > \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}\right] \le$$

$$\le \mathbb{P}\left[L < \hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}}\right] + \mathbb{P}\left[U > \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}\right] \le \alpha + \varepsilon,$$

*provided $K_L$ and $K_U$ are sufficiently large.*

*Then,*

$$\mathbb{P}\left[\hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}} < V_0 < \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}\right] = 1 - \mathbb{P}\left[V_0 < \hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}} \text{ or } V_0 > \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}\right] \ge$$

$$\ge 1 - \alpha - \varepsilon.$$

*Asymptotically, we can reduce $\varepsilon \to 0$, and hence, it follows that:*

$$\left[\hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}}, \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}\right]$$

*is a $1-\alpha$ confidence interval for $V_0$.*

*Summing up, we can derive the next algorithm:*

**Algorithm 2** Deep Optimal Stopping Time algorithm - Upper Bound

```
1: Simulate $K_U$ paths $z_n^k$ of $(X_n)_{n=0}^N$, getting a matrix $z[n][k]$.
2: For each $z_n^k$, simulate other $J$ paths $\tilde{z}_n^{k,j}$ of $(X_n)_{n=0}^N$, the continuation paths, getting a matrix $\tilde{z}[n,k,j]$.
3: With that, estimate the continuation values $C_n^k$ with:
       $$C_n^k = \frac{1}{J}\sum_{j=1}^{J} g\left(\tau_{n+1}^{k,j}, \tilde{z}_{\tau_{n+1}^{k,j}}^{k,j}\right)$$
   getting a matrix $C[n,k]$. Note that $\tau_{n+1}^{k,j}$ is the matrix resulting of estimating $\tau^{\Theta}$ along the paths $tilde\_z_n^k$.
4: With the continuation values, we calculate the noisy estimates $\Delta M[n,k] = z\_Mf[n,k]g(n,z[n,k]) + (1-z\_Mf[n,k])C[n,k] - C[n-1,k]$, getting a matrix $\Delta M[n,k]$. Note that $z\_Mf$ is the matrix resulting of estimating the stopping decisions of $\tau^{\Theta}$ along the paths $z_n^k$ (hence, we have $Mf$, $y\_Mf$, $z\_Mf$ and $tilde\_z\_Mf$ and idem with $MT$.
5: Also, we get the matrix $M$ with $M[0,k] = 0$ and $M[n,k] = M[n-1,k] + \Delta M[n,k]$.
6: Finally, compute $\hat{U} = V_U$.
```

*And, eventually, get the $V$ point estimate and its associated confidence interval for a given $\alpha$:*

**Algorithm 3** Deep Optimal Stopping Time algorithm - Point estimate and confidence interval

```
1: Calculate $V = (V_U + V_L)/2 = (\hat{U} + \hat{L})/2$.
2: Compute $\hat{\sigma}_L$ and $\hat{\sigma}_U$.
3: Calculate the lower point of the confidence interval $CI_L(V,\alpha) = \hat{L} - z_{\alpha/2}\frac{\hat{\sigma}_L}{\sqrt{K_L}}$.
4: Calculate the upper point of the confidence interval $CI_U(V,\alpha) = \hat{U} + z_{\alpha/2}\frac{\hat{\sigma}_U}{\sqrt{K_U}}$.
5: $CI(V,\alpha) = [CI_L(V,\alpha), CI_U(V,\alpha)]$.
```

*As we will see later, there are some remarks and adjustments to be made to the algorithm so that it runs fast enough. Hence, these pseudo-algorithms are to be interpreted as only the idea of the steps to be followed.*

# 5. Applications

## 5.1 Road map

*As an application of the method we have presented, we will apply it to the valuation of financial derivatives, especially to American (or Bermudan) options with different underlying asset models.*

*We will first use our algorithm treating synthetic data, data treated by different literature references to test our method versus the results from Becker et al to test its correctness. Afterwards, we will show how to treat real-world data, doing model calibration to apply our method.*

*Although we provide summary results only for Black-Scholes Bermudan and Black-Scholes MBRCs, with the code given one can easily compute Kou/Merton/Heston Bermudan and Barrier options because of the architecture of the code. Please note that we attach the full code (not just snippets or chunks of it) at the Appendix C. Also, it is enough to edit ua_model = "____" to Bach, BS, Kou, Merton or Heston to change the underlying asset model.*

**Table of contents of the Applications section:**

1. *Underlying asset modelling.*
2. *Options as financial instruments.*

*Then, we will apply our algorithm to a number of optimal stopping problems involving the pricing of different classes of options.*

i. *Method*

ii. *Bermudan options.*

iii. *Barrier options (MBRCs).*

iv. *Model calibration and real-world options.*

*In the Underlying asset modelling subsection, we present different popular models used in option pricing, we exemplify a way of efficiently simulate them in Appendix B. In the next subsection we introduce options and briefly explore the different classes of options there are.*

*In the second part of the Applications section, we first describe our method's optimization techniques. Then, we use our method to price American/Bermudan options, we tackle a complex class of Barrier option and shortly rough out how the methodology could be exploited to price other kind of options. Finally, we explain how could one explore different assets, calibrate parameters to choose and adjust a model for our simulated path to follow and then apply the methodology of this thesis to price an option.*

## 5.2 Underlying asset modelling

*To model asset prices, different stochastic processes have been used. Neither of the models that one can elaborate is right or wrong by itself, but it adapts better or worse to each asset depending on certain market conditions and characteristics of the asset, such as volatility, interest rates and dividends.*

*The following models were taught to me in the framework of the Computational Finance course [9], and we also consulted the following references for, respectively, some more insights on the Kou and the Heston model [1, 2].*

*In this section, we will just explore some of the main perks of these models, without going into much detail since there is already a lot of interesting literature about them. All the references will be provided. Notice that we are making in this section the following abuse of notation: we are using indistinctly the concept of model and the concept of dynamics (in a stochastic sense), and hence, when we refer to the Bachelier model, for instance, we are referring to the arithmetic Brownian motion applied to our model.*

### 5.2.1 Bachelier model

*The Bachelier model is often used as a starting point for more complex option pricing models such as the Black-Scholes model. Let $T > 0$ be the fixed maturity, and let $(W_t)_{t\in[0,T]}$ be the standard Brownian motion. Then, the dynamics of the underlying asset is:*

$$dS_t = \sigma_t dW_t$$

*where $\sigma_t = \sigma$ is constant.*

*In this model, the asset price changes over time are normally distributed with constant volatility $\sigma$. The Bachelier model is considered to be of little use in practice. However, models are not correct or incorrect, just accurate or not. As a curiosity, an example of this happened on April 2020. This month, for the first time in history, a major oil future contract closed below zero, raising fear amongst market participants on the real possibility of negative prices. Since having positive or zero prices is an assumption when pricing financial derivatives, the Chicago Mercantile Exchange (CME) announced that the Bachelier model would be used instead of the standard Black-Scholes model accounting for pricing negative strike options*[2].

*The following are 20 simulated paths of an asset following a Bachelier model, with the following parameters: $T = 1$ maturity, $N = 10$ number of discretized dates (we discretize the continuous time in time steps $dt = T/N$, $NSim = 20$ number of simulations, $\mu = 0.05$, $\sigma = 0.20$ and finally the current spot price being $S0 = 100$.*

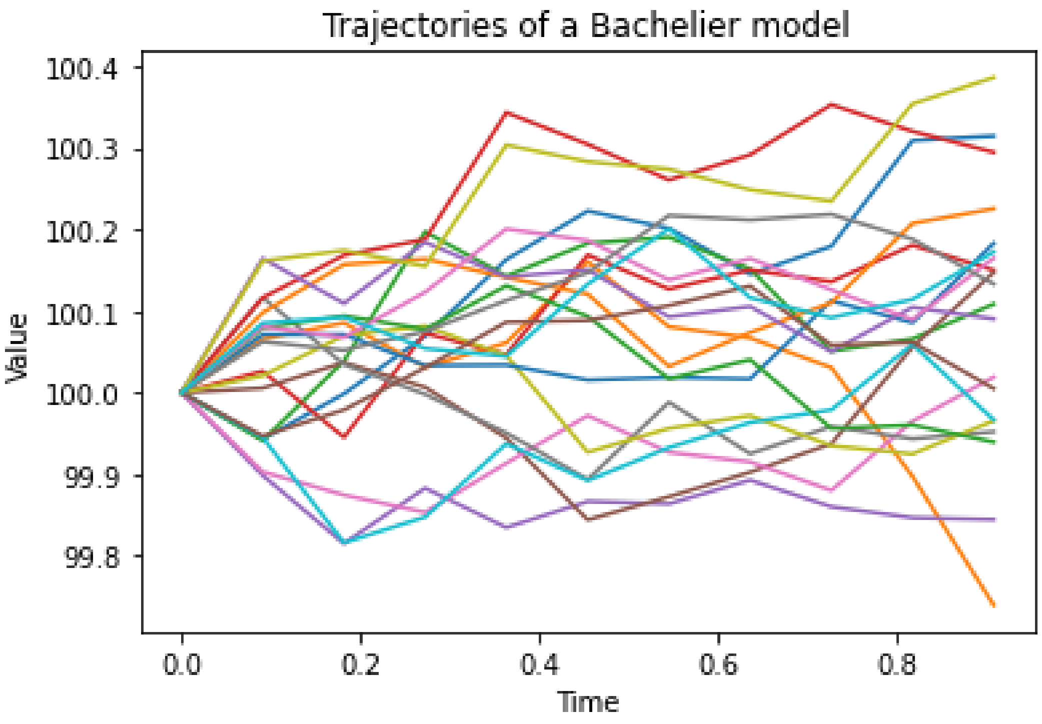


Figure 3: Simulation of 20 paths of an asset modelled with a Bachelier.

[2]For the efficient simulation of this and the following models, please refer to the Appendices

### 5.2.2 Black-Scholes model

*The most popular model is the Black-Scholes model. This model also assumes constant volatility, but instead of having as underlying an arithmetic Brownian motion, the model assumes a geometric Brownian motion.*

*Let $T > 0$ be the fixed maturity, and let $(W_t)_{t\in[0,T]}$ be the standard Brownian motion. Then, the dynamics of the underlying asset are:*

$$dS_t = \mu_t S_t dt + \sigma_t S_t dW_t$$

*where $\mu_t = \mu$ and $\sigma_t = \sigma$.*

*In this case, the asset prices are assumed to be distributed log-normally. This model is considered to be the standard of the option pricing models, and hence, in this thesis we will mainly focus on applying our methodology to the Black-Scholes model.*

*Since this model does not contemplate high and unexpected changes in the stock prices, it is interesting to add jumps to the stock prices, and we will present two jump diffusion models with different jump distributions.*

*Jumps in the dynamics of a stochastic process are typically introduced adding a compound Poisson process to the dynamics of the process.*

*The following are 20 simulated paths of an asset following a Black-Scholes model, with the following parameters: $T = 1$ maturity, $N = 10$ number of discretized dates (we discretize the continuous time in time steps $dt = T/N$, $NSim = 20$ number of simulations, $\mu = 0.05$, $\sigma = 0.20$ and finally the current spot price being $S0 = 100$.*

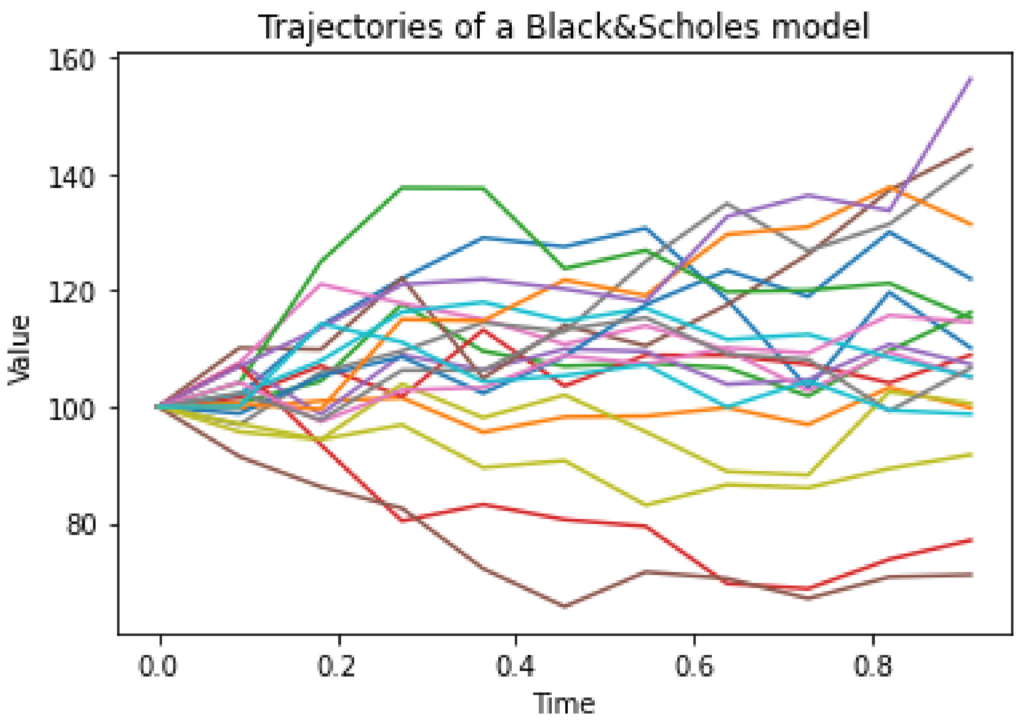


Figure 4: Simulation of 20 paths of an asset modelled with a Black-Scholes.

### 5.2.3 Lévy processes: Merton and Kou models

*To build a compound Poisson process, we will need the exponential random variable and the Poisson distribution. We will further introduce Lévy processes, that will be used afterwards throughout the Applications section.*

**Definition 5.1** (Exponential random variable). A continuous random variable $Y \geq 0$ is $Y \sim Exp(\lambda)$ if the probability distribution function (pdf) is $f(y) = \lambda e^{-\lambda y} \mathbb{1}_{\{y \geq 0\}}$.

*We also have the following characterization:*

**Proposition 5.2** (Absence of memory). *Let $T \geq 0$ be a random variable such that $\mathbb{P}(T > t + s \,|\, T > t) = \mathbb{P}(T > s) \iff T \sim Exp$.*

*The exponential distribution models the time between events. On the other hand, the Poisson distribution models the number of events that occur within a fixed interval of time.*

**Definition 5.3** (Poisson random variable). Given a discrete random variable $N$ in $\mathbb{N}$, $N$ is $N \sim Poiss(\lambda)$ if $\mathbb{P}(N = n) = e^{-\lambda}\lambda^n/n!$ for all $n \in \mathbb{N}$.

*We can generalize to the stochastic processes.*

**Definition 5.4** (Poisson process or Lévy Counting process). Given $(\tau_i)_{i\geq 1}$ sequence of independent identically distributed random variables distributed as a $Exp(\lambda)$, we construct $T_n = \sum_{i=1}^{n} \tau_i$ and we define $N_t$ as a Poisson process with intensity $\lambda$ in the following way:

$$N_t = \sum_{n=1}^{\infty} \mathbb{1}_{t \geq T_n}.$$

*Some interesting properties of the Poisson process are the following:*

**Proposition 5.5.**

- *Fixed $t$, $N_t$ is a random variable that follows a $Poiss(\lambda t)$ distribution.*
- *$N_t$ is a càdlàg process. A càdlàg process is a stochastic process in which the paths are right-continuous and has left limits: as time approaches a specific point, the process converges from the left side.*
- *$N_t$ has independent increments.*

*With the Poisson process, we can build other processes, for example:*

**Definition 5.6** (Compensated Poisson process). We define the compensated Poisson process as $\tilde{N}_t = N_t - \lambda t$. Notice that $\tilde{N}_0 = 0$, $\mathbb{E}[\tilde{N}_t] = 0$ and that $\tilde{N}_t$ is a martingale.

$$\mathbb{E}[\tilde{N}_t|\mathcal{F}_s] = \mathbb{E}[N_t - \lambda t|\mathcal{F}_s] = \mathbb{E}[N_t - N_s + N_s - \lambda t|\mathcal{F}_s] = \mathbb{E}[N_t - N_s|\mathcal{F}_s] - \lambda(t-s) = \lambda(t-s) - \lambda(t-s) = 0$$

**Definition 5.7** (Compound Poisson process). A compound Poisson process is a continuous-time stochastic process such that the jumps arrive randomly according to a Poisson process and the size of the jumps is also random, according to a jump distribution $J$. The process is given by:

$$Y_t = \sum_{i=1}^{N_t} D_i$$

where $N_t$ is a counting Poisson process and $\{D_i\}_i$ are independent identically distributed random variables with distribution function $J$, independent of $\{N_t\}_t$.

More specifically, let $N_t$ be the number of events that occur in the interval $[0, t)$:

$$N_t = N(t) = \max\{k \mid T_k \leq t\}$$

where $T_k$ is a Poisson of rate $\lambda$. Notice that a Poisson process is a compound Poisson process with $J$ such that $\mathbb{P}(D_i = 1) = 1$.

*Now, on Lévy processes. These are stochastic processes that have really good properties for modelling phenomena.*

**Definition 5.8** (Lévy process). Let $(\Omega = \mathbb{R}^n, F, \mathbb{P})$ be a probability space. A càdlàg process $X_t$ such that $X_0 = 0$ is a Lévy if:

(i) Independent increments: Given $0 \leq t_1 < t_2 < t_3 < ... < t_n$ and $\perp$ denoting independence:

$$X_{t_0} \perp X_{t_1} - X_{t_0} \perp X_{t_2 - t_1} \perp ... \perp X_{t_n} - X_{t_{n-1}}$$

(ii) Stationary increments: $X_{t+h} - X_t \sim X_h$.

(iii) Stochastic continuity: $\forall \varepsilon > 0 \; \mathbb{P}(|X_{t+h} - X_t| \geq \varepsilon) \overset{h \to 0}{\to} 0$

*These are good properties since these three imply a fourth one: the infinite divisibility property: that is, the ability to be expressed as the sum of independent and identically distributed (i.i.d) random variables:*

$$X_t = \sum_n Y_n$$

*This property plays an important role in asset pricing, since it allows the decomposition of option payoffs into sums of simpler components.*

*Lévy processes, in general, let stable distributions arise: regardless of the number of increments, the distribution of the sum of independent increments remains the same, and that facilitates more flexible modelling.*

*Referring back to the processes we introduced, $N_t$ is the only counting Lévy process and $\sum_{i=1}^{N_t} Y_i$ is the only piece-wise constant Lévy.*

*We will model jumps introducing Lévy processes:*

$$log(S_t/S_0) = X_t = \mu t + \sigma W_t + \sum_{i=1}^{N_t} D_i$$

*These are precisely the basis for the Kou and Merton models, in which we will vary the jump distribution $J$ for the $D_i$s:*

**Kou model**

*Let $T > 0$ be the fixed maturity, and let $(W_t)_{t\in[0,T]}$ be the standard Brownian motion. Let $Y_t$ be a compound Poisson process such that $Y_t = \sum_{i=1}^{N_t} D_i$ where $N_t$ is the Lévy counting process distributed as a Poisson of parameter $\lambda t$ and $D_i$ have the next jump distribution pdf:*

$$f_{D_i}(z) = p\lambda_+ e^{\lambda_+ z}\mathbb{1}_{z\geq 0} + (1-p)\lambda_- e^{-|\lambda_-|z}\mathbb{1}_{z<0}$$

*where $\lambda_-, \lambda_+ > 0$ and $0 < p < 1$ is the probability of having a positive jump (upwards). It is needed that $\lambda_+ > 1$ to ensure good conditions such as $\mathbb{E}[S_t] < \infty$, which are reasonable.*

*Notice that this is a double exponential density function in the following sense: probability of positive jump times the pdf of an exponential of parameter $\lambda_+$ plus probability times the pdf of a negative exponential of parameter $\lambda_-$. Then, the dynamics of the underlying asset is:*

$$\frac{dS_t}{S_t} = dX_t = \mu dt + \sigma dW_t + d(\sum_{i=1}^{N_t}(V_i - 1)$$

*where $\sigma_t = \sigma$ is constant, and $V_i$ is a sequence of positive iid random variables such that $D_i = log(V_i)$.*

*Solving the SDE gives the dynamics of the asset price:*

$$S_t = S_0 exp\left\{(\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right\}\prod_{i=1}^{N_t} V_i$$

*The choice of using the Kou model over other jump diffusion models relies, among other reasons, on the flexibility we have when modelling both upward and downward jumps, and hence, it is of good value when we have complex and asymmetrical jump behaviour. Also, the Kou model has been empirically good when applied to the pricing of, especially, the assets that exhibit leptokurtic (that is, fat tails) distributions and skewness, and thus, it is worth to keep in mind the Kou model when we want to capture extreme events.*

*The following are 20 simulated paths of an asset following a Kou model, with the following parameters: $T = 1$ maturity, $N = 10$ number of discretized dates (we discretize the continuous time in time steps $dt = T/N$, $NSim = 20$ number of simulations, $\mu = 0.05$, $\sigma = 0.20$, $\lambda_t = \lambda = 3$ ($N_t \sim Poiss(\lambda t)$), $p = 0.6$ probability of having a positive jump, $\lambda_+ = \lambda_- = 1/25$ and finally the current spot price being $S0 = 100$.*

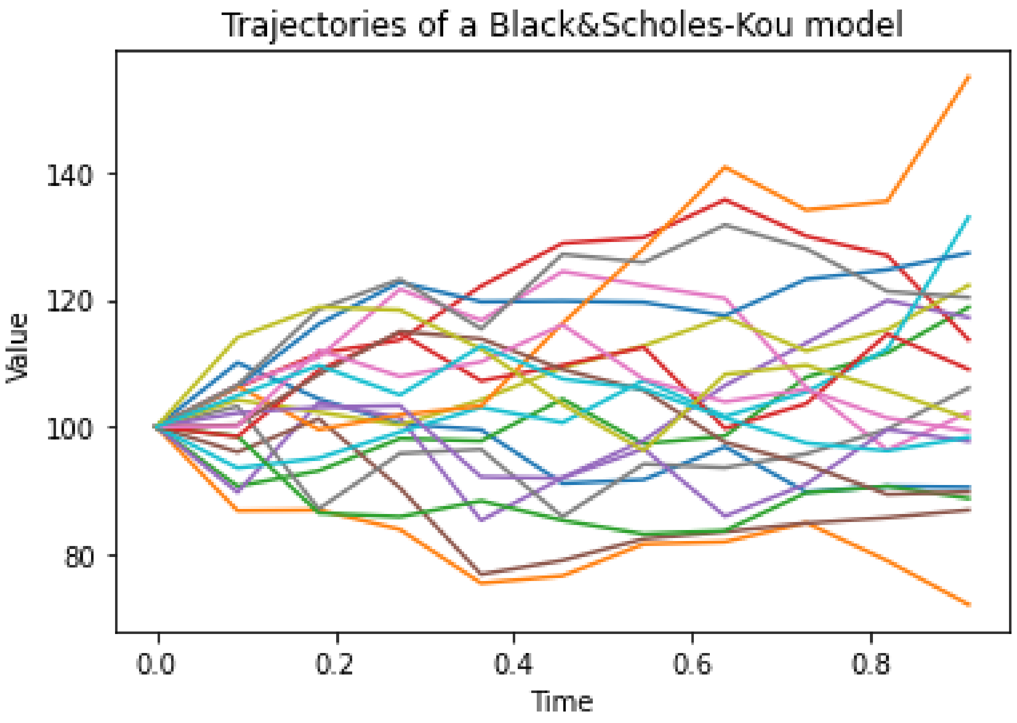


Figure 5: Simulation of 20 paths of an asset modelled with a Kou.

**Merton model**

*Let $T > 0$ be the fixed maturity, and let $(W_t)_{t\in[0,T]}$ be the standard Brownian motion. Let $Y_t$ be a compound Poisson process such that $Y_t = \sum_{i=1}^{N_t} D_i$ where $N_t$ is the Lévy counting process distributed as a Poisson of parameter $\lambda t$ and $D_i$ be distributed as a normal random variable $N(\mu_J, \sigma_J^2)$. Then, the dynamics of the underlying asset is:*

$$\frac{dS_t}{S_t} = dX_t = \mu dt + \sigma dW_t + d(\sum_{i=1}^{N_t}(V_i - 1)$$

*where $\sigma_t = \sigma$ is constant, and $V_i$ is a sequence of positive iid random variables such that $D_i = log(V_i)$.*

*Solving the SDE gives the dynamics of the asset price:*

$$S_t = S_0 exp\left\{(\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right\} \prod_{i=1}^{N_t} V_i$$

*One of the reasons to choose the Merton model over other jump diffusion models is, for instance, the lesser computational effort it is needed compared to others. It also has been widely used especially for equity options (options based on an equity index or a stock).*

*The following are 20 simulated paths of an asset following a Merton model, with the following parameters: $T = 1$ maturity, $N = 10$ number of discretized dates (we discretize the continuous time in time steps $dt = T/N$, $NSim = 20$ number of simulations, $\mu = 0.05$, $\sigma = 0.20$, $\lambda_t = \lambda = 3$ ($N_t \sim Poiss(\lambda t)$), $\mu_J = 0$ and $\sigma_J = 1$ and finally the current spot price being $S0 = 100$.*

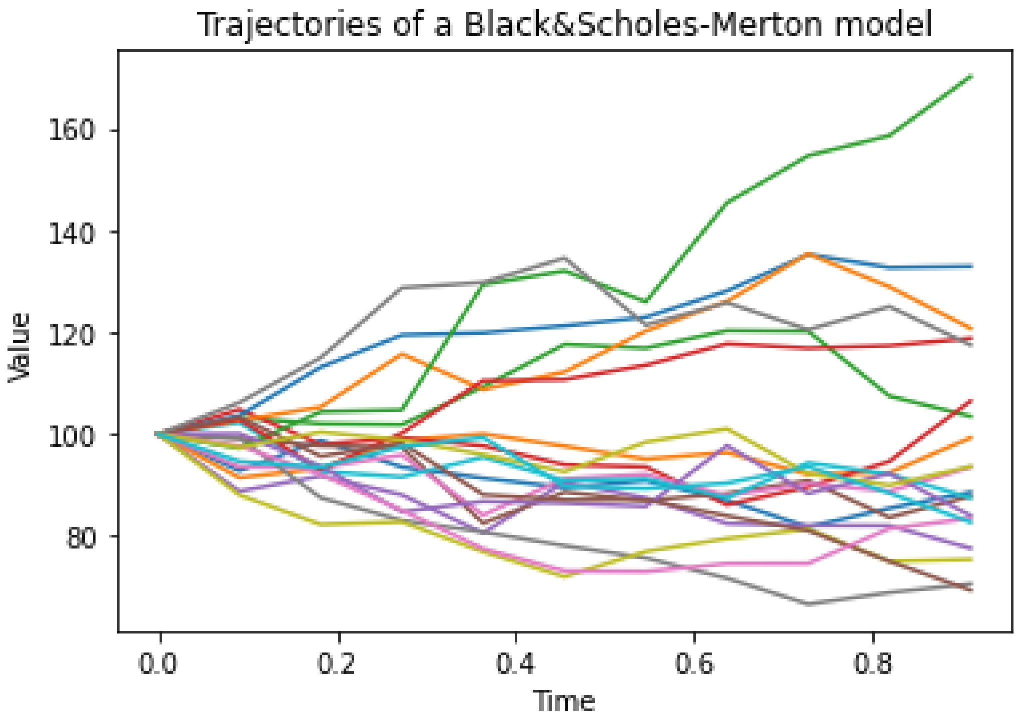


Figure 6: Simulation of 20 paths of an asset modelled with a Merton.

### 5.2.4 Stochastic volatility: the Heston model

*Now, we will present an example of a class of models that use the following dynamics for the price of the underlying asset:*

$$dS_t = \mu S_t dt + \sigma_t S_t dW_t,$$

*where $\sigma(t) = f(y(t))$ a function of $y(t)$, where $y(t)$ is described by an ODE:*

$$dy_t = \kappa(\theta - y_t)dt + \nu d\hat{W}_t,$$

*where $\theta$ is called mean reversing term*[3], *$\kappa$ is the speed of the mean reversion and $\nu$ depends on the model.*

*Also,*

$$dW_t \cdot d\hat{W}_t = \rho dt$$

*where $\rho$ is the correlation between the Brownian motions.*

*Note that $\nu$ and $f$ from $\sigma_t = f(y_t)$ depends on the model chosen. The stochastic volatility model we are going to use is the CIR (Cox-Ingersoll-Ross) Heston model: fixing $\rho < 0$, $f(y) = \sqrt{y}$ and $\nu = \xi\sqrt{y_t}$, where $\xi$ is often called vol-of-vol, as it can be thought of being the volatility of the volatility.*

*All in all, the Heston model is the following:*

$$\begin{cases} dS_t = \hat{\mu} S_t dt + \sigma(t, S_t) dW_t^S \\ dy_t = \kappa(\theta - y_t)dt + \xi\sqrt{y_t} dW_t^y \end{cases}$$

*The following are 20 simulated paths of an asset following a Heston model, with the following parameters: $T = 1$ maturity, $N = 10$ number of discretized dates (we discretize the continuous time in time steps $dt = T/N$, $NSim = 20$ number of simulations, $\theta = 0.2$, $\mu = 0.05$, $k = 5$, $\xi = 0.05$, $\sigma = 0.20$, $\rho = -0.2$ and finally the current spot price being $S0 = 100$ and the initial volatility being $\sigma_0 = 0.2$.*

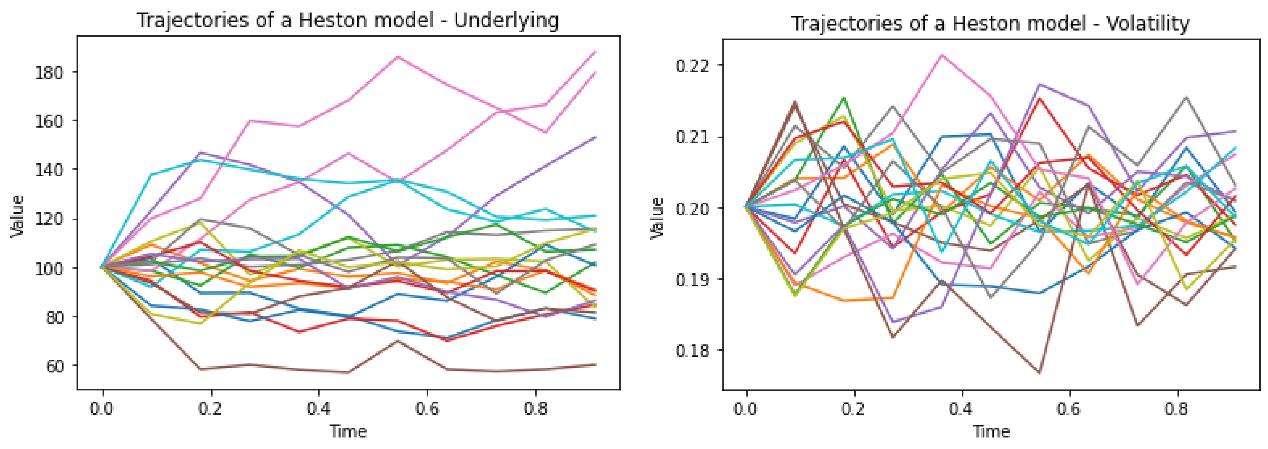


Figure 7: Simulation of 20 paths of an asset modelled with a Heston.

[3]Note that if $\theta - y(t) < 0$ the process decreases, if $\theta - y(t) = 0$ $y_t$ is a 0-drift process and if $\eta - y(t) > 0$ the process increases.

## 5.3 Options as financial instruments

*A* **call/put option** *is a financial contract in which the buyer (of the contract) is given the option of buying/selling the underlying asset at an agreed predetermined price or* **strike** *$K$ at the maturity $T$ (if is it an European option) or the option is given at discrete time stamps $t_1 < t_2 < \cdots < t_n = T$ until $T$ inclusive (American option).*

*Hence, there is the following classification for standard options:*

- *Call or Put.*
- *American or European style.*

*In practice, when pricing American options, we have to discretize the time, and so, we call them* **Bermudan options**. *A Bermudan Option is, then, an option that, for $t \leq \tau \leq T$, we can exercise at each $\tau$ (American optionality) such that $\tau \in I$, where $I$ is a collection of time stamps (discrete monitoring).*

*Mathematically, a financial derivative is expressed as a function $D_T$ such that $D_T : \Omega \to \mathbb{R}$ is measurable at $\mathcal{F}_T$. Therefore, the pricing of any financial derivative at any time $t \in [0, T]$ is:*

$$D_t = e^{-r*(T-t)} \cdot \mathbb{E}_q[D_T \,|\, \mathcal{F}_t].$$

*where $q$ is the risk-neutral measure and the exponential term is the discount factor.*

*We will not display extensively the details about the risk-neutral measure $q$ but, all in all, can be thought of a "fair", "just" probability of a certain event occurring or not adjusted to risk, and, using different theoretical results such as having the discounted price process as a martingale or the exponential martingale criterion (amongst others), one can determine the conditions that need to be established for the dynamics of the underlying asset to be in $q$ and hence, the expectation can be computed in $q$ as well.*

*Back to options, a call option is characterized by the following $D_T$:*

$$D_T = max(S_T - K, 0)$$

*A put option, on the other hand, is $D_T = max(K - S_T, 0)$, where $K$ is the strike price.*

*However, there exist more complex options called* **exotic options**, *hybrid (not American nor European) options that are often customizable to the needs of the investor. They allow for more flexibility, and there are also different contracts depending on whether the strike is fixed $K$ or floating: for example, set as the strike an average of the prices of the underlying asset up until maturity. This is the case of an Asian option.*

*Also, one could set up an option that takes into account $d$ different assets, thus being a* **multidimensional option**.

*Notice that there are endless conditions one can establish and hence, endless different options.*

## 5.4 Method

*If one codes the way the pseudo-algorithms are detailed, one will find that the code runs slow. By slow, we mean days-slow (even weeks, if we want to achieve at least comparable bounds to the ones in the paper). This is why we need* **optimization techniques**. *We are going to detail the ones we used.*

### 5.4.1 Optimization techniques

**Batch normalization**. *Batch normalization is a technique that aims to improve the consistency and the computational efficiency of neural networks during training. A usual problem found in Machine Learning (ML) is the internal covariate shift (change in distribution of input values as the parameters are updated by the net).*

*The technique is based on the normalization of the activations of a given layer, given a batch (that is, a collection) of inputs. Having the activations with zero mean and unit variance (normalized) help the network converge faster and more consistently: else, depending on the run, we would have different results.*

*Our code is based in PyTorch, (refer to the Appendices to learn how to run the code) a ML library for Python. This is our neural network's architecture, and this is how we made use of Batch Normalization:*

Listing 1: Notice torch.nn.BatchNorm1d.

```
class Neural_Net_NN(torch.nn.Module):
    def __init__(self, M,shape_input):
        super(Neural_Net_NN, self).__init__()
        self.dense1 = nn.Linear(shape_input,M)
        self.dense2 = nn.Linear(M,1)
        self.bachnorm1 = nn.BatchNorm1d(M)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.bachnorm1(self.relu(self.dense1(x.float())))
        x = self.relu(self.dense2(x))
        return x
```

*The net involves two hidden layers, as stated by the authors. Each of the hidden layers are linear. Also, note that we did not use the sigmoid, as one can apply the sigmoid to the output of the net to get $F_n$ anyway. $M$ is the size of the hidden layers' size, which is $q_1 = q_2 = (d + 40)$, and shape input is the dimension of the output, which is d $(+1$, now we will explore the reason why).*

**Training the payoff**. *As mentioned in the paper, it seems as if training $Y_t = (X, g(t, X_t))$ the network converges much faster, maybe because instead of accessing a function outside the net's architecture, one has all the necessary components to train inside the net. This is how we create $Y_t$ and hand it to the neural network.*

Listing 2: Subtleties to catch sight of.

```
neural_net = Neural_Net_NN(num_neurons,d+1).to(device)
X,p_,g_tau = x, p, p[:,:,n-1]
X,p_,g_tau = X.to(device), p_.to(device),g_tau.to(device)
state = torch.cat((X,p_),axis = 1)
loss = np.zeros(1)
# Afterwards in the code...
net_n = neural_net(state[:,:,n])
F_n   = torch.sigmoid(net_n)
loss -= torch.mean(p_[:, :, n] * F_n + g_tau * (1. - F_n))
g_tau = torch.where(net_n > 0, p_[:, :, n], g_tau)
```

*Notice how, making use of torch.cat, one can concatenate both $X$ and p=$g(X)$, where $g$ is one or another depending on the optimal stopping problem we are trying to solve. Also, notice how, at first, $g_\tau$ is $g$ applied* *to $X_N$ and we hand it backwards in time according to the net applied to $Y_n$ and instead of the condition $F_n > 0.5$, we use $net_n > 0$, and hence the sigmoid is only used to compute the loss at every step.*

*It is important to note that we send our PyTorch tensors (easier to manipulate by the package than numpy arrays, for instance) to device, where device is:*

Listing 3: Device.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

*Torch.cuda is a module that allows the program to use GPU (Graphical Processing Units), accelerating computations (CUDA is a parallel computing platform).*

**Communication with Sebastian Becker**

*Up until this point, we had codes (not the ones snippets of which we are providing) that, to compute the prices and bounds with sufficient simulations to be comparable to the Becker et al results for the Bermudan option, we had to wait for more than 7 days. Shortly before finishing this thesis, we sent an email to the authors, asking for any clue, out of mere curiosity, to how to drastically speed up the code (as Becker et al compute the results for the Bermudan option in 20 seconds, for $S_0 = 90, d = 2$, for instance). We are extremely grateful, especially to Sebastian Becker. Almost immediately after sending him an email, he responded to us, providing key insights and hints to accelerate the computations. We are deeply thankful to him for his valuable assistance.*

*In the personal communication we had with him, he mentioned the following:*

- *Adding the payoff at every exercise point makes the algorithm converge faster.*
- *Training all NN at once makes it faster.*
- *Tweaking the learning rate and other parameters of ADAM makes it faster and more consistent.*

*After this, we began coding again from scratch and, instead of using our code that, although working, was slow, we programmed a code that is extremely fast and competitive. The main drawback of this code is that we do not use the upper bound calculation, in favor of the speed of our code. The positive part is that our lower bound converges really fast to the Becker et al values that they provide. It is important to note that the confidence intervals we provide are the ones attributed to the lower bound.*

**Training the nets simultaneously** *and* **scheduling and tweaking the learning rates***. Here is the snippet of the code that trains all the nets simultaneously.*

Listing 4: Training.

```
    neural_net = Neural_Net_NN(num_neurons,d+1).to(device)
    optimizer = torch.optim.Adam(neural_net.parameters(), lr=0.05)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,
        milestones=lr_boundaries, gamma=0.1)
    px_hist = []
    max_px = 0
    for train_step in range(training_steps):
        X,p_,g_tau = x, p, p[:,:,n-1]
        X,p_,g_tau = X.to(device), p_.to(device),g_tau.to(device)
        state = torch.cat((X,p_),axis = 1)
        loss = np.zeros(1)
        loss = torch.tensor(loss).to(device)
```

```
12          for n in range(N-2, -1, -1):
13              net_n = neural_net(state[:,:,n])
14              F_n   = torch.sigmoid(net_n)
15              loss -= torch.mean(p_[:, :, n] * F_n + g_tau * (1. - F_n))
16              g_tau = torch.where(net_n > 0, p_[:, :, n], g_tau)
17          px_mean_batch = torch.mean(g_tau)
18          loss = torch.mean(loss)
19          px_hist.append(px_mean_batch.item())
20          if train_step > 10:
21              if px_mean_batch.item()>max_px and
                   px_mean_batch.item()<np.mean(px_hist[-10:-1])*1.50:
22                  torch.save(neural_net.state_dict(), 'best_model.pt')
23                  max_px = px_mean_batch.item()
24          if train_step%100 == 0:
25              print('| Train step: {:5.0f} | Loss: {:3.3f} | V: {:3.3f} | Lr:
                   {:1.6f}
                   |'.format(train_step,loss.item(),px_mean_batch.item(),optimizer.param_g
26          optimizer.zero_grad()
27          loss.backward()
28          optimizer.step()
29          scheduler.step()
```

*First, notice the clever trick of training all the nets simultaneously: computing the forward pass of the neural network at time $n$ (from $T - T/N$ to $T/N$) is enough, we do not need $N$ different nets with the same architecture.*

*On the other hand, the optimizer (ADAM, as mentioned by Becker), goal of which is to minimize the cost function (loss, in our case) by adjusting the parameters, is wrapped in the Learning Rate (LR) scheduler MultiStepLR. The function of the LR scheduler is to decay the learning rate of each parameter group by gamma once the number of training iterations (epochs) reaches a milestone. In other words, one can establish the expected epochs in which it is important to change the learning rate of the parameters and every time the boundary epoch is crossed, $LR = LR * gamma$. This is useful because if the LR is too low, we have a slow convergence, needing more epochs, and if the LR is too high, the model is unstable and it may not converge. Therefore, it is important to delicately decide on which LR boundaries to fix.*

*Another aspect is to fix* **financially acceptable conditions***: we keep the history of the mean of every estimated current price, that is, an estimate of $V$ and subject it to the following conditions (look at line* 21*): if the current estimated price is higher than the previous observed maximum price and the current estimated price is less than* 50% *higher than the last* 10 *estimated prices (acceptability range), then take it into account for the estimation of $V$, else, we consider that it is not a realistic value and repel it. In practice, this will eliminate outliers arising from the underlying asset model.*

### 5.4.2 Multi-dimensional Bermudan option

*We will apply the optimal stopping problem to the pricing of multi-dimensional Bermudan max-call options with the underlying asset being driven by a multi-dimensional Black-Scholes model, replicating the results obtained by Becker et al and comparing them with the literature.*

**Multidimensional Black-Scholes underlying assets.** *Let us assume:*

$$S_t^i = s_0^i \exp\left(\left[r - \delta_i - \sigma_i^2/2\right] \cdot t + \sigma_i W_t^i\right), \quad i = 1, 2, \dots, d, \tag{20}$$

*where:*

- $s_0^i \in (0,\infty)$ *are the initial values.*
- $r \in \mathbb{R}$ *is the risk-free interest rate.*
- $\delta_i \in [0,\infty)$ *are the dividend yields.*
- $\sigma_i \in (0,\infty)$ *are the volatilities.*
- *$W$ is a $d$-dimensional Brownian motion with constant instantaneous correlations $\rho_{ij} \in \mathbb{R}$ between the components $W^i$ and $W^j$. That is:*

$$\mathbb{E}\left[(W_t^i - W_s^i)(W_t^j - W_s^j)\right] = \rho_{ij}(t-s)$$

  *for all $i \neq j$ and $s < t$.*

*A Bermudan max-call option on $S^1, S^2, \dots, S^d$ has payoff $P = \left(\max_{1\le i\le d} S_t^i - K)\right)^+$. It can be exercised at any point of a time grid $0 = t_0 < t_1 < \dots < t_N$. Its price is given by:*

$$\sup_{\tau} \mathbb{E}\left[\exp^{-r\tau}\left(\max_{1\le i\le d} S_\tau^i - K\right)^+\right],$$

*where the supremum is over all $S$-stopping times taking values in $\{t_0, t_1, \dots, t_N\}$.*

*The translation of this problem to our method starts with denoting $X_n^i = S_{t_n}^i$, $n = 0, 1, \dots, N$, and letting $\mathcal{T}$ be the set of $X$-stopping times. Then, the price can be written as $\sup_{\tau\in\mathcal{T}} \mathbb{E}g(\tau, X_\tau)$ for:*

$$g(n,x) = e^{-rt_n}\left(\max_{1\le i\le d} x^i - K\right)^+.$$

*For the replication of the results with respect to the ones in Becker et al, we will assume the time grid to be of the form $t_n = nT/N$, $n = 0, 1, \dots, N$ for a maturity $T > 0$ and $N + 1$ equidistant exercise dates.*

*Other adjustments that the authors fix are:*

- *Number of nodes of the hidden layers: $q_1 = q_2 = d + 40$.*
- *Number of training steps: $3,000 + d$*
- *For each training step, number of paths of $(X_n)_{n=0}^N$: $8,192$.*
- *Number of trial paths for the lower bound $L$: $K_L = 4,096,000$.*
- *Number of trial paths for the upper bound $U$: $K_U = 1,024$ $(z_n^k)_{n=0}^N$, $k = 1, \dots, K_U$ of $(X_n)_{n=0}^N$.*
- *Also, $K_U \times J$ realizations $(v_n^{k,j})_{n=1}^N$, $k = 1, \dots, K_U$ and $j = 1, \dots, J$ of the increments $(W_{t_n} - W_{t_{n-1}})_{n=1}^N$ with $J = 16,384$.*

  *Then, for all $n$ and $k$, the $i$-th component of the $j$-th continuation path departing from $z_n^k$ is:*

$$\tilde{z}_m^{i,k,j} = z_n^{i,k} \exp\left([r - \delta_i - \sigma_i^2/2](m-n)\delta t + \sigma_i[v_{n+1}^{i,k,j} + \dots + v_m^{i,k,j}]\right), \quad m = n+1, \dots, N.$$

**Symmetrical case**

*The first case is supposing the volatilities are not only constant but the same for every asset: $\sigma_i = \sigma$, and also $s_0^i = s_0$, $\delta_i = \delta$ and $\rho_{ij} = \rho$ for all $i \neq j$.*

**Asymmetrical case**

*As a second case, $s_0^i = s_0$, $\delta_i = \delta$ and $\rho_{ij} = \rho$ for all $i \neq j$ but different volatilities, $\sigma_1 < \sigma_2 < \cdots < \sigma_d$. Further specifications on this example are that for $d \leq 5$, we choose $\sigma_i = 0.08 + 0.32 \times (i-1)/(d-1)$, $i = 1, 2, \dots, d$. For $d > 5$, $\sigma_i = 0.1 + i/(2d)$, $i = 1, 2, \dots, d$.*

*Regarding the rest of the parameters, the parameters for the summary results of the code that Becker et al used are: $r = 5\%$, $\delta = 10\%$, $\rho = 0$, $K = 100$, $T = 3$, $N = 9$. Although the implementation is in the code provided, we will not provide the summary results for this asymmetrical case.*

*Here is the comparison of some of the results for the valuation of the American Option with different d and $S_0$ against a fixed strike $K$ of 100, comparing it with both the results of Becker and the literature. For literature, we understand the results (using a binomial lattice method) of Andersen and Brodie (2004) (for $d = 2, 3$) and Broadie and Cao (2008) (for $d = 5$). Notice that, in some cases, the best we can achieve in the literature is a confidence interval instead of a point estimate.*

Table 1: **Summary results for the Bermudan: symmetrical case**

| *d* | *$S_0$* | *B's point estimate* | *B's 95% CI* | *Point estimate* | *95% CI* | *Literature* |
|---|---|---|---|---|---|---|
| *2* | *90* | *8.074* | *[8.060, 8.081]* | *8.078* | *[7.964, 8.193]* | *8.075* |
| *2* | *100* | *13.899* | *[13.880, 13.910]* | *14.215* | *[14.083, 14.347]* | *13.902* |
| *2* | *110* | *21.349* | *[21.336, 21.354]* | *20.443* | *[20.301, 20.584]* | *21.345* |
| *3* | *90* | *11.287* | *[11.276, 11.290]* | *11.075* | *[10.967, 11.183]* | *11.29* |
| *3* | *100* | *18.690* | *[18.673, 18.699]* | *19.998* | *[19.831, 20.165]* | *18.69* |
| *3* | *110* | *27.573* | *[27.545,27.591]* | *28.318* | *[28.158, 28.478]* | *27.58* |
| *5* | *90* | *16.644* | *[16.633, 16.648]* | *16.224* | *[16.075, 16.373]* | *[16.620, 16.653]* |
| *5* | *100* | *26.159* | *[26.138, 26.174]* | *26.529* | *[26.351, 26.708]* | *[26.115, 26.164]* |
| *5* | *110* | *36.772* | *[36.745, 36.789]* | *37.93* | *[37.719, 38.145]* | *[36.710, 36.798]* |
| *10* | *90* | *26.240* | *[26.189, 26.289]* | *26.469* | *[26.3, 26.637]* | |
| *10* | *100* | *38.337* | *[38.3, 38.367]* | *39.632* | *[39.447, 39.817]* | |
| *10* | *110* | *50.886* | *[50.834, 50.937]* | *50.934* | *[50.729, 51.138]* | |
| *20* | *90* | *37.802* | *[37.681, 37.942]* | *37.477* | *[37.311, 37.642]* | |
| *20* | *100* | *51.668* | *[51.549, 51.803]* | *51.864* | *[51.665, 52.062]* | |
| *20* | *110* | *65.628* | *[65.470, 65.812]* | *66.592* | *[66.373, 66.811]* | |
| *30* | *90* | *44.953* | *[44.777, 45.161]* | *44.716* | *[44.536, 44.895]* | |
| *30* | *100* | *59.659* | *[59.476, 59.872]* | *59.416* | *[59.224, 59.609]* | |
| *30* | *110* | *74.221* | *[74.196, 74.566]* | *73.282* | *[73.059, 73.505]* | |
| *50* | *90* | *54.057* | *[53.883, 54.266]* | *56.868* | *[56.51, 57.226]* | |
| *50* | *100* | *69.736* | *[69.560, 69.945]* | *74.964* | *[74.588, 75.34]* | |
| *50* | *110* | *85.463* | *[85.204, 85.763]* | *87.767* | *[87.306, 88.223]* | |
| *100* | *90* | *66.556* | *[66.321, 66.842]* | *68.330* | *[67.928,68.733]* | |
| *100* | *100* | *83.584* | *[83.357, 83.862]* | *84.491* | *[84.082, 84.900]* | |
| *100* | *110* | *100.663* | *[100.394, 100.989]* | *102.012* | *[101.574, 102.450]* | |
| *200* | *90* | *79.174* | *[78.971, 79.416]* | *78.444* | *[77.983, 78.905]* | |
| *200* | *100* | *85.463* | *[83.357, 83.862]* | *83.812* | *[83.259, 84.365]* | |
| *200* | *110* | *116.088* | *[115.774, 116.472]* | *118.482* | *[117.745, 119.215]* | |
| *500* | *90* | *96.147* | *[95.934, 96.407]* | *92.836* | *[92.291, 93.381]* | |
| *500* | *100* | *116.425* | *[116.210, 116.685]* | *113.203* | *[112.427, 113.979]* | |
| *500* | *110* | *136.765* | *[136.521, 137.064]* | *134.536* | *[133.738, 135.33]* | |

*Given the high computational cost of the problem and our (both limited) hardware and programming skills (there is still room for improvement in the codes provided in the thesis), the results do not fully align with Becker et al. However, the maximum error is always relatively little, hence the pricing is of theoretical importance. We believe the reason why our results do not completely align is that our convergence is a bit off because we do not tweak sufficiently the learning rates when training the nets. Other reasons would involve our "financially reasonable" or "acceptable" conditions for the batches of data are not correct or are too harsh and are missing out for important data.*

*We also compare the computation times. Please note that the computation time for Becker et al takes into account the upper bound, which is harder to calculate than the lower bound. This table is only to get a grasp on the difference of times we can achieve.*

Table 2: **Running times in Bermudan: symmetrical case**

| *d* | $S_0$ | *B's computation time (seconds)* | *Computation time (seconds)* |
|---|---|---|---|
| 2 | 90 | 54 | 32 |
| 2 | 100 | 54 | 32 |
| 2 | 110 | 54 | 32 |
| 3 | 90 | 55 | 32 |
| 3 | 100 | 55 | 32 |
| 3 | 110 | 53 | 32 |
| 5 | 90 | 56 | 33 |
| 5 | 100 | 56 | 33 |
| 5 | 110 | 54 | 33 |
| 10 | 90 | 64 | 36 |
| 10 | 100 | 64 | 35 |
| 10 | 110 | 64 | 35 |
| 20 | 90 | 81 | 42 |
| 20 | 100 | 81 | 44 |
| 20 | 110 | 81 | 43 |
| 30 | 90 | 101 | 53 |
| 30 | 100 | 101 | 53 |
| 30 | 110 | 101 | 54 |
| 50 | 90 | 108 | 40 |
| 50 | 100 | 102 | 40 |
| 50 | 110 | 102 | 41 |
| 100 | 90 | 243 | 66 |
| 100 | 100 | 243 | 66 |
| 100 | 110 | 243 | 65 |
| 200 | 90 | 444 | 96 |
| 200 | 100 | 444 | 97 |
| 200 | 110 | 444 | 96 |
| 500 | 90 | 1254 | 273 |
| 500 | 100 | 1255 | 274 |
| 500 | 110 | 1255 | 278 |

### 5.4.3 Callable Multi Barrier Reverse Convertibles

*A* **Multi Barrier Reverse Convertible** *(MBRC) is a complex financial investment that combines elements of a bond and an option. It allows investors to earn income while being exposed to market changes. When you invest in an MBRC, you are essentially lending money to a financial institution for a specific period. In return, the institution provides you with regular interest payments known as coupon payments.*

*MBRCs also have an additional feature related to the performance of an underlying asset, which could be a stock, an index, or a group of assets. Here's how it works: The issuer sets barrier levels for the underlying asset, typically lower than its initial price. If the price of the underlying asset touches or falls below any of these barriers, instead of receiving the principal of the investment, you may receive a predetermined number of shares of the underlying asset.*

*A Callable MBRC has an extra feature where the issuer has the option to redeem the underlying asset before its maturity date. If the issuer chooses to do so, you, as the investor, receive the principal amount along with an interest payment.*

*In simpler terms, a Callable MBRC can be thought of as a bond that can convert into shares of the worst-performing d underlying assets if a specific trigger event happens.*

*We will propose the assumption that Becker et al also make, which is that the price of the i-th underlying asset in percent of its starting value follows the dynamics:*

$$S_t^i = \begin{cases} 100exp([r-\sigma_i^2/2]t+\sigma_i W_t^i) & \text{for } t \in [0, T_i) \\ 100(1-\delta_i)exp([r-\sigma_i^2/2]t+\sigma_i W_t^i) & \text{for } t \in [T_i, T] \end{cases}$$

*where $r$ interest rate, $\sigma_i$ volatility, $T$ maturity, the dividend rate $\delta_i$, a d-dimensional Brownian motion $W$ with constant correlations $\rho$ and $T_i$ being the dividend payment time.*

*We also will consider a MBRC that pays $c$ at each $N$ time points $t_n = nT/N$ for $n = 1, 2, \dots, N$ and makes a payment at time $T$ of:*

$$G = \begin{cases} F & \text{if } \min_{1\le i\le d}\min_{1\le m\le M} S_{u_m}^i > B \text{ or } \min_{1\le i\le d} S_T^i > K \\ \min_{1\le i\le d} S_T^i & \text{if } \min_{1\le i\le d}\min_{1\le m\le M} S_{u_m}^i \le B \text{ or } \min_{1\le i\le d} S_T^i \le K \end{cases}$$

*where $F \in [0,\infty)$ is the nominal amount, $B \in [0,\infty)$ a barrier, $K \in [0,\infty)$ a strike price and $u_m$ the end of the m-th trading day.*

*The value of the MBRC is, then:*

$$\sum_{n=1}^{N} e^{-rt_n}c + e^{-rT}\mathbb{E}G$$

*and will be estimated via Monte Carlo approximation. The $e{-}rt_n$ discount factor term multiplied by the coupon $c$ is translating the price of the coupon at $t_n$ at the current time and correspondingly for $e^{-rT}$.*

*Since the MBRC can be redeemed at any time $t_1, t_2, \dots, t_{N-1}$ by paying back the notional, to reduce costs, the issuer of the MBRC will try an find a stopping time between $t_1, t_2, \dots, T$ such that:*

$$\mathbb{E}\left[\sum_{n=1}^{\tau} e^{-rt_n}c + \mathbb{1}_{\{\tau<T\}}e^{-r\tau}F + \mathbb{1}_{\{\tau=T\}}e^{-rT}G\right]$$

*is minimal. That is, minimize the payment he has to issue to the holder of the MBRC.*

*A usual trick when simulating barriers is to increase by one the dimension in the following sense. Let $(X_n)_{n=1}^N$ be the $d+1$-dimensional Markov process given by $X_n^i = S_{t_n}^i$ for $i = 1, \dots, d$ and:*

$$X_n^{d+1} := \begin{cases} 1 & \text{if the barrier has been breached before or at } t_n. \\ 0 & \text{else.} \end{cases}$$

*Then the problem is reduced to:*

$$\inf_{\tau\in\mathcal{T}} \mathbb{E}g(\tau, X_\tau),$$

*where $\mathcal{T}$ is the set of all X-stopping times and*

$$g(n,x) = \begin{cases} \sum_{m=1}^{n} e^{-rt_m}c + e^{-rt_n}F & \text{if } 1 \le n \le N-1 \text{ or } x^{d+1} = 0 \\ \sum_{m=1}^{N} e^{-rt_m}c + e^{-rt_N}h(x) & \text{if } n = N \text{ and } x^{d+1} = 1 \end{cases}$$

*where*

$$h(x) = \begin{cases} F & \text{if } \min_{1\le d\le d} x^i > K \\ \min_{1\le i\le d} x^i & \text{if } \min_{1\le i\le d} x^i \le K \end{cases}$$

*Since it is a minimization problem, the cost function must be the opposite, that is*

$$loss \to -loss.$$

*Some adjustments that the authors fix are:*

- *Number of nodes of the hidden layers:* $q_1 = q_2 = d + 40$.
- *Number of training steps:* $3,000 + d$
- *For each training step, number of paths of* $(X_n)_{n=0}^N$: $8,192$.
- *Number of trial paths for the lower bound L:* $K_L = 4,096,000$.
- *Number of trial paths for the upper bound U:* $K_U = 1,024$ $(z_n^k)_{n=0}^N$, $k = 1, \dots, K_U$ *of* $(X_n)_{n=0}^N$.
- *Also,* $K_U \times J$ *realizations* $(v_n^{k,j})_{n=1}^N$, $k = 1, \dots, K_U$ *and* $j = 1, \dots, J$ *of the increments* $(W_{t_n} - W_{t_{n-1}})_{n=1}^N$ *with* $J = 1024$. *Then, for all n and k, the i-th component of the j-th continuation path departing from* $z_n^k$ *is:*

$$\tilde{z}_m^{i,k,j} = z_n^{i,k} \exp\left([r - \delta_i - \sigma_i^2/2](m-n)\delta t + \sigma_i[v_{n+1}^{i,k,j} + \dots + v_m^{i,k,j}]\right), \quad m = n+1, \dots, N.$$

*The parameters that the authors adjust are* $F = K = 100$, *barrier* $B = 70$, *maturity* $T = 1$, $N = 12$, $c = 7/12$, $\delta_i = 5\%$, $T_i = 1/2$, $r = 0$, $\sigma_i = 0.2$ *and* $\rho_{ij} = \rho$.

Table 3: **Summary results for the MBRCs**

| $d$ | $\rho$ | *B's point estimate* | *B's 95% CI* | *Point estimate* | *95% CI* |
|---|---|---|---|---|---|
| 2 | 0.6 | 98.243 | [98.213, 98.263] | 99.81570281982422 | [99.81570280794416 , 99.81570283170427] |
| 2 | 0.1 | 97.634 | [97.609, 97.646] | 100.18012084960938 | [100.1801208139692 , 100.18012088524955] |
| 3 | 0.6 | 97.634 | [96.906, 96.948] | 100.17860717773438 | [100.17860715397426 , 100.17860720149449] |
| 3 | 0.1 | 95.244 | [95.216, 95.258] | 100.3432403564453 | [100.34324032080514 , 100.34324039208549] |
| 5 | 0.6 | 94.872 | [94.837, 94.894] | 99.71725769042969 | [99.71725769042969 , 99.71725769042969] |
| 5 | 0.1 | 90.810 | [90.775, 90.828] | 99.94874725341796 | [99.94874724153792 , 99.94874726529802] |
| 10 | 0.6 | 91.599 | [91.536, 91.645] | 100.40985107421875 | [100.40985102669852 , 100.40985112173898] |
| 10 | 0.1 | 83.123 | [83.078, 83.153] | 100.08015594482421 | [100.0801559210641 , 100.08015596858434] |
| 30 | 0.6 | 86.126 | [86.041, 86.180] | 100.04830780029297 | [100.04830776465279 , 100.04830783593314] |
| 30 | 0.1 | 72.393 | [71.830, 72.760] | 100.34488372802734 | [100.34488369238717 , 100.34488376366753] |
| 50 | 0.6 | | | 100.14592590332032 | [100.14592583118117, 100.14592597545945] |
| 50 | 0.1 | | | 84.43875732421876 | [84.43875727612598, 84.43875737231151 ] |

*Again, notice the importance of settling correctly the neural networks: in complex financial derivatives, the error seems to amplify. In this case, the pricing is not of great importance since the error committed is too high.*

*This is our g (most probably inaccurate) to price MBRCs:*

```

def g(_t, _x, barrier = None, axis = 1):
    elif barrier == "MBRC":
            c = 7/12.
```

```
        B = 70
        F = 100
        Breach = torch.zeros(outer_batch_size, N)
        G = torch.zeros(outer_batch_size, 1, N)
        for j in range(outer_batch_size):
            Breach[j, 0] = torch.tensor([0])
            for n in range(N):
                suma = 0
                H = torch.min(_x[j, :, n])
                if n-1 == 0:
                    Breach[j, n] = torch.where(H > B,
                        torch.tensor([1]), torch.tensor([0]))
                else:
                    Breach[j, n] = torch.max(Breach[j,n-1],
                        torch.where(H > B, torch.tensor([1]),
                        torch.tensor([0])))
                if Breach[j, n] == torch.tensor([0]) or n < N-1:
                    for m in range(n):
                        suma += (torch.exp(-r*_t[m])*c)
                    suma += torch.exp(-r*_t[n])*F
                else:
                    for m in range(n):
                        suma += (torch.exp(-r*_t[m])*c)
                    H = torch.min(_x[j,:,-1])
                    a = torch.tensor([K])
                    if H > a:
                        H = K
                    suma += (torch.exp(-r*_t[-1]) * H)
                G[j,0,n] = suma
        return G
```

*The main idea to price barrier options relies on the Breach tensor / array. Breach*$[n] == 1$ *if the barrier price* $B$ *has been breached at time* $t_n$ *or before. Then, it triggers a specific outcome. If Breach*$[n]$ *were* $0$ *for all* $n \in \mathcal{T}$, *then it would be a standard Bermudan option.*

### 5.4.4 Model calibration and real-world options

*Hereafter we outline how to calibrate our model to price real-world options. Afterwards, we will propose different* $g$*'s to solve different options and even other optimal stopping problems.*

- *Gather data of* $M$ *different (American, Barrier, ...) options from the same asset and similar maturities* $T_i$.
- *Choose a model for the underlying asset. There are many different criteria. We will not go beyond the scope of the thesis, however we will recommend to choose a model based on: the ease of calibrating, the assessment of risk and empirical evidence.*
    - **Ease of calibrating**. *With Least Squares Minimization (a simple calibration technique), for instance, it will be easier to calibrate a Merton model than a Heston model, given that the number of parameters is higher for the Heston model and it would faster capture the dynamics*

  *of the market for the assigned asset or assets. It would also be wise to take into account the computational efficiency of calibrating as well.*
  - **Risk assessment**. *As we slightly mentioned, some models perform better with underlying assets bound to have unlikely events. Also, for instance, models with jumps help give an intuition for dividend payments or financial, unexpected crisis.*
  - **Empirical evidence**. *The models that we explored in this thesis have been tested vastly, and hence, there are a lot of empirical studies of the most popular assets in the financial market.*

- *Given a model, choose an initial guess of the parameters.*
- *We choose to perform Least Squares Minimization (LSM):*

$$\min_{params} \sum_{i=1}^{M} \left( \hat{V}_i - V_i(params) \right)^2$$

  *where $\hat{V}_i$ is the real price of the option and $V_i(params)$ is the price of the option using our method and the parameters $params$.*

- *Validate the model with out-of-sample data (reserved data that we will use from the same asset for testing and validation).*

*Once gathered the data for the $M$ option prices and chosen a model for the underlying asset, we propose the following code.*

Listing 5: Calibration

```python
import numpy as np
import underlying_asset_simulation as ua
from scipy.optimize import minimize
def least_squares_calibration(option_prices, strikes, N, Nsim, T, S0):
    def kou_model_loss(params):
        mu, sigma, lambdat, p, lambdap, lambdam = params
        model_prices = DOS(X_BS_Kou(N, Nsim, T, mu, sigma, S0, lambdat, p,
            lambdap, lambdam))
        mse = np.mean((model_prices - option_prices) ** 2)
        return mse

    initial_params = [0.1, 0.2, 0.5, 0.6, 1.0, 1.0]  # Initial guess for
        the parameters
    bounds = [(None, None), (0, None), (0, None), (0, 1), (0, None), (0,
        None)]  # Bounds for the parameters
    result = minimize(kou_model_loss, initial_params, bounds=bounds)
    return result.x
option_prices = [10.2, 5.1, 8.7]  # Placeholder values for option prices
strikes = [100, 110, 90]  # Placeholder values for option strikes
N = 100  # Number of time steps
Nsim = 10000  # Number of simulation paths
T = 1.0  # Time horizon
S0 = 100.0  # Initial stock price
calibrated_params = least_squares_calibration(option_prices, strikes, N,
    Nsim, T, S0)
print("Calibrated parameters:", calibrated_params)
```

*Here, we suppose the model we have chosen is the Kou model, that $M$ number of options is 3 and the maturities are all the same. Then, we assume that $DOS(x)$ computes the option price $V_i(params)$ given an underlying $X$. Note that is vital to set the bounds for the parameters, as, for instance, $p \in [0, 1]$ by definition of probability.*

## Further research

*The main focus of future research on this method would be the reduction of the high computational cost when performing the implementation proposed regarding the upper bound, this would make our implementation yet more interesting.*
*Another idea that we propose would be, from the different models for the underlying asset, to develop a professional library to have the full method automated for a broad range of options and underlying asset models:*

1. *Selection of the kind of option.*
2. *Selection of the underlying asset model.*
3. *Calibration of the code given a sample data set*
4. *Provide the confidence interval, plot of the price $V$, and the calibrated underlying asset.*

*Another future work would be to perform the model calibration for different assets in different financial sectors (energy commodities, technology assets, ...) and try the model for different settings.*

# References

*[1] Abbasi, W. et al. (2016).* Kou Jump Diffusion Model: An Application to the Standard and Poor 500, Nasdaq 100 and Russell 2000 Index Options *International Journal of Economics and Financial Issues, 2016, 6(4), 1918-1929.*

*[2] Andersen, Leif B.G. (2007).* Efficient Simulation of the Heston Stochastic Volatility Model.

*[3] Arratia, A. (2023). Lecture notes from the course* Data Science Applied to Finance (Degree of Data Science Engineering, UPC).

[4] Becker, S., Cheridito, P. and Jentzen, A. *Deep Optimal Stopping.* Journal of Machine Learning Research 20 (2019) **1-25**.

[5] Haugh, M., Kogan, L. (2004). *Pricing American options: a duality approach.* Operations Research, 52(2):258–270.

[6] Hornik, Stinchcombe, White (1989). *Multilayer feedforward networks are universal approximators.* Neural Networks 2, **359-366**.

[7] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). *Multilayer feed-forward networks with a nonpolynomial activation function can approximate any function.* Neural Networks, **6:861–867**.

[8] Masdemont, J., Planas, F. (2023). *Lecture notes from the course Mathematical Finance.*

[9] Marazzina, D. (2022). *Lecture notes from the course Computational Finance.*

[10] Øksendal, B. (2003). *Stochastic Differential Equations*, 6th ed. Springer-Verlag Berlin Heidelberg.

[11] Peskir, G. and Shiryaev, A. (2006). *Optimal stopping and free-boundary problems*, ser. Lectures in Mathematics ETH Zürich. Birkhäuser Verlag, Basel.

[12] Rogers, L., (2015). *Bermudan options by simulation.*

[13] Taraldsen, G. (2018). *Optimal Learning from the Doob-Dynkin lemma*,

# A. Finite Horizon theorem: the theoretical optimal stopping time

*We will construct a series of random variables $(S_n)_{0\leq n\leq N}^N$ that represents the gain and solves the problem in a stochastic sense. For $S_n$ to be the solution, we can derive some properties by analyzing the problem (3) we want to solve.*

*For $n = N$ we stop immediately and our gain is $S_N^N = g(N, X_N)$.*

*For $n = N - 1$ we can either stop or continue:*

- *If we stop, then our gain $S_{N-1}^N = g(N-1, X_{N-1})$.*
- *Else, if we continue optimally, our gain $S_{N-1}^N = \mathbb{E}[S_N^N \,|\, \mathcal{F}_{N-1}]$. This shows that our decision about stopping or continuing at time $n = N - 1$ must be based on information contained in $\mathcal{F}_{N-1}$.*

*Remark* A.1. We will denote $G_k := g(k, X_k)$ from now on, for simplicity.

*Since we seek to maximize the reward function $g$, then the method leads to a sequence of random variables $(S_n^N)_{0\leq n\leq N}$ defined recursively as (we will denote this the method of backward induction):*

$$\begin{aligned} S_n^N &= G_N && \text{for } n = N, \\ S_n^N &= \max(G_n, \mathbb{E}[S_{n+1}^N \,|\, \mathcal{F}_n]) && \text{for } n = N-1, \dots, 0. \end{aligned}$$

*Notice that the method itself decodes an optimal stopping time:*

$$\tau_n^N = \inf\{n \leq k \leq N \,:\, S_k^N = G_k\}$$

*The fundamentals of this method are enclosed in the Finite Horizon theorem. Before that, let us define what the Snell envelope is:*

**Definition A.2** (Snell envelope (of a stochastic process))**.** Intuitively, the Snell envelope is the smallest supermartingale dominating a stochastic process. The formal definition is the following. Given a filtered space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\in[0,T]}, \mathbb{P})$ and an absolutely continuous probability measure $\mathbb{Q} \ll \mathbb{P}$, then an adapted process $U = (U_t)_{t\in[0,T]}$ is the Snell envelope with respect to $\mathbb{Q}$ of the process $X = (X_t)_{t\in[0,T]}$ if

- $U$ is a $\mathbb{Q}$-supermartingale.
- $U$ dominates $X$. That is, $U_t \geq X_t$ $\mathbb{Q}$-almost surely for all $t \in [0, T]$.
- If $V = (V_t)_{t\in[0,T]}$ is a $\mathbb{Q}$-supermartingale which dominates $X$, then $V$ dominates $U$.

**Theorem A.3** (Finite Horizon)**.**

$$S_n^N \geq \mathbb{E}[g(\tau, X_\tau) \,|\, \mathcal{F}_n], \text{ for each } \tau \in \mathcal{T}_n^N, \tag{21}$$

$$S_n^N = \mathbb{E}[g(\tau_n^N, X_{\tau_n^N})]. \tag{22}$$

*Moreover, if $0 \leq n \leq N$ is given and fixed, then we have:*

- *the stopping time $\tau_n^N$ is optimal in $V_n^N = \sup_{n\leq\tau\leq N}\mathbb{E}[g(\tau, X_\tau)$.*
- *If $\tau^*$ is an optimal stopping time in $V_n^N$, then $\tau_n^N \leq \tau^*$ almost surely.*
- *The sequence $(S_k^N)_{n\leq k\leq N}$ is the smallest supermartingale that dominates $(g(k, X_k))_{n\leq k\leq N}$ i.e, $(S_k^N)_{n\leq k\leq N}$ is the Snell envelope of $(g(k, X_k))_{n\leq k\leq N}$.*
- *The stopped sequence $(S_{k\wedge\tau_n^N})_{n\leq k\leq N}$ is a martingale.*

*Proof.* The proof is carried out mainly by induction over $n = N, N-1, \dots, 0$.
As the base case, suppose $n = N$. Then:

$$g(N, X_N) = S_N^N \geq \mathbb{E}[G_\tau \,|\, \mathcal{F}_N]$$

for each $\tau \in \mathcal{T}_N^N$, but $\mathcal{T}_N^N = \{\tau = N\}$. Also,

$$g(N, X_N) = S_N^N = \mathbb{E}[G_{\tau_N^N} \,|\, \mathcal{F}_N] = g(N, X_N),$$

since $\tau_N^N = N$.

Let us now assume $S_n^N \geq \mathbb{E}[G_\tau \,|\, \mathcal{F}]$ for each $\tau \in \mathcal{T}_n^N$ and $S_n^N = \mathbb{E}[G_{\tau_n^N} \,|\, \mathcal{F}]$ both hold for $n = N, N-1, \dots, k$, where $k \geq 1$. Showing that it is also true for $n = k-1$, we will have proven both statements.

For (21):
Take $\tau \in \mathcal{T}_{k-1}^N$ and set $\bar{\tau} = \tau \wedge k$. Then, $\bar{\tau} \in \mathcal{T}_k^N$, and, since $\{\tau \geq k\} \in \mathcal{F}_{k-1}$ (at the time tic $k-1$, either $\tau = k-1$ or it is bigger than that):

$$\mathbb{E}[g(\tau, X_\tau) \,|\, \mathcal{F}_{k-1}] \overset{(1)}{=} \mathbb{E}\left[\mathbb{1}_{\tau=k-1} g(k-1, X_{k-1}) \,|\, \mathcal{F}_{k-1}\right] + \mathbb{E}\left[\mathbb{1}_{\tau\geq k} g(\bar{\tau}, X_{\bar{\tau}}) \,|\, \mathcal{F}_{k-1}\right] \overset{(2)}{=}$$

$$= \mathbb{1}_{\tau=k-1} g(k-1, X_{k-1}) + \mathbb{1}_{\tau\geq k}\mathbb{E}[\mathbb{E}[G_{\bar{\tau}} \,|\, \mathcal{F}_k] \,|\, \mathcal{F}_{k-1}] \overset{(3)}{\leq} \mathbb{1}_{\tau=k-1} S_{k-1}^N + \mathbb{1}_{\tau\geq k} S_{k-1}^N =$$
$$= S_{k-1}^N.$$

where

(1) By the algebra of optimal stopping times.

(2) Factoring the characteristic function out of the expectation.

(3) By the induction hypothesis, (21) holds for $k$, so, since $\bar{\tau} \in \mathcal{T}_k^N$, $\mathbb{E}[G_{\bar{\tau}} \,|\, \mathcal{F}_k] \leq S_k^N$. Also, by the justification of the definition of $(S_k^N)_{n\leq k\leq N}$ in our argumentation of the method of backward induction, both $g(k-1, X_{k-1}) \leq S_{k-1}^N$ and $\mathbb{E}[S_k^N \,|\, \mathcal{F}_{k-1}] \leq S_{k-1}^N$.

Now, let us prove (22). It will be enough to check that the inequality in (3) remains an equality when $\tau = \tau_{k-1}^N$.

Considering the definition of $\tau_n^N$, note that, when on $\{\tau_{k-1}^N \geq k\}$, $\tau_{k-1}^N = \tau_K^N$, and so:

$$\mathbb{E}[G_{\tau_{k-1}^N} \,|\, \mathcal{F}_{k-1}] = \mathbb{1}_{\{\tau_{k-1}^N=k-1\}} G_{k-1} + \mathbb{1}_{\{\tau_{k-1}^N\geq k\}}\mathbb{E}[\mathbb{E}[G_{\tau_k^N} \,|\, \mathcal{F}_k] \,|\, \mathcal{F}_{k-1}] \overset{(4)}{=}$$

$$= \mathbb{1}_{\{\tau_{k-1}^N=k-1\}} G_{k-1} + \mathbb{1}_{\{\tau_{k-1}^N\geq k\}}\mathbb{E}[S_k^N \,|\, \mathcal{F}_{k-1}] \overset{(5)}{=} \mathbb{1}_{\{\tau_{k-1}^N=k-1\}} S_{k-1}^N + \mathbb{1}_{\{\tau_{k-1}^N\geq k\}} S_{k-1}^N = S_{k-1}^N.$$

where

(4) Factoring the characteristic function out of the expectation.

(5) The first term because $G_{k-1} = S^N_{k-1}$ on $\{\tau^N_{k-1} = k-1\}$ by definition, and the second term because of the induction hypothesis.

- To show that the stopping time $\tau^N_n$ is optimal in $V^N_n = \sup_{\tau \in \mathcal{T}^N_n} \mathbb{E} g(\tau, X_\tau)$:

  Taking expectations on $S^N_k \geq \mathbb{E}[G_\tau \,|\, \mathcal{F}_k]$, we get $\mathbb{E}[S^N_n] \geq \mathbb{E}[G_\tau]$ for each $\tau \in \mathcal{T}^N_n$. So, taking the supremum over all the $\tau \in \mathcal{T}^N_n$, we see that $\mathbb{E}S^N_n \geq V^N_n$.

  On the other hand, taking expectations on $S^N_n = \mathbb{E}[G_{\tau^N_n} \,|\, \mathcal{F}_k]$, we get $\mathbb{E}[S^N_n] = \mathbb{E}[G_{\tau^N_n}]$. However, recall that the supremum is the lowest of the upper bounds (but still an upper bound), and so,

$$\mathbb{E}S^N_n = \mathbb{E}G_{\tau^N_n} \leq \sup_{\tau \in \mathcal{T}^N_n} \mathbb{E}G_\tau = V^N_n .$$

  Hence, we have an equality joining both of the inequalities together. Then, $\mathbb{E}[G_{\tau^N_n}] = \mathbb{E}[S^N_n] = V^N_n$, proving that $\tau^N_n$ is optimal for $V^N_n$.

- If $\tau^*$ is an optimal stopping time for $V^N_n$, then $\tau^N_n \leq \tau^*$ $P - a.s.$ (almost surely):

  To prove this, we introduce a claim:

  **Claim A.4.** *Optimality for $\tau^*$ implies $S^N_{\tau^*} = G_{\tau^*}$ $P - a.s.$*

  *Proof.* (Proof by contradiction) Otherwise, $S^N_k \geq G_k$ for $k$ such that $n \leq k \leq N$ (by the argument used in the method of backward induction). This implies that $S^N_{\tau^*} \geq G_{\tau^*}$ with probability $P(S_{\tau^*} > G_{\tau^*}) > 0$. It follows that

$$\mathbb{E}[G_{\tau^*} \,|\, \mathcal{F}_n] \overset{(6)}{<} \mathbb{E}[S^N_{\tau^*} \,|\, \mathcal{F}_n] \overset{(7)}{\leq} \mathbb{E}[S^N_n \,|\, \mathcal{F}_n] \overset{(8)}{=} V^N_n$$

  where

  (6) Since $S^N_{\tau^*} \neq G_{\tau^*}$ $P - a.s.$ This strict inequality will contradict the fact that $\tau^*$ is optimal.

  (7) By the optional stopping theorem (Theorem 4.15) and the supermartingale condition of the theorem.

  (8) By the argument in the above proof.

  Thus, $S^N_{\tau^*} = G_{\tau^*}$ $P - a.s.$ as claimed. □

  The fact that $\tau^N_n \leq \tau^*$ follows from the definition of $\tau^N_n = \inf\{n \leq k \leq N \,:\, S_k = G_k\}$. Hence, $\tau^N_n$ is a lower bound for any $\tau^*$ that fulfils $S_{\tau^*} = G_{\tau^*}$.

- The supermartingale property:
  From the backward induction method follows that $S^N_k \geq \mathbb{E}[S^N_{k+1} \,|\, \mathcal{F}_k]$ for all $k$ such that $n \leq k \leq N$. So, $(S^N_k)_{n \leq k \leq N}$ is a supermartingale.

  Again from the same argumentation when defining the method, $S^N_k \geq G_k$ $P - a.s.$ for $k$ such that $n \leq k \leq N$, so $(S^N_k)_{n \leq k \leq N}$ dominates $(G_k)_{n \leq k \leq N}$.

The fact that if $\tilde{S}_k$ is another supermartingale, then $\tilde{S}_k \geq S_k^N$ $P - a.s.$ can be verified by induction over $k = N, N-1, \dots, l$, where $l \in \mathbb{N}$.

If $k = N$ it follows by the argument of the backward induction method. Assuming $\tilde{S}_k \geq S_k^N$ $P - a.s.$ for $k = N, N-1, \dots, l$ with $l \geq n+1$, then

$$S_{l-1}^N = \max(G_{l-1}, \mathbb{E}[S_l^N \,|\, \mathcal{F}_{l-1}]) \overset{(9)}{\leq} \max(G_{l-1}, \mathbb{E}[\tilde{S}_l \,|\, \mathcal{F}_{l-1}]) \leq \tilde{S}_{l-1} \quad P - a.s.$$

(9) Using the induction hypothesis, i.e., the supermartingale property for $(S_k)_{n \leq k \leq N}$.

- The martingale property:

$$\mathbb{E}[S_{(k+1)\wedge\tau_n^N}^N \,|\, \mathcal{F}_k] = \mathbb{E}[\mathbb{1}_{\{\tau_n^m \leq k\}} S_{k\wedge\tau_n^N}^N \,|\, \mathcal{F}_k] + \mathbb{E}[\mathbb{1}_{\{\tau_n^m = k+1\}} S_{(k+1)}^N \,|\, \mathcal{F}_k] \overset{(10)}{=}$$

$$\mathbb{1}_{\{\tau_n^m \leq k\}} S_{k\wedge\tau_n^N}^N + \mathbb{1}_{\{\tau_n^m = k+1\}} S_k^N \overset{(11)}{=} S_{k\wedge\tau_n^N}^N$$

(10) Follows from $S_k^N = \mathbb{E}[S_{k+1}^N \,|\, \mathcal{F}]$ on the information $\{\tau_n^N \geq k+1\}$, and this belongs to $\mathcal{F}_k$ because $\tau_n^N$ is a stopping time.

(11) By the definition of $S_{k\wedge\tau_n^N}^N$.

The consequences of this theorem are the following:

- $V_0^N = V$ is solved inductively by solving the problems $V_n^N$ for $n = N, N-1, \dots, 0$.
- Also, the optimal stopping time rule $\tau_n^N$ for $V_n^N$ satisfies $\tau_n^N = \tau_k^N$ on $\{\tau_n^N \geq k\}$ for $0 \leq n < k \leq N$ having $\tau_k^N$ optimal stopping time for $V_k^N$.

  This means that, if it was not optimal to stop within $\{n, n+1, \dots, k-1\}$, then starting the observation at time $n$ and based on the information $\mathcal{F}_n$, the same optimality (i.e., the same solution $\tau_n^N$) applies for $\{k, k+1, \dots, N\}$. We note that this is the dynamic programming paradigm for algorithms.

□

# B. Simulating models

*Given an SDE, we will show how to efficiently simulate the dynamics.*

*Firstly, how to simulate one of the most important sources of randomness in stochastic processes: the Wiener process or Brownian motion. We will introduce the process and how to simulate its increments without being mathematically rigorous, but explaining the necessary concepts to follow and understand the final codes.*

**Definition B.1** (Wiener process or Brownian motion)**.** The Wiener process is defined as a continuous-time stochastic process $\{W_t\}_{t\geq 0}$. The properties that characterize the Wiener process are the following:

1. $W_0 = 0$
2. For any $s, t \in I \subset \mathbb{R}$ (assuming WLOG $s < t$),

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

   and the increment is also stationary and independent of the past.

3. The paths of a Wiener process are continuous (in the sense of a function) $W_t(w)$ for all $w \in Omega$.

*Let us focus on the second item on the definition of the Wiener process. We construct $\Delta W_t := W_{t+h} - W_t$. Then, for any $h$, $\Delta W_t \sim N(0, \Delta t)$, where $\Delta t = t + h - t$. We can take limits:*

$$dW_t \sim N(0, dt) \implies dW_t \sim \sqrt{dt} \cdot N(0,1).$$

*Now, let us tackle extend Brownian motions to multiple dimensions.*

**Definition B.2** (2-dimensional Wiener process)**.** Let us consider two Brownian motions $W_t^1, W_t^2$, and we consider a multidimensional process $W_t = \{W_t^1, W_t^2\}$. The characterization of a 2-dimensional Wiener process is similar to the one-dimensional case:

1. $W_0^1 = W_0^2 = 0$.
2. For any $s, t \in I \subset \mathbb{R}$ (assuming WLOG $s < t$):

$$W_t^1 - W_s^1 \sim N(0, t-s) \tag{23}$$
$$W_t^2 - W_s^2 \sim N(0, t-s) \tag{24}$$

   and the increments are stationary and independent of the past.

3. There is a concept of correlation coefficient between the components, given by $\rho_{12} = \rho_{21} \in [-1, 1]$ where:

$$\mathbb{E}[(W_t^1 - W_s^1)(W_t^2 - W_s^2)] = \rho_{12}(t-s)$$

   for $s < t$.

*Now, we can extend this definition to dimension $d$, and a $d$-dimensional Wiener processed is modelled based on a multivariate normal distribution and the Cholesky decomposition.*

**Definition B.3** (Cholesky decomposition)**.** Matrix factorization technique that converts a positive definite matrix into the product of a lower triangular matrix and its transpose.

$$M = L \cdot L^T.$$

*We can construct a variance-covariance matrix $C$, associated with the $d$-dimensional Brownian motion. We obtain $L$, with the property that, when multiplied by a vector of independent standard normal variables, we obtain a vector of correlated random variables. This vector $W$ will represent the simulated increments of the Brownian motion for each dimension.*

## Simulate Bachelier & Black-Scholes

*Let us remember the Bachelier model:*

$$dS_t = \mu dt + \sigma dW_t$$

*And the Black-Scholes model:*

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

*We will proceed with the Euler-Maruyama approximation method, and we partition the time inverval $[0, T]$ into $N$ equal sub-intervals of size $\Delta t = T/N$ with timestamps $t_0 = 0 < t_1 = \Delta t < t_2 < \cdots < t_N = T$. Then, we recursively define $S_i = S_{i-1} + \mu\Delta t + \sigma\Delta W_i$.*

*In Python, first we import the necessary libraries for all of the underlying asset simulation models.*

```python
import numpy as np
from scipy.stats import expon, poisson, norm
```

*Now, to simulate the one-dimensional Bachelier model paths, where the number of paths is Nsim:*

```python
def Bach_X_BS(N, Nsim, T, mu, sigma, S0):
    XT = mu * T + sigma * np.sqrt(T) * np.random.randn(Nsim, 1)
    ST = S0 * np.exp(XT)
    dt = T / N
    t = np.linspace(0, T, N+1)
    X = np.zeros((N+1,Nsim))
    X[0,:] = S0*np.ones(Nsim)
    for i in range(1, N+1):
        X[i,:] = X[i-1,:] + mu*dt + sigma*np.sqrt(dt)*np.random.randn(Nsim)
    return X
```

*To simulate the one-dimensional Black-Scholes model paths, where the number of paths is Nsim, we also use the $X_t := \log(S_t/S_0)$ transformation for simplicity:*

```python
def X_BS(N, Nsim, T, mu, sigma, S0):
    XT = mu * T + sigma * np.sqrt(T) * np.random.randn(Nsim, 1)
    ST = S0 * np.exp(XT)
    dt = T / N
    t = np.linspace(0, T, N+1)
    X = np.zeros((N+1,Nsim))
    for i in range(1, N+1):
        X[i,:] = X[i-1,:] + mu*dt + sigma*np.sqrt(dt)*np.random.randn(Nsim)
    return S0*np.exp(X)
```

*Now let us simulate the d-dimensional counterpart of both models. In higher dimensions, we will simulate 1 path of the model and then loop to get the desired number of simulation paths.*

*For the multidimensional Bachelier:*

```
def MD_X_Bach(N, T, mu, sigma, S0):
    time_step = T/N
    size = len(S0)
    rho = sigma[0][1]
    sigma_diag = np.diagonal(sigma)
    gaussian = np.random.normal(size=(N+1,size))
    path = np.zeros((N+1,size))
    path[0] = S0
    drift = (mu*time_step)
    CHOL = np.linalg.cholesky(np.eye(size)+rho*(np.ones((size,size)) -
       np.eye(size)))
    for i in range(1, N+1):
        for d in range(size):
            scale_cholesky_gaussian = np.dot(CHOL[d,:], gaussian[i])
            computed_share = path[i-1, d] + drift[d] +
               np.exp(sigma_diag[d]*np.sqrt(time_step)*scale_cholesky_gaussian)
            path[i,d] = computed_share
    return path
```

*Now, the multidimensional Black-Scholes:*

```
def MD_X_BS(N, T, mu, sigma, S0):
    time_step = T/N
    size = len(S0)
    rho = sigma[0][1]
    sigma_diag = np.diagonal(sigma)
    gaussian = np.random.normal(size=(N+1,size))
    path = np.zeros((N+1,size))
    path[0] = S0
    expo = np.exp(mu*time_step)
    CHOL = np.linalg.cholesky(np.eye(size)+rho*(np.ones((size,size)) -
       np.eye(size)))
    for i in range(1, N+1):
        for d in range(size):
            scale_cholesky_gaussian = np.dot(CHOL[d,:], gaussian[i])
            computed_share = path[i-1,
               d]*expo[d]*np.exp(sigma_diag[d]*np.sqrt(time_step)*scale_cholesky_gauss
            path[i,d] = computed_share
    return path
```

## Adding jumps: Kou and Merton

*For both Kou and Merton, the novelty is the jump simulation.*

*This is our simulation of the 1D Kou model:*

```
def X_BS_Kou(N,Nsim, T,mu,sigma, S0, lambdat, p, lambdap, lambdam):
    dt = T / N
    t = np.linspace(0, T, N+1)
    X = np.zeros((N+1, Nsim))
    NT = poisson.ppf(np.random.rand(Nsim, 1), lambdat * T)
    for j in range(Nsim):
        Tj = np.sort(T * np.random.rand(int(NT[j])))
        Z = np.random.randn(1, N)
        for i in range(1, N+1):
            X[i,j] = X[i-1,j] + mu * dt + sigma * np.sqrt(dt) * Z[0, i-1]
            for k in range(int(NT[j])):
                if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                    uniform_value = np.random.rand(1)
                    if uniform_value < p:
                        Y = expon.ppf(uniform_value, 1/lambdap)
                    else:
                        Y = -expon.ppf(uniform_value, 1/lambdam)
                    X[i,j] = X[i,j] + Y
    return S0*np.exp(X)
```

*And this is our simulation of the 1D Merton model:*

```
def X_BS_Merton(N,Nsim, T,mu,sigma, S0, lambdat, muJ, deltaJ):
    dt = T / N
    t = np.linspace(0, T, N+1)
    X = np.zeros((N+1, Nsim))
    NT = poisson.ppf(np.random.rand(Nsim, 1), lambdat * T)
    for j in range(Nsim):
        Tj = np.sort(T * np.random.rand(int(NT[j])))
        Z = np.random.randn(1, N)
        for i in range(1, N+1):
            X[i,j] = X[i-1,j] + mu * dt + sigma * np.sqrt(dt) * Z[0, i-1]
            for k in range(int(NT[j])):
                if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                    Y = norm.ppf(q = np.random.rand(1), loc= muJ, scale=
                        deltaJ)
                    X[i,j] = X[i,j] + Y
    return S0*np.exp(X)
```

*The extension to dimension d is the following. For MD Kou:*

```
def MD_X_BS_Kou(N, Nsim, T, mu, sigma, S0, lambdat, p, lambdap, lambdam, d):
    dt = T / N
    t = np.linspace(0, T, N+1)
    X = np.zeros((N+1, Nsim, d))
    NT = poisson.ppf(np.random.rand(Nsim, 1), lambdat * T)
    for j in range(Nsim):
        Tj = np.sort(T * np.random.rand(int(NT[j])))
        Z = np.random.randn(N, d)
        for i in range(1, N+1):
            X[i,j, :] = X[i-1,j, :] + mu * dt + sigma * np.sqrt(dt) *
                Z[i-1, :]
            for k in range(int(NT[j])):
                if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                    uniform_value = np.random.rand(1)
                    if uniform_value < p:
                        Y = expon.ppf(uniform_value, 1/lambdap)
                    else:
                        Y = -expon.ppf(uniform_value, 1/lambdam)
                    J = np.zeros(d)
                    for l in range(d):
                        J[l] = Y * np.random.randn(1)
                    X[i,j,:] = X[i,j,:] + J

    return S0*np.exp(X)
```

*And, for MD Merton:*

```
def MD_X_BS_Merton(N, Nsim, T, mu, sigma, S0, lambdat, muJ, deltaJ):
    dt = T / N
    t = np.linspace(0, T, N+1)
    d = len(mu)
    X = np.zeros((N+1, Nsim, d))
    NT = poisson.ppf(np.random.rand(Nsim, 1), lambdat * T)
    for j in range(Nsim):
        Tj = np.sort(T * np.random.rand(int(NT[j])))
        Z = np.random.randn(d, N)
        for i in range(1, N+1):
            X[i,j] = X[i-1,j] + mu * dt + sigma * np.sqrt(dt) * Z[:, i-1]
            for k in range(int(NT[j])):
                if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                    Y = np.random.normal(muJ, deltaJ, d)
                    X[i,j] = X[i,j] + Y
    return S0*np.exp(X)
```

## Stochastic volatility: Heston

*We are going to follow the steps that Leif Andersen [Andersen] makes. We are just going to outline the steps.*

*First, we discretize and simulate $V$; if the Feller condition is satisfied, then one can execute the Quadratic-Exponential Scheme (QE Scheme): For high values of $V(t)$):*

$$V(t+\Delta) = a\cdot(b+Z_V)^2 \quad \text{Quadratic scheme}$$

*where $a$ and $b$ are constants and $Z_V$ is a standard normal variable. For low values of $V(t)$, we supplement the quadratic scheme with the exponential scheme:*

$$Pr(V(t+\Delta)\in[x,x+dx]) \sim (p\delta(0)+\beta(1-p)e^{-\beta x})dx \quad x\geq 0,$$

*where $\delta$ is a Dirac delta-function, and $p$ and $\beta$ are constants. Having $\Psi(x) = Pr(V(t+\Delta)\leq x) = p+(1-p)(1-e^{-\beta x})$ for $x\geq 0$:*

$$V(t+\Delta) = \Psi^{-1}(U_V;p,\beta)$$

*where $U_V$ is a draw from a uniform distribution.*

*Then, we model from $X$ with another discretization scheme, that can be easily derived from the code we provide below.*

*Here is our implementation of the Heston (Andersen) model:*

```python
def Heston(N, Nsim, T, theta, k, epsilon, r, X0, rho, V0):
    V = np.zeros((N+1, Nsim))
    X = np.zeros((N+1, Nsim))
    X[0,:] = X0
    V[0,:] = V0
    Feller_condition = (epsilon**2 - 2*k*theta)
    dt = T / N
    if Feller_condition > 0: # QE Scheme
        Psi_cutoff = 1.5
        for i in range(N): # Discretise V (volatility) and calculate m, Psi, s2
            m = theta + (V[i,:] - theta)*np.exp(-k*dt)
            m2 = m**2
            s2 = V[i,:]*epsilon**2*np.exp(-k*dt)*(1-np.exp(-k*dt))/k + theta*epsilon**2*(1-np.exp(-k*dt))**2/(2*k)
            Psi = (s2)/(m2)
            index = np.where(Psi_cutoff < Psi)[0]

            # Exponential approx scheme if Psi > Psi_cutoff
            p_exp = (Psi[index]-1)/(Psi[index]+1)
            beta_exp = (1-p_exp)/m[index]
            U = np.random.rand(np.size(index))
            V[i+1, index] = (np.log((1-p_exp)/(1-U)/beta_exp)*(U>p_exp))

            # Quadratic approx scheme if 0 < Psi < Psi_cutoff
            index = np.where(Psi <= Psi_cutoff)[0]
            invPsi = 1/Psi[index]
            b2_quad = 2*invPsi - 1 + np.sqrt(2*invPsi)*np.sqrt(2*invPsi-1)
            a_quad  = m[index]/(1+b2_quad)
            V[i+1, index] = a_quad*(np.sqrt(b2_quad)+np.random.randn(np.size(index)))**2

        # Central discretisation scheme
        gamma1 = 0.5
        gamma2 = 0.5
        k0 = r*dt -rho*k*theta*dt/epsilon
        k1 = gamma1*dt*(k*rho/epsilon-0.5)-rho/epsilon
        k2 = gamma2*dt*(k*rho/epsilon-0.5)+rho/epsilon
        k3 = gamma1*dt*(1-rho**2)
        k4 = gamma2*dt*(1-rho**2)
        for i in range(N):
            X[i+1,:] = np.exp(np.log(X[:,i]))+k0+k1*V[i,:]+k2*V[i+1,:]+np.sqrt(k3*V[i,:])+k4*V
        return X, V
    else:
        mu = [0, 0]
        X = np.zeros((N+1, Nsim))
        VC = [[1,rho],[rho,1]]
        for i in range(N):
            Z = np.random.multivariate_normal(mu,VC,Nsim)
            X[i+1,:] = X[i,:] + (r-V[i,:]/2)*dt+np.sqrt(V[i,:]*dt)*Z[:,0]
            V[i+1,:] = V[i,:] +
                k*(theta-V[i,:])*dt+epsilon*np.sqrt(V[i,:]*dt)*Z[:,1]
        return X0*np.exp(X), V
```

And the extension to d-dimensional Heston is the following:

```
def MD_Heston(N, Nsim, T, theta, k, epsilon, r, X0, rho, V0):
    d = len(X0)
    V = np.zeros((Nsim, N, d))
    X = np.zeros((Nsim, N+1, d))
    X[:,:,0] = X0
    V[:,:,0] = V0
    # Feller condition
    Feller_condition = (epsilon**2 - 2*k*theta)
    dt = T / N
    if Feller_condition > 0: # QE Scheme
        Psi_cutoff = 1.5
        for i in range(N): # Discretise V (volatility) and calculate m,
            Psi, s2 for each dimension
            m = theta + (V[:,i,:] - theta)*np.exp(-k*dt)
            m2 = m**2
            s2 = V[:,i,:]*epsilon**2*np.exp(-k*dt)*(1-np.exp(-k*dt))/k +
                theta*epsilon**2*(1-np.exp(-k*dt))**2/(2*k)
            Psi = (s2)/(m2)
            index = np.where(Psi_cutoff < Psi)

            # Exponential approx scheme if Psi > Psi_cutoff
            p_exp = (Psi[index]-1)/(Psi[index]+1)
            beta_exp = (1-p_exp)/m[index]
            U = np.random.rand(np.size(index))
            V[index,i+1,:] = (np.log((1-p_exp)/(1-U)/beta_exp)*(U>p_exp))

            # Quadratic approx scheme if 0 < Psi < Psi_cutoff
            index = np.where(Psi <= Psi_cutoff)
            invPsi = 1/Psi[index]
            b2_quad = 2*invPsi - 1 + np.sqrt(2*invPsi)*np.sqrt(2*invPsi-1)
            a_quad  = m[index]/(1+b2_quad)
            V[index, i+1,:] = a_quad*(np.sqrt(b2_quad)+
                np.random.randn(np.size(index), d))**2

        # Central discretisation scheme
            gamma1 = 0.5
            gamma2 = 0.5
            k0 = r*dt -rho*k*theta*dt/epsilon
            k1 = gamma1*dt*(k*rho/epsilon-0.5)-rho/epsilon
            k2 = gamma2*dt*(k*rho/epsilon-0.5)+rho/epsilon
            k3 = gamma1*dt*(1-rho**2)
            k4 = gamma2*dt*(1-rho**2)
            for i in range(N):
                X[:,i+1] =
                    np.exp(np.log(X[:,i]))+k0+k1*V[:,i]+k2*V[:,i+1]+np.sqrt(k3*V[:,i])+
            return X
    else:
            mu = [0, 0]
            VC = [[1,rho],[rho,1]]
            for i in range(N):
                Z = np.random.multivariate_normal(mu,VC,Nsim)
                X[:,i+1] = X[:,i] +
                    (r-V[:,i]/2)*dt+np.sqrt(V[:,i]*dt)*Z[:,1]
                V[:,i+1] = V[:,i] +
                    k*(theta-V[:,i])*dt+epsilon*np.sqrt(V[:,i]*dt)*Z[:,2]
            return np.exp(X)
```

# C. Deep optimal stopping. Implementation

*Here, we provide the entirety of the code, that both computed MBRCs and Bermudan options. Notice that plenty of improvements and variations can be made, and it is to be interpreted as a vanilla version of a procedure to which add other g's to compute other options or even other kinds of problems.*

Listing 6: QuickDOSwdiffmodels.py

```python

import numpy as np
import torch
import underlying_asset_simulation as ua
import scipy.stats
import matplotlib.pyplot as plt
import time
import torch.utils.data as tdata
import torch.nn as nn
import multiprocessing as mp
import sys
import torch.nn.functional as F
import os
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


def write_file(text):
    f = open("ended_output.txt", "a")
    f.write(text)
    f.close()

def write_table(model_name,N,P,CI_L, CI_U):
    f = open('ended_output_table_'+model_name+'.txt', 'a')
    text = str(N)+';'+str(P)+';'+str(CI_L)+';'+str(CI_U)+'\n'
    f.write(text)
    f.close()
def symmetric_case(sigma, symm, d):
    if(symm):
        # Create a matrix with diagonal vector 'sigma_diag_vec' and
           off-diagonal elements equal to 'rho'
        sigma_diag_vec = sigma * np.ones(d)
        rho = 0
        sigma_matrix = rho * np.ones((d, d))
        np.fill_diagonal(sigma_matrix, sigma_diag_vec)
        return sigma_matrix
    else:
        sigma_diag_vec = np.zeros(d)
        if(d > 5):
            for dimension in range(d):
                sigma_diag_vec[dimension] = 0.1 + (dimension+1)/(2*d)
        else:
            for dimension in range(d):
                sigma_diag_vec[dimension] = 0.08 + 0.32 * (dimension)/(d-1)
        rho = 0
        sigma_matrix = rho * np.ones((d, d))
        np.fill_diagonal(sigma_matrix, sigma_diag_vec)
        return sigma_diag_vec

def g(_t, _x, barrier = None, axis = 1):
    if barrier == []:
        payoff = torch.max(_x, dim=axis, keepdim = True).values
        return torch.exp(-r*_t)*torch.max(payoff-K,torch.tensor(0.))
    elif barrier == "MBRC":
        c = 7/12.
        B = 70
        F = 100
        Breach = torch.zeros(outer_batch_size, N)
        G = torch.zeros(outer_batch_size, 1, N)

        for j in range(outer_batch_size):
            Breach[j, 0] = torch.tensor(0.)
            for n in range(N):
                suma = 0
                H = torch.min(_x[j, :, n])
                if n-1 == 0:
                    Breach[j, n] = torch.where(H > B, torch.tensor([1]),
                        torch.tensor([0]))
                else:
                    Breach[j, n] = torch.max(Breach[j,n-1], torch.where(H >
                        B, torch.tensor([1]), torch.tensor([0])))
                if Breach[j, n] == torch.tensor([0]) or n < N-1:
                    for m in range(n):
                        suma += (torch.exp(-r*_t[m])*c)
                    suma += torch.exp(-r*_t[n])*F
                else:
                    for m in range(n):
                        suma += (torch.exp(-r*_t[m])*c)
                    H = torch.min(_x[j,:,-1])
                    a = torch.tensor([K])
                    if H > a:
                        H = K
                    suma += (torch.exp(-r*_t[-1]) * H)
                G[j,0,n] = suma
        return G
def sample_paths(bsize, _d, x_0, rho, ua_model = 'BS', symm = True):

    if ua_model == 'BarrierBS':
        Ti = 1/2
        Tb = 1
        deltaib = 0.05
        rb = 0
        mu = rb- sigma**2/2
        time_step = dt = Tb/N
        gaussian = np.random.normal(size=(N,d))
        CHOL = np.linalg.cholesky(np.eye(d)+rho*(np.ones((d,d)) -
            np.eye(d)))
        pre_wi = torch.zeros(size = (bsize, d, N))

        for i in range(N):
            for j in range(bsize):
                scale_cholesky_gaussian = sigma*np.sqrt(dt)*
                   np.dot(CHOL,gaussian[i])
                scale_cholesky_gaussian =
                   torch.from_numpy(scale_cholesky_gaussian)
                if(i*dt < Ti):
                    for k in range(d):
                        pre_wi[j,k,i] = pre_wi[j,k,i] +
                           scale_cholesky_gaussian[k]
                else:
                    for k in range(d):
                        pre_wi[j,k,i] = pre_wi[j,k,i] +
                           scale_cholesky_gaussian[k]



        _wi = torch.cumsum(pre_wi[:,:,:N//2], dim = 2)
        #_wi = torch.cumsum(pre_wi[:,:,:int(np.floor(N/2))], dim = 2)
        _ti = torch.linspace(Tb/N, Ti, N//2)
        _wo = torch.cumsum(pre_wi[:,:,N//2:], dim = 2)
        #_wo = torch.cumsum(pre_wi[:,:,int(np.ceil(N/2)):], dim = 2)
        _to = torch.linspace(Ti, Tb, N//2)
        _t = torch.linspace(Tb/N, Tb, N)


        _xi = torch.exp((rb - sigma ** 2 / 2.) * _ti + _wi) * x_0
        _xo = torch.exp((rb - sigma ** 2 / 2.) * _to + _wo) *
           x_0*(1-deltaib)
        _x = torch.cat((_xi, _xo), dim = 2)

        return _x, _t

    if ua_model == 'BS':
        if symm == False:
            time_step = T/N
            pre_w = torch.zeros(size = (bsize, d, N))
            symm_sigma = symmetric_case(sigma, symm,_d)
            gaussian = np.random.normal(size=(N+1,_d))
            path = torch.zeros(bsize, _d, N+1)
            mu = torch.zeros(_d)
            for i in range(_d):
                mu[i] = (r - delta - symm_sigma[i] ** 2 / 2.)

            CHOL = np.linalg.cholesky(np.eye(_d)+rho*(np.ones((_d,_d)) -
               np.eye(_d)))
            for j in range(bsize):
                for i in range(N):
                    for z in range(_d):
                        scale_cholesky_gaussian =
                                symm_sigma[z]*np.sqrt(time_step)*
                                np.dot(CHOL[z,:],gaussian[i])
                            pre_w[j,z,i] = pre_w[j,z,i] +
                                scale_cholesky_gaussian
            _w = torch.cumsum(pre_w, dim = 2)
            _t = torch.linspace(T/N, T, N)
            for j in range(bsize):
                for i in range(1,N):
                    for z in range(_d):
                        if i == 1:
                            path[j,z,i] = x_0*torch.exp(mu[z]*time_step+
                                pre_w[j,z,i])
                        else:
                            path[j,z,i] =
                                path[j,z,i-1]*torch.exp(mu[z]*time_step+
                                pre_w[j,z,i])

            _x = path[:,:,1:]
            return _x, _t
        else:
            pre_w = torch.normal(mean = 0.0, std = sigma*np.sqrt(T/N),
                size=(bsize, d, N))
            _w = torch.cumsum(pre_w, dim = 2)
            _t = torch.linspace(T/N, T, N)
            _x = torch.exp((r - delta - sigma ** 2 / 2.) * _t + _w) * x_0

            return _x, _t
    elif ua_model == 'Bach':
        pre_w = torch.normal(mean = 0.0, std = sigma*np.sqrt(T/N),
            size=(bsize, d, N))

        _w = torch.cumsum(pre_w, dim = 2)
        _t = torch.linspace(T/N, T, N)
        _x = x_0 + ((r - delta - sigma ** 2 / 2.) * _t + _w)
        return _x, _t
    elif ua_model == 'Kou':
        dt = T / N
        lambdat = 0.08
        p = 0.4
        lambdam = lambdap = 1000
        _t = torch.linspace(T/N, T, N)
        mu = (r - delta - sigma ** 2 / 2.)
        t = np.linspace(0, T, N+1)
        X = np.zeros((bsize, d, N+1))
        NT = scipy.stats.poisson.ppf(np.random.rand(bsize, 1), lambdat * T)
        for j in range(bsize):
            Tj = np.sort(T * np.random.rand(int(NT[j])))
            Z = np.random.randn(N, d)
            for i in range(1, N+1):
                X[j,:,i] = X[j,:,i-1] + mu * dt + sigma * np.sqrt(dt) *
                    Z[i-1, :]
                for k in range(int(NT[j])):
                    if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                        uniform_value = np.random.rand(1)
                        if uniform_value < p:
                            Y = scipy.stats.expon.ppf(uniform_value, 1/lambdap)
                        else:
                            Y = -scipy.stats.expon.ppf(uniform_value, 1/lambdam)
                        J = np.zeros(d)
                        for l in range(d):
                            J[l] = Y * np.random.randn(1)
                        X[j,:,i ] = X[j,:,i ] + J
        X = X[:,:,1:]
        X = torch.from_numpy(X)
        return x_0*torch.exp(X), _t
    elif ua_model == 'Merton':
        pre_w = torch.normal(mean = 0.0, std = sigma*np.sqrt(T/N), size=(bsize, d, N))
        lambdat, muJ, deltaJ = 3, 0, 0.25
        dt = T / N
        lambdat = 0.08
        p = 0.4
        _t = torch.linspace(T/N, T, N)
        mu = (r - delta - sigma ** 2 / 2.)
        t = np.linspace(0, T, N+1)
        X = np.zeros((bsize, d, N+1))
        NT = scipy.stats.poisson.ppf(np.random.rand(bsize, 1), lambdat * T)
        for j in range(bsize):
            Tj = np.sort(T * np.random.rand(int(NT[j])))
            Z = np.random.randn(N, d)
            for i in range(1, N+1):
                X[j,:,i] = X[j,:,i-1] + mu * dt + sigma * np.sqrt(dt) * Z[i-1, :]
                for k in range(int(NT[j])):
                    if (int(Tj[k]) > int(t[i-1])) & (int(Tj[k]) <= int(t[i])):
                        Y = np.random.normal(muJ, deltaJ, d)
                        X[j,:,i] = X[j,:,i] + Y
        X = X[:,:,1:]
        X = torch.from_numpy(X)
        return x_0*torch.exp(X), _t
    elif ua_model == 'Heston':
        dt = T/N
        _t = torch.linspace(T/N, T, N)
        time_step = T / N
        size = d
        epsilon = 0.05
        k = 3
        theta = 0.2
        rho = -0.2

```

```
V = np.zeros((bsize, N+1, d))
X = np.zeros((bsize, N+1, d))
X[:,:,0] = x_0
V[:,:,0] = 0.2

Feller_condition = (epsilon**2 - 2*k*theta)
dt = T / N

if Feller_condition > 0: # QE Scheme
    Psi_cutoff = 1.5
    for i in range(N): # Discretise V (volatility) and calculate m, Psi, s2 for each dimension
        m = theta + (V[:,i,:] - theta)*np.exp(-k*dt)
        m2 = m**2
        s2 = V[:,i,:]*epsilon**2*np.exp(-k*dt)*(1-np.exp(-k*dt))/k + theta*epsilon**2*(1-np.exp(-k*dt))**2/(2*k)
        Psi = (s2)/(m2)
        index = np.where(Psi_cutoff < Psi)

        # Exponential approx scheme if Psi > Psi_cutoff
        p_exp = (Psi[index]-1)/(Psi[index]+1)
        beta_exp = (1-p_exp)/m[index]
        U = np.random.rand(np.size(index))
        V[index,i+1,:] = (np.log((1-p_exp)/(1-U)/beta_exp)*(U>p_exp))

        # Quadratic approx scheme if 0 < Psi < Psi_cutoff
        index = np.where(Psi <= Psi_cutoff)
        invPsi = 1/Psi[index]
        b2_quad = 2*invPsi - 1 + np.sqrt(2*invPsi)*np.sqrt(2*invPsi-1)
        a_quad  = m[index]/(1+b2_quad)
        V[index, i+1,:] = a_quad*(np.sqrt(b2_quad)+ np.random.randn(np.size(index), d))**2

    # Central discretisation scheme
        gamma1 = 0.5
        gamma2 = 0.5
        k0 = r*dt -rho*k*theta*dt/epsilon
        k1 = gamma1*dt*(k*rho/epsilon-0.5)-rho/epsilon
        k2 = gamma2*dt*(k*rho/epsilon-0.5)+rho/epsilon
        k3 = gamma1*dt*(1-rho**2)
        k4 = gamma2*dt*(1-rho**2)
        for j in range(bsize):
            for i in range(N):
                    X[j,i+1,:] = np.exp(np.log(X[j,i,:]))+k0+k1*V[j,i,:]+k2*V[j,i+1,:]+r
        X = X[:,1:,:]
        X = X.permute(0, 2, 1)
        return X, _t
else:
        mu = [0, 0]
```

```
                VC = [[1,rho],[rho,1]]
                for i in range(N):
                    Z = np.random.multivariate_normal(mu,VC,bsize)
                    X[:,i+1,:] = X[:,i,:] +
                       (r-V[:,i,:]/2)*dt+np.sqrt(V[:,i,:]*dt)*Z[:,0]
                    V[:,i+1] = V[:,i] +
                       k*(theta-V[:,i])*dt+epsilon*np.sqrt(V[:,i]*dt)*Z[:,1]
                X = X[:, 1:]
                X = X.permute(0, 2, 1)
                return np.exp(X)

class Neural_Net_NN(torch.nn.Module):
    def __init__(self, M,shape_input):
        super(Neural_Net_NN, self).__init__()
        self.dense1 = nn.Linear(shape_input,M)
        self.dense2 = nn.Linear(M,1)
        self.bachnorm1 = nn.BatchNorm1d(M)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.bachnorm1(self.relu(self.dense1(x.float())))
        x = self.relu(self.dense2(x))
        return x

def train_and_price(x, p, n, batch_size, num_neurons, train_steps, mc_runs,
                    lr_boundaries, path, barrier, epsilon=0.1,
                       dtype=torch.float32):

    neural_net = Neural_Net_NN(num_neurons,d+1).to(device)
    optimizer = torch.optim.Adam(neural_net.parameters(), lr=0.05)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,
       milestones=lr_boundaries, gamma=0.1)
    px_hist = []
    print('\n\n Training phase:\n\n')
    max_px = 0
    for train_step in range(0,training_steps+1):
        X,p_,g_tau = x, p, p[:,:,n-1]
        X,p_,g_tau = X.to(device), p_.to(device),g_tau.to(device)
        state = torch.cat((X,p_),axis = 1)
        loss = np.zeros(1)
        loss = torch.tensor(loss).to(device)
        for n in range(N-2, -1, -1):
            net_n = neural_net(state[:,:,n])
            F_n   = torch.sigmoid(net_n)
            if barrier != []:
                loss = torch.mean(p_[:, :, n] * F_n + g_tau * (1. - F_n))
            loss -= torch.mean(p_[:, :, n] * F_n + g_tau * (1. - F_n))
            g_tau = torch.where(net_n > 0, p_[:, :, n], g_tau)

        px_mean_batch = torch.mean(g_tau)
        loss = torch.mean(loss)
        px_hist.append(px_mean_batch.item())

        if train_step > 10:
            if px_mean_batch.item()>max_px and px_mean_batch.item()<np.mean(px_hist[-10:-1])*1.50:
                torch.save(neural_net.state_dict(), 'best_model.pt')
                max_px = px_mean_batch.item()


        if train_step%100 == 0:
            print('| Train step: {:5.0f} | Loss: {:3.3f} | V: {:3.3f} | Lr: {:1.6f} |'.format(train_step,loss.item(),px_mean_batch.item(),optimizer.param_g
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

    print('\n\n Evaluation phase:\n\n')
    px_vec = []
    px_var_vec = []
    neural_net = Neural_Net_NN(num_neurons,d+1).to(device)
    neural_net.load_state_dict(torch.load('best_model.pt'))
    neural_net.eval()


    for mc_step in range(0,mc_runs+1):
        X,p_,g_tau = x, p, p[:,:,n-1]
        X,p_,g_tau = X.to(device), p_.to(device),g_tau.to(device)
        state = torch.cat((X,p_),axis = 1)
        for n in range(N-2, -1, -1): # loop from T-T/N to T/N
            net_n = neural_net(state[:,:,n])
            g_tau = torch.where(net_n > 0, p_[:, :, n], g_tau)

        px_mean_batch = torch.mean(g_tau)
        px_var_batch = torch.var(g_tau)

        if px_mean_batch.item() < torch.mean(torch.max(p_,2)[0]).item():
            px_vec.append(px_mean_batch)
            px_var_vec.append(px_var_batch)
        if mc_step%100 == 0:
            print('| MC run step: {:5.0f} | V: {:3.3f} |'.format(mc_step,px_mean_batch.item()))

    if px_vec == []:
        px_mean = torch.tensor([0])
        px_std = torch.tensor([0])
    else:
        px_mean = torch.mean(torch.stack(px_vec))
        px_std = torch.std(torch.stack(px_vec))
    np_px_mean = px_mean.detach().numpy()

    p_v = np.sum(np.array(px_var_vec)) / mc_runs + np.var(np_px_mean)
    tmp = scipy.stats.norm.ppf(0.975) * np.sqrt(p_v / (mc_runs * batch_size - 1.))
    return px_mean.item(), px_mean.item() - tmp, px_mean.item()+tmp
    del neural_net
    torch.cuda.empty_cache()

path = os.getcwd()
K,T,N,r,delta,sigma,batch_size, = 100,3,9,0.05,0.1,0.2,8192
#N = 12
#ua_model = 'BarrierBS'
ua_model = 'BS'
runs = 3
#barrier = 'MBRC'
barrier = []
symm = True

for d in [2, 3, 5, 10, 20, 30, 50, 100, 200, 500]:
    num_neurons = d+40
    training_steps, mc_runs_px = (3000+d), 500
    lr_boundaries = [500 + d // 5, 1500 + 3 * d // 5]
    if d <= 30:
        bsize = batch_size // 512 * 8
    elif d <= 100:
        bsize = batch_size // 2048 * 8
    elif d <= 200:
        bsize = batch_size // 4096 * 8
    else:
        bsize = batch_size // 8192 * 8
    if barrier == []:
        for s_0 in [90., 100., 110.]:
            V_L_S = 0
            C_L_S = 0
            C_U_S = 0
            print(f'For d = {d}, S0 = {s_0}:')
            t0 = time.time()
            for _ in range(runs):

                X, t = sample_paths(bsize, d, s_0, 0, ua_model, symm)

                V_L, C_L, C_U = train_and_price(X, g(t,X, barrier), N, bsize, num_neurons, training_steps, mc_runs_px,
                                        lr_boundaries, path, barrier, epsilon=0.1, dtype=torch.float32)

                V_L_S += V_L
                C_L_S += C_L
                C_U_S += C_U
                text = '\n Params: N = '+str(N)+' | d = '+str(d)+' | V = '+str(V_L)+' CI_L = '+str(C_L)+' C_I_U = '+str(C_U)
            t1 = time.time()
            write_file('\n For d = {:5.0f}, S0 = {:3.3f}:\n'.format(d, s_0))
            text = '\n Params: time = '+str(int(t1-t0))+' seconds' + 'N =
                 '+str(N)+' | d = '+str(d)+' | V = '+ str(V_L_S/runs)+' CI_L
                 = '+str(C_L_S/runs)+' C_I_U = '+str(C_U_S/runs)
            write_file(text)
            write_table(ua_model, N,V_L_S/runs,C_L_S/runs, C_U_S/runs)
    else:
        s_0 = 100.
        for rho in [0.6, 0.1]:
            V_L_S = 0
            C_L_S = 0
            C_U_S = 0
            print(f'For d = {d}, rho = {rho}:')
            t0 = time.time()
            for _ in range(runs):
                X, t = sample_paths(bsize, d, s_0, rho, ua_model =
                    "BarrierBS", symm = False)
                V_L, C_L, C_U = train_and_price(X, g(t,X, barrier), N,
                    bsize, num_neurons, training_steps, mc_runs_px,
                                     lr_boundaries, path, barrier,
                                        epsilon=0.1, dtype=torch.float32)
                V_L_S += V_L
                C_L_S += C_L
                C_U_S += C_U
                text = '\n Params: N = '+str(N)+' | d = '+str(d)+' | V =
                    '+ str(V_L)+' CI_L = '+str(C_L)+' C_I_U = '+str(C_U)

            t1 = time.time()
            write_file('\n For d = {:5.0f}, rho = {:3.3f}:\n'.format(d,
               rho))
            text = '\n Params: time = '+str(int(t1-t0))+' seconds' + 'N =
                 '+str(N)+' | d = '+str(d)+' | V = '+ str(V_L_S/runs)+' CI_L
                 = '+str(C_L_S/runs)+' C_I_U = '+str(C_U_S/runs)
            write_file(text)
            write_table(ua_model, N,V_L_S/runs,C_L_S/runs, C_U_S/runs)
```