UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2023:1

Degree project 30 credits

Master`s Programme in Mathematics

February 2023

Pricing and Hedging American-Style Options with Deep Learning: Algorithmic implementation

Mohammed Moniruzzaman Khan

Department of Mathematics, Uppsala University

Supervisor: Kaj Nyström

Examiner: Jörgen Östensson

Abstract

This thesis aims at evaluating and implementing Longstaff & Schwarz approach for approximating the value of American options. American options are generally hard to value, exercised at any time up to its expiration and moreover, there is no closed-form solution for an American option's price.

The proposed algorithm permits to estimate, starting from the data, a candidate optimal stopping strategy, lower and upper bounds for the value of the option and confidence intervals for these estimates. The computed lower and upper bounds are used to estimate a point for the value of the option. Finally, we have generated a set of realizations of a geometric Brownian motion (GBM) to simulate the price of an asset and the Deep Learning method used for training Neural Networks for the determination of optimal stopping strategy is presented.

We tackled the question of precision and to estimate complexity of the algorithms based on these criteria.

Contents

Chapter 1

Introduction

American options are common instruments in today's markets and in this article, we consider American options in discrete time. Despite significant advances, the optimal stopping problem is found in areas of statistics, economics, and in financial mathematics to finding the optimal time to stop in order to maximize an expected reward. However, it remains one of the most challenging problems in optimization, in particular when more than one factor affects the expected reward. As indicated in Becker et al. [2020], early exercise options are notoriously difficult to value. It is further stated that for up to three underlying risk factors, Tree-based and classical PDE approximation methods usually produce good numerical results. In the case of higher-dimensional problems, several simulation-based methods have been developed; see Longstaff and Schwartz [2001] and the references in Becker et al. [2020]. Other techniques such as artificial neural networks to estimate continuation values have also been used; see Kohler et al. [2010]. In the last four years, continuous-time optimal stopping problems have been solved by approximating the solutions of the corresponding free boundary PDEs with deep neural networks and also a deep learning has been used to learn directly stop strategies (see Sirignano and Spiliopoulos [2018] and Becker et al. [2019] and Becker et al. [2021]).

In this thesis, we present a variant of the Longstaff–Schwartz algorithm as in Becker et al. [2020]. More specifically, let $X = (X_n)_{n=0}^N$ be an $\mathbb{R}^d$-valued discrete-time Markov process on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where N and d are positive integers. $X = (X_n)_{n=0}^N$ is the underlying stochastic process describing e.g. the prices of the underlying and the financial environment (like interest rates, etc.). We denote by $\mathcal{F}_n$ the σ-algebra generated by $X_0, X_1, \ldots, X_n$ and the filtration $\mathbb{F} = \{\mathcal{F}_n\}_{n=0}^N$. Next, we show a deep learning method developed by Becker et al. [2020] that can

efficiently learn an optimal policy for stopping problems of the form

$$\sup_{\tau \in \mathcal{T}} \mathbb{E} g\left(\tau, X_\tau\right),$$

and

$$\operatorname{ess\,sup}_{\tau \in \mathcal{T}_n} \mathbb{E}\left[g\left(\tau, X_\tau\right) \mid \mathcal{F}_n\right]$$

where $g : \{0, 1, \ldots, N\} \times \mathbb{R}^d \to \mathbb{R}$ is a measurable function and $\mathcal{T}$ denotes the set of all $\mathbb{F}$-stopping times $\tau : \Omega \to \{0, 1, \cdots, N\}$ and $\mathcal{T}_n$ is the set of all $\mathbb{F}$-stopping times satisfying $n \leq \tau \leq N$.

In theory, optimal stopping problems with finitely many stopping opportunities can be solved exactly. Therefore, we use the previous stochastic framework to model an American-style option that can be exercised at any one of finitely many times $0 = t_0 < t_1 < \cdots < t_N = T$. The model is as follows, if exercised at time t_n, it yields a discounted payoff given by a square-integrable random variable G_n defined on a filtered probability space $\left(\Omega, \mathcal{F}, \mathbb{F} = (\mathcal{F}_n)_{n=0}^N, \mathbb{P}\right)$ where $\mathcal{F}_n$ describes the information available at time t_n and G_n is of the form $g\left(n, X_n\right)$ for a measurable function $g : \{0, 1, \ldots, N\} \times \mathbb{R}^d \to [0, \infty)$ and the underlying stochastic process $X = (X_n)_{n=0}^N$ is a Markov process.

For the Markov process $(X_n)_{n=0}^N$ with X_0 to be deterministic and $\mathbb{P}$ to be the pricing measure, the value of the option at time 0 is given by

$$V = \sup_{\tau \in \mathcal{T}} \mathbb{E} G_\tau = \sup_{\tau \in \mathcal{T}} \mathbb{E} g(\tau, X_\tau). \tag{1.1}$$

If the option has not been exercised before time t_n, its discounted value at that time is

$$V_{t_n} = \operatorname{ess\,sup}_{\tau \in \mathcal{T}_n} \mathbb{E}\left[G_\tau \mid \mathcal{F}_n\right] = \operatorname{ess\,sup}_{\tau \in \mathcal{T}_n} \mathbb{E}\left[g(\tau, X_\tau \mid \mathcal{F}_n\right] \tag{1.2}$$

where $\mathcal{T}_n$ is the set of all $\mathbb{F}$-stopping times satisfying $n \leq \tau \leq N$.

To understand the intuition behind this approach, recall that at any exercise time, the holder of an American option optimally compares the payoff from immediate exercise with the expected payoff from continuation, and then exercises if the

immediate payoff is higher. Thus the optimal exercise strategy is fundamentally determined by the conditional expectation of the payoff from continuing to keep the option alive. The key insight underlying the approach of Becker et al. [2020] is that this conditional expectation can be using a variant of the Longstaff-Schwartz algorithm.

Now based on the following result (see Bru and Heinich [1985]): $\mathbb{E}\left[G_{\tau_{n+1}} \mid X_n\right]$ is of the form $c\left(X_n\right)$, where $c : \mathbb{R}^d \to \mathbb{R}$ minimizes the mean squared distance $\mathbb{E}\left[\left\{G_{\tau_{n+1}} - c\left(X_n\right)\right\}^2\right]$ over all Borel measurable functions from $\mathbb{R}^d$ to $\mathbb{R}$, we can use the Longstaff-Schwartz algorithm to approximates $\mathbb{E}\left[G_{\tau_{n+1}} \mid X_n\right]$ by projecting $G_{\tau_{n+1}}$ on the linear span of finitely many basis functions. Here is where the deep learning methodology comes in. Next, we describe our neural network version of the Longstaff–Schwartz algorithm to estimate continuation values and construct a candidate optimal stopping strategy. For that, let $t_n = nT/N$ for $n = 0, 1, \ldots, N$.

(i) Simulate $\left\{s_n^k\right\}_{n=0}^N$, $k = 1, \cdots, M$ of the underlying $\{S_n\}_{n=0}^N$.

(ii) $l_N^k \equiv N$ for all $k = 1, \ldots, M$.

(iii) Evaluate for $k = 1, \ldots, M$, $Y_N^k = g\left(l_N^k, s_{l_N^k}^k\right)$.

For $1 \leq n \leq N-1$,

(iv) Approximate $E\left[G_{\tau_{n+1}} \mid S_n\right]$ with $c^{\theta_n}\left(S_n\right)$ by minimizing the sum

$$J(\theta) = \frac{1}{M}\sum_{k=1}^{M}\left(Y_{n+1}^k - c^{\theta}\left(s_n^k\right)\right)^2 \text{ over } \theta.$$

and evaluate for $k = 1, \ldots, M$, $Y_{n+1}^k = g\left(l_{n+1}^k, s_{l_{n+1}^k}^k\right)$

(v) Set

$$l_n^k := \begin{cases} n & \text{if } g\left(n, s_n^k\right) \geq c^{\theta_n}\left(s_n^k\right) \\ l_{n+1}^k & \text{otherwise} \end{cases}$$

(vi) Define $\theta_0 := \frac{1}{M}\sum_{k=1}^{M} Y_{l_1^k}^k$, and set c^{θ_0} constantly equal to θ_0.

In this way we specify the neural network given by c^{Θ}.

To illustrate how this procedure works we consider a geometric Brownian motion as the underlying asset for the case $d = 1$.

The rest of the thesis is organized as follows: In Chapter 2, we consider an Introduction to Option Pricing. In Chapter 3, we describe the method of the Deep Neural Learning. In Chapter4, we rewrite the result of Becker et al. [2020] of Pricing and Hedging American-Style Options. In Chapter 5, we illustrate how the procedure of deep learning works considering a geometric Brownian motion as the underlying asset for the case $d = 1$. In Chapter 6, Conclusion.

Chapter 2

Introduction to Option Pricing

In this chapter, we borrow the work of Gustafsson [2015] for an introduction of option pricing. Options are one of the most common financial derivatives used on financial markets. They are contracts giving the buyer of the option the right, but not the obligation, to buy or sell an underlying asset to a specified strike price. Options can be of two types, put or call. A put option gives the owner the right to sell the underlying asset at the agreed upon strike price, and a call option the right to buy the asset for the strike price.

There are several different kinds of options, where the most common are called European and American options. The difference is that a European option only gives the owner the right to exercise at a specific time in the future, whereas an American option can be exercised at any time up until the expiration time.

The value of the option over time can be described as a partial differential equation, called the Black-Scholes equation, which is

$$\frac{\partial u}{\partial t} + rS\frac{\partial u}{S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - ru = 0 \tag{2.1}$$

where $S = S(t)$ is the price of the underlying asset, $u = u(S,t)$ is the value of the option, r is the risk-free interest rate, σ is the volatility of the underlying asset, K is the strike price and $t \in [0,T]$. Also, $t = 0$ denotes the present time and $t = T$ denotes the expiration time.

If u satisfies the equation given by (2.1) with the boundary condition

$$u(S,T) = \max(K - S, 0)$$

which corresponds to the special case of a European put option, where $\max(K-S,0)$ is the payoff of a put option, the Black-Scholes equation has an explicit solution.

For American options, the Black-Scholes equation becomes

$$\frac{\partial u}{\partial t} + rS\frac{\partial u}{S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - ru \leq 0$$

with the boundary conditions

$$u(S,T) = \max(K-S,0) \text{ and } u(S,t) \geq \max(K-S,0)$$

for an American put. This equation does not have an analytic solution, since the option can be exercised at any time up until expiration. The problem then becomes to find the **optimal time** to exercise, that is the time when the payoff of immediate exercise is greater than the expected reward of keeping the option. So that a number of numerical methods have been developed, such as finite-difference methods, binomial tree models, Monte Carlo methods, among others.

Geometric Brownian motion

One of the essential assumptions of the Black-Scholes model is that the underlying asset, most commonly a stock, is modelled as a geometric Brownian motion.

First, we need to define a standard Brownian motion, or Wiener process.

Definition 2.1. *A random process $\{W(t)\}$ is said to be a standard Brownian motion on $[0,T]$ if it satisfies:*

1. $W(0) = 0$.
2. $\{W(t)\}$ *has independent and stationary increments.*
3. $W(t) - W(s) \sim \mathcal{N}(0, t-s)$ *for any* $0 \leq s \leq t \leq T$.
4. $\{W(t)\}$ *has almost surely continuous trajectories.*

From this definition it follows that

$$W(t) \sim \mathcal{N}(0,t) \text{ for } 0 < t \leq T$$

which will be important when simulating the Brownian motion. Next we need to define a Brownian motion with drift and diffusion coefficient.

Definition 2.2. *A process* $\{X(t)\}$ *is said to be a Brownian motion with drift* $\mu > 0$ *and diffusion coefficient* $\sigma^2 > 0$ *if*

$$\frac{X(t) - \mu t}{\sigma}$$

is a standard Brownian motion.

This means that

$$X(t) \sim \mathcal{N}\left(\mu t, \sigma^2 t\right) \text{ for } 0 < t \leq T$$

and that we can construct the process $\{X(t)\}$ using a standard Brownian motion $\{W(t)\}$ by putting

$$X(t) = \mu t + \sigma W(t) \tag{2.2}$$

Next, we consider the stochastic differential equation (SDE)

$$dX(t) = \mu t + \sigma dW(t). \tag{2.3}$$

To solve this SDE we need to use some Itô calculus, which is a generalised calculus for stochastic processes (see Oksendal [2013]).

The process $\{X(t)\}$ given in (2.2) solves the above stochastic differential equation (SDE).

Remark 2.3. *A Brownian motion is a continuous stochastic process, and as such it cannot be modelled exactly but has to be discretized for a finite number of time steps* $t_0 < t_1 < \ldots < t_N$. *Let* $t_0 = 0, X(0) = 0$, *and* $Z_1, \ldots, Z_N$ *be a set of i.i.d.* $\mathcal{N}(0,1)$ *random variables. Then the process* $\{X(t)\}$ *can be simulated* as

$$X\left(t_{i+1}\right) = X\left(t_i\right) + \mu\left(t_{i+1} - t_i\right) + \sigma\sqrt{t_{i+1} - t_i} Z_{i+1} \tag{2.4}$$

for $i = 0, \ldots, N-1$.

A regular Brownian motion cannot be used as a model for an asset price, since it can assume negative values. For this we need the geometric Brownian motion (GBM), which is essentially an exponentiated Brownian motion.

Definition 2.4. *A process* $\{S(t)\}$ *is said to be a geometric Brownian motion if it solves the stochastic differential equation*

$$dS(t) = \mu S(t) dt + \sigma S(t) dW(t). \tag{2.5}$$

The next theorem shows the solution of the equation defined above.

Theorem 2.5. *We consider the process defined by*

$$S(t_1) = S(t_0)\, \mathrm{e}^{\left(\mu - \frac{1}{2}\sigma^2\right)(t_1 - t_0) + \sigma(W(t_1) - W(t_0))} \tag{2.6}$$

then is solution of the stochastic differential equation given by (2.5).

Remark 2.6. *Since $W(t)$ is a standard Brownian motion, equation (2.6) gives a straight forward way to simulate the GBM $S(t)$ analogue to that of a regular Brownian motion in equation (2.6):*

$$S(t_{i+1}) = S(t_i)\, \mathrm{e}^{\left(\mu - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i} Z_{i+1}}$$

for $i = 0, \ldots, N-1, Z_1, \ldots, Z_N$ i.i.d. $\mathcal{N}(0,1)$ random variables and $S(t_0) = S(0)$ some initial value.

When using a GBM to simulate the price of an asset, the drift μ is often denoted r and represents the risk-free interest rate, as in the Black-Scholes model.

2.1 Longstaff-Schwartz method

A common algorithm for pricing American options is the Longstaff-Schwartz method, called LSM algorithm. The LSM algorithm uses Monte Carlo (MC) to estimate the value of the option.

Monte Carlo

To illustrate this part we present an excellent introduction given in Ross [2022].

One of the earliest applications of random numbers was in the computation of integrals. Let $f(x)$ be a function and suppose we wanted to compute μ where

$$\mu = \int_0^1 f(x)dx$$

To compute the value of μ, note that if U is uniformly distributed over $(0,1)$, then we can express μ as

$$\mu = E[f(U)]$$

If $U_1, \ldots, U_k$ are independent uniform $(0,1)$ random variables, it thus follows that the random variables $f(U_1), \ldots, f(U_k)$ are independent and identically distributed

random variables having mean μ. Therefore, by the strong law of large numbers, it follows that, with probability 1 ,

$$\sum_{k=1}^{k} \frac{f\left(U_i\right)}{k} \rightarrow E[f(U)] = \mu \quad \text{as } k \rightarrow \infty$$

Hence we can approximate μ by generating a large number of random numbers u_i and taking as our approximation the average value of $f\left(u_i\right)$. This approach to approximating integrals is called the Monte Carlo approach. If we wanted to compute

$$\mu = \int_a^b f(x)dx$$

then, by making the substitution $y = (x-a)/(b-a), dy = dx/(b-a)$, we see that

$$\begin{aligned} \mu &= \int_0^1 f(a + [b-a]y(b-a)dy \\ &= \int_0^1 h(y)dy \end{aligned}$$

where $h(y) = (b-a)f(a + [b-a]y)$. Thus, we can approximate μ by continually generating random numbers and then taking the average value of h evaluated at these random numbers. Similarly, if we wanted

$$\mu = \int_0^\infty f(x)dx$$

we could apply the substitution $y = 1/(x+1), dy = -dx/(x+1)^2 = -y^2dx$, to obtain the identity

$$\mu = \int_0^1 h(y)dy$$

where

$$h(y) = \frac{f\left(\frac{1}{y} - 1\right)}{y^2}.$$

The LSM algorithm

The problem with pricing American options is that they can be exercised at all times up until the expiration of the option, unlike a European option that can only be exercised at the expiration time. Let

$$\begin{aligned} g(S(t)) &= \max(K - S(t), 0) \quad \text{for a put option, and} \\ g(S(t)) &= \max(S(t) - K, 0) \quad \text{for a call option} \end{aligned}$$

be the payoff at time t, and assume that the underlying asset is modelled using the GBM

$$S(t) = S(0)\mathrm{e}^{\left(r-\frac{1}{2}\sigma^2\right)t+\sigma W(t)}.$$

Using the assumptions above, the value at time 0 of a European option can be described as

$$u(S,0) = \mathrm{E}\left[\mathrm{e}^{-rT}g(S(T))\right]$$

that is, the expected value of the discounted payoff at time T. In a similar way the value of an American option at time 0 is given by

$$u(S,0) = \sup_{t\in[0,T]} \mathrm{E}\left[\mathrm{e}^{-rt}g(S(t))\right] \tag{2.7}$$

the expected value of the discounted payoff at the time of exercise that yields the greatest payoff. The reason for this is the assumption of no arbitrage, that is the chance for risk-free reward, which means the option must cost as much as the maximum expected reward from it. This corresponds to the optimization problem of finding the optimal stopping time

$$t^* = \inf\left\{t \geq 0 \mid S(t) \leq b^*(t)\right\}$$

for some unknown exercise boundary b^*, as illustrated in Figure 2.1.

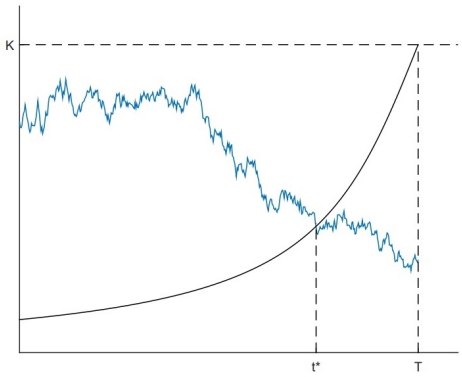

Figure 2.1: Underlying process

So in order to price an American option we need to find the optimal stopping time t^*, and then estimate the expected value

$$u(S,0) = \mathrm{E}\left[\mathrm{e}^{-rt^*}g\left(S\left(t^*\right)\right)\right]$$

The LSM method, developed by Longstaff and Schwartz in Longstaff and Schwartz [2001], uses a dynamic programming approach to find the optimal stopping time, and Monte Carlo to approximate the expected value. Dynamic programming is a general method for solving optimization problems by dividing it into smaller sub problems and combining their solution to solve the problem. In this case, this means that we divide the interval $[0, T]$ into a finite set of time points $\{0, t_1, t_2, \ldots, t_N\}, t_N = T$, and for each of these decide if it is better to exercise than to hold on to the option. Starting from time T and working backwards to time 0 , we update the stopping time each time and we find a time where it is better to exercise until we have found the smallest time where exercise is better.

Let $V(S(t_i))$ denotes the value of holding on to the option at time t_i, from now on called the continuation value, and let the value of exercise at time t_i be the payoff $g(S(t_i))$.

Using the same arguments as in Equation (2.7), the continuation value at time t_i can be described as the conditional expectation

$$V(S(t_i)) = \mathrm{E}\left[\mathrm{e}^{-r(t^*-t_i)} g(S(t^*)) \mid S(t_i)\right]$$

where t^* is the optimal stopping time in $\{t_{i+1}, \ldots, t_N\}$.

To estimate this conditional expectation, the LSM method uses regular least squares regression. This can be done since the conditional expectation is an element in L^2 space, which has an infinite countable orthonormal basis and thus all elements can be represented as a linear combination of a suitable set of basis functions.Therefore, to estimate this we need to choose a (finite) set of orthogonal basis functions, and project the discounted payoffs onto the space spanned by these.However, in this thesis we use a variant of the Longstaff–Schwartz algorithm as in Becker et al. [2020].

Chapter 3

Deep Neural Learning

The goal of this chapter is to introduce several tools from the theory of deep learning, which will be useful throughout this work. We borrow the following preliminaries from Šoljić [2019], Gareth et al. [2013] to make a comprehensive presentation of this thesis.

3.1 Activation Functions

An activation function is a mathematical function which transforms the input data from the previous layer into a meaningful representation, which is closer to the expected output.

Activation Functions can be

- Linear Activation Function;
- Nonlinear Activation Functions

Linear Activation Function

The linear activation function is a straight line (linear) and the output of these functions will range from $-\infty$ to $+\infty$.

Nonlinear Activation Function

Most real world problems are nonlinear in nature and therefore we resort to a nonlinear mapping (function) called activation functions in deep learning. A neural network must be able to take any input from $-\infty$ to $+\infty$, and map it to an output that ranges between $[0, 1]$ or $[-1, 1]$, etc. Nonlinear activation functions are used in the hidden layers of neural networks and are only segregated on the basis of their range or the degree of nonlinearity.

Sigmoid

The sigmoid function is widely used in machine learning for binary classification in logistic regression and neural network implementations for binary classification.

The sigmoid activation of the linear function Z is represented as

$$\sigma(Z) = \frac{1}{1 + e^{-Z}}.$$

The sigmoid function maps the linear input to a nonlinear output.

Hyperbolic Tangent

Like the sigmoid function, the *tanh* function is also sigmoidal ("S"-shaped), but outputs values between -1 and $+1$. Large negative inputs to the *tanh* function will give negative outputs. Further, only zero-valued inputs are mapped to near-zero outputs. These properties does make the network less likely to get "stuck", unlike the sigmoid activation.

The *tanh* activation of Z can be represented by

$$g(Z) = \frac{e^{Z} - e^{-Z}}{e^{Z} + e^{-Z}}$$

The gradient of *tanh* is stronger than sigmoid (i.e., the derivatives are steeper). Like sigmoid, tanh activation also has the vanishing gradient problem.

Derivatives of Activation Functions

The derivative of a function is the rate of change of one quantity over another.What this implies is that we can measure the rate of change of the output error (loss) with

respect to the network weights. If we know how the error changes with the weights, we can change those weights in a direction that decreases the error.

The partial derivative of a function is the rate of change of one quantity over another, irrespective of another quantity if more than two factors are in the function. Partial derivatives come into play because we train neural networks with gradient descent, where we deal with multiple variables.

Derivative of Sigmoid

The sigmoid function outputs a probability score for a specific input and we normally use it as the final layer in neural networks for binary classification. During backpropagation, we need to calculate the derivative (gradient), so that we can pass it back to the previous layer. The first derivative of the sigmoid function will be positive if the input is greater than or equal to zero or negative, if the number is less than or equal to zero.

The gradient of the sigmoid function is

$$\frac{d}{dZ}\sigma(Z) = \sigma(Z)(1 - \sigma(Z)).$$

Derivative of tanh

The derivative of the hyperbolic tangent function is

$$\frac{d}{dZ}tanh = 1 - tanh^2 Z.$$

3.2 Deep Neural Network

In this section we define the network c^θ recursively. Let a function $c^\Theta : \mathbb{R} \to \mathbb{R}$ and

- Let $I \geq 1$ denotes the depth and $q_0, \ldots, q_I$ the numbers of nodes in the diferents layers.
- For $j = 1, \ldots, I-1$, $a^j : \mathbb{R}^{q_j} \to \mathbb{R}^{qj+1}$ are affine functions.
- $\varphi : \mathbb{R} \to \mathbb{R}$ is a activation function.

Now given an input $x \in \mathbb{R}$, the action of the network is given that by letting $a_j^{[i]}$ denote the output, or activation, from neuron j at layer i. Then

$$\begin{aligned} a^{[1]} &= x \\ a^{[l]} &= \varphi(W^{[l]} a^{[l-1]} + b^{[l]}) \text{ for } l = 1, \ldots, I \end{aligned}$$

with this notation $c^\theta(x) = a^{[I]}$.

Remark 3.1. *The weights and biases take the form of matrices and vectors, we are denote by a single vector that we call θ. The training of a network corresponds to choosing the parameters, that is, the weights and biases, that minimize the cost function $J(\theta)$.*

Stochastic gradient descent

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties as differentiable or subdifferentiable. We now introduce a classical method in optimization that is often referred to as steepest descent or gradient descent.

The method proceeds iteratively, computing a sequence of vectors in $\mathbb{R}^q$ with the aim of converging to a vector that minimizes the cost function J. Suppose that our current vector is θ, Taylor series expansion gives

$$J(\theta + \Delta\theta) \approx J(\theta) + \nabla J(\theta)^T \Delta\theta \tag{3.1}$$

Our aim is to reduce the value of the cost function. The relation (3.1) motivates the idea of choosing $\Delta\theta$ to make $\nabla J(\theta)^T \Delta\theta$ as negative as possible. We can address this problem via the Cauchy–Schwarz inequality, which states that for any $f, g \in \mathbb{R}^s$, we have $|f^T g| \leq \|f^2\|_2 \|g^2\|_2$. So the most negative that $f^T g$ can be is $-\|f^2\|_2 \|g^2\|_2$, which happens when $f = -g$.

Hence, based on (3.1), we should choose $\Delta\theta$ to lie in the direction $-\nabla J(\theta)^T$ and keeping in mind that (3.1) is an approximation that is relevant only for small $\Delta\theta$, we will limit ourselves to a small step in that direction. This leads to the update

$$\theta \to \theta - \eta \nabla J(\theta)^T \tag{3.2}$$

where η is a small stepsize that, in this context, is known as the learning rate. This equation defines the steepest descent method. We choose an initial vector and iterate

with (3.2) until some stopping criterion.

From the definition of the cost function

$$\nabla J(\theta) = \frac{1}{K}\sum_{k=1}^{K} \nabla \left(Y_{n+1}^k - C^\theta\left(s_n^k\right)\right)^2 =: \frac{1}{K}\sum_{k=1}^{K} \nabla J^k(\theta) \tag{3.3}$$

When we have a large number of parameters and a large number of training points, computing the gradient vector (3.3) at every iteration of the steepest descent method (3.2) can be prohibitively expensive. A much cheaper alternative is to replace the mean of the individual gradients over all training points by the gradient at a single, randomly chosen, training point. This leads to the simplest form of what is called the stochastic gradient method.

A single step may be summarized as follows:

1. Choose an integer i uniformly at random from $\{1, 2, 3, ..., N\}$.
2. Update
$$\theta \to \theta - \eta \nabla J^i(\theta)$$

The version of the stochastic gradient method that we introduced in (3.5) is only a posibility, i was chosen by sampling with replacement—after using a training point, it is returned to the training set and is just as likely as any other point to be chosen at the next step.An alternative is to sample without replacement; that is, to cycle through each of the N training points in a random order. May be summarized as follows (algorithm epoch):

1. Shuffle the integers $\{1, 2, 3, ..., N\}$ into a new order, $\{k_1, k_2, k_3, \ldots, k_N\}$.
2. Update
$$\theta \to \theta - \eta \nabla J^{k_i}(\theta) \tag{3.4}$$

If we regard the stochastic gradient method as approximating the mean over all training points by a single sample, then it is natural to consider a compromise where we use a small sample average. For some $m \leq N$ we could take steps of the following form:

1. Choose m integers, $\{k_1, k_2, k_3, \ldots, k_m\}$ uniformly at random from $\{1, 2, 3, ..., N\}$

2. Update

$$\theta \to \theta - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla J^{k_i}(\theta) \tag{3.5}$$

In this iteration, the set $\{s^{k_i}\}_{\{i=1:m\}}$ is known as a minibatch. There is a without replacement alternative where, assuming $N = Km$ for some K, we split the training set randomly into K distinct minibatches and cycle through them.

Back Propagation

We are now in a position to apply the stochastic gradient method in order to train an artificial neural network. Our task is to compute partial derivatives of the cost function with respect to each $W_{jk}^{[l]}$ and $b_j^{[l]}$. We have seen that the idea behind the stochastic gradient method to exploit the structure of the cost function: because J is a linear combination of individual terms that runs over the training data, the same is true of its partial derivatives. We therefore focus our attention on computing those individual partial derivatives.

To derive worthwhile expressions for the partial derivatives, it is useful to introduce two further sets of variables. First we let

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \in \mathbb{R}^{nl} \text{ for } l = 2, 3, \ldots, L. \tag{3.6}$$

We refer to $z[l]j$ as the weighted input for neuron j at layer l. The fundamental relation (3.6) that propagates information through the network may then be written as

$$a^{[l]} = \sigma z^{[l]} \text{ for } l = 2, 3, \ldots, L. \tag{3.7}$$

Second, we let $\delta^{[l]} \in \mathbb{R}^{n_l}$ be defined by

$$\delta_j^{[l]} = \frac{\partial J}{\partial z_j^{[l]}} \quad \text{for } 1 \leq j \leq n_l \quad \text{and} \quad 2 \leq l \leq L \tag{3.8}$$

This expression, which is often called the error in the j neuron at layer l, is an intermediate quantity that is useful for both analysis and computation. However, we point out that this usage of the term error is somewhat ambiguous. At a general, hidden layer, it is not clear how much to "blame" each neuron for discrepancies in the final output. Also, at the output layer, L, the expression (3.8) does not quantify those discrepancies directly. The idea of referring to $\delta_j^{[l]}$ in (3.8) as an error seems to have arisen because the cost function can only be at a minimum if all partial derivatives are zero, so $\delta_j^{[l]} = 0$ is a useful goal. As we mention later in this section,

it may be more helpful to keep in mind that $\delta_j^{[l]}$ measures the sensitivity of the cost function to the weighted input for neuron j at layer l.

At this stage we also need to define the Hadamard, or component-wise, product of two vectors. If $x, y \in \mathbb{R}^n$, then $x \circ y \in \mathbb{R}^n$ is defined by $(x \circ y)i = xiy_i$. In words, the Hadamard product is formed by pairwise multiplication of the corresponding components.

With this notation, the following results are a consequence of the chain rule.

Lemma 3.2. *We have*

$$\delta^{[L]} = \sigma'\left(z^{[L]}\right) \circ \left(a^L - y\right) \tag{3.9}$$

$$\delta^{[l]} = \sigma'\left(z^{[l]}\right) \circ \left(W^{[l+1]}\right)^T \delta^{[l+1]} \quad \textit{for } 2 \leq l \leq L-1 \tag{3.10}$$

$$\frac{\partial J}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \textit{for } 2 \leq l \leq L \tag{3.11}$$

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \textit{for } 2 \leq l \leq L \tag{3.12}$$

Proof. We begin by proving (3.9). The relation (3.7) with $l = L$ shows that $z_j^{[L]}$ and $a_j^{[L]}$ are connected by $a^{[L]} = \sigma\left(z^{[L]}\right)$, and hence

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'\left(z_j^{[L]}\right)$$

Also, from J definition,

$$\frac{\partial J}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{k=1}^{n_L} \left(s_k - a_k^{[L]}\right)^2 = -\left(s_j - a_j^{[L]}\right)$$

So, using the chain rule,

$$\delta_j^{[L]} = \frac{\partial J}{\partial z_j^{[L]}} = \frac{\partial J}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \left(a_j^{[L]} - s_j\right) \sigma'\left(z_j^{[L]}\right)$$

which is the componentwise form of (3.9). To show (3.10), we use the chain rule to convert from $z_j^{[l]}$ to $\left\{z_k^{[l+1]}\right\}_{k=1}^{nl+1}$. Applying the chain rule and using the definition

(3.8),

$$\delta_j^{[l]} = \frac{\partial J}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial J}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \tag{3.13}$$

Now, from (3.6) we know that $z_k^{[l+1]}$ and $z_j^{[l]}$ are connected via

$$z_k^{[l+1]} = \sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma\left(z_s^{[l]}\right) + b_k^{[l+1]}.$$

Hence,

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \sigma'\left(z_j^{[l]}\right)$$

In (3.13) this gives

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'\left(z_j^{[l]}\right)$$

which may be rearranged as

$$\delta_j^{[l]} = \sigma'\left(z_j^{[l]}\right) \left(\left(W^{[l+1]}\right)^T \delta^{[l+1]}\right)_j$$

This is the componentwise form of (3.10). To show (3.11), we note from (3.6) and (3.7) that $z_j^{[l]}$ is connected to $b_j^{[l]}$ by

$$z_j^{[l]} = \left(W^{[l]} \sigma\left(z^{[l-1]}\right)\right) j + bj^{[l]}.$$

Since $z^{[l-1]}$ does not depend on $b_j^{[l]}$, we find that

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$$

Then, from the chain rule,

$$\frac{\partial J}{\partial b_j^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} = \delta_j^{[l]}$$

using the definition (3.8). This gives (3.11) . Finally, to obtain (3.12) we start with the componentwise version of (3.6),

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}$$

which gives

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}, \quad \text{independently of } j, \tag{3.14}$$

and

$$\frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0 \quad \text{for } s \neq j \tag{3.15}$$

In words, (3.14) and (3.15) follow because the j th neuron at layer l uses the weights from only the j th row of $W^{[l]}$ and applies these weights linearly. Then, from the chain rule, (3.14) and (3.15) give

$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial J}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]}$$

where the last step used the definition of $\delta_j^{[l]}$ in (3.8). This completes the proof.

There are many aspects of Lemma 1 that deserve our attention. We recall from (3.6), and (3.7) that the output $a^{[L]}$ can be evaluated from a forward pass through the network, computing $a^{[1]}, z^{[2]}, a^{[2]}, z^{[3]}, \ldots, a^{[L]}$ in order. Having done this, we see from (3.9) that $\delta^{[L]}$ is immediately available. Then, from (3.10), $\delta^{[L-1]}$, $\delta^{[L-2]}, \ldots, \delta^{[2]}$ may be computed in a backward pass. From (3.11) and (3.12), we then have access to the partial derivatives. Computing gradients in this way is known as back propagation.

To gain further understanding of the back propagation formulas (3.11) and (3.12) in Lemma 5.1, it is useful to recall the fundamental definition of a partial derivative. The quantity $\partial J/\partial w_{jk}^{[l]}$ measures how C changes when we make a small perturbation to $w_{jk}^{[l]}$. It is clear that a change in this weight has no effect on the output of previous layers, so to work out $\partial J/\partial w_{43}^{[3]}$ we do not need to know about partial derivatives at previous layers. It should, however, be possible to express $\partial J/\partial w_{43}^{[3]}$ in terms of partial derivatives at subsequent layers. More precisely, the activation feeding into the fourth neuron on layer 3 is $z_4^{[3]}$, and, by definition, $\delta_4^{[3]}$ measures the sensitivity of J with respect to this input. Feeding in to this neuron we have $w_{43}^{[3]} a_3^{[2]} +$ constant, so it makes sense that

$$\frac{\partial J}{\partial w_{43}^{[3]}} = \delta_4^{[3]} a_3^{[2]}$$

Similarly, in terms of the bias, $b_4^{[3]}+$ constant is feeding in to the neuron, which explains why

$$\frac{\partial J}{\partial b_4^{[3]}} = \delta_4^{[3]} \times 1$$

We may avoid the Hadamard product notation in (3.9) and (3.10) by introducing diagonal matrices. Let $D^{[l]} \in \mathbb{R}^{n_l \times n_l}$ denote the diagonal matrix with (i,i) entry given by $\sigma'\left(z_i^{[l]}\right)$. Then we see that $\delta^{[L]} = D^{[L]}\left(a^{[L]} - y\right)$ and $\delta^{[l]} = D^{[l]}\left(W^{[l+1]}\right)^T \delta^{[l+1]}$. We could expand this out as

$$\delta^{[l]} = D^{[l]}\left(W^{[l+1]}\right)^T D^{[l+1]}\left(W^{[l+2]}\right)^T \cdots D^{[L-1]}\left(W^{[L]}\right)^T D^{[L]}\left(a^{[L]} - y\right)$$

We also recall from (2.2) that $\sigma'(z)$ is trivial to compute.

The relation (3.11) shows that $\delta^{[l]}$ corresponds precisely to the gradient of the cost function with respect to the biases at layer l. If we regard $\partial C/\partial w_{jk}^{[l]}$ as defining the (j,k) component in a matrix of partial derivatives at layer l, then (3.12) shows this matrix to be the outer product $\delta^{[l]} a^{[l-1]^T} \in \mathbb{R}^{n_l \times n_{l-1}}$.

Putting this together, we may write the following pseudocode for an algorithm that trains a network using a fixed number, Niter, of stochastic gradient iterations. For simplicity, we consider the basic version (3.5) where single samples are chosen with replacement. For each training point, we perform a forward pass through the network in order to evaluate the activations, weighted inputs, and overall output $a^{[L]}$. Then we perform a backward pass to compute the errors and updates.

For counter = 1 upto Niter

1. Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$.
2. $s^{\{k\}}$ is current training data point $a^{[1]} = s^{\{k\}}$.
3. For $l = 2$ upto L

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} a^{[}l]$$
$$a^{[l]} = \sigma(z^{[l]})$$
$$D^{[l]} = diag(\sigma'(z^{[l]}))$$

end

$$\delta^{[L]} = D^{[L]}(a^{[L]} - s(x^k))$$

4. For $l = 2$ upto L

$$\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$$

end

5. For $l = L$ downto 2

$$W^{[l]} \to W^{[l]} - \eta \delta^{[l]} a^{[l-1]^T}$$
$$b^{[l]} \to b^{[l]} - \eta \delta^{[l]}$$

end

end

Chapter 4

Pricing and Hedging American-Style Options

In this chapter we introduce a deep learning method for pricing and hedging American-style options as Becker et al. [2020]. In this part, we only rewrite the results of this author to show bounds, point estimates and confidence intervals for pricing. Also, we show an approximate dynamic hedging strategy of this author.

4.1 Calculating a Candidate Optimal Stopping Strategy

Let $X = (X_n)_{n=0}^N$ be an $\mathbb{R}^d$-valued discrete-time process on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where N and d are positive integers. We denote by $\mathcal{F}_n$ the σ-algebra generated by $X_0, X_1, \ldots, X_n$. We consider the filtration $\mathbb{F} = \{\mathcal{F}_n\}_{n=0}^N$.

Definition 4.1. *The process $X = (X_n)_{n=0}^N$ is called a Markov process if X_n is $\mathcal{F}_n$-measurable, and*

$$\mathbb{E}\left[f\left(X_{n+1}\right) \mid \mathcal{F}_n\right] = \mathbb{E}\left[f\left(X_{n+1}\right) \mid X_n\right]$$

for all $n \leq N-1$ and every measurable function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f\left(X_{n+1}\right)$ is integrable.

Definition 4.2. *We call a random variable $\tau : \Omega \to \{0, 1, \ldots, N\}$ an $\mathbb{F}$-stopping time if the event $\{\tau = n\}$ belongs to $\mathcal{F}_n$ for all $n \in \{0, 1, \ldots, N\}$.*

Our aim is to develop a deep learning method that can efficiently learn an optimal policy for stopping problems of the form

$$\sup_{\tau \in \mathcal{T}} \mathbb{E} g\left(\tau, X_\tau\right),$$

and

$$\operatorname*{ess\,sup}_{\tau\in\mathcal{T}_n}\mathbb{E}\left[g\left(\tau,X_\tau\right)\mid\mathcal{F}_n\right]$$

where $g:\{0,1,\ldots,N\}\times\mathbb{R}^d\to\mathbb{R}$ is a measurable function and $\mathcal{T}$ denotes the set of all $\mathbb{F}$-stopping times $\tau:\Omega\to\{0,1,\cdots,N\}$ and $\mathcal{T}_n$ is the set of all $\mathbb{F}$-stopping times satisfying $n\le\tau\le N$.

We use the previous stochastic framework to model an American-style option that can be exercised at any one of finitely many times $0=t_0<t_1<\cdots<t_N=T$. Now, if exercised at time t_n, it yields a discounted payoff given by a square-integrable random variable G_n defined on a filtered probability space $\left(\Omega,\mathcal{F},\mathbb{F}=(\mathcal{F}_n)_{n=0}^N,\mathbb{P}\right)$. We assume that $\mathcal{F}_n$ describes the information available at time t_n and G_n is of the form $g\left(n,X_n\right)$ for a measurable function $g:\{0,1,\ldots,N\}\times\mathbb{R}^d\to[0,\infty)$ and the process $X=(X_n)_{n=0}^N$ is a Markov process.

For the Markov process $(X_n)_{n=0}^N$ with X_0 to be deterministic and $\mathbb{P}$ to be the pricing measure, the value of the option at time 0 is given by

$$V=\sup_{\tau\in\mathcal{T}}\mathbb{E}g(\tau,X_\tau). \tag{4.1}$$

If the option has not been exercised before time t_n, its discounted value at that time is

$$V_{t_n}=\operatorname*{ess\,sup}_{\tau\in\mathcal{T}_n}\mathbb{E}\left[g(\tau,X_\tau\mid\mathcal{F}_n\right] \tag{4.2}$$

where $\mathcal{T}_n$ is the set of all $\mathbb{F}$-stopping times satisfying $n\le\tau\le N$.

Remark 4.3. *Note that $V_T=V_{t_N}\operatorname{ess\,sup}_{\tau\in\mathcal{T}_N}\mathbb{E}\left[g(\tau,X_\tau\mid\mathcal{F}_N\right]$. So that $\tau_N\equiv N$ is optimal for $V_T=G_N$ since $\mathcal{T}_N$ is the set of all $\mathbb{F}$-stopping times satisfying $N\le\tau\le N$.*

Proposition 4.4. *Let $\tau_N\equiv N$ and τ_n be the stopping times recursively construct by*

$$\tau_n:=\begin{cases}n & \text{if } G_n\ge\mathbb{E}\left[G_{\tau_{n+1}}\mid X_n\right]\\ \tau_{n+1} & \text{if } G_n<\mathbb{E}\left[G_{\tau_{n+1}}\mid X_n\right]\end{cases}. \tag{4.3}$$

Then

$$V_{t_n}=\mathbb{E}\left[G_{\tau_n}\mid\mathcal{F}_n\right]=\max\{G_n,\mathbb{E}\left[V_{t_{n+1}}\mid X_n\right]\}\quad\text{for all } n\le N-1$$

Moreover, τ_n is an optimizer of (4.2).

Proof: The proof can be done inductively.

We need the following result (see Bru and Heinich [1985]).

Lemma 4.5. $\mathbb{E}\left[G_{\tau_{n+1}} \mid X_n\right]$ *is of the form* $c(X_n)$*, where* $c : \mathbb{R}^d \to \mathbb{R}$ *minimizes the mean squared distance* $\mathbb{E}\left[\left\{G_{\tau_{n+1}} - c(X_n)\right\}^2\right]$ *over all Borel measurable functions from* $\mathbb{R}^d$ *to* $\mathbb{R}$

4.2 Neural Network Approximation

In this section we describe our neural network version of the Longstaff–Schwartz algorithm to estimate continuation values and construct a candidate optimal stopping strategy.

The Longstaff-Schwartz method is a backward iteration algorithm, which steps backward in time from the maturity date. At each exercise date, the algorithm approximates the continuation value, which is the value of the option if it is not exercised.

The Longstaff-Schwartz algorithm approximates $\mathbb{E}\left[G_{\tau_{n+1}} \mid X_n\right]$ by projecting $G_{\tau_{n+1}}$ on the linear span of finitely many basis functions.

We assume in the following that $t_n = nT/N$ for $n = 0, 1, \ldots, N$. We present the following variant of the Longstaff-Schwartz algorithm:

(i) Simulate $\left\{x_n^k\right\}_{n=0}^N$, $k = 1, \cdots, M$ of the underlying $\{X_n\}_{n=0}^N$.

(ii) $l_N^k \equiv N$ for all $k = 1, \ldots, M$.

(iii) Evaluate for $k = 1, \ldots, M$, $Y_N^k = g\left(l_N^k, x_{l_N^k}^k\right)$.

For $1 \leq n \leq N-1$,

(iv) Approximate $E\left[G_{\tau_{n+1}} \mid X_n\right]$ with $c^{\theta_n}(X_n)$ by minimizing the sum

$$J(\theta) = \frac{1}{M}\sum_{k=1}^{M}\left(Y_{n+1}^k - c^{\theta}\left(s_n^k\right)\right)^2 \text{ over } \theta.$$

where $Y_{n+1}^k = g\left(l_{n+1}^k, x_{l_{n+1}^k}^k\right)$ for $k = 1, \ldots, M$,

(v) Set

$$l_n^k := \begin{cases} n & \text{if } g\left(n, x_n^k\right) \geq c^{\theta_n}\left(x_n^k\right) \\ l_{n+1}^k & \text{otherwise} \end{cases}$$

(vi) Define $\theta_0 := \frac{1}{M}\sum_{k=1}^{M} Y_{l_1^k}^k$, and set c^{θ_0} constantly equal to θ_0.

From now on we specify c^{Θ} as a feedforward neural network as in Section 3.2.

4.3 Pricing: Bounds, Point Estimates and Confidence Intervals

In this section we derive lower and upper bounds as well as point estimates and confidence intervals for the optimal value

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

Lower Bound

We set $\Theta = (\theta_0, \ldots, \theta_{N-1})$ where $\theta_0, \theta_1, \ldots, \theta_{N-1}$ are the parameters determined by the variant algorithm of the Longstaff–Schwartz algorithm in section 4.2. We set $A_{N-1} = \{0, 1, \ldots, N-1\}$ and define

$$\tau^{\Theta} := \min\left\{n \in A_{N-1} : g\left(n, X_n\right) \geq c^{\theta_n}\left(X_n\right)\right\}, \tag{4.4}$$

where $\min \varnothing$ is understood as N.

Lemma 4.6. *τ^{Θ} defined as above is a valid $\mathbb{F}$-stopping time. Moreover, $L = \mathbb{E}g\left(\tau^{\Theta}, X_{\tau^{\theta}}\right)$ is a lower bound for the optimal value V.*

Remark 4.7. *Usually, it is not possible to calculate the expectation $L = \mathbb{E}g\left(\tau^{\Theta}, X_{\tau^{\theta}}\right)$ exactly. So that, we generate simulations y_{Θ}^k of $Y_{\Theta} = g\left(\tau^{\Theta}, X_{\tau^{\Theta}}\right) = \max(0, X_{\tau^{\Theta}} - K)$ based on independent sample paths $\left(x_n^k\right)_{n=0}^N$, $\quad k = M+1, \ldots, M+ML$, of $(X_n)_{n=0}^N$ and the Monte Carlo average*

$$\hat{L} = \frac{1}{ML}\sum_{k=M+1}^{M+ML} y_{\Theta}^k$$

gives an estimate of the lower bound L.

Also, by the law of large numbers, $\hat{L}$ converges to L for $ML \to \infty$.

Upper Bound

To derive the upper bound for V, we need the Proposition 7 of Becker et al. [2019]. Now, to present this result we use the preliminary results of the same author as follows.

The Snell envelope of the process $(g(n, X_n))_{n=0}^N$ is the smallest supermartingale with respect to $(\mathcal{F}_n)_{n=0}^N$ that dominates $(g(n, X_n))_{n=0}^N$. It is given by

$$H_n = \operatorname{esssup}_{\tau \in \mathcal{T}_n} \mathbb{E}\left[g(\tau, X_\tau) \mid \mathcal{F}_n\right], \quad n = 0, 1, \ldots, N.$$

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

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

and A^H is the nondecreasing $(\mathcal{F}_n)$-predictable process given [6] by

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

The next proposition is the Proposition 7 of Becker et al. [2019].

Proposition 4.8. *Let $(\varepsilon_n)_{n=0}^N$ be a sequence of integrable random variables on $(\Omega, \mathcal{F}, \mathbb{P})$. Then*

$$V \geq \mathbb{E}\left[\max_{0 \leq n \leq N} \left(g(n, X_n) - M_n^H - \varepsilon_n\right)\right] + \mathbb{E}\left[\min_{0 \leq n \leq N} \left(A_n^H + \varepsilon_n\right)\right]. \tag{4.5}$$

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

$$V \leq \mathbb{E}\left[\max_{0 \leq n \leq N} \left(g(n, X_n) - M_n - \varepsilon_n\right)\right] \tag{4.6}$$

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

Note that, if we choose $M = M^H$ and $\varepsilon \equiv 0$, we get from (4.5)

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

and from (4.6), we conclude that the optimal value V can be written as

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

So we try to use our candidate optimal stopping time τ^Θ to construct a martingale close to M^H. So that, the best value process

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

corresponding to τ^Θ approximates the Snell envelope $(H_n)_{n=0}^N$. The martingale part of $\left(H_n^\Theta\right)_{n=0}^N$ is given by $M_0^\Theta = 0$ and

$$M_n^\Theta = M_{n-1}^\Theta + H_n^\Theta - \mathbb{E}\left[H_n^\Theta \mid \mathcal{F}_{n-1}\right], n \geq 1$$

obtained from the stopping decisions implied by the trained continuation value functions c^{θ_n}, $\quad n = 0, 1, \ldots, N-1$.

Now, from (4.6) we have that

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

is an upper bound for V.

Now proceeding as in Becker et al. [2019] and Becker et al. [2020]. To estimate M^Θ, we generate a third set of independent realizations $\left(x_n^k\right)_{n=0}^N, k = M + ML + 1, \ldots, M + ML + MU$, of $(X_n)_{n=0}^N$. In addition, for every x_n^k, we simulate J continuation paths $\tilde{x}_{n+1}^{k,j}, \ldots, \tilde{x}_N^{k,j}, j = 1, \ldots, J$, that are conditionally independent of each other and of $x_{n+1}^k, \ldots, x_N^k$. Let us denote by $\tau_{n+1}^{k,j}$ the value of τ_{n+1}^Θ along $\tilde{x}_{n+1}^{k,j}, \ldots, \tilde{x}_N^{k,j}$. Estimating the continuation values as

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

yields the noisy estimates (see (18) in Becker et al. [2020])

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

of the increments $M_n^\Theta - M_{n-1}^\Theta$ along the k-th simulated path $x_0^k, \ldots, x_N^k$. Here

$$f^{\theta_n}\left(x_n^k\right) = \mathbb{1}_{\left(g\left(n, x_n^k\right) \geq C^\theta\left(x_n^k\right)\right)}$$

So

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

can be viewed as realizations of $M_n^\Theta + \varepsilon_n$ for estimation errors ε_n with standard deviations proportional to $1/\sqrt{J}$ such that $\mathbb{E}\left[\varepsilon_n \mid \mathcal{F}_n\right] = 0$ for all n. Accordingly,

$$\hat{U} = \frac{1}{MU} \sum_{k=M+ML+1}^{M+ML+MU} \max_{0 \leq n \leq N} \left(g\left(n, x_n^k\right) - m_n^k \right)$$

is an unbiased estimate of the upper bound of U and which, by the law of large numbers, converges to U for $MU \to \infty$.

Remark 4.9. *The use of nested simulation ensures that m_n^k are unbiased estimates of $M_n^\ominus$, which is crucial for the validity of the upper bound. In particular, we do not directly approximate $M_n^\ominus$ with the estimated continuation value functions c^{θ_n}.*

Point Estimate and Confidence Intervals

Our point estimate of V is

$$\hat{V} = \frac{\hat{L} + \hat{U}}{2}$$

In the following proposition we derive a confidence interval for the lower bound L.

Proposition 4.10. *Let $z_{a/2}$ be the $1 - \alpha/2$ quantile of the standard normal distribution and consider the sample standard deviation*

$$\hat{\sigma}_L = \sqrt{\frac{1}{ML - 1} \sum_{k=M+1}^{M+ML} \left(g^k - \hat{L} \right)^2}.$$

Then

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

is an asymptotically valid $1 - \alpha/2$ confidence interval for L.

Proof: We have the result from the Central Limit Theorem since for large ML, $\hat{L}$ is approximately normally distributed with mean L and variance $\hat{\sigma}_L^2/ML$.

The next proposition we derive a confidence interval for the upper bound U.

Proposition 4.11. *Let $z_{a/2}$ be the $1-\alpha/2$ quantile of the standard normal distribution and consider the sample standard deviation of the estimator $\hat{U}$, given by*

$$\hat{\sigma}_U = \sqrt{\frac{1}{MU-1}\sum_{k=M+ML+1}^{M+ML+MU}\left(\max_{0\leq n\leq N}\left(g\left(n,x_n^k\right)-m_n^k\right)-\hat{U}\right)^2},$$

Then

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

is an asymptotically valid $1-\alpha/2$ confidence interval for U.

Proof: the proof is similar to the above proposition.

The previous two propositions can be used to construct the asymptotically valid two-sided $1-\alpha$ confidence interval

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

for the true value V.

4.4 Hedging

We now consider a savings account together with $e \in \mathbb{N}$ financial securities as hedging instruments. We fix a positive integer M and introduce a time grid $0 = u_1 < u_2 < \cdots < u_{NM}$ such that $u_{nM} = t_n$ for all $n = 0, 1, \ldots, N$.

We will need the following assumptions:

- The information available at time u_m is described by $\mathcal{H}_m$, where $\mathbb{H} = (\mathcal{H}_m)_{m=0}^{MN}$ is a filtration satisfying $\mathcal{H}_{nM} = \mathcal{F}_n$ for all n.
- If any of the financial securities pay dividends, they are immediately reinvested.
- The resulting discounted value processes are of the form $P_{u_m} = p_m(Y_m)$ for measurable functions $p_m : \mathbb{R}^d \to \mathbb{R}^e$ and an $\mathbb{H}$-Markov process $(Y_m)_{m=0}^{NM}$ such that $Y_{nM} = X_n$ for all $n = 0, \ldots, N$.

Definition 4.12. *A hedging strategy consists of a sequence*

$$h = (h_m)_{m=0}^{NM-1}$$

of functions $h_m : \mathbb{R}^d \to \mathbb{R}^e$ specifying the time- u_m holdings in $P_{u_m}^1, \ldots, P_{u_m}^e$.

Remark 4.13. *As usual, money is dynamically deposited in or borrowed from the savings account to make the strategy self-financing. The resulting discounted gains at time u_m are given by*

$$(h\cdot P)_{u_m} := \sum_{j=0}^{m-1} h_j\left(Y_j\right)\cdot\left(p_{j+1}\left(Y_{j+1}\right)-p_j\left(Y_j\right)\right) := \sum_{j=0}^{m-1}\sum_{i=1}^{e} h_j^i\left(Y_j\right)\left(p_{j+1}^i\left(Y_{j+1}\right)-p_j^i\left(Y_j\right)\right)$$

4.4.1 Hedging Until the First Possible Exercise Date

Here we choose $t_n = n\Delta$ for a small amount of time Δ such as a day. We assume τ^Θ does not stop at time 0. Otherwise, there is nothing to hedge. In a first step, we only compute the hedge until time t_1. If the option is still alive at time t_1, the hedge can then be computed until time t_2 and so on.

Construction of the hedge from time 0 to t_1

- Approximate the time-t_1 value of the option with

  $$V_{t_1}^{\theta_1} = \nu^{\theta_1}(X_1)$$

  for the function

  $$\nu^{\theta_1}(x) = \max\{g(1,x), c^{\theta_1}(x)\},$$

  where $c^{\theta_1} : \mathbb{R}^d \to \mathbb{R}$ is the time-t_1 continuation value function estimated in Section 4.2.

- Next, search for hedging positions $h_m, m = 0, 1, \ldots, M-1$, that minimize the mean squared error

  $$\mathbb{E}\left[\left(\hat{V} + (h\cdot P)_{t_1} - V_{t_1}^{\theta_1}\right)^2\right].$$

- To do that we approximate the functions h_m with neural networks $h^\lambda : \mathbb{R}^d \to \mathbb{R}^e$ as in Section 4.2 and try to find parameters $\lambda_0, \ldots, \lambda_{M-1}$ that minimize

  $$\sum_{k=1}^{K_H}\left(\hat{V} + \sum_{m=0}^{M-1} h^{\lambda_m}\left(y_m^k\right)\cdot\left(p_{m+1}\left(y_{m+1}^k\right) - p_m\left(y_m^k\right)\right) - v^{\theta_1}\left(y_M^k\right)\right)^2$$

  for independent realizations of $\left(y_m^k\right)_{m=0'}^M k = 1, \ldots, K_H$ of $(Y_m)_{m=0}^M$. We train the networks $h^{\lambda_0}, \ldots, h^{\lambda_{M-1}}$ together, again using a stochastic gradient descent method.

Quality of the hedge

Once $\lambda_0, \ldots, \lambda_{M-1}$ have been determined, we assess the quality of the hedge by simulating new independent realizations $\left(y_m^k\right)_{m=0}^M, k = K_H + 1, \ldots, K_H + K_E$ of $(Y_m)_{m=0}^M$ and calculating the average hedging error

$$\frac{1}{K_E} \sum_{k=K_H+1}^{K_H+K_E} \left(\hat{V} + \sum_{m=0}^{M-1} h^{\lambda_m}\left(y_m^k\right) \cdot \left(p_{m+1}\left(y_{m+1}^k\right) - p_m\left(y_m^k\right)\right) - v^{\theta_1}\left(y_M^k\right) \right) \tag{4.7}$$

and the empirical hedging shortfall

$$\frac{1}{K_E} \sum_{k=K_H+1}^{K_H+K_E} \left(\hat{V} + \sum_{m=0}^{M-1} h^{\lambda_m}\left(y_m^k\right) \cdot \left(p_{m+1}\left(y_{m+1}^k\right) - p_m\left(y_m^k\right)\right) - v^{\theta_1}\left(y_M^k\right) \right)^- \tag{4.8}$$

over the time interval $[0, t_1]$.

4.4.2 Hedging Until the Exercise Time

In this case, we can precompute the whole hedging strategy from time 0 to T and then use it until the option is exercised. In order to do that we introduce the functions

$$v^{\theta_n}(x) := \max\{g(n,x), c^{\theta_n}(x)\}, \qquad C^{\theta_n}(x) := \max\{0, c^{\theta_n}(x)\}, \quad x \in \mathbb{R}^d,$$

and hedge the difference $v^{\theta_n}\left(Y_{nM}\right) - C^{\theta_{n-1}}\left(Y_{(n-1)M}\right)$ on each of the time intervals $[t_{n-1}, t_n]$, $n = 1, \ldots, N$, separately. v^{θ_n} describes the approximate value of the option at time t_n if it has not been exercised before, and the definition of C^{θ_n} takes into account that the continuation values are non-negative due to the non-negativity of the payoff function g.

The hedging strategy

The hedging strategy can be computed as the hedge from time 0 to t_1, except that we now have to simulate complete paths $\left(y_m^k\right)_{m=0}^{NM}$ of $(Y_m)_{m=0}^{NM}$ $k = 1, \ldots, K_H$, and then for all $n = 1, \ldots, N$, find parameters $\lambda_{(n-1)M}, \ldots, \lambda_{nM-1}$ which minimize

$$\sum_{k=1}^{K_H} \left(C^{\theta_{n-1}}\left(y_{(n-1)M}^k\right) + \sum_{m=(n-1)M}^{nM-1} h^{\lambda_m}\left(y_m^k\right) \cdot \left(p_{m+1}\left(y_{m+1}^k\right) - p_m\left(y_m^k\right)\right) - v^{\theta_n}\left(y_{nM}^k\right) \right)^2$$

Average hedging error and the empirical hedging shortfall

Once the hedging strategy has been trained, we simulate independent samples $\left(y_m^k\right)_{m=0'}^{NM} k = K_H+1,\ldots,K_H+K_E$ of $(Y_m)_{m=0}^{NM}$ and denote the realization of τ^Θ along each sample path $\left(y_m^k\right)_{m=0}^{NM}$ by τ^k. The corresponding average hedging error is given by

$$\frac{1}{K_E}\sum_{k=K_H+1}^{K_H+K_E}\left(\hat{V}+\sum_{m=0}^{\tau^k M-1}h^{\lambda_m}\left(y_m^k\right)\cdot\left(p_{m+1}\left(y_{m+1}^k\right)-p_m\left(y_m^k\right)\right)-g\left(\tau^k,X_{\tau^k}\right)\right) \tag{4.9}$$

and the empirical hedging shortfall by

$$\frac{1}{K_E}\sum_{k=K_H+1}^{K_H+K_E}\left(\hat{V}+\sum_{m=0}^{\tau^k M-1}h^{\lambda_m}\left(y_m^k\right)\cdot\left(p_{m+1}\left(y_{m+1}^k\right)-p_m\left(y_m^k\right)\right)-g\left(\tau^k,X_{\tau^k}\right)\right)^- . \tag{4.10}$$

So far the Markov process $X=\{X_n\}_{n=0}^N$ is arbitrary. Here we present an example of a Markov process that we will use in the next chapter to show the numerical results.

Example 4.14. *Consider a d-dimensional geometric Brownian motion*

$$S=(S^1,\cdots,S^d)$$

where

$$S_t^i=s_0^i\exp\left(\left[r-\delta_i-\sigma_i^2/2\right]t+\sigma_i W_t^i\right),\quad i=1,2,\ldots,d,$$

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

We can consider a Bermudan max-call option on d financial securities with risk-neutral price dynamics the d-dimensional geometric Brownian motion for a risk-free interest rate r, initial values s_0^i*, dividend yields* δ_i *and volatilities* σ_i*. The option has time-t payoff* $(\max_{1\le i\le d}S_t^i-K)^+$ *for a strike price* $K\in[0,\infty)$ *and can be exercised at one of finitely many times* $0=t_0<t_1<\cdots<t_N=T$*. In addition, we suppose there is a savings account where money can be deposited and borrowed at rate r.*

So that we can construct an example of Markov process by $X_n=S_{t_n}, n=0,1,\ldots,N$*.*

So that the price of the option is given by

$$\sup_\tau\mathbb{E}\left[e^{-r\frac{\tau T}{N}}\left(\max_{1\le i\le d}X_\tau^i-K\right)^+\right]$$

where the supremum is over all stopping times $\tau : \Omega \to \{0, 1, \ldots, N\}$ *with respect to the filtration generated by* $(X_n)_{n=0^*}^N$.

Chapter 5

Results

In this section, we implement the previous algorithms for a Bermudan max-call option with $d = 1$ and price dynamics

$$S_t = S_0 \exp\left(\left[r - \delta - \sigma^2/2\right] t + \sigma W_t\right) \tag{5.1}$$

for a risk-free rate $r \in \mathbb{R}$, initial values $S_0 \in (0, +\infty)$, dividend yields $\delta \in [0, \infty)$, volatility σ and a Brownian motion W. We suppose there is a savings account where money can be deposited and borrowed at rate r. We assume that have risk-neutral price dynamics

The option has time- t payoff $g(t, S_t) = \max\{0, S_t - K\}$ and can be exercised at one of finitely times $t_1 < t_2 < \ldots < t_N = T$ for a strike price $K \in [0, \infty)$ and can be exercised at one of finitely many times $0 = t_0 < t_1 < \cdots < t_N = T$.

For notational simplicity, we assume in the following that $t_n = nT/N$ for $n = 0, 1, \cdots, N$, as in Section 4.2 and let $\{X_n\}$ be the process $X_n = S_{t_n}$, for all $n = 0, 1, \cdots, N$.

Simulation

The asset price model is solution of the stochastic differential equation (SDE) defined by

$$dS_t = (r - \delta)S_t dt + \sigma S_t dW_t,$$

this equation can be discretized by the Euler-Maruyama method and from this discretization obtain a simulation. In our case we can simulate directly from the exact

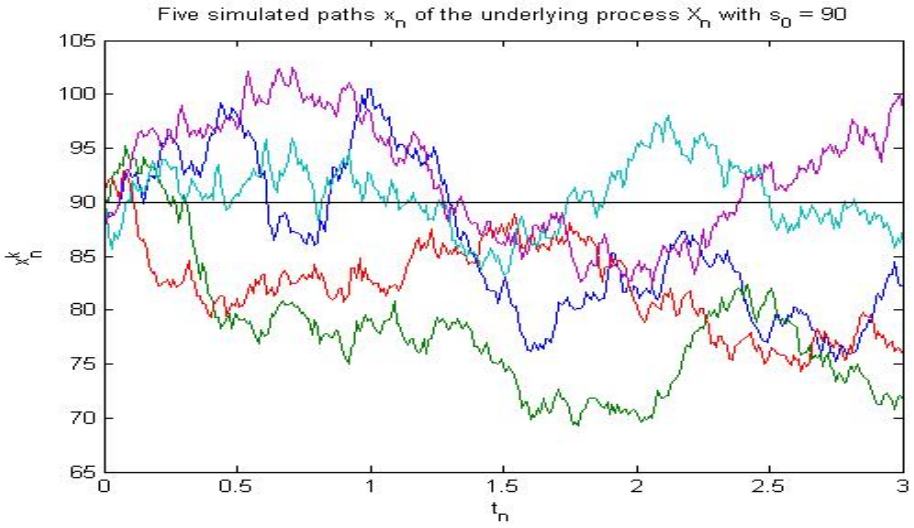

Figure 5.1: Underlying process

solution from Brownian motion simulations.

In the Appendix A.1, we present an algorithm from Higham and Higham [2019] to simulate the underling $\{X_n\}_{n=0}^{N}$. From this algorithm we get the paths of the underlying process $\{X_n\}_{n=0}^{N}$:

The code to deep learning

We have written a MATLAB function named netbpBERMUDA.m to simulate K paths of the underlying process $(X_n)_{n=0}^{N}$ and to use this data set for training back propagation Neural Networks that approximate the expected value of $G_{\tau_{n+1}}$ given X_n. Using the K simulated values X_i corresponding to a specific time a Neural Network is trained to learn to predict the expected value of $G_{\tau_{i+1}}$. Next paragraphs describe the algorithm.

Simulating the underlying process.

The code lines as in Appendix A.2.1 simulate M paths of the process. Each path consists of N intervals samples at the specific times in the array t.

When the program runs it calculates a bidimensional array S with M rows consisting of N_intervals+1 values corresponding to M simulations of the price dynamics X_n.

Scaling the training input data

As we will explain next, we use a back propagation Neural Network with Sigmoide activation function. This function sets a limit to the output so we should scale the desired output. The lines of code as in A.2.2, scales input and output data.

Setting Neural Network parameters

As the authors in Becker et al. [2020], we use back Propagation Neural Networks with two hidden layers and use Sigmoide as activation function. The number of computing elements (neurons) in each layer is not fixed but can be set by the user. Given limitations of the available computing system, the number of neurons could not be set to 50 as the authors in Becker et al. [2020] did. The lines of code as in A.2.3 set number of neurons in each layer.

Training the Networks

The algorithm used for training all the networks is the classical stochastic gradient as it has been described in Becker et al. [2020]. The code used for training the Networks is in A.2.4.

Monitoring the training process

For the training process we have set a fixed number of forward and backward iterations of the back propagation algorithm instead of setting an error bound. In order to evaluate the progress of the training and the learning process of the networks periodically we calculate the error after a fixed number of iterations. To calculate this error we evaluate the Network using the full input data set and compare the Network output with the desired output and calculate the sum of the squared errors. The lines of code to calculate the errors during the training process and store them for monitoring the training process is in A.2.5.

Numerical results of the training process

We have used a data set consisting of 50 simulated paths of the underlying process $(X_n)_{n=0}^N$ to train a Neural Network with two neurons in layer two, three neurons in layer three and only one neuron in layer four. With a learning rate eta = 0.05, the progress of the Neural Network error through the training process is shown in the Figure 5.2.

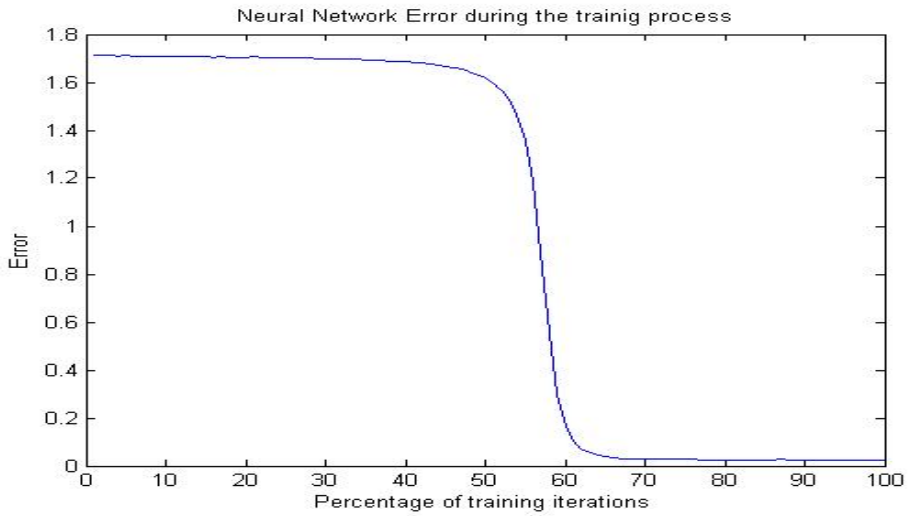

Figure 5.2: Underlying process

When we change the learning rate to eta = 0.10, the progress of the Neural Network error through the training process is shown in the Figure 5.3.

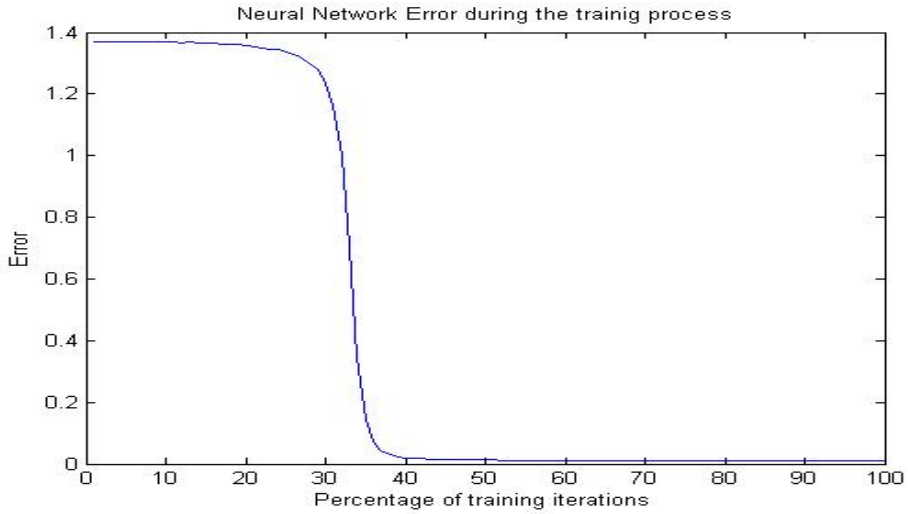

Figure 5.3: Underlying process

We have written a Matlab program named **main.m** to simulate realizations of the underlying process $(X_n)_{n=0}^N$, train a set of neural networks to approximate the expected value of $G_{\tau_{n+1}}$ given X_n and use the trained neural networks to approximate a lower bound L for the optimal value V. The main program uses Matlab functions **simulate_paths.m**, **train_networks.m**. In the following we describe each one of these functions, their input parameters, what they do and their corresponding outputs.

The Matlab function **simulate_paths.m** receives as input parameters

```
Tmax: the total simulation time
N_intervals: number of subintervals
r : the risk-free interest rate
delta: dividend yields
S_0: initial value of S at t=0
Sigma: the volatility
M: number of paths to be simulated
K: strike Price
```

When the function **simulate_paths.m** runs it simulates M realizations of the underlying process, each realization consisting of $N_{intervals}+1$ samples equally spaced along $Tmax$. The functions yields as output the following structures

```
t: array of N+1 times at which the process has been simulated.
S: M×t matrix with the values of the simulated process.
g: M×t matrix with the g values (for each entry in the matrix,
g_((i,j) )=max(0,S_((i,j) )-K).
```

Once the main program executes the function **simulate_paths.m**, we have matrix S and g which can be used to train the set of neural networks. The Matlab function **train_networks.m** receives as input parameters

```
neurons_in_layer_2: number of neurons in second layer of the neural network.
neurons_in_layer_3: number of neurons in third layer of the neural network.
neurons_in_layer_4: number of neurons in fourth layer of the neural network.
S: M×t matrix with the values of the simulated process.
g: M×t matrix with the g values (for each entry in the matrix,
g_((i,j) )=max(0,S_((i,j) )-K).
eta: learning rate of the training algorithm.
Niter: number of iterations of the training algorithm.
Niter_to_plot: number of iterations between points to be plotted.
```

When the function **train_networks.m** runs it uses the stochastic conjugated gradient to train a set of neural networks to approximate the expected value of $G_{\tau_{n+1}}$ given X_n. The functions yields as output the following structures

BNN: array of Matlab structures. Each entry in the array stores one structure with the weights **w** and vectors **b** of each layer of the corresponding neural network.

savecost: array with the registered errors of the neural network along the training process. This array can be checked to verify that each neural network has been adequately trained.

Once the function **train_networks.m** has run, the program **main_lower.m** plots the data registered in **savecost** to obtain the error curves of the neural networks during the training process. See the Figure 5.4

The code of **main_lower.m**, **train_networks.m** and **simulate_paths.m** are given in A.2.6.

From these algorithms we estimate lower and upper bounds as well as point estimates and confidence intervals for the optimal value

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

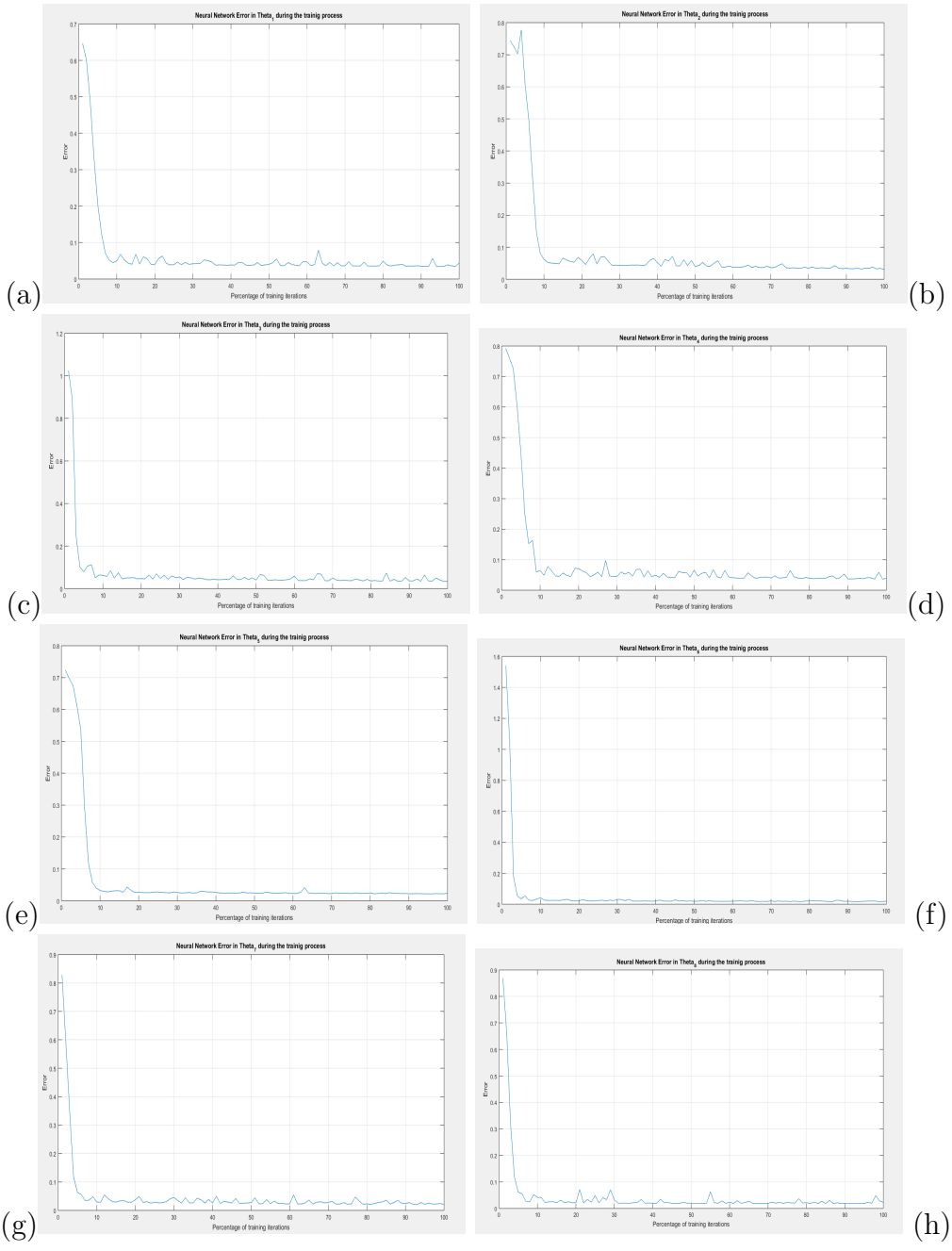

Figure 5.4: *Errors.*

Lower Bound and Upper Bound

Here we use the Remark 4.7 to estimate the Lower Bound.

We consider an American-style option that can be exercised at any one of finitely many times $0 < t_1 < \cdots < t_9 = Tmax$.

We have written a Matlab program named **main.m** to simulate realizations of the underlying process $(X_n)_{n=0}^N$, train a set of neural networks to approximate the expected value of $G_{\tau_{n+1}}$ given X_n and use the trained neural networks to approximate a lower bound L for the optimal value V. The main program uses Matlab functions **simulate_paths.m**, **train_networks.m**.

We have used a data set consisting of $M = 50$ simulated paths of the underlying process $(X_n)_{n=0}^N$ to train a Neural Network with 51 neurons in layer two, 51 neurons in layer three and only one neuron in layer four.

Now we recall that $\Theta = (\theta_0, \ldots, \theta_{N-1})$ where $\theta_0, \theta_1, \ldots, \theta_{N-1}$ are the parameters determined by the variant algorithm of the Longstaff–Schwartz algorithm in section 4.2. In our version we do not use θ_0 and since t_9 is also the expiration date, we train 8 neural networks.

Once the Neural Networks $\theta_1, \theta_2, \ldots, \theta_8$, have been trained and all the weights and bias have been determined, these neural networks can be used to estimate the Lower Bound $\hat{L}$ for the optimal value V. To estimate $\hat{L}$ the program calls Matlab function simulate paths to generate $ML = 10000$ realizations of the underlying process $(X_n)_{n=0}^N$. For each of these paths the program calculates the minimum time n such that

$$g\left(n, X_n\right) \geq c^{\theta_n}\left(X_n\right)$$

That is, we compute τ^Θ. Next, we obtain $y_\Theta = g(\tau^\Theta, X_{\tau^\Theta})$ and the Monte Carlo average

$$\hat{L} = \frac{1}{ML} \sum_{k=M+1}^{M+ML} y_\Theta^k$$

gives an estimate of the lower bound L. Finally, we get an asymptotically valid 95% confidence interval for L.

When the ML realizations of the process have been evaluated, the estimated lower bound $\hat{L}$ is calculated.

Price estimates for max-call options on 1 asset for parameter values of $r = 5\%, \delta = 10\%, \sigma = 20\%, K = 100, Tmax = 3, N = 9$. t_L is the number of seconds it took to train τ^Θ and compute $\hat{L}$ and CI is confidence interval. In the following Table we show the results.

d	S_0	$\hat{L}$	t_L	95% CI
1	90	0.5509642	324	$[0.5230989, +\infty)$
1	100	2.5705165	348	$[2.5168442, +\infty)$
1	110	9.3651444	538	$[9.280237, +\infty)$

Table 5.1

In Becker et al. [2019], it is simulated $ML = 4,096,000$ paths of $(X_n)_{n=0}^N$ to estimate L.

To approximate the upper bound for V we proceed as in the section 4.3. For your estimate of the upper bound U, they produced $MU = 2,048$ paths $\left(x_n^k\right)_{n=0}^N, k = M+ML+1, \ldots, M+ML+MU$, of $(X_n)_{n=0}^N$ and $MU \times J$ realizations $\left(v_n^{k,j}\right)_{n=1}^N, k = 1, \ldots, MU, j = 1, \ldots, J$, of $\left(W_{t_n} - W_{t_{n-1}}\right)_{n=1}^N$ with $J = 2,048$. Then for all n and k, they generated the j-th continuation path departing from x_n^k according to

$$\tilde{x}_m^{k,j} = x_n^k \exp\left(\left[r - \delta - \sigma^2/2\right](m-n)\Delta t + \sigma\left[v_{n+1}^{k,j} + \cdots + v_m^{k,j}\right]\right), \quad m = n+1, \ldots, N$$

Point estimate of V

Once we have estimated $\hat{L}$ and $\widehat{U}$, our point estimate of V is

$$\hat{V} = \frac{\hat{L} + \hat{U}}{2}$$

To obtain high pricing accuracy the neural networks used in the construction of the candidate optimal stopping strategy had to be trained for a longer time and a higher number of nodes should be used. Both strategies are out of our limited computing power. However, the approach yields estimates continuation values and derive a candidate optimal stopping rule. This learned stopping rule can be used to yields a lower biased estimate of the price. All this gives a high biased estimate and confidence interval for the price.

Complexity of the Longstaff-Schwartz algorithm

In this section, we present some factors that can affect the complexity of the Longstaff-Schwartz algorithm. Here we are only going to consider the one-dimensional case as shown previously. Our presentation is similar to the one given in Bernard and

Ahmed. The first factor is the number of M paths of the underlying process. Next, we have considered the sigmoid function as the activation function, but other activation functions could affect the complexity of the algorithm. The choice of the learning rate as a very sensitive parameter is another factor that can influence the complexity of the algorithm. In the Figures 5.2 and 5.3 , we show the progress of the Neural Network error through the training process with a learning rate $eta = 0.05$ and $eta = 0.1$ respectively, where a significant decrease in the number of iterations can be observed with the largest training rate. As is known, advances in optimization algorithms have been an important factor in reducing algorithm complexity. So this is another hugely important factor in reducing complexity. The number the hidden layers in the Neural Network is another factor. Finally, it is important to determine the number of iterations of the neural network to avoid the phenomena of underfitting and overfitting.

We have made some runs of the Neural Network training process setting the number of iterations of the Back Propagation algorithm to $k \times 10^5$ iterations with $1 \leq k \leq 10$. The following table shows the required time in each test.

k	Required training time (s)
1	10.718
2	21.474
3	32.293
4	43.129
5	53.776
6	64.668
7	75.136
8	86.184
9	97.249
10	107.363

The Figure 5.5 shows a plot of time versus number of iterations.

In the previous figure, the linear growth of the time necessary for the training of the neural network is clearly seen.

To total the time in Matlab we use the instructions TIC to start the counter and TOC to stop it. TOC totals the time since the last TIC.

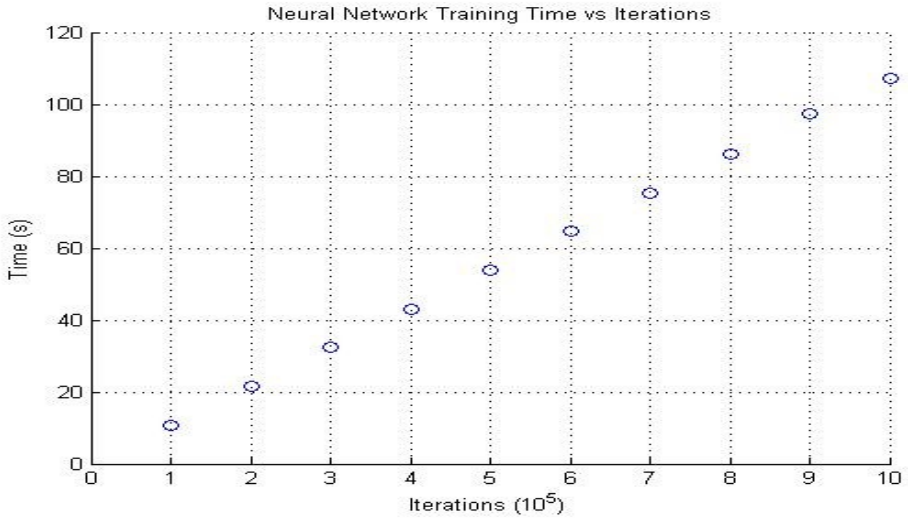

Figure 5.5: Underlying process

Chapter 6

Conclusion

In this thesis, we have studied the problem of pricing of American options assuming that the price of the underlyings are given Markov processes.

We have studied the algorithms of the stochastic gradient descent and that of back propagation.

A variant based on Deep Learning of the Longstaff - Schwartz algorithm was also presented. The proposed algorithm allowed to estimate, starting from the data, a candidate optimal stopping strategy and to estimate lower and upper bounds for the value of the option and confidence intervals for these estimates. The lower and upper bounds computed were used to estimate a point for the value of the option.

Finally, a set of realizations of a geometric Brownian motion (GBM) were generated to simulate the price of an asset and the Deep Learning method used for training Neural Netwoks for the determination of optimal stopping strategy was presented. The sigmoid was used as activation function.

Recommendations

This work can be extended considering d underlying assets to study the multi-dimensional Black-Scholes model. Also, we can the Hedging Until the First Possible Exercise Date computing the Average hedging errors and the empirical hedging short-falls using (4.7), (4.8) and the Hedging Until the Exercise Time with (4.9), (4.10) to study the quality of the hedge.

Finally, it is proposed to use the methodology given in Herrera et al. [2021] where

the parameters of the hidden layers are generated randomly and only the last layer is trained to approximate the solutions of optimal stopping problems,

Bibliography

Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Pricing and hedging american-style options with deep learning. *Journal of Risk and Financial Management*, 13(7):158, 2020.

Francis Longstaff and Eduardo Schwartz. Valuing american options by simulation: a simple least-squares approach. *The review of financial studies*, 14(1):113–147, 2001.

Michael Kohler, Adam Krzyżak, and Nebojsa Todorovic. Pricing of high-dimensional american options by neural networks. *Mathematical Finance: An International Journal of Mathematics, Statistics and Financial Economics*, 20(3):383–410, 2010.

Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375: 1339–1364, 2018.

Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Deep optimal stopping. *Journal of Machine Learning Research*, 20:74, 2019.

Sebastian Becker, Patrick Cheridito, Arnulf Jentzen, and Timo Welti. Solving high-dimensional optimal stopping problems using deep learning. *European Journal of Applied Mathematics*, 32(3):470–514, 2021.

Bernard Bru and Henri Heinich. Meilleures approximations et médianes conditionnelles. In *Annales de l'IHP Probabilités et statistiques*, volume 21, pages 197–224, 1985.

William Gustafsson. Evaluating the longstaff-schwartz method for pricing of american options, 2015.

Bernt Oksendal. *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

Sheldon M Ross. *Simulation*. academic press, 2022.

Filip Šoljić. Sandro skansi: Introduction to deep learning. from logical calculus to artificial intelligence. *Synthesis philosophica*, 34(2):477–479, 2019.

James Gareth, Witten Daniela, Hastie Trevor, and Tibshirani Robert. *An introduction to statistical learning: with applications in R*. Spinger, 2013.

Catherine Higham and Desmond Higham. Deep learning: An introduction for applied mathematicians. *Siam review*, 61(4):860–891, 2019.

M LAPEYRE Bernard and M KEBAIER Ahmed. Pricing of american options using neural networks.

Calypso Herrera, Florian Krach, Pierre Ruyssen, and Josef Teichmann. Optimal stopping via randomized neural networks. *arXiv preprint arXiv:2104.13669*, 2021.

Appendix A

The code

A.1 The underling $\{S_n\}_{n=0}^{N}$

```
Algorithm

Let N=270 , dt=1/N;$, t=[dt:dt:3] ;
r=r_0, delta=delta_0,
M=50;  % path simultaneus}
dW = sqrt(dt)*randn(M,N);      % increments
W = cumsum(dW,2); % cumulative sum
S = S_0exp((r-delta)*repmat(t,[M 1]) - (sigma/2)*W);
Umean = mean(U);
plot([0,t],[1,Umean],'b-'), hold on % plot mean over M paths
plot([0,t],[ones(5,1),U(1:5,:)],'r--'), hold off % plot 5 individual paths
xlabel('t','FontSize',16)
ylabel('S(t)','FontSize',16,'Rotation',0,'HorizontalAlignment','right')
legend('mean of 50 paths','5individual paths',2)
%averr = norm((Umean - exp(9*t/8)),'inf') % sample error
```

A.2 The code to deep learning

A.2.1 Simulating the underlying process

```
tmax = 3; % Total simulation time
N_intervals = 270; % Number of subintervals
dt = tmax/N_intervals; % Time step between two consecutive samples
```

```
t = 0:dt:tmax; % Array of time instants
r = 0.05; % Risk-free interest rate
delta = 0.1; % Dividend yields
S_0 = 90; % Initial value of S at t=0
sigma = 0.2; % Volatility
M = 50; % Number of paths to be simulated
K = 100; % Strike price
dW = sqrt(dt)*randn(M,N_intervals+1);
W = cumsum(dW,2);
S = S_0*exp((r-delta)*repmat(t,[M 1]) - (sigma/2)*W);
g = max(0,S-K);
```

A.2.2 Scaling the training input data

```
scale_g = 1/(1.1*max(max(g)));
g = g*scale_g;
scale_S = 1/(1.1*max(max(S)));
S = S*scale_S;
```

A.2.3 Neural Network parameters

```
% Set Neural Network parameters
neurons_in_layer_2 = 50;
neurons_in_layer_3 = 50;
neurons_in_layer_4 = 1;
```

A.2.4 Training the Networks

```
% Randomly setting of neurons weights and bias in each layer
W2 = 0.5*randn(neurons_in_layer_2,1);
b2 = 0.5*randn(neurons_in_layer_2,1);
W3 = 0.5*randn(neurons_in_layer_3,neurons_in_layer_2);
b3 = 0.5*randn(neurons_in_layer_3,1);
W4 = 0.5*randn(neurons_in_layer_4,neurons_in_layer_3);
b4 = 0.5*randn(neurons_in_layer_4,1);
% Select the n-th simulated data set of the underlying process S and
% the corresponding (n+1)-th simulated expectation of G(Sn) and use
% them to train the Neural Network
```

```
y = g(:,N_intervals);
xx = S(:,N_intervals-1);
% Set Neural Network training parameters
eta = 0.05; % Learning Rate
Niter = 1e6; % Number of Stochastic Gradient iterations
savecost = zeros(100,1); % value of cost function at each iteration
% Initiate the training loop for a specific Neural Network
i_to_plot = 0;
point_to_plot = 0;
for counter = 1:Niter
    k = randi(M); % Randomly choose a training point
    x = xx(k);
    % Forward pass
    a2 = 1./(1+exp(-(W2*x+b2)));
    a3 = 1./(1+exp(-(W3*a2+b3)));
    a4 = 1./(1+exp(-(W4*a3+b4)));
    % Backward pass
    delta4 = a4.*(1-a4).*(a4-y(k));
    delta3 = a3.*(1-a3).*(W4'*delta4);
    delta2 = a2.*(1-a2).*(W3'*delta3);
    % Gradient step
    W2 = W2 - eta*delta2*x';
    W3 = W3 - eta*delta3*a2';
    W4 = W4 - eta*delta4*a3';
    b2 = b2 - eta*delta2;
    b3 = b3 - eta*delta3;
    b4 = b4 - eta*delta4;
end
```

A.2.5 Monitoring the training process

```
 % Monitor progress
    i_to_plot = i_to_plot + 1;
    if (i_to_plot == Niter_to_plot)
        point_to_plot = point_to_plot + 1;
        i_to_plot = 0;
        costvec = zeros(M,1);
        for j = 1:M
```

```
            a2 = 1./(1+exp(-(W2*xx(j)+b2)));
            a3 = 1./(1+exp(-(W3*a2+b3)));
            a4 = 1./(1+exp(-(W4*a3+b4)));
            costvec(j) = norm(y(j) - a4,2);
        end
        savecost(point_to_plot) = norm(costvec,2)^2;
    end
```

A.2.6 Train a set of neural networks

```
% main program
% close all figures, removes all variables from the current workspace and
% clears all the text from the Command Window clearing the screen.
close all
clear
clc

% Set parameters for simulated data paths generation
tmax = 3; % Total simulation time
N_intervals = 270; % Number of subintervals
r = 0.05; % Risk-free interest rate
delta = 0.1; % Dividend yields
S_0 = 90; % Initial value of S at t=0
sigma = 0.2; % Volatility
M = 50; % Number of paths to be simulated
K = 100; % Strike price

[t,S,g] = simulate_paths(tmax,N_intervals,r,delta,S_0,sigma,M,K);

% Set Neural Network parameters
neurons_in_layer_2 = 2;
n2 = neurons_in_layer_2;
neurons_in_layer_3 = 3;
n3 = neurons_in_layer_3;
neurons_in_layer_4 = 1;
n4 = neurons_in_layer_4;

% Set Neural Network training parameters
```

```
eta = 0.15; % Learning Rate
Niter = 1e6; % Number of Stochastic Gradient iterations
Niter_to_plot = 1e4; % Number of iterations between progress evaluations

tic
[BNN,savecost] = train_network(n2,n3,n4,S,g,eta,Niter,Niter_to_plot);
toc

% Generate ML simulated paths of the process to use them to calculate the
% lower bound.
ML = 50; % Number of path to calculate the lower bound
% simulate ML paths to be used to calculate the lower bound
[t,S,g] = simulate_paths(tmax,N_intervals,r,delta,S_0,sigma,ML,K);
delta = 30;
% Use the trained Neural Network to predict the minimu
Lsum = 0;
% frequency = zeros(9,1);
lower_g = zeros(1:ML);
for i = 1:ML
    % Read the data to estimate lower bound
    St = S(i,delta:delta:270);
    gt = g(i,delta:delta:270);
    j = 1;
    % Set the default time to sell at the maximum time
    lower_g(i) = g(9);
    for j = 1:8
        NN = BNN(j); % Read the j-th Neural Network from the array BNN
        W2 = NN.W2; % Store weights and bias of the Neural Network
        b2 = NN.b2;
        W3 = NN.W3;
        b3 = NN.b3;
        W4 = NN.W4;
        b4 = NN.b4;
    % Calculate Neural Network output with the j-th sample of St as input
        a2 = 1./(1+exp(-(W2*St(j) + b2)));
        a3 = 1./(1+exp(-(W3*a2+b3)));
        a4 = 1./(1+exp(-(W4*a3+b4))); % a4 is the Neural Network output
        if(gt(j) >= a4)
```

```
            lower_g(i) = gt(j);
            break
        end
    end
    % frequency(minimum_time) = frequency(minimum_time)+1;
    % Lsum = Lsum + minimum_time;
end
lower_bound = sum(lower_g)/ML;
standard_deviation = sqrt((sum((lower_g - lower_bound).^2))/(ML-1));
% fprintf('Minimum Stopping Time Frequencies.\n')
% for i = 1:9
%     fprintf('%.0f = %.0f   ',i,frequency(i))
% end
fprintf('\nThe calculated Lower Bound for the optimal value V is %.7f\n',lower_bound)
fprintf('\nThe Standard deviation of the calculated Lower Bound is %.7f\n',standard_d

for i = 1:8
    figure (i)
    % figure(3)
    plot(1:length(savecost(1,:)),savecost(i,:))
    xlabel('Percentage of training iterations')
    ylabel('Error')
    title(sprintf('Neural Network Error in Theta_%d during the trainig process',i))
    grid on
end



    function [t,S,g] = simulate_paths(tmax,N_intervals,r,delta,S_0,sigma,M,K)
dt = tmax/N_intervals; % Time step between two consecutive samples
t = 0:dt:tmax; % Array of time instants
dW = sqrt(dt)*randn(M,N_intervals+1);
W = cumsum(dW,2);
S = S_0*exp((r-delta)*repmat(t,[M 1]) - (sigma/2)*W);
g = max(0,S-K);
end

    function [BNN,savecost] = train_network(neurons_2,neurons_3,neurons_4,S,g,eta,Nit
% scale input data to fit into the interval [0,1]
```

```
scale_g = 1/(1.1*max(max(g)));
g = g*scale_g;
scale_S = 1/(1.1*max(max(S)));
S = S*scale_S;

[rows,columns] = size(S);
M = rows;
N_intervals = columns - 1;

savecost = zeros(9,100); % value of cost function at each iteration

% Enter a loop for training Neural Networks to approximate the conditional
% expectations of G(xn). There should be a trained Neural Network for each tn.
for network_index = 8:-1:1

% Randomly initialize each neural network setting neurons weights and bias
W2 = 0.5*randn(neurons_2,1);
b2 = 0.5*randn(neurons_2,1);
W3 = 0.5*randn(neurons_3,neurons_2);
b3 = 0.5*randn(neurons_3,1);
W4 = 0.5*randn(neurons_4,neurons_3);
b4 = 0.5*randn(neurons_4,1);

% Select the n-th simulated data set of the underlying process S and
% the corresponding (n+1)-th simulated expectation of G(Sn) and use
% them to train the Neural Network
delta_data = N_intervals/9;
y = g(:,(network_index+1)*delta_data);
% xx = S(:,network_index*delta_data);
xx = S(:,(network_index+1)*delta_data-1);


    % Initiate the training loop for a specific Neural Network



    i_to_plot = 0;
    point_to_plot = 0;
```

```
for counter = 1:Niter
    k = randi(M); % Randomly choose a training point
    x = xx(k);
    % Forward pass
    a2 = 1./(1+exp(-(W2*x  + b2)));
    a3 = 1./(1+exp(-(W3*a2 + b3)));
    a4 = 1./(1+exp(-(W4*a3 + b4)));

    % Backward pass
    delta4 = a4.*(1-a4).*(a4-y(k));
    delta3 = a3.*(1-a3).*(W4'*delta4);
    delta2 = a2.*(1-a2).*(W3'*delta3);

    % Gradient step
    W2 = W2 - eta*delta2*x';
    W3 = W3 - eta*delta3*a2';
    W4 = W4 - eta*delta4*a3';
    b2 = b2 - eta*delta2;
    b3 = b3 - eta*delta3;
    b4 = b4 - eta*delta4;



    % Monitor progress
    i_to_plot = i_to_plot + 1;
    if (i_to_plot == Niter_to_plot)
        point_to_plot = point_to_plot + 1;
        i_to_plot = 0;
        costvec = zeros(M,1);
        for j = 1:M
            a2 = 1./(1+exp(-(W2*xx(j) + b2)));
            a3 = 1./(1+exp(-(W3*a2+b3)));
            a4 = 1./(1+exp(-(W4*a3+b4)));
            costvec(j) = norm(y(j) - a4,2);
        end
        savecost(network_index,point_to_plot) = norm(costvec,2)^2;
    end
end
```

```
    NN.W2 = W2;
    NN.b2 = b2;
    NN.W3 = W3;
    NN.b3 = b3;
    NN.W4 = W4;
    NN.b4 = b4;
    BNN(network_index) = NN;

end


% figure (2)
% semilogy(1:point_to_plot,savecost(1:point_to_plot))
% xlabel('Percentage of training iterations')
% ylabel('Log(error)')
% title('Neural Network Error in logaritmic scale during the trainig process')
% figure(3)
% plot(1:point_to_plot,savecost(1:point_to_plot))
% xlabel('Percentage of training iterations')
% ylabel('Error')
% title('Neural Network Error during the trainig process')

end
```

Appendix B

Mathematical Preliminaries

B.1 Probability Spaces

Definition B.1. *A probability space is a measure space with total measure one. The standard notation is* $(\Omega, \mathcal{F}, \mathbb{P})$ *where:*

- Ω *is a set (sometimes called a sample space in elementary probability). Elements of* Ω *are denoted* ω *and are sometimes called outcomes.*
- $\mathcal{F}$ *is a* σ*-algebra (or* σ*-field, we will use these terms synonymously) of subsets of* Ω*. Sets in* $\mathcal{F}$ *are called events.*
- $\mathbb{P}$ *is a function from* $\mathcal{F}$ *to* $[0,1]$ *with* $\mathbb{P}(\Omega) = 1$ *and such that if* $E_1, E_2, \ldots \in \mathcal{F}$ *are disjoint,*

$$\mathbb{P}\left[\bigcup_{j=1}^{\infty} E_j\right] = \sum_{j=1}^{\infty} \mathbb{P}\left[E_j\right]$$

We say "probability of E" for $\mathbb{P}(E)$. A discrete probability space is a probability space such that Ω is finite or countably infinite. In this case we usually choose $\mathcal{F}$ to be all the subsets of Ω (this can be written $\mathcal{F} = 2^{\Omega}$), and the probability measure $\mathbb{P}$ is given by a function $p : \Omega \to [0,1]$ with $\sum_{\omega\in\Omega} p(\omega) = 1$. Then,

$$\mathbb{P}(E) = \sum_{\omega\in E} p(\omega).$$

B.2 Random Variables and Random Vectors

Definition B.2. *A random variable X is a measurable function from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ to the reals , i.e., it is a function*

$$X : \Omega \longrightarrow (-\infty, \infty)$$

such that for every Borel set B,

$$X^{-1}(B) = \{X \in B\} \in \mathcal{F}$$

Here we use the shorthand notation

$$\{X \in B\} = \{\omega \in \Omega : X(\omega) \in B\}.$$

If X is a random variable, then for every Borel subset B of $\mathbb{R}, X^{-1}(B) \in \mathcal{F}$. We can define a function on Borel sets by

$$\mu_X(B) = \mathbb{P}\{X \in B\} = \mathbb{P}\left[X^{-1}(B)\right]$$

This function is in fact a measure, and $(\mathbb{R}, \mathcal{B}, \mu_X)$ is a probability space. The measure μ_X is called the distribution of the random variable. If μ_X gives measure one to a countable set of reals, then X is called a discrete random variable. If μ_X gives zero measure to every singleton set, and hence to every countable set, X is called a continuous random variable. Every random variable can be written as a sum of a discrete random variable and a continuous random variable. All random variables defined on a discrete probability space are discrete.

B.3 Conditional expectation

Definition B.3. *The conditional expectation $E[Y \mid \mathcal{F}_n]$ is the unique random variable satisfying the following.*

- *$E[Y \mid \mathcal{F}_n]$ is $\mathcal{F}_n$-measurable.*
- *For every $\mathcal{F}_n$-measurable event A,*

$$\mathbb{E}\left[E[Y \mid \mathcal{F}_n] 1_A\right] = \mathbb{E}\left[Y 1_A\right]$$

B.4 Stochastic Processes

Definition B.4. *$(X_n)_{n=0}^N$ a d-dimensional $\mathbb{F}$-Markov process if X_n is $\mathcal{F}_n$-measurable, and $\mathbb{E}[f(X_{n+1}) \mid \mathcal{F}_n] = \mathbb{E}[f(X_{n+1}) \mid X_n]$ for all $n \leq N-1$ and every measurable function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f(X_{n+1})$ is integrable.*

B.5 Martingale and Supermartingale

Suppose we have a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and an increasing sequence of σ-algebras

$$\mathcal{F}_0 \subset \mathcal{F}_1 \subset \dots$$

Such a sequence is called a **filtration**, and we think of $\mathcal{F}_n$ as representing the amount of knowledge we have at time n. The inclusion of the σ-algebras imply that we do not lose any knowledge that we have gained.

Definition B.5. *A martingale (with respect to the filtration $\{\mathcal{F}_n\}$) is a sequence of integrable random variables M_n such that M_n is $\mathcal{F}_n$-measurable for each n and for all $n < m$,*

$$\mathbb{E}[M_m \mid \mathcal{F}_n] = M_n.$$

We note that to prove the formula, it suffices to show that for all n,

$$\mathbb{E}[M_{n+1} \mid \mathcal{F}_n] = M_n.$$

Example B.6.

- *If $X_1, X_2, \dots$ are independent, mean-zero random variables and $S_n = X_1 + \dots + X_n$, then S_n is a martingale with respect to the filtration generated by $X_1, X_2, \dots$*
- *If $X_1, X_2, \dots$ are as above, $\mathbb{E}\left[X_j^2\right] = \sigma_j^2 < \infty$ for each j, then*

$$M_n = S_n^2 - \sum_{j=1}^{n} \sigma_j^2$$

is a martingale with respect to the filtration generated by $X_1, X_2, \dots$.

B.6 Stopping Times

Definition B.7. *A stopping time with respect to a filtration $\{\mathcal{F}_n\}$ is a random variable taking values in $\{0, 1, 2 \dots\} \cup \{\infty\}$ such that for each n, the event*

$$\{T \leq n\} \in \mathcal{F}_n.$$

In other words, we only need the information at time n to know whether we have stopped at time n.

Example B.8.

- *Constant random variables are stopping times.*
- *If M_n is a martingale with respect to $\{\mathcal{F}_n\}$ and $V \subset \mathbb{R}$ is a Borel set, then*

$$T = \min\{j : M_j \in V\}$$

  is a stopping time.

- *If T_1, T_2 are stopping times, then so are $T_1 \wedge T_2$ and $T_1 \vee T_2$.*
- *In particular, if T is a stopping time then so are $T_n = T \wedge n$. Note that $T_0 \leq T_1 \leq \ldots$ and $T = \lim T_n$.*

Theorem B.9. *If M_n is a martingale and T is a stopping time with respect to $\{\mathcal{F}_n\}$, then $Y_n = M_{T \wedge n}$ is a martingale with respect to $\{\mathcal{F}_n\}$. In particular, $\mathbb{E}[Y_n] = \mathbb{E}[Y_0] = \mathbb{E}[M_0]$.*

We denote by $\mathcal{T}$ the set of all $\mathbb{F}$-stopping times $\tau : \Omega \to \{0, 1, \cdots, N\}$.

B.7 Optimal stopping

The theory of optimal stopping is concerned with the problem of choosing a time to take a particular action, in order to maximise an expected reward or minimise an expected cost. Optimal stopping problems can be found in areas of statistics, economics, and mathematical finance (related to the pricing of American options). A key example of an optimal stopping problem is the secretary problem. Optimal stopping problems can often be written in the form of a Bellman equation, and are therefore often solved using dynamic programming.

B.8 Discrete time case

Stopping rule problems are associated with two objects:

1. A sequence of random variables $X_1, X_2, \cdots$, whose joint distribution is something assumed to be known
2. A sequence of "reward" functions $(y_i)_{i \geq 1}$ which depend on the observed values of the random variables in 1: $y_i = y_i(x_1, \ldots, x_i)$

Given those objects, the problem is as follows:

- You are observing the sequence of random variables, and at each step i, you can choose to either stop observing or continue.
- If you stop observing at step i, you will receive reward y_i.
- You want to choose a stopping rule to maximize your expected reward (or equivalently, minimize your expected loss).