yuan_junwei_01045395.pdf

by Junwei Yuan

Submission date: 05-Sep-2022 10:07PM (UTC+0100)
Submission ID: 185689989
File name: yuan_junwei_01045395.pdf (4.6M)
Word count: 21460
Character count: 88899

Imperial College London

Imperial College London

Department of Mathematics

Deep learning for unconstrained Markov regime-switching quadratic utility maximisation

Author: Junwei Yuan (CID: 01045395)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2021-2022

Declaration

I certify that the work contained in this thesis is my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Acknowledge

I would like to express my sincere gratitude towards my supervisor Professor Zheng for his time and effort spent in tutoring me, and for his valuable guidance throughout this project.

I would also like to thank my parents and my girlfriend Feilun for their support during my study.

Abstract

Empirically, economists believe the economy has different regimes such as growth, recession, recovery and depression. Each regime has its own market dynamics and the change from one regime to another is typically abrupt. Continuous Markov chain is found to be suited for modelling such change. As a result, Markov regime-switching stochastic control problems have gained increasing interest in both academia and the industry. However, it is often intractable to find analytical solutions for stochastic control problems even without regime switching. This motivates the development of numerical methods.

In this study, we will focus on unconstrained regime-switching quadratic utility maximisation problem. We will develop numerical methods for this particular regime-switching problem based on two deep learning algorithms originally designed for standard utility maximisation problems. Moreover, as this problem does has an anlytical solution, we will then present a numerical example and compare the numerical solutions against the analytical solutions.

Contents

Chapter 1

Introduction

1.1 Background

Stochastic control theory provides a mathematical framework for finding the optimal control within dynamical systems driven by observable uncertainty. In particular, in a market where asset prices are partly driven by random noises, utility maximisation studies how an investor can maximise her utility (happiness) by investing her capital optimally. Besides the return for capital, various factors can also impact the utility such as risks involved in producing the return. Solving this problem is of great importance and has many applications in the finance industry.

Markov regime-switching utility maximisation extends the standard utility maximisation by considering the market regime. Empirically, economists and investors believe the market has separate states such as growth, recession, recovery and depression. Each state behaves markedly different and this difference is manifested in market volatility, interest rate, growth rate, etc. The move between the states are typically abrupt, hence it can be modelled using continuous-time Markov chain over finite discrete state space. This combination of continuous stochastic diffusion process and continuous finite-state Markov process has gained great deal of traction since its first usage in modelling macro economic event in 1970. In recent years, regime-switching stochastic control problems have gained ever-increasing attention for its additional capacity.

There are by far two mainstream approaches tackling stochastic control problems, i.e. Hamilton–Jacobi–Bellman (HJB) equation and Stochastic Maximisation Principle (SMP). The former is derived from Bellman's dynamic principle [2] which states that the optimal control over entire horizon must contain the optimal control for any sub-interval within the horizon. By zooming into infinitesimally small interval, we then reduces the problem into solving a second order partial differential equation, which is the called Hamilton–Jacobi–Bellman (HJB) equation. On the other hand, SMP extends Pontryagin's maximisation principle which was originally introduced for classical dynamic system in 1956. SMP requires that any optimal control must maximise the Hamiltonian control function.

However, HJB equation is highly non-linear which is intractable in most cases. For the solvable cases, the problem setting is too constrained to be applicable for real-world applications. Achieving the optimality condition for SMP is also a daunting task. The difficulty in obtaining analytical solutions then motivates the development of numerical methods. While there are various algorithms for standard control problems, there is a clear lack of numerical methods for regime-switching control problems despite its prospect and importance.

Deep learning is the most fast-growing discipline in computer science within the recent 10 years. Although a lot of the fundamental results were proven back in the 20th century, it was not computationally feasible at the time. It was the development of modern computing hardware in the 21st century that finally put the theories into real applications. In particular, the development in hardware such as graphics processing unit (GPU) has greatly enhanced our computation capability.

Probably the most famous application of deep neural network nowadays is in computer vision

where convolutional neural network, a kind of deep neural network, is used to micmic human eyes such as to identify objects in the pictures. Under the hood, we can treat the human eyes as some function of pixels in the picture, and neural networks are merely approximating that function. This power of approximation was rigorously shown in the Universal Approximation Theorem [7] which states that a feedforward neural network can approximate arbitrary continuous function to any desired degree of accuracy provided that there are sufficient amount of hidden nodes. Therefore, we would like to leverage neural networks' approximation capacity in finding a numerical solution to stochastic control problems.

1.2 Current research

There are previous research work in solving the quadratic utility maximisation problem without regime-switching. Bian [3] studied the problem using HJB, while Li [12] approached the problem with SMP. As for the extended problem with regime-switching, Krishnakumar [10] performed an extensive study and solved the problem with both HJB and SMP. Krishnakumar also derived the dual problem and demonstrated the equivalence between primal and dual solutions. There are also research in more general Markov regime-switching control problems. Li [11] extended SMP to regime-switching problems and introduced the weak necessary and sufficient conditions. Azevedo [1] employed the HJB equation to solve regime-switching control problems.

In terms of numerical methods, Ashley [4] introduced two deep learning based numerical methods for standard utility maximisation problems. To the best of our knowledge, there is no deep learning based numerical methods in the literature for Markov regime-switching control problems.

1.3 Aim and outline

In this thesis, we will study the unconstrained Markov regime-swtiching quadratic utility maximisation. Krishnakumar [10] has solved this problem analytically by both HJB and SMP. He has also proved the equivalence between primal and dual solutions. However, his report did not provide any numerical examples. One of the contribution of this study is to verify Krishnakumar's theoretical results with numerical examples.

The paper [4] introduces two novel deep learning based numerical methods for standard utility maximisation problems. The major contribution of this study is to extend both algorithms and design numerical methods for Markov regime-switching quadratic utility maximisation. We will then assess the numerical methods by comparing the numerical solutions against analytical solutions from Krishnakumar's report [10]. It is worth noting that although our particular regime-switching problem has analytical solutions, our numerical algorithms can be extended with relative ease to solve problems without such analytical solutions.

The remainder of this thesis is outlined as follows. Chapter 2, 3, 4 will cover preliminary knowledge that are already known. In chapter 2, we will introduce general stochastic control problem and the standard utility maximisation problem. We will also briefly mention HJB and SMP. In chapter 3, we will present the unconstrained Markov regime-switching quadratic utility maximisation problem along with any theoretical result from [10]. In chapter 4, we will introduce the two deep learning based numerical methods from [4]. In chapter 5, we will present the main results of this thesis, that is, the numerical methods for unconstrained Markov regime-switching quadratic utility maximisation problem. In chapter 6, we will present a numerical example and compute the analytical solutions. We will then verify any theoretical results and compare numerical solutions with the analytical solutions. Finally, chapter 7 concludes the thesis.

Chapter 2

Stochastic control and utility maximisation

2.1 Stochastic control problem

In this section, we introduce the general settings for stochastic control problems. A stochastic control problem is a control problem where the underlying state X is a stochastic process. Without loss of generality, we may assume X is valued on $\mathbb{R}$. We also assume the problem has finite time horizon [0, T] where T is the terminal time. The process X follows the dynamics:

$$\mathrm{d}X_t = b(t, X_t, \pi_t)\mathrm{d}t + \sigma(t, X_t, \pi_t)\mathrm{d}W_t, \quad X_0 = x_0 \tag{2.1.1}$$

where W is a n-dimensional Wiener process. The function b and σ are valued in $\mathbb{R}$ and $\mathbb{R}^n$ respectively and they satisfy some linear growth conditions.

The process π is called the control process whose value is in some set A. Let us denote the natural filtration of W by $(\mathcal{F}_t)_{t\in[0,T]}$ and control process π is progressively measurable with respect to $\mathcal{F}$. Normally, there are some criteria for a control process to be admissible. We denote the set of all admissible controls as $\mathcal{A}$.

Our goal is to maximise the gain function J over $\mathcal{A}$ where J is defined as:

$$J(\pi) := \mathbb{E}\left[\int_0^T f(t, X_t, \pi_t)\mathrm{d}t + U(X_T)\right].$$

The function $f : [0, T] \times \mathbb{R} \times A \to \mathbb{R}$ is called the running cost and $U : \mathbb{R} \to \mathbb{R}$ the terminal gain. In the following, we will look at two mainstream methods for solving stochastic control problems, i.e. HJB and SMP.

2.1.1 Hamilton–Jacobi–Bellman equation

The Hamilton-Jacobi-Bellman (HJB) equation is developed based on Bellman's dynamic programming principle (DPP) [2]. Intuitively, DPP states that a control between time $[t, T]$ is optimal if and only if it is optimal within interval $[t, t + h)$ and $[t + h, T]$ for any $h > 0$. In rigid terms, define the value function $u : [0, T] \times \mathbb{R} \to \mathbb{R}$:

$$u(t, x) := \sup_{\pi} \mathbb{E}\left[\int_t^T f(t, X_s, \pi_s)\mathrm{d}s + U(X_T) \mid X_t = x\right].$$

The value function represents the expected utility achieved by the optimal control when starting at time t with initial state x. Note that $J(\pi) = u(0, x_0)$ by construction. Then DPP states that,

for any $h > 0$, the following holds:

$$u(t,x) = \sup_{\pi \in \mathcal{A}} \mathbb{E}\left[\int_t^{t+h} f(t, X_s, \pi_s)\mathrm{d}s + u(t+h, X_{t+h}) \mid X_t = x\right] \tag{2.1.2}$$

where π here is the control between time $[t, t+h)$.

Next, we derive the HJB equation. Applying Ito's formula to $u(t+h, X_{t+h})$ and expanding $\mathrm{d}X_s$ and $\mathrm{d}[X]_s$, we have

$$\begin{aligned} u(t+h, X_{t+h}) - u(t, X_t) &= \int_t^{t+h} \partial_t u(s, X_s)\mathrm{d}s + \int_t^{t+h} \partial_x u(s, X_s)\mathrm{d}X_s + \frac{1}{2}\int_t^{t+h} \partial_{xx} u(s, X_s)\mathrm{d}[X]_s \\ &= \int_t^{t+h} (\partial_t u) + (\partial_x u) b_s + \frac{1}{2}(\partial_{xx} u)\sigma_s^\top \sigma_s \mathrm{d}s + \int_t^{t+h} (\partial_x u)\sigma_s \mathrm{d}W_s \end{aligned}$$

where $b_s := b(s, X_s, \pi_s)$ and $\sigma_s := \sigma(s, X_s, \pi_s)$. Then substituting into (2.1.2). As W is a brownian motion, $\mathrm{d}W_s$ term vanishes within the expectation, so equation (2.1.2) beomes:

$$0 = \sup_{\pi} \mathbb{E}\left[\int_t^{t+h} f(t, X_s, \pi_s) + (\partial_t u) + (\partial_x u) b_s + \frac{1}{2}(\partial_{xx} u)\sigma_s^\top \sigma_s \mathrm{d}s\right].$$

Taking $h \to 0$, so $X_s \to X_t = x$, $\pi_s \to \pi_t = a$, we arrive at:

$$0 = \sup_{a \in A} \mathbb{E}\left[f(t,x,a) + \partial_t u(t,x) + \partial_x u(t,x) b(t,x,a) + \frac{1}{2}\partial_{xx} u(t,x)\sigma(t,x,a)^\top \sigma(t,x,a)\right]$$

Finally, we assume that f, b and σ are deterministic functions of (t,x,a), so we can drop the expectation. We arrive at

$$0 = \partial_t u(t,x) + \sup_{a \in A}\left\{f(t,x,a) + \partial_x u(t,x) b(t,x,a) + \frac{1}{2}\partial_{xx} u(t,x)\sigma(t,x,a)^\top \sigma(t,x,a)\right\}.$$

The final equation is called the HJB equation. Typically, in order to solve this equation, we need to guess an ansatz for u and substitute into HJB equation. However, there is no systematic way to guess the ansatz except when terminal gain U is of form x^p for some p.

We can also define the Hamiltonian control $F : [0,T] \times \mathbb{R} \times A \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$:

$$F(t,x,a,z,\gamma) := f(t,x,a) + b(t,x,a)z + \frac{1}{2}\sigma(t,x,a)^\top \sigma(t,x,a)\gamma.$$

The HJB equation can then be rewritten as

$$0 = \partial_t u(t,x) + \sup_{a \in A}\left\{F\left(t,x,a,\partial_x u(t,x), \partial_{xx} u(t,x)\right)\right\}.$$

2.1.2 Stochastic maximisation principle

Pontryagin's maximisation principle was originally introduced for classical dynamic system in 1956. It states that any optimal control to a dynamic system must satisfy two conditions: the associated state must solve the Hamiltonian system; and the optimal control must maximises the Hamiltonian control function. This idea was later extended for stochastic control problems. [13, Section 6.4.2, page 159] provides a detailed explanation of this topic. Here, we recall the main theorem.

Consider the settings in section 2.1. We can define the generalised Hamiltonian $\mathcal{H} : [0,T] \times \mathbb{R} \times A \times \mathbb{R} \times \mathbb{R}^n$:

$$\mathcal{H}(t,x,a,y,z) := b(t,x,a)y + \sigma(t,x,a)^\top z + f(t,x,a). \tag{2.1.3}$$

We further assume that $\mathcal{H}$ is differentiable with respect to x and denote the differential as $\mathcal{D}_x\mathcal{H}$. Moreover, for each $\pi \in \mathcal{A}$, we have the so-called adjoint processes (Y,Z) which satisfy the adjoint BSDE equation:

$$-\mathrm{d}Y_t = \mathcal{D}_x\mathcal{H}(t, X_t, \pi_t, Y_t, Z_t)\mathrm{d}t - Z_t \mathrm{d}W_t, \quad Y_T = \mathcal{D}_x U(X_T). \tag{2.1.4}$$

Theorem 2.1.1. *Let $\hat{\pi} \in \mathcal{A}$ and $\hat{X}$ be the associated controlled diffusion as* (2.1.1)*. Suppose that there exists a solution $(\hat{Y}, \hat{Z})$ to the associated BSDE* (2.1.4) *such that*

$$\mathcal{H}(t, \hat{X}_t, \hat{\pi}_t, \hat{Y}_t, \hat{Z}_t) = \max_{a \in A} \mathcal{H}(t, \hat{X}_t, a, \hat{Y}_t, \hat{Z}_t)$$

for all $t \in [0, T]$ a.s. and

$$(x, a) \to \mathcal{H}(t, x, a, \hat{Y}_t, \hat{Z}_t) \text{ is a concave function for all } t \in [0, T].$$

Then $\hat{\pi}$ is an optimal control, i.e.

$$J(\hat{\pi}) = \sup_{\pi \in \mathcal{A}} J(\pi).$$

It is worth pointing out that theorem (2.1.1) does not require function b and σ to be deterministic functions of X_t and π_t, while HJB does. This means SMP can be applied to a broader set of control problems than HJB from this perspective. On the other hand, it does impose additional constraint such as the optimality of the generalised Hamiltonian (2.1.3).

2.2 Utility maximisation problem

2.2.1 Primal problem

Let $(\Omega, \mathcal{A}, \mathbb{P})$ be the probability space. We consider a market with $n+1$ financial instruments. In particular, S_0 denotes the bank account with interest rate r, and $S_1, \ldots, S_n$ denote prices of n stocks. For clarity, we will write the value of process S_j at time t as $S_j(t)$. We assume stock prices are driven by some Brownian motions W_j and follow the dynamics:

$$\mathrm{d}S_j(t) = \mu_j(t) S_j(t) \mathrm{d}t + \sum_{k=1}^{n} \sigma_{jk}(t) S_j(t) \mathrm{d}W_j(t) \tag{2.2.1}$$

where $\mu : [0, T] \to \mathbb{R}^n$, $\sigma_{ij} : [0, T] \to \mathbb{R}^{n \times n}$ denote growth rate and volatility respectively. Note that W_i, W_j are independent and σ_{ij} captures the correlations between stock i and j. Moreover, we might not assume the functions r, σ and μ to be deterministic.

The question is how one can invest in this market with initial capital x_0 and achieve maximum utility of the wealth. Denote the portfolio, i.e. the quantity of different stocks held in the portfolio at time $t \in [0, T]$, by $\pi_t \in A \subset \mathbb{R}^n$. π is called the control process of the problem. Note that we consider only self-financing portfolio, i.e., the portfolio starts with some initial capital and no injection or extraction of capital before terminal time T. Then the wealth process, denoted by X_t and valued in $\mathbb{R}$, follows the dynamics:

$$\mathrm{d}X_t = \sum_{j=1}^{n} \frac{X_t \pi_j(t)}{S_j(t)} \mathrm{d}S_j(t) + X_t \left(1 - \sum_{j=1}^{n} \pi_j(t)\right) r_t \mathrm{d}t.$$

Substitute equation (2.2.1) into above and re-write in vector form to remove the subscript j:

$$\begin{aligned} \mathrm{d}X_t &= X_t \pi_t^\top (\mu(t)\mathrm{d}t + \sigma(t)\mathrm{d}W_t) + X_t(1 - \pi_t^\top \mathbb{1}) r(t)\mathrm{d}t \\ &= X_t \left(r(t) + \pi_t^\top (\mu(t) - \mathbb{1}\, r(t))\right) \mathrm{d}t + X_t \pi_t^\top \sigma(t) \mathrm{d}W_t \\ &= X_t \left(r(t) + \pi_t^\top \sigma(t) \theta(t)\right) \mathrm{d}t + X_t \pi_t^\top \sigma(t) \mathrm{d}W_t \end{aligned} \tag{2.2.2}$$

where $\theta(t) = [\sigma(t)]^{-1}(\mu(t) - \mathbb{1}\, r(t))$ is the market reward for risk.

Let $U(x) : \mathbb{R} \to \mathbb{R}$ be the terminal utility function at time T. Then primal maximisation problem can be formulated as finding the optimal portfolio π that maximises:

$$J(\pi) := \mathbb{E}\left[U(X_T) \mid X_0 = x_0\right].$$

We can then define the value function $u : [0, T] \times \mathbb{R} \to \mathbb{R}$:

$$u(t, x) := \sup_{\pi \in \mathcal{A}} \mathbb{E}\left[U(X_T) \mid X_t = x\right]. \tag{2.2.3}$$

The HJB equation is derived [4, Chapter 3, page 5] as

$$0 = \partial_t u(t, x) + \sup_{a \in A} F(t, x, a, \partial_x u(t, x), \partial_{xx} u(t, x))$$

where the Hamiltonian control $F : [0, T] \times \mathbb{R} \times A \times \mathbb{R} \times \mathbb{R}$ is defined as:

$$F(t, x, a, z, \gamma) := (r(t) + a^\top \sigma(t)\theta(t))xz + \frac{1}{2}|\sigma(t)^\top a|^2 x^2 \gamma \tag{2.2.4}$$

2.2.2 Dual problem

Like many other mathematic problems, we can deploy conjugate duality to derive the dual problem. The advantage is that dual problem is often much nicer to solve. Moreover, the dual value function provides an upper bound to the primal value solution. If strong duality can be achieved, the dual value function is exactly the same as the primal value function. Here, we will state the dual problem. For details, readers can refer to [12, Chapter 2].

The dual utility function is defined as the conjugate dual of U:

$$\tilde{U}(y) := \sup_x \{U(x) - xy\}.$$

We then try to find the dual process Y such that $\{X_t Y_t\}_{t \geq 0}$ is a super-martingale, i.e.:

$$\mathbb{E}[X_t Y_t] \leq X_0 Y_0 = x_0 y_0$$

where we denote $X_0 = x_0$ and $Y_0 = y_0$. It can be shown that Y has the following representation:

$$\mathrm{d}Y_t = -Y_t \left(r(t) + \delta_A(v_t)\right) \mathrm{d}t - Y_t \left(\theta(t) + [\sigma(t)]^{-1} v_t\right)^\top \mathrm{d}W_t. \tag{2.2.5}$$

The process v and y_0 are the dual control. δ_K is the support function of the set $-A$, defined by:

$$\delta_A(v) := \sup_{\pi \in A} \left\{-\pi^\top v\right\}$$

over $v \in \mathbb{R}^n$.

Now, we show the duality relationship between primal and dual value function. By definition of $\tilde{U}$, we know

$$U(X_T) \leq \tilde{U}(Y_T) + X_T Y_T,$$

then as XY is a super-martingale, we have

$$\mathbb{E}[U(X_T)] \leq \mathbb{E}[\tilde{U}(Y_T)] + \mathbb{E}[X_T Y_T] \leq \mathbb{E}[\tilde{U}(Y_T)] + x_0 y_0.$$

We now take supremum over primal control π and infimum over dual control v, y_0:

$$u(0, x_0) := \sup_\pi \mathbb{E}[U(X_T) \mid X_0 = x_0] \leq \inf_{y_0, v} \left\{\mathbb{E}[\tilde{U}(Y_T)] + x_0 y_0\right\}.$$

As we can see, the solution to the primal maximisation problem is always bounded by the solution to the dual minimisation problem which is on the right hand side of above inequality.

Chapter 3

Markov regime-switching quadratic utility maximisation

3.1 Continuous Markov chain

We shall introduce the concept of continuous Markov chain first as it will be used to model the state of the market in our problem settings.

Let $(\Omega, \mathcal{A}, \mathbb{P})$ be the probability space. Following the convention in [5, Appendix B], a continuous Markov chain X is a stochastic process with value in finite set $S = \{e_1, e_2, \ldots, e_d\} \subset \mathbb{R}^d$ where e_i is a unit basis vector of $\mathbb{R}^d$. For convenience, we sometimes write $X_t = i$ to mean $X_t = e_i$. The continuous Markov chain satisfies Markov property

$$\mathbb{P}[X_{t+h} = j \mid \mathcal{F}_t] = \mathbb{P}[X_{t+h} = j \mid X_t = i]$$

where $\mathcal{F}_t$ is the filtration at time t. A continuous Markov chain is characterised by its generator matrix $Q(t) \in \mathbb{R}^{d \times d}$ which satisfies

$$Q_{ii}(t) = -\sum_{j \neq i}^{d} Q_{ij}(t), \quad Q_{ij}(t) \geq 0, \quad Q_{ii}(t) \leq 0$$

for all t and $i \in S$. The Markov chain process is homogeneous if its generator matrix is time invariant, i.e. $Q(t) = Q(0)$.

3.1.1 Transition probabilities matrix

Assume that generator matrix Q is homogeneous, we can write the transition probability as matrix $P(t) \in \mathbb{R}^{d \times d}$:

$$P_{ij}(t) := \mathbb{P}[X_t = j \mid X_0 = i].$$

As stated in [5, Appendix B], P should satisfy the forward and backward equation:

$$\mathrm{d}P(t) = P(t)Q, \quad \mathrm{d}P(t) = QP(t).$$

This can be solved by

$$P(t) := \exp(t \cdot Q)$$

where the exponential in above formula is the matrix exponential:

$$e^A = \sum_{n}^{\infty} \frac{1}{n!} A^n.$$

Therefore,

$$\mathbb{P}[X_t = j \mid X_0 = i] = \{\exp(t \cdot Q)\}_{ij} \tag{3.1.1}$$

for all interval $t > 0$.

3.1.2 Simulation

Consider the problem of simulating a continuous Markov chain α between time $[0, T]$. Denote the generator matrix as Q. We can start by splitting the time horizon into discrete steps, each step has width Δt. Suppose the current step is k and $\alpha_k = i$. Using the equation (3.1.1), the expression

$$\boldsymbol{p} := [\exp(\Delta t \cdot Q)]^\top \alpha_k$$

is a d-dimension vector with p_j representing the probability jumping from i to state j at time t_{k+1}. The next step is to sample from this distribution. As α_{k+1} is discrete, we firstly compute

$$s_j := \sum_{l=1}^{j} p_l, \quad s_0 := 0$$

for $j = 1, \cdots, d$. It is not hard to see $s_d = 1$. Then, we instantiate a uniform variable $u := U(0, 1)$ and find $\hat{j}$ such that

$$s_{\hat{j}-1} \leq u \leq s_{\hat{j}}.$$

Finally, set $\alpha_{k+1} := e_{\hat{j}}$. We can repeat this procedure for every step until reaching the end.

3.1.3 Associated martingales

Given a continuous Markov chain α with homogeneous generator A, we can define a martingale M_t by:

$$M_t := \alpha_t - \int_0^t A^T \alpha_s \mathrm{d}s. \tag{3.1.2}$$

Interested reader can refer to [5, Appendix B] for proof. Moreover, we can define a process Q on $\mathbb{R}^{d\times d}$ as

$$Q_{ij}(t) := [Q_{ij}](t) - \langle Q_{ij} \rangle (t), \quad Q_{ii}(t) := 0. \tag{3.1.3}$$

The process $[Q_{ij}]$ is a counting process that counts how many times α jumps from state i to state j. In other words:

$$[Q_{ij}](t) := \sum_{s\leq t} \mathbb{1}_{\{\alpha_{s^-}=i\}} \mathbb{1}_{\{\alpha_s=j\}}.$$

On the other hand, the process $\langle Q_{ij} \rangle$ is the compensator process that shows the expected number of jumps from state i to state j, i.e.

$$\langle Q_{ij} \rangle := A_{ij} \sum_{s\leq t} \mathbf{1}_{\{\alpha_{s^-}=i\}}$$

where A_{ij} is the corresponding element in the generator matrix A. It can be shown that the process $\{Q_t\}_{t\geq 0}$ is a martingale [5].

3.2 Unconstrained Markov regime-switching quadratic utility maximisation

In this section, we introduce unconstrained quadratic utility maximisation with Markov regime-switching extension. This is the main problem studied in this report. Later, we will introduce two neural network based approximation algorithms for this problem. Before that, we want to briefly recall the theoretical results in [10] which shows that we can solve this problem analytically. This is useful as we can use the analytical results as a benchmark when designing and verifying our neural-network based algorithms.

3.2.1 Primal problem

We consider the same market model as in section 2.2. There are n stocks $S_1, \cdots, S_n$ driven by n-dimensional Wiener process W. The stocks have growth rate and volatility represented by function μ and σ respectively. There is also a bank account S_0 with interest rate r.

We then assume the economy has different states and model it as a continuous Markov chain α whose value is in $I := \{e_1, \cdots, e_d\} \subset \mathbb{R}^d$. For each state of the Markov chain, we have different market dynamics $r_{\alpha_t}(t)$, $\sigma_{\alpha_t}(t)$, $\theta_{\alpha_t}(t)$ at any time t. Adapting the convention from [10], we denote functions of Markov process α by placing it in the subscript, i.e., $r_{\alpha_t}(t) := r(t, \alpha_t)$. We assume the Markov process is homogeneous and has generator matrix $Q \in \mathbb{R}^{d\times d}$. Moreover, we assume Q is known. Finally, we assume Wiener process W and Markov chain α are independent.

The settings in [10] is slightly different from section 2.2. The portfolio π, valued in $\mathbb{R}^n$, is the amount of cash invested in different stocks instead of the number of shares. Hence the wealth process X_t follows a different dynamics

$$\mathrm{d}X_t = r_{\alpha_t}(t)X_t\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\theta_{\alpha_t}(t)\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t, \quad X_0 = x_0. \tag{3.2.1}$$

This leads to different SMP conditions and also that wealth X might go negative under this setting. Moreover, the problem is unconstrained, i.e. $A := \mathbb{R}^n$.

We have a quadratic terminal utility $U : I \times \mathbb{R} \to \mathbb{R}$

$$U_i(x) = -\frac{1}{2}G_i x^2 - g_i x$$

where G_i and g_i are measurable random variables at terminal time T. Note that U is concave in x. The optimisation problem is to find the optimal control $\hat{\pi}$ that maximises:

$$J_\alpha(\pi) := \mathbb{E}\left[U_{\alpha_T}(X_T) \mid X_0 = x_0, \alpha_0 = i_0\right], \tag{3.2.2}$$

that is

$$J_\alpha(\hat{\pi}) = \sup_{\pi\in\mathcal{A}} J_\alpha(\pi).$$

We can now define the value function $u : [0, T] \times I \times \mathbb{R} \to \mathbb{R}$:

$$u_i(t, x) = u(t, i, x) := \sup_{\pi\in\mathcal{A}} \mathbb{E}\left[U_{\alpha_T}(X_T) \mid X_t = x, \alpha_t = i\right]. \tag{3.2.3}$$

As derived in [10, Appendix A.1], for any i, t and x, the HJB equation is:

$$0 = \partial_t u_i(t, x) + \sup_{a\in A} F(t, i, x, a, \partial_x u_i(t, x), \partial_{xx} u_i(t, x)) + \sum_{j\neq i}^{d} q_{ij} u_j(t, x)$$

where q_{ij} is element of generator matrix Q and $F : [0, T] \times I \times \mathbb{R} \times A \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is the Hamiltonian control:

$$F(t, i, x, a, z, \gamma) := \left(r_i(t)x + a^\top \sigma_i(t)\theta_i(t)\right) z + \frac{1}{2}\left|\sigma_i(t)^T a\right|^2 \gamma. \tag{3.2.4}$$

Evaluate the supremum by $\partial_a F = 0$ and equating the terms, then the optimal control $\hat{\pi}$ should satisfy:

$$\hat{\pi}_t = -\frac{\partial_x u(t, x)}{\partial_{xx} u(t, x)} \left[\theta_i(t)^\top \sigma_i(t)^{-1}\right]^\top \tag{3.2.5}$$

3.2.2 Dual problem

Solving the dual problem typically is simpler compared to solving the primal problem directly. We now re-state results from [10, Section 2.2, page 7] for the dual problem. Consider the martingale $M : [0, T] \times \Omega \to \mathbb{R}^d$ as defined in (3.1.2):

$$M_t := \alpha_t - \int_0^\top Q^\top \alpha_s \mathrm{d}s.$$

The dual process Y is derived to be:

$$\mathrm{d}Y_t = -r_{\alpha_t}(t)Y_t\mathrm{d}t - \theta_{\alpha_t}^{\top}(t)Y_t\mathrm{d}W_t + \gamma_t^{\top}\mathrm{d}M_t, \quad Y_0 = y_0 \tag{3.2.6}$$

The process γ and initial value y_0 are the dual control. The dual problem is then:

$$\inf_{\gamma, y_0} \mathbb{E}[m_{\alpha_T}(Y_T) \mid Y_0 = y_0, \alpha_0 = i_0] + x_0 y_0 \tag{3.2.7}$$

where $m : I \times \mathbb{R} \to \mathbb{R}$ is the dual function of utility U:

$$m_i(y) = \sup_x \{U_i(x) - xy\}.$$

Finally, duality relationship can be shown [10, Section 2.3.2, page 10]:

$$\sup_{\pi \in \mathcal{A}} \mathbb{E}[U(X_T) \mid X_0 = x_0, \alpha_0 = i_0] \leq \inf_{y_0, v} \{\mathbb{E}[m_{\alpha_T}(Y_T) \mid Y_0 = y_0, \alpha_0 = i_0] + xy\}. \tag{3.2.8}$$

3.3 Analytical solutions to unconstrained regime-switching quadratic utility maximisation

In [10, Chapter 2], it applies both HJB and SMP approaches to solve unconstrained quadratic utility maximisation with Markov-switching. In both cases, it guesses an ansatz and reduces the problem into solving a system of ODEs. We will recall the main results here. In later chapter, we will verify these theoretical results with our numerical examples.

3.3.1 Primal problem with HJB

As proven in [10, Appendix A.1], the HJB is

$$\begin{aligned} 0 = (\partial_t u_i) + \sup_{\pi} \left\{ (\partial_x u_i) \left(r_i x + \pi^T \sigma_i \theta_i \right) + \frac{1}{2} (\partial_{xx} u_i) \left| \sigma_i^T \pi \right|^2 \right\} \\ + \sum_{j \neq i}^{d} q_{ij} \left(u_j(t,x) - u_i(t,x) \right). \end{aligned} \tag{3.3.1}$$

We use ansatz:

$$u_i(t,x) := a_i(t) + b_i(t) + \frac{1}{2} c_i(t) x^2 \tag{3.3.2}$$

and substituting into equation (3.3.1), then equating the parameters for x^2, x and constant, we obtain a system of ODEs:

$$\begin{aligned} &\mathbf{c}' = \mathbf{A_c}\mathbf{c} && (\mathbf{A_c})_{m,n} = \begin{cases} |\theta_m|^2 - 2r_m + \sum_{j \neq m}^{d} q_{mj} & m = n \\ -q_{mn} & m \neq n \end{cases} \\ &\mathbf{b}' = \mathbf{A_b}\mathbf{b} && (\mathbf{A_b})_{m,n} = \begin{cases} |\theta_m|^2 - r_m + \sum_{j \neq m}^{d} q_{mj} & m = n \\ -q_{mn} & m \neq n \end{cases} \\ &\mathbf{a}' = -\mathbf{Q}\mathbf{a} + \frac{1}{2} \left(\frac{\mathbf{b}^2}{\mathbf{c}} \right) |\theta_i|^2 && \frac{\mathbf{b}^2}{\mathbf{c}} = \left(b_1^2/c_1, \ldots, b_d^2/c_d \right)^T \end{aligned}$$

where $\boldsymbol{c}(t) = (c_1(t), c_2(t))^{\top}$, $\boldsymbol{b}(t) = (b_1(t), b_2(t))^{\top}$. The terminal boundary conditions are

$$\mathbf{c}(T) = -\mathbf{G}, \quad \mathbf{b}(T) = -\mathbf{g}, \quad \mathbf{a}(T) = \mathbf{0}$$

where $\boldsymbol{G} = (G_1, G_2)^{\top}$ and $\boldsymbol{g} = (g_1, g_2)^{\top}$.

3.3.2 Dual problem with HJB

We define the dual value function v as:

$$v_i(t,y) = \inf_{\gamma} \ \mathbb{E}[m_{\alpha_T}(Y_T) \mid Y_t = y, \alpha_t = i].$$

The HJB equation is proven in [10, Appendix A.2] to be:

$$0 = \inf_{\gamma} \left\{ \partial_t v_i + (\partial_y v_i) \left(-r_i y - \sum_{j=1}^{d} q_{ij}\gamma_j \right) + \frac{1}{2} (\partial_{yy} v_i) |\theta_i y|^2 + \sum_{j \neq i}^{d} q_{ij} v_j (t, y + \gamma_j - \gamma_i) \right\} \quad (3.3.3)$$

Similar to the primal case, we use a quadratic ansatz:

$$v_i(t,y) := \tilde{a}_i(t) + \tilde{b}_i(t)x + \frac{1}{2}\tilde{c}_i(t)x^2.$$

Substitute into HJB and equate the terms, we ends up again with a system of ODEs for $\tilde{a}_i(t)$, $\tilde{b}_i(t)$ and $\tilde{c}_i(t)$:

$$\begin{aligned}
&\tilde{c}_i' + \tilde{c}_i \left(-2r_i + |\theta_i|^2 \right) = \sum_{j \neq i} q_{ij} \left[\frac{\tilde{c}_i^2}{\tilde{c}_j} - \tilde{c}_i \right] && \tilde{c}_i(T) = \frac{1}{G_i} \\
&\tilde{b}_i' + \tilde{b}_i (-r_i) = \sum_{j \neq i} q_{ij} \frac{\tilde{c}_i}{\tilde{c}_j} \left(\tilde{b}_i - \tilde{b}_j \right) && \tilde{b}_i(T) = \frac{g_i}{G_i} \\
&\tilde{a}_i' = \sum_{j \neq i} q_{ij} \left\{ (\tilde{a}_i - \tilde{a}_j) + \frac{\left(\tilde{b}_i - \tilde{b}_j \right)^2}{2\tilde{c}_j} \right\} && \tilde{a}_i(T) = \frac{g_i^2}{2G_i}
\end{aligned}$$

Equivalence between primal HJB and dual HJB solutions

By strong duality between the primal and dual problem, we should be able to recover the primal HJB solution from the dual HJB solution. The relationship is:

$$c_i = \frac{-1}{\tilde{c}_i}, \quad b_i = \frac{-\tilde{b}_i}{\tilde{c}_i}, \quad a_i = \left(\tilde{a}_i - \frac{\tilde{b}_i^2}{2\tilde{c}_i} \right) \quad (3.3.4)$$

The proof is covered in [10, Section 2.3.2].

3.3.3 Primal problem with SMP

Recall that wealth process X has dynamics:

$$\mathrm{d}X_t = \left[r_{\alpha_t}(t) X_t + \pi_t^\top \sigma_{\alpha_t}(t) \theta_{\alpha_t}(t) \right] \mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t) \mathrm{d}W_t, \quad X_0 = x_0.$$

The generalised Hamiltonian $\mathcal{H} : [0,T] \times \mathbb{R} \times \mathbb{R}^n \times I \times \mathbb{R} \times \mathbb{R}^n$ is defined as:

$$\mathcal{H}(t,x,\pi,i,p,q) := \left(r_i(t)x + \pi^\top \sigma_i(t)\theta_i(t) \right) p + \pi^\top \sigma_i(t) q.$$

The adjoint processes (p,q,s) are hence defined, adapted from [11], with the following BSDE:

$$\begin{aligned}
\mathrm{d}p_t &= -r_{\alpha_t}(t) p_t \mathrm{d}t + q_t^\top \mathrm{d}W_t + s_t \bullet \mathrm{d}Q_t \\
p_T &= -G_{\alpha_t} X_T - g_{\alpha_t}
\end{aligned}$$

where

$$s_t \bullet dQ_t := \sum_{i=1}^{d} \sum_{j \neq i}^{d} s_{ij}(t) dQ_{ij}(t)$$

and the process Q is defined in equation (3.1.3).

Now, to solve the control problem, we use an ansatz:

$$p_t = p_{\alpha_{t^-}}(t) := \phi_{\alpha_{t^-}}(t)X_t + \psi_{\alpha_{t^-}}(t)$$

where the $p.(\cdot)$ is a function and ϕ and ψ are functions valued in $\mathbb{R}$. Then, we can show [10, Section 2.5] that ϕ and ψ satisfy a system of ODEs:

$$\partial_t\psi_i(t) + \psi_i(t)\left(-|\theta_i(t)|^2 + r_i(t)\right) = -\sum_{j\neq i}^{d} q_{ij}\left(\psi_j(t) - \psi_i(t)\right) \tag{3.3.5}$$

$$\partial_t\phi_i(t) + \phi_i(t)\left(-|\theta_i(t)|^2 + 2r_i(t)\right) = -\sum_{j\neq i}^{d} q_{ij}\left(\phi_j(t) - \phi_i(t)\right) \tag{3.3.6}$$

with boundary condition:

$$\phi_i(T) = -G_i$$
$$\psi_i(T) = -g_i.$$

Once we solved ϕ and ψ, we can recover the optimal control π by [10, Section 2.5]:

$$\pi_t^\top = -[\theta_{\alpha_t}(t)]^\top\left(X_t + \frac{\psi_{\alpha_{t^-}}(t)}{\phi_{\alpha_{t^-}}(t)}\right)\sigma_{\alpha_t}^{-1}.$$

Equivalence between primal HJB and primal SMP solutions

It can be shown [10, Section 2.5.1] that

$$b_i(t) = \psi_i(t), \quad c_i(t) = \phi_i(t).$$

3.3.4 Dual problem with SMP

Recall that dual process Y is

$$\mathrm{d}Y_t = -r_{\alpha_t}Y_t\mathrm{d}t - \theta_{\alpha_t}^\top Y_t\mathrm{d}W_t + \gamma_t^\top\mathrm{d}M_t, \quad Y_0 = y.$$

Then the generalised Hamiltonian $\mathcal{H} : [0,T]\times\mathbb{R}\times\mathbb{R}^n\times I\times\mathbb{R}\times\mathbb{R}^n\times\mathbb{R}^{d\times d}$ is defined as:

$$\mathcal{H}(t,y,\gamma,i,p,q,s) := -r_i(t)yp - \theta_i^T(t)yq + \sum_{j\neq i}^{d} q_{ij}\gamma_j s_{ij}.$$

Hence the BSDE for adjoint processes (p,q,s) are defined, adapted from [11], as:

$$dp_t = \left(r_{\alpha_t}(t)p_t + \theta_{\alpha_t}^T(t)q_t\right)\mathrm{d}t + q_t^T\mathrm{d}W_t + s_t\bullet\mathrm{d}Q_t \tag{3.3.7}$$

$$p_T = \frac{-1}{G_{\alpha_T}}Y_T - \frac{g_{\alpha_T}}{G_{\alpha_T}} \tag{3.3.8}$$

Theorem 3.3.1. *[10, Appendix B.2] Let $(\hat{y},\hat{\gamma})$ be some admissible dual control. Then $(\hat{y}_0,\hat{\gamma})$ is optimal if and only if the solution $(\hat{Y},\hat{p},\hat{q},\hat{s})$ of the FBSDE*

$$\begin{cases} d\hat{Y}_t = -r_{\alpha_t}\hat{Y}_t\mathrm{d}t - \theta_{\alpha_t}\hat{Y}^\top\mathrm{d}W_t + \hat{\gamma}_t^\top\mathrm{d}M_t, \\ \hat{Y}_0 = \hat{y}_0 \\ d\hat{p}_t = \left[r_{\alpha_t}(t)\hat{p}_t + \hat{q}_2^\top\theta_{\alpha_t}(t)\right]\mathrm{d}t + \hat{q}_t^\top\mathrm{d}W_t + \hat{s}_t\bullet\mathrm{d}Q_t, \\ \hat{p}_T = -\frac{1}{G_{\alpha_t}}\left[\hat{Y}_T + g_{\alpha_t}\right] \end{cases}$$

satisfy the following condition:

$$\begin{cases} \hat{p}_0 = x_0 \\ \hat{s}_{ij} = 0 \end{cases}$$

Note that the second condition $\hat{s}_{ij} = 0$ means the term $\hat{s}_t \bullet \mathrm{d}Q_t$ is zero and can be dropped from the BSDE (5.2.1). Then we select an ansatz:

$$p_t = p_{\alpha_t}(t) := \tilde{\phi}_{\alpha_t}(t) Y_t + \tilde{\psi}_{\alpha_t}(t),$$

where $\tilde{\phi}, \tilde{\psi} : I \times [0,T] \to \mathbb{R}$ are some functions we need to solve for. Substitute into the BSDE and equating the terms, we get

$$q_t = -\tilde{\phi}_{\alpha_t} \theta_{\alpha_t} Y_t. \tag{3.3.9}$$

We will also obtain, for every $i \in I$, two system of ODEs

$$\begin{aligned} \partial_t \tilde{\phi}_i + \tilde{\phi}_i \left(-2r_i + |\theta_i|^2 \right) &= \sum_{j \neq i} q_{ij} \left(\frac{\tilde{\phi}_i^2}{\tilde{\phi}_j} - \tilde{\phi}_i \right) \\ \tilde{\phi}_i(T) &= -\frac{1}{G_i} \end{aligned} \tag{3.3.10}$$

and

$$\begin{aligned} \partial_t \tilde{\psi}_i + \tilde{\psi}_i \left(-r_i \right) &= \sum_{j \neq i} q_{ij} \frac{\tilde{\phi}_i}{\tilde{\phi}_j} \left(\tilde{\psi}_i - \tilde{\psi}_j \right) \\ \tilde{\psi}_i(T) &= -\frac{g_i}{G_i}. \end{aligned} \tag{3.3.11}$$

Once we solved $\tilde{\phi}$ and $\tilde{\psi}$, we can retrieve the optimal dual control by:

$$\gamma_j - \gamma_i = -Y_t \frac{\left(\tilde{\phi}_j - \tilde{\phi}_i \right)}{\tilde{\phi}_j} - \frac{\left(\tilde{\psi}_j - \tilde{\psi}_i \right)}{\tilde{\phi}_j}. \tag{3.3.12}$$

Equivalence between dual SMP and primal SMP solutions

It can be shown that:

$$\begin{aligned} \tilde{b}_i(t) &= -\tilde{\psi}_i(t) \\ \tilde{c}_i(t) &= -\tilde{\phi}_i(t) \end{aligned}$$

Chapter 4

Deep learning algorithms

4.1 Neural network

In this section, we introduce the basic concepts of neural networks, in particular, the multi-layer fully-connected neural networks.

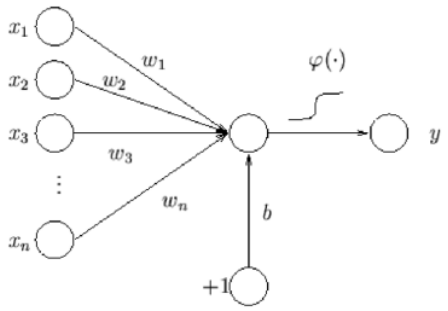

Figure 4.1: Sketch of a perceptron with n inputs from [6], the node has bias b and activation function $\varphi(\cdot)$

A "perception" or a "node" is the basic unit of a neural network. In essence, it takes an input $x \in \mathbb{R}^n$ and computes an output $y \in \mathbb{R}$. In more details, it performs a weighted sum with weights $\omega \in \mathbb{R}^n$. It might also add a bias, denoted by b in figure 4.1. Finally it passes the sum through the "activation" function denoted by $\varphi(\cdot)$. The purpose of the activation function is to add non-linearity as otherwise the perceptron can only output linear terms of input x. The typical activation functions are ReLU and sigmoid:

$$\begin{aligned} \mathrm{ReLU}(x) &= \max(x, 0), \\ \mathrm{sigmoid}(x) &= \frac{1}{1+e^{-x}}. \end{aligned}$$

The choice of activation functions can have a significant impact on the accuracy and the training speed of the neural network. To put everything into formula, a perceptron computes:

$$y := \varphi\left(\sum_{i=1}^{n} \omega_i x_i + b\right) = \varphi\left(\boldsymbol{\omega}^\top \boldsymbol{x} + b\mathbb{1}\right)$$

where $\omega \in \mathbb{R}^n$ and $b \in \mathbb{R}$. In a neural network, the output from the perceptron can be used as an input to another perceptron. A graph of perceptrons is then called a neural network. Note that

nodes are not required to be organised into layers. Also, the graph does not need to be acyclic. For example, recurrent neural network passes output of the node back to the node as one of the input.

A multi-layer feedforward neural network is a particular kind of arrangement for neural networks. It contains an input layer, an output layer and a number of hidden layers. The number of input nodes match the number of inputs to the network, and the output nodes match the number of expected output. We will denote the number of hidden layers by L. Moreover, each hidden layer has l hidden nodes. In figure 4.2, we have $L = 1$ and $l = 4$. The neural network has 2 inputs and 3 outputs. Furthermore, the feedforward network is fully connected, that is, every node in the previous layer is connected to every node in the next layer. Moreover, the activation function is not applied to the outputs of output nodes.

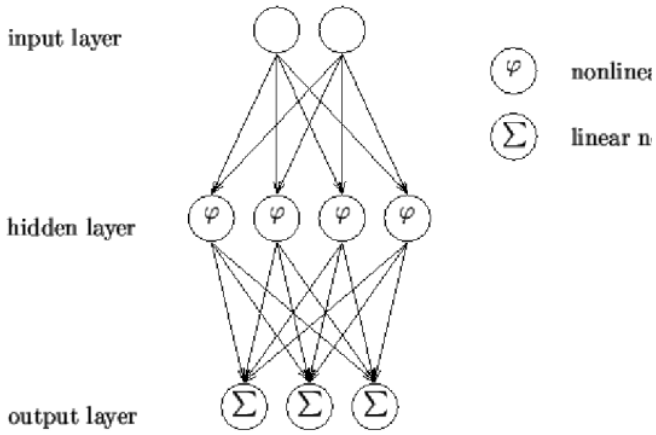

Figure 4.2: Sketch of fully connected multi-layer feedforward neural network from [6], the network has one hidden layer with four hidden nodes

Finally, we can count the number of parameters involved in the neural work, as this directly relates the amount of computation required to train the network. Denote the number of inputs as p and number of outputs as q. The input layer has no parameters. The first hidden layer has $p \times l$ weights and l bias. Any subsequent hidden layer has l^2 weights and l bias. The output layer has $l \times q$ weights and q bias. In total,

$$(p \times l + l) + (L - 1) \times (l^2 + l) + (l \times q + q).$$

Therefore, the number of parameters increases linearly with L, p, q and quadratically with l.

Theorem 4.1.1 (Universal approximation theorem)**.** *[7] Let $(\Omega, \mathcal{A}, \mu)$ be probability space with measure μ, and G be any Borel-measurable continuous function from a finite dimensional space to another. Then a feed-forward multi-layer neural network can approximate G to any desired degree of accuracy, provided sufficiently many hidden units are availble.*

4.1.1 Gradient descent

In machine learning, neural networks are typically used for solving "supervised learning" problems. The problem usually specifies a collection of example inputs and the expected outputs. The neural network is trained to minimise the difference between its outputs and the expected outputs.

In formal terms, suppose we have an example $(\boldsymbol{x}, \boldsymbol{y})$ and the network has output $\hat{\boldsymbol{y}}$ for $\boldsymbol{x}$. Then we can compute the loss $\mathcal{L}$, or sometimes called cost, by:

$$\mathcal{L} := f(\boldsymbol{y}, \hat{\boldsymbol{y}})$$

where $f : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ is the loss function. Note that $(\boldsymbol{y}, \boldsymbol{x})$ is given so can be seen as fixed, and $\hat{\boldsymbol{y}}$ only depends on the parameters of the neural network. Denote the parameters as Θ, we can then express the loss as a function of the parameters:

$$\mathcal{L}(\Theta \mid \boldsymbol{y}, \boldsymbol{x}) := f(\boldsymbol{y}, \hat{\boldsymbol{y}}), \quad \hat{\boldsymbol{y}} := N_\Theta(\boldsymbol{x}).$$

Here, we use N_Θ to denote the neural network. Moreover, there can be n examples, so we want to sum up the losses and find $\hat{\Theta}$ that minimises:

$$\mathcal{L}(\Theta \mid X, Y) := \sum_{i=1}^{n} f(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i), \quad \hat{\boldsymbol{y}}_i := N_\Theta(\boldsymbol{x}_i). \tag{4.1.1}$$

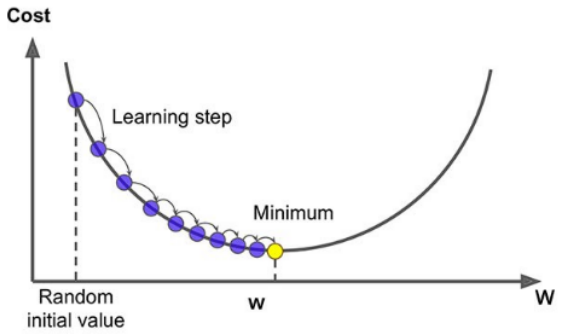

Figure 4.3: Illustration of gradient decent for training parameter w from [8]

To find $\hat{\Theta}$, we can solve analytically

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\Theta} = 0.$$

However, this is usually intractable. The alternative is the gradient descent algorithm. This is an iterative algorithm that updates Θ in steps until it converges. To elaborate, denote the current step by i and the current parameters by Θ_i, then compute Θ_{i+1} by

$$\Theta_{i+1} := \Theta_i - \gamma \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\Theta}\bigg|_{\Theta_i}.$$

The key insight is that moving Θ in the reverse direction of the derivative will reduce the loss. The algorithm is illustrated in figure 4.3 where the parameter w takes several steps and moves gradually towards the minimal point.

At last, γ in above equation is called the learning rate which controls the step size. It is important to choose γ appropriately as small γ leads to slow convergence and large γ risks stepping over the optimal point and failure to converge. There also exists adaptive algorithm like Adam [9] that chooses γ automatically.

The back-propagation algorithm applies gradient descent algorithm to train neural networks. The idea is to compute gradient for parameters in hidden layers and input layers via chain rule and so we can update them with gradient descent. The details are too involved to be covered in this report. Interested readers can refer to [15].

4.1.2 Stochastic gradient descent and batching

Note that (4.1.1) computes the gradient over n training examples. Stochastic gradient descent suggests that we can instead compute the gradient for individual example, which serves as an

estimation of the actual gradient. This can dramatically speeds up training as we do not need to compute over all examples. However, the training might likely converge slower.

Batching is a middle ground between gradient descent and stochastic descent. It computes the gradient for a mini-batch of size B. On one hand, the benefit of stochastic descent remains. On the other hand, it can leverage modern computers' parallelism capability to compute gradient for the whole mini-batch simultaneously. The typical mini-batch has size 256 or 512.

4.1.3 Python implementation

We will use PyTorch to implement any deep learning algorithm in this report.

4.2 Deep learning for utility maximisation problem

The paper [4] introduces two novel approximation algorithm based on deep neural networks. While the paper develops the methods for general utility maximisation problem, our study extends both methods to solve Markov-switching quadratic utility maximisation. In this section, we will explain both algorithms introduced in [4].

4.2.1 Deep controlled 2BSDE method for Markovian problem

The deep controlled 2BSDE method was introduced in [4, Chapter 3]. It is derived from HJB approach hence can only be applied to Markovian problems where r, μ and σ are deterministic functions of time t. Furthermore, we need to assume the optimal control π can be expressed as a deterministic function of wealth process X, i.e.

$$\pi_t := \pi(t, X_t).$$

Recall the primal value function u for utility maximisation is defined as in (2.2.3).

Theorem 4.2.1. *Suppose that the value function $u \in \mathcal{C}^{1,3}([0,T)\times\mathbb{R}) \cap \mathcal{C}^0([0,T]\times\mathbb{R})$, and there exists an optimal control $\pi \in \mathcal{A}$. Then there exist continuous processes (V, Z, Γ) solving the following 2BSDE:*

$$\begin{aligned} \mathrm{d}V_t &= Z_t \pi_t^\top \sigma(t) \mathrm{d}W_t \\ \mathrm{d}Z_t &= -Z_t r(t)\mathrm{d}t + \Gamma_t \pi_t^\top \sigma(t)\mathrm{d}W_t \end{aligned}$$

with boundary conditions $V_T = U(X_T)$ and $Z_T = \partial_x U(X_T)$.

In essence, the processes V, Z are defined as $V_t := u(t, X_t)$, $Z_t := \partial_x u(t, X_t)$. Recall that wealth process X_t follows dynamics in equation (2.2.2):

$$\mathrm{d}X_t = X_t \left(r(t) + \pi_t^\top \sigma(t)\theta(t)\right) \mathrm{d}t + X_t \pi_t^\top \sigma(t) \mathrm{d}W_t$$

with initial condition $X_0 = x_0$. Together with dynamics in theorem 4.2.1, we can simulate process X, V and Z from $t = 0$ to $t = T$. To perform the simulation, split the interval $[0, T]$ into N discrete steps and index by $k = 0, \ldots, N$. Then starting with

$$X_0 = x_0, \quad V_0 = v_0, \quad Z_0 = z_0,$$

we can apply Euler-Maruyama scheme using:

$$\begin{aligned} X_{k+1} - X_k &= X_k(r(t_k) + \pi_k^\top \sigma(t_k)\theta(t_k))\Delta t_k + X_k \pi_t^\top \sigma(t)\Delta W_k \\ V_{k+1} - V_k &= Z_k \pi_k^\top \sigma(t_k)\Delta W_k \\ Z_{k+1} - Z_k &= -Z_k r(t_k)\Delta t_k + \Gamma_k \pi_k^\top \sigma(t_k)\Delta W_k \end{aligned}$$

where $\Delta t_k := t_{k+1} - t_k, \Delta W_k := W_{k+1} - W_k$. Suppose for some v_0 and z_0 such that the simulated V_T and Z_T match $U(X_T)$ and $\partial_x U(X_T)$, then as $v_0 = u(0, x_0)$ by construction, we have found the value of u at x_0.

However, the process Γ and control π are still unknown. Note that Γ_t is approximating $\partial_{xx}u(t, X_t)$, and control π is assumed to be a function of X_t. Therefore, Deep 2BSDE uses neural network to approximate both Γ_t and π_t by:

$$\Gamma_t = N_{\Theta^\Gamma}(t, X_t) \tag{4.2.1}$$
$$\pi_t = N_{\Theta^\pi}(t, X_t) \tag{4.2.2}$$

where $N_\Theta(\cdot)$ stands for some neural network with parameter Θ. Instead of training a single neural network that takes time and wealth as input, it is suggested to have one network per time step k. For every step k, we then approximate

$$\Gamma_k = N_{\Theta_k^\Gamma}(X_k)$$
$$\pi_k = N_{\Theta_k^\pi}(X_k)$$

where Θ_k^Γ and Θ_k^π are for step k. This replaces a single complicated model with multiple simpler and smaller models. Although it has a cost in memory, it is often faster to train a collection of smaller models.

Training neural network

Putting all the parameters together, we define

$$\Theta^\Gamma := \{\Theta_k^\Gamma\}_k, \quad \Theta^\pi := \{\Theta_k^\pi\}_k$$

and we need to train $v_0, z_0, \Theta^\Gamma, \Theta^\pi$. For each epoch, the algorithm performs two rounds of simulations. The first simulation freezes Θ^π and trains v_0, z_0, Θ^Γ using dynamic loss:

$$\mathcal{L}_D := \mathcal{L}(v_0, z_0, \Theta^\Gamma) := |V_N - U(X_N)|^2 + \frac{1}{2}|Z_N - \partial_x U(X_N)|^2.$$

Minimising $\mathcal{L}_D$ trains the neural net to reach the terminal condition. In the second simulation, it uses the updated v_0, z_0, Θ^Γ. With these parameters fixed, it trains Θ^π step by step, and for each step k, it trains Θ_k^π by

$$\mathcal{L}_\pi(k) := \mathcal{L}(\Theta_k^\pi, k) := -F(t_k, X_k, \pi_k, Z_k, \Gamma_k)$$

where F is the Hamiltonian control defined in (2.2.4). Minimising $\mathcal{L}_\pi$ trains Θ^π to find optimal control π that maximises the Hamiltonian. The algorithm repeats until the parameters converge.

4.2.2 Deep SMP method for non-Markovian problem

As previously discussed, HJB requires the control problem to be Markovian. However, this is not a constraint for SMP. The deep SMP method is derived from SMP and so inherits this advantage. The algorithm aims to solve the dual problem first and recover the primal solution via the duality relationship. This is because the dual SMP condition is simpler to achieve than its primal counterpart. Recall the dual problem from section 2.2.2. The dual process Y has dynamics :

$$\mathrm{d}Y_t = -Y_t\left(r(t) + \delta_K(v_t)\right)\mathrm{d}t - Y_t\left(\theta(t) + [\sigma(t)]^{-1}v_t\right)^\top \mathrm{d}W_t, \quad Y_0 = y_0.$$

The dual problem has adjoint processes, see [11, Chapter 2], (P_2, Q_2) that satisfy the BSDE below:

$$\begin{aligned} \mathrm{d}P_2(t) &= \left[(r(t) + \delta_K(v_t)P_2(t) + Q_2(t)^\top\left(\theta(t) + [\sigma(t)]^{-1}v_t\right)\right]\mathrm{d}t + Q_2(t)^\top \mathrm{d}W_t, \\ P_2(T) &= -\mathcal{D}_y\tilde{U}(Y_T). \end{aligned} \tag{4.2.3}$$

Note that P_2 and Q_2 are processes and we do not write t in subscript to avoid clustering of symbols.

Theorem 4.2.2. *[4, Theorem 4.1] Let (y_0, v) be an admissible dual control. Then (y_0, v) is optimal if and only if the solutions (P_2, Q_2) in (4.2.3) satisfy the following:*

$$P_2(0) = x_0, \quad \frac{[\sigma(t)^\top]^{-1} Q_2(t)}{P_2(t)} \in A, \quad P_2(t)\delta_A(v_t) + Q_2(t)^\top \sigma(t)^{-1} v_t = 0$$

for all $t \in [0, T]$ and almost surely.

Notice that the process P_2 has both initial value at $t = 0$ and terminal value at $t = T$ specified. This system of FBSDE is one of the motivation for deep SMP method. The algorithm works as follows. We split time horizon $[0, T]$ into N discrete equal-size steps, indexed by $k = 0, \ldots, N$. Then we use Euler-Maruyama scheme to simulate processes Y and P_2 starting from $Y_0 := y_0$, $P_2(0) := x_0$:

$$\begin{aligned} Y_{k+1} &= Y_k - (t_{k+1} - t_k) Y_k (r(t_k) + \delta_A(v_k)) - Y_k \left(\theta(t_k) + \sigma(t_k)^{-1} v_k\right)^\top \mathrm{d}W_k, \\ P_2(k+1) &= P_2(k) - (t_{k+1} - t_k) \left[r(t_k) P_2(k) + Q_2(k)^\top \theta(t_k)\right] + Q_2(k) \mathrm{d}W_k. \end{aligned}$$

As the process Q_2 and the dual control v are unknown, we will approximate them at each step k by neural networks:

$$\begin{aligned} Q_2(k) &= P_2(k) \sigma(t_k)^\top h_A \left(N_{\Theta_k^Q}(Y_k)\right) \\ v_k &= N_{\Theta_k^v}(Y_i) \end{aligned}$$

where $N_\Theta(\cdot)$ denotes a neural network with parameters Θ. h_A is some differentiable surjective function that ensures second condition in theorem 4.2.2 is satisfied. Also, as we do not know initial value of Y, i.e. y, so we need to learn it from our training as well. Hence, putting all parameters together, we have

$$y_0, \quad \Theta^Q := \{\Theta_k^Q\}_{k=0}^{N-1}, \quad \Theta^v := \{\Theta_k^v\}_{k=0}^{N-1}.$$

In order to learn these parameters, we can compute their losses separately and apply back-propagation to update them individually:

$$\begin{aligned} \mathcal{L}(y_0) &:= \mathbb{E}\left[\tilde{U}(Y_N)\right] + x_0 y_0, \\ \mathcal{L}(\Theta^Q) &:= \mathbb{E}\left[\left|\mathcal{D}_y \tilde{U}(Y_N) + P_2(N)\right|^2\right], \\ \mathcal{L}(\Theta_k^v) &:= \mathbb{E}\left[\left|P_2(k)\delta_A(v_k) + Q_2(k)^\top \sigma(t_k)^{-1} v_k\right|^2\right]. \end{aligned}$$

It is not hard to see the origins of these losses: y_0's loss is to satisfy the dual objective; Q_2's loss is derived from boundary condition of BSDE (4.2.3); v's loss is to satisfy the third condition of theorem 4.2.2.

Remark 4.2.3. It is worth noticing that the Wiener process W is the only source of randomness during the simulation. As we want to update y_0, Θ^Q and Θ^v separately, we typically perform three simulations per training epoch. It is recommended to keep using the same W instead of spawning three different instances of W. This not only saves computation cost but also increases convergence rate of the parameters.

Theorem 4.2.4. *[4, Theorem 3.7] Suppose $(\hat{y}_0, \hat{v})$ is the optimal dual control with corresponding state process $\hat{Y}$ and $(\hat{P}_2, \hat{Q}_2)$ satisfy the BSDE (4.2.3). Then the primal state process $\hat{X}$ associated with the optimal primal control $\hat{\pi}$ satisfies:*

$$\hat{X}_t = P_2(t).$$

Note that unlike the deep 2BSDE method, we do not have a parameter representing the primal solution $u(0, x_0)$. To workaround this, we recall the duality relationship:

$$u(0, x_0) := \sup_\pi \mathbb{E}[U(X_T) \mid X_0 = x_0] \le \inf_{y_0, v} \left\{\mathbb{E}[\tilde{U}(Y_T) \mid Y_0 = y_0] + x_0 y_0\right\}.$$

With theorem 4.2.4, we can recover the primal state X. Therefore, we can estimate an upper and lower bound for $u(0, x_0)$ by Monte-Carlo method:

$$\frac{1}{M} \sum_{i=1}^M U(P_2(N)) \le u(0, x_0) \le \frac{1}{M} \sum_{i=1}^M \tilde{U}(Y_N) + x_0 y_0.$$

Chapter 5

Main results

Previous chapters have covered the preliminary knowledge. In this chapter, we will present the main results of this thesis, that is, the numerical methods for unconstrained Markov regime-switching quadratic utility maximisation problem.

5.1 Deep 2BSDE for Markov regime-switching utility maximisation

As theorem 4.2.1 is only proven for ordinary utility maximisation, we first extend it to handle the regime-switching case. We assume the problem is Markovian, i.e. the optimal control π can be expressed as function of X_t and α_t:

$$\pi_t := \pi(t, \alpha_t, X_t)$$

where π has one more argument α_t compared to the ordinary case. We also assume the problem is Markovian, i.e., market dynamics r, σ, μ are deterministic functions of time t and α_t. Recall the definition of value function u from (3.2.3).

Theorem 5.1.1. *If for any $i \in [1, \dots, d]$, we have $u_i \in \mathcal{C}^{1,3}([0,T) \times \mathbb{R}) \cap \mathcal{C}^0([0,T] \times \mathbb{R})$; and there exists an optimal control $\pi \in \mathcal{A}$. Then there exist continuous processes (V, Z, Γ) solving the following 2BSDE:*

$$\mathrm{d}V_t = Z_t \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t + \boldsymbol{u}_t^\top \mathrm{d}M_t \tag{5.1.1}$$

$$\mathrm{d}Z_t = -Z_t r_{\alpha_t}(t)\mathrm{d}t + \Gamma_t \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t + \boldsymbol{z}_t^\top \mathrm{d}M_t \tag{5.1.2}$$

with terminal conditions

$$V_T = U_{\alpha_T}(X_T), \quad Z_T = \partial_x U_{\alpha_T}(X_T).$$

In above, we define the d-dimensional process

$$\boldsymbol{u}_t := \begin{pmatrix} u_1(t, X_t) \\ \vdots \\ u_d(t, X_t) \end{pmatrix}, \quad \boldsymbol{z}_t := \partial_x \boldsymbol{u}_t = \begin{pmatrix} \partial_x u_1(t, X_t) \\ \vdots \\ \partial_x u_d(t, X_t) \end{pmatrix}.$$

Moreover, M is a martingale process associated with Markov chain α and is defined in (3.1.2)*:*

$$M_t = \alpha_t - \int_0^\top Q^\top \alpha_s \mathrm{d}s.$$

The proof of theorem 5.1.1 is shown in appendix A.1. Compared to theorem 4.2.1, theorem 5.1.1 has added a jump term to the dynamics of process V and Z. Notice that if α_t is fixed then $\mathrm{d}M_t = 0$ and we have recovered theorem 4.2.1. Similar to section 4.2.1, we effectively define V, Z such that

$$V_t := u_{\alpha_t}(t, X_t), \quad Z_t := \partial_x u_{\alpha_t}(t, X_t).$$

5.1.1 Front-to-back training

The extended 2BSDE algorithm works as follows. Theorem 5.1.1 provides the dynamics for process V and Z. Together with dynamics of X, we can simulate X, V, Z from $t = 0$ to $t = T$. Starting with initial condition

$$X_0 := x_0, \quad \alpha_0 := i_0, \quad V_0 := v_0, \quad Z_0 := z_0,$$

where x_0, i_0 is given as input to the problem, v_0, z_0 is unknown. We would like to find v_0 and z_0 such that the simulated V_T, Z_T match $U_{\alpha_T}(X_T)$ and $\partial_x U_{\alpha_T}(X_T)$. If so, v_0 is the solution to value function u at $(0, i_0, x_0)$.

To perform the simulation, we split time $[0, T]$ into discrete equal-sized steps indexed by $k = 0, \dots, N$. As the Markov chain α is independent of Brownian motion W, we can simulate α independently using algorithm in section 3.1. Then, we can simulate X, V, Z using Euler-Maruyama's scheme:

$$\begin{aligned} V_{k+1} - V_k &= Z_k \pi_k^\top \sigma_{\alpha_k}(t_k) \Delta W_k + \boldsymbol{u}_k^\top \mathrm{d}M_k \\ Z_{k+1} - Z_k &= -Z_k r_{\alpha_k}(t_k) \Delta t_k + \Gamma_k \pi_k^\top \sigma_{\alpha_k}(t_k) \Delta W_k + \boldsymbol{z}_k^\top \Delta M_k \end{aligned}$$

where

$$\begin{aligned} \Delta t_k &:= t_{k+1} - t_k \\ \Delta M_k &:= M_{k+1} - M_k = (\alpha_{k+1} - \alpha_k) - Q^\top \alpha_k (t_{k+1} - t_k) \\ \Delta W_k &\sim \mathcal{N}(0, t_{k+1} - t_k). \end{aligned}$$

In above, $\mathcal{N}$ denotes normal distribution. Moreover, the definition of ΔM_k implicitly assumes that Markov chain α jumps at most once within $[t_k, t_{k+1})$. Moreover, the jump happens at the beginning of next step.

We will then need to approximate $\pi_k, \Gamma_k, \boldsymbol{u}_k, \boldsymbol{z}_k$ with neural networks. By definition of these processes, it is natural to express $\Gamma_k, \boldsymbol{u}_k, \boldsymbol{z}_k$ as functions of t, X_t, α_t. Moreover, we make the assumption that the control is Markovian, so that it can be expressed as function of t, X_t, α_t too. Similar to the ordinary case, instead of having a single model and passing time as input, we will have one model per step k. Moreover, instead of having α_t as the input to the neural network, we will have one neural network per Markov state $i = 1, \dots, d$. This is because α_t is discrete and any function that takes it as input is discontinuous. Neural networks are only suited for approximating continuous functions. To summarise, denote the current step as k, then

$$\begin{aligned} \boldsymbol{u}_k &= (N_{\Theta^u_{k,1}}(X_k), \dots, N_{\Theta^u_{k,d}}(X_k))^\top \\ \boldsymbol{z}_k &= (N_{\Theta^z_{k,1}}(X_k), \dots, N_{\Theta^z_{k,d}}(X_k))^\top \\ \Gamma_k &= N_{\Theta^\Gamma_{k,\alpha_k}}(X_k) \\ \pi_k &= N_{\Theta^\pi_{k,\alpha_k}}(X_k) \end{aligned}$$

where $N_\Theta(\cdot)$ denotes a neural network with paramteres Θ.x For convenience, we define

$$\Theta^u := \{\Theta^u_{k,i}\}_{k \geq 0, i \in I}, \quad \Theta^z := \{\Theta^z_{k,i}\}_{k \geq 0, i \in I}, \quad \Theta^\Gamma := \{\Theta^\Gamma_{k,i}\}_{k \geq 0, i \in I}.$$

In addition, we define $\Theta^D := (v_0, z_0, \Theta^u, \Theta^z, \Theta^\Gamma)$. To train Θ^D, we compute the dynamic loss $\mathcal{L}_D$ defined as:

$$\mathcal{L}_D := \mathcal{L}(\Theta^D) := |V_N - U_{\alpha_N}(X_N)|^2 + \frac{1}{2} |Z_N - \partial_x U_{\alpha_N}(X_N)|^2. \tag{5.1.3}$$

This trains the network to satisfy the terminal conditions for process V and Z. Next, the control loss $\mathcal{L}_\pi(k)$ is computed for each step k:

$$\mathcal{L}_\pi(k) := \mathcal{L}(\Theta^\pi_{k,\alpha_k}) = -F(t_k, \alpha_k, X_k, \pi_k, Z_k, \Gamma_k)$$

where F is the Hamiltonian defined in (3.2.4). This motivates the network to find the optimal control π that maximises Hamiltonian F.

The neural nets are trained iteratively. Similar to algorithm presented in section 4.2.1, we perform two rounds of simulations per epoch. In the first round, we freeze Θ^π and train Θ^D using dynamic loss $\mathcal{L}_D$. We then update parameters Θ^D. In the next simulation, we will use the updated Θ^D and train Θ^π using control loss $\mathcal{L}_\pi(\cdot)$. The training completes when the paramteres converge.

5.1.2 Back-to-front training

The training algorithm mentioned in previous section is called "front-to-back" as the simulation runs from $t = 0$ to $t = T$. As we will see in chapter 6, the "front-to-back" approach trains the network only at the hotspots, hence neural networks do not learn the overall shape of their targets. In other words, the network is able to learn the optimal control π when state X is at the most probable path. However, for X that deviates from the mean, it produces terrible estimation.

We can try different x_0. However, this is messy as for each x_0, we have a separate set of neural networks approximating $\boldsymbol{u}$, $\boldsymbol{z}$, Γ and π, and it is a challenge to combine them. The most intuitive solution is to train the same neural network but with different starting point x_0. Moreover, as we will now effectively approximate the value function u and its derivative $\partial_x u$ for a range of x in each step k. It is then inspired to use the approximation from step $k+1$ to train step k.

We call this new approach "back-to-front". It works as follows. Suppose step $k+1$ is already trained and we are now training step k. We choose x_k, i_k randomly from range $\mathcal{R}_k$ and $\{1, \dots, d\}$. We then initialise X, α, V, Z as

$$X_k := x_k, \quad \alpha_k := i_k, \quad V_k := N_{\Theta^u_{k,i_k}}(x_k), \quad Z_k := N_{\Theta^z_{k,i_k}}(x_k).$$

We simulate as described in front-to-back but only for single step, and obtain X_{k+1}, V_{k+1}, Z_{k+1}. By construction, we should have

$$V_{k+1} = u(t_{k+1}, \alpha_{k+1}, X_{k+1}), \quad V_{k+1} = \partial_x u(t_{k+1}, \alpha_{k+1}, X_{k+1}).$$

As step $k+1$ is already trained, we can approximate u and $\partial_x u$ in above equation. Finally, we compute the dynamic loss as

$$\mathcal{L}_D := |V_{k+1} - u(t_{k+1}, \alpha_{k+1}, X_{k+1})|^2 + \frac{1}{2}|Z_{k+1} - \partial_x u(t_{k+1}, \alpha_{k+1}, X_{k+1})|^2.$$

The control loss $\mathcal{L}_\pi(k)$ is the same as before. For the special case $k = N-1$, we do not need to use approximations for u and $\partial_x u$ as we can compute them directly:

$$u_i(T, x) = U_i(x), \quad \partial_x u_i(T, x) = \partial_x U(x).$$

Compared to front-to-back, back-to-front offers huge time saving as we no longer need to perform the full simulation. Moreover, at any point of the training, we only need to keep neural networks for step k and $k+1$ in memory. This means memory consumption is no longer linear with N.

One important hyper-parameter is the ranges $\{\mathcal{R}_k\}_k$. For example, we can fit against $\mathcal{R}_{k+1}$ at step $k+1$. It might so happen that X_{k+1} simulated from step k is not within $\mathcal{R}_{k+1}$. When this happens, we have to drop this sample path as step $k+1$ cannot approximate the value function for this value of X_{k+1}. It can also happen that the optimal path is not within the range. This will no doubt lead to approximation error. In order to select the ranges, we can run a front-to-back simulation and record the ranges of X_k for all steps k. Then we can fit the model back-to-front with the estimated ranges.

It is also possible to establish the range for X_t for any time t. Recall X's dynamics (3.2.1):

$$\mathrm{d}X_t = r_{\alpha_t}(t)X_t\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\theta_{\alpha_t}(t)\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t.$$

As Markov chain α jumps, $\mathrm{d}X$ might jump but process X is still continuous. Also, $\mathrm{d}X_t$ is a normal variable with mean being the drift term

$$r_{\alpha_t}(t)X_t\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\theta_{\alpha_t}(t)\mathrm{d}t$$

and standard deviation being

$$\pi_t^\top \sigma_{\alpha_t}(t)\sqrt{\mathrm{d}t}.$$

Therefore, X_t, being the summation of all $\{\mathrm{d}X_s\}_{s \le t}$ and x_0, is also a normal variable. Moreover, as W is a Brownian motion, $\mathrm{d}W_t$ is independent of each other. Therefore,

$$\mathbb{E}[X_t] = x_0 + \int_0^{t-} \mathbb{E}\left[r_{\alpha_t}(s)X_s + \pi_s^\top \sigma_{\alpha_s}(s)\theta_{\alpha_s}(s)\right]\mathrm{d}s$$

and

$$\mathbb{V}[X_t] = \int_0^{t-} \mathbb{V}[\pi_t^\top \sigma_{\alpha_t}(t)]\mathrm{d}t.$$

Suppose we are able to calculate the mean and variance, we can then compute a range where we are confident X_t will be in.

5.2 Deep SMP method for Markov regime-switching

In this section, we extend deep SMP method introduced in section 4.2.2 for Markov-switching quadratic utility maximisation. Recall the primal problem is to find:

$$\sup_{\pi\in\mathcal{A}} \mathbb{E}[U_{\alpha_T}(X_T) \mid X_0 = x_0, \alpha_0 = i_0]$$

for some $x_0 \in \mathbb{R}$ and $i_0 \in I$. Similar to the original method, we will do so via the dual problem as its SMP condition is easier to achieve. From section 3.2.2, we have dual process Y:

$$\mathrm{d}Y_t = -r_{\alpha_t} Y_t \mathrm{d}t - \theta_{\alpha_t}^\top Y_t \mathrm{d}W_t + \gamma_t^\top \mathrm{d}M_t, \quad Y_0 = y_0$$

where y_0 and γ are the dual control. The dual's adjoint processes (p, q, s) satisfy BSDE:

$$\begin{aligned} dp_t &= \left(r_{\alpha_t}(t)p_t + \theta_{\alpha_t}^T(t)q_t\right)\mathrm{d}t + q_t^T \mathrm{d}W_t + s_t \bullet \mathrm{d}Q_t \\ p_T &= \frac{-1}{G_{\alpha_T}} Y_T - \frac{g_{\alpha_T}}{G_{\alpha_T}}. \end{aligned}$$

In fact, by theorem 3.3.1, we can omit the process s and obtain a FBSDE:

$$\begin{aligned} dp_t &= \left(r_{\alpha_t}(t)p_t + \theta_{\alpha_t}^\top(t)q_t\right)\mathrm{d}t + q_t^T \mathrm{d}W_t \\ p_T &= \frac{-1}{G_{\alpha_T}} Y_T - \frac{g_{\alpha_T}}{G_{\alpha_T}} \\ p_0 &= x_0. \end{aligned}$$

Note that p has both initial and terminal value specified.

The extended deep SMP method works as follows. We split time horizon $[0, T]$ into N discrete equal-size steps, indexed by $k = 0, \ldots, N$. We can simulate the continuous Markov chain α using algorithm mentioned in section 3.1 with $\alpha_0 := i_0$. Moreover, from the dynamics of Y and p, we can simulate them by Euler-Maruyama scheme. Starting from

$$Y_0 := y_0, \quad p_0 := x_0,$$

and denoting the current step by k, we can approximate the dynamics of Y by:

$$Y_{k+1} = Y_k - r_{\alpha_k}(t_k) Y_k (t_{k+1} - t_k) - \theta_{\alpha_t}^\top(t_k) Y_k \Delta W_k + \gamma_k^\top \Delta M_k$$

where

$$\begin{aligned} &\Delta M_k := M_{k+1} - M_k = (\alpha_{k+1} - \alpha_k) - Q^\top \alpha_k (t_{k+1} - t_k) \\ &\Delta W_k \sim \mathcal{N}(0, t_{k+1} - t_k) \end{aligned}$$

where $\mathcal{N}$ stands for normal distribution. Similar to the deep 2BSDE, we assume the continuous Markov chain α jumps at most once within interval $[t_k, t_{k+1})$. More importantly, we assume the jump happens at the start of the interval. Note that we expect the resultant error to decrease as we reduce the step size. Similarly, we can compute

$$p_{k+1} = p_k + \left(r_{\alpha_k}(t_k)p_k + \theta_{\alpha_k}^\top(t_k)\right)(t_{k+1} - t_k) + q_k^\top \Delta W_k.$$

However, we do not actually know the values of y_0, γ_k and q_k. To workaround this, we assume that

$$\gamma_t = f_1(t, Y_t)$$
$$q_t = f_2(t, Y_t, \alpha_t)$$

where $f_1 : [0,T] \times \mathbb{R} \to \mathbb{R}^d$, $f_2 : [0,T] \times \mathbb{R} \times I \to \mathbb{R}^n$ are some functions. We then approximate f_1 and f_2 by neural networks. To elaborate, for each step $k = 0, \dots, N-1$, we will have one neural network with parameter Θ_k^q such that

$$f_1(t_k, Y_k) = N_{\Theta_k^\gamma}(Y_k)$$

with $N_\Theta(\cdot)$ representing the neural network. As mentioned in the 2BSDE case, having step-wise models reduce the overall training time. As for f_2, for every step k and Markov state $i \in I$, we define a neural network with parameter $\Theta_{k,i}^q$ such that:

$$f_2(t_k, Y_k, \alpha_k) = N_{\Theta_{k,i}^q}(Y_k).$$

Remark 5.2.1. For f_2, it is important to have one neural network per Markov state i. Suppose we instead use single neural network such that

$$f_2(t_k, Y_k, \alpha_k) = N_{\Theta_k^q}(Y_k, \alpha_k).$$

As α_k is discrete, q_k is also discrete when fixing Y_k. This discontinuity is problematic as neural networks are not suited for approximating discontinuous functions.

For convenience, define

$$\Theta^\gamma := \{\Theta_k^\gamma\}_k, \quad \Theta^q := \{\Theta_{k,i}^q\}_{k,i}.$$

In order to train these parameters, we need to compute their losses:

$$\mathcal{L}(y_0, \Theta^\gamma) := m_{\alpha_T}(Y_T) + x_0 y_0, \tag{5.2.1}$$
$$\mathcal{L}(\Theta^q) := \left(p_T + \frac{Y_T + g_{\alpha_T}}{G_{\alpha_T}} \right)^2. \tag{5.2.2}$$

The first loss guarantees the optimality of dual control. The second loss aims at the terminal condition of the FBSDE. For each training epoch, we run two rounds of simulations. In first round, we freeze Θ^q and update (y_0, Θ^γ). In second simulation, we freeze (y_0, Θ^γ) and train Θ^q. For both simulations, we should use the same instance of α and W.

Theorem 5.2.2 (Equivalence of dual and primal solution)**.** *[10, Appendix B.3] Suppose $(\hat{y}_0, \hat{\gamma})$ is optimal for the dual problem. Let $\hat{Y}$ be the associated dual state and $(\hat{p}, \hat{q}, \hat{s})$ be the associated adjoint process that satisfy BSDE* (5.2.1)*. Then the optimal control $\hat{\pi}$ and the associated state process $\hat{X}$ satisfy:*

$$\hat{\pi}_t = \left[\sigma_{\alpha_t}^\top\right]^{-1} \hat{q}_t$$
$$\hat{X}_t = \hat{p}_t$$

for all $t \in [0,T]$.

Finally, with theorem 5.2.2, we can recover the optimal state process by $X_t := p_t$. Recall the duality relationship (3.2.8):

$$\sup_{\pi \in \mathcal{A}} \mathbb{E}[U_{\alpha_T}(X_T) \mid X_0 = x_0, \alpha_0 = i_0] \le \inf_{y,v} \left\{ \mathbb{E}[m_{\alpha_T}(Y_T) \mid Y_0 = y_0, \alpha_0 = i_0] + x_0 y_0 \right\}.$$

We can therefore use Monte-Carlo simulation to approximate the expectation on both sides. This gives an upper bound and a lower bound to solution of primal problem. That is:

$$\frac{1}{M} \sum_{j=1}^{M} U_{\alpha_T}(p_T) \le \sup_{\pi \in \mathcal{A}} \mathbb{E}[U_{\alpha_T}(X_T) \mid X_0 = x_0, \alpha_0 = i_0] \le \frac{1}{M} \sum_{j=1}^{M} m_{\alpha_T}(Y_T) + x_0 y_0.$$

Chapter 6

Numerical example

In this chapter, we will present a numerical example, verify the theoretical results in chapter 3 and assess the numerical solutions from algorithms introduced in chapter 5.

6.1 Crafting a numerical example

In our numerical example, we make further simplification that market dynamics r, σ, θ are constant in time and only depend on state of Markov chain. To reflect the simplification, the dynamics of wealth process X becomes

$$\mathrm{d}X_t = \left\{ r_{\alpha_t} X_t + \pi_t^\top \sigma_{\alpha_t} \theta_{\alpha_t} \right\} \mathrm{d}t + \pi_t^\top \sigma_{\alpha_t} \mathrm{d}W_t, \quad X_0 = x_0.$$

And the value function becomes

$$u_i(t,x) = \sup_\pi \ \mathbb{E}\left[-\frac{1}{2} G_{\alpha_T} X_T^2 - g_{\alpha_T} X_T \ \middle|\ X_t = t, \alpha_t = i \right].$$

We consider a Markov chain with $d = 2$ and the market with generator matrix:

$$Q = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$

which assumes we are equally likely to jump between two states. Also, there should be one jump on average within interval $[0,1]$. Moreover, we consider $n = 2$ stocks. The time horizon is $[0,1]$, i.e., $T = 1$. The following lists the other parameters:

$$\begin{aligned} r_1 &= 0.08 & r_2 &= 0.06 \\ \mu_1 &= \begin{pmatrix} 0.1 & 0.06 \end{pmatrix}^\top & \mu_2 &= \begin{pmatrix} 0.1 & 0.08 \end{pmatrix}^\top \\ \sigma_1 &= \begin{pmatrix} 0.2 & 0 \\ 0 & 0.2 \end{pmatrix} & \sigma_2 &= \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix} \\ G_1 &= 1.5 & G_2 &= 0.5 \\ g_1 &= -2 & g_2 &= -1 \end{aligned}$$

We would like to solve the value function $u_i(t,x)$ for $i_0 = 1$ and $x_0 = 1.5$. The state 1 represents market during recession and state 2 represents market during growth. During recession, interest rate is higher, stock growth is lower and market volatility is higher. On the other hand, people are easier to be satisfied. That is, the utility per wealth gain is higher. Figure 6.1 demonstrates the difference in utility between two states.

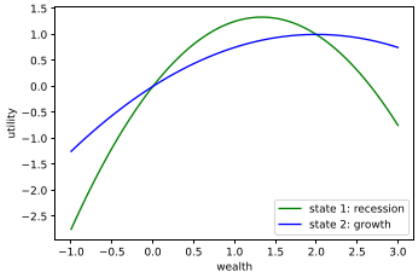

Figure 6.1: Utility versus wealth during recession (state 1) and growth (state 2)

6.2 Verify theoretical results

6.2.1 Primal problem with HJB

We will solve the ODEs in section 3.3.1. Note that $\mathbf{b}$ and $\mathbf{c}$ have systems of homogeneous ODE so can be solved analytically. For example, $\boldsymbol{c}$ has solution:

$$\mathbf{c}(t) = \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{pmatrix} \begin{pmatrix} z_1 e^{\lambda_1 t} \\ z_2 e^{\lambda_2 t} \end{pmatrix}, \quad \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{pmatrix}^\top (-\boldsymbol{G}) \tag{6.2.1}$$

where $\mathbf{v}$ and λ are the eigenvectors and corresponding eigenvalues of matrix $\mathbf{A_c}$. However, it is difficult to solve $\mathbf{a}$ analytically as its system of ODEs is not homogeneous.

Instead, we can approximate $a_i(t)$, $b_i(t)$, $c_i(t)$ numerically. This belongs to a broader class of problems called initial value problem. The approximation algorithm works as follows. We split $[0, T]$ into discrete steps and perform step-by-step integration. Suppose we would like to approximate $x(t)$ for $t > 0$, and $x(t)$ satisfies ODE:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(t, x)$$

where f is some function and $x(0)$ is known. We can start from $x(0)$ and compute for some small $h > 0$:

$$x(t + h) = x(t) + f(t, x(t)) * h.$$

This is called the Euler's method. There are more advanced algorithms such as those covered in [14]. I skip the details as this is not the main focus of this report.

I have chosen to use Runge-Kutta method as it was already implemented in multiple python packages. To assess the accuracy of the Runge-Kutta approximation, we can compare the approximation for $\mathbf{c}$ against its analytical results from equation 6.2.1. As shown in table 6.1, the relative error is below 0.1 percent at time 0.

	$c_1(0)$	$c_2(0)$
Runge-Kutta	-0.096725	-0.096713
analytical	-0.069190	-0.069205
relative error(%)	0.0123	-0.0219

Table 6.1: Relative error of Runge-Kutta method for $\boldsymbol{c}$

Remark 6.2.1. As we know $c_i(T)$ instead of $c_i(0)$, we actually approximate $\tilde{c}_i(\tau) = c_i(T - \tau)$ with Runge-Kutta method then convert back to $c_i(t)$.

Figure 6.2 shows the approximated $c_i(t)$, $b_i(t)$, $a_i(t)$ for $t \in [0, 1]$.

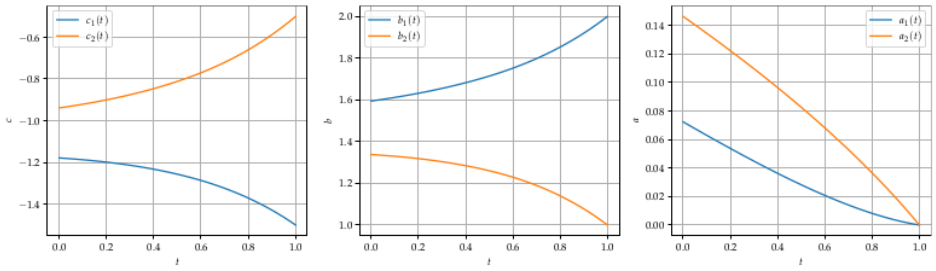

Figure 6.2: Primal problem with HJB, plotted approximated $c_i(t)$, $b_i(t)$, $a_i(t)$ from Runge-Kutta method for time $t \in [0, 1]$

6.2.2 Dual problem with HJB

Unlike the primal case, ODEs for $\tilde{c}_i(t)$, $\tilde{b}_i(t)$, $\tilde{a}_i(t)$ from section 3.3.2 are non-linear. We can still approximate their solutions with Runge-Kutta method. The results are shown in figure 6.3.

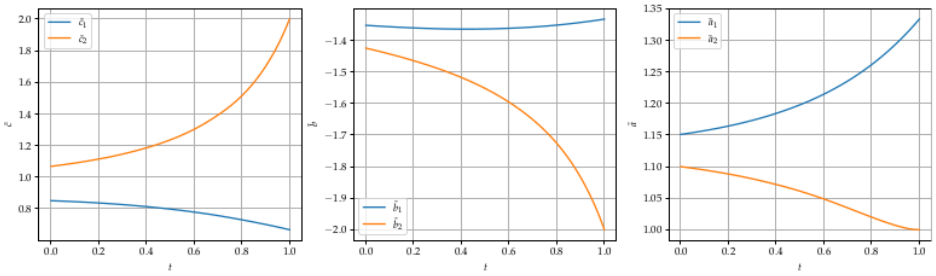

Figure 6.3: Dual problem with HJB, plotted approximated $\tilde{c}_i(t)$, $\tilde{b}_i(t)$, $\tilde{a}_i(t)$ from Runge-Kutta method for time $t \in [0, 1]$

To verify the equivalence relationship, we transform the dual solutions into primal solutions according to (3.3.4). We then plot the transformed primal solutions (solid) versus the actual primal HJB solutions (dash) in figure 6.4. We can see that solutions match precisely for $c_i(t)$ and $b_i(t)$. There is a small difference for $a_i(t)$ which is likely due to approximation error.

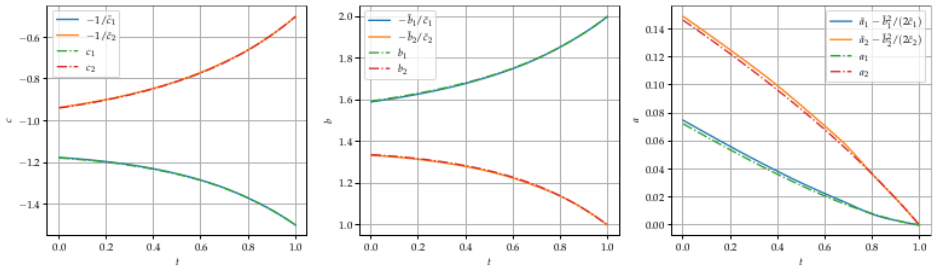

Figure 6.4: Equivalence between primal and dual HJB solutions, plotted transformed dual solutions versus actual primal solutions

6.2.3 Primal problem with SMP

Recall from section 3.3.3, we can solve the adjoint process p by solving $\psi_i(\cdot), \phi_i(\cdot) : [0,T] \times \mathbb{R}$ for every $i = 1, \ldots, d$. Figure 6.5 plots the solutions to equation (3.3.5) and (3.3.6) obtained by Runge-Kutta method. Moreover, we also verified the equivalence between HJB and SMP solutions as shown in figure 6.6.

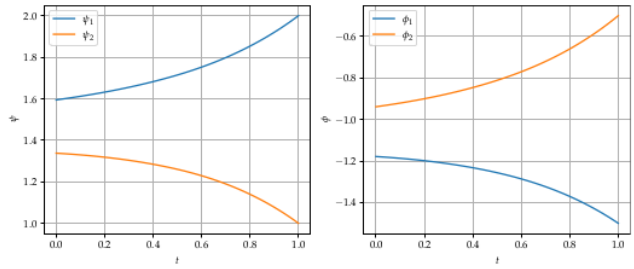

Figure 6.5: SMP solution to primal problem, $\psi_i(\cdot)$ and $\phi_i(\cdot)$ plotted between $[0,1]$ for $i = 1, 2$

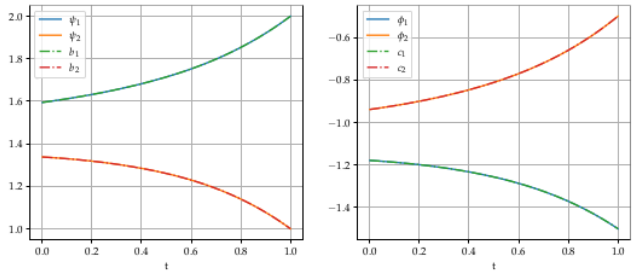

Figure 6.6: Showing equivalence between HJB and SMP solutions to primal problem; left graph plots ψ_i (solid) against b_i (dash); right graph plots ϕ_i (solid) against c_i (dash); all curves match precisely so equivalence relation holds in our numerical example

6.2.4 Dual problem with SMP

Recall from section 3.3.4 that we can solve the dual problem with SMP by solving $\tilde{\phi}_i(\cdot), \tilde{\psi}_i(\cdot) : [0,T] \to \mathbb{R}$ for $i = 1, \ldots, d$. We can again solve (3.3.10) and (3.3.11) by Runge-Kutta method. The results are plotted in figure 6.7.

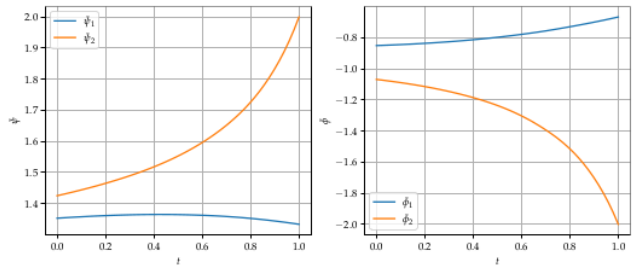

Figure 6.7: SMP solution to dual problem, $\tilde{\psi}_i(\cdot)$ and $\tilde{\phi}_i(\cdot)$ plotted between $[0,1]$ for $i = 1, 2$

Furthermore, we can verify the equivalence relationship between HJB and SMP solutions. This

is shown in figure 6.8. As we can see, the solid lines and the dash lines are indistinguishable, so equivalence relationship holds.

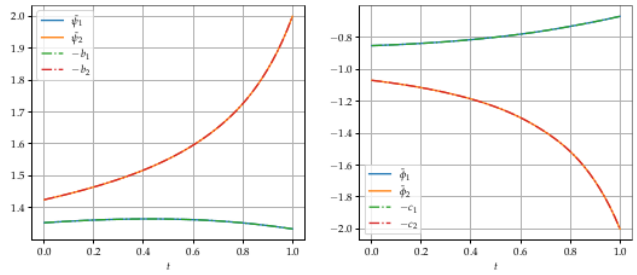

Figure 6.8: Showing equivalence between HJB and SMP solutions to dual problem; left graph plots $\tilde{\psi}_i$ (solid) against $-b_i$ (dash); right graph plots $\tilde{\phi}_i$ (solid) against $-c_i$ (dash); all curves match precisely so equivalence relation holds in our numerical example

6.3 Deep 2BSDE method

In this section, we use deep 2BSDE method introduced in section 5.1.2 to solve the primal problem and report the results here. Recall that we aim to find the value of $u_{i_0}(0, x_0)$ where $i_0 := 1, x_0 := 1.5$. We will also approximate the corresponding optimal control π.

6.3.1 Baseline

For the baseline result, we use $N = 10$. Recall that for every step $k = 0, \ldots, N-1$ and Markov state $i = 1, \ldots, d$, we have four neural networks approximating $\boldsymbol{u}_k, \boldsymbol{z}_k, \Gamma_k, \pi_k$ respectively. We will use the same architecture for all neural networks. It is a fully-connected three layer network with one input layer, one output layer, and $L = 2$ hidden layers. Each hidden layer has $l = 10$ hidden nodes.

The left subplot in figure 6.9 shows the evolution of dynamic loss $\mathcal{L}_D$ defined in (5.1.3). It is an indicator for determining if the training has converged. Note that the graph shows the training of $5,000$ epochs and x-axis represents the percentage of epochs passed. As shown, the dynamic loss starts to converge at around 0 after 40% of epochs.

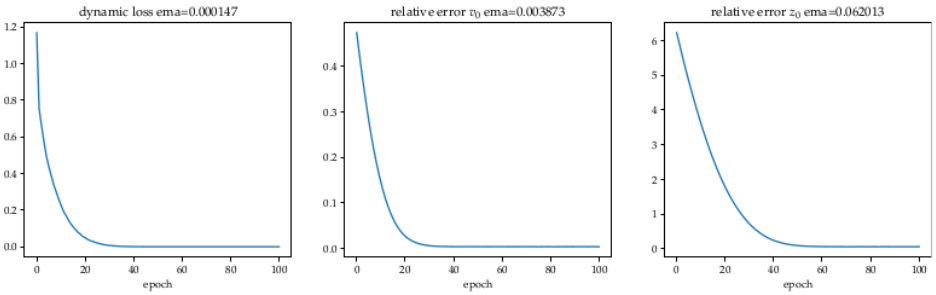

Figure 6.9: Evolution of losses during training, (left) shows dynamic loss $\mathcal{L}_D$, (centre) shows relative error of v_0, (right) shows relative error of z_0

Moreover, the parameter v_0 and z_0 is supposed to be trained towards $u_{i_0}(0, x_0)$ and $\partial_x u_{i_0}(0, x_0)$.

From section 3.3.1, we know that u can be expressed as an ansatz (3.3.2)

$$u_i(t,x) = \frac{1}{2}c_i(t)x^2 + b_i(t)x + a_i(t),$$

and c, b, a are solved for our numerical example in section 6.2.1. Hence, we can compute the analytical value

$$u_0(0, 1.5) = 1.1383, \quad \partial_x u_0(0, 1.5) = -0.1732.$$

Then we can compare our approximated v_0, z_0 against this and calculate the relative error. The centre and right subplots of figure 6.9 then shows the evolution of these losses. Firstly, we can see that dynamic loss plotted on the left is a good proxy for the actual approximation error. The relative errors follow the same pattern as dynamic error and converges around 0 after 40% of the epochs. Moreover, the title shows the exponential moving average error which indicates that v_0 has only 0.3% relative error at the end of training. This is a remarkable result. On the other hand, the error in z_0 is at 6% which is much larger.

Next, we examine the paths of state process X. We take a batch of 512 simulations and plot them in left subplot of figure 6.10. We can see that X fans out from a single point $x_0 = 1.5$. This is as expected. The paths are forming a single cluster. This is also expected from X's dynamics (3.2.1):

$$\mathrm{d}X_t = r_{\alpha_t}(t)X_t\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\theta_{\alpha_t}(t)\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t.$$

As Markov chain α jumps, $\mathrm{d}X$ might jump but process X is still continuous. In contrast to this, the dual process Y has a jump term in its dynamics (3.2.6). As we will later, process Y's trajectories will split into a couple of clusters. Furthermore, we then look at the sample probability density of X_{N-1} as shown in 2nd plot of figure 6.10. We have also plot normal distribution with sample mean and sample deviation (orange dash line). It is obvious that X_{N-1} follows a normal distribution. This is again as expected as $\mathrm{d}X$ is a normal variable and so sum of $\mathrm{d}X_k$ for $k \leq N-1$ is still a normal variable.

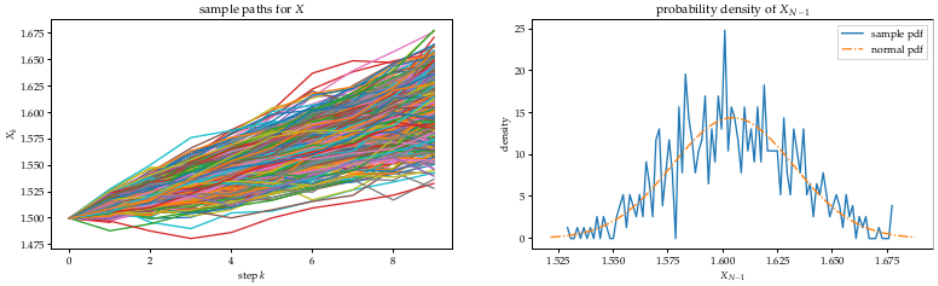

Figure 6.10: Sample trajectories of process X, (left) shows the paths on all steps, (right) plots the sample probability density of X_{N-1} and compare to normal distribution

Finally, we want to assess how well the neural networks approximate their targets. Recall that for every step k and Markov state i, we approximate $\boldsymbol{u}_k, \boldsymbol{z}_k, \Gamma_k, \pi_k$ as functions of X_k. We can also find the analytical values. As shown in section 6.2.1, we have an ansatz (3.3.2) for value function u. Then we have analytical forms for $\boldsymbol{u}_k, \boldsymbol{z}_k, \Gamma_k$ from their definitions. As for π, we can obtain its analytical expression from (3.2.5). Finally, if we fix $k = N-1$, $t = t_k$, and $\alpha_k = i$, then the analytical forms become just a function of X_k. So we can compare the analytical values and numerical estimations over a range of x. We have chosen to use the x range from our batch of simulations. This is plotted in figure 6.11.

Starting with $\boldsymbol{u}$ in the first column, it seems the approximation is far from the analytical values. This is because $\boldsymbol{u}$ is only used in $\boldsymbol{u}_t^\top \mathrm{d}M_t$ term in (5.1.1). As Q in our numerical example is symmetric, i.e. equal chance to jump between state 1 and state 2, $\boldsymbol{u}_t^\top \mathrm{d}M_t$ results into some

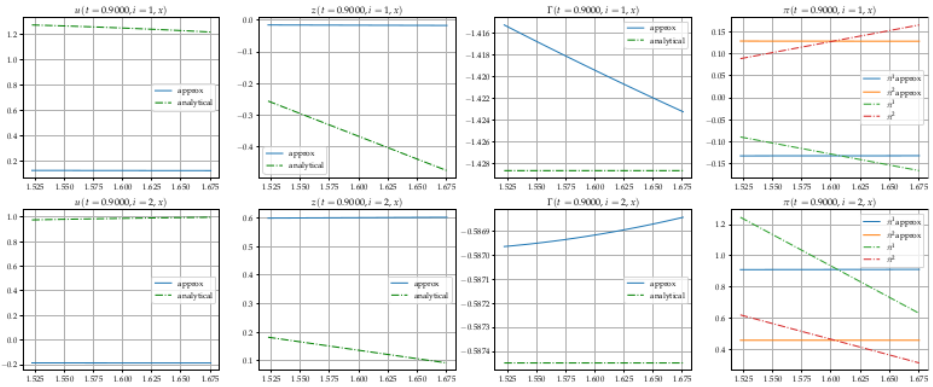

Figure 6.11: Approximations to $\boldsymbol{u}, \boldsymbol{z}, \Gamma, \pi$ at last step, the first row shows for $\alpha_k = i = 1$ and second row shows for $i = 2$; compare against analytical solutions (dash)

multiply of $(u_2(t,x) - u_1(t,x))$. Hence only the difference is trained. If we instead compute $u_2 - u_1$, we see that it is around -0.35 from numerical values and around -0.3 from analytical values. There is still a gap which is likely due to training error. The same applies to $\boldsymbol{z}$ which is shown in second column. $\boldsymbol{z}$ is only used along with $\mathrm{d}M_t$. The error is larger in $\boldsymbol{z}$ which is consistent with what we observed between v_0 and z_0.

Next, we look at Γ_k in 3rd column. Notice that the y-scale is much smaller comparing to other graphs. The relative error in both $i = 1$ and $i = 2$ is well below 1%. Neural networks have approximated Γ pretty well. Finally, we look at π_k. Note that as $n = 2$, $\pi_k \in \mathbb{R}^2$ so there are two curves for each state. We can see that the fitting is good around $x = 1.6$ but gets poorer as x moves away from 1.6. To understand why the network is well trained at 1.6, notice that 1.6 is the mean of normal distribution in the right plot of figure 6.10. As paths centre around 1.6, parameters activated at $x = 1.6$ have the biggest impact on final loss, so get the most training. It is no surprise they are trained the best. The more intriguing is why it does not train well besides $x = 1.6$. Notice that the gap between approximation and analytical is actually symmetrical and has opposite sign around $x = 1.6$. Recall that Θ_k^π is trained to maximise Hamiltonian F (3.2.4) which includes a linear term of π. We therefore suspect that any [over/under]estimation in one side is cancelled out by [under/over]estimation on the other side.

6.3.2 Impact of step size

N	L	l	epochs	B	relative error v_0	relative error z_0	Time(s)
5	2	10	5,000	512	0.0079	0.1260	01:23
10	2	10	5,000	512	0.0039	0.0611	02:48
20	2	10	5,000	512	0.0021	0.0300	05:34
50	2	10	5,000	512	0.0016	0.0238	06:52

Table 6.2: Impact of different number of steps N for 2BSDE front-to-back

In previous section, we have established the baseline model using $N = 10$. In this section, we experiment with different number of steps N and evaluate the impact on training accuracy and training time. We will keep other hyper-parameters as before. For each N, we will train the network with same number of epochs and same batch size B per epoch. In all cases, the training converges. The results are shown in table 6.2. As shown in figure 6.12, the increase in training time is roughly linearly with number of steps N. On the other hand, the relative errors show a non-linear improvement. We do expect improvement from larger N as quantisation error from Euler-

Maruyama scheme is reduced when Δt reduces. However, it was not clear why the improvement is not linear.

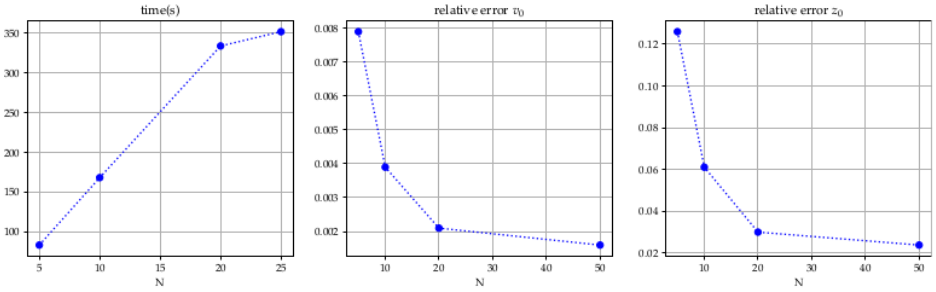

Figure 6.12: Plotting of impact of training time and relative errors against different N

6.3.3 Impact of deeper and wider network

In this section, we will experiment with different neural network architectures including changing number of hidden layers L and number of hidden nodes per hidden layer l. of steps N and evaluate the impact on training accuracy and training time. The results are shown in table 6.3. As shown, there is no obvious improvement in relative error when we increase or decrease L and l. I think this can be attributed to the fact that, in our numerical example, the target functions are all roughly linear. This is obvious to see from figure 6.11. Hence making the model wider/deeper, which increases non-linearity of the model, does not improve the accuracy.

N	L	l	epochs	B	relative error v_0	relative error z_0	Time(s)
10	1	10	5,000	512	0.0039	0.0613	01:51
10	2	10	5,000	512	0.0039	0.0611	02:48
10	2	20	5,000	512	0.0039	0.0613	03:14
10	3	10	5,000	512	0.0040	0.0612	03:31

Table 6.3: Impact of different number of steps N for 2BSDE front-to-back

6.4 Deep 2BSDE method back-to-front

In previous section, we use the front-to-back approach for deep 2BSDE method. As shown in figure 6.11, we can see that neural networks do not learn the shape of their target functions. This is because the networks are trained only at the hotspots. In this section, we will apply the back-to-front approach which should learn the shapes of the target functions well. We will use the same network architecture as previous section. It will be a fully-connected multi-layer neural networks consists of one input layer, one output layer and L hidden layers. Each hidden layer contains l hidden nodes. We use sigmoid as the activation function.

6.4.1 Baseline

We will use $N = 5$, $L = 2$ and $l = 10$. As mentioned, one additional hyper-parameter for back-to-front is the ranges $\mathcal{R}_k$ each step k is trained on. We will use the same range for all steps:

$$\mathcal{R}_k := \mathcal{R} := [1.0, 2.0].$$

Notice that $x_0 = 1.5$ is the mid of this interval.

We firstly train the last step, i.e. $k = N-1$, for 30,000 epochs and a batch size of 512. Figure 6.13 shows the evolution of its losses. Note that the x-axis is scaled down to represent the percentage of epochs done. The left figure plots the dynamic loss $\mathcal{L}_D$ (5.1.3) against training epochs. We also compute the relative error between analytical value of $u_i(t, x)$ (3.3.2), obatined by Runge-Kutta method, and approximations from neural networks. The error is computed and averaged over 50 evenly spaced points within range $\mathcal{R}$. The centre diagram plots for state $i = 1$ and the right diagram for state $i = 2$. It is worth noting that relative error is not used during training as it requires an analytical solution which most problems do not have. This is used only for validating the performance of our neural networks. Moreover, both the dynamic loss and relative error decrease swiftly and converge around 0. Furthermore, the dynamic loss and absolute errors demonstrate similar patterns which suggests that dynamic loss is a good proxy for the absolute error. This makes sense as dynamic loss is partly a square of the absolute error.

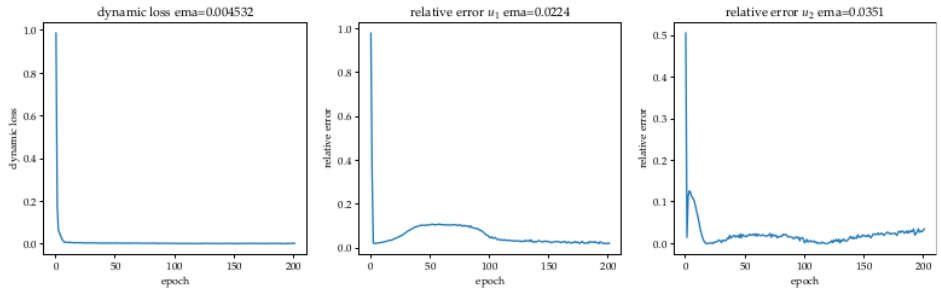

Figure 6.13: Evolution of losses of step $N-1$ during training, plots of dynamic losses (left), relative loss of u for $i = 1$ (centre) and $i = 2$ (right), x-axis represents percentage of epochs done

Finally, the title of centre and right plot states the exponential moving average of relative error, and it is around 2% and 3.5%. This is certainly not as good as the 0.4% from front-to-back approach. However, that was computed only for $x_0 = 1.5$ instead of averaging over 50 points. Also, back-to-front appraoch is approximating the entire curve instead of a single point, so additional error is inevitable.

After determining the training has converged, we now examine the predictions from neural networks and compare them against the benchmark. In order to compute the analytical values as benchmark, recall that u has antasz (3.3.2). Then by construction and (3.2.5), we can obtain analytical forms of $\boldsymbol{u}_t$, $\boldsymbol{z}_t$, Γ_t and π_t, which is some function of t, α_t and X_t. Then for any step k (so $t := t_k$) and

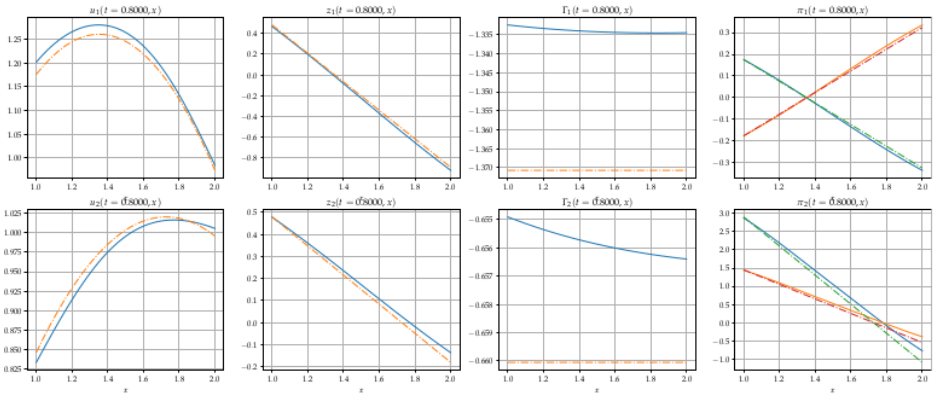

Figure 6.14: Compare approximations at step $N-1$ for $\boldsymbol{u}$, $\boldsymbol{z}$, Γ and π versus their analytical values; approximation is in solid line and analytical in dash, note that π has 2 lines, one for each stock

Markov state $\alpha_k := i$, they are functions of x. Similarly, our neural network takes X_t as input, hence we can compare approximation with analytical solutions over a range of x. This is shown in figure 6.14 with the solid line being our approximations and dash line being the benchmark.

From figure 6.14, the approximations to u, z and π have matched with the benchmark for both Markov states $i = 1$ and $i = 2$. As for Γ, it might seen at first that there is a large mismatch. This is actually due to the small y-axis scale. The actual relative error is around 0.5%. The neural networks have managed to learn the correct shapes and values. This is in great contrast with figure 6.11 which shows front-to-back does not learn the actual shapes of target functions.

Finally, it is interesting to observe the approximations outside the designated training region. The last step is trained with $x \in [1., 2.]$. We can instead plot for $x \in [-1, 4]$ as shown in figure 6.15. Unsurprisingly, the neural network performed really poorly. In general, neural network does not generalise well outside of the training domain. As mentioned in the previous discussion, this motivates the requirement of capping x when training back-to-front, that is, if x is outside of the training range of next step, we will ignore this training sample and do not compute the dynamic loss. This is also a problem with front-to-back training. When X_t advances into a rare path, neural net performs really poorly and generate large gradient, causing instability for training.

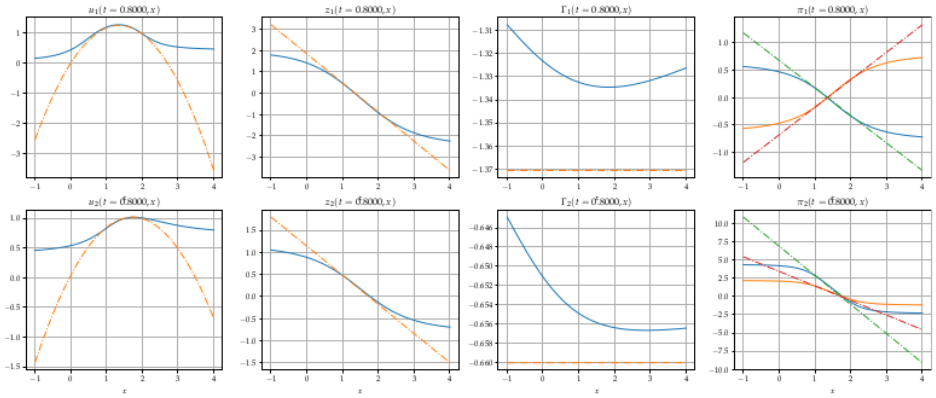

Figure 6.15: Compare approximations at step $N-1$ for $\boldsymbol{u}$, $\boldsymbol{z}$, Γ and π versus their analytical values with extended range where x is untrained

So far, we have looked at the behaviour of neural networks at last step or $k = N - 1$. We now look at the results of first step $k = 0$. As the back-to-front training describes, we can recursively train step k by step $k+1$. Eventually, we will reach step 0. Figure 6.16 then show the evolution of dynamic loss and relative errors of $u_i(0, x)$ for $i = 1$ and $i = 2$. Note that the dynamic loss (left) is computed with respect to $\boldsymbol{u}_k$, $\boldsymbol{z}_k$ estimated from step $k = 1$, while the relative errors (centre, right) are between approximations of $k = 0$ and analytical values.

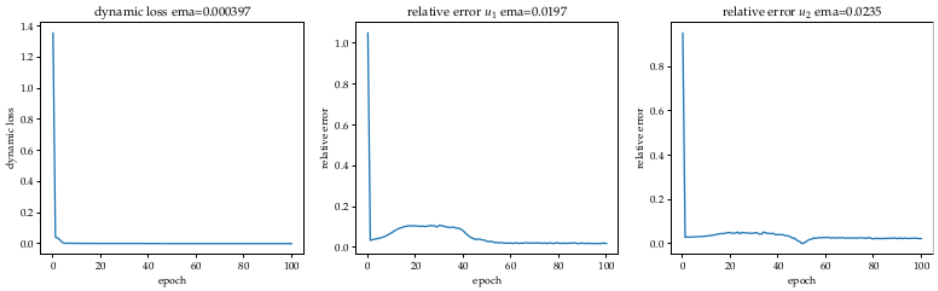

Figure 6.16: Evolution of losses of step $k = 0$ during training, plots of dynamic losses (left), relative loss of u for $i = 1$ (centre) and $i = 2$ (right), x-axis represents percentage of epochs done

Overall, we can see that dynamic losses and relative error has converged in the first half the epochs. The exponential moving average of relative error of u_1 and u_2 are around 2%. This is surprisingly good given that we have 2% error already in step $k = N - 1$.

To further understand the approximation error at step 0, note that each step k is trained to approximate the next step $k + 1$, so its approximation error is the summation all errors from all subsequent steps. Suppose each step has bound ϵ on its approximation error, then step 0 will have approximation error $N \times \epsilon$ compared to the actual benchmark value in the worst case. Note that as each step can over-approximate or under-approximate, so these errors can cancel each out. Moreover, it will be useful to see how approximations of u evolves from step $N - 1$ to 0. We plot in figure 6.17 the approximations at steps $k = 0, 1, 2, 3$ except the last. Pay particular attention to $u_2(t, x)$ on the second row, we can see that step $k = 3$ is underestimating the benchmark. On top of this, step $k = 2$ further underestimates the benchmark. However, step $k = 0$ was able to recover by likely overestimating. So under approximation and over approximation cancel each other out.

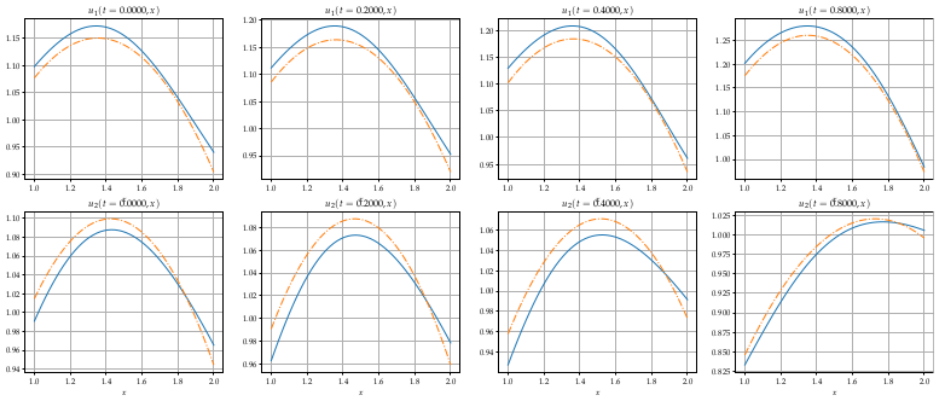

Figure 6.17: Evolution of approximations to $u_i(t, x)$ by steps $k = 0, 1, 2, 3$, the first row shows for $i = 1$ and second for $i = 2$

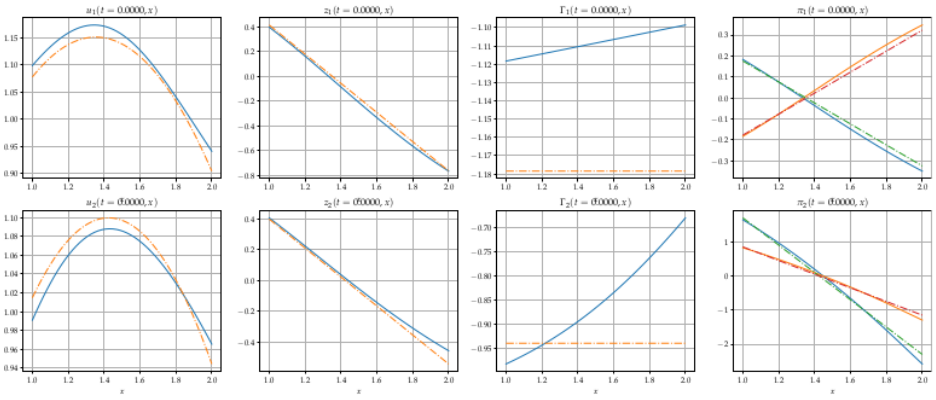

Figure 6.18: Compare approximations at step $k = 0$ for $\boldsymbol{u}$, $\boldsymbol{z}$, Γ and π versus their analytical values

Finally, we look at the approximations by step $k = 0$ to $\boldsymbol{u}$, $\boldsymbol{z}$, Γ, π. This is plotted in figure 6.18. As we can see, the fitting is good overall except for Γ at Markov state $i = 2$. On the other hand, π has a very decent fit.

6.4.2 Impact of step size

Recall the simulation is based on Euler-Maruyama's scheme. The idea is to replace the infinitesimally small $\mathrm{d}t$ in the dynamics with finite Δt and so we can simulate the processes in finite number of steps. Hence, the bigger Δt, the bigger the approximation error. Therefore, we should be able to reduce the approximation error in each step by using smaller step sizes, or equivalently bigger N. In this section, we experiment with different N and see how it impacts the relative error

We start with training only the last step, i.e., $k = N - 1$. We will fix other hyper-parameters and train for the same amount of epochs. The results are shown in table 6.4.

N	L	l	epochs	B	min x	max x	relative error u_1	relative error u_2	Time(s)
5	2	10	30,000	512	1.0	2.0	0.0224	0.0351	1:52
10	2	10	30,000	512	1.0	2.0	0.0073	0.0129	1:54
20	2	10	30,000	512	1.0	2.0	0.0038	0.0075	1:54
50	2	10	30,000	512	1.0	2.0	0.0013	0.0018	1:54

Table 6.4: Relative error and training time of step $k = N - 1$ with different number of steps N

As we can see, increasing N reduces approximation error in last step more than linearly. As we discussed in previous section, the relative error at step 0 is the summation of approximation error of all subsequent steps, so linear with N and per-step approximation error ϵ. As increasing N reduces ϵ more than linearly, the overall relative error in step 0 should decrease. Moreover, if we can assume the stepwise approximation error ϵ as independent random variables with mean 0, which is most likely the case, then approximation errors from different steps can cancel out each other. To see it differently, the final approximation error can be defined as

$$s := \sum_{k=0}^{N-1} \epsilon_k.$$

By Central Limit Theorem, with large enough N, $s \to N\mathbb{E}[\epsilon] = 0$.

Table 6.5 shows the relative error at step $k = 0$. We can see that increasing N indeed reduces the relative error in step 0. However, one obvious limitation of "back-to-front" is also made obvious

in this table. It takes 30,000 epochs, 40 minutes to train a neural network with $N = 20$ while it takes only 5,000 epochs and 5 minutes for front-to-back as shown in table 6.2. Not only it takes more epochs, each epoch of front-to-back takes longer (0.08s vs 0.03s). The increase in number of epochs is expected as we are training for entire curve while front-to-back is mostly training for a single point.

N	L	l	epochs	B	min x	max x	relative error u_1	relative error u_2	Time(s)
5	2	10	30,000	512	1.0	2.0	0.0197	0.0235	10:12
10	2	10	30,000	512	1.0	2.0	0.0153	0.0106	20:42
20	2	10	30,000	512	1.0	2.0	0.0092	0.0080	41:38

Table 6.5: Relative error and training time of step $k = 0$ with different number of steps N

6.4.3 Impact of choice of ranges $\mathcal{R}_k$

In this section, we want to examine the impact of the choice of $\mathcal{R}$. In the baseline, we have chosen to use $[1.0, 2.0]$. In this section, we will try ranges of sizes 0.2, 0.6 and 1.0 but all with midpoint at $x_0 = 1.5$. Intuitively, the approximation error should increase when the approximation range increases. The result is shown in table 6.6. Surprisingly, the relative error for small range $\mathcal{R}$ is not noticeably better than the error for bigger range.

N	L	l	epochs	B	min x	max x	relative error u_1	relative error u_2	Time(s)
10	1	10	30,000	512	1.4	1.6	0.0080	0.0125	20:53
10	1	10	30,000	512	1.2	1.8	0.0194	0.0086	20:42
10	1	10	30,000	512	1.0	2.0	0.0153	0.0106	20:42

Table 6.6: Relative error and training time of step $k = 0$ with different number choices of $\mathcal{R}$

6.5 Deep SMP method

Recall that we are trying to approximate the primal value function u and the optimal primal control π with $x_0 = 1.5$ and $i_0 = 1$. In this section, we will apply the deep SMP method introduced in section 5.2.

6.5.1 Baseline

In terms of neural networks, we used the same architecture for $N_{\Theta^q_{k,i}}(\cdot)$ and $N_{\Theta^\gamma_k}(\cdot)$. Each neural network is a fully-connected multi-layer perception with one input layer, one output layer and $L = 2$ hidden layers. Each hidden layer has $l = 10$ hidden nodes. We use sigmoid function as the activation functions and perceptions have bias. Finally, we will split the time horizon into $N = 10$ steps. We now report the results.

In deep SMP method, we can estimate an upper bound and a lower bound for the primal value $u_{i_0}(0, x_0)$ using the primal and dual gain function:

$$u_{\text{lower}} := \frac{1}{M}\sum_{j=1}^{M} U_{\alpha_T}(p_T)$$

$$u_{\text{upper}} := \frac{1}{M}\sum_{j=1}^{M} m_{\alpha_T}(Y_T) + x_0 y_0.$$

Moreover, from section 3.3, we know

$$u_i(t, x) = \frac{1}{2}c_i(t)x^2 + b_i(t)x + a_i(t)$$

and functions $c_i(\cdot)$, $b_i(\cdot)$, $a_i(\cdot)$ are solved in section 6.2.1. Therefore, we can obtain the exact analytical value of $u_{i_0}(0, x_0)$. In the left subplot of figure 6.19, we plot the estimated upper bound and lower bound at different stages of training. We also plot, in dash line, the analytical value $u_{i_0}(0, x_0)$. As we can see, the bounds converge at around the exact value.

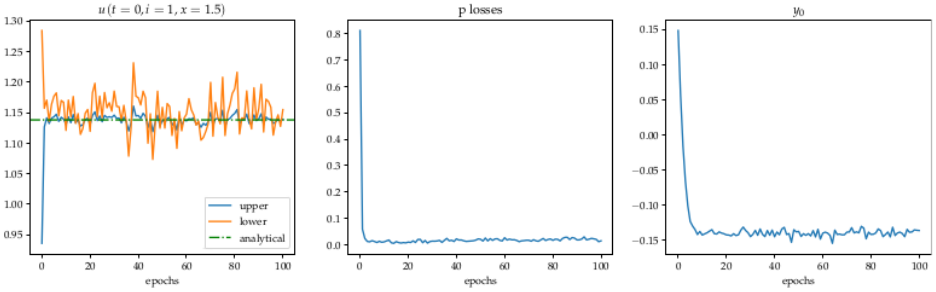

Figure 6.19: Evolution of value estimation (left), loss of process p (centre) and parameter y_0 (right) during training

In the centre subplot of figure 6.19, we plot the loss $\mathcal{L}(\Theta^q)$ defined in equation (5.2.2). We can see that the loss is going down fast before flattening around 0. This indicates the training has converged. Finally, the right subplot shows the parameter y_0 which is the starting point of dual process Y. Similarly, we can see that it has converged as well.

Note that we do not see much detail in figure 6.19 as the initial bounds and losses skew the y-axis. Figure 6.20 shows the evolution if we train the neural network for additional epochs. This offers interesting details after the network has converged. First of all, notice that the bounds (left) and loss (centre) are simply oscillating without improving. This hints that we have reached the optimal point.

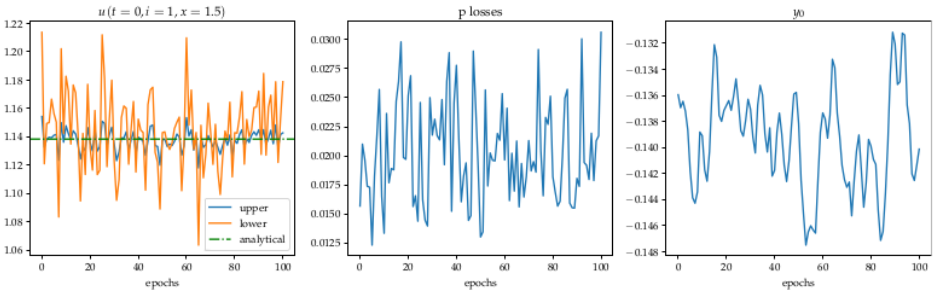

Figure 6.20: Evolution of value estimation (left), loss of process p (centre) and parameter y_0 (right) after training has converged

Moreover, it might seem odd that the upper bound estimation (orange) can go below the lower bound estimation (blue). This is okay as the duality (3.2.8) only depicts the inequality in expectations

$$\mathbb{E}[U(X_T)] \leq \{\mathbb{E}[m_{\alpha_T}(Y_T)] + x_0 y_0\}\,.$$

Individual path can exist where dual gain is below primal gain. Furthermore, we can see the dual gain is oscillating in much bigger extent than the primal gain. We suspect this is related to the dual process dynamic containing a jump term.

Finally, we compute the relative error of the bounds:

$$\begin{aligned}\text{RelErr}_{\text{upper}} &:= \frac{u_{\text{upper}} - u_{i_0}(0,x_0)}{u_{i_0}(0,x_0)} = 0.0074 \pm 0.0022\\ \text{RelErr}_{\text{lower}} &:= \frac{u_{i_0}(0,x_0) - u_{\text{lower}}}{u_{i_0}(0,x_0)} = 0.0002 \pm 0.0006\end{aligned}$$

where $u_{i_0}(0,x_0)$ is obtained analytically. Note that relative error is in fact a random variable so we include its standard deviation. Both bounds, especially the lower bound, have error below 1% which is a promising sign.

6.5.2 Approximation versus analytical

Instead of looking at the final loss, we can also examine each step and compare approximation of γ and q against their analytical counterparts. This helps us understand how well the network has learned these intermediate values. In particular, we look at final step $k = N-1$.

Recall that for step k and Markov state $\alpha_k = i$, we approximate q_k by neural network $N_{\Theta^q_{k,i}}(Y_k)$. From equation (3.3.9), we see that q can be expressed as a function of (t, α_t, Y_t)

$$q_t = q_{\alpha_t}(t, Y_t) = -\tilde{\phi}_{\alpha_t}\theta_{\alpha_t} Y_t.$$

We have solved function $\tilde{\phi}$ in section 6.2.4. With $t = t_k$ and $\alpha_k = i$ fixed, we can compare our approximation with the analytical solution for a range of y. This is plotted in figure 6.21 for both $i = 1$ and $i = 2$. Note that $q_i(t_k, y) \in \mathbb{R}^n$, we denote each component by superscript l in q_i^l.

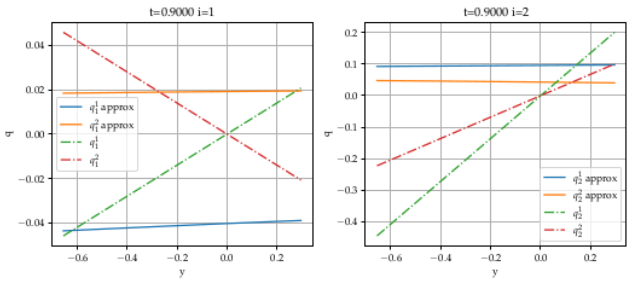

Figure 6.21: Plotting approximations of q_i versus exact values at step $k = N - 1$; left plot shows for state $i = 1$ and right plot shows for state $i = 2$

At first glance, the neural networks seem to fit poorly. However, there is a logical explanation for this. The problem is with the range of Y the neural networks are trained on. To elaborate, we shall observe the trajectories of Y. We take a batch of 512 simulations and plot them in figure 6.22. The first plot shows the trajectories. Notice that Y fans out from a single point as we have a single starting point y_0.

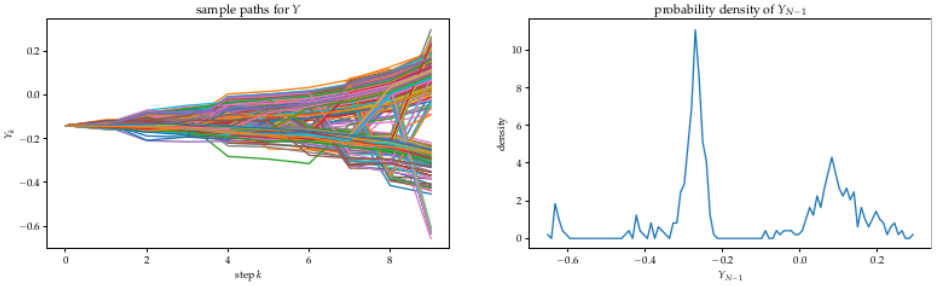

Figure 6.22: Sample trajectories of dual process Y (left); distribution of Y_{N-1} (right)

Moreover, the paths gradually split up into two clusters. This is more noticeable in the second plot which shows the sample probability density of Y_{N-1}. As we can see, there is a major peak at around -0.3. There is another slightly flatter peak at around 0.1. To understand the origins of these two bell-shaped distributions, recall that our numerical example has generator matrix

$$Q = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}.$$

As "the interval before jump from state i" is an exponential random variable with rate Q_{ii}, we expect there to be one jump within the interval $[0, T]$ where $T = 1$. The further into the horizon, the more likely the jump will occur. Once the jump occurs, the Markov chain α is likely to stay in the new state until the end. Hence, the first peak corresponds to a path without any jump and stays at state $i_0 = 1$. This is the most likely case so obtains the highest density. The second peak represents the paths that perform single jump to state 2. As the jump can occur at different steps, the bell shape is less concentrated. We can also see from the left plot in figure 6.22 that jumps occur more often in the later half of the paths, which is expected. The last but not the least, recall that dynamics of dual process Y (3.2.6) is driven by the jump term and a Brownian motion:

$$\mathrm{d}Y_t = -r_{\alpha_t} Y_t \mathrm{d}t - \theta_{\alpha_t}^\top Y_t \mathrm{d}W_t + \gamma_t^\top \mathrm{d}M_t.$$

Without the jump term, Y_{N-1} is a normal variable which is why the peaks are bell-shaped.

To conclude, the neural networks are trained mostly at around -0.3 and 0.1. More importantly, when Y_{N-1} is around -0.3, the Markov chain α_{N-1} is most likely to be 1. Vice versa, when Y_{N-1}

is near 0.1, α_{N-1} is most likely to be 2. Hence, the neural network $N_{\Theta^q_{k,i}}(\cdot)$ is only trained around -0.3 for $i = 1$ and 0.1 for $i = 2$. Now, if we look at figure 6.21 again, we can see this is indeed the case. For $i = 1$ and $y = -0.3$, the neural network is very accurate for q_1^2 and slightly off for q_1^2. On the other hand, for $i = 2$ and $y = 0.1$, the neural network has a good approximation for both q_2^1, q_2^2. As neural network cannot generalise outside of its training domain, it is expected that it does not fit well elsewhere.

Finally, we will assess approximations of γ. The process γ is valued in $\mathbb{R}^d$ and we denote its element by γ_i for $i = 1, \ldots, d$. Section 6.2.4 shows that processes γ_j and γ_i should satisfy relation (3.3.12) and section 6.2.4 solves $\tilde{\psi}$ and $\tilde{\phi}$. We can then evaluate $\gamma_j(t) - \gamma_i(t)$ for $t \in [0, T]$:

$$\gamma_j(t) - \gamma_i(t) = f(t, Y_t)$$

for some function f. On the other hand, for every step k with $t = t_k$, we approximate $\gamma_k = N_{\Theta^\gamma_k}(Y_k)$. So we can compare $N_{\Theta^\gamma_k}(y)$ with $f(t_k, y)$ for a range of y.This is shown in figure 6.23. Similar to process q, the networks does not fit well overall. However, for $i = 1$, the approximation is precisely the same as the analytical value at $y = -0.3$. Similarly, for $i = 2$, the approximation matches nicely with analytical value at values close to $y = 0.1$. Note that for $i = 2$, the best matching does not occur at $y = 0.1$ actually. We suspect this is due to training error.

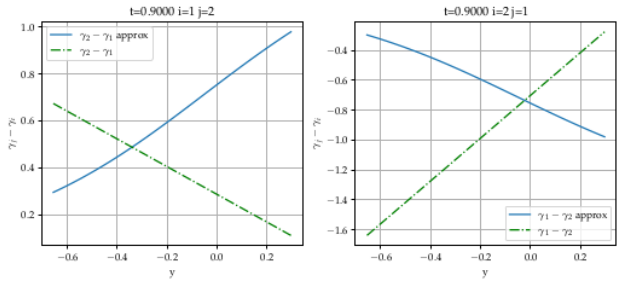

Figure 6.23: Plotting approximations of γ versus analytical values at step $k = N - 1$; left plot shows for state $\gamma_2 - \gamma_1$; right plot shows for state $\gamma_1 - \gamma_2$

6.5.3 Impact of step size

In previous section, we have established the baseline model using $N = 10$. In this section, we experiment with different number of steps N and evaluate the impact on training accuracy and training time. We will keep other hyper-parameters as before. For each N, we will train the network with same number of epochs and same batch size B per epoch. In all cases, the training converges. The results are shown in table 6.7.

N	R	l	epochs	B	$\text{RelErr}_{\text{upper}}$	$\text{RelErr}_{\text{lower}}$	Time(s)
5	2	10	5,000	512	0.0080 ± 0.0026	0.0003 ± 0.0006	00:38
10	2	10	5,000	512	0.0074 ± 0.0022	0.0002 ± 0.0006	01:06
20	2	10	5,000	512	0.0058 ± 0.0022	0.0002 ± 0.0005	02:23
50	2	10	5,000	512	0.0034 ± 0.0024	0.0003 ± 0.0006	06:16

Table 6.7: Relative error and training time with different number of steps N

As shown in figure 6.24, we can see an almost linear relationship between N and training time, and between N and relative error in the upper bound. It is intuitive to see improvement as N increases. For example, recall that we make the assumption α_t only jumps once and only jumps at the begining of the interval. Any error derived from this assumption should reduce as Δt reduces. Similarly, quantisation error from Euler-Maruyama scheme should also reduce as time

step Δt reduces, although this is less significant as r, μ, σ are all time independent. Furthermore, $\text{RelErr}_{\text{lower}}$ does not improve at all, which is surprising.

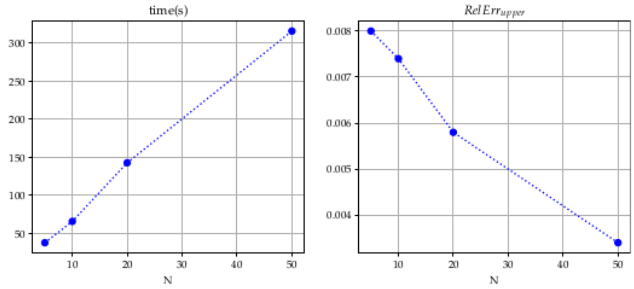

Figure 6.24: Impact of different number of steps N; left plot shows impact on training time and right plot shows impact on relative error in upper bound

6.5.4 Impact of deeper and wider network

In this section, we examine the impact of depth and breadth of neural network by changing number of hidden layers R and hidden nodes per layer l. Intuitively, the deeper and broader the network, the more non-linearity can be modelled by the network. Note that we did not keep the number of epochs the same as bigger models require more epochs to train.

N	R	l	epochs	B	$\text{RelErr}_{\text{upper}}$	$\text{RelErr}_{\text{lower}}$	Time(s)
10	1	10	5,000	512	0.0064 ± 0.0023	0.0004 ± 0.0006	00:38
10	2	10	5,000	512	0.0074 ± 0.0022	0.0002 ± 0.0006	01:06
10	3	10	20,000	512	0.0068 ± 0.0026	0.0004 ± 0.0006	06:17
10	2	20	10,000	512	0.0075 ± 0.0021	0.0003 ± 0.0005	03:06

Table 6.8: Relative error and training time with different hidden layers and hidden nodes

The results are shown in table 6.8. The error bands for $L = 1, 2, 3$ basically overlap each other so any improvement is not statistically significant. It is similar for $l = 10, 20$. This is unsurprising as (3.3.9) and (3.3.12) indicates that γ and q can be expressed as linear functions of Y_t. Therefore, in fact, we can instead fit a linear curve of Y_t. Making the neural network more wider and deeper makes it better for approximating non-linear function which does not help in this case. On the other hand, it increases the risk of overfitting.

6.5.5 Limitation of deep SMP method

As shown in section 6.5.2, neural networks are fitted around only a couple values of the domain. This makes it difficult to learn the shape of the overall solution. For example, in figure 6.21, the curve of approximation is no where close to the analytical solution.

Conclusion

In this study, we successfully designed two numerical algorithms for unconstrained Markov regime-switching quadratic utility maximisation problem. We used analytical solutions to assess the quality of the approximations, and the results are promising. We have also verified theoretical results from previous work with numerical examples. It is worth emphasising that the numerical methods developed in this study can be easily extended to solve problems where analytical solutions cannot be found.

In terms of future work, it will be interesting to deploy the numerical algorithms for Markov regime-switching utility maximisation with constrained controls. The control constraints adds flexibility to the problem but also impose additional complexity. One example of such constraint can be no short-selling. Moreover, the numerical methods can be extended to tackle other utility functions under Markov regime-switching settings.

Appendix A

Technical Proofs

A.1 Extension of 2BSDE

Proof. As stated in [10, Appendix A.1], with u being the value function, take arbitrary i, t and x, the HJB equation for Markov switching problem is:

$$\begin{aligned} 0 &= (\partial_t u) + \sup_{\pi} \left\{ (\partial_x u) \left(r_i(t)x + \pi^T \sigma_i(t)\theta_i(t) \right) + \frac{1}{2} (\partial_{xx} u) \left| \sigma_i(t)^T \pi \right|^2 \right\} + \sum_{j \neq i}^{d} q_{ij} u_j(t,x) \\ &= (\partial_t u) + \sup_{\pi} F(t,i,x,\pi,\partial_x u, \partial_{xx} u) + \sum_{j \neq i}^{d} q_{ij} u_j(t,x) \end{aligned}$$

where F is the Hamiltonian control:

$$F(t,i,x,\pi,z,\gamma) = z \left(r_i(t)x + \pi^\top \sigma_i(t)\theta_i(t) \right) + \frac{1}{2}\gamma \left| \sigma_i(t)^T \pi \right|^2.$$

It would be convenient to simply the HJB equation. Note that

$$\sum_{j \neq i}^{d} q_{ij} \left(u_j(t,x) - u_i(t,x) \right) = \left(\sum_{j \neq i}^{d} q_{ij} u_j(t,x) \right) - u_i(t,x) \underbrace{\sum_{j \neq i}^{d} q_{ij}}_{-q_{ii}} = \sum_{j=1}^{d} q_{ij} u_j(t,x)$$

and this is equivalent to

$$\begin{pmatrix} u_1(t,x) & \dots & u_d(t,x) \end{pmatrix} \begin{pmatrix} q_{11} & q_{21} & \dots & u_{d1} \\ q_{12} & q_{22} & \dots & u_{d2} \\ \dots & \dots & \dots & \dots \\ q_{1d} & q_{2d} & \dots & u_{dd} \end{pmatrix} e_i = \boldsymbol{u}^\top Q^\top e_i$$

where $\boldsymbol{u} := (u_1(t,x), \dots, u_d(t,x))^\top$. Hence we can simplify the notation as

$$0 = (\partial_t u) + F(t,i,x,\pi^*,\partial_x u, \partial_{xx} u) + \boldsymbol{u}^\top Q^\top e_i$$

where π^* is the strategy that solves supremum.

Let π be the optimal control, X be the associated wealth process and α be the Markov chain, we have:

$$0 = (\partial_t u) + F(t, \alpha_t, X_t, \pi_t, \partial_x u, \partial_{xx} u) + \boldsymbol{u}_t^\top Q^\top \alpha_t. \tag{A.1.1}$$

where $\boldsymbol{u}_t := (u_1(t,X_t), \dots, u_d(t,X_t))^\top$.

Next, define processes

$$V_t := u_{\alpha_t}(t, X_t). \tag{A.1.2}$$

Apply Ito's formula:

$$\mathrm{d}V_t = (\partial_t u)\mathrm{d}t + (\partial_x u)\mathrm{d}X_t + \frac{1}{2}(\partial_{xx} u)\mathrm{d}[X]_t + (V_t - V_{t-}) \tag{A.1.3}$$

where $(V_t - V_{t-})$ represents the jump in V_t when Markov chain jumps. Recall that $\alpha_t \in \mathbb{R}^d$, it is more convenient to write the jump term as:

$$V_t - V_{t-} = \boldsymbol{u}_t^\top(\alpha_t - \alpha_{t-}) = \boldsymbol{u}^\top \mathrm{d}\alpha_t.$$

From [5, Appendix B], there is a martingale process M associated with the Markov chain α:

$$M_t = \alpha_t - \int_0^\top Q^\top \alpha_s \mathrm{d}s.$$

Here, Q is the generator matrix of α. Hence the jump term is

$$V_t - V_{t-} = \boldsymbol{u}_t^\top Q^\top \alpha_t \mathrm{d}t + \boldsymbol{u}_t^\top \mathrm{d}M_t.$$

Substituting above equation back into A.1.3 and expand the other terms, we arrive at

$$\begin{aligned}
\mathrm{d}V_t &= \left\{ (\partial_t u) + (\partial_x u)\left(r_{\alpha_t} X_t + \pi_t^\top \sigma_{\alpha_t}\theta_{\alpha_t}\right) + (\partial_{xx} u)\cdot\frac{1}{2}\left|\sigma_{\alpha_t}^T \pi_t\right|^2 \right\}\mathrm{d}t \\
&\quad + (\partial_x u)\pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t + \boldsymbol{u}_t^\top Q^\top \alpha_t \mathrm{d}t + \boldsymbol{u}_t^\top \mathrm{d}M_t \\
&= \left\{ (\partial_t u) + F\left(t, \alpha_t, X_t, \pi_t, \partial_x u, \partial_{xx} u\right) + \boldsymbol{u}_t^\top Q^\top \alpha_t \right\}\mathrm{d}t + (\partial_x u)\pi_t^\top \sigma_{\alpha_t}\mathrm{d}W_t + \boldsymbol{u}_t^\top \mathrm{d}M_t,
\end{aligned}$$

and note that the first term is zero as it is the HJB A.1.1, so we have

$$\mathrm{d}V_t = (\partial_x u)\pi_t^\top \sigma_{\alpha_t}\mathrm{d}W_t + \boldsymbol{u}_t^\top \mathrm{d}M_t.$$

Next, define

$$Z_t := \partial_x u_{\alpha_t}(t, X_t). \tag{A.1.4}$$

We similarly apply Ito

$$\begin{aligned}
\mathrm{d}Z_t &= \left(\frac{\partial^2 u}{\partial x \partial t}\right)\mathrm{d}t + (\partial_{xx} u)\mathrm{d}X_t + \frac{1}{2}\left(\frac{\partial^3 u}{\partial x^3}\right)\mathrm{d}[X]_t + (Z_t - Z_{t-}) \\
&= \left(\frac{\partial^2 u}{\partial x \partial t}\right)\mathrm{d}t + (\partial_{xx} u)\left\{\left(r_{\alpha_t} X_t + \pi_t^\top \sigma_{\alpha_t}\theta_{\alpha_t}\right)\mathrm{d}t + \pi_t^\top \sigma_{\alpha_t}(t)\mathrm{d}W_t\right\} \\
&\quad + \frac{1}{2}\left(\frac{\partial^3 u}{\partial x^3}\right)\left|\sigma_{\alpha_t}^T \pi_t\right|^2 \mathrm{d}t + (Z_t - Z_{t-}).
\end{aligned}$$

Again, we can rewrite the jump term $Z_t - Z_{t-}$ as

$$Z_t - Z_{t-} = \boldsymbol{z}_t^\top(\alpha_t - \alpha_{t-}) = \boldsymbol{z}_t^\top Q^\top \alpha_t \mathrm{d}t + \boldsymbol{z}_t^\top \mathrm{d}M_t,$$

where $\boldsymbol{z}_t := \partial_x \boldsymbol{u}_t$. Furthermore, by HJB A.1.1, we can write

$$\begin{aligned}
\frac{\partial^2 u}{\partial x \partial t} = \frac{\partial}{\partial x}\frac{\partial}{\partial t}u &= -\frac{\partial}{\partial x}\left\{F(t, \alpha_t, X_t, \pi_t, \partial_x u, \partial_{xx} u) + \boldsymbol{u}_t^\top Q^\top \alpha_t\right\} \\
&= -\frac{\partial}{\partial x}F(t, \alpha_t, X_t, \pi_t, \partial_x u, \partial_{xx} u) - \boldsymbol{z}_t^\top Q^\top \alpha_t.
\end{aligned}$$

Further expanding,

$$\begin{aligned}
\frac{\partial}{\partial x}F(t, \alpha_t, X_t, \pi_t, \partial_x u, \partial_{xx} u) &= \frac{\partial}{\partial x}\left\{(\partial_x u)\left(r_{\alpha_t} X_t + \pi_t^T \sigma_{\alpha_t}\theta_{\alpha_t}\right) + \frac{1}{2}(\partial_{xx} u)\left|\sigma_{\alpha_t}^T \pi_t\right|^2\right\} \\
&= (\partial_{xx} u)\left(r_{\alpha_t} X_t + \pi_t^T \sigma_{\alpha_t}\theta_{\alpha_t}\right) + (\partial_x u) r_{\alpha_t} + \frac{1}{2}\left(\frac{\partial^3 u}{\partial x^3}\right)\left|\sigma_{\alpha_t}^T \pi_t\right|^2.
\end{aligned}$$

Putting all terms together, we have

$$\mathrm{d}Z_t = -(\partial_x u) r_{\alpha_t}\mathrm{d}t + (\partial_{xx} u)\pi_t^\top \sigma_{\alpha_t}\mathrm{d}W_t + \boldsymbol{z}_t^\top \mathrm{d}M_t$$

Finally, we define

$$\Gamma_t := \partial_{xx} u_{\alpha_t}(t, X_t).$$

Tidying up the terms, we get:

$$\begin{aligned}
\mathrm{d}V_t &= Z_t \pi_t^\top \sigma_{\alpha_t} \mathrm{d}W_t + \boldsymbol{u}_t^\top \mathrm{d}M_t \\
\mathrm{d}Z_t &= -Z_t r_{\alpha_t} \mathrm{d}t + \Gamma_t \pi_t^\top \sigma_{\alpha_t} \mathrm{d}W_t + \boldsymbol{z}_t^\top \mathrm{d}M_t
\end{aligned}$$

□

Bibliography

[1] N. Azevedo, D. Pinheiro, and G. W Weber. Dynamic programming for a markov-switching jumpdiffusion. *Journal of computational and applied mathematics*, 267:1–19, Sep 2014.

[2] Richard Bellman. *Dynamic programming*. Princeton University Press, Princeton N.J, 1957. Incudes bibliographical references and index.; ID: alma991448814401591.

[3] Baojun Bian, Sheng Miao, and Harry Zheng. Smooth value functions for a class of nonsmooth utility maximization problems. *SIAM journal on financial mathematics*, 2(1):727–747, 2011.

[4] Ashley Davey and Harry Zheng. Deep learning for constrained utility maximisation. *Methodology and computing in applied probability*, 24(2):661–692, Nov 26, 2021.

[5] Robert J. (Robert James) Elliott, Lakhdar Aggoun, and John B. (John Barratt) Moore. *Hidden Markov models estimation and control*. Springer-Verlag, New York, 1995.

[6] Mehrali Hemmatinezhad, Mohammad Gholizadeh, Mohammadrahim Ramezaniyan, Shahram Shafiee, and Amin Ghazi Zahedi. Predicting the success of nations in asian games using neural network. 2022.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ID: 271125.

[8] Sahdev Kansal. Quick guide to gradient descent and its variants.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[10] Anand Krishnakumar and Harry Zheng. Utility maximisation in an incomplete market. Technical report, 2022.

[11] Yusong Li and Harry Zheng. Weak necessary and sufficient stochastic maximum principle for markovian regime-switching diffusion models. *Applied mathematics and optimization*, 71(1):39–77, May 07, 2014.

[12] Yusong Li and Harry Zheng. Constrained quadratic risk minimization via forward and backward stochastic differential equations. *SIAM journal on control and optimization*, 56(2):1130–1153, Jan 2018.

[13] Huyen Pham. *Continuous-time stochastic control and optimization with financial applications*. Springer, Dordrecht, 2009.

[14] Endre Sli and D. F. (D Mayers. *An introduction to numerical analysis*. Cambridge University Press, Cambridge, 2003.

[15] Wikipedia contributors. Backpropagation — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=1104872812`, 2022. [Online; accessed 3-September-2022].