Doctoral Thesis

Evaluating and Mitigating Transaction Costs with Recurrent Neural Networks

Author:

Ran Li

BSc, MSc

Supervisors:

Prof. Paolo Guasoni,

Dublin City University

Dr. Kwok Chuen Wong,

Dublin City University

Dissertation submitted in part fulfilment of the requirement for the award of

Doctor of Philosophy

in the

School of Mathematical Sciences

Dublin City University

August, 2024

Declaration of Authorship

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Ran Li

ID No.: 19215274

Date: 28/8/2024

Acknowledgements

I would first express my sincere thanks to my advisors, Prof. Paolo Guasoni, for his unwavering, patience, guidance and support throughout my research; Dr. Kwok Chuen Wong for his kind advisory for my research and career. Their expertise and insight were irreplaceable, and their encouragement kept me motivated during challenging times, especially during the pandemic. I would like to extend my gratitude to Prof. Thomas Conlon from UCD, Smurfit Business School, who has given useful advice on benchmark.

I am grateful to the administrative and technical staff at Dublin City University, whose programmes, assistance and support formulated the logistical aspects of my research. Your dedication and efficiency are greatly appreciated. I also thank my colleagues and friends in school of mathematics, for providing a supportive and stimulating environment. I also appreciated the accommodation provided by DCU during the pandemic when it is difficult to find somewhere to live, especially in Dublin.

I would like to acknowledge the financial support from Science Foundation Ireland (SFI), which made this research possible.

Best thanks to my family, especially my parents, for their unwavering support, encouragement, and love. Your belief in me has been my constant source of strength.

The thesis is dedicated to everyone who has been a part of my academic journey. Thank you for being there and for trusting in me. Although the PhD study is to be finished, my love to you will last.

List of Publications & Presentations

The works included in this thesis have been presented in:

- Workshop on Stochastic Control, Machine Learning, and Quantitative Finance (2024) by Prof. Paolo Guasoni
- Irish Academy of Finance (IAF) 5^{th} Annual Conference (2024) by Ran Li

Contents

List of Figures

List of Tables

Notations & Explanations

Notations:

d Number of assets in a portfolio.

T Time of maturity.

t Time point t, $t \leq T$.

$t-$ Intermediate time before changes are made at t.

S_t^i Price of the i^{th} asset at time t.

r_t^i Return of the i^{th} asset during time interval $[t-1, t]$.

r_t Risk free rate during time $[t-1, t]$.

θ_t^i Number of shares of the i^{th} asset held at time t.

π_t^i Weight invested in the i^{th} asset at time t.

μ^i Drift of the i^{th} asset.

σ^i Volatility of the i^{th} asset.

X_t Wealth of a portfolio at time t.

Σ Covariance matrix of the portfolio.

γ Risk aversion parameter of power utility.

ε_i Transaction cost of the i^{th} asset.

$r_t^{\pi,\varepsilon}$ Portfolio return with strategy π and transaction cost ε during time $[t-1,t]$.

N Number of paths in a Monte Carlo simulation.

w_i The i^{th} sample in the simulated data.

ϵ Tolerance level.

e_i Identity vector of the i^{th} dimension.

ξ_i i^{th} eigenvalue of a matrix.

P, Q Probability measure.

Explanations:

ETF Exchange-Traded Fund.

HJB Hamilton-Jacobi-Bellman.

ESR Equivalent safe rate.

RNN Recurrent Neural Network.

CNN Convolutional Neural Network.

DNN Deep Neural Network.

NTR No-trade region.

EQL An equally-weighted sector ETF fund which can be found in ALPS [2023].

DUBLIN CITY UNIVERSITY

Abstract

Faculty of Science and Health

School of Mathematical Sciences

Doctor of Philosophy

Evaluating and Mitigating Transaction Costs with Recurrent Neural Networks

by Ran Li

Abstract

This thesis develops a method for evaluating and mitigating the effect of transaction costs on trading strategies with many assets. An iteration procedure yields the cost-adjusted portfolio return, enabling the formulation of portfolio-choice problems as optimization of Recurrent Neural Networks (RNN). This method reproduces the theoretical results available for one risky asset and the numerical approximations available for two risky assets through finite-elements. Crucially, the RNN model scales to several assets and is fully interpretable, as its parameters identify their no-trade region. Importance-sampling significantly enhances the model's performance, especially with several assets. An application to equally-weighted funds demonstrates the method's ability to reduce both tracking error and tracking difference from an empirical target.

Chapter 1

Introduction

1.1 Background

According to iShares [2023], Exchange-Traded Funds (ETFs) are a significant fraction of the total global financial market in both equities and fixed income, ranging from 4.4% to 12.7% of equities, and continue to increase. This indicates a high demand for asset management over the world. Managing assets in a frictionless market is simple - fixed proportions invested in risky assets respectively is the optimal solution (Merton [1969], Merton [1971]) based on the assumption of Geometric Brownian Motion risky assets. However, real markets have several frictions. Possible transaction costs consist of bid-ask spread (whereby an investor has to buy at higher price than they could sell for), broker fees, and market impact for specific investors. iShares [2023] also points out that the US market witnesses an average 12 bp bid-ask spread in the US ETF market, while this number is even higher in Europe and Asian-Pacific market. The stock market is a little bit more liquid, as the average bid-ask spread of NYSE and NASDAQ are around 0.5% and 1% respectively.

Investors who ignore transaction costs and stick to their target strategy may suffer a loss and go bankrupt in the worst scenario. A general way to fix the problem is to perform a buy-and-hold strategy, thereby reducing trading frequency. However, no explicit solution is made for such a question with large portfolios. When one asks what is exactly the optimal strategy, what are the impacts of such transaction costs, and how

to evaluate the performance of a portfolio accurately with such costs, the problem soon turns into solving a n-dimensional nonlinear stochastic partial differential equation (SPDE), which is too complicated in general.

Although a number of numerical solutions has been made to solve the problem and impressive achievements have been obtained, most studies restrict the number of risky assets to a small number (i.e, two risky assets). We want to study this question through machine learning methods with several risky assets.

1.2 Literature

Markowitz [1952] was a pioneer in the development of the quantitative portfolio optimization problem. Merton [1969] and Merton [1971] later addressed the portfolio optimization problem in a continuous market, demonstrating that an investor should maintain a constant proportion of safe and risky assets. However, the presence of transaction costs complicates the problem, as sellers receive less than the market price. Consequently, the Merton strategy would inevitably lead to bankruptcy if executed continuously.

Magill and Constantinides [1976] were the first to investigate the Merton problem in the context of transaction costs, concluding that a no-trade region exists wherein the investor should refrain from trading. Davis and Norman [1990] explored the theoretical characterization of the partial differential equation (PDE) for determining the no-trade region. Liu and Loewenstein [2002] established the boundary in a finite horizon, while Liu [2004] examined the multi-dimensional model with uncorrelated assets. Guasoni and Muhle-Karbe [2015] and Gerhold et al. [2014] derived the explicit form of the boundaries in a market with a single riskless and risky asset subject to bid-ask spread. Bichuch and Guasoni [2018] studied a two-asset model featuring correlated liquid and illiquid assets. Brown and Smith [2011] constructed a dynamic model to approximate the value function directly. Other relevant research includes asymptotic analysis and numerical solutions (Muthuraman and Kumar [2004], Altarovici et al. [2016], Chen et al. [2022]), rebalancing frequencies (Ekren and Liu [2018], Sun et al. [2006]), and

the consumption problem (Gerhold et al. [2013]).

This thesis is primarily concerned with explicit and numerical solutions of optimal trading strategy with transaction costs. In a market where the bid-ask spread $\varepsilon \in (0, 1)$ represents the sole transaction cost, sellers receive only $(1 - \varepsilon)S_t$, where S_t denotes the asset prices. This study aims to develop a potential machine learning algorithm for numerically determining the optimal strategy in a market comprising multiple correlated assets with varying transaction costs. Most related research tends to employ Convolutional Neural Networks (CNN) and Reinforcement Learning (RL) to pinpoint the optimal strategy or identify trading opportunities by analyzing financial signals (Deng et al. [2017], Ugur Gudelek et al. [2018]). Zhang and Zhou [2019] utilized an RL model and compared the resulting no-trade region with PDE solutions but did not fully address the impact of correlation. Gaegauf et al. [2023] figure out a way to model the no-trade regions with machine learning, Gaussian process regressions and dynamic programming. Most of the listed works focus on one risky asset or two risky assets. However, we are more interested in studying the no-trade regions of multiple dimensional assets from their geometric implication, comparing the outcomes with existing results and validating empirical results.

1.3 Contributions

To achieve this objective, we first develop an algorithm-friendly function that can serve as a layer in the machine learning model, calculating the overall return of a trading strategy under specific transaction costs. Next, we create the asymptotic boundaries derived from explicit and numerical solutions presented Gerhold et al. [2014], Muthuraman and Kumar [2004] and Bichuch and Guasoni [2018] in the form of activation functions for the RNN model. Our Recurrent Neural Network (RNN) model can produce accurate solutions for a single risky asset and approximate solutions for two risky assets, and can be extended to higher dimensions. The simulated asset data is processed through the activation function and return function before reaching the loss function, followed by backpropagation. The model is trained multiple times,

and the resulting solutions are compared with relevant Partial differential equations (PDEs) and numerical results.

Results indicate that the single-asset model demonstrates excellent performance with rapid convergence, and the two-asset model exhibits similar results. The high-dimensional model provides approximations of no-trade regions, which have been effectively tested in 3-asset and 10-asset systems. However, the high-dimensional model consumes significantly more system resources and memory than the one-asset and two-asset models, and this issue has been partially resolved with the importance sampling method (Guasoni and Robertson [2008]). A hyperellipsoid method is also introduced to help the convergence of high dimension problems at the cost of less accuracy. Such method is partially effective and need extra improvement to reduce the side effect of unforeseen gradients. An empirical study to equally-weighted ETF fund EQL demonstrates the method's ability to reduce both tracking error and tracking difference from a target trading strategy.

In conclusion, the novel contributions of this thesis consist of:

(i) Use of RNN for portfolio management with transaction costs.

(ii) Development of an iterative procedure for portfolio optimization with transaction costs.

(iii) Comparison of performance of the proposed method with available explicit results.

(iv) Development of machine learning algorithms that are applicable to portfolios with many assets.

(v) Proposal of hyperellipsoid no-trade regions as practical compromise between performance and tractability.

(vi) Performance-enhancement through the application of importance-sampling.

1.4 Thesis structure

This rest of the thesis is organised as follows:

- In Chapter 2, we present an approach for replicating a portfolio return with trading costs. The approach exploits iterations to obtain fast convergence of portfolio returns with transaction costs. We also design a recurrent neural network to calculate such portfolio return from relevant assets. In detail, the model takes assets returns, costs, and a target strategy as inputs and outputs a customised loss function.

- In Chapter 3, we demonstrate the formulation of the one-asset and two-assets model. Both models are based on the utilization of double-ReLU functions. Monte Carlo methods are employed for simulating numerical results, which are compared to existing outcomes. ESR (Equivalent safe rate) is introduced to evaluate model outputs.

- In Chapter 4, we formulate a multiple assets model by constructing a vectorization and bisection method with approaches to reduce complexity. A simplified hyperellipsoid method is then included to help the convergence with higher dimension. In addition, this chapter pertinent examples featuring more than two risky assets: simulated portfolios with 2, 3, and 10 risky assets. 3-D no-trade region of 3 assets and ESR are also presented.

- In Chapter 5, we introduce a number of variance reduction methods among which we choose the importance sampling method. We prove the change of measure of importance sampling for the wealth process with transaction costs and apply the method on high dimensional assets model. We compare simulation results with and without variance reduction, to ascertain the performance increase.

- In Chapter 6, we introduce tracking error and tracking difference to evaluate ETF (Exchange-traded fund) performance. We import an equally-weighted ETF (EQL) to demonstrate the RNN method's ability to reduce both tracking error

and tracking difference from an empirical target. The performance of the hyper-ellipsoid method is also examined.

- In Chapter 7, we conclude the thesis, discussing the future of this work and limitations, outlining some potential research directions for the future.

Chapter 2

Portfolio Returns Model

This chapter presents a novel approach for constructing a portfolio return with transaction costs. The approach is suggested to be working with iterations and algorithm friendly. Mathematical proof of convergence is presented and an example is also introduced. Then an structure of recurrent neural network is designed to calculate such portfolio return from relevant assets. In detail, the model takes assets returns, costs a target strategy as input and generates loss function through a layer with customised activation function which will be modeled in following chapters.

2.1 Portfolio Returns with Proportional Trading Costs

In a financial market with d risky assets, denote by r_t^i the simple return of the i-th risky asset in the period $[t-1, t]$ and by r_{t-1} the safe return (the subscript of each quantity reflects the time it is observed). In the absence of trading costs, a portfolio that holds a fraction of wealth π_{t-1}^i in the i-risky asset has the familiar return

$$r_t^{\pi,0} = \left(1 - \sum_{i=1}^{d} \pi_{t-1}^i\right) r_t + \sum_{i=1}^{d} \pi_{t-1}^i r_t^i, \tag{2.1}$$

which is the weighted average, using portfolio weights, of the assets' returns during the same period.

In the presence of proportional trading costs, portfolios' returns are no longer

weighted averages of assets' returns. In fact, such returns are not even determined by assets' returns alone, as they must include also the cost of adjusting a portfolio at the end of the period.[1]

If trading the i-th asset incurs a proportional transaction cost of ε^i, in that the ask price at time t is $S_t^i(1+\varepsilon^i)$ while the bid price is $S_t^i(1-\varepsilon^i)$, the following result shows how to obtain the return with trading costs of a solvent trading strategy through simple iteration.

Proposition 2.1.1. *Let the prices S_t^i be positive. If a solvent strategy $\pi = (\pi_t^i)_{0\le t<\infty}^{1\le i\le d}$ satisfies the condition $L := \sup_{1\le t\le T}\sum_{i=1}^d \varepsilon^i|\pi_t^i| < 1$, then the function*

$$F^{\pi,\varepsilon}(x) := \sum_{i=1}^d \pi_{t-1}^i r_t^i + \left(1-\sum_{i=1}^d \pi_{t-1}^i\right) r_t - \sum_{i=1}^d \varepsilon^i|\pi_t^i(1+x) - \pi_{t-1}^i(1+r_t^i)|, \quad (2.2)$$

has a fixed point, and it is the return r_t^π of the strategy π with transaction costs. The sequence defined by $r_t^{\pi,0}$ as in (2.1) and $r^{\pi,n+1} = F^{\pi,\varepsilon}(r^{\pi,n})$ converges to r_t^π and

$$|r^\pi - r^{\pi,k}| \le \frac{L^k}{1-L}\sum_{i=1}^d \varepsilon^i \left|\pi_t^i(1+r_t^{\pi,0}) - \pi_{t-1}^i(1+r_t^i)\right|. \quad (2.3)$$

Proof. Denoting by θ_t^i the number of shares of the i-th asset held at time t in the portfolio and by X_t the portfolio value at time t, the budget equation with proportional transaction costs is

$$X_t - X_{t-1} = \sum_{i=1}^d \theta_{t-1}^i(S_t^i - S_{t-1}^i) + \left(X_{t-1} - \sum_{i=1}^d \theta_{t-1}^i S_{t-1}^i\right) r_{t-1} - \sum_{i=1}^d \varepsilon^i S_t^i|\theta_t^i - \theta_{t-1}^i| \quad (2.4)$$

Dividing the right-hand side by X_{t-1},

$$\frac{X_t - X_{t-1}}{X_{t-1}} = \sum_{i=1}^d \frac{S_{t-1}^i\theta_{t-1}^i}{X_{t-1}}\cdot\frac{S_t^i - S_{t-1}^i}{S_{t-1}^i} + \left(1-\sum_{i=1}^d \frac{\theta_{t-1}^i S_{t-1}^i}{X_{t-1}}\right) r_{t-1} - \sum_{i=1}^d \varepsilon^i \frac{S_t^i}{X_{t-1}}|\theta_t^i - \theta_{t-1}^i|$$

[1] Adjustment at the beginning or at the end of the period is a matter of convention. We adopt the latter convention so as to consider only the two portfolio weights π_{t-1} and π_t in the period $[t-1,t]$ rather than $\pi_{t-2},\pi_{t-1},\pi_t$.

Denoting by $r_t^\pi = \frac{X_t - X_{t-1}}{X_{t-1}}$ the portfolio return, by $r_t^i = \frac{S_t^i - S_{t-1}^i}{S_{t-1}^i}$ the return on the i-th asset, and by $\pi_t^i = \frac{\theta_t^i S_t^i}{X_t}$ the portfolio weight of the i-th asset, it follows that

$$r_t^\pi = \sum_{i=1}^d \pi_{t-1}^i r_t^i + \left(1 - \sum_{i=1}^d \pi_{t-1}^i\right) r_{t-1} - \sum_{i=1}^d \varepsilon^i \frac{S_t^i}{X_{t-1}} |\theta_t^i - \theta_{t-1}^i| \tag{2.5}$$

To rewrite also the last term as a function of weights and returns, note that, because both S_t^i and X_t are positive (the latter by solvency):

$$\begin{aligned}\sum_{i=1}^d \varepsilon^i \frac{S_t^i}{X_{t-1}} |\theta_t^i - \theta_{t-1}^i| &= \sum_{i=1}^d \varepsilon^i \left| \frac{\theta_t^i S_t^i}{X_t} \frac{X_t}{X_{t-1}} - \frac{\theta_{t-1}^i S_{t-1}^i}{X_{t-1}} \frac{S_t^i}{S_{t-1}^i} \right| \\ &= \sum_{i=1}^d \varepsilon^i \left| \pi_t^i (1 + r_t^X) - \pi_{t-1}^i (1 + r_t^i) \right|\end{aligned}$$

Thus, equation (2.5) becomes:

$$\begin{aligned} r_t^\pi &= \sum_{i=1}^d \pi_{t-1}^i r_t^i - \sum_{i=1}^d \varepsilon^i \left| \pi_t^i (1 + r_t^\pi) - \pi_{t-1}^i (1 + r_t^i) \right| + \left(1 - \sum_{i=1}^d \pi_{t-1}^i\right) r_{t-1} \\ &= F^{\pi,\varepsilon}(r_t^\pi) \end{aligned} \tag{2.6}$$

which identifies r_t^π as a fixed point of the map F.

To establish the convergence of the sequence $(r^{\pi,n})_{n\geq 1}$, it suffices to show that F is a contraction: for some $\delta > 0$ it holds that $|F(x) - F(y)| \leq (1-\delta)|x - y|$ for all $x, y \in \mathbb{R}$, that is, F is Lipschitz continuous with constant strictly less than one. Thus, it is enough to check that the derivative of F is defined up to a finite number of points, and that its absolute value is strictly less than one. Indeed,

$$|(F^{\pi,\varepsilon})'(x)| = \left| -\sum_{i=1}^d \varepsilon^i \pi_t^i \operatorname{sgn}\left(\pi_t^i (1 + x) - \pi_{t-1}^i (1 + r_t^i)\right) \right| \leq \sum_{i=1}^d \varepsilon^i |\pi_t^i| \leq L < 1. \tag{2.7}$$

Thus, F has a unique fixed point, to which every iteration converges. In particular, starting the iteration at the frictionless return $r_t^{\pi,0}$ yields

$$|r_t^{\pi,n} - r_t^{\pi,n-1}| \leq L^{n-1} |r_t^{\pi,1} - r_t^{\pi,0}| = L^{n-1} \sum_{i=1}^d \varepsilon^i \left| \pi_t^i (1 + r_t^{\pi,0}) - \pi_{t-1}^i (1 + r_t^i) \right| \tag{2.8}$$

whence, as claimed,

$$|r_t^\pi - r_t^{\pi,k}| \leq \sum_{n=k+1}^{\infty} |r_t^{\pi,n} - r_t^{\pi,n-1}| \leq \frac{L^k}{1-L} \sum_{i=1}^{d} \varepsilon^i \left| \pi_t^i (1 + r_t^{\pi,0}) - \pi_{t-1}^i (1 + r_t^i) \right| \quad (2.9)$$

□

Proposition 2.1.1 yields a simple procedure to compute a portfolio's return with transaction costs. Starting from the frictionless return, iterate the map in equation (2.3) a few times until the desired precision is reached. Equation (2.9) yields a pointwise bound on the error in terms of the frictionless return. Note also that Proposition 2.1.1 only requires portfolio weights and assets' returns, not asset prices and number of shares, thereby superseding the calculation of such auxiliary quantities.

The iteration converges under the mild condition $\sum_{i=1}^{d} \varepsilon^i |\pi_t^i| < 1$, which is satisfied in any realistic situation. For example, the condition holds if $(\max_{1 \leq i \leq d} \varepsilon^i) \sum_{i=1}^{d} |\pi_t^i| < 1$, which means that the total risky position, as a multiple of the portfolio value, does not exceed the reciprocal of the transaction cost on the most illiquid asset. Even in the extreme case of $\varepsilon = 5\%$, the restriction is merely that leverage does not exceed 20, which is well beyond the margin requirements that any broker would place on such an asset.

Two or three iterations are sufficient in typical settings. For the sake of prudence, all the results in this thesis are derived with five iterations. (Table 2.1 shows the effect of transaction costs at various iterations for the momentum, which is known to be sensitive to frictions.) A computational advantage a fixed number of iterations is that the return's calculation is seamless to integrate with automatic differentiation, enabling its use in conjunction with machine learning frameworks.

2.2 Recurrent Neural Network

Recurrent Neural Networks (RNNs, Rumelhart et al. [1986] and Elman [1990]) constitute a category of deep neural networks (DNNs) characterized by connections between nodes across a given sequence, enabling the simulation of temporal dynamic behavior

Iterations (n)	0	1	2	3	4	5
Return	1.56	0.83	0.68	0.68	0.68	0.68
Trading costs	0	0.73	0.88	0.88	0.88	0.88

Table 2.1: Average monthly returns (August 1963 to November 2023)

Value-weighted momentum strategy (Jegadeesh and Titman [1993]) replicated with the method of Novy-Marx and Velikov [2016] and transaction costs from Hasbrouck [2009]. Each column shows the time average return $r_t^{r,\pi}$ (first row) and the average transaction costs (second row) obtained from n iterations of the map in (2.2).

through subsequent layers. Owing to such features, RNNs are particularly effective for tasks where the context of previous inputs is crucial, such as time series analysis (Connor et al. [1994]), natural language processing (Li and Wu [2015]), and speech recognition. Unlike traditional neural networks, RNNs have loops that allow information to persist, enabling them to maintain a "memory" of previous inputs. A simple RNN exemplifies a fully connected RNN, where in the outputs of all neurons are connected to the inputs of all neurons. At each time step, the input is propagated forward in conjunction with the preceding values of the hidden layer. Consequently, the RNN can retain essential information from the previous layer, facilitating prediction.

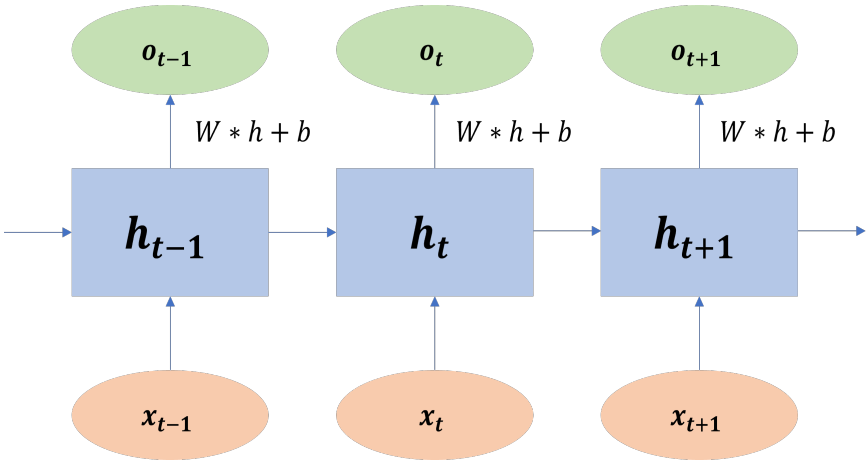


Figure 2.1: Structure of A Basic RNN

The framework contains three components: input, hidden layer and output, which follows the corresponding equation:

$$h_t = \sigma_h(W_x x_t + W_h h_{t-1} + b_h) \tag{2.10}$$

$$o_t = \sigma_o(W_o h_t + b_o) \tag{2.11}$$

where x_t is the input data, h_t is the value of hidden layer and o_t is the output. σ_h is the activation function and σ_o is a linear layer.

Long Short-Term Memory (LSTM) networks, introduced by Hochreiter [1997], target at the vanishing gradient problem by including a more complicated structure. An LSTM unit usually consists of a cell, an input gate, an output gate, and the most important forget gate. These gates manage the sequence information, allowing the network to retain or drop information when needed. This architecture enables LSTMs to capture long-range dependencies more effectively than standard RNNs. Cho [2014] came up with the Gated Recurrent Unit (GRU), which is a simplified version of LSTM. It inherits the core concept of LSTM by combining the input and forget gates into a single gate and merge the cell state with hidden state. This architecture reduces the complexity while maintaining performance comparable to basic LSTMs.

Though most application of RNNs are on language studies, researchers find them suitable for time series analysis. In accordance to our target, we would replicate the rebalancing behavior of a portfolio manager step by step. Therefore, RNN is selected as the fundamental model for our study. As the information we use for training are part of Monte Carlo simulation, we will start with the basic RNN without a forget/update gate. LSTMs and GRUs might be also included if necessary.

2.3 RNN with trading costs

It is expected that the optimal trading strategy with transaction cost can be approximated through the implementation of a machine learning model, characterized by a specialized double ReLU activation function and a corresponding loss function. Figure 2.2 provides a general work procedure of the expected machine learning model. The model takes a target strategy as input and the layer with activation function is supposed to output a modified strategy with respect to the target. The modified strategy is then used to generalize a sequence of returns for creating the loss.

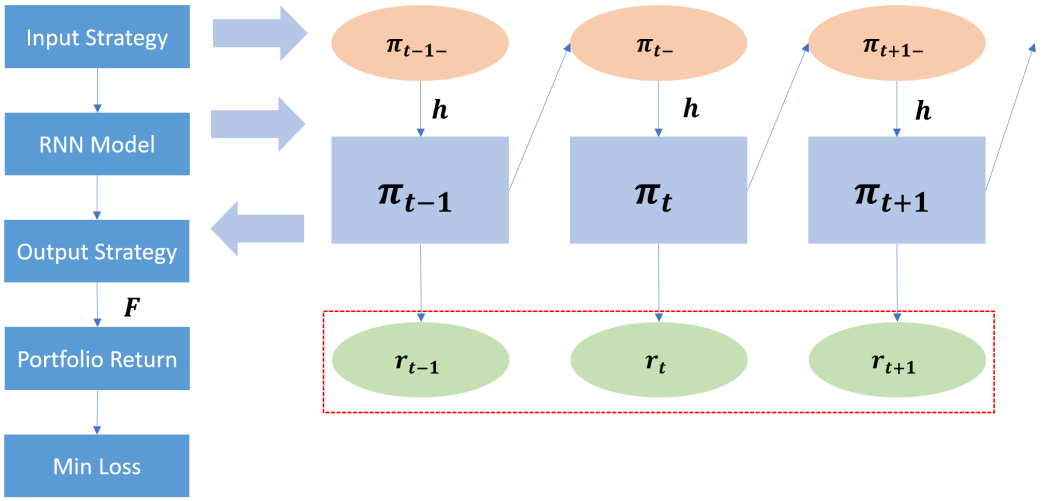


Figure 2.2: A high-level overview of the model in Section 3.1.1

In addition, the RNN model relevant portfolio returns with respect to individual assets.

Chapter 3

One And Two Dimensional Models

In this chapter, we will demonstrate model formulation related results using several examples. Initially, we examine the case of a single risky asset, construct the model utilizing the double ReLU as the activation function, and compare the results of our deep learning no-trade region with the theoretically optimal strategy (Gerhold et al. [2014]). Subsequently, we attempt to replicate a two-risky-assets no-trade region model where the activation function is replaced by a double ReLU based parallelogram and compare the results with proven numerical outcomes (Altarovici et al. [2016]). All corresponding results showcase the impressive performance of the RNN model. Monte Carlo methods will be employed for all simulations to compute the expected utility. The algorithm will operate on the following platform:

Platform	Framework	CUDA
Google Colab	Pytorch 2.2	12.3

3.1 One Asset

Consider a market with a constant safe rate r and a single risky asset with constant excess return $\mu \in \mathbb{R}$ and volatility $\sigma > 0$, i.e., with price S_t described by the diffusion

$$\frac{dS_t}{S_t} = (\mu + r)dt + \sigma dW_t, \tag{3.1}$$

where W_t is a Brownian Motion. If the transaction costs ε are small, the optimal strategy for an investor with isoelastic utility $U(x) = x^{1-\gamma}/(1-\gamma)$ in this market is not to trade as long as the portfolio weight π_t remains in the region $[\pi_-, \pi_+]$, where

$$\pi_\pm = \pi_* \pm \left(\frac{3}{2\gamma}(\pi^*)^2(1-\pi^*)^2\right)^{1/3} \varepsilon^{1/3} + O(\varepsilon)^{2/3} \tag{3.2}$$

and $\pi^* = \frac{\mu}{\gamma\sigma^2}$ denotes the Merton portfolio, while buying (resp. selling) minimally to restore the weight to π_- (resp. π_+), as to restore it to the interval $[\pi_-, \pi_+]$.[1]

At the first order, this strategy is optimal for agents that maximize expected utility from (i) consumption in infinite horizon (Shreve and Soner [1994], Janeček and Shreve [2004]), (ii) terminal wealth on a finite horizon (Bichuch [2012]), and (iii) terminal wealth on a long horizon (Gerhold et al. [2014]).

A convenient measure of the expected utility from terminal wealth is the *equivalent safe rate*, defined as the hypothetical safe rate that would make an investor indifferent between investing optimally and earning such rate on all wealth:

$$ESR_T = \frac{1}{T}\log E[X_T^{1-\gamma}]^{\frac{1}{1-\gamma}} \tag{3.3}$$

With one asset, the equivalent safe rate has the expression Bichuch [2012], Gerhold et al. [2014]

$$\lim_{T\to\infty} ESR_T = r + \frac{\mu^2}{2\gamma\sigma^2} - \frac{\gamma\sigma^2}{2}\left(\frac{3}{2\gamma}(\pi^*)^2(1-\pi^*)^2\right)^{2/3} \varepsilon^{2/3} + O(\varepsilon^{4/3}) \tag{3.4}$$

The importance of these asymptotic expressions is twofold: First, they provide an analytical benchmark to evaluate the performance of numerical methods. Second, they provide reasonable starting points for models with several assets, for which asymptotics are not available.

[1] Different papers use different notation to denote transaction costs. In this thesis, ε denotes transaction cost against the mid-price, *not* the relative bid-ask spread, which is $\frac{\varepsilon}{1+\varepsilon}$.

3.1.1 Optimization Model

As theoretical work prescribes that the optimal trading strategy is identified by the no-trade region $[\pi_-, \pi_+]$, the computational challenge is to identify the boundaries π_- and π_+ numerically. To this end, observe that portfolio updating in discrete-time for a strategy based on a no-trade region is governed by a two-step process. In the first step, the portfolio weight π_{t-1} changes because the asset price changes from $t-1$ to t, becoming π_{t-} defined as

$$\pi_{t-} = \pi_{t-1} \frac{1 + r_t^1}{1 + \pi_{t-1} r_t^1} \tag{3.5}$$

In the second step, the portfolio weight changes from π_{t-}^1 to π_t^1 because the agent trades at time t. The strategy based on the no-trade region $[\pi_-, \pi_+]$ requires that

$$\pi_t = \begin{cases} \pi_- & \pi_{t-} < \pi_- \\ \pi_{t-} & \pi_{t-} \in [\pi_-, \pi_+] \\ \pi_+ & \pi_{t-} > \pi_+ \end{cases} \tag{3.6}$$

or, equivalently,

$$\pi_t = -\max(-\max(\pi_{t-} - \pi_-, 0) + \pi_+ - \pi_-, 0) + \pi_+. \tag{3.7}$$

This latter representation is convenient because it specifies the updating procedure as the composition of affine functions ($x \mapsto ax + b$) and the positive-part transformation ($x \mapsto x^+$), known in the machine-learning literature as the ReLU (rectified linear unit) activation.

These observations indicate that the wealth-portfolio pair (X_t, π_t), which evolves

over time as follows:

$$\pi_{t-} = \pi_{t-1}\frac{1+r_t^1}{1+\pi_{t-1}r_t^1} \qquad \text{(Input layer)} \tag{3.8}$$

$$\pi_t = -\max(-\max(\pi_{t-}-\pi_-,0)+\pi_+-\pi_-,0)+\pi_+ \qquad \text{(Activation)} \tag{3.9}$$

$$\frac{X_t}{X_{t-1}} = 1+r_t^{\pi,\varepsilon} \quad \text{where } r_t^{\pi,\varepsilon} \text{ follows from Proposition 2.1.1} \qquad \text{(Output)} \tag{3.10}$$

(In particular, π_{t-} is used only to obtain π_t but is not required at subsequent steps.)

This recursive updating of wealth and portfolio weights fits into the framework and structure of Recurrent Neural Networks (RNN) (Goodfellow et al. [2016]). The simplest type of such networks (Elman [1990]) is the recursion:

$$h_t = \sigma_h(W_h h_{t-} + b_h) \tag{3.11}$$

$$o_t = \sigma_o(F^{\pi,\varepsilon}(W_o h_t) + b_o) \tag{3.12}$$

where, in machine-learning terminology, h_t denotes the *hidden state*, o_t the *output*, and x_t the *input*. The functions σ_h and σ_o are arbitrary nonlinear maps, while W_h, W_x, W_o, b_h, b_o are model parameters, to be estimated.

The analogy with the present model is as follows: the input is the asset's return r_t^1, which enters the dynamics of the hidden state – the portfolio π_t – along with the previous value of the hidden state. The output is the portfolio return r_t^π.

The portfolio optimization problem is similar in spirit, but does not exactly fit this algebraic specification for two reasons: First, the recursion in (3.8) and (3.9) entails two nonlinear maps (the reciprocal in (3.8) and the ReLU-composition in (3.9)) following linear maps of the arguments. Second, the portfolio return (output) depends both on the portfolio weight (hidden state) and on the asset's return (input). Notwithstanding such differences, the resulting recurrent network is also straightforward to implement through automatic differentiation frameworks.

The RNN in (3.10) contains only two parameters, the boundaries π_- and π_+, and their estimation proceeds as follows.

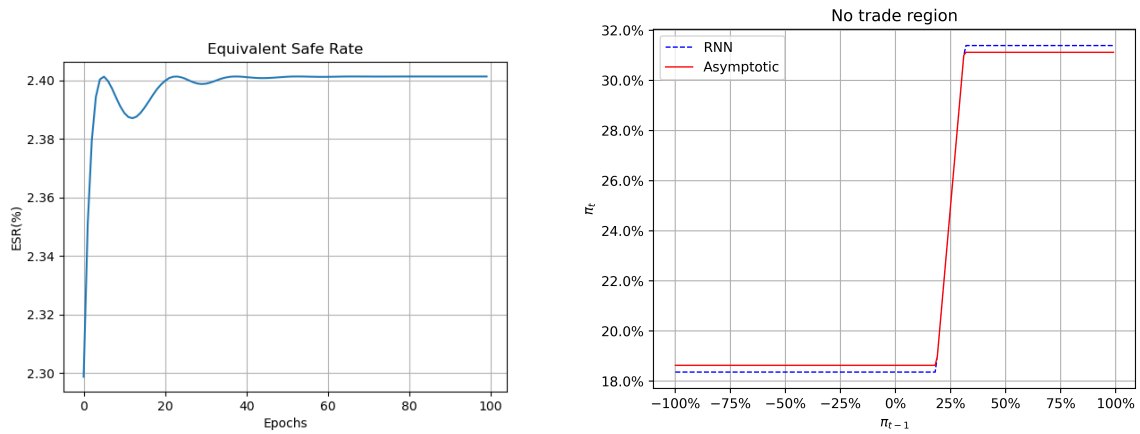


Figure 3.1: Convergence of the equivalent safe rate (left panel) and optimal buy-sell boundaries (right). $\mu = 0.2, \sigma = 0.5, \gamma = 3$, whence $\pi^* = 26.7\%$. T is ten years and Δt half month ($T/\Delta t = 240$). Transaction costs are $\varepsilon = 3\%$. The simulation is performed with $N = 10^4$ paths. The left panel reports $-E\left[\frac{X_T^{1-\gamma}-1}{1-\gamma}\right]$, which yields the loss.

(i) Simulate the IID returns $(r_t^1(\omega_i))_{1\le t\le T}^{1\le i\le N}$, where T denotes the number of time-periods and N the number of paths, with prescribed expected excess return μ and volatility σ.

(ii) Use a steepest-gradient algorithm to maximize the objective (minimize the inverse as the loss):

$$\max_{\pi_-,\pi_+} \frac{1}{N}\sum_{i=1}^{N} \frac{X_T(\omega_i)^{1-\gamma}-1}{1-\gamma} \qquad \text{where } X_T(\omega_i) = \prod_{t=1}^{T}(1 + r_t^{\pi}(\omega_i))$$

3.1.2 Numerical Results

Figure 3.1 reports the calibration results for typical parameter values, using relatively large transaction costs $\varepsilon = 3\%$ to enhance visibility.[2] The results confirm that the numerical optimizers obtained from the algorithm closely align to the theoretical quantities.

Platform	GPU Memory	Time
Google Colab	2.2 GB	<1 min

Table 3.1: One-Asset Model Performance (approximation)

[2] All machine-learning results in this thesis were obtained in the Pytorch 2.2 framework with the CUDA 12.3 libraries.

Table 3.1 presents the efficiency of the one-asset model, which converges fast with little computational resources required. Deviations arises from three sources: (i) statistical variability related to the number of paths in the simulation; (ii) time-discretization, which is half-monthly in the simulation while assumed continuous in the asymptotic formulas, and (iii) approximation error in the asymptotic formula.

In fact, the effect of such deviations is statistically and economically insignificant, as Figure 3.2 shows. The equivalent safe rate of the calibrated policy is only a few basis points lower than the theoretical value implied by the explicit formula, which consistently lies within the estimate's confidence interval. The confidence intervals applied in this thesis are theoretically based on normal distribution with a confidence level equal to 95%.

In summary, numerical optimization results in the one-asset model are consistent, within statistical and economic significance, with the first-order explicit formulas obtained in theoretical papers. The next section examines the two-dimensional setting, highlighting how the problem's complexity increases with dimensionality, and comparing the the performance of the method in this thesis with previous work.

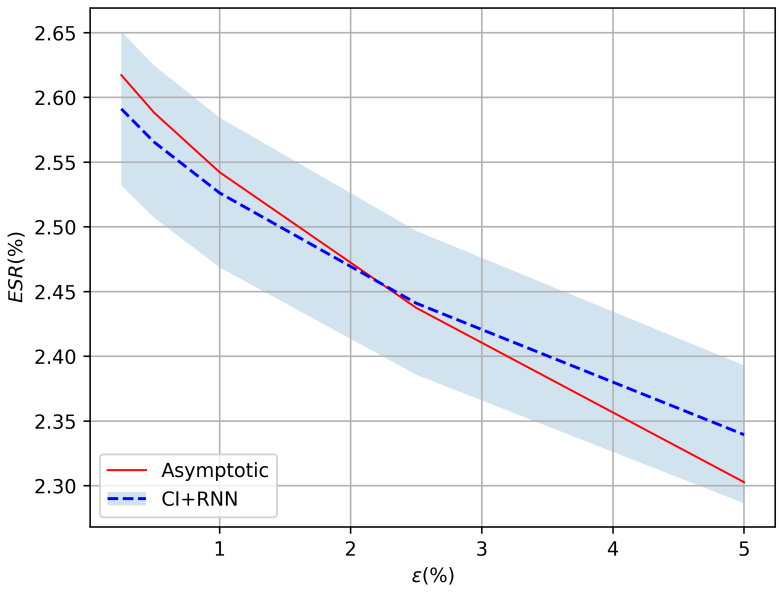


Figure 3.2: Equivalent Safe Rate of the calibrated optimal policy (vertical, in percent, estimate in dashed blue, confidence interval in light blue) and of the explicit formula (3.4) (solid red) against transaction cost (horizontal, in percent). Parameters as in Figure 3.1.

3.2 Two Assets

The two-asset model is important for three reasons: First, as the simplest setting in which no-trade regions are not intervals, it illustrates the challenges of modeling their shapes. Second, in this setting explicit asymptotic formulas are known only in special cases but numerical studies with finite-differences methods are available in general, hence a comparison of our approach with others numerical work is possible. Third, the two-asset model immediately reveals the challenges of higher dimensions and makes it clear which model approaches may scale to higher dimensions and which ones may not.

3.2.1 Optimization Model

The two-asset version of price dynamics is:

$$\frac{dS_t^i}{S_t^i} = (\mu^i + r)dt + \sum_{j=1}^{d} \sigma^{ij} dW_t^j, \quad i = 1, 2$$

where $(\mu^i)^{i=1,2}$ is a vector and $(\sigma^{ij})^{i,j=1,2}$ an invertible matrix. The assets' returns' covariance matrix is $\Sigma = \sigma\sigma'$ (the prime sign denotes matrix transposition). Their correlation is $\rho = \Sigma^{12}/\sqrt{\Sigma^{11}\Sigma^{22}}$. Recall that in this market the optimal portfolio for an investor with relative risk aversion γ is the Markowitz-Merton portfolio $\pi_* = \frac{\Sigma^{-1}\mu}{\gamma}$, for utility maximization from both consumption and terminal wealth, both with a finite and infinite horizons.

Accordingly, equation (3.8) with two assets becomes:

$$\pi_{t-}^i = \pi_{t-1}^i \frac{1 + r_t^i}{1 + \sum_{j=1,2} \pi_{t-1}^j r_t^j} \qquad i = 1, 2 \tag{3.13}$$

while (3.10) remains the same, as Proposition 2.1.1 encompasses an arbitrary number of assets. The central question is how to replace (3.9), i.e., how to describe the no-trade region and rebalancing.

The exact shape of no-trade regions is unknown in multiple dimensions, but the

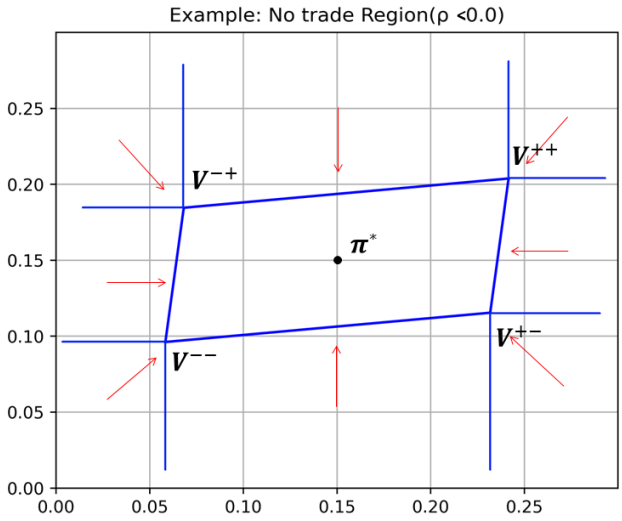


Figure 3.3: No-trade region (centered blue parallelogram) and eight trading regions (other combinations of up-center-down with left-center-right) for uncorrelated (left panel) and correlated (right) assets. Red arrows indicate the boundary components towards which rebalancing occurs.

numerical studies of Muthuraman and Kumar [2004], Possamaï et al. [2015], and Altarovici et al. [2016] indicate that, if transaction costs are small, such regions are well approximated by parallelograms centered near the frictionless optimizer π^*. In addition, the low sensitivity of expected utility to the exact location of the no-trade boundaries near optimality, which is present even with one asset, suggests that focusing on parallelogram-shaped no-trade regions offers a promising class of strategies to approximate optimality. The parametrization of such strategies is visualized in Figure 3.3. Define the matrix

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{3.14}$$

Each of its four vertices is represented as

$$V^{--} = \begin{bmatrix} v_1^{--} \\ v_2^{--} \end{bmatrix} = M \begin{bmatrix} -1 \\ 0 \end{bmatrix} + \pi^* \qquad V^{-+} = \begin{bmatrix} v_1^{-+} \\ v_2^{-+} \end{bmatrix} = M \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \pi^*$$

$$V^{++} = \begin{bmatrix} v_1^{++} \\ v_2^{++} \end{bmatrix} = M \begin{bmatrix} +1 \\ 0 \end{bmatrix} + \pi^* \qquad V^{+-} = \begin{bmatrix} v_1^{+-} \\ v_2^{+-} \end{bmatrix} = M \begin{bmatrix} 0 \\ +1 \end{bmatrix} + \pi^*$$

With these definitions, the two-dimensional analogue of the rebalancing function (3.9) takes two separate expressions, depending on the orientation of the parallelograms, and still relies on the map:

$$S(x, b_-, b_+) := -\max(-\max(x - b_-, 0) + b_+ - b_-, 0) + b_+$$

for suitable choices of b_- and b_+. With matrix (3.14), we formulate two dynamic lines:

$$l_1 = \frac{m_{11} - m_{12}}{m_{21} - m_{22}} \qquad l_2 = \frac{m_{21} + m_{22}}{m_{11} + m_{12}} \tag{3.15}$$

and a double ReLu function in two dimensions with correlation:

$$\pi_t^i = S(\pi_{t-}^i, b_-^i, b_+^i) \qquad i = 1, 2 \tag{3.16}$$

Specifically, if the top-right diagonal is longer than the down-right diagonal ($(l_1 > 0$, $l_2 > 0)$), then

$$b_- = \begin{bmatrix} -(-(l_1 \cdot (\pi_{t-}^2 - v_2^{--}))^+ + v_1^{-+} - v_1^{--})^+ + v_1^{-+} \\ -(-(l_2 \cdot (\pi_{t-}^1 - v_1^{--}))^+ + v_2^{+-} - v_2^{--})^+ + v_2^{+-} \end{bmatrix} \tag{3.17}$$

$$b_+ = \begin{bmatrix} -(-(l_1 \cdot (\pi_{t-}^2 - v_2^{+-}))^+ + v_1^{++} - v_1^{+-})^+ + v_1^{++} \\ -(-(l_2 \cdot (\pi_{t-}^1 - v_1^{-+}))^+ + v_2^{++} - v_2^{-+})^+ + v_2^{++} \end{bmatrix} \tag{3.18}$$

Vice versa, if the top-right diagonal is shorter than the down-right diagonal ($l_1 < 0$, $l_2 < 0$), then

$$b_- = \begin{bmatrix} -(-(l_1 \cdot (\pi_t^2 - v_2^{-+}))^+ + v_1^{--} - v_1^{-+})^+ + v_1^{--} \\ -(-(l_2 \cdot (\pi_t^1 - v_1^{+-}))^+ + v_2^{--} - v_2^{+-})^+ + v_2^{--} \end{bmatrix} \tag{3.19}$$

$$b_+ = \begin{bmatrix} -(-(l_1 \cdot (\pi_t^2 - v_2^{++}))^+ + v_1^{+-} - v_1^{++})^+ + v_1^{+-} \\ -(-(l_2 \cdot (\pi_t^1 - v_1^{++}))^+ + v_2^{-+} - v_2^{++})^+ + v_2^{-+} \end{bmatrix} \tag{3.20}$$

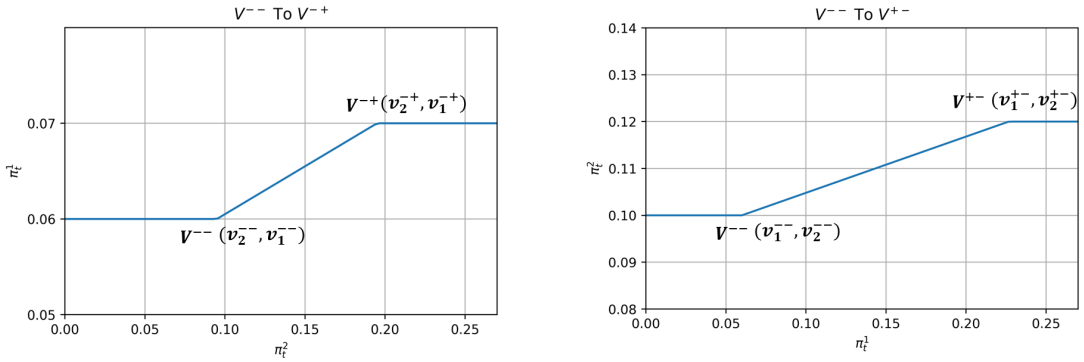


Figure 3.4: Lower bounds of the no-trade region for each asset (b_-^1 on left, b_-^2 on right) against portfolio weight in the other asset. The non-zero slopes in each plot are $l_1 = \frac{m_{11}-m_{12}}{m_{21}-m_{22}} = \frac{v_1^{-+}-v_1^{--}}{v_2^{-+}-v_2^{--}}$ and $l_2 = \frac{m_{21}+m_{22}}{m_{11}+m_{12}} = \frac{v_2^{+-}-v_2^{--}}{v_1^{+-}-v_1^{--}}$ respectively. The left plot is the lower bound of asset 1: $b_-^1(\pi_{t-}^2)$ while the right plot refers to lower bound of asset 2: $\pi_t^2 = b_-^2(\pi_{t-}^1)$.

As a result, a similar structure as the one-asset case follows:

$$\pi_{t-}^i = \pi_{t-1}^i \frac{1 + r_t^i}{1 + \sum_{j=1,2} \pi_{t-1}^j r_t^j} \qquad i = 1, 2 \qquad \text{(Input layer)}$$

$$\pi_t^i = S(\pi_{t-}^i, b_-^i, b_+^i) \qquad i = 1, 2 \qquad \text{(Activation)}$$

$$\frac{X_t}{X_{t-1}} = 1 + r_t^{\pi,\varepsilon} \qquad \text{(Output)}$$

Where the most significant difference is in the activation function, compared with the one-asset case.

3.2.2 Numerical Results

Numerical simulation performance depends on the convergence of the model and the initial guess has significant effect on the convergence. It is suggested that we take into consideration the role that matrix M plays in the model. We notice that M contains the four learnable parameters that defines the four vertexes of the parallelogram.

$$M = \begin{bmatrix} \delta_1 & \delta_1 \\ \delta_2 & -\delta_2 \end{bmatrix}$$

Portfolio	Asset	μ	σ	ρ	ε	γ	π
A	1	12.00%	57%	23%	1%	2	15%
	2	12.00%	57%		1%		15%
B	1	12.00%	67%	-11%	1%	2	15%
	2	12.00%	67%		1%		15%
C	1	3.62%	15%	22%	0%	3	51.73%
	2	0.36%	4%		5%		32.32%

Table 3.2: Parameter values for the three portfolio configurations.

is a chosen as initial value to start with where δ_1 and δ_2 are the distance of boundaries to the Merton portfolio respectively, without considering the correlation between assets. With such M, we starts to estimate the no trade region from a simple no trade region.

Table 3.2 reports three combinations of parameters considered in the literature. The first two combinations were used by Muthuraman and Kumar [2004] in a numerical solution of the HJB partial differential equation with the finite-element method. The last combination was used in Bichuch and Guasoni [2018] to solve the HJB ordinary differential equation arising in a market with one liquid and one illiquid asset.

The location and size of the no-trade regions obtained with the method in this thesis are essentially the same as in Muthuraman and Kumar [2004] (top and center panels in Figure 3.5), but the shapes differ, as the regions in Muthuraman and Kumar [2004] depart from parallelograms, especially in the left and bottom sides. This discrepancy is largely due to the difference in the objective considered: Muthuraman and Kumar [2004] study the HJB equation of the problem of optimal investment and *consumption*, rather than optimal investment alone. Consumption entails a continuous outflow of cash, therefore it induces agents to accept larger cash balances before buying more stocks, lest they may have to sell them again soon. This effect is more pronounced if the agent's time-preference rate (hence consumption rate) is higher, and indeed Muthuraman and Kumar [2004] use a 10% discount rate, leading to an annual consumption rate above 9%.

As long as asymptotics are well approximated in the liquid-illiquid model, it is also possible to evaluate whether the results are consistent with previous work. In pursuit of a comprehensive analysis, we examine a case involving assets with distinct trading

costs. Portfolio C (Table 3.2) encompasses one liquid asset without trading costs and one illiquid asset, with data derived from estimated values of U.S. Equities and Real Estate (1975âĂŞ2013) (Bichuch and Guasoni [2018]). The results in the liquid-illiquid asset model (bottom panel in Figure 3.5) are closely aligned with those in of Bichuch and Guasoni [2018]. Because one asset is nearly frictionless (0.01%), the no-trade region is a thin oblique line, whose slope reflects the use of the liquid asset to hedge the fluctuating illiquid position. In summary, the no-trade regions identifies with the RNN-based approach in this thesis are broadly consistent with previous work, with some discrepancies related to differences in the models considered in different studies.

Platform	GPU Memory	Time
Google Colab	8.6 GB	12 min

Table 3.3: Two-Assets Model Performance (approximation)

Table 3.3 presents the efficiency of the two-assets model, where computing time and GPU memory requirement increases significantly. The main reason could be the increase number of logical reasoning with respect to both assets when the model calculates the slopes of the edges of the parallelogram.

3.2.3 Equivalent Safe Rates

Recall that we can estimate the ESR of a portfolio using Equation (3.3). Nevertheless, there is insufficient evidence of an explicit theoretical solution for a two-asset portfolio with trading costs. Consequently, an alternative comparison is introduced for the case of Portfolio C, which combines a liquid asset and an illiquid asset, as there exists an asymptotic solution (Bichuch and Guasoni [2018]):

$$ESR = r + \frac{\mu^2\sigma_1^2 - 2\mu_1\mu_2\rho\sigma_1\sigma_2 + \mu_1^2\sigma_2^2}{2\gamma(1-\rho^2)\sigma_1^2\sigma_2^2} - \frac{\lambda^2}{2\gamma\sigma_2^2(1-\rho^2)} \tag{3.21}$$

where λ is derived from the following equations:

$$\lambda = \left(\frac{3}{4}\gamma(\pi^*)^2\sigma_I^4\left((\beta_2 - \frac{\mu_1}{\gamma\sigma_I})^2\sigma_1^2 + (1-\pi_2^*)^2\sigma_I^2\right)\right)^{1/3}\varepsilon^{1/3} + O(\varepsilon^{2/3})$$

$$\beta_1 = \frac{\rho\sigma_1\sigma_2}{\sigma_2^2}, \beta_2 = \frac{\rho\sigma_1\sigma_2}{\sigma_1^2}$$

$$\mu_2 = \beta_2\mu_2 + \alpha, \quad \sigma_2^2 = (\beta_2\sigma_1)^2 + \sigma_I^2$$

By substituting parameters in Portfolio C to Equation(3.21), we can have both simulated and theoretical ESR:

Asset	ESR RNN(%)	95%Confidence Interval	Asymptotic ESR(%)
Portfolio C	0.992	[0.959%,1.022%]	0.984

Table 3.4: ESR on Portfolio C in Table(3.2)

Consequently, it is clear that the RNN model delivers a robust approximation of the no-trade region and the corresponding equivalent safe rate in the long term when the portfolio comprises a combination of liquid and illiquid assets. Additionally, we investigate a range of distinct trading costs associated with the illiquid asset:

ε_1	ε_2	ε_3	ε_4	ε_5
0.25%	0.5%	1%	2.5%	5%

and we thereby plot the simulated ESR and theoretical ESR with respect to such trading costs in Figure 3.6. There are some reasons resulting in the slow change of ESR with respect to trading costs. For example, low volatility of illiquid asset results in low possibility of trading the illiquid asset. The asymptotic result also indicates that ESR is not strongly impacted by trading cost of the illiquid asset.

Still, there is compelling evidence supporting the strong performance of the RNN model, as the generated confidence interval encompasses the theoretical solution across various transaction costs.

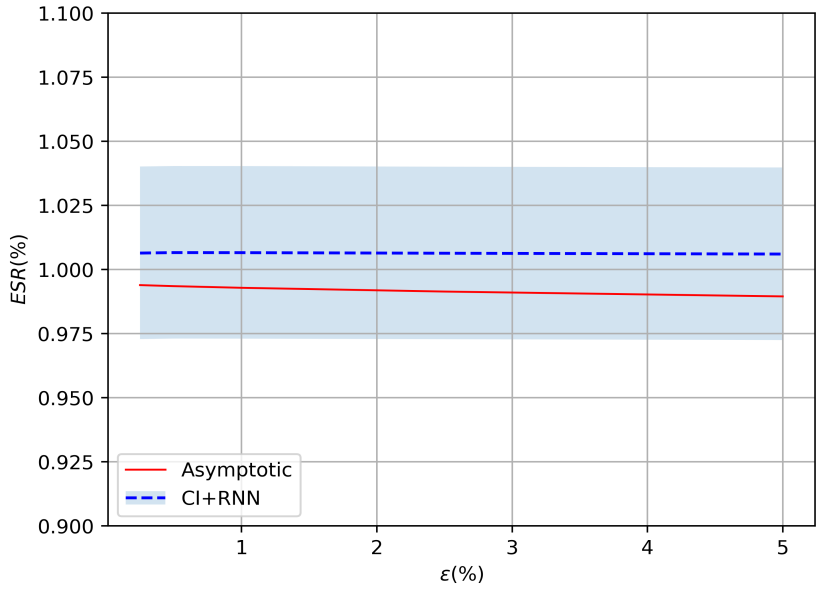
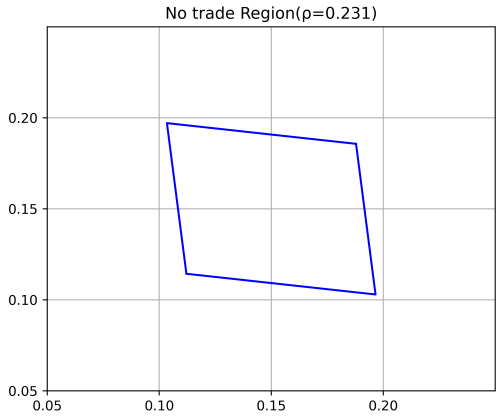
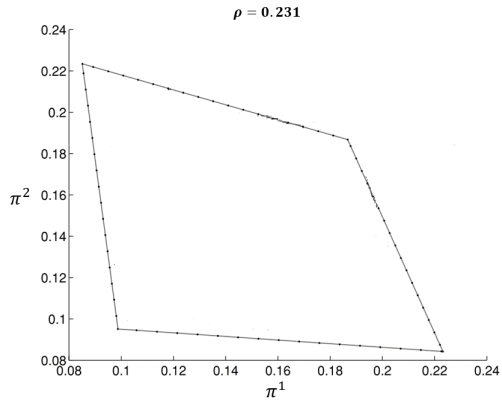
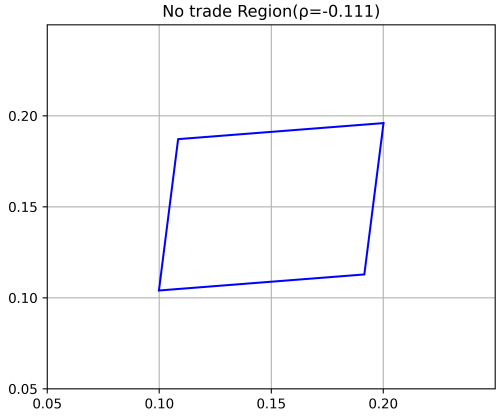
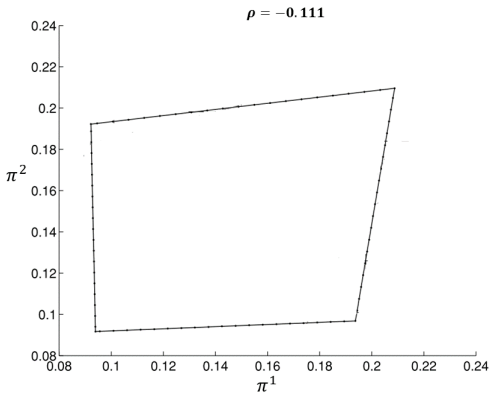


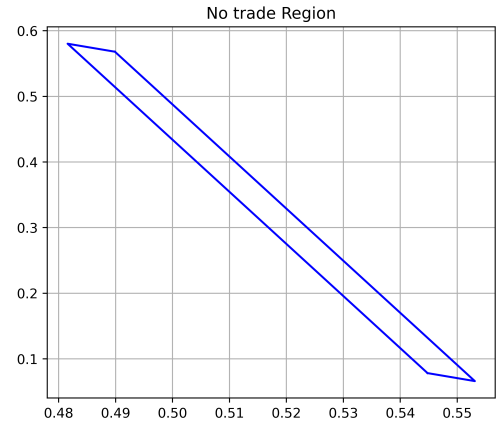
Figure 3.6: Portfolio C: ESR Comparison











Asset	Low	Frictionless	High
π_1	50.79%	51.73%	52.68%
π_2	16.18%	32.32%	48.46%

Figure 3.5: Comparison of the no-trade regions in this thesis (left) with the ones in Muthuraman and Kumar [2004] (top and center) and the numerical bounds in Bichuch and Guasoni [2018] (bottom).

Chapter 4

Multiple Dimensional Models

This chapter presents a path from a two assets model to the formulation of multiple assets model. With the two-assets model, we manually locate the coordinates of each vertex of the no-trade region which formulates a parallelogram. However, it is impossible to replicate the same method in high-dimensional portfolio as the number of vertices increase to 2^d, with more edges and hypersurfaces. We introduce a vectorization and bisection model (also called hyperparallelogram model) to progress. The the model is integrated with several method for simplicity. A new hyperellipsoid model is then constructed to improve computation efficiency and is tested with benchmark in comparison with basic multi-dimensional model. Both methods are embedded into the RNN model by replacing the activation function in the hidden layer and keep the general model structure unchanged. In addition, this chapter pertinent examples featuring more than two risky assets: simulated portfolios with 2, 3, and 10 risky assets.

4.1 Model formulation

Considering a market with $d(d > 2, d \in \mathbf{N})$ risky assets and $r = 0$ as well:

$$\frac{dS_t^i}{S_t^i} = \mu^i dt + \sum_{j=1}^{d} \sigma^{ij} dW_t^j, \quad i = 1, 2, \ldots, d$$

with ρ^{ij} representing the correlation between asset i and j. In contrast to the scenario in section(3.2), the number of assets increases, subsequently elevating the complexity of devising a model with explicit boundaries for each risk asset with trading costs. This is due to the necessity of extracting each individual component from the rotation matrix of size $d \times d$, in order to manually formulate at least $2d$ boundaries. Consequently, it becomes crucial to develop an alternative model that focuses on the multi-asset case by reducing complexity, even at the expense of accuracy. As a result, a more generalized vectorization model will be introduced.

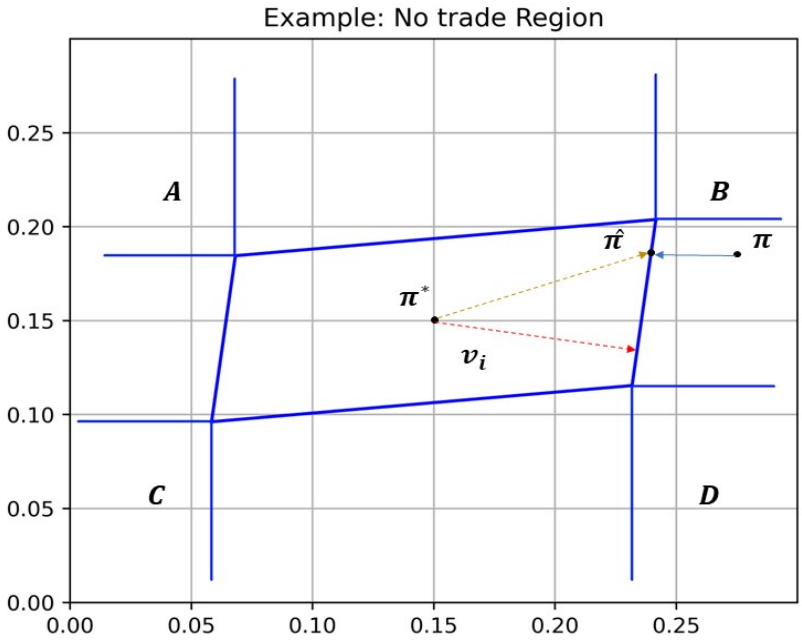


Figure 4.1: The Vectorization Model

Figure 4.1 presents a two-asset example within a suitable vectorization model. Analogous to the content in Figure 3.3, π^* represents the Merton portfolio, while π denotes a strategy outside the specific no-trade region following a price change in assets. $\hat{\pi}$ is the anticipated portfolio after rebalancing, which is precisely what we aim to determine. The transition from π to $\hat{\pi}$ is referred to as one "projection". The vector v_i serves as normalized direction of asset i $(i = 1, 2, \ldots, d)$ and is set to be always orthogonal to the boundary of asset i. The aforementioned variables have the

following relationship:

$$\alpha_i^+ = v_i \cdot (\hat{\pi} - \pi^*) \tag{4.1}$$

$$\hat{\pi} = \pi + \lambda_i \cdot e_i \tag{4.2}$$

Where equation (4.1)) is recognized as the boundary condition, and α_i^+ is the upper boundary of asset i, which is forced to be positive. In the same way, we can evidently presume the existence of a symmetric virtual lower boundary for asset i, denoted by α_i^-. Equation (4.2) illustrates the blue arrow, "projection," in Figure 4.1, where λ_i represents the distance between π and $\hat{\pi}$, and e_i is the standard unit vector of a specific axis. Once again, it is clear that we have:

$$\lambda_i = \frac{\alpha_i^+ - v_i \cdot (\pi - \pi^*)}{v_i \cdot e_i} \tag{4.3}$$

when

$$v_i \cdot (\pi - \pi^*) > \alpha_i^+$$

we will alternatively have:

$$\lambda_i = \frac{\alpha_i^- - v_i \cdot (\pi - \pi^*)}{v_i \cdot e_i} \tag{4.4}$$

when

$$v_i \cdot (\pi - \pi^*) < \alpha_i^-$$

As a result, the "projection" method is to ensure that after one "projection", we can guarantee:

$$\alpha_i^- \leq v_i \cdot (\hat{\pi} - \pi^*) \leq \alpha_i^+$$

which restores the portfolio to the no-trade region except for those in regions A, B, C, and D. However, such a “projection" is not applicable in regions A, B, C, and D since both α_i^+ and α_i^- would force π to be projected onto some extension line of boundaries that remain outside the no-trade region. Consequently, an alternative bisection method is introduced to address this situation:

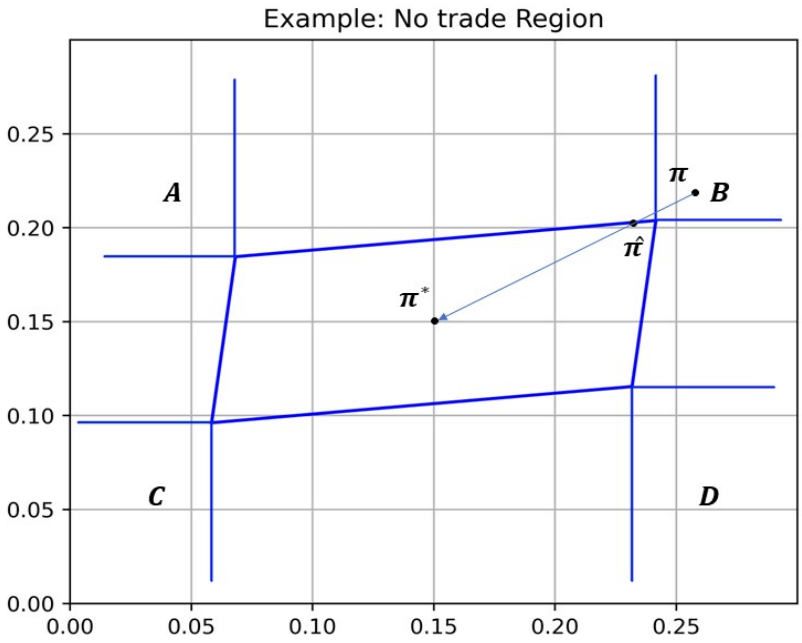


Figure 4.2: The Vertex Model

Figure 4.2 indicates a example of a bisection method. Once π falls into region B, instead of performing a "projection", we formulate the link π between π^* to find the middle point:

$$\pi_m = \frac{\pi + \pi^*}{2}$$

Repeating this process by replacing either π or π^* with π_m during each loop until $v_i \cdot (\hat{\pi} - \pi^*) \in (\alpha_i^+ - \epsilon, \alpha_i^+ + \epsilon)$ or $v_i \cdot (\hat{\pi} - \pi^*) \in (\alpha_i^- - \epsilon, \alpha_i^- + \epsilon)$ or the maximum loop is reached. Ultimately, $\hat{\pi}$ (the last generated π_m) becomes the anticipated strategy. This algorithm of course introduces some inaccuracy, as illustrated in Figure 4.2. The $\hat{\pi}$ will mostly be situated at the boundary close to the vertex rather than precisely at the

vertex, as depicted in Figure 3.3. Moreover, with the increase in dimension d (number of assets), the algorithm becomes less accurate since there tends to be edges, surfaces or other hyper-surfaces serves as the vertex in the two assets case. Nonetheless, as price changes are less likely to be substantial within small time intervals, the method remains applicable, provided that rapid rebalancing is employed.

In summary, the combination of the "projection" method and bisection method contributes to the modeling of the no-trade region. It is necessary to first determine if the strategy is inside the vertex region (i.e., A, B, C, D) using the discriminant that strategies outside the vertex region will be within the no-trade region in one "projection", while those inside the vertex region will not. The full algorithm is outlined as follows:

Algorithm 1 Projection and Bisection (Projection)

$\epsilon = 1e-5$
$n = 8$
for $i \leftarrow 1, 2, \ldots, d$ **do** ▷ Loop over assets
    $v_i = \textbf{Normalize}(v)[i, :]$
    $\lambda^+ = \alpha_i^+ - v_i(\pi - \pi^*)/e_i v_i$
    $\lambda^- = \alpha_i^- - v_i(\pi - \pi^*)/e_i v_i$
    $\hat{\pi} = \pi + \lambda^+ \times (v_i(\pi - \pi^*) > \alpha_i^+) \times e_i + \lambda^- \times (v_i(\pi - \pi^*) < \alpha_i^-) \times e_i$
    $judge = v(\hat{\pi} - \pi^*) < \alpha^+ + \epsilon$ **and** $v(\hat{\pi} - \pi^*) > \alpha^- - \epsilon$ ▷ Judge variable indicating the status of all assets after one projection trade of the i-th asset
    $judge\_vec[i] = \min(judge, dim = 0)$ ▷ Judge vector recording if an asset is suitable for projection
for $i \leftarrow 1, 2, \ldots, d$ **do**
    $v_i = \textbf{Normalize}(v)[i, :]$
    $\lambda^+ = \alpha_i^+ - v_i(\pi - \pi^*)/e_i v_i$
    $\lambda^- = \alpha_i^- - v_i(\pi - \pi^*)/e_i v_i$
    $\hat{\pi} = \pi + \lambda^+ \times (v_i(\hat{\pi} - \pi^*) > \alpha_i^+) \times e_i + \lambda^- \times (v_i(\hat{\pi} - \pi^*) < \alpha_i^-) \times e_i$

Algorithm 2 Projection and Bisection (Bisection)

$\pi_{in} = \pi^*$
$\pi_{out} = \pi_m = \pi$
$judge\_bi = v(\pi_m - \pi^*) \in (\alpha^+ - \epsilon, \alpha^+ + \epsilon)$ **or** $v(\pi_m - \pi^*) \in (\alpha^- - \epsilon, \alpha^- + \epsilon)$▷ This variable is used to judge whether the π_m is exactly on the boundary of no trade region
while not $judge\_bi$ **and** $k < n$ **do**
    $k++$
    $\pi_m = (\pi_{in} + \pi_{out})/2$
    if $(v(\pi_m - \pi^*) < \alpha^+ + \epsilon$ **and** $v(\pi_m - \pi^*) > \alpha^- - \epsilon)$ **then**
        $\pi_{in} = \pi_m$
    else
        $\pi_{out} = \pi_m$
    $judge\_bi = v(\pi_m - \pi^*) \in (\alpha^+ - \epsilon, \alpha^+ + \epsilon)$ **or** $v(\pi_m - \pi^*) \in (\alpha^- - \epsilon, \alpha^- + \epsilon)$
return $judge\_vec \times \hat{\pi} + (1 - judge\_vec) \times \pi_m$ ▷ Apply projection algorithm to the assets that suits projection algorithm and bisection to others

The whole function of hyperparallelogram method can be summarized as:

$$\hat{\pi}_t = HP(\pi_{t-}). \tag{4.5}$$

4.2 Model Simplification

Analogously, for the projection and bisection model delineated in Section 4.1, it is imperative to estimate a matrix v and two vectors, α^+ and α^-. To mitigate computational complexity, we posit $\alpha^- = -\alpha^+$, thereby reducing the estimation to a single vector. An alternative approach to further diminish model complexity involves initially establishing an input value as follows:

$$v = \begin{bmatrix} v_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & v_d \end{bmatrix} = \begin{bmatrix} \frac{1}{\delta_1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\delta_d} \end{bmatrix}$$

Subsequently, one could refrain from normalizing vector v and permanently set $-\alpha^- = \alpha^+ = 1$. This methodology effectively eliminates at least n variables, allowing vector v to encapsulate more information. However, it is crucial to note that since δ varies

among different assets and vector v is no longer normalized, distinct portfolios may yield substantially different v values. This discrepancy can precipitate convergence difficulties when selecting an appropriate learning rate.

An alternative strategy involves maintaining $-\alpha_i^- = \alpha_i^+ = \delta_i$, which consequently leads to initial inputs tending towards:

$$v = \begin{bmatrix} v_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & v_d \end{bmatrix} = \begin{bmatrix} 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{bmatrix}$$

This approach effectively mitigates the impact of unbounded initial estimates. For both aforementioned methodologies, it is crucial to judiciously select the learning rate to ensure the convergence of the simulation, as the optimal learning rate may vary significantly across different portfolios. The techniques elucidated above successfully reduce the number of parameters requiring estimation from d^2 to $d^2 - d$, thereby enhancing computational efficiency to a small extent.

To expound further, the reduction in parameter space not only alleviates computational burden but also potentially mitigates over-fitting risks. This parsimony in model complexity is particularly advantageous when dealing with high-dimensional asset spaces, where the curse of dimension can otherwise pose significant challenges.

It is worth noting that while these simplifications offer computational advantages, they may introduce certain trade-offs in model flexibility. Therefore, it is imperative to validate the model's performance empirically and ensure that the reduced parameter space does not unduly constrain the model's ability to capture essential market dynamics.

Furthermore, the selection of an appropriate learning rate becomes even more critical in this context. One might consider implementing adaptive learning rate techniques, such as Adam, to dynamically adjust the learning rate during the optimization process. This adaptive approach could potentially address the portfolio-specific variations in optimal learning rates, thereby enhancing the robustness and convergence

properties of the model across diverse portfolio compositions.

4.3 Ellipsoid Method

4.3.1 The Hyperellipsoid model

With portfolios comprising more than three assets exhibiting low correlation, the initial projection method yields results wherein the learned no-trade region approximates a general hyperellipsoid. We can mathematically represent this general ellipse using the following equation (Strang [2006]):

$$(\pi - \pi^*)^T A(\pi - \pi^*) = 1 \tag{4.6}$$

where A denotes a positive definite symmetric matrix (Golub and Van Loan [1996]) that defines the hyperellipsoid. This formulation allows for a more nuanced representation of the no-trade region in higher-dimensional asset spaces. The decision-making process for a particular input strategy π_{t-} can be articulated as follows:

- if a particular input strategy π_{t-} lies inside the expected no-trade region, we maintain the current position: $\hat{\pi}_t$;

- Conversely, when π_{t-} is situated outside the no-trade region:

$$(\pi_{t-} - \pi^*)^T A(\pi_{t-} - \pi^*) > 1$$

We can introduce a scalar τ to indicate the direction pointing from π^* to π_{t-}, thus establishing the following relationship:

$$\tau^2(\pi_{t-} - \pi^*)^T A(\pi_{t-} - \pi^*) = 1$$

This formulation allows us to solve for τ, yielding:

$$\tau^2 = \frac{1}{(\pi_{t-} - \pi^*)^T A(\pi_{t-} - \pi^*)} \tag{4.7}$$

The projected point onto the surface of the hyperellipsoid, representing the modified strategy, can be elegantly expressed as:

$$\hat{\pi}_t = HE(\pi_{t-}) = \pi^* + \tau(\pi_{t-} - \pi^*) \tag{4.8}$$

This method indeed offers a significant reduction in the number of parameters to be estimated, from d^2 to $\frac{d^2+d}{2}$ and ensures that when rebalancing is necessary (i.e., when π_{t-} lies outside the no-trade region), the new portfolio allocation $\hat{\pi}_t$ is always positioned on the boundary of the no-trade region. This approach minimizes transaction costs by executing the smallest possible trade to bring the portfolio back to an acceptable position.

It is imperative to acknowledge that the hyperellipsoid assumption stems from a phenomenon observed in high-dimensional bisection methods. Specifically, these methods tend to obfuscate the boundaries of hyperparallelograms, causing them to approximate hyperellipsoids. This characteristic represents a notable limitation of the projection and bisection approach. Consequently, the hyperellipsoid approximation scarcely enhances model accuracy and finds its applicability primarily in high-dimensional portfolios exhibiting low correlation among their components. The transformation from parallelograms to hyperellipsoids in higher dimensions is a consequence of the "curse of dimensionality", wherein the geometric properties of objects become less significant as the number of dimensions increases. This phenomenon underscores the complexity inherent in high-dimensional optimization problems and reinforces the need for careful consideration when applying such approximations in portfolio theory.

The validation of the hyperellipsoid method necessitates a comparative analysis of the convergence properties of both the relative ESR and the geometric characteristics of the hyperellipsoid, specifically its boundaries and axes lengths. Although subject to

rotation, the hyperellipsoid fundamentally remains an n-dimensional ellipse. Its initial axes lengths can be expressed as functions of the eigenvalues of the positive definite matrix A:

$$a_1, a_2, ..., a_d = \frac{1}{\sqrt{\xi_1}}, \frac{1}{\sqrt{\xi_2}}, ..., \frac{1}{\sqrt{\xi_d}}$$

where $a_i (i = 1, 2...d)$ are the axes length of the hyperellipsoid and $\xi_i (i = 1, 2...d)$ are eigenvalues of matrix A. The boundaries of the parallelogram, however, are not easy to obtain through the algorithm. As a result, we apply:

$$|\alpha_i| \quad i = 1, 2...d$$

to approximate the boundaries of such a parallelogram. It is clear that the hyperparallelogram and hyperellipsoid methods share similar structure:

$$\begin{aligned}
\pi^i_{t-} =& \pi^i_{t-1} \frac{1 + r^i_t}{1 + \sum_{j=1,2} \pi^j_{t-1} r^j_t} \qquad i = 1, 2, ...d && \text{(Input layer)} \\
\pi_t =& HP(\pi_{t-}) \quad \text{or} \quad HE(\pi^i_{t-}) && \text{(Activation)} \\
\frac{X_t}{X_{t-1}} =& 1 + r_t^{\pi,\varepsilon} && \text{(Output)}
\end{aligned}$$

It is crucial to recognize that this estimation, while informative, is not precise due to several geometric considerations. The vectors α_i do not necessarily align with the diagonals of the hyperparallelogram, nor are they guaranteed to be mutually orthogonal. This misalignment introduces a degree of imprecision into our approximation. The utility of this approach lies in its ability to provide a reasonable approximation. Our expectation is that the axes lengths of the hyperellipsoid and their corresponding α_i values exhibit a close correspondence, both in magnitude and relative scale.

4.3.2 Example: Ten risky assets

To validate the performance of this complexity reduction method, we introduce a comprehensive benchmark for comparative analysis. This benchmark facilitates a systematic comparison between the proposed hyperellipsoid method and the origi-

nal methodology elucidated in Section 4.6. For the purpose of this evaluation, we employ a diverse ten-asset portfolio, the composition of which is meticulously detailed in Table 4.6.

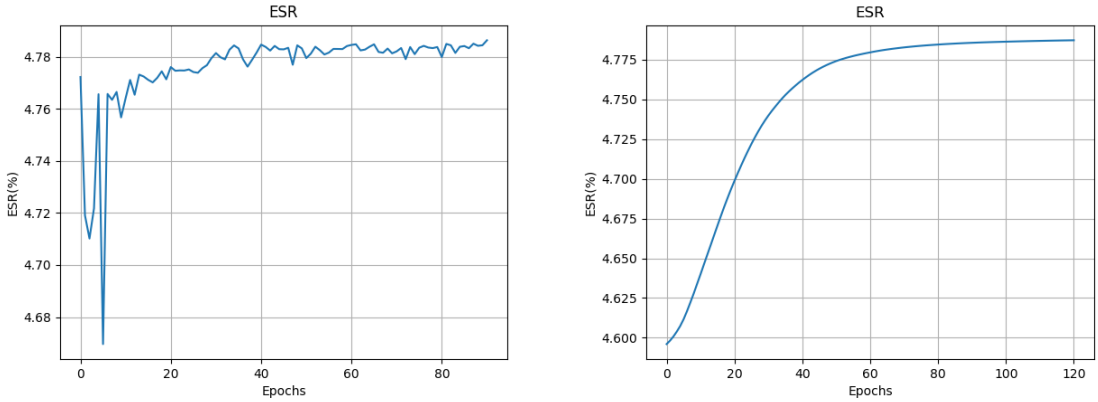


Figure 4.3: Portfolio F Comparison

The left plot is from the original projection and bisection model in section 4.1 and the right plot is from the hyperellipsoid method introduced in this section with $lr = 0.1$.

	$ESR_{min}(\%)$	ESR RNN(%)	$ESR_{max}(\%)$
Portfolio F	4.72	4.79	4.88
Portfolio F (Hyperellipsoid)	4.72	4.78	4.88

Table 4.1: Portfolio ESR in Table 4.6

Based on the results presented in Figure 4.3 and Table 4.1, it can be conclusively inferred that the hyperellipsoid method performs admirably, aligning with theoretical expectations.

Method	Platform	GPU Memory	Time
Hyperparallelogram	Google Colab	22 GB	25 min
Hyperellipsoid	Google Colab	9.3 GB	2 min

Table 4.2: Ten-Assets Model Performance Comparison (approximation)

The method's efficacy (Table 4.2) is evidenced by several key observations:

(i) **Convergence:** The hyperellipsoid method demonstrates a markedly smoother convergence trajectory compared to the original method, even when initialized with random input values. This enhanced stability in convergence is a significant advantage in optimization processes.

(ii) **Computational Efficiency:** The method exhibits substantial improvements in computational performance. Specifically, it achieves a reduction of at least 95% in simulation time and 50% in memory utilization. This remarkable efficiency gain allows for the incorporation of additional epochs and scenarios, potentially leading to increased overall accuracy.

(iii) **Accuracy and Efficiency:** As evidenced in Table 4.1, the hyperellipsoid method introduces a marginal reduction in ESR of approximately 0.01%. This slight decrease in performance is offset by the significant enhancements in computational efficiency and resource utilization.

The implementation of this method presents users with a nuanced trade-off between accuracy, computational speed, and memory management. While the slight reduction in ESR may be a consideration, the substantial gains in efficiency often brings this minor loss in precision for many practical applications. As a result, we highlight that this method should only be applied to portfolios with multiple assets, as the higher the dimension is, the more the no-trade region of original projection and bisection method tends to be close to a hyperellipsoid. Hence, the difference between two methods will also reduce.

Assets	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
$\frac{1}{\sqrt{\xi_i}}$	0.039	0.046	0.049	0.059	0.077	0.116	0.163	0.310	8.330	0.469
$\lvert\alpha_i\rvert$	0.022	0.109	0.022	0.028	0.021	0.018	0.048	0.030	0.017	0.029
Euclidean Distance	8.332									

Table 4.3: Comparison of Axes Lengths and boundaries on Portfolio F

Table 4.3 provides a comprehensive comparative analysis of the axes lengths derived from the hyperellipsoid model and the corresponding $|\alpha_i|$ values obtained from the hyperparallelogram model. A detailed examination of these results reveals visible disparities between the two approaches, and Key observations from this comparison include:

(i) The hyperparallelogram model exhibits a notable property of providing stationary boundaries for all assets within the portfolio. This uniformity in boundary

definition suggests a consistent treatment of all assets, regardless of their individual characteristics or correlations.

(ii) In contrast, the hyperellipsoid model generates a set of axes lengths characterized by extreme variability, causing a huge Euclidean distance between two methods.

One reason could be that gradients vanish at some directions of the hyperellipsoid. This issue may arises due to the inherent curvature of the hyperellipsoid's surface and has significant implications for the optimization process. This is a drawback of the hyperellipsoid model since several assets strategy may be faced with outage during the training and trading. With the dimension increasing, this adverse impact may also increase while the importance of a particular asset may decrease, compensating for such impact at certain level. Therefore, this method can remain cautiously optimistic for the speed of convergence, but extra work should be conducted to correct and improve such model.

4.4 Example: Two risky assets

Having developed the projection and bisection model in section 4.1, it is worthwhile to compare the simulation results of the new model with relevant existing results. We begin with the two-asset cases, i.e., portfolio A and portfolio C. The comparisons are presented as follows:

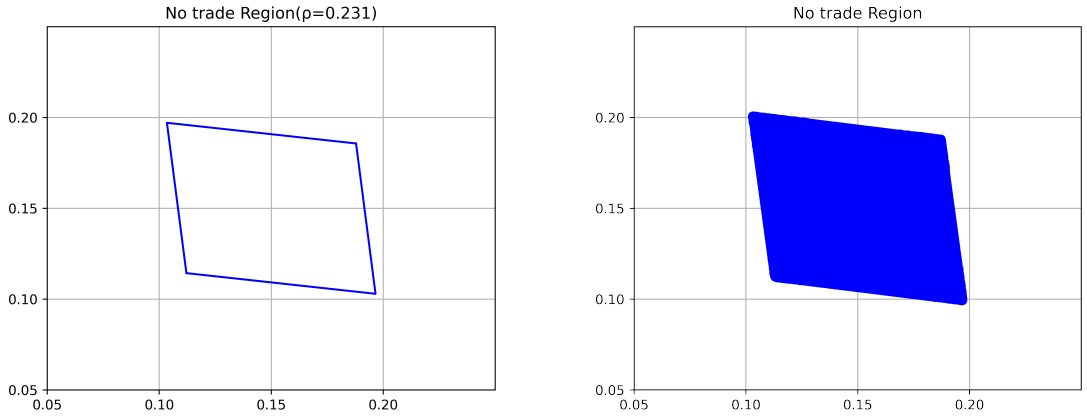


Figure 4.4: Portfolio A Comparison

The left plot is from the parallelogram model in section 3.2 and the right plot is from the projection and bisection model in section 4.1.

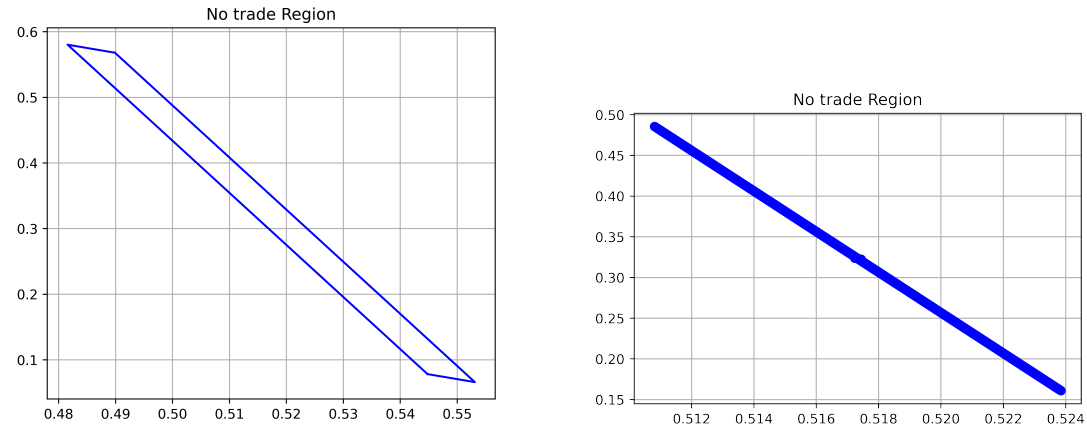


Figure 4.5: Portfolio C Comparison

The left plot is from the RNN model in section 3.2 and the right plot is simulated by projection and bisection model in section 4.1.

The plots of the projection and bisection model are generated using the Monte Carlo method, as the estimated parameters are not expected to calibrate exact boundaries. The comparison of portfolio A demonstrates substantial consistency between the two models, with similar plots. The comparison of portfolio C reveals a better approximation by the projection and bisection model, as the distance between the upper and lower bounds of asset 1 should be very close, due to the fact that it has full liquidation. In summary, we can confidently conclude that the new projection and bisection model

provides a good fit for the two-asset system according to previous work.

4.5 Example: Three risky assets

To study the performance of projection and bisection model with multiple assets, we construct two new portfolio with 3 assets and $\gamma = 2$, $\varepsilon = 1\%$.

	μ	σ	ε
Asset1(π_1)	0.12	0.4	1%
Asset2(π_2,)	0.12	0.4	1%
Asset3(π_3,)	0.12	0.4	1%

Table 4.4: Portfolios D & E

Portfolios D & E have similar assets with $\rho_D = 0$ and $\rho_E = 20\%$.

The corresponding 3D plots are presented below. In accordance with Portfolio D, we expect to see a 3D square due to the zero correlation between assets. The plot strongly supports this theory, as it closely resembles a square with only minor deviations due to random error. With respect to Portfolio E, as small trading costs are introduced, we anticipate a parallelepiped, which is indeed what we observe in Figure 4.6.

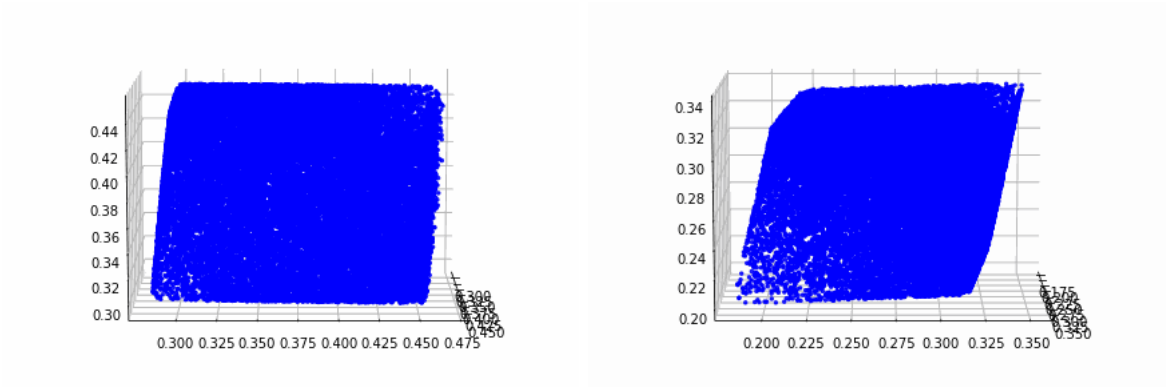


Figure 4.6: No-trade region of Portfolio D & Portfolio E

The left plot is from Portfolio D and the right plot is from Portfolio E

Although the plots in Figure 4.6 visually corroborate the efficacy of the projection and bisection model, it remains crucial to quantitatively assess the results. Despite the absence of an explicit formula for determining either the no-trade region or the

equivalent safe rate of high-dimensional assets, we can approximate the boundaries of the ESR utilizing certain formulas derived from the work of Bichuch and Guasoni [2018]. In a market consisting of $n+1$ assets, where n assets are liquid with μ_1 and Σ_1, and only one asset is illiquid with μ_2 and σ_2^2, the ESR is approximated as the safe rate in the absence of trading costs, diminished by a term contingent on the trading costs and the correlation between the illiquid asset and the liquid assets:

$$ESR := \frac{\mu' \Sigma^{-1} \mu}{2\gamma} - \frac{\lambda^2}{2\gamma\sigma_2^2 (1-\Phi)} \tag{4.9}$$

where

$$\Sigma = \begin{pmatrix} \Sigma_1 & \Sigma_{12} \\ \Sigma_{21} & \sigma_2^2 \end{pmatrix}$$

$$\lambda = \left(\frac{3}{4}\gamma^2 (\pi_2^*)^2 \sigma_I^4 \left(\left(\Sigma_{12} - \frac{\mu_1}{\gamma} \right)' \Sigma_1^{-1} \left(\Sigma_{12} - \frac{\mu_1}{\gamma} \right) + (1-\pi_2^*)^2 \sigma_I^2 \right) \right)^{1/3} \varepsilon'^{1/3} + O\left(\varepsilon'^{2/3}\right)$$

$$\Phi = \frac{\Sigma_{21} \Sigma_1^{-1} \Sigma_{12}}{\sigma_2^2}$$

and Σ represents the comprehensive covariance matrix by part, while Σ_{12} is the covariance between all liquid assets and the particular illiquid asset. The n-dimensional vector $\beta = \Sigma_1^{-1}\Sigma_{12}$ signifies the exposures of the illiquid asset to the risk factors of all liquid assets, and $\sigma_I^2 = \sigma_2^2(1-\Phi)$ designates the corresponding asset-specific risk. Moreover, the frictionless optimal illiquid asset is:

$$\bar{\pi}_2 = \frac{\mu_2 - \beta' \mu_1}{\gamma \sigma_2^2 (1-\Phi)}$$

Utilizing the aforementioned formulas, it becomes feasible to compute the ESR of a multi-asset portfolio. Nevertheless, as our multi-asset portfolio comprises $d (d > 2)$ assets, rather than containing merely one illiquid asset, Equation (4.9) ceases to be effective. One approach to approximate the ESR involves treating each asset as the

illiquid one while considering the others as liquid, and subsequently subtracting the cumulative impact of trading costs from the ESR devoid of trading costs. Although this estimation tends to underestimate the ESR, it adapts Equation (4.9) as follows:

$$ESR_{min} := \frac{\mu' \Sigma^{-1} \mu}{2\gamma} - \sum_{i=1}^{d} \frac{\lambda_i^2}{2\gamma \sigma_i^2 (1 - \Phi_i)} \tag{4.10}$$

$$\lambda_i = \left(\frac{3}{4} \gamma^2 (\pi_i^*)^2 \sigma_{I,i}^4 \left(\left(\Sigma_{1i} - \frac{\mu_1}{\gamma} \right)' \Sigma_1^{-1} \left(\Sigma_{1i} - \frac{\mu_1}{\gamma} \right) + (1 - \pi_i^*)^2 \sigma_{I,i}^2 \right) \right)^{1/3} \varepsilon'^{1/3} + O\left(\varepsilon'^{2/3}\right)$$

and similarly we have $\beta_i = \Sigma_1^{-1} \Sigma_{1i}$, $\sigma_{I,i}^2 = \sigma_i^2 (1 - \Phi_i)$ and the frictionless optimal illiquid asset is still:

$$\bar{\pi}_i = \frac{\mu_i - \beta_i' \mu_1}{\gamma \sigma_i^2 (1 - \Phi_i)}$$

where i denotes the i^{th} asset which is treated as the illiquid one when others are temporarily assumed to be liquid, and Φ_i is defined by:

$$\Phi_i = \frac{\Sigma_{i1} \Sigma_1^{-1} \Sigma_{1i}}{\sigma_i^2}$$

The upper bound of ESR is supposed to be:

$$ESR_{max} := \frac{\mu' \Sigma^{-1} \mu}{2\gamma} - \max_i \frac{\lambda_i^2}{2\gamma \sigma_i^2 (1 - \rho_i' \rho_i)} \tag{4.11}$$

It is clear that ideally the simulated ESR are located between ESR_{min} and ESR_{max}. With the help of equation (4.10) and (4.11), it is now feasible to quantify and compare the performance of the model with different portfolios. It becomes obvious that the

	ESR RNN(%)	ESR_{min}(%)	ESR_{max}(%)
Portfolio D	6.46	6.37	6.62
Portfolio E	4.70	4.61	4.75

Table 4.5: ESR on Portfolios D & E

simulated equivalent safe rates of both portfolios lie within the bounds of the maximum and minimum estimates of their theoretical equivalent safe rates. Consequently, the dependability of the projection and bisection model is substantiated in the context of

a three-asset scenario.

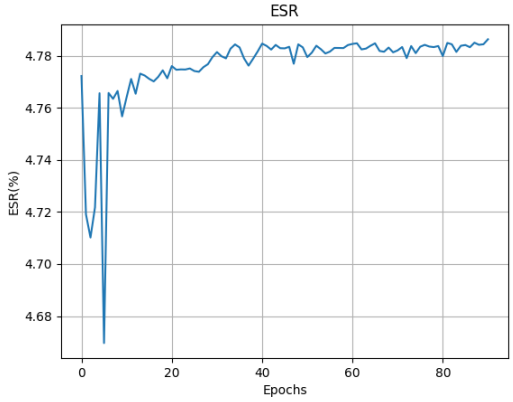
4.6 Example: Ten risky assets

Upon examining the plot of the three-asset model, we gain a degree of confidence in the model's suitability, as evidenced by the plausible shapes of the squares. However, three dimensions typically represent the upper limit for visualizing asset dimensions, implying that drawing straightforward conclusions from visualization becomes increasingly challenging as dimensions increase. Consequently, it is of paramount importance to scrutinize the model's reliability in higher-dimensional scenarios. To accomplish this, we construct a new portfolio consisting of 10 assets, maintaining $\gamma = 2$ and $\varepsilon = 1\%$.

	μ	σ	ρ	ε
Asset1∼10	0.12	0.4	0.4	1%

Table 4.6: Portfolio F

The simulated results, estimated results, together with ESR, are displayed



	ESR Min	ESR RNN	ESR Max
Portfolio F	4.76	4.79	4.88

Table 4.7: Results on Portfolio in Table 4.6

The table on the right hand side indicates the ESR in percentage (%).

In summary, the ESR function exhibits a rapid but not smooth convergence at approximately 50 epochs with a learning rate of $l = 0.01$ due to the similarities of included assets. The model is anticipated to converge more smoothly with a higher learning rate. Simultaneously, the simulated equivalent safe rate resides squarely between its

maximum and minimum approximations, which consequently furnishes compelling evidence corroborating the model's suitability for a 10-asset market. We also notice that this simulation results benefits from high correlation between assets, and when correlation reduces, model can still converge but can hardly improve the ESR from its initial input.

Chapter 5

Variance Reduction

The difficulty of the model, as is presented in the previous chapters, lies in the increase of computational resources requirement under high dimension. We aim at finding a way to reduce the number of simulated paths for decreasing the size of training data without losing accuracy of Monte Carlo simulation. This chapter introduces a short review about existing methods reducing variance with respect to Monte Carlo simulation. The importance sampling method is chosen afterwards as the technique to be utilized in this article. The scalar of importance sampling is then proved for the wealth process X_t with trading costs and is then utilized in high-dimensional assets model. The scalar is extracted from simulated data and then multiply by the output layer $X_t^{1-\gamma}$ under the new measure. Comparison of simulation results between with and without variance reduction is then presented.

5.1 Reduce Variance and Increase Speed

Both parallelogram method and projection with bisection method target at reducing the loss function (opposite of ESR). The session is then reduced to the approximation and evaluation of the loss under certain probability measure P. In practice, the loss function becomes:

$$Loss = \min_{\pi_-,\pi_+} -\bar{G}_n, \quad \bar{G}_n = \frac{1}{n}\sum_{\omega} U(X_T(\omega)) \tag{5.1}$$

The Monte Carlo simulation is then applied and loss is estimated by the sample average in equation[5.1] from the iid samples of returns and the corresponding selection of strategy. It is then clear to learn from the central limit theorem that the asymptotic confidence interval of $E_P[U(X_T(\omega))]$ is:

$$(\bar{G}_n - \xi_n, \bar{G}_n + \xi_n)$$

and

$$\xi_n = q_\alpha \frac{\sigma_n}{\sqrt{n}}, \quad \sigma_n = \frac{1}{n}\sum_{\omega}(U(X_T(\omega)) - \bar{G}_n)^2,$$

where σ_n is the sample variance and q_α is the $1 - \frac{\alpha}{2}$ quantile of the standard normal distribution. In this sense, we can only achieve relatively accurate results when we have small σ_n or large n and if there exists a large variance, we have to increase the number of path to preserve enough accuracy. However, because the model is to remember and process return and strategy data for all time steps and scenarios, the restriction of computing and memory resource suggests that we cannot increase the path n without any limitation. This tension highlights the trade off between model accuracy and computing speed.

Monte Carlo simulation has been studied for a long time and many approaches has been come up with to reduce variance and increase accuracy. The following sections review several widely-used variance reduction methods and address their benefits and drawbacks with respect to the integration to our model.

5.2 Methods for Variance Reduction

A number of methodologies has been developed to address this problem. We will discuss control variates, antithetic variates, stratified sampling, and importance sampling.

5.2.1 Antithetic Variates

The method of antithetic variates (Hammersley and Morton [1956]) aims to reduce variance by introducing negative dependence between pairs of replications and is widely used.

In particular, considering a Monte Carlo simulation Y_i driven by standard normal random variables ($i.i.d.N(0,1)$): $Z_1, Z_2, Z_3, ..., Z_n$, the method of antithetic variates simply suggest including another sequence: $-Z_1, -Z_2, -Z_3, ..., -Z_n$ to double the number of simulation and the extra samples are represented as $\hat{Y}_i$. The antithetic variates estimator is simply the average of all 2n observations. We derive:

$$Var[\frac{Y_i + \hat{Y}_i}{2}] = \frac{1}{4}(Var(Y_i) + Var(\hat{Y}_i) + 2Cov(Y_i, \hat{Y}_i)) < Var[\frac{Y_i + Y_j}{2}]$$

provided that $Cov(Y_i, \hat{Y}_i) < 0$ and Y_j is another generated sample independent from Y_i, thus the method is proved to work by reducing the efforts to generate simulations and variance simultaneously. However, the difficulty of our model is consuming too much resources to process samples when applying stochastic gradient method rather than preparing samples. Therefore, antithetic variates is not the best candidate for our requirements.

5.2.2 Control Variates

The method of control variates (Rubinstein and Marcus [1985]) is among the most widely-used techniques for enhancing Monte Carlo simulation. It exploits information about the errors in estimates of known quantities to reduce the error in an estimate of an unknown quantity.

The key point of this method is to use an alternative variable $Y_i(b)$ to estimate the original Y_i:

$$Y_i(b) = Y_i - k\,(U_i - \mathrm{E}[U])$$

where U_i is from some distribution with known expectation and known correlation

with Y_i. The mean of Y_i can be computed as:

$$\bar{Y}(b) = \bar{Y} - b(\bar{U} - \mathrm{E}[U]) = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - b\left(U_i - \mathrm{E}[U]\right)\right).$$

As a result, $\bar{Y}(b)$ is an unbiased estimation of $\mathrm{E}[Y]$. The variance of the estimator:

$$\begin{aligned} Var\left[Y_i(b)\right] &= Var\left[Y_i - b\left(U_i - \mathrm{E}[U]\right)\right] \\ &= \sigma_Y^2 - 2b\sigma_U\sigma_Y\rho_{UY} + b^2\sigma_U^2 \end{aligned}$$

could be minimized through solving a simple quadratic function and then applying $b = \frac{\sigma_Y}{\sigma_U}\rho_{UY} = \frac{Cov(U,Y)}{Var(U)}$. Consequently, we derive:

$$Var[Y_i(b)] = (1 - \rho_{UY}^2)Var[Y_i].$$

It turns out that the variance is then reduced when $\rho_{UY} > 0$. However, such a method is well applicable with a chosen candidate random variable U_i such that the expectation of U_i and the correlation between U_i and the target Y_i is known and determined. In contrast, since the potential trading strategy is not determined, it is difficult to understand the explicit distribution of the wealth process and almost impossible to choose a good candidate as the distribution U. As a result, the method of control variates might not be the best way to improve our model.

5.3 Stratified Sampling

Stratified sampling is more complicated and uses other samples which constrain the fraction of observations drawn from specific subsets of the target sample space (Glasserman [2013]). This method is mostly based on the following equation:

$$\mathrm{E}[Y] = \sum_{i=1}^{K} P\left(Y \in B_i\right)\mathrm{E}\left[Y \mid Y \in B_i\right] = \sum_{i=1}^{K} p_i \mathrm{E}\left[Y \mid Y \in B_i\right]$$

where Y is the samples to be simulated, $B_1, B_2, ...B_K$ are disjoint partitions that satisfy $P\left(Y \in \cup_i B_i\right) = 1$ and $p_i = P(Y \in B_i)$. Assuming $n_i = np_i$, then

$$\hat{Y} = \sum_{i=1}^{K} p_i \cdot \frac{1}{n_i} \sum_{j=1}^{n_i} Y_{ij} = \frac{1}{n} \sum_{i=1}^{K} \sum_{j=1}^{n_i} Y_{ij}$$

is an unbiased estimator of $\mathrm{E}[Y]$ where Y_{ij} denotes draws from the conditional distribution $Y \in B_i$. Now we process with another variable H and relevant subsets, $P\left(H \in \cup_i B_i\right) = 1$, and then we have:

$$\mathrm{E}[Y] = \sum_{i=1}^{K} P\left(H \in A_i\right) \mathrm{E}\left[Y \mid H \in A_i\right] = \sum_{i=1}^{K} p_i \mathrm{E}\left[Y \mid H \in A_i\right]$$

where $p_i = P(H \in B_i)$ and then the estimator of $\mathrm{E}[Y]$ turns out to be:

$$\hat{Y} = \sum_{i=1}^{K} p_i \cdot \frac{1}{n_i} \sum_{j=1}^{n_i} Y_{ij} = \frac{1}{n} \sum_{i=1}^{K} \frac{p_i}{q_i} \sum_{j=1}^{n_i} Y_{ij}$$

with $q_i = \frac{n_i}{n}$ being the fraction of observations drawn from stratum i. The process includes:

- Choose the candidate variable H and relevant disjoint strata $B_1, B_2, ...B_K$.
- Generate samples from the distribution of paired (H, Y) with $H \in B_i$.

The variance of the estimator turns out to be:

$$Var[\hat{Y}] = \sum_{i=1}^{K} p_i^2 Var\left[\frac{1}{n_i} \sum_{j=1}^{n_i} Y_{ij}\right] = \sum_{i=1}^{K} p_i^2 \frac{\sigma_i^2}{n_i}$$

with the optimal solution of q:

$$q_i^* = \frac{p_i \sigma_i}{\sum_{l=1}^{K} p_l \sigma_l}, \quad i = 1, \ldots, K.$$

The variance is well reduced when the stratum standard deviations vary. However, with our model, it is difficult to choose a stratum with varying standard deviations as the potential candidate for H, which is likely to be the assets price, is determined

and we are not aware of the stratum. As a result, we will not perform the method of stratified sampling.

5.4 Importance Sampling Method

Importance sampling method (Tokdar and Kass [2010]) is the general description of a set of Monte Carlo improvement methods where a mathematical expectation of a target distribution is approximated by the adjusted expectation under some other distribution. More precisely, it simultaneously changes the probability measure P and the payoff $\bar{G}$ to keep the same expectation and meanwhile, significantly reduce variance (Guasoni and Robertson [2008]). The target of our problem, $E_P[U(X_T)]$ is redefined as:

$$E_P[U(X_T)] = E_Q[H_T]$$

where $H_T = U(X_T)\frac{dP}{dQ}$. With the feature of power utility, it is equivalent to start with $E_P[X_T^{1-\gamma}]$ instead of $E_P[U(X_T)]$ as $E_P[U(X_T)] = \frac{(E_P[X_T^{1-\gamma}]-1)}{1-\gamma}$. The target is to find H_T and $\frac{dP}{dQ}$ which indicates the Brownian Motion under Q measure. As a result, we then introduce

$$E_P[X_T^{1-\gamma}] = E_Q[X_T^{1-\gamma}\frac{dP}{dQ}] \tag{5.2}$$

where:

$$\frac{dQ}{dP} = \exp\left(-\frac{1}{2}\int_0^T \dot{h}_t^2 dt + \int_0^T \dot{h}_t dW_t\right) \tag{5.3}$$

which indicates

$$W_t^Q = W_t - h_t \tag{5.4}$$

W^Q is a Q-Brownian Motion by Cameron-Martin theorem and Gisanov theorem with $\dot{h}_t$ being the first order derivative of h_t. In this problem, we expect the unique solution of $\dot{h}_t$ to be a constant vector. To solve the problem, we formulate the dynamic of X_T:

$$dX_t = \mu\pi_t X_t dt + \sigma\pi_t X_t dW_t - \varepsilon S_t |d\phi_t|$$

and it turns out that:

$$X_T = \exp\left(\int_0^T (\mu\pi_t - \frac{1}{2}\sigma^2\pi_t^2)dt + \int_0^T \sigma\pi_t dW_t - \int_0^T \varepsilon\pi_t dv_t \right) \tag{5.5}$$

under the P-measure with $dv_t = \frac{|d\phi_t|}{\phi_t}$ as the term describing the trading cost. Since randomness comes from the $\int_0^T \sigma\pi_t dW_t$ term, with the listed equations, we can derive the unique solution of h_t with the help of the Euler-Lagrange equation.

$$h_t = (1-\gamma)\sigma\bar{\pi}t, \quad \dot{h}_t = (1-\gamma)\sigma\bar{\pi} \tag{5.6}$$

where $\bar{\pi}$ is the Merton optimal strategy $\frac{\mu}{\gamma\sigma^2}$ without trading costs ε under one dimension and $\frac{\Sigma^{-1}\mu}{\gamma}$, $\quad \sigma\sigma^T = \Sigma$ for more than one asset. The progress of optimisation turns into:

$$E_P[U(X_T)] = \frac{(E_P[X_T^{1-\gamma}] - 1)}{1-\gamma} = \frac{1}{1-\gamma}\left((E_Q[X_T^{1-\gamma}\frac{dP}{dQ}] - 1) \right)$$

where:

$$\frac{dP}{dQ} = \exp\left(-\frac{1}{2}\int_0^T \dot{h}_t^2 dt - \int_0^T \dot{h}_t dW_t \right)$$

With Q-measure, the X_t is simulated with the same volatility but an transformed drift.

$$\hat{\mu} = \mu + \sigma\dot{h}_t$$

From a theoretical perspective, the variance is eliminated and scenarios could be reduced to only one under the assumption of no trading costs. With the existence of trading costs, the story is different as the volatility term is $((1-\gamma)\int_0^T(\pi_t - \bar{\pi})\sigma dW_t)$ rather than 0 since $\pi_t - \bar{\pi} \neq 0$. However, we can still expect that the importance sampling method could notably reduce the variance (hardly to 0) and computing complexity as the difference $\pi_t - \bar{\pi}$ is not supposed to be big when the strategy π_t will be controlled by some upper bound and lower bound with the important assumption of small transaction costs.

5.5 Results Comparison

In order to validate the performance of the importance sampling method, we start the comparison from the simple case with simulated basic assets. The comparison will include direct plots of no-trade regions.

5.5.1 One Asset Case

We start from the basic one asset model, we should notice from previous chapter that this problem with relevant no-trade region has already been resolved and we can compare directly with the asymptotic solution. We use the asset below:

μ	σ	T	ε	γ	Nsteps
0.02	0.2	10	1%	2	120

The results without importance sampling contains the comparison between the same asset with two different number of scenarios.

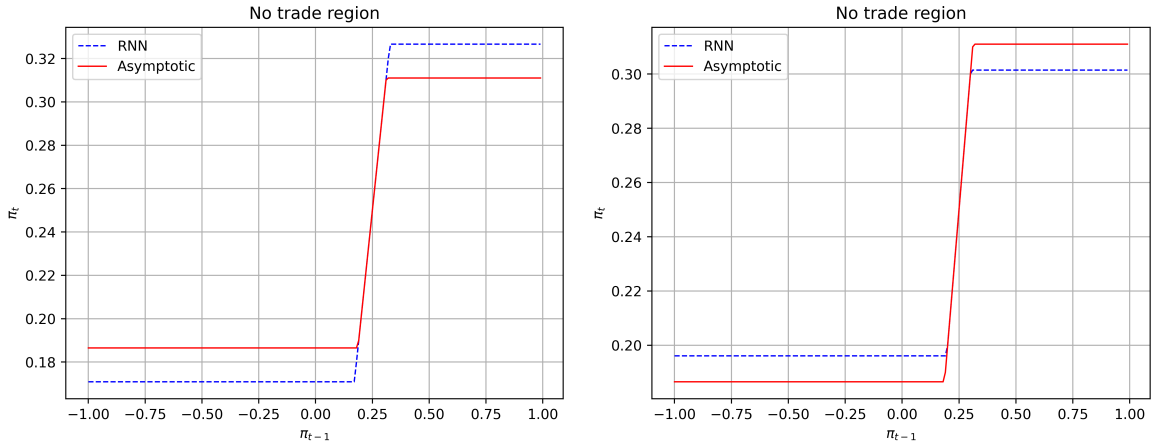


Figure 5.1: No Trade Region: 3000 Paths & 10000 Paths

The left plot is from 3000 scenarios and the right plot is from 10000 scenarios

Figure 5.1 indicates that with the increasing number of scenarios, there is no doubt that the simulated no-trade region turns to be close to the asymptotic solution, which means increase the scenarios can help reduce variance. We should also notice the randomness of the samples since 3000 and 10000 are not a large number of scenarios and the plot could vary due to a changing random seed.

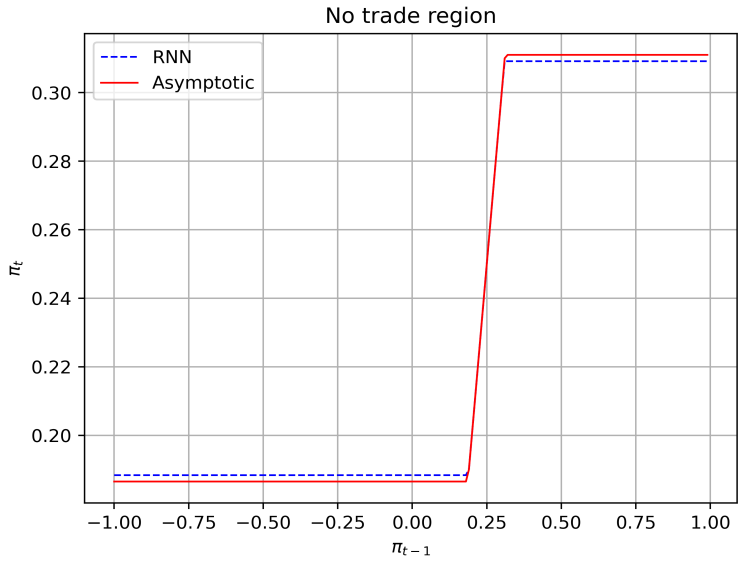


Figure 5.2: No Trade Region: 3000 Paths with Importance Sampling

Figure 5.2 shows that with the same asset and the importance sampling method, the 3000 scenarios simulation performs better than the 10000 scenarios simulation without importance sampling, accurately reproducing the asymptotic solution.

Method	IS	ESR(%)	LOW	UP
3000	NO	0.194	0.171	0.323
10000	NO	0.218	0.196	0.302
3000	YES	0.223	0.188	0.309
Asymptotic	-	0.232	-	-
NTR	-	-	0.187	0.311

Table 5.1: One Asset Importance Sampling Performance on ESR

IS denotes importance sampling, NTR denotes asymptotic no trade region while low and up refers to lower bound and upper bound of such no trade region respectively

The equivalent safe rates estimation also suggests the superiority of importance sampling method as the 3000 scenarios simulation of importance sampling provides a equivalent safe rate high than the 10000 scenarios simulation. The ESR is closer to the asymptotic derived from equation (3.4). The examples and the theory indicates that importance sampling method significantly improves the simulation and reduces variance in some situation without introducing extra computing requirements in the one-asset model.

5.5.2 Two Assets Case

To validate the importance sampling model with more than one asset, we still begin with the dynamic portfolio model established by Muthuraman and Kumar [2004] for consistency. The portfolios in Table 3.2 contains 2 same assets with covariance matrix:

$$\Sigma = \begin{bmatrix} 0.4 - k & k \\ k & 0.4 - k \end{bmatrix}$$

with $k \in (-\infty, 0.2]$. The covariance matrix preserves the feature of a constant Merton optimal strategy with varying correlations. We compare different numbers of scenarios with and without importance sampling.

μ_1	μ_2	k	T	ε	γ	Nsteps
0.12	0.12	0.15	10	1%	2	120

The results presented in Table 5.2 clearly demonstrate the improvements made by importance sampling method. The ESR of direct ETF strategy without doing anything about trading costs are comparatively low in all scenarios dependent on the rebalancing frequency. With the increase of scenarios, accuracy of all results increases. The performance of 10000 scenarios with importance sampling method is close to 30000 scenarios with importance sampling, and is better than both 10000 and 30000 scenarios without importance sampling, providing a higher ESR and smaller confidence interval. The results indicate that importance sampling method plays positive role in reducing computing complexity and variance. However, it still requires a certain number of scenarios to maintain the accuracy of simulation, as the variance cannot be completely removed.

Npaths	IS	ESR Merton(%)	ESR ETF(%)	ESR NTR(%)	ESR RNN(%)
10000	Yes	1.80	1.48	1.68 (4.33)	1.70 (4.32)
30000	Yes	1.80	1.48	1.68 (4.33)	1.70 (4.32)
10000	No	1.80	1.45	1.65 (4.30)	1.66 (4.29)
30000	No	1.80	1.45	1.65 (4.30)	1.67 (4.30)

Table 5.2: Two Assets Importance Sampling Performance

IS denotes importance sampling, NTR represents the simple no-trade region strategy without considering the correlation

Chapter 6

Empirical Results

This chapter presents an introduction to the fundamental concepts of tracking error and tracking difference, two pivotal metrics extensively employed in the comparative analysis of Exchange-Traded Fund (ETF) performance and a specific benchmark. The application of these metrics to an equally-weighted ETF serves as a demonstrative case study, illustrating the efficacy of the RNN methodologies in mitigating both tracking error and tracking difference from an empirically determined target.

6.1 Evaluating Exchange-Traded Funds (ETFs)

The goal of the thesis, overall, is to relate this RNN model to empirical findings on trading strategy modification considering the impact of trading costs. As the strategy deal with portfolios, and the most common portfolios one can find in the market are Exchange-Traded Funds, a type of investment fund that tracks an index, a commodity, bonds, a basket of assets, or other asset types and is traded on stock exchanges, which are known as ETFs. A passive ETF is a type of ETF that aims to replicate the performance of a particular index or benchmark, and the strategy of a passive ETF could be clear, making it possible to track. The tracking difference (TrD):

$$\begin{aligned} TrD_T &= 100 * ((1 + R_{iT} - R_{bT})^{\frac{1}{T}} - 1) \\ &\approx \frac{1}{T}(R_{iT} - R_{bT}) * 100 \end{aligned} \tag{6.1}$$

is the annualized average of difference between the fund and the benchmark. $R_{iT} - R_{bT}$ is the cumulative difference of returns during a specific time period and T is the number of years during such period. The tracking error (TrE):

$$TrE_T = \sqrt{\frac{1}{T}\sum_{t=1}^{KT}(r_{it} - r_{bt} - \overline{TrD_T})^2} \tag{6.2}$$

describes the annualized standard deviation of the fund's and the benchmarkâĂŹs returns difference, where r_{it} and r_{bt} are returns of the fund and benchmark during a time interval and K is number of such intervals in a year. Both tracking error and tracking difference are widely used in Frino and Gallagher [2001], Charteris and McCullough [2020] and Guasoni and Mayerhofer [2023] to evaluate the performance of ETFs and leveraged portfolios. This thesis is going to use both tracking error and tracking difference together with equivalent safe rate(ESR) to evaluate the performance of trading cost mitigation with the RNN model. In detail, we choose the returns of strategies modified in different ways as the input r_{it} and the ideal returns of optimal strategy without transaction cost as the benchmark r_{bt}.

6.2 Model Performance on Small ETF

ALPS [2023] is a small Fund of Fund (FOF) product using equal weight strategy to replicate NYSE Equal Sector Weight Index (NYXLEW) and it is rebalanced quarterly. The target is to replicate the ETF in three different ways. One way is to replicate with rigorously equal weight strategy with zero trading cost, and this provides a multiple of the indexâĂŹ return above the safe rate as a benchmark. The second way is to replicate with a simple no trade region (simple NTR) strategy:

$$\pi_{\pm} = \pi_* \pm (\frac{3}{4\gamma}\pi_*^2(1-\pi_*)^2)^{1/3}\varepsilon'^{1/3}$$

which approximately considers the proportional trading costs with a no trade region in the absence of correlation. The last way is to replicate with simulated strategy from

the RNN model which consider the proportional trading costs with a no trade region with regard to correlation. With varying trading costs, we simulate training data with parameters:

T	γ	Npaths	Nsteps
10	4	20000	120

The covariance matrix, noted as Σ, is sourced from Yahoo Finance, covering the returns from August 1998 to June 2023 with:

$$\mu = \gamma \Sigma \pi_*$$

Where π^* denotes the equal weight strategy, and this equation lead to the assumption of drift μ of assets included in the ETF. For convenience, we construct a testing data set with exactly same input parameters but different random seed and the ESR performance of testing data implies positive results.

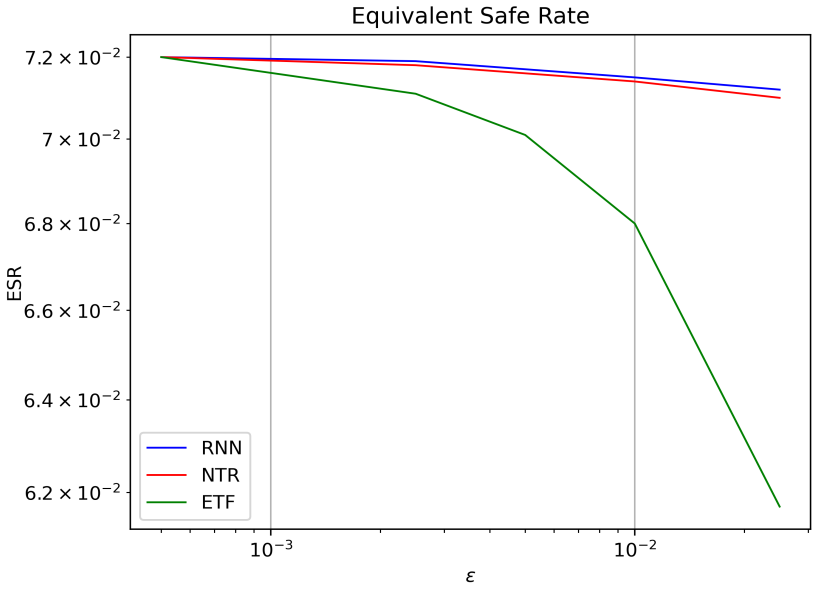


Figure 6.1: EQL ESR Estimation with Different Costs

Results in Figure 6.1 clarifies that with the increase of trading cost, the ETF strategy which is strictly equal weight, witnesses a significant decrease in terms of its ESR, indicting the challenge to replicate the optimal return of a strategy which trades illiquid assets.

ε (%)	ESR Merton(%)	ESR ETF(%)	ESR NTR(%)	ESR RNN(%)
0.05	7.20	7.20	7.20 (5.97)	7.20 (5.97)
0.25	7.20	7.11	7.18 (5.95)	7.19 (5.99)
0.50	7.20	7.01	7.16 (5.95)	7.17 (5.95)
1.00	7.20	6.80	7.14 (5.95)	7.15 (6.00)
2.50	7.20	6.17	7.10 (5.97)	7.12 (6.01)

Table 6.1: Strategy ESR on EQL

ETF denotes the direct ETF strategy without no-trade region, NTR represents the simple no-trade region. The evaluating data including returns of EQL and relevant ETFs from August 2009 to June 2023

Meanwhile, the RNN strategy, exceeds both simple no trade region strategy with only slight lead and both are much better than ETF strategy. This could be due to the low correlation between assets. In fact, the assets included in the ETF are indexes of different industrial sectors which, have low correlation.

ε (%)	TrD RNN(%)	TrD ETF(%)	TrD NTR(%)	TrE RNN(%)	TrE ETF(%)	TrE NTR(%)	t RNN	t ETF	t NTR
0.05	0.22	-0.39	0.21	0.50	1.20	0.49	1.65	-1.22	1.63
0.25	0.39	-0.39	0.42	0.65	1.20	0.71	2.24	-1.22	2.18
0.50	0.42	-0.39	0.46	0.62	1.20	0.81	2.47	-1.22	2.12
1.00	0.57	-0.39	0.51	0.94	1.20	0.94	2.25	-1.22	1.99
2.50	0.37	-0.39	0.47	1.37	1.20	1.16	0.99	-1.22	1.50

Table 6.2: Strategy Performance on EQL

ETF denotes the direct ETF strategy without no-trade region, NTR represents the simple no-trade region. The evaluating data including returns of EQL and relevant ETFs from August 2009 to June 2023

The result presented in Table 6.2 indicates that with extremely small trading costs, the tracking difference and tracking error are also extremely small, and both mitigated strategies do not surpass the ETF strategy significantly. As trading costs slowly increases, both strategy become significant and the RNN model start to function to improve the strategy. However, when trading costs reaches a certain level, a decrease in significance appears again. As all assets are very illiquid, there is hardly a "close" strategy to keep high tracking difference and low tracking error simultaneously. In this situation tracking error retains a high level to maintain a high tracking difference due to a low level of risk aversion. In general, the comparison indicates that the RNN model surpasses the strategy of the ETF significantly with better tracking difference

and less tracking error. However, the RNN model does not surpass the simple no-trade region strategy by a large margin: it is better, but only to a small extent. As a result, it would be meaningful to use both modification methods to improve the performance of an equally-weighted ETF, but the simple no-trade region method might be an easier solution in some cases when assets correlations are low.

(a) $\varepsilon = 0.05\%$

Assets	XLE	XLF	XLP	XLV	XLB	XLK	XLY	XLI	XLU	XLRE	XLC
$\frac{1}{\sqrt{\xi_i}}$	0.023	0.046	0.064	0.074	0.099	0.124	0.175	0.213	1.980	0.340	0.453
$\lvert\alpha_i\rvert$	0.119	0.011	0.123	0.128	0.067	0.061	0.007	0.130	0.092	0.017	0.079
ED	1.967										

(b) $\varepsilon = 0.25\%$

Assets	XLE	XLF	XLP	XLV	XLB	XLK	XLY	XLI	XLU	XLRE	XLC
$\frac{1}{\sqrt{\xi_i}}$	0.027	0.034	0.054	0.072	0.118	0.110	0.177	0.207	0.280	1.080	0.517
$\lvert\alpha_i\rvert$	0.021	0.023	0.089	0.089	0.023	0.024	0.10	0.100	0.11	0.100	0.025
ED	1.202										

(c) $\varepsilon = 0.5\%$

Assets	XLE	XLF	XLP	XLV	XLB	XLK	XLY	XLI	XLU	XLRE	XLC
$\frac{1}{\sqrt{\xi_i}}$	0.034	0.038	0.078	0.089	0.122	0.200	28.087	0.487	0.376	0.260	0.286
$\lvert\alpha_i\rvert$	0.020	0.018	0.130	0.139	0.022	0.018	0.117	0.094	0.158	0.013	0.019
ED	27.977										

(d) $\varepsilon = 1\%$

Assets	XLE	XLF	XLP	XLV	XLB	XLK	XLY	XLI	XLU	XLRE	XLC
$\frac{1}{\sqrt{\xi_i}}$	0.041	0.089	0.114	0.189	0.207	0.263	0.298	0.361	0.431	0.974	7.775
$\lvert\alpha_i\rvert$	0.034	0.027	0.127	0.140	0.033	0.030	0.144	0.107	0.166	0.033	0.034
ED	7.814										

(e) $\varepsilon = 2.5\%$

Assets	XLE	XLF	XLP	XLV	XLB	XLK	XLY	XLI	XLU	XLRE	XLC
$\frac{1}{\sqrt{\xi_i}}$	0.186	0.250	0.351	0.339	0.498	0.819	1.078	2.013	5.920	1.966	3.352
$\lvert\alpha_i\rvert$	0.033	0.192	0.097	0.171	0.193	0.029	0.165	0.140	0.180	0.177	0.046
ED	7.229										

Table 6.3: Comparison of Axes Lengths and boundaries on EQL ETF

Table 6.3 compares the performance of optimizing the ETF using the hyperparallelogram and hyperellipsoid methodologies respectively. Analogous to the results observed for Portfolio F, the hyperparallelogram exports stable $|\alpha_i|$ whereas the hyperellipsoid method occasionally generates exploding axe lengths on certain assets. Experiments have been done with different number of scenarios, learning rates and random seed while results remain with extreme axis length values (eigenvalues) which may vary but can hardly been reduced significantly. Table 6.4 apply the same metrics

in Table 6.2 to evaluate the tracing error and tracking difference of such ETF with the hyperellipsoid method. Results indicates that the hyperellipsoid, can always target at improving tracing error at the cost of higher tracing difference. It performs well with low transaction costs. However, if transaction costs increase, the tracking differences start to increase and ends up with low t-value causing it not as good as the simple no-trade region algorithm. The reason is that overall the mathematically optimized no-trade region can never be an exact hyperellipsoid with low correlation and the behaviors of some axe lengths, possibly because of gradients also reduce accuracy and this problem will be further studied.

ε (%)	TrD ELIP(%)	TrD ETF(%)	TrD NTR(%)	TrE ELIP(%)	TrE ETF(%)	TrE NTR(%)	t ELIP	t ETF	t NTR
0.05	0.49	-0.39	0.21	0.94	1.20	0.49	1.93	-1.22	1.63
0.25	0.59	-0.39	0.42	0.99	1.20	0.71	2.2	-1.22	2.18
0.50	0.51	-0.39	0.46	0.92	1.20	0.81	2.06	-1.22	2.12
1.00	0.60	-0.39	0.51	1.18	1.20	0.94	1.91	-1.22	1.99
2.50	0.56	-0.39	0.47	1.79	1.20	1.16	1.16	-1.22	1.5

Table 6.4: Hyperellipsoid Strategy Performance on EQL

Evaluating data is similar to Table 6.2 and ELIP refers to results from the hyperellipsoid method.

The concurrent consideration of hyperparallelogram and hyperellipsoid models in portfolio optimization underscores a fundamental principle in quantitative finance: the existence of multiple variable approaches to strategy optimization. This multiplicity of solutions presents each method's applicability, strengths, and limitations. It is imperative to delineate clearly the contexts in which each approach is most appropriately and effectively employed.

Chapter 7

Conclusion

In summary, this thesis investigates the effect of transaction costs on portfolio rebalancing, and develops an algorithm capable of efficiently approximating a trading strategy accounting for transaction costs. Leveraging this algorithm, machine learning theory, and pertinent mathematical theory on asymptotic analysis of the no-trade region, we have devised a supervised recurrent neural network model to address the multi-asset trading problem with transaction costs. This model effectively incorporates existing findings, yielding numerically optimal strategies given a specific portfolio's risk premium and covariance.

The primary contribution of this thesis lies in presenting a model that tackles high-dimensional trading problems with transaction costs without sacrificing interpretability from both mathematical and financial perspectives, by concurrently integrating relevant mathematical and financial insights with machine learning methodologies. Thus, utilizing this model to achieve an effective trading strategy is promising. The industry may find it appealing to adopt a mathematically explicable model that assists in managing a fixed-assets portfolio or modifying an existing strategy, such as an original equal-weighted ETF, by assuming $\mu = \gamma\sigma\pi^*$. Further research of model application to small ETF demonstrates that our strategy surpasses established equal-weight and basic buy-and-hold strategies with small transaction costs.

The model does have a few limitations where Zero safe rates can be easily upgraded with a modified drift term. First and the most obvious is that it does not work

with fixed trading costs, and as a result, it is not suitable to those assets with high fixed trading costs, such as Crypto currencies. More specifically, considering a power utility with constant risk aversion, Altarovici et al. [2016] point out that the optimal strategy with proportional and fixed transaction costs under one dimension is a double-boundary no-trade region. The region has a narrow boundary covered by a wider boundary. The trader should keep passive within the wider no-trade region and trade when the weight exceeds the wider no-trade region, which leads to trading the weight of the asset back to the narrow boundary instead of the wider one. It is obvious that if the proportion of the fixed costs is extremely small compared to the overall wealth X_t, then both no-trade regions converge and the effects of fixed costs is small. This method can be done by updating the activation function with an extra boundary covering the original no-trade region. Our model can also replicate the case by redefining the activation function on the basis that one asset can be extended to more assets model. This work could be covered in future research.

The model's underlying theory relies on constant volatility, potentially hindering its performance in markets better characterized by stochastic volatility models. Consumption rate c_t is not involved and it should be modeled in the future as consumption model usually possesses greater research significance assuming the manager continuously charges management fees. The model is also challenged by the real market with low correlation. It is evident that most assets, especially those widely used as parts of ETFs, are not highly dependent on each other. When the correlation reduces to a certain level (i.e. 10%), it sometimes turns out that the simple no trade region and buy and hold strategy is more cost effective in terms of calculation. Rebalancing frequency is also a factor; higher frequencies necessitate more computational resources for larger simulation datasets over a trading horizon and relevant market regulations may affect availability at specific trading frequencies (e.g., the Chinese market).

Additionally, the simulation and deployment of the model requires significant memory and computing resources, with demands increasing as portfolio size grows. This problem is partially resolved by introducing the hyperellipsoid model in the previous

chapters. However, this method is not perfect and causes inaccuracy in strategies of some assets when dealing with large dimension portfolio. Future work will involve model fix, improvement and computing resource optimizing for this model.

To summarize, our research offers an algorithm capable of optimizing trading strategies for multiple-assets portfolios with transaction costs. This algorithm is not only amenable to programming but also mathematically and financially interpretable, and compares favorably to existing approaches in the literature.

Appendix A

Codes

In this section, relevant codes are included but only key contents will be covered, for codes that can execute directly, move to Github: Li [2024].

A.1 One Asset

A.1.1 Functions

```
# Function to calculate the final return with trading cost with
    respect to a particular strategy
def cal_return(strat, returns, cost, n_iterations=5):
    r =  strat[:-1, :]*returns
    r0 = strat[:-1, :]*returns
    for _ in range(n_iterations):
        r = r0-cost*abs((r+1)*strat[1:, :]-(returns+1)*strat[:-1, :])
    return r
```

Code Listing A.1: Calculate return

```
# Module for importance sampling
def importance_sampling(is_importance,seed,mu,sigma,s0,npaths,
    seq_length,gamma,T):
    """
    Simulates stock returns under a different probability measure
    using importance sampling.
```

```
5
6     Parameters:
7     - is_importance (bool): Flag to determine if importance sampling
      should be used.
8     - seed (int): Random seed for reproducibility.
9     - mu (float): Drift rate of the stock under the original measure.
10    - sigma (float): Volatility of the stock.
11    - s0 (float): Initial stock price.
12    - npaths (int): Number of simulation paths.
13    - seq_length (int): The sequence length of the simulation.
14    - gamma (float): Risk aversion parameter.
15    - T (float): Time horizon for the simulation.
16    - device (str or torch.device): The device on which to perform
      the computations.
17
18    Returns:
19    Tuple of (mu_importance, returns, scaler) under the Q measure.
20    """
21    if not is_importance:
22        stock = dg.OneStock(seed,mu,sigma,s0,npaths,seq_length-1,T)
23        returns = torch.tensor(stock.Returns(),dtype=torch.float).to(
      device).transpose(0,1)
24        scaler = 1
25
26    else:
27        # define the h
28        h = (1-gamma)/gamma*mu/sigma
29        # define the new drift under Q measure
30        mu_importance = (1-gamma)/gamma*mu+mu
31        # Simulate stock under Q measure
32        stock = dg.OneStock(seed,mu_importance,sigma,s0,npaths,
      seq_length-1,T)
33        # Returns under Q measure
34        returns = torch.tensor(stock.Returns(),dtype=torch.float).to(
      device).transpose(0,1)
```

```
        # Extract Brownian Motion under Q measure
        BM_last = stock.BM()[-1,:]
        # The scaler
        scaler  = torch.exp(torch.tensor(-1/2*h*h*T-h*BM_last)).to(
    device)
    return returns, scaler
```

Code Listing A.2: Importance Sampling

```
# Make a portfolio
def make_portfolio(seed,mu,sigma,s0,npaths,seq_length,T,trading_cost,
    gamma):
    # Create a stock simulation with prices, returns
    # seed, mu, sigma, S0, paths, steps, T
    # Define the Merton_optimal strategy
    Merton_opt = mu/(sigma*sigma*gamma)
    # Define the distance for initial non trade region
    delta = np.power(np.power(Merton_opt*(1-Merton_opt)*trading_cost
    ,2),1/3)
    stock = dg.OneStock(seed,mu,sigma,s0,npaths,seq_length-1,T)
    returns = torch.tensor(stock.Returns(),dtype=torch.float).to(
    device).transpose(0,1)
    # Create a default strategy as initial input, better use the
    optimal strategy without cost
    strategy = Merton_opt*torch.ones((seq_length,1),dtype=torch.float
    ).to(device)
    # Create a trading cost
    cost =  torch.tensor(trading_cost*np.ones([seq_length-1,1]),dtype
    =torch.double).to(device)
    return returns, strategy, cost, Merton_opt, delta
```

Code Listing A.3: Simulate Portfolio

```
def make_model(input_size, hidden_size, n_layers, npaths, seq_length,
    delta, gamma, learning_rate):
    model = NOA.WealthRNN(input_size, hidden_size, n_layers, npaths,
    seq_length).to(device)
```

```
    model.update_bias(delta)
    model.to(device)
    # Use utility as the loss function
    criterion = UL.PowerUtilityLoss(gamma)
    #model.rnn.fc2_param.bias.requires_grad = True
    optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.
    parameters()), lr=learning_rate)
    return model, criterion, optimizer
```

Code Listing A.4: Create Model

```
# Train and plot model
def train_model(strategy, target, returns, cost, scaler, model,
    criterion, optimizer,n_epochs,gamma,T):
    losses = np.zeros(n_epochs)
    for epoch in range(n_epochs):
        fina_strat, outputs = model(strategy.double(), target,
    returns, cost, None)
        loss = criterion(outputs,scaler)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses[epoch] += loss
    # Plot the model
    POA.loss_esr(n_epochs, losses,gamma,T)
    for name, param in model.named_parameters():
        print (name, param.data)
    return model, losses
```

Code Listing A.5: Train Model

```
# Calculate different ESR:
def ESR(mu,sigma,gamma,data, T,trading_cost, confidence=0.95):
    # Merton strategy
    Merton_opt = mu/(sigma*sigma*gamma)
    # ESR from the RNN simulation
    ESR_simulated = 1/T*np.log(np.power(np.mean(np.power(data,1-gamma
```

```
    )),1/(1-gamma)))
    # Confidence interval
    CI = 1/T*np.log(np.power(calculate_confidence_interval(np.power(
    data,1-gamma),confidence),1/(1-gamma)))
    # Theoretical ESR without cost
    ESR_opt = mu*mu/sigma/sigma/2/gamma
    # Asymtotic ESR
    ESR_real = mu*mu/sigma/sigma/2/gamma-gamma*sigma*sigma/2*np.power
    (trading_cost,2/3)*np.power(3/2/gamma*Merton_opt*Merton_opt*(1-
    Merton_opt)*(1-Merton_opt),2/3)
    return ESR_simulated, CI, ESR_opt, ESR_real
```

Code Listing A.6: Calculate ESR

A.1.2 Neural Networks

```
# Customize a RNN layer with double relu
# considering returns data to build a changed strategy weight
   according to price change
class NoTradeRegionRNN(nn.Module):

    def __init__(self, input_size, hidden_size, batch_size):
        """Initialize params."""
        super(NoTradeRegionRNN, self).__init__()
        # read input parameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size

        self.input_param = nn.Linear(input_size,  hidden_size,bias =
    False).to(device)
        self.hidden_param = nn.Linear(hidden_size,  hidden_size).to(
    device)
        self.fc1_param = nn.Linear(hidden_size,hidden_size).to(device
    )
```

```
16        self.fc2_param = nn.Linear(hidden_size,hidden_size,bias =
    False).to(device)
17
18     # Forward function allows a form:
19     # h_t = w_fc2*relu(w_fc1*relu(w_inp*x_t+b_inp+w_h*h_{t-1}+b_h)+
    b_fc1)+b_fc2+b_fc1-b_h1
20     def forward(self, input, target, returns, hidden):
21        pi_bar = torch.tensor(target,dtype=torch.float).to(device)
22        """Propogate input through the network."""
23        def recurrence(input, hidden):
24            """Recurrence helper."""
25
26            ingate = self.input_param(input) + self.hidden_param.
    weight*hidden-(pi_bar-self.hidden_param.bias)
27            ingate2 = self.fc1_param.weight*F.relu(ingate)+2*self.
    hidden_param.bias
28            h = self.fc2_param.weight*F.relu(ingate2)+pi_bar+self.
    hidden_param.bias
29            return h
30
31        # Loop to formulate the rnn
32        output = []
33        steps = range(input.size(0))
34        myret = returns.view(input.size(0)-1,self.batch_size,self.
    hidden_size)
35        for i in steps:
36            if i ==0:
37                hidden = input[0]*torch.ones(self.hidden_size,self.
    batch_size,self.hidden_size).to(device)
38            else:
39            # pi_t = pi_{t-1}*(1+r_t)/(1+pi_{t-1}*r_t) due to change
    of price after rebalance
40                hidden = recurrence(input[i], hidden*(1+myret[i-1])
    /(1+hidden*myret[i-1]))
41            if isinstance(hidden, tuple):
```

```
                output.append(hidden[0])
            else:
                output.append(hidden)

        output = torch.cat(output, 0).view(input.size(0), self.
    batch_size, self.hidden_size)
        return output, hidden
```

Code Listing A.7: RNN layer of no-trade region

```
class WealthRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers, batch_size,
     seq_length):
        super(WealthRNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.batch_size = batch_size
        self.seq_length = seq_length
        # the rnn layer which works as out, hidden_t = f(out_(t),
    hidden_(t-1)), used to approximate pi^*_(t)= f(pi^*_(t-1),pi_t)
        self.rnn = NoTradeRegionRNN(input_size, hidden_size,
    batch_size).to(device)
        self.out = nn.Linear(hidden_size, hidden_size,bias=False).to(
    device)
        # initialize some bias and weight
        self.rnn.input_param.weight = torch.nn.Parameter(torch.zeros
    (1,1))
        self.rnn.hidden_param.weight = torch.nn.Parameter(torch.ones
    (1,1))
        self.rnn.hidden_param.bias = torch.nn.Parameter(0*torch.ones
    (1,1))
        self.rnn.fc1_param.bias = torch.nn.Parameter(0*torch.ones
    (1,1))
        self.rnn.fc1_param.weight = torch.nn.Parameter(-1*torch.ones
    (1,1))
```

```
18        self.rnn.fc2_param.weight = torch.nn.Parameter(-1*torch.ones
    (1,1))
19        self.out.weight = torch.nn.Parameter(torch.ones(hidden_size,
    hidden_size))
20
21     def update_bias(self,value):
22        self.rnn.hidden_param.bias = torch.nn.Parameter(value*torch.
    ones(1,1))
23        self.out.weight.requires_grad = False
24        self.rnn.input_param.weight.requires_grad = False
25        self.rnn.hidden_param.weight.requires_grad = False
26        self.rnn.fc1_param.weight.requires_grad = False
27        self.rnn.fc2_param.weight.requires_grad = False
28
29     def step(self, input, target, returns, cost, hidden=None):
30        output, hidden = self.rnn(input, target, returns, hidden).to(
    device)
31        output2 = self.out(output)
32        return output, output2
33
34     def forward(self, inputs, target, returns, cost,hidden=None):
35     #     hidden = self.__init__hidden().to(device)
36        hidden = inputs[0]*torch.ones(self.n_layers, self.batch_size,
     self.hidden_size, dtype=torch.float64).to(device)
37        output, hidden = self.rnn(inputs.float(), target, returns,
    hidden.float())
38        # output_temp the overall wealth at time T for importance
    sampling
39        output2 = torch.prod(FOA.cal_return(output.float().view(self.
    seq_length,self.batch_size),returns,cost).to(device)+1,0)
40        return  output, output2
```

Code Listing A.8: Nonlinear layer of wealth process

A.2 Two Assets

A.2.1 Functions

```
# Create a square simple no trade region strategy without correlation
def simpleNTR(input, returns, upper, lower):
    output = []
    steps = range(input.size(1))
    for i in steps:
        if i == 0:
            hidden = input[:, 0, :].view(input.size(0), 1, input.size(2)).to(device)
        else:
            adjust_pi = hidden.view(input.size(0), 1, input.size(2)) * (1 + returns[:, i - 1, :].view(input.size(0), 1, input.size(2))) \
                        / (1 + torch.sum(hidden.view(input.size(0), 1, input.size(2)) * returns[:, i - 1, :].view(input.size(0), 1, input.size(2)), 0))
            # Apply the simple no trade region strategy
            hidden = torch.where(adjust_pi < lower, lower, torch.where(adjust_pi > upper, upper, adjust_pi))
        output.append(hidden)
    output = torch.cat(output, 1)
    return output
```

Code Listing A.9: Create a simple no-trade region

```
# Create function which calculate ESR with strategy returns and cost
def cal_esr(strategy, returns, cost, utility_gamma, T, scaler = 1):
  x = torch.prod(cal_return(strategy,returns,cost)+1,0)
  esr = torch.log(torch.pow(torch.mean(torch.pow(x,1-utility_gamma)*scaler),1/(1-utility_gamma)))/T
  #I = np.log(np.power(confidence_interval((torch.pow(x,1-utility_gamma)*scaler).detach().cpu().numpy()),1/(1-utility_gamma)))/T
```

```
  std = torch.std(torch.log(torch.pow((torch.pow(x,1-utility_gamma)*
   scaler),1/(1-utility_gamma)))/T)
  return esr,std
```

Code Listing A.10: Calculate ESR and its std

```
# A function that compares different ESR under different models
def ESR(input_size, hidden_size, n_layers,num_stocks, seq_length,
   npaths,\
        model_state_dict,  strategy, returns, cost, utility_gamma, T,
    scaler = 1, confidence = 0.95):

    batch_size = npaths
    seq_length = seq_length
    dim_size = num_stocks
    # create a simple no trade region
    lower = strategy[:,0,:].view(num_stocks,1,npaths)-torch.pow(1.5/
   utility_gamma*(strategy*strategy*(1-strategy)*(1-strategy))
   [:,0,:].view(num_stocks,1,npaths)*cost[:,0,:].view(num_stocks,1,
   npaths),1/3)
    upper = strategy[:,0,:].view(num_stocks,1,npaths)+torch.pow(1.5/
   utility_gamma*(strategy*strategy*(1-strategy)*(1-strategy))
   [:,0,:].view(num_stocks,1,npaths)*cost[:,0,:].view(num_stocks,1,
   npaths),1/3)
    stra_NTR_the = simpleNTR(strategy, returns, upper, lower)
    # create a new model
    model3 = NTA.WealthRNN(input_size, hidden_size, n_layers,
   batch_size, seq_length,dim_size).to(device)
    # load saved data
    model3.load_state_dict(model_state_dict)
    # outputs of testing data
    _, outputs = model3(strategy.to(device),strategy[:,0,0], returns,
    cost, None)
    # condidence interval
    CI = 1/T*np.log(np.power(confidence_interval(np.power(outputs.
   detach().cpu().numpy(),1-utility_gamma),confidence),1/(1-
```

```
    utility_gamma)))

    return cal_esr(strategy,returns,cost,utility_gamma, T),cal_esr(
    stra_NTR_the,returns,cost,utility_gamma, T),cal_esr(_,returns,cost
    ,utility_gamma, T),CI
```

Code Listing A.11: Compare final ESR

A.2.2 Neural Networks

```
class NoTradeRegionRNN(nn.Module):

    def __init__(self, input_size, hidden_size, batch_size,dim_size):
        """Initialize params."""
        super(NoTradeRegionRNN, self).__init__()
        # read input parameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.dim_size = dim_size

        # Define parameters of double relu function for activation
    function
        self.input_param = nn.Linear(input_size, dim_size,
    hidden_size).to(device)
        self.hidden_param = nn.Linear(hidden_size,  dim_size,
    hidden_size).to(device)
        self.fc1_param = nn.Linear(hidden_size, dim_size,hidden_size)
    .to(device)
        self.fc2_param = nn.Linear(hidden_size, dim_size, hidden_size
    ).to(device)
        self.rotate_param = nn.Linear(dim_size,dim_size,bias=False)

    # Forward function allows a form:
    # h_t = w_fc2*relu(w_fc1*relu(w_inp*x_t+b_inp+w_h*h_{t-1}+b_h)+
    b_fc1)+b_fc2+b_fc1-b_h1
```

```
21     def forward(self, input, target, returns_partition, hidden):
22         def myrotate(input_rotate):
23             input_new = input_rotate.squeeze(1)
24             out = torch.matmul(self.rotate_param.weight,input_new)
25             return out.unsqueeze(1)
26 
27         # a function that creates the corner of the rotated no trade
    region
28         def corner():
29             mat2 = np.ones([2,4])
30             mat2[:,0] = np.array([-1,0])
31             mat2[:,1] = np.array([0,-1])
32             mat2[:,2] = np.array([1,0])
33             mat2[:,3] = np.array([0,1])
34             index_matrix2 = torch.tensor(mat2,dtype = torch.float).to
    (device)
35             res = torch.matmul(self.rotate_param.weight,index_matrix2
    ).T+target
36             return res
37 
38         # Create the corner and related slope of different lines
39         Corner = corner()
40         ac = (self.rotate_param.weight[1,0]+self.rotate_param.weight
    [1,1])\
41             /(self.rotate_param.weight[0,0]+self.rotate_param.weight
    [0,1])
42 
43         bd = (self.rotate_param.weight[0,0]-self.rotate_param.weight
    [0,1])\
44             /(self.rotate_param.weight[1,0]-self.rotate_param.weight
    [1,1])
45 
46         # Create the lower and upper bounds of each assets
47         def lower_bound_x(x):
48             ingate = (x.squeeze(1)[1]-Corner[0,1]*(bd>=0)-Corner
```

```
[1,1]*(bd<0))*bd

            ingate2 = -F.relu(ingate)+torch.abs(Corner[1,0]-Corner
[0,0])

            res = -F.relu(ingate2)+Corner[1,0]*(bd>=0)+Corner[0,0]*(
bd<0)
            return res

        def upper_bound_x(x):
            ingate = (x.squeeze(1)[1]-Corner[3,1]*(bd>=0)-Corner
[2,1]*(bd<0))*bd

            ingate2 = -F.relu(ingate)+torch.abs(Corner[2,0]-Corner
[3,0])

            res = -F.relu(ingate2)+Corner[2,0]*(bd>=0)+Corner[3,0]*(
bd<0)
            return res

        def lower_bound_y(x):
            ingate = (x.squeeze(1)[0]-Corner[0,0]*(ac>=0)-Corner
[3,0]*(ac<0))*ac

            ingate2 = -F.relu(ingate)+torch.abs(Corner[0,1]-Corner
[3,1])

            res = -F.relu(ingate2)+Corner[3,1]*(ac>=0)+Corner[0,1]*(
ac<0)
            return res

        def upper_bound_y(x):
            ingate = (x.squeeze(1)[0]-Corner[1,0]*(ac>=0)-Corner
[2,0]*(ac<0))*ac

```

```
74            ingate2 = -F.relu(ingate)+torch.abs(Corner[1,1]-Corner
    [2,1])
75
76            res = -F.relu(ingate2)+Corner[2,1]*(ac>=0)+Corner[1,1]*(
    ac<0)
77            return res
78
79        def lower_bound(x):
80            return torch.stack((lower_bound_x(x),lower_bound_y(x))).
    view(self.dim_size,self.hidden_size,self.batch_size)
81
82        def upper_bound(x):
83            return torch.stack((upper_bound_x(x),upper_bound_y(x))).
    view(self.dim_size,self.hidden_size,self.batch_size)
84
85        def recurrence(input, hidden):
86            # w_inp*x_t+b_inp+w_h*h_{t-1}+b_h
87            ingate = self.input_param.weight.view(self.dim_size,self.
    input_size,self.hidden_size)*input \
88                    + self.hidden_param.weight.view(self.dim_size,
    self.hidden_size,self.hidden_size)*hidden - lower_bound(hidden)
89            # w_fc1*relu(ingate)+upper-lower
90            ingate2 = self.fc1_param.weight.view(self.dim_size,self.
    hidden_size,self.hidden_size)*F.relu(ingate)\
91                    + upper_bound(hidden) - lower_bound(hidden)
92            # w_fc2*relu(ingate2)+upper
93            h       = self.fc2_param.weight.view(self.dim_size,self.
    hidden_size,self.hidden_size)*F.relu(ingate2) + upper_bound(hidden
    )
94            return h
95
96        output = []
97        steps = range(input.size(1))
98        for i in steps:
99            if i ==0:
```

```
100                 hidden = input[:,0,:].view(self.dim_size,1,self.
    batch_size).to(device)
101                 #hidden = (torch.tensor(Markowitz_opt,dtype=torch.
    float).view(self.dim_size,1,1)*torch.ones((self.dim_size,1,self.
    batch_size),dtype=torch.float)).to(device)
102             else:
103                 # pi_t = myrotate(pi_{t-1}*(1+r_t)/(1+sum(pi_{t-1}*
    r_t))) due to change of price after rebalance
104                 adjust_pi = hidden.view(self.dim_size,1,self.
    batch_size)*(1+returns_partition[:,i-1,:].view(self.dim_size,1,
    self.batch_size))\
105                                     /(1+torch.sum(hidden.view(self.
    dim_size,1,self.batch_size)*returns_partition[:,i-1,:].view(self.
    dim_size,1,\
106                                     self.batch_size),0))
107
108
109
110                 hidden = recurrence(input[:,i,:].view(self.dim_size,
    self.input_size,self.batch_size), adjust_pi)
111
112             output.append(hidden)
113
114         output = torch.cat(output, 1)
115
116         return output, hidden
```

Code Listing A.12: NN layer of no-trade region

```
1 class WealthRNN(nn.Module):
2     def __init__(self, input_size, hidden_size, n_layers, batch_size,
    seq_length, dim_size):
3         super(WealthRNN, self).__init__()
4         self.input_size = input_size
5         self.hidden_size = hidden_size
6         self.n_layers = n_layers
```

```
7         self.batch_size = batch_size
8         self.seq_length = seq_length
9         self.dim_size = dim_size
10        # the rnn layer which works as out, hidden_t = f(out_(t),
    hidden_(t-1)), used to approximate pi^*_(t)= f(pi^*_(t-1),pi_t)
11        self.rnn = NoTradeRegionRNN2(input_size, hidden_size,
    batch_size, dim_size).to(device)
12        self.out = nn.Linear(dim_size, hidden_size,bias=False).to(
    device)
13        # initialize some bias and weight
14        self.rnn.fc1_param.weight = torch.nn.Parameter(-1*torch.
    ones_like(self.rnn.fc1_param.weight))
15        self.rnn.fc2_param.weight = torch.nn.Parameter(-1*torch.
    ones_like(self.rnn.fc2_param.weight))
16        self.rnn.input_param.weight = torch.nn.Parameter(torch.
    zeros_like(self.rnn.input_param.weight))
17        self.rnn.hidden_param.weight = torch.nn.Parameter(torch.
    ones_like(self.rnn.hidden_param.weight))
18        self.out.weight = torch.nn.Parameter(*torch.ones_like(self.
    out.weight))
19
20    def update_bias(self,value):
21        self.rnn.hidden_param.bias = torch.nn.Parameter(value)
22        self.rnn.fc1_param.bias = torch.nn.Parameter(2*value)
23        self.rnn.fc1_param.bias.requires_grad = False
24
25    def update_weight(self,value):
26
27        self.rnn.rotate_param.weight = torch.nn.Parameter(value)
28
29        self.rnn.input_param.weight.requires_grad = False
30        self.rnn.hidden_param.weight.requires_grad = False
31        self.rnn.fc1_param.weight.requires_grad = False
32        self.rnn.fc2_param.weight.requires_grad = False
33        self.out.weight.requires_grad = False
```

```
34
35
36      def step(self, input,target, returns_partition, cost_partition,
    hidden=None):
37          output, hidden = self.rnn(input, target, returns_partition,
    hidden).to(device)
38          output2 = self.out.weight.view(self.dim_size,self.hidden_size
    ,self.hidden_size)*output
39          return output, output2
40
41      def forward(self, inputs, target, returns_partition,
    cost_partition,hidden=None):
42          hidden = self.__init__hidden().to(device)
43          output, hidden = self.rnn(inputs.float(),target,
    returns_partition, hidden.float())
44          # output2 the overall wealth at time T
45          output2 = torch.prod(FTA.cal_return(output,returns_partition,
    cost_partition).to(device)+1,0)
46          return  output, output2
47          #return  output
48
49      def __init__hidden(self):
50          hidden = 0.0*torch.ones(self.dim_size, self.hidden_size,
    self.batch_size,dtype=torch.float64).to(device)
51          return hidden
```

Code Listing A.13: Nonlinear layer of wealth process

A.3 Multiple Assets

A.3.1 Functions

```
1 # Create a positive definite symetric matrix close to the input
    matrix
2 def return_matrix_modification(matrix, n):
```

```
    try:
        # Attempt to perform a Cholesky decomposition
        np.linalg.cholesky(matrix)
        # If successful, return the matrix itself
        return matrix
    except np.linalg.LinAlgError:
        # If the decomposition fails, return the specified alternative
        return (matrix*0.995+np.identity(n)*0.005)
```

Code Listing A.14: Positive definite matrix approximation

A.3.2 Neural Networks

```
# Customize a RNN layer with double relu for multiple assets
# considering returns data to build a changed strategy weight according to price change
# v normalized
class NoTradeRegionRNN(nn.Module):

    def __init__(self, input_size, hidden_size, batch_size,dim_size):
        """Initialize params."""
        super(NoTradeRegionRNN, self).__init__()
        # read input parameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.dim_size = dim_size
        self.edge_coef = nn.Linear(dim_size,dim_size).to(device)

    # Forward function allows a form:
    # h_t = w_fc2*relu(w_fc1*relu(w_inp*x_t+b_inp+w_h*h_{t-1}+b_h)+b_fc1)+b_fc2+b_fc1-b_h1
    def forward(self, input, target, returns_partition, hidden):
      # create the pi_bar(merton optimal) and identity matrix
```

```
20      pi_bar = (torch.tensor(target,dtype=torch.float).to(device).
    view(self.dim_size,self.hidden_size,self.hidden_size)*\
21                  torch.ones((self.dim_size,self.hidden_size,self.
    batch_size),dtype=torch.float).to(device)).squeeze(1)
22      e_matrix = torch.eye(self.dim_size).to(device)
23
24      def recurrence(input, hidden):
25        #creating scalars, empty vectors and normalized v
26        eps = 1e-5
27        hidden = hidden.squeeze(1)
28        hidden_temp = hidden
29        judge_mat = torch.zeros([self.dim_size,self.batch_size]).to(
    device)
30        v = torch.nn.functional.normalize(self.edge_coef.weight, p
    =2.0, dim=1, eps=1e-12, out = None)
31
32        # loop once to find the fitness of each asset
33        for j in range(self.dim_size):
34          # create v for each asset
35          vj = torch.nn.functional.normalize(self.edge_coef.weight, p
    =2.0, dim=1, eps=1e-12, out = None)[j,:]
36          # calculate lambda for all assets
37          lambda_pi_plus = (torch.abs(self.edge_coef.bias[j])*torch.
    ones(self.batch_size).to(device) - torch.matmul(vj,hidden-pi_bar))
    /(torch.matmul(vj,e_matrix[j,:]))
38          lambda_pi_minus = (-torch.abs(self.edge_coef.bias[j])*torch
    .ones(self.batch_size).to(device) - torch.matmul(vj,hidden-pi_bar)
    )/(torch.matmul(vj,e_matrix[j,:]))
39          hidden_new = hidden + (lambda_pi_plus.view(self.batch_size
    ,1)*e_matrix[j,:]).T*(torch.matmul(vj,hidden-pi_bar)>torch.abs(
    self.edge_coef.bias[j]))+\
40                    (lambda_pi_minus.view(self.batch_size,1)*e_matrix[j
    ,:]).T*(torch.matmul(vj,hidden-pi_bar)<-torch.abs(self.edge_coef.
    bias[j]))
41          # create a matrix recording the fitness of such asset
```

```
42          judge = (torch.matmul(v,hidden_new-pi_bar)<torch.abs(self.
    edge_coef.bias.view(self.dim_size,1))+eps) & (torch.matmul(v,
    hidden_new-pi_bar)>-torch.abs(self.edge_coef.bias.view(self.
    dim_size,1))-eps)
43
44          judge = torch.min(judge,0).values
45          judge_mat[j,:] = judge
46      # create a matrix recording the assets which project to
    notrade region with only one projection
47      judge_mat = torch.max(judge_mat,0).values
48      del hidden_new
49      torch.cuda.empty_cache()
50
51      for j in range(self.dim_size):
52          # create v for each asset
53          vj = torch.nn.functional.normalize(self.edge_coef.weight, p
    =2.0, dim=1, eps=1e-12, out = None)[j,:]
54          # calculate lambda for all assets
55          lambda_pi_plus = (torch.abs(self.edge_coef.bias[j])*torch.
    ones(self.batch_size).to(device) - torch.matmul(vj,hidden_temp-
    pi_bar))/(torch.matmul(vj,e_matrix[j,:]))
56          lambda_pi_minus = (-torch.abs(self.edge_coef.bias[j])*torch
    .ones(self.batch_size).to(device) - torch.matmul(vj,hidden_temp-
    pi_bar))/(torch.matmul(vj,e_matrix[j,:]))
57          # one step projection of each asset
58          hidden_temp = hidden_temp + (lambda_pi_plus.view(self.
    batch_size,1)*e_matrix[j,:]).T*(torch.matmul(vj,hidden_temp-pi_bar
    )>torch.abs(self.edge_coef.bias[j]))+\
59                  (lambda_pi_minus.view(self.batch_size,1)*e_matrix[j
    ,:]).T*(torch.matmul(vj,hidden_temp-pi_bar)<-torch.abs(self.
    edge_coef.bias[j]))
60
61      del lambda_pi_plus
62      del lambda_pi_minus
63      torch.cuda.empty_cache()
```

```
64        # start a bisection method to find the exact boundary of
  assets without one fitness projection
65        h_in =  pi_bar
66        h_out = (1-judge_mat)*hidden
67        for i in range(10):
68          h_m = h_in+(-h_in+h_out)/2
69          judge = (torch.matmul(v,h_m-pi_bar)<=torch.abs(self.
  edge_coef.bias.view(self.dim_size,1))+eps) & (torch.matmul(v,h_m-
  pi_bar)>=-torch.abs(self.edge_coef.bias.view(self.dim_size,1))-eps
  )
70          judge = torch.min(judge,0).values
71          h_out = (~judge)*h_m+(judge)*h_out
72          h_in = (judge)*h_m+(~judge)*h_in
73        hidden = (judge_mat*hidden_temp+(1-judge_mat)*h_m).unsqueeze
  (1)
74        return hidden
75
76     output = []
77     steps = range(input.size(1))
78     #myret = returns
79     for i in steps:
80         if i ==0:
81             hidden = input[:,0,:].view(self.dim_size,1,self.
  batch_size).to(device)
82             #hidden = (torch.tensor(Markowitz_opt,dtype=torch.float
  ).view(self.dim_size,1,1)*torch.ones((self.dim_size,1,self.
  batch_size),dtype=torch.float)).to(device)
83         else:
84             # pi_t = myrotate(pi_{t-1}*(1+r_t)/(1+sum(pi_{t-1}*r_t)
  )) due to change of price after rebalance
85             adjust_pi = hidden.view(self.dim_size,1,self.batch_size
  )*(1+returns_partition[:,i-1,:].view(self.dim_size,1,self.
  batch_size))\
86                                     /(1+torch.sum(hidden.view(self.
  dim_size,1,self.batch_size)*returns_partition[:,i-1,:].view(self.
```

```
    dim_size,1,\
87                                      self.batch_size),0))
88
89              hidden = recurrence(input[:,i,:].view(self.dim_size,
    self.input_size,self.batch_size),  adjust_pi)
90          output.append(hidden)
91       output = torch.cat(output, 1)
92       return output, hidden
```

Code Listing A.15: NN layer of no-trade region (normalized v)

```
1  # Customize a RNN layer with double relu for multiple assets
2  # considering returns data to build a changed strategy weight
       according to price change
3  # v not normalized
4  class NoTradeRegionRNN2(nn.Module):
5
6      def __init__(self, input_size, hidden_size, batch_size,dim_size):
7          """Initialize params."""
8          super(NoTradeRegionRNN2, self).__init__()
9          # read input parameters
10         self.input_size = input_size
11         self.hidden_size = hidden_size
12         self.batch_size = batch_size
13         self.dim_size = dim_size
14         self.edge_coef = nn.Linear(dim_size,dim_size).to(device)
15
16     # Forward function allows a form:
17     # h_t = w_fc2*relu(w_fc1*relu(w_inp*x_t+b_inp+w_h*h_{t-1}+b_h)+
    b_fc1)+b_fc2+b_fc1-b_h1
18     def forward(self, input, target, returns_partition, hidden):
19       # create the pi_bar(merton optimal) and identity matrix
20       pi_bar = (torch.tensor(target,dtype=torch.float).to(device).
    view(self.dim_size,self.hidden_size,self.hidden_size)*\
21                 torch.ones((self.dim_size,self.hidden_size,self.
    batch_size),dtype=torch.float).to(device)).squeeze(1)
```

```
22      e_matrix = torch.eye(self.dim_size).to(device)
23
24      def recurrence(input, hidden):
25        #creating scalars, empty vectors and normalized v
26        eps = 1e-5
27        hidden = hidden.squeeze(1)
28        hidden_temp = hidden
29        judge_mat = torch.zeros([self.dim_size,self.batch_size]).to(
    device)
30        v = self.edge_coef.weight
31
32        # loop once to find the fitness of each asset
33        for j in range(self.dim_size):
34          # create v for each asset
35          vj = self.edge_coef.weight[j,:]
36          # calculate lambda for all assets
37          lambda_pi_plus = (torch.abs(self.edge_coef.bias[j])*torch.
    ones(self.batch_size).to(device) - torch.matmul(vj,hidden-pi_bar))
    /(torch.matmul(vj,e_matrix[j,:]))
38          lambda_pi_minus = (-torch.abs(self.edge_coef.bias[j])*torch
    .ones(self.batch_size).to(device) - torch.matmul(vj,hidden-pi_bar)
    )/(torch.matmul(vj,e_matrix[j,:]))
39          hidden_new = hidden + (lambda_pi_plus.view(self.batch_size
    ,1)*e_matrix[j,:]).T*(torch.matmul(vj,hidden-pi_bar)>torch.abs(
    self.edge_coef.bias[j]))+\
40                  (lambda_pi_minus.view(self.batch_size,1)*e_matrix[j
    ,:]).T*(torch.matmul(vj,hidden-pi_bar)<-torch.abs(self.edge_coef.
    bias[j]))
41          # create a matrix recording the fitness of such asset
42          judge = (torch.matmul(v,hidden_new-pi_bar)<torch.abs(self.
    edge_coef.bias.view(self.dim_size,1))+eps) & (torch.matmul(v,
    hidden_new-pi_bar)>-torch.abs(self.edge_coef.bias.view(self.
    dim_size,1))-eps)
43
44          judge = torch.min(judge,0).values
```

```
45          judge_mat[j,:] = judge
46        # create a matrix recording the assets which project to
    notrade region with only one projection
47        judge_mat = torch.max(judge_mat,0).values
48        del hidden_new
49        torch.cuda.empty_cache()
50
51        for j in range(self.dim_size):
52          # create v for each asset
53          vj = self.edge_coef.weight[j,:]
54          # calculate lambda for all assets
55          lambda_pi_plus = (torch.abs(self.edge_coef.bias[j])*torch.
    ones(self.batch_size).to(device) - torch.matmul(vj,hidden_temp-
    pi_bar))/(torch.matmul(vj,e_matrix[j,:]))
56          lambda_pi_minus = (-torch.abs(self.edge_coef.bias[j])*torch
    .ones(self.batch_size).to(device) - torch.matmul(vj,hidden_temp-
    pi_bar))/(torch.matmul(vj,e_matrix[j,:]))
57          # one step projection of each asset
58          hidden_temp = hidden_temp + (lambda_pi_plus.view(self.
    batch_size,1)*e_matrix[j,:]).T*(torch.matmul(vj,hidden_temp-pi_bar
    )>torch.abs(self.edge_coef.bias[j]))+\
59                  (lambda_pi_minus.view(self.batch_size,1)*e_matrix[j
    ,:]).T*(torch.matmul(vj,hidden_temp-pi_bar)<-torch.abs(self.
    edge_coef.bias[j]))
60
61        del lambda_pi_plus
62        del lambda_pi_minus
63        torch.cuda.empty_cache()
64        # start a bisection method to find the exact boundary of
    assets without one fitness projection
65        h_in =  pi_bar
66        h_out = (1-judge_mat)*hidden
67        for i in range(10):
68          h_m = h_in+(-h_in+h_out)/2
69          judge = (torch.matmul(v,h_m-pi_bar)<=torch.abs(self.
```

```
edge_coef.bias.view(self.dim_size,1))+eps) & (torch.matmul(v,h_m-
pi_bar)>=-torch.abs(self.edge_coef.bias.view(self.dim_size,1))-eps
)
70          judge = torch.min(judge,0).values
71          h_out = (~judge)*h_m+(judge)*h_out
72          h_in = (judge)*h_m+(~judge)*h_in
73        hidden = (judge_mat*hidden_temp+(1-judge_mat)*h_m).unsqueeze
(1)
74        return hidden
75
76     output = []
77     steps = range(input.size(1))
78     #myret = returns
79     for i in steps:
80         if i ==0:
81             hidden = input[:,0,:].view(self.dim_size,1,self.
batch_size).to(device)
82             #hidden = (torch.tensor(Markowitz_opt,dtype=torch.float
).view(self.dim_size,1,1)*torch.ones((self.dim_size,1,self.
batch_size),dtype=torch.float)).to(device)
83         else:
84             # pi_t = myrotate(pi_{t-1}*(1+r_t)/(1+sum(pi_{t-1}*r_t)
)) due to change of price after rebalance
85             adjust_pi = hidden.view(self.dim_size,1,self.batch_size
)*(1+returns_partition[:,i-1,:].view(self.dim_size,1,self.
batch_size))\
86                                     /(1+torch.sum(hidden.view(self.
dim_size,1,self.batch_size)*returns_partition[:,i-1,:].view(self.
dim_size,1,\
87                                     self.batch_size),0))
88
89             hidden = recurrence(input[:,i,:].view(self.dim_size,
self.input_size,self.batch_size),  adjust_pi)
90
91         output.append(hidden)
```

```
92        output = torch.cat(output, 1)
93
94        return output, hidden
```

Code Listing A.16: NN layer of no-trade region (unnormalized v)

```
1  class NoTradeRegionRNN_Ellipse(nn.Module):
2      def __init__(self, input_size, hidden_size, batch_size, dim_size)
       :
3          """Initialize params."""
4          super(NoTradeRegionRNN_Ellipse, self).__init__()
5          # Read input parameters
6          self.input_size = input_size
7          self.hidden_size = hidden_size
8          self.batch_size = batch_size
9          self.dim_size = dim_size
10
11         self.k = int(self.dim_size * (self.dim_size - 1) / 2)
12         self.num_elements = int(self.dim_size * (self.dim_size + 1) /
       2)
13         self.upper_triangular_elements = nn.Parameter(torch.randn(
       self.num_elements))
14
15     def rotate_matrix(self):
16         skew_matrix = torch.zeros(self.dim_size, self.dim_size, dtype
       =torch.float32).to(device)
17         triu_indices = torch.triu_indices(self.dim_size, self.
       dim_size, offset=0).to(device)
18         skew_matrix[triu_indices[0], triu_indices[1]] = self.
       upper_triangular_elements
19         rotate = skew_matrix.T @ skew_matrix
20         return rotate
21
22     def project_to_ellipse(self, x, c, R):
23         # Direction vector from center to point
24         direction = x - c  # Shape: (n,)
```

```
25        # The scalar to scale the direction vector such that the
    point lies on the ellipse boundary
26        dad = direction.T @ R @ direction
27        # Compute the scalar t
28        t = 1.0 / torch.sqrt(dad)
29        # New point on the ellipse boundary
30        new_point = c + t * direction  # Shape: (n,)
31        return new_point
32
33     def forward(self, input, target, returns_partition, hidden):
34        # Create the pi_bar (Merton optimal) and identity matrix
35        pi_bar = (torch.tensor(target, dtype=torch.float).to(device).
    view(self.dim_size, self.hidden_size, self.hidden_size) *
36                  torch.ones((self.dim_size, self.hidden_size, self.
    batch_size), dtype=torch.float).to(device)).squeeze(1)
37        rotated_Q = self.rotate_matrix()
38
39        def recurrence(input, hidden):
40            # Creating scalars, empty vectors and normalized v
41            eps = 1e-5
42            hidden = hidden.squeeze(1)
43            # (x-c)
44            diff = (hidden - pi_bar).T.unsqueeze(2)
45            # x is inside or outside the ellipse
46            value_ellipse = (diff.transpose(1, 2) @ rotated_Q @ diff)
    .squeeze(2)
47            judge = value_ellipse > 1.0
48
49            # If the point is outside the ellipse, project it to the
    ellipse boundary
50            new_hidden = hidden.clone()
51            for i in range(self.batch_size):
52                if judge[i]:
53                    new_hidden[:, i] = self.project_to_ellipse(hidden
    [:, i], pi_bar[:, i], rotated_Q)
```

```
54
55              return new_hidden.unsqueeze(1)
56
57          output = []
58          steps = range(input.size(1))
59          for i in steps:
60              if i == 0:
61                  hidden = input[:, 0, :].view(self.dim_size, 1, self.
    batch_size).to(device)
62              else:
63                  adjust_pi = hidden.view(self.dim_size, 1, self.
    batch_size) * (1 + returns_partition[:, i - 1, :].view(self.
    dim_size, 1, self.batch_size)) \
64                              / (1 + torch.sum(hidden.view(self.
    dim_size, 1, self.batch_size) * returns_partition[:, i - 1, :].
    view(self.dim_size, 1, self.batch_size), 0))
65
66                  hidden = recurrence(input[:, i, :].view(self.dim_size
    , self.input_size, self.batch_size), adjust_pi)
67
68              output.append(hidden)
69          output = torch.cat(output, 1)
70
71          return output, hidden
```

Code Listing A.17: NN layer of no-trade region (ellipsoid)

```
1 # wealth process
2 class WealthRNN(nn.Module):
3     def __init__(self, input_size, hidden_size, n_layers, batch_size,
      seq_length, dim_size):
4         super(WealthRNN, self).__init__()
5         self.input_size = input_size
6         self.hidden_size = hidden_size
7         self.n_layers = n_layers
8         self.batch_size = batch_size
```

```
9         self.seq_length = seq_length
10        self.dim_size = dim_size
11        # the rnn layer which works as out, hidden_t = f(out_(t),
    hidden_(t-1)), used to approximate pi^*_(t)= f(pi^*_(t-1),pi_t)
12        self.rnn = NoTradeRegionRNN(input_size, hidden_size,
    batch_size, dim_size).to(device)
13        self.out = nn.Linear(dim_size, hidden_size,bias=False).to(
    device)
14        # initialize some bias and weight
15        self.rnn.edge_coef.weight = torch.nn.Parameter(torch.eye(self
    .dim_size).to(device))
16        #self.rnn.edge_coef.weight = torch.nn.functional.normalize(
    self.rnn.edge_coef.weight, p=2.0, dim=1, eps=1e-12, out = None)
17        self.out.weight = torch.nn.Parameter(torch.ones_like(self.out
    .weight))
18        self.out.weight.requires_grad = False
19
20    def update_bias(self,value):
21        self.rnn.edge_coef.bias = torch.nn.Parameter(value)
22
23    def step(self, input, target, returns_partition, cost_partition,
    hidden=None):
24        output, hidden = self.rnn(input, target, returns_partition,
    hidden).to(device)
25        output2 = self.out.weight.view(self.dim_size,self.hidden_size
    ,self.hidden_size)*output
26        return output, output2
27
28    def forward(self, inputs, target, returns_partition,
    cost_partition, hidden=None):
29        hidden = inputs[:,0,:].to(device)
30        output, hidden = self.rnn(inputs.float(), target,
    returns_partition, hidden.float())
31        # output2 the overall wealth at time T
32        output2 = torch.prod(FMA.cal_return(output,returns_partition,
```

```
    cost_partition)+1,0)
33          return  output, output2
```

Code Listing A.18: Nonlinear layer of wealth process

```
1  # wealth function for third ellipse method
2  class WealthRNN_Ellipse(nn.Module):
3      def __init__(self, input_size, hidden_size, n_layers, batch_size,
       seq_length, dim_size):
4          super(WealthRNN_Ellipse, self).__init__()
5          self.input_size = input_size
6          self.hidden_size = hidden_size
7          self.n_layers = n_layers
8          self.batch_size = batch_size
9          self.seq_length = seq_length
10         self.dim_size = dim_size
11         # the rnn layer which works as out, hidden_t = f(out_(t),
       hidden_(t-1)), used to approximate pi^*_(t)= f(pi^*_(t-1),pi_t)
12         self.rnn = NoTradeRegionRNN_Ellipse(input_size, hidden_size,
       batch_size, dim_size).to(device)
13         self.out = nn.Linear(dim_size, hidden_size,bias=False).to(
       device)
14         self.out.weight = torch.nn.Parameter(torch.ones_like(self.out
       .weight)).to(device)
15         self.out.weight.requires_grad = False
16
17     def step(self, input, target, returns_partition, cost_partition,
       hidden=None):
18         output, hidden = self.rnn(input, target, returns_partition,
       hidden).to(device)
19         output2 = self.out.weight.view(self.dim_size,self.hidden_size
       ,self.hidden_size)*output
20         return output, output2
21
22     def forward(self, inputs, target, returns_partition,
       cost_partition, hidden=None):
```

```
23          hidden = inputs[:,0,:].to(device)
24          output, hidden = self.rnn(inputs.float(), target,
    returns_partition, hidden.float())
25          # output2 the overall wealth at time T
26          output2 = torch.prod(FMA.cal_return(output,returns_partition,
    cost_partition)+1,0)
27          return  output, output2
```

Code Listing A.19: Nonlinear layer of wealth process (ellipsoid)

Bibliography

ALPS. Alps equal sector weight etf. https://www.alpsfunds.com/exchange-traded-funds/eql, 2023.

A. Altarovici, M. Reppen, and H. M. Soner. Optimal Consumption and Investment with Fixed and Proportional Transaction Costs. *SIAM Journal on Control and Optimization*, 52(1):282–307, 2016.

M. Bichuch. Asymptotic analysis for optimal investment in finite time with transaction costs. *SIAM Journal on Financial Mathematics*, 3(1):433–458, 2012.

M. Bichuch and P. Guasoni. Investing With Liquid and Illiquid Assets. *Mathematical Finance*, 28(1):119–152, 2018.

D. B. Brown and J. E. Smith. Dynamic Portfolio Optimization with Transaction Costs: Heuristics and Dual Bounds. *Management Science*, 57(10):1752–1770, oct 2011. ISSN 0025-1909.

A. Charteris and K. McCullough. Tracking error vs tracking difference: Does it matter? *Investment Analysts Journal*, 49(3):269–287, jul 2020.

X. Chen, M. Dai, W. Jiang, and C. Qin. Asymptotic analysis of long-term investment with two illiquid and correlated assets. *Mathematical Finance*, 32(4):1133–1169, 2022. ISSN 14679965.

K. Cho. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

J. Connor, R. Martin, and L. Atlas. Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254, 1994.

M. H. A. Davis and A. R. Norman. Portfolio Selection with Transaction Costs. *Mathematics of Operations Research*, 15(4):676–713, nov 1990. ISSN 0364-765X.

Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, 2017. ISSN 21622388.

I. Ekren and R. Liu. Optimal rebalancing frequencies for multidimensional portfolios. *Mathematics and Financial Economics*, 12(2):165–191, 2018. ISSN 1862-9660.

J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

A. Frino and D. Gallagher. Tracking s&p 500 index funds. *Journal of portfolio Management*, 28(1), 2001.

L. Gaegauf, S. Scheidegger, and F. Trojani. A Comprehensive Machine Learning Framework for Dynamic Portfolio Choice With Transaction Costs. *SSRN Electronic Journal*, 2023. ISSN 1556-5068.

S. Gerhold, J. Muhle-Karbe, and W. Schachermayer. The dual optimizer for the growth-optimal portfolio under transaction costs. *Finance and Stochastics*, 17(3): 325–354, 2013.

S. Gerhold, P. Guasoni, J. Muhle-Karbe, and W. Schachermayer. Transaction costs, trading volume, and the liquidity premium. *Finance and Stochastics*, 18(1):1–37, 2014. ISSN 09492984.

P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Stochastic Modelling and Applied Probability. Springer New York, 2013. ISBN 9780387216171.

G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

P. Guasoni and E. Mayerhofer. Leveraged funds: robust replication and performance evaluation. *Quantitative Finance*, 23(7-8):1155–1176, 2023. ISSN 14697696.

P. Guasoni and J. Muhle-Karbe. Long horizons, high risk aversion, and Endogenous Spreads. *Mathematical Finance*, 25(4):724–753, 2015. ISSN 14679965.

P. Guasoni and S. Robertson. Optimal importance sampling with explicit formulas in continuous time. *Finance & Stochastics*, 12(1), 2008.

J. M. Hammersley and K. W. Morton. A new monte carlo technique: antithetic variates. *Mathematical Proceedings of the Cambridge Philosophical Society*, 52(3): 449âĂŞ475, 1956.

J. Hasbrouck. Trading costs and returns for U.S. equities: Estimating effective costs from daily data. *Journal of Finance*, 64(3):1445–1477, 2009. ISSN 00221082.

S. Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

iShares. Global etf market facts: three things to know from q3 2023. https://www.ishares.com/us/insights/global-etf-facts, 2023.

K. Janeček and S. E. Shreve. Asymptotic analysis for optimal investment and consumption with transaction costs. *Finance and Stochastics*, 8(2):181–206, 2004.

N. Jegadeesh and S. Titman. Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency. *The Journal of Finance*, 48(1):65, 1993. ISSN 00221082.

R. Li. Transaction cost of portfolio. https://github.com/jsxzliran/TransactionCost, 2024.

X. Li and X. Wu. Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. In *2015 IEEE International*

Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 4520–4524, 2015.

H. Liu. Optimal Consumption and Investment with Transaction Costs and Multiple Risky Assets. *The Journal of Finance*, 59(1):289–338, feb 2004. ISSN 00221082.

H. Liu and M. Loewenstein. Optimal Portfolio Selection with Transaction Costs and Finite Horizons. *Review of Financial Studies*, 15(3):805–835, 2002. ISSN 08939454.

M. J. Magill and G. M. Constantinides. Portfolio selection with transactions costs. *Journal of Economic Theory*, 13(2):245–263, oct 1976. ISSN 00220531.

H. Markowitz. Portfolio Selection. *The Journal of Finance*, 7(1):77–91, 1952. ISSN 15406261.

R. C. Merton. Lifetime Portfolio Selection under Uncertainty: The Continuous-Time Case. *The Review of Economics and Statistics*, 51(3):247, 1969. ISSN 00346535.

R. C. Merton. Optimum Consumption and Portfolio Rules in a Continuous-Time Model. *Stochastic Optimization Models in Finance*, 3(December):621–661, 1971.

K. Muthuraman and S. Kumar. Multi-dimensional Portfolio Optimization with Proportional Transaction Costs. *SSRN Electronic Journal*, pages 285–287, 2004. ISSN 1556-5068.

R. Novy-Marx and M. Velikov. A Taxonomy of Anomalies and Their Trading Costs. *Review of Financial Studies*, 29(1):104–147, 2016. ISSN 0893-9454.

D. Possamaï, H. M. Soner, and N. Touzi. Homogenization and asymptotics for small transaction costs: the multidimensional case. *Communications in Partial Differential Equations*, 40(11):2005–2046, 2015.

R. Y. Rubinstein and R. Marcus. Efficiency of Multivariate Control Variates in Monte Carlo Simulation. *Operations Research*, 33(3):661–677, jun 1985. ISSN 0030-364X.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

S. E. Shreve and H. M. Soner. Optimal investment and consumption with transaction costs. *The Annals of Applied Probability*, pages 609–692, 1994.

G. Strang. *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, 2006. ISBN 9780534422004.

W. Sun, A. Fan, L.-W. Chen, T. Schouwenaars, and M. A. Albota. Optimal Rebalancing for Institutional Portfolios. *The Journal of Portfolio Management*, 32(2):33–43, jan 2006. ISSN 0095-4918.

S. T. Tokdar and R. E. Kass. Importance sampling: A review. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(1):54–60, 2010. ISSN 19395108.

M. Ugur Gudelek, S. Arda Boluk, and A. Murat Ozbayoglu. A deep learning based stock trading model with 2-D CNN trend detection. *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017 - Proceedings*, 2018-January:1–8, 2018.

W. Zhang and C. Zhou. Deep Learning Algorithm to solve Portfolio Management with Proportional Transaction Cost. *CIFEr 2019 - IEEE Conference on Computational Intelligence for Financial Engineering and Economics*, 2019.