ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΒΙΟΜΗΧΑΝΙΚΗΣ ΣΧΕΔΙΑΣΗΣ ΚΑΙ ΠΑΡΑΓΩΓΗΣ

http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
Θηβών 250, Αθήνα-Αιγάλεω 12241
Τηλ: +30 210 538-1614

Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών
Τεχνητή Νοημοσύνη και Βαθιά Μάθηση
https://aidl.uniwa.gr/

UNIVERSITY OF WEST ATTICA
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING
DEPARTMENT OF INDUSTRIAL DESIGN AND PRODUCTION ENGINEERING

http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
250, Thivon Str., Athens, GR-12241, Greece
Tel: +30 210 538-1614

Master of Science in
Artificial Intelligence and Deep Learning
https://aidl.uniwa.gr/

Master of Science Thesis

Applying Reinforcement Learning algorithms for profitable strategies in a stock market simulator

Student: Theodore Stavrothanasis
Registration Number: AIDL-0015

MSc Thesis Supervisor

Panagiotis Kasnesis
Lecturer

ATHENS-EGALEO, September 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΒΙΟΜΗΧΑΝΙΚΗΣ ΣΧΕΔΙΑΣΗΣ ΚΑΙ ΠΑΡΑΓΩΓΗΣ

http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
Θηβών 250, Αθήνα-Αιγάλεω 12241
Τηλ: +30 210 538-1614

UNIVERSITY OF WEST ATTICA
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING
DEPARTMENT OF INDUSTRIAL DESIGN AND PRODUCTION ENGINEERING

http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
250, Thivon Str., Athens, GR-12241, Greece
Tel: +30 210 538-1614

Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών
Τεχνητή Νοημοσύνη και Βαθιά Μάθηση
https://aidl.uniwa.gr/

Master of Science in
Artificial Intelligence and Deep Learning
https://aidl.uniwa.gr/

Μεταπτυχιακή Διπλωματική Εργασία

Εφαρμογή αλγορίθμων ενισχυτικής μάθησης στην εξομοίωση κερδοφόρων χρηματιστηριακών στρατηγικών

Φοιτητής: (Σταυροθανάσης Θεόδωρος)
AM: AIDL-0015

Επιβλέπων Καθηγητής

Παναγιώτης Κασνέσης
Λέκτορας

ΑΘΗΝΑ-ΑΙΓΑΛΕΩ, Σεπτέμβριος 2023

This MSc Thesis has been accepted, evaluated and graded by the following committee:

Supervisor	Member	Member
Panagiotis Kasnesis Digitally signed by Panagiotis Kasnesis Date: 2023.11.06 10:04:25 +02'00'	[illegible] Digitally signed by Charalampos Patrikakis Date: 2023.11.06 10:54:45 +02'00'	[illegible] Digitally signed by Georgios Priniotakis Date: 2023.11.06 11:14:36 +02'00'
Kasnesis Panagiotis	Patrikakis Charalampos	Priniotakis Georgios
Lecturer	Professor	Professor
Electrical & Electronics Engineering	Electrical & Electronics Engineering	Industrial Design & Production Engineering Department
University of West Attica	University of West Attica	University of West Attica

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΜΕΤΑΠΤΥΧΙΑΚΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος Σταυροθανάσης Θεόδωρος του Χριστοφόρου, με αριθμό μητρώου ADDL_0015 μεταπτυχιακός φοιτητής του ΔΠΜΣ «Τεχνητή Νοημοσύνη και Βαθιά Μάθηση» του Τμήματος Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών και του Τμήματος Μηχανικών Βιομηχανικής Σχεδίασης και Παραγωγής, της Σχολής Μηχανικών του Πανεπιστημίου Δυτικής Αττικής,

δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της μεταπτυχιακής διπλωματικής εργασίας και κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος. Η εργασία δεν έχει κατατεθεί στο πλαίσιο των απαιτήσεων για τη λήψη άλλου τίτλου σπουδών ή επαγγελματικής πιστοποίησης πλην του παρόντος.
Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του διπλώματός μου.»

Ο Δηλών
Σταυροθανάσης Θεόδωρος

[illegible]

(Υπογραφή φοιτητή)

Declaration of the author of this MSc thesis

I, Theodore Stavrothanasis (author name, including father's name) with the following student registration number: AIDL_0015, postgraduate student of the MSc program in "Artificial Intelligence and Deep Learning", which is organized by the Department of Electrical and Electronic Engineering and the Department of Industrial Design and Production Engineering of the Faculty of Engineering of the University of West Attica, hereby declare that:
I am the author of this MSc thesis and any help I may have received is clearly mentioned in the thesis. Additionally, all the sources I have used (e.g., to extract data, ideas, words or phrases) are cited with full reference to the corresponding authors, the publishing house or the journal; this also applies to the Internet sources that I have used. I also confirm that I have personally written this thesis and the intellectual property rights belong to myself and to the University of West Attica. This work has not been submitted for any other degree or professional qualification except as specified in it.
Any violations of my academic responsibilities, as stated above, constitutes substantial reason for the cancellation of the conferred MSc degree.

The author
Stavrothanasis Theodore

[illegible]

(Signature)

To my beloved parents Christophoros and Vassiliki who are no longer with me.

To the esteemed professors of this postgraduate program that contributed to my educational evolution and academic advancement.

Abstract

Financial time series present unique characteristics that an investor, analyst or algorithmic trader has to take always into account. One of the defining features is their inherent volatility and non-linearity. Unlike many other forms of data financial markets are influenced by economic indicators, geopolitical events and investor sentiment. These factors can cause sudden and unpredictable price movements resulting in extreme volatility and this violates the assumption of linearity, making traditional statistical methods less effective. They also exhibit auto-correlation where the value of a variable at one time point is correlated with its value at a previous point. This auto-correlation can persist over multiple time lags, leading to trends in the data. Identifying and modeling these trends is crucial for making informed investment decisions. They also suffer from "fat-tailed" distributions which mean there are frequent market crashes and price swings that could not be expected in a normal distribution. Financial time series are often non-stationary, with statistical properties like mean, standard deviation, skewness, kurtosis changing over different periods. Financial markets are not only influenced by macroeconomic factors but also by their own microstructure, which includes factors like bid-ask spreads, trading volumes, and market orders. Understanding and modeling market microstructure is crucial for accurately capturing its dynamics.

The main problem in financial markets is how to make profitable investment strategies with the lower risk that maximize returns. In this thesis we examine the use of Reinforcement Learning as a tool of decision making which can lead us to strategies with better performance than buy and hold the underlying asset.

We create ten out-of-sample synthetic time series based on standard normal distribution and simulate a trading game where we evaluate the effectiveness of major two RL algorithms Q-learning and REINFORCE.

Our trading simulations demonstrated that the performance of reinforcement learning algorithms, Q-learning and REINFORCE can be influenced by the stochastic nature of underlying data. REINFORCE showed an advantage in terms of P/L (profit or loss) for most seeds, while Q-learning displayed greater consistency in risk-adjusted returns. The unexpected success of a buy-and-hold strategy for specific seeds underscores the importance of considering diverse approaches in trading scenarios. These findings emphasize the dynamic nature of algorithmic trading, where the choice of the optimal strategy depends on the specific characteristics of the underlying data.

Finally the construction of a portfolio of the ten single equity curves showed acceptable performance while minimizing the risk. The results seemed quite promising.

Keywords

Betting system, policy based, Q-learning, decision making, trading game, random process modeling.

Περίληψη

Οι χρηματιστηριακές χρονοσειρές έχουν την ιδιεταιρότητα να παρουσιάζουν κάποια μοναδικά χαρακτηριστικά που πρέπει να λαμβάνει υπόψη ένας επενδυτής, αναλυτής ή αλγόριθμος επενδύσεων όπως η μεταβλητότητα και η μη γραμμικότητά τους. Σε αντίθεση με άλλες μορφές δεδομένων, οι χρηματιστηριακές αγορές επηρεάζονται από οικονομικούς δείκτες, γεωπολιτικά γεγονότα και το συναίσθημα των επενδυτών. Αυτοί οι παράγοντες μπορούν να προκαλέσουν ξαφνικές και απρόβλεπτες κινήσεις των τιμών με αποτέλεσμα την ακραία αστάθεια παραβιάζοντας την υπόθεση της γραμμικότητας και καθιστώντας λιγότερο αποτελεσματικές τις παραδοσιακές στατιστικές μεθόδους. Παρουσιάζουν επίσης αυτοσυσχέτιση όπου η τιμή μιας μεταβλητής σε ένα χρονικό σημείο συσχετίζεται με την τιμή της σε ένα προηγούμενο σημείο. Η αυτοσυσχέτιση αυτή μπορεί να παραμείνει για πολλά χρονικά σημεία, δημιουργώντας τάσεις στα δεδομένα. Ο εντοπισμός και η μοντελοποίηση αυτών των τάσεων είναι ζωτικής σημασίας για τη λήψη τεκμηριωμένων επενδυτικών αποφάσεων.

Οι κατανομές τους παρουσιάζουν το πρόβλημα της υπερυψωμένης ουράς λόγω των συχνών απότομων βυθισμάτων και διακυμάνσεων των αγορών κάτι που δεν βλέπουμε σε μια κανονική κατανομή. Τις περισσότερες φορές ειναι non-stationary, με στατιστικές ιδιότητες όπως η μέση τιμή, η τυπική απόκλιση, η λοξότητα και η κύρτωση να αλλάζουν σε διάφορες περιόδους. Ενα ακόμη χαρακτηριστικό τους αποτελεί η μικροδομή τους, η οποία περιλαμβάνει παράγοντες όπως τα spreads προσφοράς-ζήτησης αλλά και οι όγκοι των συναλλαγών. Η κατανόηση και η μοντελοποίηση αυτής της μικροδομής είναι ζωτικής σημασίας για την ακριβή αποτύπωση της δυναμικής της.

Το κύριο πρόβλημα στις χρηματιστηριακές αγορές είναι πώς να δημιουργηθούν κερδοφόρες επενδυτικές στρατηγικές με το χαμηλότερο ρίσκο και τη υψηλότερη απόδοση. Σε αυτή τη διατριβή εξετάζουμε τη χρήση της Ενισχυτικής Μάθησης ως εργαλείου λήψης αποφάσεων που μπορεί να μας οδηγήσει σε στρατηγικές υψηλότερης απόδοσης από ότι θα μας έδινε μια τυπική αγορά και διακράτηση του υποκείμενου εργαλείου.

Δημιουργούμε δέκα συνθετικές χρονοσειρές με βάση την τυπική κανονική κατανομή και προσομοιώνουμε ένα παιχνίδι συναλλαγών όπου αξιολογούμε την αποτελεσματικότητα των δύο κύριων αλγορίθμων RL Q-learning και REINFORCE.

Οι προσομοιώσεις συναλλαγών μας έδειξαν ότι η απόδοση των αλγορίθμων ενισχυτικής μάθησης Q-learning και REINFORCE μπορούν να επηρεαστεί από τη στοχαστική φύση των υποκείμενων δεδομένων. Ο REINFORCE έδειξε ότι πλεονεκτεί όσον αφορά το P/L (κέρδος ή ζημιά) για τις περισσότερες εξομοιώσεις, ο Q-learning εμφάνισε μεγαλύτερη ακρίβεια στις προσαρμοσμένες στον κίνδυνο αποδόσεις. Η απροσδόκητη επιτυχία της στρατηγικής “buy and hold” για συγκεκριμένες εξομοιώσεις υπογραμμίζει τη σημασία της εξέτασης διαφορετικών προσεγγίσεων. Αυτά τα ευρήματα δείχνουν τη δυναμική φύση των αλγοριθμικών συναλλαγών, όπου η επιλογή της βέλτιστης στρατηγικής εξαρτάται από τα ειδικά χαρακτηριστικά των υποκείμενων δεδομένων.

Τέλος, η κατασκευή ενός χαρτοφυλακίου συνδυάζοντας τις δέκα μεμονωμένες εξομοιώσεις έδειξε αποδεκτές επιδόσεις ελαχιστοποιώντας τον κίνδυνο. Τα αποτελέσματα εμφανίζονται αρκετά ικανοποιητικά.

Λέξεις – κλειδιά

Σύστημα στοιχηματισμού, policy based, Q-learning, παιχνίδι συναλλαγών, μοντελοποίηση στοχαστικής διαδιακασίας.

Table of Contents

List of Tables **12**

List of figures **12**

Acronym Index **15**

INTRODUCTION **16**
The subject of this thesis **16**
Aim and objectives **16**
Methodology **16**
Structure **17**

1 CHAPTER 1: Stochastic processes **18**
1.1 Random Walk **18**
1.2 Autocorrelation in Financial Time Series **19**
1.2.1 Autocorrelation Function (ACF) 19
1.2.2 Practical Significance 20
1.2.3 How to interpret it. 20

2 CHAPTER 2: Reinforcement Learning **22**
2.1 Components of the Reinforcement Learning Process **22**
2.2 Deterministic Policy **23**
2.3 Stochastic Policy **23**
2.4 Markov Decision Process (MDP) **24**
2.5 Policy and Value Functions **25**
2.6 Bellman Optimality Equations in Reinforcement Learning **25**
2.6.1 Optimal State-Value Function 25
2.6.2 Optimal Action-Value Function 26
2.6.3 Solving the Bellman Optimality Equations 26
2.6.4 Value Iteration Algorithm 26
2.7 Key Concepts and Algorithms in Reinforcement Learning **27**
2.8 Applications of Reinforcement Learning **27**
2.9 Temporal Difference Learning **28**
2.9.1 TD(0) Algorithm 28
2.9.2 Advantages of Temporal Difference Learning 29
2.10 Q-Learning **29**
2.10.1 Q-Learning Algorithm 30
2.10.2 ϵ-greedy Strategy 31
2.10.3 Some characteristics of off-policy algorithms: 31
2.11 Policy Gradients **32**
2.12 REINFORCE algorithm **33**
2.12.1 Trajectories in Reinforcement Learning 33
2.12.2 Trajectory Return 34
2.12.3 Expected return 35
2.12.4 Gradient ascent 35
2.12.5 Sampling and estimate 36
2.12.6 Gradient estimation formula 36

3 CHAPTER 3: Related Work **38**

4 CHAPTER 4: Trading game 41
4.1 Random Walk generation 41
4.2 Charts and definitions 42
4.3 Environment 43
4.3.1 RandomWalkEvn class 44
4.3.2 States 45
4.3.3 Reward function 47
4.4 Discretized Q-learning 48
4.4.1 Training the discreteQAgent. 49
4.4.2 Simulating a trading game with discreteQAgent. 50
4.5 Policy Gradient 50
4.5.1 Training with the policy gradient algorithm 51
4.6 Implementation and results 54
4.6.1 Sharpe ratio 55
4.6.2 Random process with Seed=19 57
4.6.3 Random process with Seed=20 58
4.6.4 Random process with Seed=21 59
4.6.5 Random process with Seed=22 60
4.6.6 Random process with Seed=23 61
4.6.7 Random process with Seed=24 62
4.6.8 Random process with Seed=25 63
4.6.9 Random process with Seed=26 64
4.6.10 Random process with Seed=28 65
4.6.11 Random process with Seed=29 66
4.7 Portfolio of seeds 67

5 Conclusions 69

6 Future Research 71

Bibliography – References – Online sources 73

Appendix A 75

Appendix B 75

List of Tables

Table 1 Parameters on discrete-Q train algorithm.....49

Table 2 Parameters on REINFORCE algorithm.....51

Table 3 Results of trading simulations on various random processes.....69

List of figures

Figure 1 Random Walk generated with seed=18.....18

Figure 2 Autocorrelation of Random Walk generated in Figure 1.....19

Figure 3 The interaction between an agent and its environment within a Markov Decision Process (MDP).....22

Figure 4 Example of a MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).....24

Figure 5 Q-learning algorithm.....30

Figure 6 Summary of approaches in Reinforcement Learning. The classification is based on whether we want to model the value or the policy.....32

Figure 7 Gradient ascent strategy......35

Figure 8 Random Walk generated with seed=18.....41

Figure 9 Bar transitions.....42

Figure 10 left: pc_{10} for RP with seed=18, middle: distribution, right: autocorrelation function.....46

Figure 11 left: pc_{20} for RP with seed=18, middle: distribution, right: autocorrelation function.....46

Figure 12 left: pc_{30} for RP with seed=18, middle: distribution, right: autocorrelation function.....47

Figure 13 left: pc_{50} for RP with seed=18, middle: distribution, right: autocorrelation function.....47

Figure 14 left: pc_{100} for RP with seed=18, middle: distribution, right: autocorrelation function.....47

Figure 15 Espison greedy options.....49

Figure 16 Running average of rewards for a total of 500 episodes.....50

Figure 17 Average reword during training the policy gradient algorithm 54
Figure 18 Equity Process generated with seed=19..57
Figure 19 Equity generated applying Q-learning on RP with seed=19....................................57
Figure 20 Equity generated applying REINFORCE on RP with seed=19...............................57
Figure 21 Random Process generated with seed=20...58
Figure 22 Returns generated applying Q-learning on RP with seed=20..................................58
Figure 23 Returns generated applying REINFORCE on RP with seed=20.............................58
Figure 24 Random Process generated with seed=21...59
Figure 25 Equity generated applying Q-learning on RP with seed=21....................................59
Figure 26 Equity generated applying REINFORCE on RP with seed=21...............................59
Figure 27 Random Process generated with seed=22...60
Figure 28 Equity generated applying Q-learning on RP with seed=22....................................60
Figure 29 Equity generated applying REINFORCE on RP with seed=22...............................60
Figure 30 Random Process generated with seed=23...61
Figure 31 Equity generated applying Q-learning on RP with seed=23....................................61
Figure 32 Equity generated applying Reinforce on RP with seed=23 61
Figure 33 Random Process generated with seed=24...62
Figure 34 Equity generated applying Q-learning on RP with seed=24....................................62
Figure 35 Equity generated applying REINFORCE on RP with seed=24...............................62
Figure 36 Random Process generated with seed=25...63
Figure 37 Equity generated applying Q-learning on RP with seed=25....................................63
Figure 38 Equity generated applying REINFORCE on RP with seed=25...............................63
Figure 39 Random Process generated with seed=26...64
Figure 40 Equity generated applying Q-learning on RP with seed=26....................................64
Figure 41 Equity generated applying REINFORCE on RP with seed=26...............................64
Figure 42 Random Process generated with seed=28...65
Figure 43 Equity generated applying Q-learning on RP with seed=28....................................65

Figure 44 Equity generated applying REINFORCE on RP with seed=28 65

Figure 45 Random Process generated with seed=29 66

Figure 46 Equity generated applying Q-learning on RP with seed=29 66

Figure 47 Equity generated applying REINFORCE on RP with seed=29 66

Figure 48 Portfolio of all seeds for Q-learning algorithm 67

Figure 49 Portfolio of all seeds for REINFORCE algorithm 67

Figure 50 Comparison of portfolios between the two algorithms 68

Acronym Index

ACF: Autocorrelation function

MDP: Markov Decision Process

TD: Temporal difference

DQN: Deep Q Networks

DDPG: Deep Deterministic Policy Gradient

TD: Temporal Difference

TD(0): Temporal Difference with a one-step look ahead

p/l, PnL: profit or loss

RWP: Random Walk Process

SP: Stochastic Process

RP: Random Process

SAC: Soft Actor-Critic

DRQN: deep recurrent Q-network

GRU: Gated Recurrent Unit

LSTM: Long Short-Term Memory

CNN: Convolutional Neural Network

MLP: Multi-layer perceptron

RNN: Recurrent Neural Network

DNN: Deep Neural Network

Adaptive DDPG: Adaptive Deep Deterministic Reinforcement Learning

A3C: Asynchronous Advantage Actor-Critic

SDAE: Stacked Denoising Autoencoder

PPO: Proximal Policy Optimization

PG: Policy Gradient

TFJ-DRL: Time-Driven Feature-Aware Jointly Deep Reinforcement Learning

INTRODUCTION

The evolving domain of artificial intelligence is offering solutions to intricate challenges across diverse fields. Within this landscape, we technically explore the subject of utilization of Reinforcement Learning as a tool in the field of algorithmic trading. In this thesis, we delve into the dynamics of a trading game where Reinforcement Learning can help to make informed decisions and forge profitable strategies. It is a simulated market governed by synthetic random processes, where the interplay of data, algorithms, and decision-making forms the core of our study.

The subject of this thesis

At the heart of this thesis lies the intimidating problem of financial time series the complexity and inherent chaotic behavior of these intricate data streams. In the world of trading, understanding these complex dynamics is a paramount challenge. Reinforcement Learning emerges as a potential solution, promising to unravel the intricacies of financial systems and, in turn, enable the creation of profitable trading strategies. The timeliness of this pursuit is evident in the ever-increasing need for innovative approaches to tackle the enigmatic behavior of financial markets.

Aim and objectives

The aim of this thesis is to synthesize a representative Reinforcement Learning environment that mimics the dynamics of a market. Within this dynamic environment, our primary objective is to harness the capabilities of Reinforcement Learning algorithms to craft trading strategies that outperform the rudimentary 'buy and hold' strategy. To dissect this aim further, we delineate our objectives:

1. Environment Construction: We endeavor to construct a trading environment that mirrors the complexities of financial time series
2. Algorithm Selection: Our focus turns to the selection and implementation of Reinforcement Learning algorithms, with a specific emphasis on Discretized Q-learning and Policy Gradient methods, which will steer our trading strategies.
3. Profitable Strategy Generation: Leveraging the power of Reinforcement Learning, we aim to create trading strategies that can consistently yield superior returns compared to the traditional 'buy and hold' approach.
4. Evaluation on Random Processes: Our research extends to the rigorous evaluation of these strategies across an array of out-of-sample random processes regarding effectiveness and profitability.

Methodology

We begin by constructing a simulated trading environment that mimics real-world market conditions, serving as the playground for our Reinforcement Learning agents.

Within this dynamic environment, we implement Reinforcement learning algorithms specifically, Discretized Q-learning and Policy Gradient methods. These algorithms will govern the decision-making abilities of our agents.

Then we employ these algorithms to create trading strategies. Notably, these strategies operate within the constraints of a fixed betting system, without the complexity of money or risk management, focusing solely on the direction of trades.

Lastly we rigorously evaluate and compare the performance of these strategies in a spectrum of out-of-sample random processes. This assessment offers a comprehensive view of their effectiveness and profitability.

Structure

The thesis unfolds across several chapters:

Chapter 1: Introduction to Stochastic Process creation and Autocorrelation: This chapter delves into the fundamentals of stochastic processes, exploring the practical significance of autocorrelation functions and their interpretation.

Chapter 2: Reinforcement Learning Fundamentals: Here, we delve into the technicalities of Reinforcement Learning, dissecting its structural components, deterministic and stochastic policies, Markov Decision Processes, policy and value functions, Bellman optimality equations, optimal state-value and action-value functions. We explore the algorithms at the core of our study, including Q-learning, Policy Gradients, and REINFORCE.

Chapter 3: A survey examining other research papers in the field of decision making in automated trading and related work.

Chapter 4: Implementation and Analysis: In this chapter we do our research. We construct the trading environment, delve into the generation of random processes, define state representations, and reward functions. The chapter completes with the application of Discretized Q-learning and Policy Gradient algorithms to diverse random processes, with a focus on comparing their performance.

Chapter 5: Presents the Primary discoveries and conclusions from the Trading Simulations Game. This chapter includes a comprehensive comparison of the best strategies, profitability metrics, and in-depth result assessments.

Chapter 6: Suggested some ideas for enhancing the trading simulation results, including optimizing reward functions, exploring novel metrics, implementing penalties for significant drawdowns, and refining state representation.

1 CHAPTER 1: Stochastic processes

1.1 Random Walk

Random walks are fundamental models used in diverse fields like time series analysis, finance, physics, and computer science. These models provide a straightforward yet powerful approach to simulate dynamic systems with random behavior. We use the concept of random walk generation, focusing primarily on the standard normal random walk formula:

$$x(t) = x(t-1) + \frac{N(0,1)}{100} \tag{1.1}$$

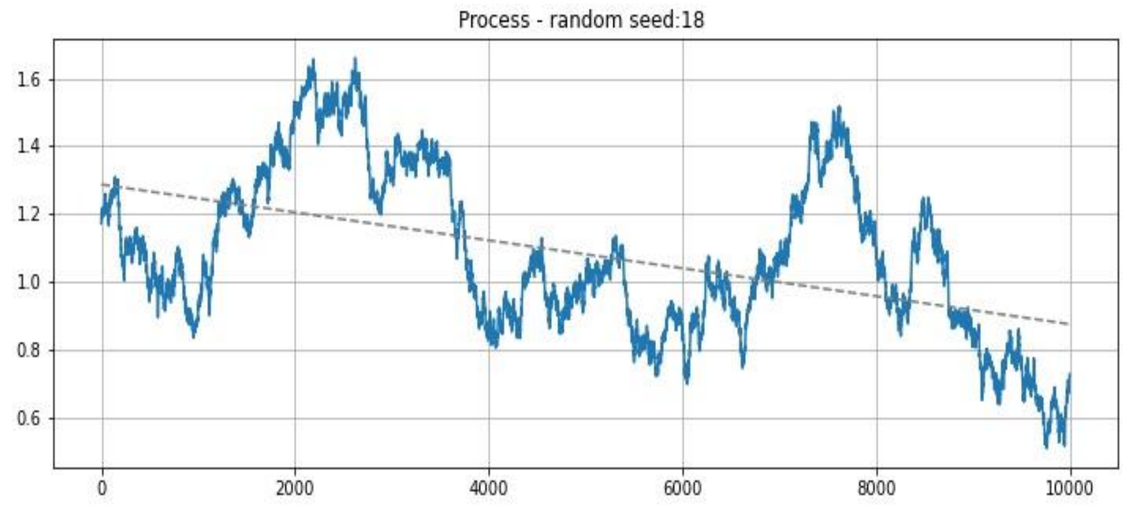

Figure 1 Random Walk generated with seed=18

The standard normal random walk involves generating a sequence of values over time, figure 1 shows a random process generated with formula (1.1) and seed=18. Each new value is obtained by adding a small random increment sampled from a standard normal distribution to the previous value. Here, x(t) represents the current value, x(t-1) is the previous value, and N(0,1) is a random variable drawn from a standard normal distribution with mean 0 and variance 1.

Random walks have found widespread applications, especially in time series analysis. In finance, they are employed to simulate stock prices and asset prices, allowing for effective risk analysis and option price estimation. In computer science, random walks are utilized in various algorithms, such as Monte Carlo simulations and search heuristics, to solve complex problems. Moreover, random walks are extensively used in physics to model particle movement, diffusion processes, and Brownian motion.

Although random walk models are versatile, they have limitations. One significant challenge is their inability to capture long-term dependencies and trends in the data. As each step in the random walk is independent and identically distributed, it neglects autocorrelation present in real-world time series data (Brockwell and Davis 2002).

1.2 Autocorrelation in Financial Time Series

Autocorrelation, also known as serial correlation, is a crucial concept in time series analysis, particularly in the realm of financial data (Campbell, Lo and MacKinlay 1997). It measures the degree of similarity between observations at different time lags, helping us understand the persistence of past values in the series. In financial time series, autocorrelation plays a significant role in revealing underlying patterns and dependencies that may impact asset prices, stock returns, and other financial variables.

1.2.1 Autocorrelation Function (ACF)

The Autocorrelation Function (ACF) is a primary tool used to quantify the autocorrelation in a time series. It calculates the correlation coefficient between the series and its lagged values at various time lags (Box, Jenkings and Reinsel G. C. 2015). For a financial time series, the ACF at lag k is denoted by ρ(k), and it can be mathematically represented as:

$$\rho(k) = \frac{cov(X_t, X_{t-k})}{\sqrt{var(X_t) * var(X_{t-k})}} \tag{1.2}$$

where X_t and X_{t-k} represent the values of the financial time series at time t and time t-k, respectively. cov() denotes the covariance function, and var() represents the variance.

In financial time series analysis, the ACF provides valuable insights into the presence of autocorrelation patterns. A positive autocorrelation coefficient ρ(k) > 0 at lag k indicates that the series tends to follow its past values, suggesting a positive serial correlation (Campbell, Lo and MacKinlay 1997). On the other hand, a negative autocorrelation coefficient ρ(k) < 0 suggests a negative serial correlation, indicating that the series exhibits alternating fluctuations over time. In Figure 2 we see the autocorrelation of RP generated in Figure 1.

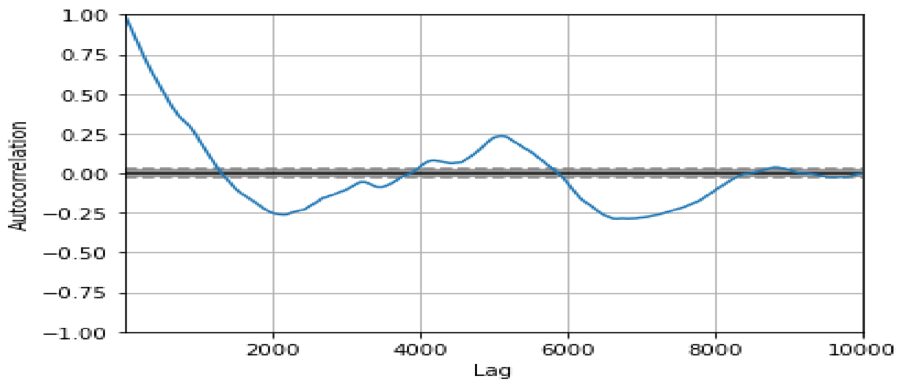

Figure 2 Autocorrelation of Random Walk generated in Figure 1

1.2.2 Practical Significance

The presence of significant autocorrelation in financial time series can have profound implications for investment strategies and risk management. Positive autocorrelation might imply momentum effects, where trends tend to persist, influencing trading decisions and investment strategies (Campbell, Lo and MacKinlay 1997). Conversely, negative autocorrelation might indicate mean reversion patterns, leading to different investment approaches that take advantage of price reversals.

Moreover, understanding the autocorrelation patterns in financial time series is essential for the development and evaluation of forecasting models. Autocorrelation helps identify the appropriate lag length for autoregressive models, such as the Autoregressive Integrated Moving Average (ARIMA) model, enabling better predictions of future asset prices and returns.
Interpreting an autocorrelation (ACF) chart is essential for understanding the temporal dependencies and patterns within a time series data. The ACF chart displays the correlation coefficients between a time series and its lagged versions (previous observations) at various lags (time intervals).

1.2.3 How to interpret it.

Lag Values on the X-Axis: The x-axis of the ACF chart represents the lag values, which indicate how many time points back you are looking in the data. Lag 0 represents the correlation of the time series with itself at the same time point, which is always 1 (perfect correlation). As you move along the x-axis, you are comparing the series at different time points in the past.

Correlation Values on the Y-Axis: The y-axis represents the correlation coefficient between the original time series and the lagged version of itself. The correlation coefficient ranges from -1 to 1, where -1 indicates a perfect negative correlation, 1 indicates a perfect positive correlation, and 0 indicates no correlation.

Interpretation of Correlation Values:

- **Positive Correlation**: If the ACF value is close to 1, it indicates a strong positive correlation between the time series and the lagged version at that lag. In simpler terms, if the value is high at lag 2, it means that observations at time t and time t-2 are positively correlated.
- **Negative Correlation**: ACF values close to -1 indicate a strong negative correlation between the time series and the lagged version. This means that observations at time t and time t-2 are negatively correlated.
- **No Correlation**: If the ACF value is close to 0, it suggests that there is little to no correlation between the time series and the lagged version at that lag.

Statistical Significance: To determine if a correlation is statistically significant, we can look at the shaded region or confidence intervals around the horizontal axis. If an ACF value crosses the upper or lower boundary of the confidence interval, it may indicate a significant correlation at that lag.

Patterns in ACF: Patterns in the ACF chart can reveal seasonality and cyclic behavior in the data. For example, if we observe regularly spaced peaks at lags of 7, 14, 21, etc., it suggests weekly seasonality in the data.

Decay in Correlation: In many time series, we may notice that the correlation tends to decay as the lag increases. This is known as a decaying ACF and indicates that recent observations have a stronger influence on the current value than observations further in the past.

Interpreting an ACF chart involves examining the correlation values at different lags to understand the temporal relationships in your time series data. It helps identify patterns, seasonality, and the influence of past observations on future values, aiding in the selection of appropriate time series models and forecasting.

2 CHAPTER 2: Reinforcement Learning

Reinforcement Learning (RL) is a powerful field of machine learning that addresses the challenge of decision-making in dynamic and uncertain environments (Sutton and Barto 2018). Unlike supervised learning, where the model is provided with labeled examples to learn from, and unsupervised learning, where the model must find patterns in unlabeled data, RL operates through trial and error to learn the best actions to take in different situations.

At the heart of Reinforcement Learning lies the interaction between an agent and an environment. The agent observes the current state of the environment, selects actions based on its policy, and receives feedback in the form of rewards (Sutton and Barto 2018). The objective of the agent is to learn an optimal policy that maximizes the cumulative rewards it receives over time.

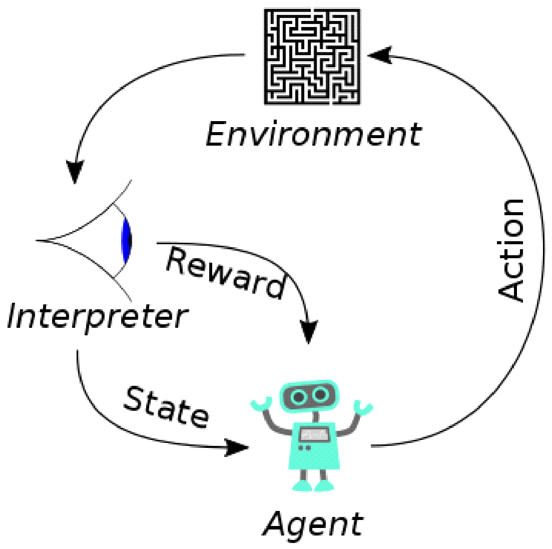

Figure 3 The interaction between an agent and its environment within a Markov Decision Process (MDP).[1]

In a Reinforcement Learning scenario, an agent interacts with an environment by taking actions based on observations, and its actions are rewarded with either a low or high score, depending on their effectiveness of its action Figure 3.

2.1 Components of the Reinforcement Learning Process

Agent: The agent is the learner or decision-maker that interacts with the environment. It is responsible for selecting actions and updating its policy to achieve the best outcomes (Sutton and Barto 2018).

[1] Photo: via Wikimedia Commons, CC 1.0 (https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg)

Environment: The environment represents the external system with which the agent interacts. It is dynamic and can change states based on the agent's actions (Sutton and Barto 2018).

State (s): The state is a representation of the current situation of the environment. It captures all relevant information that the agent needs to make decisions (Sutton and Barto 2018). If it cannot capture all the relevant information then we regard it as an observation which has partial information.

Action (a): Actions are the decisions made by the agent based on the current state. The agent's goal is to learn a policy that maps states to actions to maximize cumulative rewards (Sutton and Barto 2018).

Policy (π): The policy is the strategy followed by the agent to determine which actions to take given a particular state. It can be deterministic or stochastic (Sutton and Barto 2018).

Reward (r): The reward is a scalar feedback signal provided by the environment after each action. It represents the immediate desirability of the action and serves as the basis for the agent to update its policy (Sutton and Barto 2018).

Trajectory (τ): A trajectory is a sequence of states, actions, and rewards that the agent experiences while interacting with the environment (Sutton and Barto 2018).

2.2 Deterministic Policy

A deterministic policy is a type of policy that maps each state directly to a specific action with certainty. In other words, given a particular state, the deterministic policy will always select the same action. Mathematically, it can be represented as:

$$\pi(s) = \mathrm{a} \tag{2.1}$$

Where:

- $\pi(s)$ is the policy that maps state s to an action a.

An illustration of a deterministic policy is when a robot has a fixed set of rules, guiding it to take precise actions according to its current state.

2.3 Stochastic Policy

A stochastic policy, on the other hand, is a type of policy that introduces randomness into the action selection process. Instead of selecting a single deterministic action for each state, a stochastic policy outputs a probability distribution over the action space for a given state. This means that it can select different actions with different probabilities for the same state. Mathematically, it can be represented as:

$$\pi(\mathrm{a}|s) = P(A = \mathrm{a}|S = s) \tag{2.2}$$

Where:

- π(a|s) is the policy that specifies the probability of taking action a in state s.
- A: set of all actions
- S: set of all states

Stochastic policies are more flexible and allow the agent to explore different actions, even in situations where the optimal action is uncertain. They are often used in cases where there is a degree of uncertainty in the environment or when exploration is required to discover the best course of action.

2.4 Markov Decision Process (MDP)

Reinforcement Learning problems are often formalized as Markov Decision Processes. An MDP is defined by a tuple (S, A, P, R, γ), where:

- S is a set of possible states in the environment.
- A is a set of possible actions that the agent can take.
- P is the transition probability, which defines the probability of transitioning from one state to another after taking a specific action.
- R is a value that provides the immediate reward the agent receives after performing an action in a given state.
- γ (gamma) is the discount factor that represents the agent's preference for short-term rewards over long-term rewards (Sutton and Barto 2018).

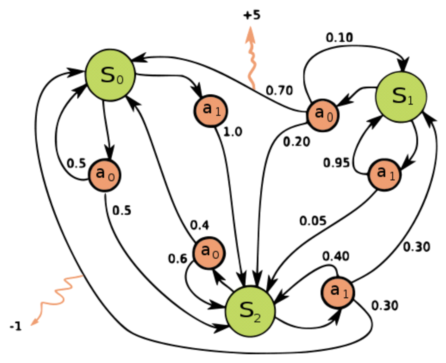

Figure 4 Example of a MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)[2]

[2] by Waldo Alvarez distributed under a CC-BY 4.0 license

2.5 Policy and Value Functions

In Reinforcement Learning, the agent aims to learn an optimal policy (π*) that maximizes the expected cumulative reward over time. The policy can be represented as a mapping from states to actions (π: S → A). Additionally, the agent can also learn value functions to evaluate the desirability of different states and actions.

State-Value Function V(s): The state-value function estimates the expected cumulative reward starting from a given state s and following a specific policy π. It can be mathematically represented as:

$$V(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right] \tag{2.3}$$

Where E_π denotes the expectation with respect to the policy π, r_t represents the reward received at time step t, and γ is the discount factor (Sutton and Barto 2018).

Action-Value Function Q(s, a): The action-value function estimates the expected cumulative reward starting from a given state s, taking action a, and following a specific policy π. It can be expressed as:

$$Q(s,\mathrm{a}) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = 0, a_0 = \mathrm{a}\right] \tag{2.4}$$

where E_π denotes the expectation with respect to the policy π, r_t represents the reward received at time step t, and γ is the discount factor (Sutton and Barto 2018).

2.6 Bellman Optimality Equations in Reinforcement Learning

Reinforcement Learning (RL) aims to enable agents to make optimal decisions in dynamic and uncertain environments. One of the fundamental concepts in RL is the Bellman optimality equation, which provides a powerful framework to compute the optimal state-value function and optimal action-value function.

2.6.1 Optimal State-Value Function

The optimal state-value function, denoted as $V_*(s)$, represents the expected cumulative reward starting from state s under the optimal policy (Sutton and Barto 2018). It satisfies the Bellman optimality equation, given by:

$$V_*(s) = \max_{\mathrm{a}} \sum_{s',r} p(s',r|s,\mathrm{a})[r + \gamma V_*(s')] \tag{2.5}$$

where:

- a is an action in the set of possible actions for state s.
- s' denotes the next state after taking action a from state s.
- r represents the immediate reward received after the transition.
- $p(s',r|s,\mathrm{a})$ is the probability of transitioning to state s' and receiving reward r given the current state s and action a.
- γ is the discount factor, determining the agent's preference for short-term rewards over long-term rewards.

The optimal state-value function captures the expected cumulative reward an agent can obtain from a specific state under the best possible decision-making strategy.

2.6.2 Optimal Action-Value Function

The optimal action-value function, denoted as $Q_*(s,\mathrm{a})$, represents the expected cumulative reward starting from state s, taking action a, and following the optimal policy thereafter (Sutton and Barto 2018). It satisfies the Bellman optimality equation, given by:

$$Q_*(s,\mathrm{a}) = \sum_{s',r} p(s',r|s,\mathrm{a})[r + \gamma \max_{\mathrm{a}'} Q_*(s',\mathrm{a}')] \tag{2.6}$$

where:

- a′ denotes an action in the set of possible actions for state s′.

The optimal action-value function quantifies the expected cumulative reward an agent can achieve by selecting a specific action in a given state and then following the optimal policy.

2.6.3 Solving the Bellman Optimality Equations

Solving the Bellman optimality equations is crucial for finding the optimal value functions, which in turn allows the agent to determine the best possible policy. Various algorithms, such as Value Iteration and Q-Learning, are employed to find the solutions.

2.6.4 Value Iteration Algorithm

Value Iteration is an iterative algorithm used to solve the Bellman optimality equation for the optimal state-value function (Bellman 1957). It starts with an arbitrary value function estimate and repeatedly updates the estimates until convergence. The update rule for Value Iteration is given by:

$$V_{k+1}(s) = \max_{\mathrm{a}} \sum_{s',r} p(s',r|s,\mathrm{a})[r + \gamma V_k(s')] \tag{2.7}$$

where:

- $V_k(s)$ represents the value function estimate at iteration k.

The Bellman optimality equations are essential tools in Reinforcement Learning for computing the optimal state-value function and optimal action-value function. These equations serve as the foundation for various RL algorithms that enable agents to learn optimal policies in uncertain environments. By solving the Bellman optimality equations, agents can make informed and optimal decisions in a wide range of real-world applications.

2.7 Key Concepts and Algorithms in Reinforcement Learning

Reinforcement Learning includes various algorithms and techniques to solve MDPs and learn optimal policies. Some of the key concepts and algorithms in RL include:

- Policy Evaluation and Policy Improvement: Policy evaluation is the process of determining the value function of a given policy π. Policy improvement involves updating the policy to make better decisions based on the estimated value function.
- Model-Based vs. Model-Free RL: RL algorithms can be categorized into model-based and model-free approaches. Model-based methods build a model of the environment's dynamics, while model-free methods learn directly from interactions with the environment.
- Temporal Difference (TD) Learning: TD learning is a type of model-free RL algorithm that combines ideas from dynamic programming and Monte Carlo methods. TD algorithms learn from incomplete sequences of experiences and update value functions iteratively.
- Q-Learning: Q-learning is a widely used off-policy RL algorithm for learning action-value functions. It involves updating the Q-values based on the Bellman equation and does not require knowledge of the environment's dynamics.
- Deep Reinforcement Learning: Deep Reinforcement Learning combines RL with deep neural networks to handle complex and high-dimensional state and action spaces. Algorithms such as Deep Q Networks (DQNs) and Deep Deterministic Policy Gradient (DDPG) have achieved impressive results in various applications.

2.8 Applications of Reinforcement Learning

Reinforcement Learning has shown promising results in various real-world applications:

- Robotics: RL is used in robotics to teach agents to perform complex tasks like object manipulation, locomotion, and control.
- Game Playing: RL algorithms have achieved superhuman performance in playing games like Go, Chess, and Atari games.
- Autonomous Vehicles: RL is employed in autonomous vehicles to make decisions on navigation, path planning, and collision avoidance.
- Finance: RL is used in financial applications for portfolio optimization, algorithmic trading, and risk management.
- Healthcare: RL has potential applications in healthcare for personalized treatment recommendations and optimizing medical treatment protocols.

2.9 Temporal Difference Learning

Temporal Difference (TD) learning is a fundamental and widely-used technique in Reinforcement Learning that combines aspects of dynamic programming and Monte Carlo methods (Sutton and Barto 2018). It is a model-free approach, meaning it does not require explicit knowledge of the environment's dynamics, making it suitable for a broad range of real-world applications. TD learning enables agents to learn from incomplete experiences through online updates, providing a powerful tool for sequential decision-making problems.

Temporal Difference learning algorithms operate by bootstrapping, using estimates of future values to update current value estimates (Sutton and Barto 2018). This approach enables agents to learn efficiently by iteratively updating their value functions based on the observed experiences without waiting for the completion of entire episodes or trajectories.

The most fundamental Temporal Difference algorithm is TD(0), which stands for Temporal Difference with a one-step look ahead (Sutton and Barto 2018). In TD(0), the value function is updated based on the immediate reward and the estimated value of the next state:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.8}$$

where:

- $V(s_t)$ is the estimated value of state s_t at time step t.
- r_{t+1} is the reward received after taking action α_t from state s_t and transitioning to state s_{t+1}.
- α is the learning rate, determining the step size for value updates.
- γ is the discount factor, representing the agent's preference for short-term rewards over long-term rewards.

2.9.1 TD(0) Algorithm

The TD(0) algorithm is a simple yet powerful method in Temporal Difference learning (Sutton and Barto 2018). The algorithm proceeds as follows:

1. Initialize the value function V(s) for all states.
2. Observe the current state s_t.
3. Take an action a_t based on the agent's policy.
4. Receive the reward r_{t+1} and observe the next state s_{t+1}.
5. Update the value function for the current state using the TD(0) update rule.

The agent repeats these steps until it reaches the termination condition or a predefined number of iterations.

2.9.2 Advantages of Temporal Difference Learning

Temporal Difference learning offers several advantages that contribute to its widespread use in Reinforcement Learning:

- Online Learning: TD algorithms can update their value estimates after each time step, making them suitable for online and real-time learning scenarios.
- Model-Free Approach: TD learning is model-free, eliminating the need for explicitly modeling the environment's dynamics, which is often challenging or impossible in real-world applications.
- Efficiency: TD algorithms use bootstrapping, enabling them to learn from incomplete experiences, resulting in more efficient learning compared to Monte Carlo methods.

2.10 Q-Learning

Q-learning is a powerful and widely-used Reinforcement Learning algorithm that aims to learn the optimal action-value function, denoted as Q(s,a) (Sutton and Barto 2018). The Q-value represents the expected cumulative reward starting from a state s, taking action a, and following an optimal policy thereafter.

Q-learning is a model-free algorithm, meaning it does not require explicit knowledge of the environment's dynamics (Sutton and Barto 2018). Instead, the agent learns from interactions with the environment, iteratively updating its Q-values to approximate the optimal action-value function. The primary goal of Q-learning is to find the optimal policy that maximizes the cumulative reward over time for any given state.

The Q-learning update rule is based on the Bellman equation for the optimal action-value function, which can be expressed as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.9}$$

where:

- $Q(s_t, a_t)$ is the Q-value for state s_t and action a_t time step t.
- α is the learning rate, determining the step size for value updates.
- r_{t+1} is the reward received after taking action a_t from state s_t and transitioning to state s_{t+1}.
- γ is the discount factor, representing the agent's preference for short-term over long-term rewards.
- $\max_a Q(s_{t+1}, a)$ represents the maximum Q-value over all possible actions in the next state s_{t+1}.

The Q-learning update rule efficiently updates the Q-values based on the observed experiences, allowing the agent to converge towards the optimal action-value function.

2.10.1 Q-Learning Algorithm

The Q-learning algorithm can be summarized as follows in Figure 5:

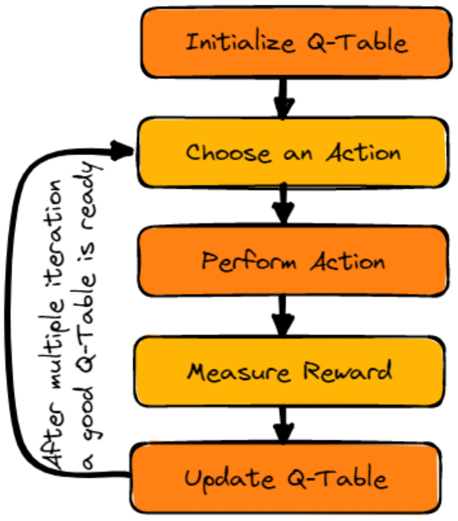

Figure 5 Q-learning algorithm[3]

1. Initialize the Q-values Q(s,a) for all state-action pairs.
2. Observe the current state s_t.
3. Select an action a_t using an exploration-exploitation strategy, such as ϵ-greedy.
4. Perform the action a_t and observe the reward r_{t+1} and the next state s_{t+1}.
5. Update the Q-value for the current state-action pair using the Q-learning update rule.

[3] image by https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial

6. Set the current state s_t to the next state s_{t+1}.
7. Repeat steps 3 to 6 until the termination condition is met or a predefined number of iterations.

2.10.2 ϵ-greedy Strategy

Initially, the agent is in exploration mode and chooses a random action to explore the environment. The Epsilon Greedy Strategy is a simple method to balance exploration and exploitation. The epsilon stands for the probability of choosing to explore and exploits when there are smaller chances of exploring.

At the start, the epsilon rate is higher, meaning the agent is in exploration mode. While exploring the environment, the epsilon decreases and agents start to exploit the environment. During exploration in each iteration, the agent becomes more confident in estimating Q-values.
Q-learning is model-free, allowing it to be applied to problems where explicit knowledge of the environment's dynamics is difficult or impossible to obtain. It is an off-policy algorithm, meaning that it learns the optimal policy while following a different policy during exploration and data collection. In other words, the agent learns from experiences generated by a different policy than the one it is trying to improve. This is in contrast to on-policy algorithms, where the agent learns and updates the policy based on the experiences it collects while following the current policy.

2.10.3 Some characteristics of off-policy algorithms:

Data Collection: Off-policy algorithms can use data generated by any policy, not just the policy being currently evaluated or updated (Sutton and Barto 2018). This feature allows the agent to collect data more efficiently since it can learn from old experiences, which may be sampled from a different, possibly exploratory, policy.

Importance Sampling: The key technique used by off-policy algorithms is importance sampling. When the agent learns from data collected by a different policy, it needs to adjust the learning process to account for the discrepancy between the current policy and the policy that generated the data. Importance sampling is used to correct for this difference in policies (Sutton and Barto 2018).

Exploration and Exploitation: Since off-policy algorithms can use data generated by a more exploratory policy, they have more flexibility in exploration. This means they can explore more widely and possibly find better solutions. On the other hand, on-policy algorithms are often more conservative in their exploration because they directly follow the current policy.

Stability and Sample Efficiency: Off-policy algorithms can be more sample-efficient and stable in learning because they can reuse data more effectively. However, importance sampling introduces variance, and the stability of off-policy algorithms can be influenced by the discrepancy between the data-generating policy and the policy being updated (Sutton and Barto 2018).

2.11 Policy Gradients

Policy gradients constitute a family of algorithms utilized for solving reinforcement learning problems by directly optimizing the policy in the policy space, as opposed to value-based approaches like Q-learning, which estimate the value function for each state (Sutton and Barto 2018). Policy gradients offer several appealing properties, such as producing stochastic policies by learning a probability distribution over actions given observations (Sutton and Barto 2018). In contrast, value-based methods are deterministic and select actions greedily with respect to the learned value function, potentially leading to under-exploration and necessitating exploration strategies like ϵ-greedy to address this issue (Sutton and Barto 2018).

A significant advantage of policy gradients is their capability to handle continuous action spaces without requiring discretization, a necessity for value-based methods (Sutton and Barto 2018). However, policy gradients suffer from high variance estimates of gradient updates, leading to noisy gradient estimates that can destabilize the learning process. To address this limitation, extensive research has focused on reducing the variance of gradient updates to improve algorithm stability (Williams 1992).

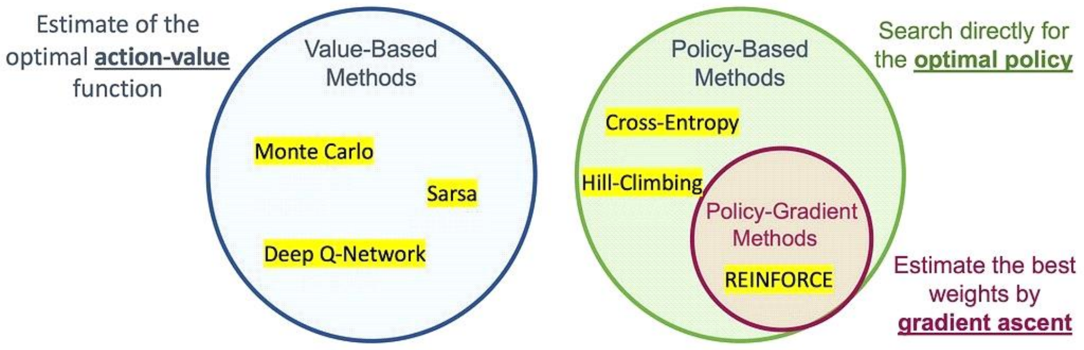
Estimate of the optimal action-value function
Value-Based Methods
Monte Carlo
Sarsa
Deep Q-Network
Policy-Based Methods
Search directly for the optimal policy
Cross-Entropy
Hill-Climbing
Policy-Gradient Methods
REINFORCE
Estimate the best weights by gradient ascent

Figure 6 Summary of approaches in Reinforcement Learning. The classification is based on whether we want to model the value or the policy[4]

[4] source: https://torres.ai

Policy-gradient methods belong to the broader category of Policy-Based methods as shown in Figure 6, which estimate the optimal policy's weights through gradient ascent (Schulman, et al. 2015). The gradient ascent process begins with an initial guess for the policy's weights that maximize the expected return. Subsequently, the algorithm evaluates the gradient at that point, indicating the direction of the steepest increase in the expected return function. Small steps are then taken in that direction, aiming to reach a new value of the policy's weights that yield a slightly higher expected return (Schulman, et al. 2015). The algorithm iteratively repeats this process of evaluating the gradient and taking steps until it converges to an estimate of the maximum expected return.

Policy-based methods can be used to learn either stochastic or deterministic policies (Sutton and Barto 2018). In the case of a stochastic policy, the neural network's output represents an action vector that forms a probability distribution, rather than returning a single deterministic action. The agent then selects an action from this probability distribution, meaning that if the agent encounters the same state twice, it may take different actions each time. This probabilistic representation of actions offers advantages, including smoother representations and more stable gradient optimization (Sutton and Barto 2018).

In contrast, deterministic policies with discrete outputs can lead to significant changes in actions even with small adjustments to the weights. However, when the output is a probability distribution, small changes to the weights typically result in minor changes in the output distribution, enhancing the stability of gradient optimization (Sutton and Barto 2018).

The core idea behind policy gradients is the reinforcement of good actions. The method iteratively adjusts the policy network weights to increase the probabilities of actions that lead to higher returns and decrease the probabilities of actions that result in lower returns, ultimately converging to the optimal policy (Sutton and Barto 2018). By reinforcing favorable actions and reducing the likelihood of unfavorable actions, policy gradients facilitate the agent's learning process in complex and uncertain environments.

Policy gradients present a powerful family of algorithms for reinforcement learning that directly optimize policies in the policy space. They offer numerous advantages, such as handling continuous action spaces and providing stochastic policies (Sutton and Barto 2018). Despite their high variance estimates, ongoing research aims to improve the stability and performance of policy gradient methods, making them a valuable tool in solving various real-world RL problems.

2.12 REINFORCE algorithm

2.12.1 Trajectories in Reinforcement Learning

In Reinforcement Learning, the term "trajectory" denotes sequence of states, actions, and rewards. Unlike episodes, trajectories are more versatile due to their ability to encompass a complete episode or a segment (Sutton και Barto 2018). The parameter ν, known as the horizon quantifies the length of the trajectory. A trajectory is composed of successive tuples, encapsulating state-action-reward transitions:

$$\tau = (s_0, \mathrm{a}_0, r_1, s_1, \mathrm{a}_1, r_2, s_2, \dots \mathrm{a}_\nu, r_{\nu+1}, s_{\nu+1}) \tag{2.10}$$

It is the foundation in REINFORCE method, as it aligns with the maximization of expected returns, offering a versatile approach applicable to both episodic and continuous tasks (Williams 1992). While the typical scenario involves employing an entire episode as a trajectory, this approach is particularly suited to episodic tasks where rewards are exclusively given at the conclusion of an episode (Williams 1992). This ensures a sufficient amount of reward information is available for the accurate estimation of expected returns.

2.12.2 Trajectory Return

The concept of return plays a pivotal role in assessing the effectiveness of various policies and strategies. In essence, the return represents the total rewards an agent can accumulate throughout a trajectory, which involves a series of transitions comprising states, actions, and rewards. This return metric helps in evaluating the efficacy of specific actions and policies.

Mathematically, the return at time step t, denoted as G_t, is defined as the summation of discounted rewards obtained along the trajectory from time step t until the end of the trajectory. This definition takes the following form:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.11}$$

wherer r_{t+k+1} denotes the reward received at time step t+k+1, and γ is the discount factor that reflects the agent's preference for immediate rewards over delayed ones (Sutton and Barto 2018). The discount factor ensures that future rewards are valued less than present ones, capturing the notion of time preference in decision-making.

The return can be seen as a measure of the cumulative "worth" of a trajectory to an agent. It encapsulates the balance between immediate rewards and the potential for future rewards. In episodic tasks, where trajectories have a clear start and end, the return is typically summed over the entire trajectory, considering rewards from the current time step until the end of the episode.

In RL algorithms, optimizing for the return is central to policy learning. The ultimate goal is to find policies that maximize the expected return over trajectories. This involves identifying actions that lead to the most favorable sequence of rewards while considering the potential long-term consequences.

2.12.3 Expected return

The objective of this algorithm is to discover the neural network weights denoted as θ, which maximize the expected return denoted as U(θ) and defined as follows:

$$U(\theta) = \sum_{\tau} P(\tau;\theta)R(\tau) \tag{2.12}$$

The return R(τ) is expressed as a function of the trajectory τ. P(τ;θ) represents the probabilities associated with each possible trajectory. That probability depends on the neural network weights θ and defines the policy used to select the actions in the trajectory, which also determines the states that the agent observes.

2.12.4 Gradient ascent

One effective approach to find the value of θ that maximizes the U(θ) function, is gradient ascent. To provide an intuitive visualization, we can think of gradient ascent as a process of ascending a hill. It involves U(θ) taking incremental steps as shown in Figure 7 in the direction of the gradient, systematically guiding the strategy towards reaching the highest point. (Sefidian n.d.)

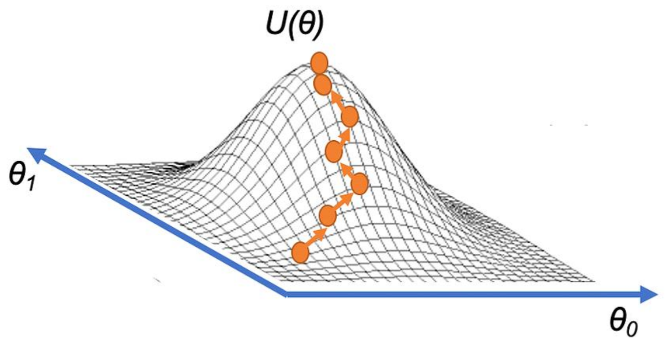

Figure 7 Gradient ascent strategy.[5]

Mathematically, the update step for gradient ascent can be expressed as:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} U(\theta) \tag{2.13}$$

where α is the step size that is generally allowed to decay over time (equivalent to the learning rate decay in deep learning). Once it's known how to estimate this gradient, this update step it is repeatedly applied, expecting that θ converges to the value that maximizes U(θ).

[5] source: https://torres.ai

2.12.5 Sampling and estimate

To apply this method, we must be able of computing the gradient ∇U(θ). However, exact gradient calculation remains computationally prohibitive, as it demands evaluating every potential trajectory a task that typically becomes infeasible. Instead, this approach employs trajectory sampling via the policy and relies on these sampled trajectories to estimate the gradient. (Sefidian n.d.)

The following pseudo code describes in more detail the behavior of this method and can be written as:

```
Input: a differentiable policy parameterization $\pi(\alpha|s;\ \theta)$
Step size $\alpha > 0$
Initialize the policy parameter $\theta$ at random
    (1) Use the policy $\pi_\theta$ to collect a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots a_v, r_{v+1}, s_{v+1})$
    (2) Estimate the return of the trajectory $\tau$: $R(\tau) = (G_0, G_1, \dots, G_v)$
        where $G_k$ is the expected return for transition k:
            $G_k \leftarrow \sum_{t=k+1}^{v+1} \gamma^{t-k-1} R_k$
    (3) Use the trajectory $\tau$ to estimate the gradient $\nabla_\theta U(\theta)$
            $\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{v+1} \nabla_\theta log\pi_\theta (\alpha_t|s_t) G_t$
    (4) Update the weights $\theta$ of the policy
            $\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$
    (5) Loop over steps 1-5 until not converged
```

2.12.6 Gradient estimation formula

The formula the gives the gradient estimation is

$$\nabla_\theta \log \pi_\theta(\alpha_t|s_t) \tag{2.14}$$

This formula will adjust the weights of the policy θ in order to increase the log probability of selecting action a_t from state s_t. In specific, the policy weights are adjusted by taking a small step in the direction of this gradient. In that case, it will increase the log probability of selecting the action from that state, and will decrease the log probability if it takes the opposite direction. (Sefidian n.d.)

The following equation performs all these updates simultaneously for each state-action pair (a_t, s_t) at each time step t in the trajectory:

$$\sum_{t=0}^{\nu} \nabla_\theta \log \pi_\theta(\alpha_t|s_t) G_t \tag{2.15}$$

In a gradient ascent algorithm where the objective is to maximize some probability p we actually optimize the log probability log(p) for some network parameter theta.

The reason is that it generally works better to optimize log(p) than p due to the gradient of log(p) that is generally more well-scaled. Probabilities are bounded by 0 and 1 by definition, so the range of values that the optimizer can operate over is limited and small. (Sefidian n.d.)

In that case, sometimes probabilities may be extremely low near to zero or very high close to one. This may cause numerical issues when optimizing on a computer with limited numerical precision. If we instead use a surrogate objective, namely log(p) (natural logarithm), we have an objective that has a larger "dynamic range" than raw probability space, since the log of probability space ranges from (-∞,0), and this makes the log probability easier to compute.

3 CHAPTER 3: Related Work

In his work (Gao 2018) conducted two idealized trading games, the first with one input of wave-like price time series the "Univariate" and a second with two inputs a random stepwise price time series and a noisy signal the "Bivariate". The first tests whether the agent can capture the underlying dynamics and the second tests whether the agent can utilize the hidden relation among the inputs. To model the Q values, various architectures were used like a Gated Recurrent Unit (GRU), a Long Short-Term Memory (LSTM), a Convolutional Neural Network (CNN), and a multi-layer perceptron (MLP). Both games ended with a profitable strategy. The GRU-based agent showed best overall performance in the "Univariate" game and the MLP-based agents showed better performance in the Bivariate game.

In his work (Huang 2018) presents a Markov Decision Process (MDP) model tailored for financial trading tasks, using the deep recurrent Q-network (DRQN) algorithm. To adapt the learning algorithm to the specifics of financial trading, several key modifications are proposed. In particular, they adopt a significantly reduced replay memory size, consisting of only a few hundred samples, in contrast to the larger sizes commonly utilized in modern deep reinforcement learning algorithms, often in the millions. An innovative action augmentation technique is introduced to reduce the reliance on random exploration. This technique provides additional feedback signals for all actions to the agent, enabling the use of a greedy policy during the learning process. Notably, this approach demonstrates strong empirical performance, particularly when compared to the more frequently employed ε-greedy exploration strategy. It's worth noting that this technique is tailored to the context of financial trading and operates under specific market assumptions. It is proposed longer sampling sequences for training recurrent neural networks (RNNs). This modification not only facilitates agent training every T steps but also significantly reduces the overall computational burden, effectively scaling down computation by a factor of T.

In their work (Chen, Luo and Yu 2021) present an approach in which over 100 short-term alpha factors are employed to characterize the states within the Markov Decision Process (MDP), diverging from the conventional parameters such as price, volume, and various technical indicators. In contrast to prior methods involving DQN (deep Q-learning) and BC (behavior cloning), the study introduces expert knowledge during the training phase. This approach takes into account both the interactions between the expert and the environment and those between the agent and the environment when designing the temporal difference error. The goal is to enhance the agents' adaptability in the inherent noise prevalent in financial data. The experimental findings demonstrate the clear advantages of this proposed methodology when compared to three typical technical analysis techniques and two deep learning-based approaches.

Jeong and Kim use reinforcement learning-based trading systems aimed to maximize profits and adapt to real financial market conditions, addressing data limitations (Jeong and Kim 2019). First, they introduce an automated trading system that predicts the number of shares to trade by combining deep Q-networks with a deep neural network (DNN) regressor. Second, they investigate various action strategies based on Q-values to optimize profits in turbulent markets. Lastly, transfer learning approaches are proposed to counter overfitting due to limited financial data. Experimental results across four stock indices show substantial profit increases, with the combined system outperforming both the market and the standard reinforcement learning model in all cases.

A novel Adaptive Deep Deterministic Reinforcement Learning approach (Adaptive DDPG) designed for portfolio allocation tasks, especially in complex and dynamic stock markets (Li, et al. 2019). This method incorporates optimistic and pessimistic deep reinforcement learning influenced by prediction errors. The study uses daily prices of Dow Jones 30 stocks for training and testing. Comparisons with vanilla DDPG, the Dow Jones Industrial Average index, and traditional min-variance and mean-variance portfolio allocation strategies reveal that Adaptive DDPG outperforms these baselines in terms of investment returns and the Sharpe ratio.

This paper (Li, Zheng and Zheng 2019) addresses challenges in algorithmic trading related to feature extraction and the design of adaptable trading strategies. Unlike previous methods that relied on domain knowledge and lacked flexibility, the authors propose a novel trading agent based on deep reinforcement learning. They extend value-based deep Q-network (DQN) and Asynchronous Advantage Actor-Critic (A3C) approaches and incorporate Stacked Denoising Autoencoders (SDAEs) and LSTM networks for robust market representation. The experimental results demonstrate that their trading agent surpasses baseline methods, consistently delivering stable risk-adjusted returns in both stock and futures markets.

Zhipeng Liang and colleagues explore the application of three cutting-edge continuous reinforcement learning algorithms, Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO), and Policy Gradient (PG), in the context of portfolio management (Liang, et al. 2018). These algorithms, widely used in fields like game playing and robot control, are evaluated under various settings, including different learning rates, objective functions, and feature combinations. The experiments, conducted in the China Stock market, reveal that PG is more suitable for financial markets than DDPG and PPO, despite the latter two being more advanced. Additionally, the paper introduces an Adversarial Training method that significantly enhances training efficiency and improves average daily return and Sharpe ratio in backtesting.

A novel model called TFJ-DRL (Time-Driven Feature-Aware Jointly Deep Reinforcement Learning) designed to tackle challenges in algorithmic trading is proposed in (Lei, et al. 2020). This model combines deep learning and reinforcement learning to enhance the learning of financial signal representations and improve decision-making in trading. It does so by adaptively selecting and reweighting financial signal features, summarizing the attention values between historical data and current trends, and iteratively training with supervised deep learning and reinforcement learning. Experimental evaluations using real-world financial data with various price trends demonstrate the robust performance and broad applicability of TFJ-DRL, particularly in terms of increasing investment returns.

In their study (Liu, et al. 2022) investigate the use of deep reinforcement learning to enhance stock trading strategies for maximizing investment returns. It employs daily prices of 30 selected stocks as the training and trading environment, training a deep reinforcement learning agent to develop an adaptive trading strategy. The agent's performance is assessed and compared to benchmarks, namely the Dow Jones Industrial Average and the traditional min-variance portfolio allocation strategy. The results indicate that the proposed deep reinforcement learning approach outperforms both baselines in terms of both the Sharpe ratio and cumulative returns.

4 CHAPTER 4: Trading game

We will try to handle the problem of acting optimally on a trading game (Gao 2018) applied not on real financial time series but on synthetic stochastic processes and in particular on random walk processes. The science of taking an action can be sufficiently modeled with Reinforcement Learning than using a conventional supervised method. We have also included the task of taking action with a supervised model and the results show that the problem is resolved more effectively with RL i.e. an agent taking decisions within an environment trying to capture the dynamics involved in the process. (Gao 2018).

4.1 Random Walk generation

We use a stochastic process to simulate the actual time series in the game. The series is created using the formula

$$x(t) = x(t-1) + \frac{N(0,1)}{100} \tag{4.1}$$

x(t) is the value of series at time t
x(t-1) the value at time t-1
N(0,1) is standard normal distribution

In the following Figure 8 we can see the result of applying this formula setting seed=18. We will use this generated time series as the basis of the research i.e. all the training to an RL algorithm or a supervised method will be using this seed. Although we refer time series for simplicity we don't include time on x axis. It can be regarded that the transition from time t to t+1 can be any time span for example 1 hour, 2 hours etc.

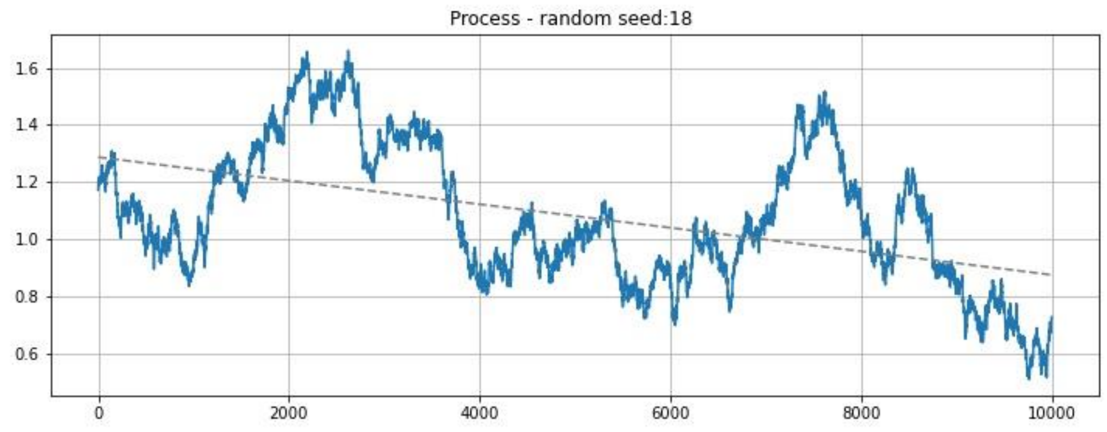

Figure 8 Random Walk generated with seed=18

Every process generated either for training or for testing will contain 10000 values. We consider that 10000 values are enough for an algorithm to capture the underlying dynamics of the model.

4.2 Charts and definitions

|In the following Figure 9 we can see the parts of the process.

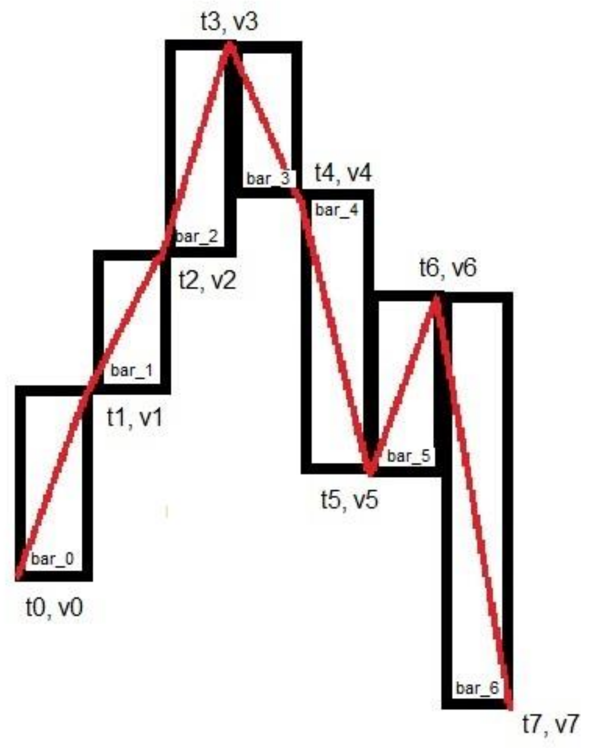

Figure 9 Bar transitions

Period or bar:

It is considered the transition of going from time t to time t+1 and the transition of value v to v+1. So in a bar (period) is considered to go from t0 to t1 and the respective value from v0 to v1.

Orders:

An order is initiated at the beginning of the bar and is either a long buy or a short sell order. At the end of the bar for either order there is a close position and at the same time (t+1 now) a new order is initiated for the next bar according to the model.

Buy order.

The model buys at time t and closes position (sells) at time t+1

$(p/l)_{t+1} = v_{t+1} - v_t$ where p/l = profit or loss of order execution at the end of the bar

At this point a profit is logged if $v_{t+1} > v_t$ or a loss if $v_{t+1} < v_t$

Short sell order.

The model short sells at time t and closes position (buys) at time t+1

$(p/l)_{t+1} = v_{t+1} - v_t$ where p/l = profit or loss of order execution at the end of the bar
At this point a profit is logged if $v_{t+1} < v_t$ or a loss if $v_{t+1} > v_t$

Buy and hold order.

While a “buy and hold” order is not directly supported by the environment, the agent can effectively replicate it by placing continuous buy orders on each price bar.

Short Sell and hold order.

The same as with “Buy and hold” the agent can effectively replicate it by placing continuous short sell orders and closing them on each price bar.

Cumulative return

Cum return is the total sum of rewards (profit or loss) over the length of the process

$$cum\_return(k) = \sum_{t=0}^{t=k} p/l_t \tag{4.2}$$

It is considered that on each trade there is a **fixed bet**. There is no money management. What we are trying to test is the efficiency of the algorithms not the money management strategies that my give quite different results.

4.3 Environment

The most critical aspect of the problem lies in the proper construction of the environment, with two key elements being of importance: defining the state component and formulating the reward function.

The environment is designed to be compatible with openAI Gym (Brockman, et al. 2016) , allowing it to seamlessly integrate with frameworks that operate within such environments. One notable framework that supports this compatibility is Stable-Baselines3 (Raffin n.d.).

Stable-Baselines3 comprises a set of robust implementations for reinforcement learning algorithms in PyTorch. It offers a clean and straightforward interface, granting access to fully-implemented reinforcement learning algorithms. To illustrate the usage of the Stable-Baselines3 framework, consider the following simple example:

```
import gym
from stable_baselines3 import SAC
# Train an agent using Soft Actor-Critic on Pendulum-v0
env = gym.make("Pendulum-v0")
model = SAC("MlpPolicy", env, verbose=1)
# Train the model
```

```
model.learn(total_timesteps=20000)
# Save the model
model.save("sac_pendulum")
# Load the trained model
model = SAC.load("sac_pendulum")
# Start a new episode
obs = env.reset()
# What action to take in state `obs`?
action, _ = model.predict(obs, deterministic=True)
(Raffin n.d.)
```

The process starts by initializing an OpenAI Gym-compatible (Brockman, et al. 2016) environment 'Pendulum-v0.' Next, it initializes the Soft Actor-Critic (SAC) algorithm. Training the agent within the environment is accomplished using the 'learn' method, and the trained model is subsequently saved using the 'save' method. To generate actions based on the current environment state, the 'predict' method is used with the state provided as an argument. This methodology mirrors the approach commonly applied in the scikit-learn library for supervised learning. After defining the algorithm, training the agent on the environment is executed with the 'learn' method, while the 'predict' method is employed to generate actions based on the current state.

4.3.1 RandomWalkEvn class

The environment is implemented with Class RandomWalkEnv(gym.Env): and is initialized with the arguments.

- size: the length of the process to be generated with a default value of 10000
- random_seed: the random seed to feed the random number generator with a default value of 18
- equity: The initial equity amount to employ. In all of our scenarios, we have consistently used 1 to ensure results are comparable.
- enable_metric: It is metric used to assess the progress of the agent. If the agent performs badly a done=False is triggered to indicate failure and complete the episode.
- zero_start: At the beginning of an episode the index of the process is zero.

Here is a brief explanation of class methods and properties.

__generate_data(seed=18):
Generates the random process with the specified seed.

reset():
Initializes the environment, a necessary step before starting an epoch. It returns an observation by which we can feed the model. Compatible with openAI.gym

step(action_idx):
It is a function used to take a single step in the game simulation. It returns observation, return, done, and info. Compatible with openai.gym

render(mode='human', close=False):
Does nothing. Just used for compatibility with openAI gym.

close():
Does nothing. Just used for compatibility with openAI gym.

disp_dataset(extra_str='', save_plot=None):
Displays a plot of the random process generated.

disp_equity(extra_str='', save_plot=None):
Display the equity after the agent has competed a simulation.

_disp_metric():
Used mostly for debugging the metric.

limits():
Shows the upper and lower limits the state can take

save_limits():
Saves the upper and lower values of the state to disk

load_limits():
Loads the upper and lower values from disk

trim_df():
Trims higher and lower state values

steps:
Returns the steps of a simulation

accuracy:
Returns the accuracy of the simulation

p_l:
Returns the total profit or loss after a simulation

observation_space_n:
Returns the number of values of the observation. In that case is a scalar of 5

action_space_n:
Returns the number of actions. In that case is 2

4.3.2 States

State space $\in \mathbb{R}^5$. Every state representing the environment is a vector (pct_{10}, pct_{20}, pct_{30}, pct_{50}, pct_{100}). The choice is heuristic and is based in the fact that we want them to have some autocorrelation that's to be able to capture some underlying trends as shown in figures 10 through 14. If we choose a vector quite near to the current time t then the autocorrelation would be negligible.

$$pct_k = \frac{X_t}{X_{t-k}} - 1 \tag{4.3}$$

Where pct = percentage change.

Using percentage changes as inputs for time series prediction offers several notable benefits. Firstly, it helps in normalizing data, making it comparable across different scales and units, which is essential for the accurate assessment of trends and patterns. Secondly, it inherently accounts for the relative variations within the data, emphasizing the proportional changes rather than the absolute values. This can be particularly advantageous when dealing with variables that exhibit different magnitudes. Furthermore, percentage changes often reveal the underlying growth or decay rates within a time series, providing valuable insights into the inherent dynamics of the data. Lastly, they can enhance the interpretability of the models by expressing predictions in relative terms, facilitating a more intuitive understanding of the forecasted outcomes.

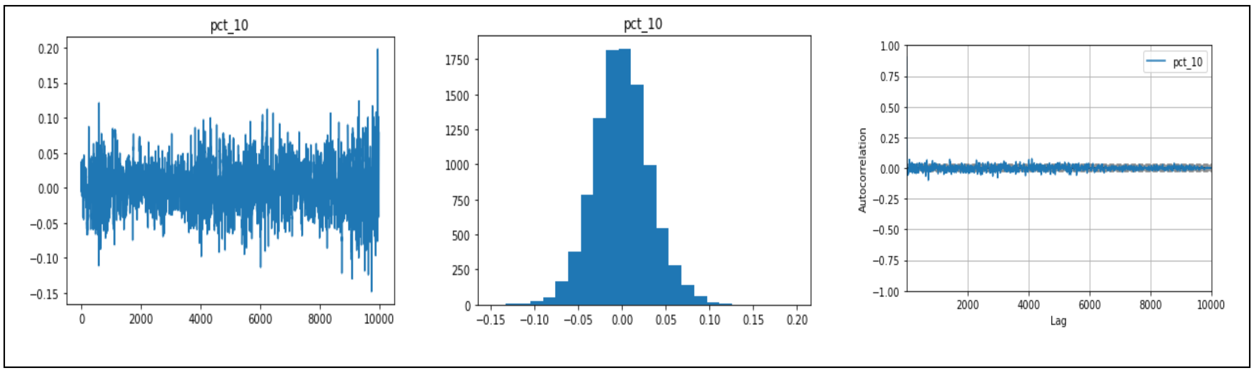

Figure 10 left: pc_{10} for RP with seed=18, middle: distribution, right: autocorrelation function

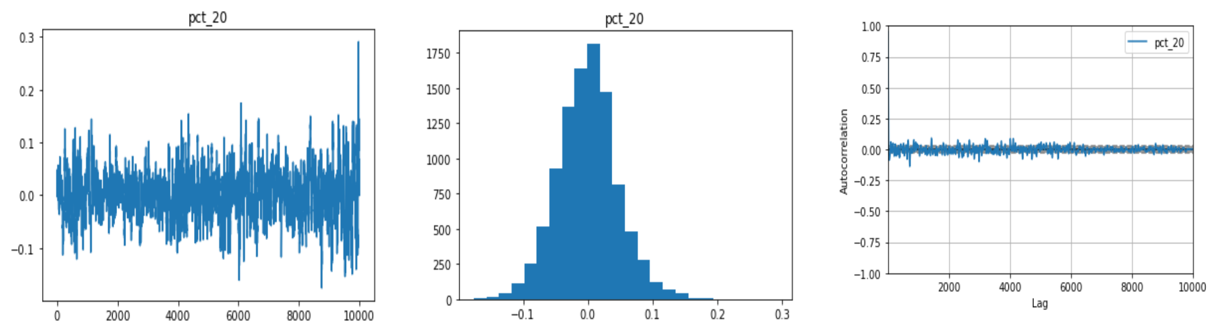

Figure 11 left: pc_{20} for RP with seed=18, middle: distribution, right: autocorrelation function

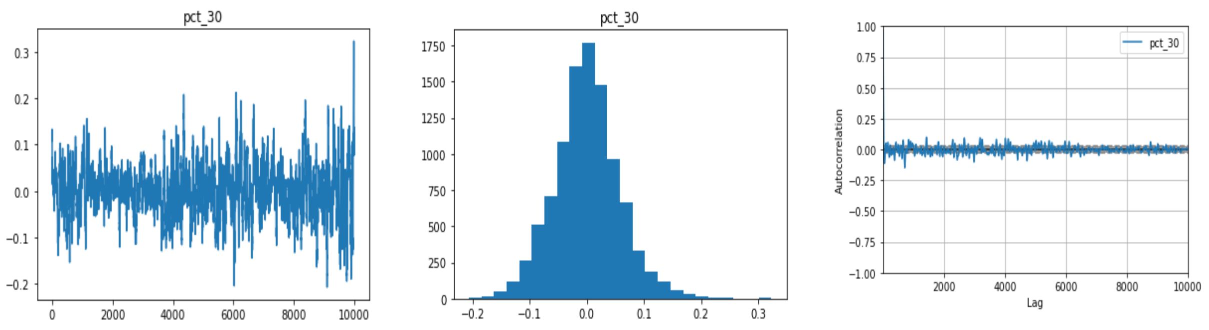

Figure 12 left: pc_{30} for RP with seed=18, middle: distribution, right: autocorrelation function

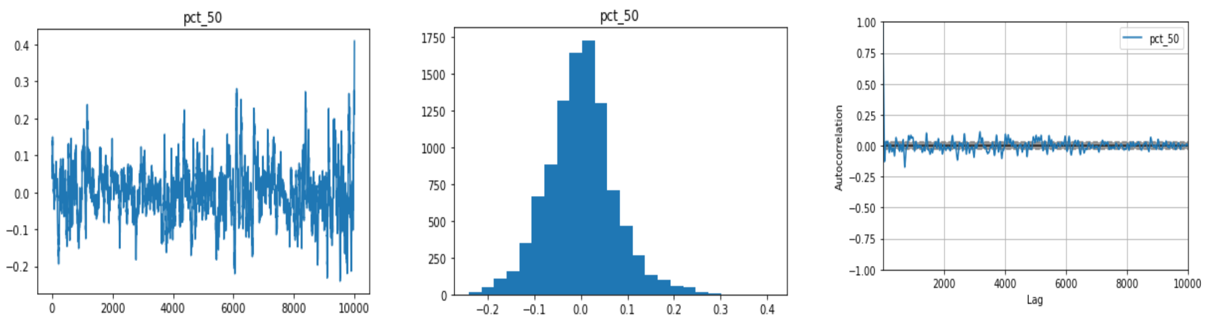

Figure 13 left: pc_{50} for RP with seed=18, middle: distribution, right: autocorrelation function

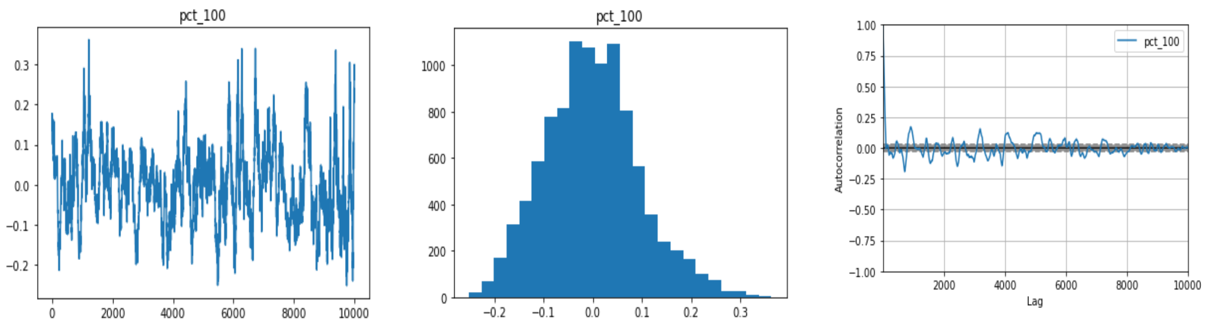

Figure 14 left: pc_{100} for RP with seed=18, middle: distribution, right: autocorrelation function

4.3.3 Reward function

Constructing an appropriate reward function is crucial in reinforcement learning for trading systems. The reward function guides the agent's learning process, helping it make effective decisions. Reward functions should provide clear signals to the agent, encouraging it to maximize desired objectives. In our case, where the goal is to maximize profit or minimize loss in a trading system with fixed bet sizes of 1 currency unit i.e. €1, here are some considerations and alternative approaches to constructing the reward function:

Profit or Loss (P/L) Reward

Reward the agent with the actual profit or loss after a position closes. This provides a clear and direct measure of the success of each trade.

Reward for long buys: reward = (exit_price - entry_price) * bet_size
Reward for short sells: reward = (entry_price - exit_price) * bet_size

This approach directly aligns with our trading objective.

4.4 Discretized Q-learning

Discrete Q-leanring algorithm is implemented in discreteQAgent class. The RandomWalkEnv environment returns observations as a five element array for example, array([0.03539416, 0.04126915, 0.03347519, 0.00899888, 0.04818559]).

So every time the discreteQAgent receives an observation has to quantize it and assigns it to a state in the Q-table.

The discreteQAgent is initialized with the following arguments.

- *env*: the environment that the discreteQAgent will interact
- *bins_n*: number of bins used for state discretization
- *alpha*: alpha parameter for Q-learning algorithm
- *gamma*: gamma parameter for Q-learning algorithm

The number of states in the Q-table is equal to the number of bins raised to the number of state elements. In our case we use 24 bins and 5 state elements so total states = 24^5= 7962624 states. So the size of the Q-table depends on the number of bins and the elements of the observation. If the size of the observation is higher the Q-table will become enormous and won't be possible to fit in memory. One way to bypass this is to decrease the number of bins but decreasing that will decrease also the discretization that's two or more actual states may be assigned to the same entry of the Q-table. So when the discreteQAgent receives an observation from the environment using the env.step() method it will assigns it to the proper state and will adjust the value function of that state. Then using Q-leanring algorithm calculates the Q-values for the two actions.

In all seeds we have initialized the algorithm with the following parameters

Parameter	**Value**
bin_n	24
alpha	0.01
gamma	0.9
enable_metric	True
Zero_start	False

Size of random process	10000
seed	18
episodes	500
eps_strategy	3

Table 1 Parameters on discrete-Q train algorithm

Regarding epsilon strategy we have three options

$$eps1(n) = \frac{1.0}{1 + n * 10^{-3}} \tag{4.4}$$

$$eps2(n) = \frac{0.5}{1 + n * 10^{-3}} \tag{4.5}$$

$$eps3(n) = \frac{1.0}{\sqrt{n+1}} \tag{4.6}$$

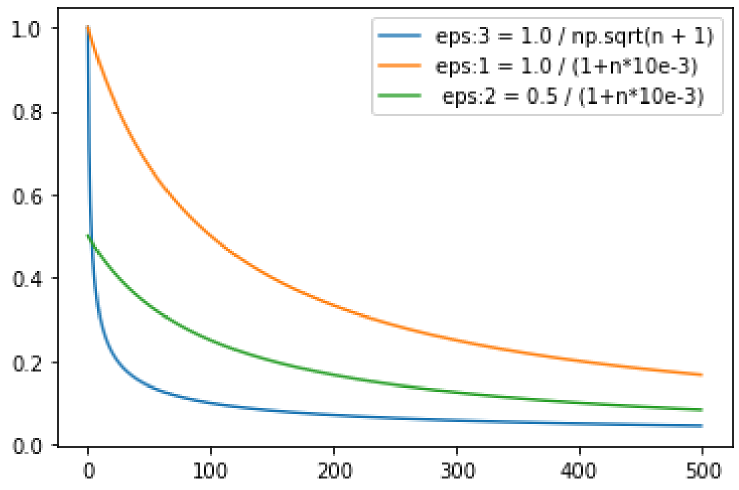

Figure 15 Epsilon greedy options

4.4.1 Training the discreteQAgent.

Here is a sample code how we trained the discreteQAgent with table 1 parameters.

```
env = RandomWalkEnv(size=10000, enable_metric=True, zero_start=False)
env.save_limits()
Q_agent = discreteQAgent(env, bins_n=24)
episodes = 500
episode_lengths, episode_rewards, Q = Q_agent.train(
episodes=episodes, eps_strategy=3)

# SAVE Q table
with open("Qtable.pkl", "wb") as pkl_handle:
     pickle.dump(Q, pkl_handle)
Q_agent.plot_running_avg(episode_rewards)
Q_agent.plot_totalrewards(episode_rewards)
```

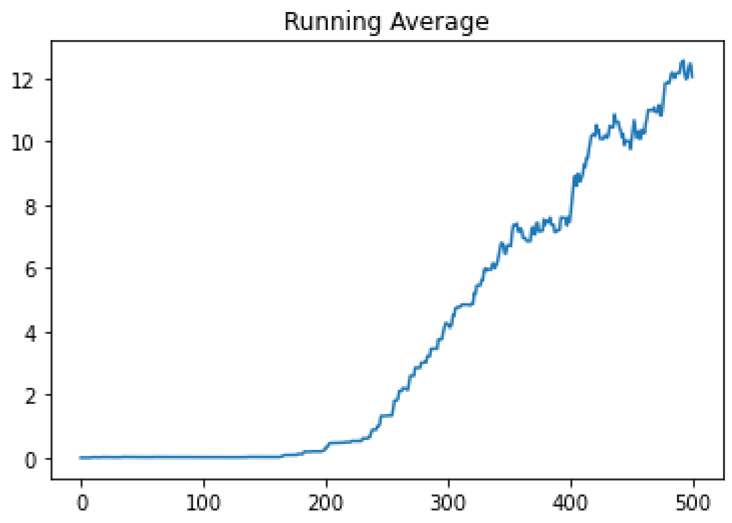

Figure 16 Running average of rewards for a total of 500 episodes

4.4.2 Simulating a trading game with discreteQAgent.

Here we have the code how to run a trading simulation using the Q-table from the previous code and generating a random process with seed=29

```
# LOAD Q table
with open("Qtable.pkl", "rb") as pkl_handle:
     Q_table = pickle.load(pkl_handle)

env = RandomWalkEnv(size=10000, random_seed=29, equity=1)
env.trim_df()
Q_agent = discreteQAgent(env, bins_n=24)

Q_agent.play_game(Q_table) # play game
Q_agent.env.disp_dataset(save_plot='Q_learning')
Q_agent.env.disp_equity(
          extra_str='accuracy: '+str(round(env.accuracy, 4)),
          save_plot='Q_learning'
          )
```

4.5 Policy Gradient

4.5.1 Training with the policy gradient algorithm

The code is training an RL agent using the policy gradient method to perform in the RandomWalkEnv environment. It aims to maximize the expected cumulative rewards by updating the policy based on the rewards obtained during training episodes. The training stops when the agent's performance meets a certain criterion regarding mean cumulative reward.

Parameter	Value
size of random process	10000
equity	1
zero_start	False
learning_rate	0.003
gamma	0.99

Table 2 Parameters on REINFORCE algorithm

The code to implement the algorithm is presented below

```
RW_LENGTH = 10000
env = RandomWalkEnv(RW_LENGTH, equity=1, zero_start=False)
score_history = []
score = 0
HIDDEN_SIZE = 256

torch.manual_seed(0)
model = torch.nn.Sequential(
             torch.nn.Linear(env.observation_space_n, HIDDEN_SIZE),
             torch.nn.ReLU(),
             torch.nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE // 2),
             torch.nn.ReLU(),
             torch.nn.Linear(HIDDEN_SIZE // 2, env.action_space_n),
             torch.nn.Softmax(dim=0)
     )

learning_rate = 0.003
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
Horizon = RW_LENGTH
MAX_TRAJECTORIES = 5000 # 25
gamma = 0.99
score = []
running_win = 500

start = time.time()
for trajectory in range(MAX_TRAJECTORIES):
    curr_state = env.reset()
    done = False
    transitions = []
    tot_reward = 0


    for t in range(Horizon):
        act_prob = model(torch.from_numpy(curr_state).float())
        # act_prob = torch.where(torch.isnan(act_prob), torch.tensor(0.5),
act_prob)
        action = np.random.choice(np.array([0,1]), p=act_prob.data.numpy())
        prev_state = curr_state
        curr_state, r1, done, _ = env.step(action)
        tot_reward += r1
        transitions.append((prev_state, action, r1))
        # transitions.append((prev_state, action, t+1))
        if done:
            break
    score.append(tot_reward)
    reward_batch = torch.Tensor([r for (s,a,r) in transitions])
gamma_powers = torch.pow(gamma, torch.arange(len(transitions)))
batch_Gvals= torch.flip(torch.cumsum(torch.flip(reward_batch, [0]) *
gamma_powers, dim=0), [0])

    expected_returns_batch = torch.FloatTensor(batch_Gvals)
    expected_returns_batch /= expected_returns_batch.max()
    state_batch = torch.Tensor([s for (s,a,r) in transitions])
    action_batch = torch.Tensor([a for (s,a,r) in transitions])
    pred_batch = model(state_batch)
    prob_batch = pred_batch.gather(dim=1, index=action_batch.long().view(-
    1,1)).squeeze()

    loss = -torch.sum(torch.log(prob_batch) * expected_returns_batch)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if trajectory > 0:
        print('Trajectory {} mean score: {:.2f}'.format(trajectory,
np.mean(score[-running_win:-1])))
    if trajectory > 1000:
        avg_score = np.mean(score[-running_win:-1])
        if avg_score > 0.12:
            break

torch.save(model, './models/reinforce_1.torch')
```

A constant RW_LENGTH is set to 10000, which is the length of a random process in the environment.

The environment is initialized with specific parameters: RW_LENGTH for the length of the random walk, equity set to 1, and zero_start set to False.

An empty list called score_history is initalized, which will be used to store scores obtained by the agent during training.

The hidden layers in the neural network to 256.

We set the random seed for PyTorch to ensure the results can be replicated.

We define a neural network model using PyTorch's Sequential module. The network consists of two linear layers (fully connected layers) with ReLU activation functions and a softmax activation in the output layer.

```
Sequential(
  (0): Linear(in_features=5, out_features=256, bias=True)
  (1): ReLU()
  (2): Linear(in_features=256, out_features=128, bias=True)
  (3): ReLU()
  (4): Linear(in_features=128, out_features=2, bias=True)
  (5): Softmax(dim=0)
)
```

The learning_rate = 0.003 for the optimizer.

An Adam optimizer is initialized to update the model's parameters during training.

The Horizon is set to the same value as RW_LENGTH, indicating the time horizon for each trajectory in the environment.

The maximum number of trajectories or episodes the agent will use for training

MAX_TRAJECTORIES = 5000: This sets the maximum number of trajectories or episodes the agent will use for training.

gamma = 0.99: The discount factor (gamma) used in the calculation of expected returns.

score = []: Initializes an empty list score to keep track of the total rewards obtained in each trajectory.

running_win = 500: Sets the window size for calculating the running average of scores during training.

The code then enters a loop that iterates through multiple trajectories (episodes) for training the agent. In each trajectory:

- curr_state is set to the initial state of the environment.
- A nested loop iterates for Horizon time steps (or until the episode ends).
- The model is used to predict the action probabilities (act_prob) based on the current state.
- An action is sampled from the action probabilities using np.random.choice.
- The action is taken in the environment (env.step(action)), and the reward is obtained (r1).
- Transitions (state, action, reward) are recorded in the transitions list.

- The episode ends if done is True (e.g., if a terminal state is reached).
- The total reward for the trajectory is accumulated in tot_reward.

score.append(tot_reward): The total reward for the current trajectory is added to the score list.

The code calculates the expected returns for each time step in the trajectory and computes the loss for policy optimization.

The model's parameters are updated using backpropagation through the loss, and the gradients are zeroed.

The code checks for a termination condition: if the average score of the last 500 trajectories (running_win) exceeds 0.12, the training loop breaks.

The training loop prints the mean score of trajectories and terminates when the termination condition is met.

The model is saved to disk

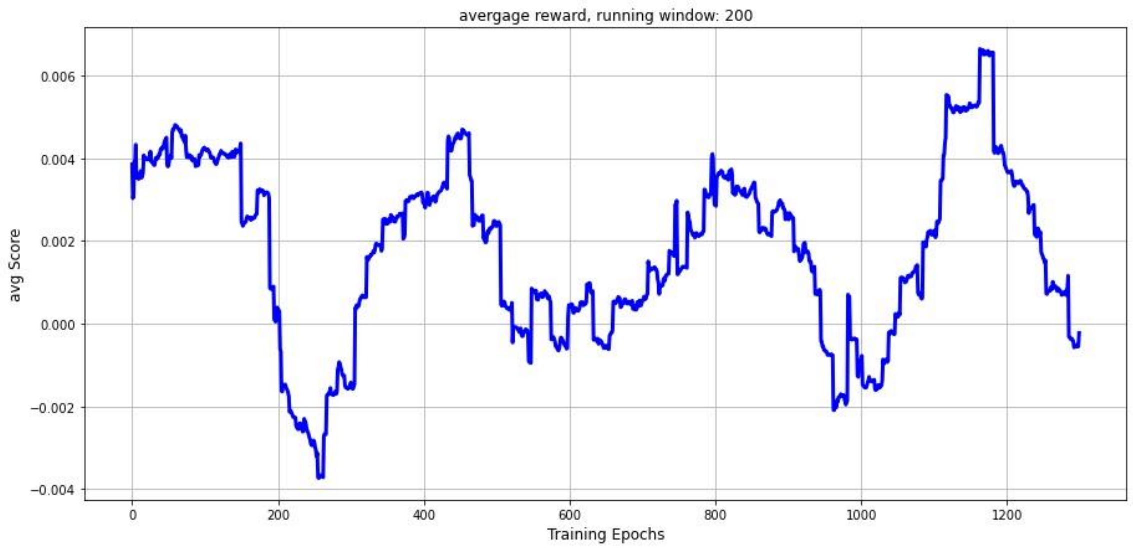

Figure 17 Average reword during training the policy gradient algorithm

4.6 Implementation and results

In this research, we will conduct a comprehensive analysis involving ten distinct random processes, each initiated with a unique seed (19, 20, 21, 22, 23, 24, 25, 26, 28, and 29). Our approach involves the application of Q-learning and REINFORCE models, both pre-trained using seed = 18, to these random processes. We will systematically evaluate the performance of these models based on multiple critical metrics.

First our evaluation will center on the Profit and Loss (PnL) generated from the resulting equity curve after applying each model. We will examine the Sharpe ratio, a key indicator of risk-adjusted returns, and assess the accuracy of these models in predicting the correct side of trades. This evaluation process will provide valuable insights into the effectiveness of each reinforcement learning algorithm in the context of trading.

To contextualize our findings, we will compare the Equity PnL of the reinforcement learning models against the performance of a standard buy-and-hold strategy applied to the underlying random processes. This comparative analysis will disclose the relative profitability and efficiency of the reinforcement learning algorithms.

Furthermore, we will delve into the unique characteristics of the equity curve, particularly focusing on drawdowns and signs that influence the associated risks. By examining these aspects, we aim to gain a deeper understanding of the risk profiles associated with each trading strategy.
Lastly, we will extend our investigation to construct a portfolio comprising all ten random processes. Within this portfolio, we will rigorously evaluate the performance of each algorithm. Additionally, we will conduct an in-depth comparative analysis between the two reinforcement learning algorithms, considering their respective strengths, weaknesses, and distinguishing attributes.

4.6.1 Sharpe ratio

The Sharpe ratio is a measure of the risk-adjusted return of an investment or portfolio. It is calculated by subtracting the risk-free rate of return from the expected return of the investment or portfolio and then dividing the result by the standard deviation of the investment's or portfolio's returns (Fernando 2023).

The formula for the Sharpe ratio is as follows:

$$Sharpe\ Ratio = \frac{R - Rf}{\sigma} \tag{4.7}$$

Where:

- R is the expected return of the investment or portfolio.
- Rf is the risk-free rate of return.
- σ (sigma) is the standard deviation of the investment's or portfolio's returns.

If the expected return (R) is less than the risk-free rate (Rf), the numerator of the Sharpe ratio will be negative. Additionally, if the investment or portfolio has a high level of volatility (measured by a high standard deviation, σ), it will also contribute to a larger denominator. Consequently, a negative numerator and a large denominator can result in a negative Sharpe ratio (Fernando 2023).

A negative Sharpe ratio indicates that the investment or portfolio is not providing an adequate risk-adjusted return, meaning that investors are not being compensated for the level of risk they are taking on. In other words, the investment is underperforming compared to the risk-free rate, after accounting for the level of risk involved. Investors typically seek investments or portfolios with positive Sharpe ratios as they represent a better trade-off between risk and return (Fernando 2023).

4.6.2 Random process with Seed=19

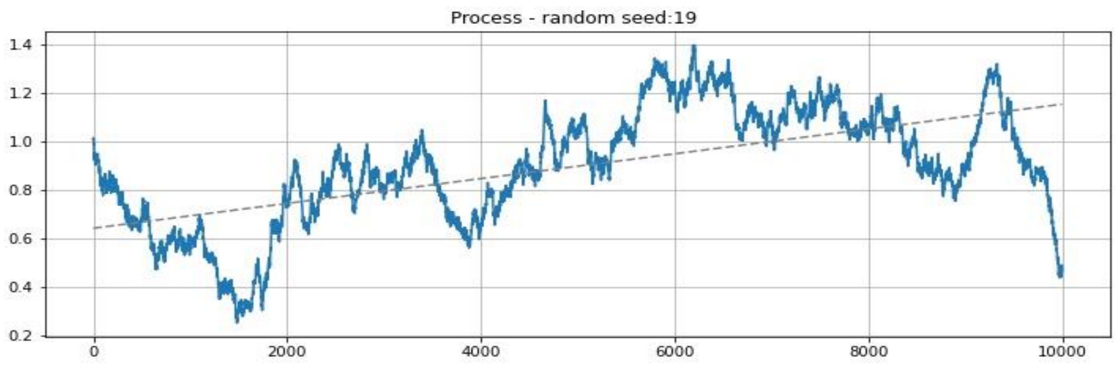

Figure 18 Random Process generated with seed=19

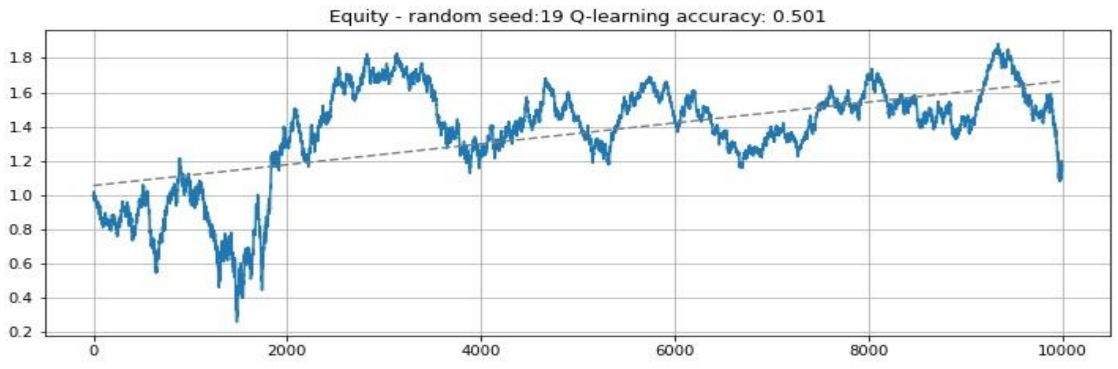

Figure 19 Equity generated applying Q-learning on RP with seed=19

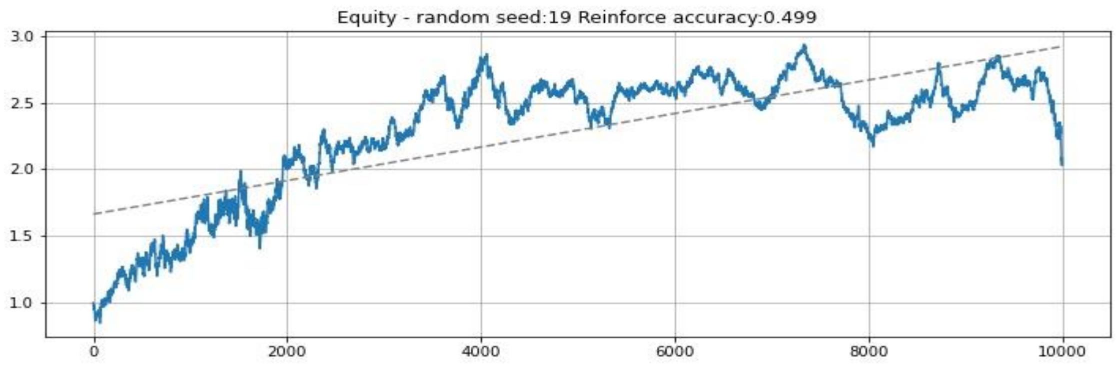

Figure 20 Equity generated applying REINFORCE on RP with seed=19

The random process (Figure 18) exhibits a loss of -0.51 in absolute terms. When comparing Q-learning and REINFORCE, Q-learning Figure 19 ends with a profit of 0.2 and a Sharpe ratio of 0.138, achieving an accuracy of 0.501. On the other hand, REINFORCE Figure 20 achieves a profit of 1.089, a Sharpe ratio of 0.22, and an accuracy of 0.499 in its predictions. Notably, in this particular seed, REINFORCE outperforms Q-learning despite Q-learning shows a slightly better accuracy.

4.6.3 Random process with Seed=20

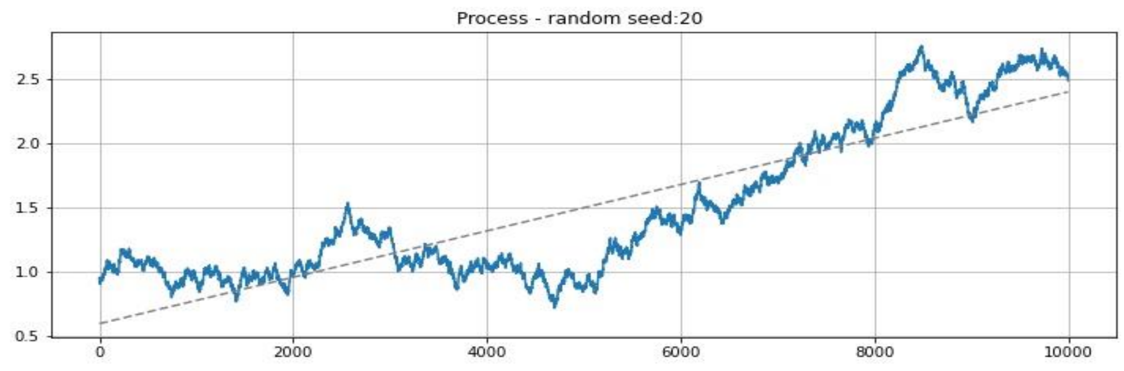

Figure 21 Random Process generated with seed=20

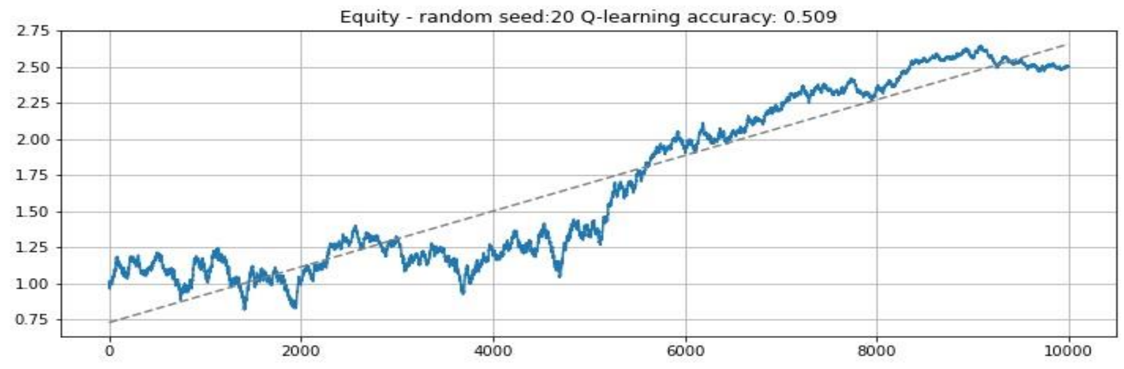

Figure 22 Returns generated applying Q-learning on RP with seed=20

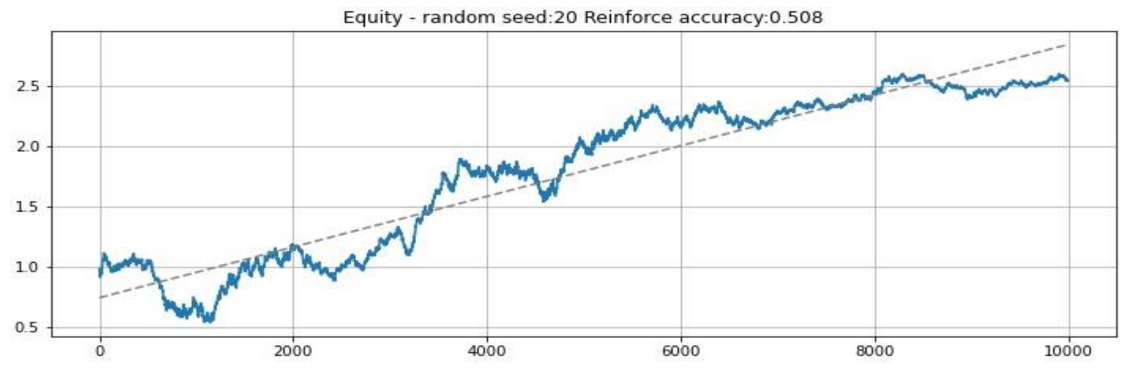

Figure 23 Returns generated applying REINFORCE on RP with seed=20

In this seed the random process ends with a remarkable profit of 1.55 in absolute value (Figure 21). Q-learning ends with a profit of 1.492 and REINFORCE with 1.599. REINFORCE slightly outperforms in profit buy and hold the random process.

4.6.4 Random process with Seed=21

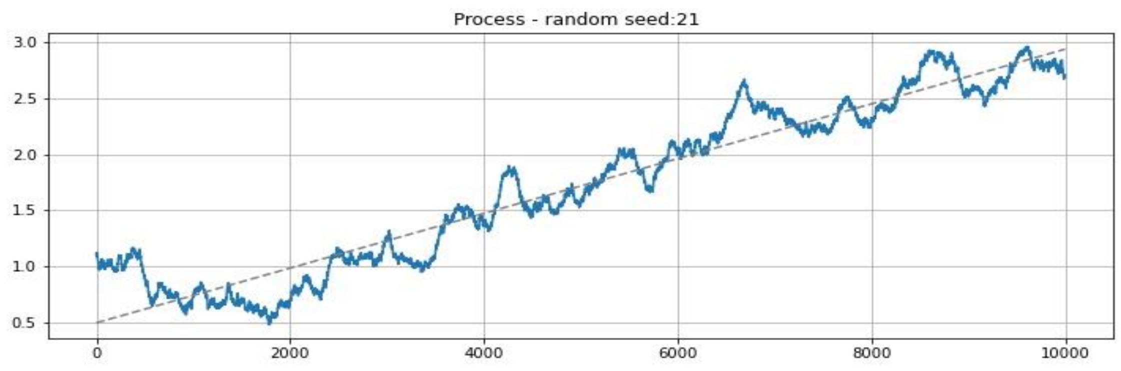

Figure 24 Random Process generated with seed=21

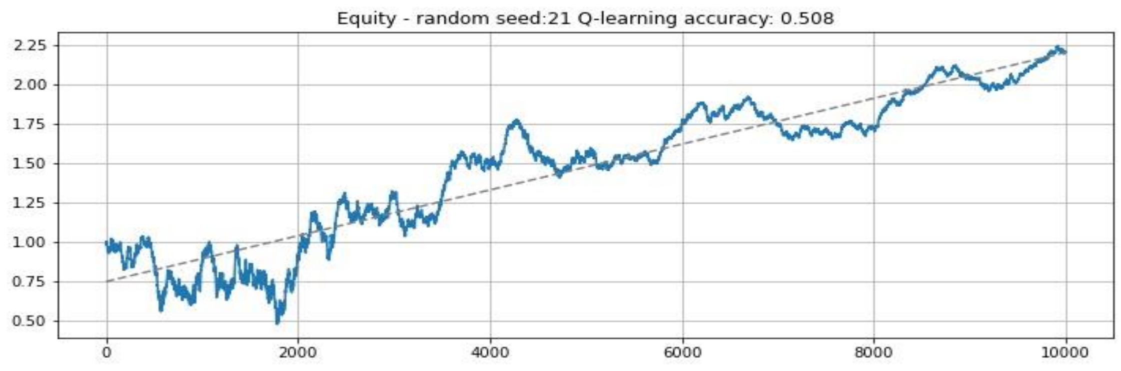

Figure 25 Equity generated applying Q-learning on RP with seed=21

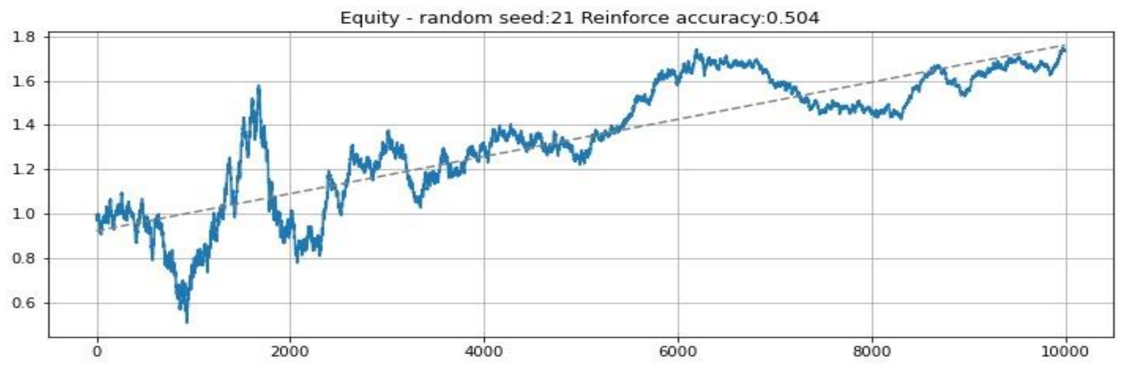

Figure 26 Equity generated applying REINFORCE on RP with seed=21

This is a strongly up-trending process (Figure 24) and buying and holding it gives a profit of 1.5945 which seems unbeatable Figure 24. Both algorithms underperform Q-learning giving 1.214 and REINFORCE 0.74. So in this case buy and hold seems to be a better choice.

4.6.5 Random process with Seed=22

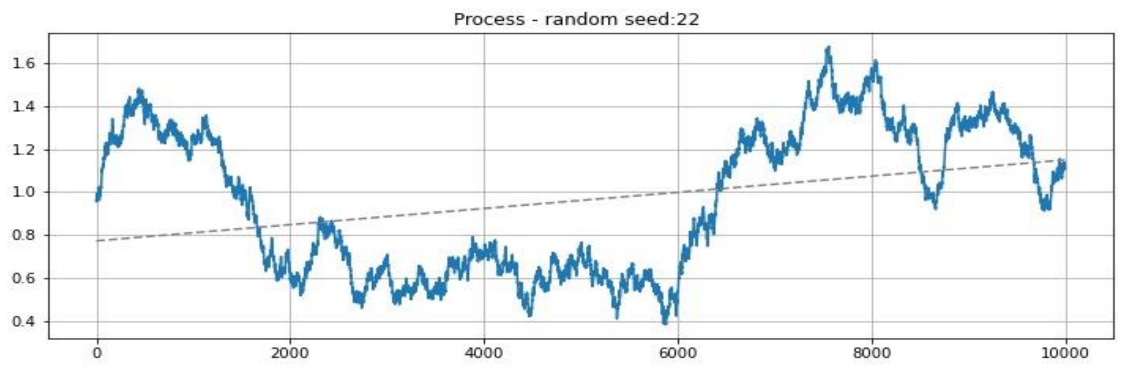

Figure 27 Random Process generated with seed=22

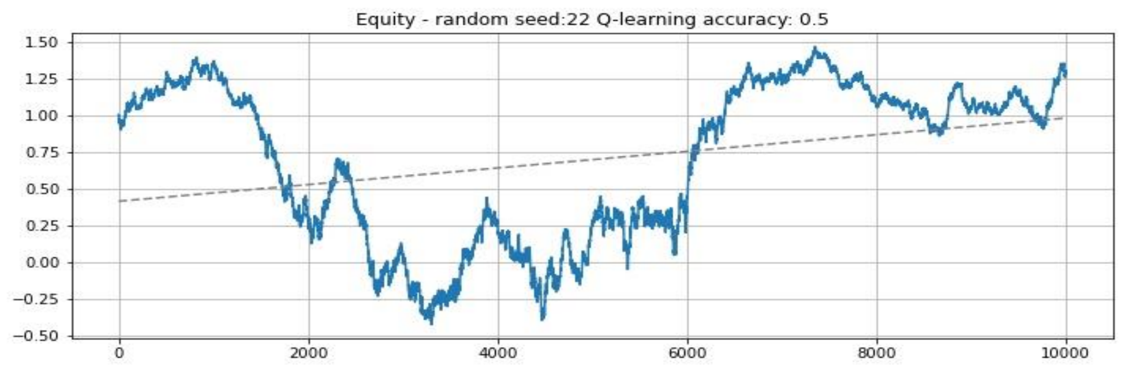

Figure 28 Equity generated applying Q-learning on RP with seed=22

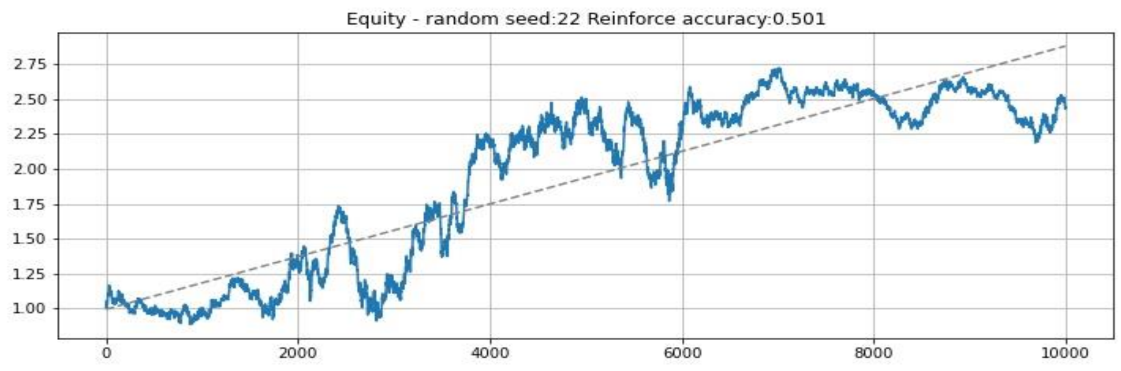

Figure 29 Equity generated applying REINFORCE on RP with seed=22

In this seed (Figure 27) buying and holding the random process gives a profit of 0.1659 whereas Q-learning gives 0.299 and REINFORCE outperforms with 1.432! In this case Reinforce seems to be the best choice.

4.6.6 Random process with Seed=23

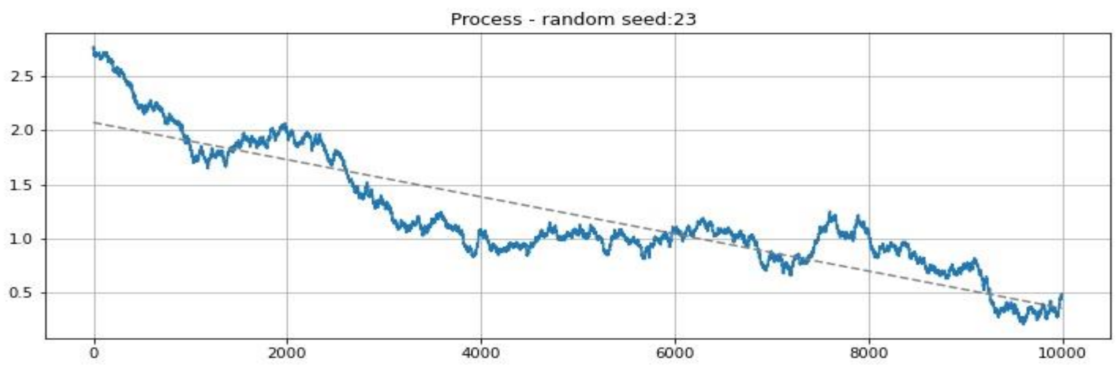

Figure 30 Random Process generated with seed=23

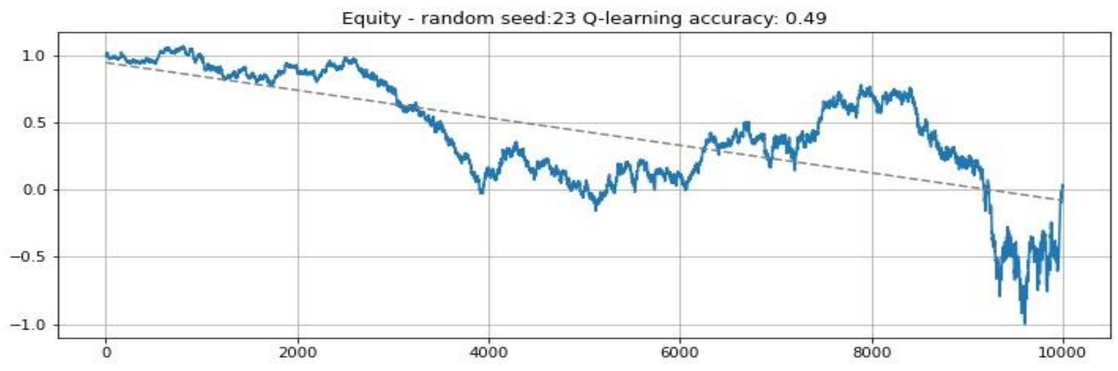

Figure 31 Equity generated applying Q-learning on RP with seed=23

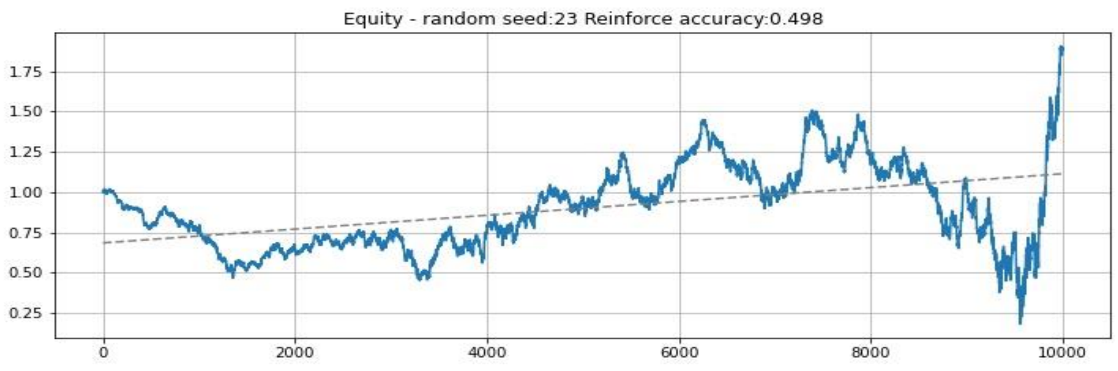

Figure 32 Equity generated applying Reinforce on RP with seed=23

A down-trending process (Figure 30) ends with a loss of -2.2792. Q-learning model does not manage to reverse the trend ending with loss. REINFORCE performs better giving a profit at the end but with severe fluctuations on its yield.

4.6.7 Random process with Seed=24

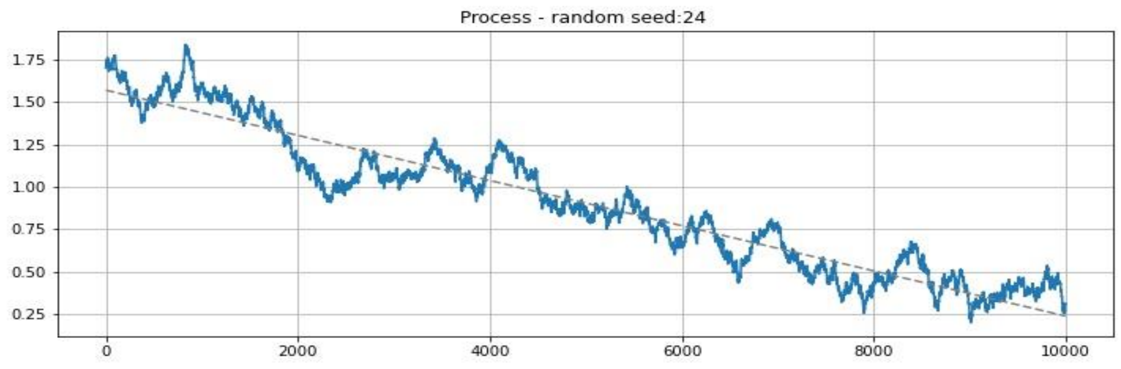

Figure 33 Random Process generated with seed=24

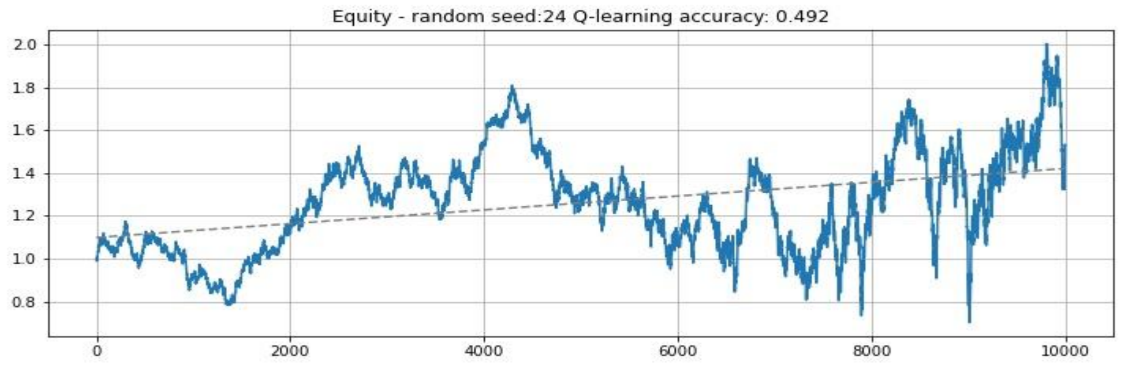

Figure 34 Equity generated applying Q-learning on RP with seed=24

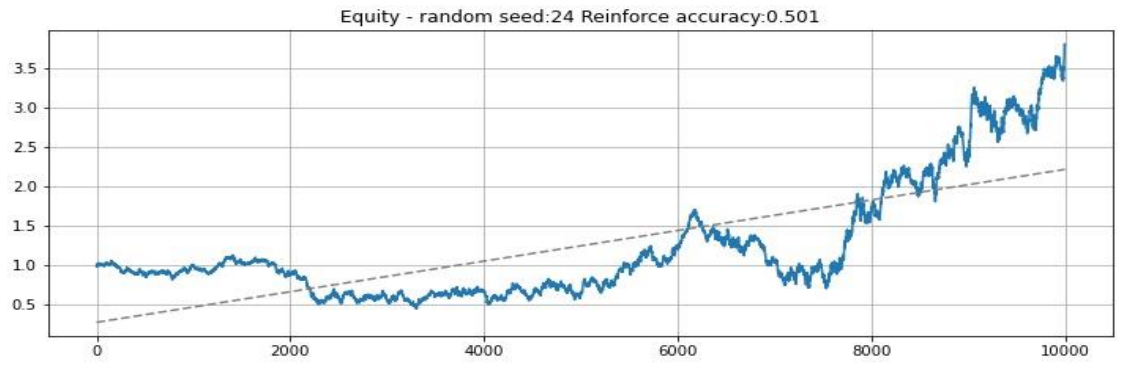

Figure 35 Equity generated applying REINFORCE on RP with seed=24

Another strong down-trending process (Figure 33) ending with -1.3904 losses. Q-learning ends with a small profit of 0.38 and severe fluctuations. REINFORCE outperforms with a profit of 2.808

4.6.8 Random process with Seed=25

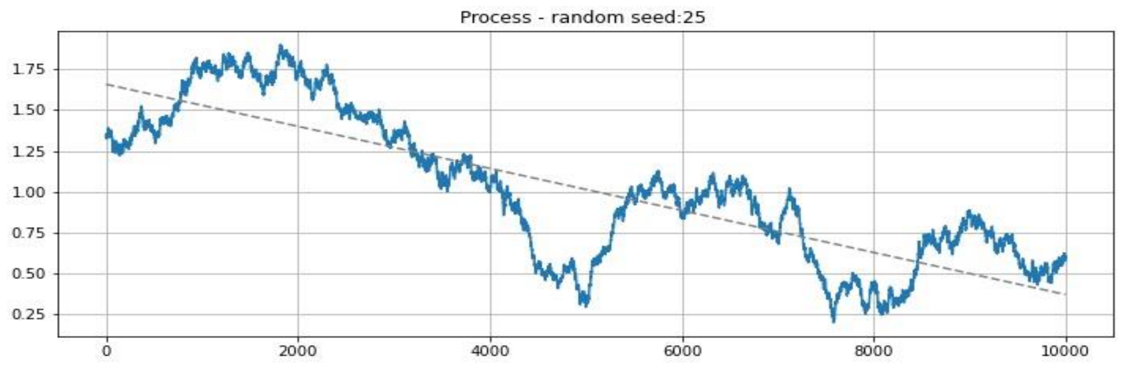

Figure 36 Random Process generated with seed=25

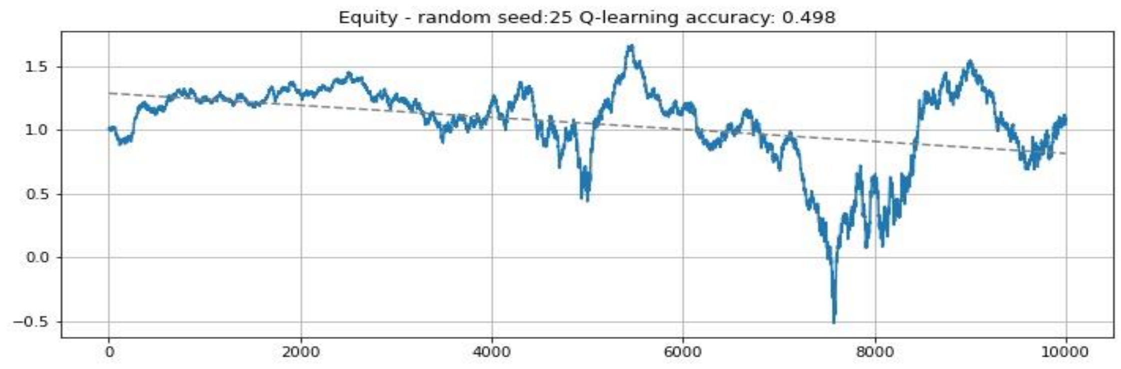

Figure 37 Equity generated applying Q-learning on RP with seed=25

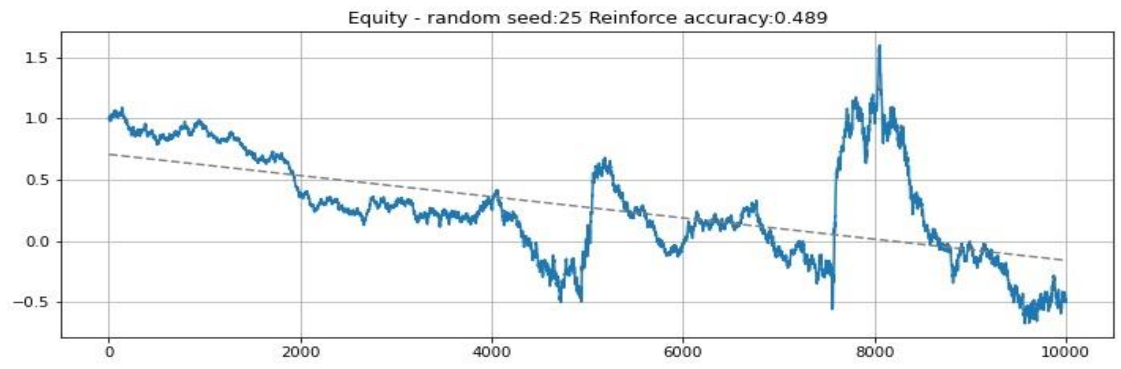

Figure 38 Equity generated applying REINFORCE on RP with seed=25

A very down-trending process (Figure 36) ending with a loss of 0.7387. Q-learning ends with a small profit but with a severe drawdown at around 7500. REINFORCE is a complete failure.

4.6.9 Random process with Seed=26

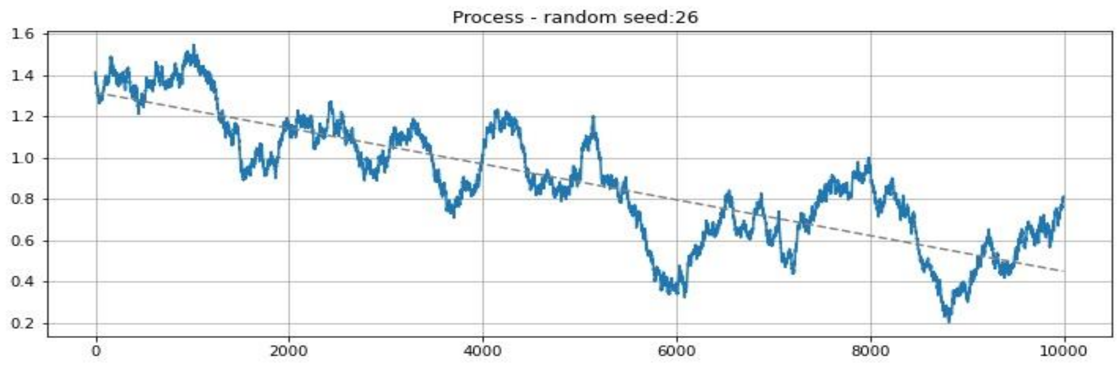

Figure 39 Random Process generated with seed=26

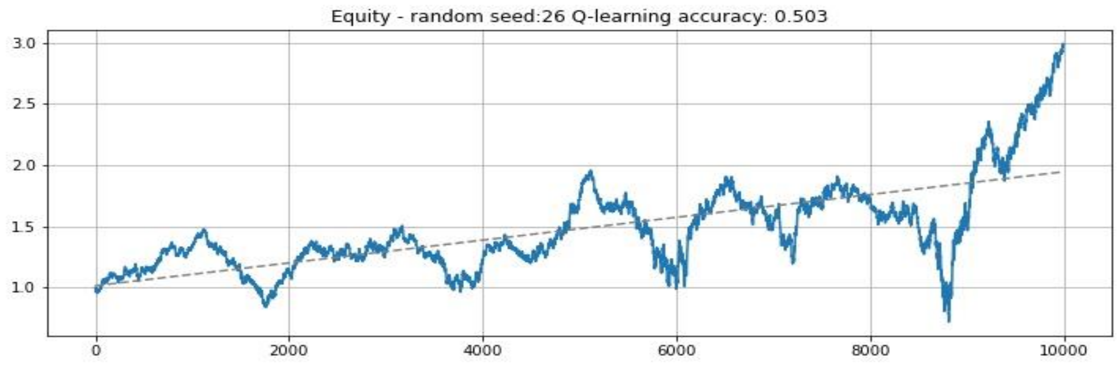

Figure 40 Equity generated applying Q-learning on RP with seed=26

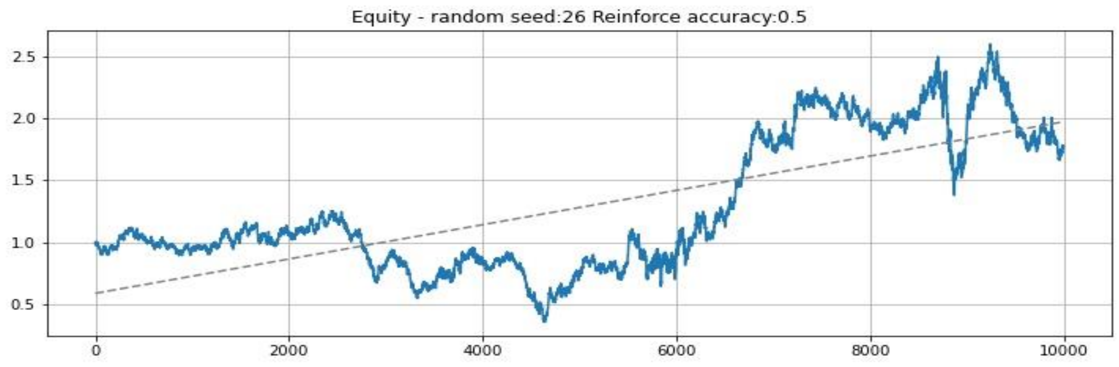

Figure 41 Equity generated applying REINFORCE on RP with seed=26

A down-trending process (Figure 39) with a loss of -0.602. Both algos succeed to reverse that trend and present profit, most notable q-learning with 1.997 and REINFORCE with 0.778. Q-learning is more stable slightly falling below original equity.

4.6.10 Random process with Seed=28

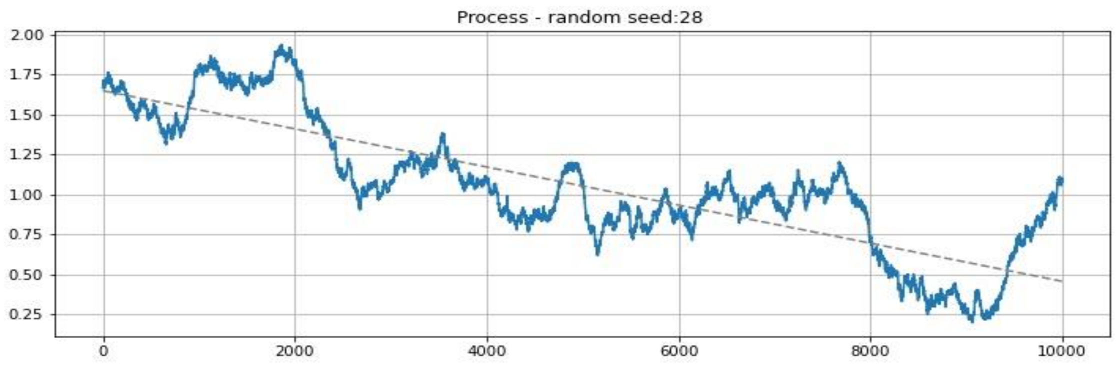

Figure 42 Random Process generated with seed=28

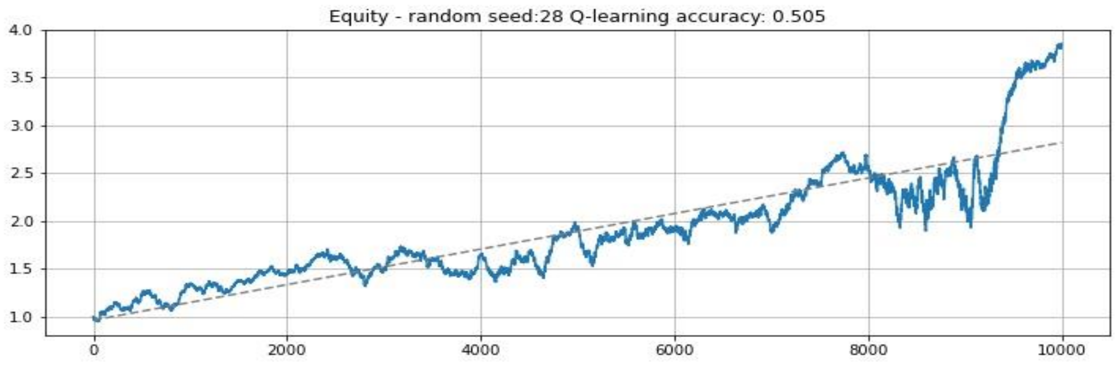

Figure 43 Equity generated applying Q-learning on RP with seed=28

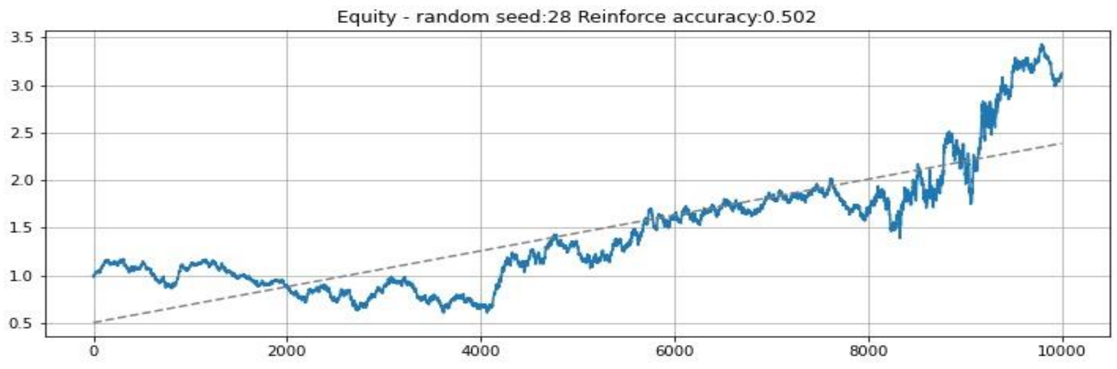

Figure 44 Equity generated applying REINFORCE on RP with seed=28

In this case we have a strong down-trending process ending with a loss -0.5707. Both algorithms show an impressive capacity to reverse this trend, ultimately concluding with substantial profits. Q-learning achieves an outstanding performance, yielding a remarkable profit of 2.857, REINFORCE also excels, securing a profit of 2.128. Throughout this process, Q-learning consistently maintains its equity above the initial level. Furthermore, Q-learning exhibits a Sharpe ratio of 0.341, surpassing REINFORCE's ratio of 0.27.

4.6.11 Random process with Seed=29

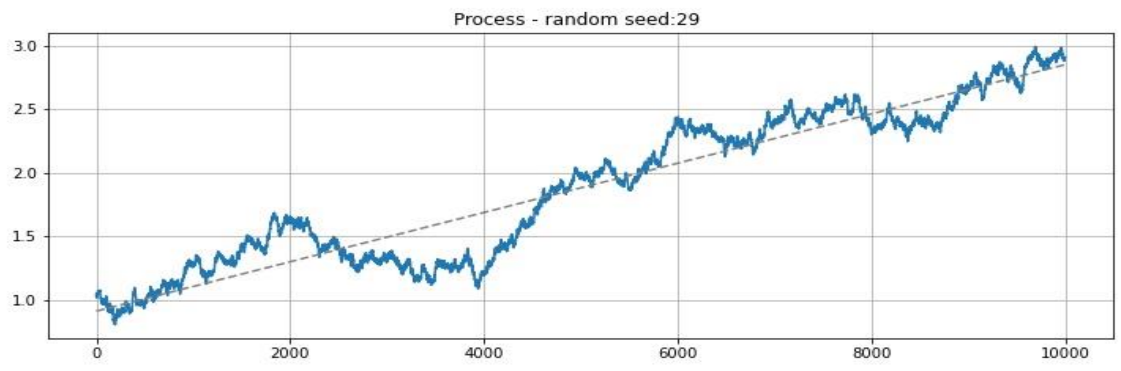

Figure 45 Random Process generated with seed=29

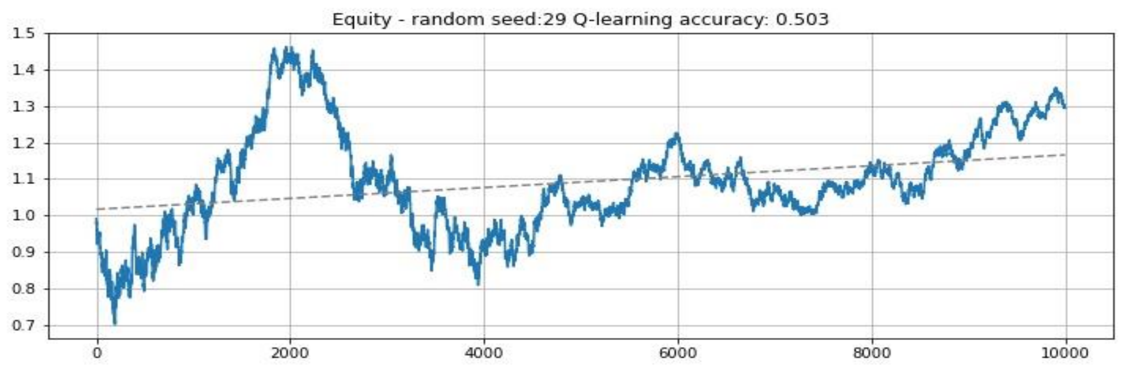

Figure 46 Equity generated applying Q-learning on RP with seed=29

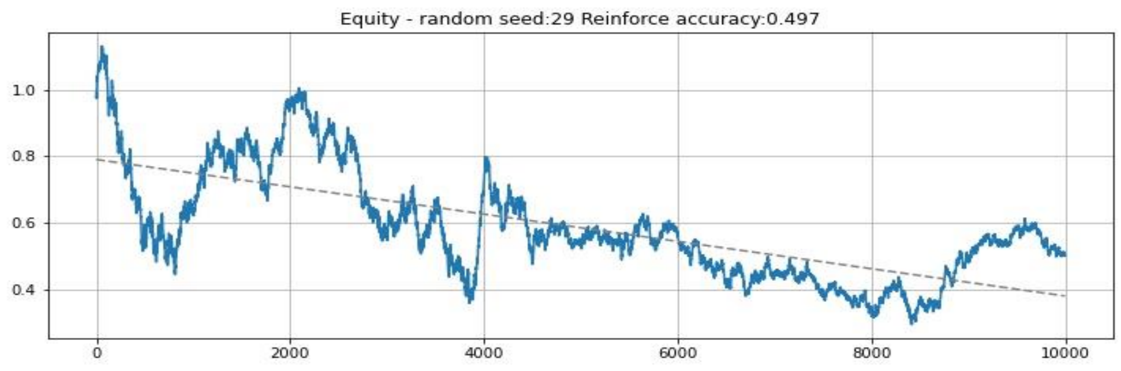

Figure 47 Equity generated applying REINFORCE on RP with seed=29

Here we have a strongly up-trending (Figure 45) case where a buy and hold strategy could yield a profit of 1.8649. Q-learning ends with a profit of 0.303 and REINFORCE fails with a loss of -0.49. So in this case the simple buy and hold works best.

4.7 Portfolio of seeds

Let’s see what would be the result if we make a portfolio of seeds that’s if we trade ten instruments seeds (19, 20, 21, 22, 23, 24, 25, 26, 28, 29) at the same time, what would be the result?

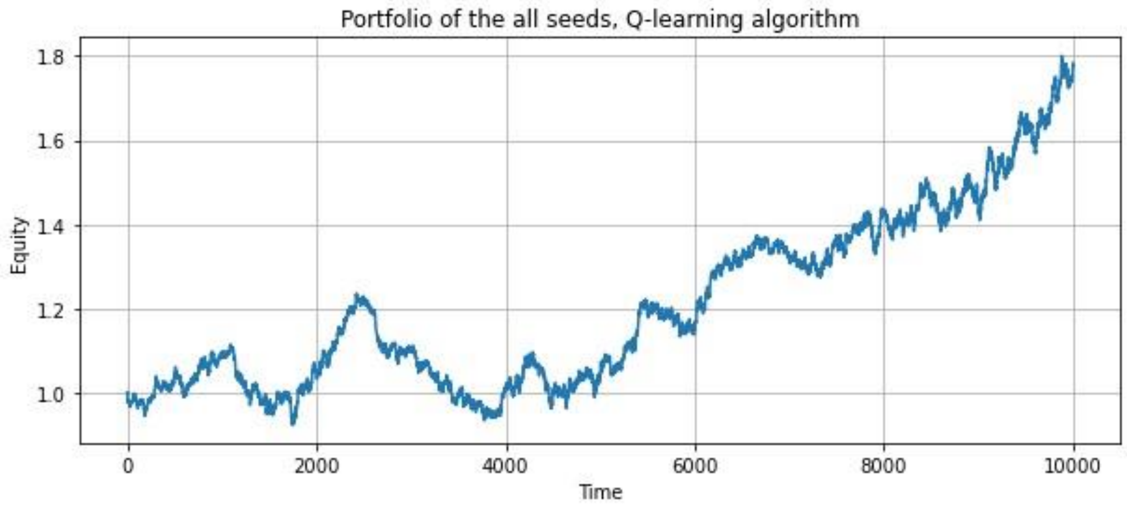

Figure 48 Portfolio of all seeds for Q-learning algorithm

The equity curve (Figure 48) is much smoother and this is very important. There are no severe draw-downs except the time between 2100 and 3900 where there is a significant drop in equity.

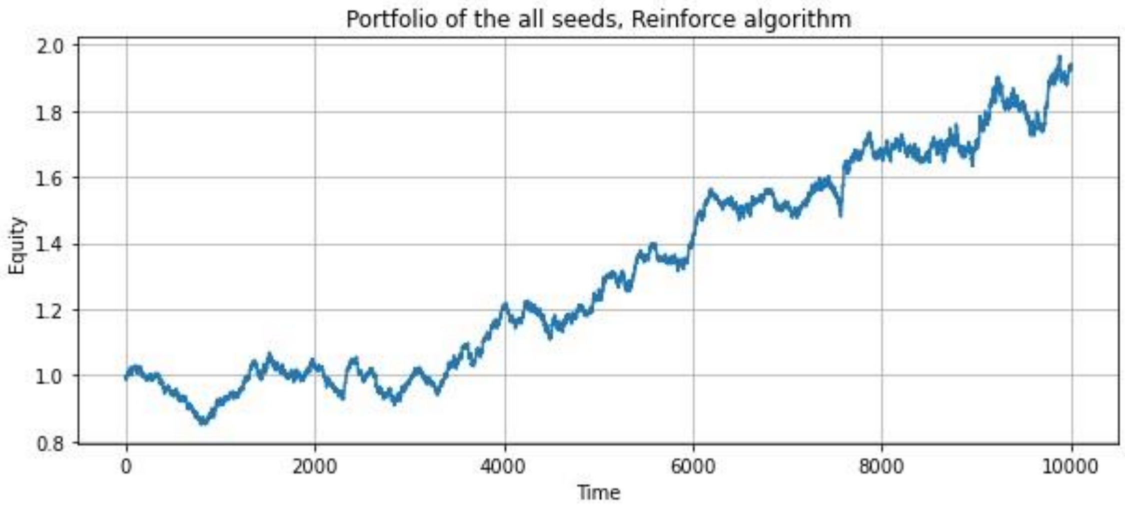

Figure 49 Portfolio of all seeds for REINFORCE algorithm

Again the equity curve (Figure 49) for the combination of the yields of results of REINFORCE is much smoother than individuals and this is much better and acceptable performance.

Let's see now a comparison of performance between the portfolios of the two algorithms (Figure 50).

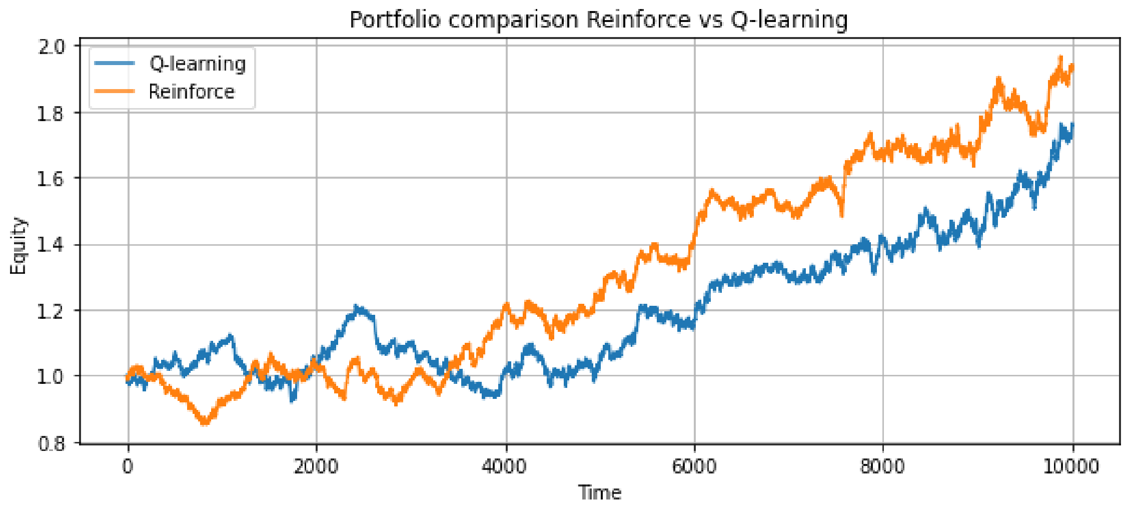

Figure 50 Comparison of portfolios between the two algorithms

The comparison shows that REINFORCE portfolio a little outperforms although in the beginning there is a small drawdown towards 0.9 in the first 500 trades. Both begin to present remarkable profit after 3000 trades and continue to profit until the end. In general there are no severe drawdowns, smooth evolution and acceptable performance.

5 Conclusions

In this study, we conducted trading simulations based on synthetic random processes with varying seeds, ranging from seed=19 to seed=29 whereas the agent's training was on seed=18. Our goal was to evaluate the performance of two prominent reinforcement learning algorithms, Q-learning and REINFORCE, in the context of trading and compare their results to a simple buy-and-hold strategy.

The agents in both cases operated within an environment where they initiated their activity from a randomly selected starting point along the synthetic random process. Throughout their interactions, the performance of these agents was continually monitored and assessed. In cases where an agent exhibited mediocre performance, the environment promptly terminated the ongoing trajectory by signaling 'done=False.' This dynamic evaluation process allowed for the exploration of various segments of the synthetic process, ensuring that the agents' strategies were rigorously tested under different conditions and performance scenarios.

Seed		Q-learning			REINFORCE			
No	**RW p/l**	**Sharpe ratio**	**Accuracy**	**p/l**	**Sharpe ratio**	**Accuracy**	**p/l**	**Best on p/l**
19	-0.5196	0.138	0.501	0.2	0.22	0.499	1.089	REINFORCE
20	1.5501	0.272	0.509	1.492	0.272	0.508	1.559	REINFORCE
21	1.5945	0.209	0.508	1.214	0.173	0.504	0.74	Buy and Hold
22	0.1659	0.073	0.5	0.299	0.238	0.501	1.432	REINFORCE
23	-2.2792	-0.303	0.491	-1.046	0.2	0.498	0.88	REINFORCE
24	-1.3904	0.156	0.491	0.38	0.267	0.501	2.808	REINFORCE
25	-0.7387	0.095	0.498	0.071	0.126	0.489	-1.485	Q-learning
26	-0.602	0.247	0.503	1.997	0.173	0.5	0.778	Q-learning
28	-0.5707	0.341	0.504	2.857	0.27	0.502	2.128	Q-learning
29	1.8649	0.117	0.503	0.303	-0.014	0.497	-0.49	Buy and Hold

Table 3 Results of trading simulations on various random processes

The key findings from our trading simulations:

Q-learning vs. REINFORCE: Analyzing the performance of the two algorithms, we observed variations in their outcomes across different seeds. While REINFORCE outperformed Q-learning in terms of profit and loss (P/L) in most cases (five to three), the results were not consistent across all seed values. This suggests that the effectiveness of these algorithms can vary depending on the underlying random process.

Best Strategy by P/L: When considering the best-performing strategy based on P/L, REINFORCE seemed as the superior choice for most seed values. This indicates that REINFORCE was better at capitalizing on the fluctuations in the synthetic random processes to generate profits.

Buy and Hold: Interestingly, for specific seeds (e.g., seed=21 and seed=29), a simple buy-and-hold strategy outperformed both Q-learning and REINFORCE in terms of P/L. This result emphasizes the significance of exploring and evaluating alternative trading strategies, as they may, at times, deliver comparable or even superior outcomes.

Risk and Sharpe Ratio: Additionally, we calculated the Sharpe ratio for both Q-learning and REINFORCE. While REINFORCE exhibited higher Sharpe ratios for some seeds, Q-learning demonstrated consistency in achieving positive Sharpe ratios, implying a more stable risk-adjusted performance.

Accuracy: Evaluating the accuracy of the reinforcement learning models the results showed that accuracy levels were generally slightly above 0.5, indicating that both Q-learning and REINFORCE had little predictive power in identifying profitable trading opportunities.

In summary, our trading simulations revealed that the performance of reinforcement learning algorithms, particularly Q-learning and REINFORCE, can be influenced by the stochastic nature of the underlying data. REINFORCE demonstrated an advantage in terms of P/L for most seeds, but Q-learning displayed greater consistency in risk-adjusted returns. Furthermore, the surprising effectiveness of a buy-and-hold strategy for specific seeds emphasizes the importance of considering a variety of approaches in trading scenarios.

These findings underline the dynamic nature of algorithmic trading, where the choice of the optimal strategy can depend on the specific characteristics of the underlying data. Future research could explore additional algorithmic trading strategies, data preprocessing techniques, and hyper-parameter tuning to further enhance the performance of reinforcement learning models in this domain.

In general, the policy based algorithm generalizes better and achieves smoother performance equity curves; something desirable perhaps due to its nature (it uses NN). He also chooses action, always acting stochastically. Highly stochastic environments are characterized by significant uncertainty and randomness in the outcomes of actions, which can make learning and decision-making challenging. Policy-based methods are often well-suited for highly stochastic environments. They can learn probabilistic policies that explicitly account for uncertainty. By parameterizing policies, they can model the distribution over actions and adapt to stochastic outcomes (Sewak 2019).

6 Future Research

Designing an effective reward function is crucial for training a reinforcement learning (RL) agent for trading. Using profit or loss after a position closes is a reasonable starting point, but there are ways to make it more efficient and informative. Here are some points to consider:

- *Market Benchmark:* Compare the agent's performance to a benchmark, like a buy-and-hold strategy on a market index. This can help the agent learn to outperform a passive strategy, which is often the goal in trading.
- *Learning from Drawdowns:* We could give a negative reward when the agent incurs significant drawdowns. This encourages the agent to learn from its mistakes and avoid risky strategies that could lead to large losses.

It's important to strike a balance between creating a reward function that encourages desired behavior and not overcomplicating it, as overly complex reward functions can lead to training instability. We can experiment with different reward functions and monitor the agent's performance closely to find the best approach.

- *Cumulative Returns:* Reward the agent based on the cumulative returns over a sequence of trades. This can help in reducing the problem of sparse rewards. Reward could be the total cumulative profit or loss over a given time horizon.
- *Custom Metrics:* Define custom metrics or composite rewards that consider other factors specific to our trading strategy, such as drawdown, winning streaks, or losing streaks.

Constructing an informative state representation is crucial in reinforcement learning. An effective state should capture relevant information about the market that the agent can use to make informed decisions. Here are some additional considerations:

Feature engineering:

- Various percentage change inputs
- Various rolling windows for Mean value
- Various rolling windows for standard deviation
- Convolution 1d
- FFT inputs

Feature Importance Analysis:

We can use techniques like SHAP (SHapley Additive exPlanations) or feature importance scores from various machine learning algorithms to assess which features are most informative. Evaluate all of them with SHAP and selecting the best.

Dimensionality Reduction:

In case there is a high-dimensional state space, we may consider dimensionality reduction techniques like Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce noise and redundancy in your state.

State Stacking:

Consider stacking multiple historical states to provide the agent with a sense of market history and trends over time.

The choice of state representation can significantly impact the learning process. It's important to strike a balance between providing enough information for the agent to make informed decisions and keeping the state space manageable to avoid the curse of dimensionality.

Bibliography – References – Online sources

Bellman, R. *Dynamic programming.* Princeton University Press, 1957.

Box, G. E., G. M. Jenkings, and Reinsel G. C. *Time Series Analysis: Forecasting and Control.* John Wiley & Sons, 2015.

Brockman, Greg, et al. "OpenAI Gym." *arXiv,* 2016: arXiv.1606.01540.

Brockwell, P. J., and R. A. Davis. *Introduction to Time Series and Forecasting.* Springer, 2002.

Campbell, J. Y., A. W. Lo, and A. C. MacKinlay. *The Econometrics of Financial Markets.* Princeton University Press, 1997.

Chen, Sihang, Weiqi Luo, and and Chao Yu. "Reinforcement Learning with Expert Trajectory for Quantitative Trading." *Arxiv,* 2021: 2105.03844.

Fernando, Jason. *Sharpe Ratio: Definition, Formula, and Examples.* 5 11, 2023. https://www.investopedia.com/terms/s/sharperatio.asp.

Gao, Xiang. "Deep reinforcement learning for time series: playing idealized trading games." *arXiv,* 2018.

Huang, Chien-Yi. "Financial Trading as a Game:A Deep Reinforcement Learning Approach." *Arixiv.org,* 2018.

Jeong, G., and H.Y. Kim. "Improving financial trading decisions using deep Q-learning: Predicting the number of." *Expert Systems with Applications,* 2019: 125–138.

Lei, K., B. Zhang, Y. Li, M. Yang, and Y Shen. "Time-driven feature-aware jointly deep reinforcement learning." *Expert Systems with Applications,* 2020: 140, 112872.

Li, X., Y. Li, Y. Zhan, and X.Y. Liu. "Optimistic bull or pessimistic bear: Adaptive deep reinforcement learning." *arXiv,* 2019: arXiv:1907.01503.

Li, Y., W. Zheng, and Z. Zheng. "Deep robust reinforcement learning for practical algorithmic trading." *IEEE Access,* 2019: 108014–108022.

Liang, Z., H. Chen, J. Zhu, K. Jiang, and Y Li. "Adversarial deep reinforcement learning in portfolio management." *arXiv,* 2018: arXiv:1808.09940.

Liu, Xiao-Yang, Zhuoran Xiong, Shan Zhong, Hongyang Yang, and Anwar Walid. "Practical Deep Reinforcement Learning Approach for Stock Trading." *arXiv,* 2022: 1811.07522.

Raffin, Antonin. *Stable-Baselines3: Reliable Reinforcement Learning Implementations.* n.d. https://araffin.github.io/post/sb3/.

Schulman, J., S. Levine, P. Abbeel, M. Jordan, and P. Moritz. "Trust region policy optimization." *arxiv.org*, 2015: 1502.05477.

Sefidian, Amir Masoud. *REINFORCE Algorithm explained in Policy-Gradient based methods with Python Code.* n.d. https://www.sefidian.com/2021/03/01/policy-g/.

Sewak, Mohit. "Policy-Based Reinforcement Learning Approaches." *Springer,* 2019: 127–140.

Sutton, R. S., and A. G Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2018.

Williams, R. J. *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Springer, 1992.

Appendix A

Code in github

Appendix B

……