The Merton Problem using model free Reinforcement Learning

John Goodacre
Supervisor: Professor Jun Wang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Research

in

Financial Computing.

Department of Computer Science

University College London

August 9, 2018

Code for this project, can be found in the following public repository:

https://github.com/jsg71/MRes-Thesis

Abstract

The Merton problem [25] asks the question as to how one should allocate one's wealth across assets and throughout time. The goal being to maximise the expected utility of terminal wealth at some distant time step T, $\max \mathbf{E}[U(w_T)]$.

As one might imagine this is an important problem both theoretically within academia and practically within the investment arena. Academically we have seen major contributions from Nobel prize winners such as Markowitz [24], Sharpe and of course Merton [25]. Markowitz and Sharpe developed the theory for the simpler single period asset allocation problem, here decisions are myopic, that is take the best action for that particular time period and repeat.

Merton's problem is more sophisticated, this is a multiple-period investment problem where all possibilities over future time periods must be taken into account at each step. The fact that we are dealing with 'utilities' is important, if we simply cared about maximising our expected final wealth, $\mathbf{E}[w_T]$ then the problem collapses into the simpler single period problem, but one where risk is not taken into account and which would only suit the 'risk neutral' investor with a linear utility curve, (or stated less politely the 50/50 gambler).

Merton's famed work [25] provides an analytical solution to this problem using techniques from stochastic optimal control under certain restrictive assumptions, however it also requires prior knowledge of both the nature of the risky process and its parameters. Given this prior knowledge it is possible to derive a solution for specific well-behaved stochastic processes given a sensible utility curve, but what if we do not know these things? and what if our assumptions about the underlying market process are simply wrong?

This dissertation addresses that question, specifically can we teach an agent to trade close to Merton optimality purely by interacting in a step-wise manner with a market environment, solely from the data, with no prior knowledge. The advantage if so is that regardless of the underlying stochastic process, its parameters or the particular utility curve, the RL agent can simply learn and adapt according to its market interactions. Model free in this context means the agent has no knowledge of the environment bar the rewards received from interactions with it, this is not to be confused with the use of models within reinforcement learning itself, where the agent can derive its own model of the environment as it gains experience.

The work thus combines elements of both finance and machine learning. In machine learning I use techniques from Reinforcement Learning (RL), first scaled down tabular methods such as Hado Van-Hassalt's Double-Q learning [19], Double-Sarsa and Rich Sutton's Dyna, [48], I then introduce more sophisticated neural network function approximators, such as Deep-Q [27], Double Deep-Q [52], and Noisy Nets [13]. On the finance side, my agents require step-wise rewards from their interactions with the market environment and a key issue is shaping and deriving these stochastic rewards, they are not simply changes in wealth. I begin with a simple derivation specific to Log-Utility and then generalise following work from Ritter [35] in 2017, where he uses a parameterised mean variance approximation to trade an Ornstein-Uhlenbeck [50] process.

Finance is very noisy, so rather than looking at some expected profit or loss, where even over multiple episodes a random agent can outperform a merton optimal one, I instead introduced a test metric examining the entire distribution of utilities (plus rewards, wealth and Sharpe ratios) between three agents, a merton optimal, a random agent and a trained RL. The hope being that the trained RL could closely match the test distribution of the merton optimal agent over millions of episodes.

Starting with lower dimensional settings I found that RL agents could learn to approximate merton optimality and indeed be trained towards specific risk-aversion behaviours, also my initial steps to increase realism (and thus dimensionality) were achieved successfully by adding neural networks of varying degrees of sophistication.

Finance has proven to be a difficult setting for machine learning, noise levels can be high and vary in a heteroscedastic manner. RL looks to be a natural setting to tackle typical finance problems and in this thesis agents were indeed capable of learning even with very high levels of market volatility and fat tails. However as noise increases the difference in distribution between even a random agent and a merton optimal agent reduces as the distributions themselves increasingly overlap. I found that many episodes of training were required for the agent to learn and believe further work on sample efficiency in reinforcement learning for noisy environments could prove beneficial.

Acknowledgements

Many thanks to Professor Jun Wang for his guidance within this project. Not least for encouraging me to look at Merton's problem in the context of Reinforcement Learning in the first place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter I give an introduction to the problem being discussed, and the motivation behind it. I then lay out the specific questions being asked before outlining the structure of the thesis and the key literature related to the work.

1.1 Motivation

The problem of how one should allocate between risky assets over multiple time-periods is highly non-trivial. Merton's work [25] derives a solution for maximising the expected utility of future wealth $\mathbf{E}[U(w_T)]$ where he allocates in continuous time between a stochastic process (for example a geometric brownian motion with a known volatility and drift) and a risk-free bond, also given a known utility function U. The Merton problem has analytical solutions for only a few stochastic processes, and requires both considerable mathematical machinery and also prior knowledge of the processes and parameters. It should be mentioned that he takes his work further by also introducing optimal consumption, however within this thesis our viewpoint is that of a portfolio manager who wishes solely to maximise the expected utility of future wealth on behalf of clients, so I am not addressing consumption.

Although Merton provides an analytical solution to this problem under certain assumptions and given prior knowledge, this may be an over-optimistic assumption in real life. It is known that real markets do not follow a geometric brownian motion and exhibit 'fat tails', it is also known that markets are heteroscedastic, parameters may be unstable though time. We can indeed assume a prior risky process, and estimate its parameters, but in some ways this is unsatisfying and besides our prior assumptions may simply be wrong. Therefore the motivation behind this thesis is to create a machine learning agent that interacts with the market but with none of this prior information and yet can still learn to trade near Merton optimality purely through realistic step-wise interactions.

The advantage if we can achieve this is that the agent can learn to approximate Merton optimal behaviour for a variety of utility curves and random processes simply by interacting with the market in an online manner. The further advantage is that in the case where the underlying stochastic is unknown - Merton is either silent or ill-specified, whereas the RL agent will adapt and learn from whatever data is given.

In terms of outcomes, a trained agent will be capable of learning solely from its interactions with the market, but also because of the nature of the Merton problem, it does this taking into account multiple time-periods and automatically incorporating risk averse behaviour due to our incorporating utilities.

Thus success enables a more general solution than a particular analytical solution (due to it being model free) and also more general than for example a Monte-Carlo simulation, due to learning occurring on a step-wise online basis, rather than waiting until the end of an episode before learning.

1.2 Structure of the thesis

The thesis covers elements from finance as well as machine learning. In finance the emphasis is on stochastic processes and utility curves as well as their solutions. In machine learning the focus is on reinforcement learning. In terms of cross-over between the two the main work lies in reformulating the end of period expected utilities into step-wise rewards suitable for an RL agent. I will quickly lay out the key points and take-aways from each chapter.

1.2.0.1 Chapter 2 - Finance Background

This chapter covers the basic finance background and intuitions necessary for the rest of the thesis.

Key Subject	Key Take-away
Portfolio theory	Motivates Merton's problem
Data Representations	Mathematical introduction of data and stochastic processes used
Utility Curves	Provides intuition on utilities, risk aversion and neutrality
Merton's Problem	Mathematical solutions for Merton optimality (Log/Power Utility)

Table 1.1: Chapter 2: Finance Background

1.2.0.2 Chapter 3 - Machine Learning Background

This chapter provides a background to reinforcement learning as well as introducing the algorithms and techniques used in later experiments.

Key Subject	Key Take-away
RL	Basics from Markov Decision Processes to Bellman
RL	Tabular RL, Double-Q/ Dyna and why DP and Monte Carlo are unsuitable
Advanced RL	Deep Learning and Double Deep Q learning
Advanced RL	Advanced exploration, prioritised experience replay and Noisy Networks

Table 1.2: Chapter 3: Machine Learning Background

1.2.0.3 Chapter 4 - Implementation

This chapter provides a detailed explanation of my implementation, covering the design of the agents, the environment and how rewards were shaped. Finally a motivation and an explanation of the test methodology is given, particularly focusing on the noise problem.

Key Subject	Key Take-away
Reward Shaping 1	Derivation of discretised rewards for Log-Utility
Reward Shaping 2	Episodic Mean-Variance equivalence, intuition
Reward Shaping 3	Step-wise Mean-Variance equivalence, reward formulation
Design	Explanation of Market and RL Agents designs
Test Metrics 1	Why realistic noise levels make testing non-trivial
Test Metrics 2	Explanation of the four test metrics

Table 1.3: Chapter 4: Implementation

1.2.0.4 Chapter 5 - Experiments

This chapter covers three groups of experiments. First a double-Q learning agent in a simple setting, volatility is low and rewards are exact (due to using Log-Utility). Second, complexities are gradually added until we have an agent capable of learning various risk aversions in a high volatility fat-tailed market at levels comparable to Merton optimality. Third, I use deep learning to move beyond ’buckets’ of wealth or prices. The neural net receives ’features’ and gives a prediction of the agent actions. This enables the agent to cope with high dimensional state spaces and adds greater realism. Finally I briefly examine more sophisticated models looking at the exploration problem and sample efficiency.

Key Subject	Key Take-away
Group 1 - Learning Merton	
Double-Q Log-Ute	Low vol, exact reward - simple setting
Double-Q Log-Ute	Medium vol, exact reward - increasing realism
Double-Q MV equiv	Generalisation to mean-variance approximation
MV equiv, Power Ute	Testing the agent on a different utility curve
Learning risk aversion	Learning a variety of risk behaviours
Group 2 - Adding realism	
Market realistic parameters	Learning closer to market realistic rates
Fat Tails	The agent more than matching Merton optimality
Group 3 - Adding model power	
Other tabular agents	Dyna and Double Sarsa results
Deep-Q Learning	Success using a neural net function approximator
Double Deep-Q Learning	Success using Double Deep-Q
Noisy net Deep-Q	More sophisticated exploration, fewer episodes

Table 1.4: Chapter 5: Experiments

1.2.0.5 Chapter 6 - Conclusions

Thesis conclusions and ideas for further work.

1.3 Literature Review

In this section I shall give a brief over-view of key literature related to areas in this thesis. This is by no means comprehensive on each of the following subjects.

1.3.1 Finance

The problem setting is a multi-period asset allocation problem, where we wish to maximise the expected utility of terminal wealth. Utility theory is itself a major subject in economics and psychology and held up as a standard of rational behaviour. The underlying theorems go all the way back to the classic work by Von Neumann and Morgenstern [54].

In terms of portfolio construction, Markowitz [24] first motivated and developed the mean-variance approach to portfolio construction, this is easier than the utility approach and is utilised by the actual finance community but also not without its flaws. The theoretical being 'Borch's paradox' where it is actually impossible to draw indifference curves in the mean-variance plane [5], and the practical being that the Markowitz paradigm (and extensions such as from Sharpe who developed the CAPM [42]) comprise a single period optimisation. Also from a practitioners viewpoint there are implementation issues, the scope of which are outside this thesis but good discussions may be found in Grinold and Kahn [16], or in Ang's excellent book on factor investing [1], with a variety of proposed solutions such as risk-parity, Roncalli [36].

The financial angle of this thesis however follows the multi-period problem introduced and solved by Merton [25], a good recent overview of both single and multi-period portfolio choice problems may be found in Chapados [9].

The analytical solution to the Merton problem borrows from areas of financial mathematics, in particular stochastic calculus and stochastic optimal control. Here there are many good works such as Bjork [4], Shreve [44] or Oksendal [31]. A recent book covering stochastic optimal control applied to not just the Merton problem, but also more sophisticated applications such as statistical arbitrage and optimal liquidation problems may be found in Cartea et al, [6], this actually covers much of the material necessary for the financial section of this thesis.

1.3.2 Machine Learning

Machine learning covers a very wide church indeed, and there are various excellent introductions to the broad area such as Hastie et al. [20], Bishop [3] and Murphy [29]. For the Merton problem I shall be using Reinforcement Learning, of which the best current exposition I have found is in Sutton and Barto [49], and also their new addition not yet printed but available online. Many of my early experiments will be using an off-policy reinforcement learning algorithm, Q-learning developed by Watkins [55], although I use a less biased extension, Double-Q learning [19]. The advantage of Q-learning from the point of view of the Merton problem and an initial proof of concept is that it is one of the simplest model-free algorithms where the agent can learn in a purely step-wise manner. It is off-policy in that its actions comprise sometimes exploring randomly as well as greedily, but its target always updates towards the optimal action. I will also use alternative tabular RL algorithms such as Double-Sarsa and Dyna [48].

A great deal of recent success has been achieved in the area of Deep Reinforcement learning, in particular Minh [27] trains agents to super-human levels at Atari games . The key idea is that where the state space has high dimensionality tabular methods no longer suffice. A deep neural network is therefore used as a function approximator and its parameters updated directly by the agent's actions. When I move beyond 'buckets' of wealth and prices to a more practical higher dimensional setting I will use the same technique, in particular I test Deep-Q learning [27] and double Deep-Q agents [52], finally I briefly test a more sophisticated agent utilitising a different method of exploration, prioritised replay and Noisy-Nets [13].

An alternative and important area of RL which would be directly applicable to the Merton problem is to use Policy Gradients or Actor-critic methods [26]. I did not cover these but consider these methods to be very interesting and in some ways more direct than Q-learning, also they can be fitted naturally to high dimensional or continuous action spaces. The market environment has a high degree of randomness and thus agents require a large number of samples to learn, exploring sample efficiency further would thus have been of interest. The reason for not also implementing these agents was really a question of scope for the project.

1.3.3 Reinforcement Learning applied to Finance

Attempts to use reinforcement learning for trading problems date back to the 1990's with papers such as Moody and Saffell [28] who derived a differential sharpe ratio reward using recurrent reinforcement learning, and also Neuneier [30] who applied Q-learning to a single instrument but without including risk aversion.

Some of the techniques applied in this thesis come from Ritter [35] who used a mean-variance approximation to trade an Ornstein-Uhlenbeck process (when I move from my exact log-utility reward formulation, this is the exact reward formulation I use). Another recent contribution comes from Halperin, 2017 [17] who used Q-learning to derive a model free agent capable of discrete time option pricing (without needing the Black-Scholes Merton formula), in philosophy this is similar to this thesis, where the particular benefits of a model-free approach are highlighted.

Finally I would point the reader to a recent paper by Spooner - 2018 [47], where instead of examining the Merton problem he uses reinforcement learning for market making, which I believe is an excellent direction. There are a variety of related financial problems that should be highly appropriate for the RL paradigm (indeed perhaps more so than the Merton problem).

In problem settings such as market making, inventory management, optimal liquidation and many statistical arbitrage problems the entire limit order book may be taken into account, we now have a situation where the agent's actions not only receive a reward but where there is a more direct influence on the next state of the environment (for example placing a limit order of a certain volume immediately changes the limit order book), the agent is no longer a pure 'price-taker' as he is for the problem setting in this thesis, but also directly influences the environment. Thus I would expect to see further activity in this area with Spooner's paper [47] being just the beginning.

Chapter 2

Finance Background

In this chapter I shall give an introduction to the financial ideas needed for this thesis, as well as the data, intuitions and mathematical machinery. Given the potentially very wide scope - I could potentially cover ideas from finance, utility theory, single and multi-period optimisation as well as the later machine learning machinery, I shall need to brush over some areas (such as utility theory) and only do a deeper dive into areas where extra intuition is useful in order to understand the later experiments.

2.1 Portfolio Theory in brief

I shall assume that the reader is aware of the works of Markowitz [24], Sharpe [42] and Merton [25].

Portfolio theory addresses the problem of how best to allocate wealth between competing assets. The seminal work in this area by Markowitz [24], who solved this problem for a single decision period using a mean-variance approach. Markowitz's original work showed the importance of diversification and quantified the risk of a portfolio via the covariance matrix of underlying stock returns. If one can estimate the returns of the underlying stocks as well as the covariance matrix (i.e. the predicted covariances between underlying assets), then Markowitz showed how to find optimal portfolio weights given sufficiently well behaved returns.

There are a great many problems in practice with naively applying this approach, For example given a large number of stocks one needs a very large time series history to ensure the covariance matrix is not singular and in any rate the covariance matrix may have a poor conditioning number, because the solution requires its inversion results can be unstable. Also financial assets may behave in a heteroscedastic manner and thus blow prior mathematical assumptions away. Various approaches have been tried in both industry and academia to overcome these problems such as dimensionality reduction by using factor models (Barr Rosenberg and associates), using a regulariser or simply applying 'risk parity' [36], of whom the most famous practitioner is the worlds largest hedge fund manager, Ray Dalio.

Figure 2.1, illustrates some key points behind Markowitz and Sharpe's work. First, given individual assets whose correlations are less than one, then the potential set of portfolios in a risk-return space will be hyperbolic. If one adds a risk-free asset then there is a 'best' set of portfolios, along the line between the risk-free rate and the tangency portfolio (that with the best risk-return trade off, this is Sharpe's capital market line). If one imagines the tangency portfolio as being capable of being represented by one asset, such as an index, then in this thesis we are choosing asset proportions along this line (I also allow borrowing and going short), but over multiple time-periods not just one.

The Markowitz paradigm is problematic in that one is locked into a fixed single investment period and thus unable to rebalance in the meantime. A re-reading of Markowitz [24] showed that even in the 1950's he was not naive to the multi-period investment issue, he illustrates how long term geometric returns are damaged by increased variance and thus justifies his approximating the geometric mean via the mean of the logarithm, and from here goes on to justify his mean variance approximation.

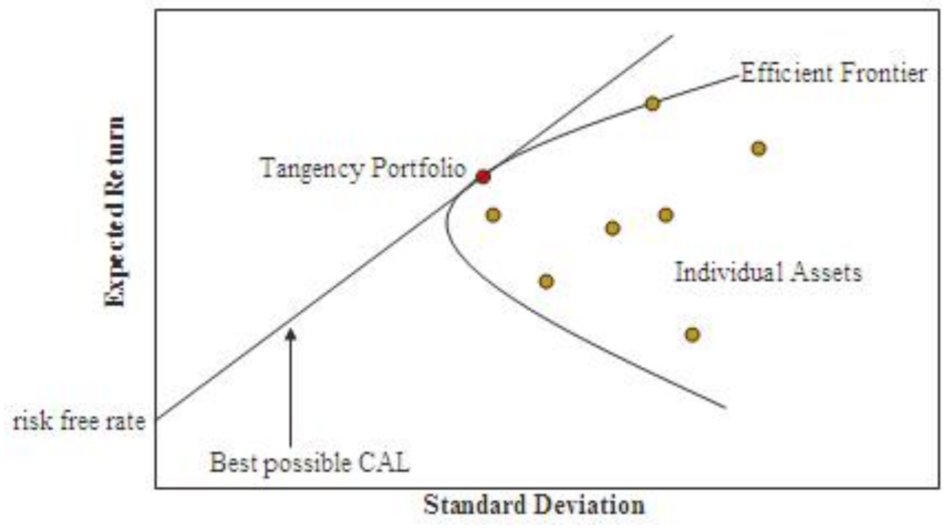


Figure 2.1: Efficient Frontier, (from Wikipedia)

In spirit this is actually incredibly similar to the techniques I use later to build the reward signal for the reinforcement learner on a step-wise basis, and of course how Ritter [35] applies his rewards to a different trading problem.

2.2 Data representations

In this section I will introduce the data that will be used in this thesis. In all problems in this project our agent will have at each point in time a choice between investing in a risk free bond with a positive rate of interest r, or an unknown stochastic process. The agent will have no knowledge of the kind of risky process it is trading, the respective drift rates μ and r for the risky process and bond, and nor will the agent have any awareness of the volatility σ. Indeed the agent also has no explicit knowledge of the utility function, however of course rewards are shaped specifically for a particular utility function.

2.2.1 Risk-free Bond

The bond is very simple. Given a riskless rate of return r, and a bond of price B_t at time t (Without loss of generality I set $B_0 = 1$). We have that:

$$dB_t = B_t r dt \text{ , therefore } B_t = e^{rt} \tag{2.1}$$

2.2.2 Geometric Brownian Motion

Geometric brownian motion is possibly the simplest stochastic process used for modelling a trending risky asset in finance. It is also somewhat naive as it is known that real markets exhibit both skew, kurtosis, jumps and heteroscedacity. However it is a good starting point and for our purposes, solving Merton's problem, enables an analytic solution, meaning we can test our agent against the 'perfect' Merton agent. Of course in more realistic circumstances the hope would be that an RL agent can adapt beyond these simplifying assumptions and learn to trade regardless. Here analytical solutions would be unavailable and Merton would be silent.

Given a complete probability space $(\Omega, \mathscr{F}, P)$, with $\mathscr{F}$ being a filtration (i.e. contains measure 0 sets and is right continuous) and if $T > 0$ is our terminal time period. Then $(Z_t)_{t\in[0,T]}$ is a brownian motion if it satisfies the following conditions (note that notationally I would normally use (W_t) or (B_t), however I use W_t for wealth at time t, and B_t for the bond price at time t):

Brownian Motion conditions

- Z is continuous
- $Z_t - Z_s$ is independent of $\mathscr{F}_s, 0 \leq s < t$
- $\forall t > 0, Z_t \sim N(0,t)$ under $P, 0 \leq s < t$

Therefore if the risky stock S_t evolves as a geometric brownian motion, then we have:

$$dS_t = S_t(\mu dt + \sigma dZ_t) \text{ , with } S_0 = s_0 > 0 \tag{2.2}$$

Without loss of generality I use $s_0 = 1$. Using Ito calculus we can derive the price evolution of our risky asset as:

$$S_t = S_0 e^{(\mu - \frac{\sigma^2}{2})t + \sigma Z_t} \tag{2.3}$$

In my implementation, I create a market environment by discretising the above equations. In general for any of my stochastic processes the wealth W_t of the agent will also evolve stochastically according to:

$$dW_t = N_t^B dB_t + N_t^S dS_t \tag{2.4}$$

Where N_t^B and N_t^S, are the numbers of bonds and stocks held at time t.

2.2.3 Fat Tails

It is commonly known that financial time series rarely seem to exhibit Gaussian noise, but have fatter tails where the probability of extreme loss is larger than a Gaussian might suggest.

For my fat tailed experiments I chose to add random increments from the Student-t distribution with 10 degrees of freedom in order to simulate adding fatter tailed noise. With a large number of degrees of freedom the Student-t approaches a Gaussian, with a low number then extreme moves become more likely. The equation for the pdf is given below.

$$\Pr(x, df) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df}\,\Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-\left(\frac{df+1}{2}\right)} \tag{2.5}$$

The Student-t pdf for varying degrees of freedom is given in figure 2.2.

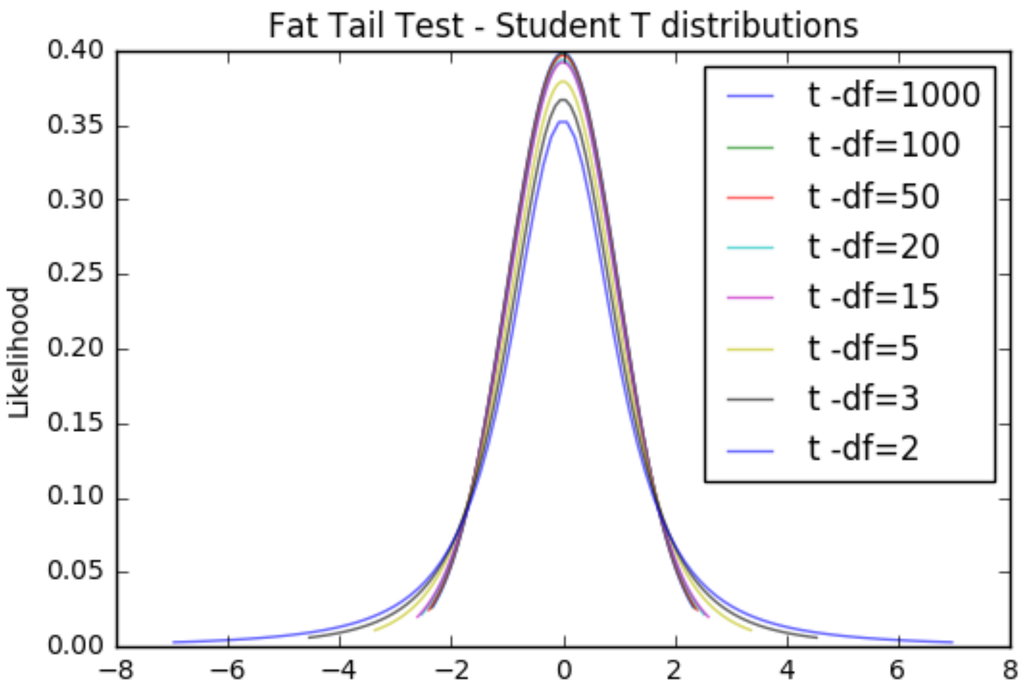


Figure 2.2: Student-T pdfs - Fat Tail test

2.3 Risk Neutrality and Utility Curves

To provide intuition, I will borrow from Professor Gordon Ritter's simple thought experiment in his lectures at NYU. Imagine you own two gold bars located abroad. You are only able to realise the value of these gold bars if they are successfully shipped home. The problem being the journey is dangerous and at present shipments have a 50% chance of being lost.

There are two ships available, should you put one gold bar on each ship and send each home separately? or should you put both gold bars on one ship and take your chances? A risk neutral investor might say, 'Well there is no difference, the expected value of either option is one gold bar'. This is of course true.

Expected terminal wealth

- One ship: $E(w_T) = \frac{1}{2} * 2 = 1$
- Two ships: $E(w_T) = \frac{1}{2} * 1 + \frac{1}{2} * 1 = 1$

Thus the expected wealth for either decision at terminal time T is one gold bar. However for most people this does not quite sit right and they might prefer to spread their risk or diversify. These are risk averse investors. If one now examines the variance.

Expected Volatility of terminal wealth

- One ship: $V(w_T) = \frac{1}{2} * (0-1)^2 + \frac{1}{2} * (2-1)^2 = 1$
- Two ships: $V(w_T) = \frac{1}{4} * (0-1)^2 + \frac{1}{4} * (2-1)^2 + \frac{1}{2} * (1-1)^2 = 0.5$

We see that those who chose to diversify by using two ships experienced half the variance of their terminal wealth. These differences can be represented by 'utility'.

In this work I will be attempting to maximise the expected utility of terminal wealth over multiple time periods. That is given a utility curve U, in all problems I wish to solve:

$$\max \mathbf{E}[U(w_T)], \text{ for } t \in [0, \dots, T] \tag{2.6}$$

2.3.1 Utility Curve Intuitions

There is a whole body of literature and theory on utility curves, which I am not going to go into. However I will give a few key takeaways.

- Utility Curves are upward sloping (greater wealth gives greater utility).
- A risk neutral investor has a linear utility curve.
- Risk Aversion introduces concavity into the utility curve.

- Introducing a utility function can introduce path dependence, extreme ups and downs gives us less utility than a smooth path. Thus risk is intimately intertwined with utility.
- Over multiple time steps it is more often NOT the case that the optimal set of actions long term is identical to repeatedly taking the optimal short term action.

The intuition as to why it may not be the cleverest thing to maximise wealth itself is perhaps more obvious when one realises that over repeated time periods such an investor would always accept ANY bet with an expectation equal to or greater than his current wealth, and be willing to risk all of his wealth to do so.

2.3.2 Mean Variance Utility

Later on I apply reinforcement learning to the Merton problem, but in order to do so I first need to reformulate the expected terminal utility into a series of rewards. I will be taking a mean-variance approximation to the terminal utility and then creating step by step rewards which in expectation should match the mean-variance approximation. Theoretically this is very similar to mean-variance utility which I now introduce.

Markowitz's [24] work which contributed to his Nobel prize, used mean variance utility but was only fully formalised in terms of expected utility many years later in a joint paper with Levy, [23]. They approximated the expected utility function as follows:

$$E(U(1+r_T) \approx U(1+E(r_T)) + \frac{1}{2}U^{''}(1+E(r_T))var(r_T) \tag{2.7}$$

Utility functions are concave, therefore the second derivative in equation 2.7 is negative and so any investor who wishes to maximise expected utility is similar to an investor maximising the expected mean return given a certain variance.

It is a common misconception that the weakness of Markowitz's work is that he requires returns to be normally distributed (which is unrealistic). One can see from the above equation that this is simply not the case, the lack of realism comes of course when means and variances are poor statistics to describe the overall return distribution.

2.4 Merton's Problem

The Merton problem is to find the investment policy over all time-periods that maximises the expected utility of wealth at some future period of time:

$$\max_u E[U(W_T)|W_0 = w_o] \text{ for } t \in [0, \ldots, T] \tag{2.8}$$

2.4.1 Merton solution - Logarithmic Utility

In my initial experiments I assume that the investor exhibits logarithmic utility. The first reason being that the solution to the Merton problem with logarithmic utility and geometric brownian motion has a very nice clean analytic formula and is thus easy to test against a perfect agent. The second reason is that this is also easy to reformulate in terms of exact myopic rewards on a step by step basis for an RL agent - which I shall derive in the implementation chapter. The solution to Merton's problem uses mathematical machinery from stochastic optimal control which I won't go into, detailed introductions to this can be found in Alvaro Cartea's book on Algorithmic and high frequency trading [6].

The proof for Merton using logarithmic utility basically follows an MSc thesis by Kinga Tikosi on Merton's problem barring one small change I made at the end, where I believe there was a minor inaccuracy.

For $U(W) = log(W)$, the optimal policy is

$$u_t^* = \frac{\mu - r}{\sigma^2}, \text{ for all } t \in [0, T] \tag{2.9}$$

The surprising result for geometric brownian motion with a logarthmic utility is that regardless of ones wealth level at every time period one should invest a fixed proportion of one's wealth dependent upon the drift of the risky asset μ, the risk-free rate of return r, and the volatility of the risky asset σ.

Proof

The wealth W_t of the agent will evolve according to:

$$dW_t = N_t^B dB_t + N_t^S dS_t \tag{2.10}$$

Where N_t^B and N_t^S, are the numbers of bonds and stocks held at time t. If we reformulate the number of stocks and bonds bought in terms of the proportion of our wealth invested in the risky asset at time t, u_t, then we have that:

$$N_t^B = \frac{(1-u_t)W_t}{B_t} \text{ and } N_t^S = \frac{u_t W_t}{S_t} \tag{2.11}$$

Given the above we can now write the evolution of our wealth process as follows:

$$\begin{aligned} dW_t &= (1-u_t)W_t r dt + u_t X_t(\mu dt + \sigma dZ_t) && (2.12)\\ &= W_t((r + u_t(\mu - r))dt + \mu\sigma dZ_t) && (2.13)\end{aligned}$$

If we imagine a solution of the form,

$$W_t = w_0 exp(\int_0^t g_s ds + \int_0^t h_s dZ_s) \tag{2.14}$$

Then applying Ito we have that,

$$\begin{aligned} dW_t &= W_t g_t dt + X_t h_t dZ_t + \frac{1}{2} W_t g_t^2 dt && (2.15)\\ &= W_t((g_t + \frac{1}{2}h_t^2)dt + h_t dZ_t) && (2.16)\end{aligned}$$

We can now read off our values for g_t and h_t, so:

$$W_t = w_0 exp(\int_0^t (r + (\mu - r)u_s - \frac{1}{2}\sigma^2 u_s^2)ds + \int_0^t \sigma u_s dZ_s \tag{2.17}$$

We wish to maximise $E(log(W_T^u)|W_0 = w_0)$, therefore using the fact that $E(\int_0^T \sigma u_t dZ_t) = 0$. We have that

$$J(w_o, u) = log w_0 + E\left(\int_0^T (r + (\mu - r)u_s - \frac{1}{2}\sigma^2 u_s^2)ds\right) \tag{2.18}$$

Taking derivatives,

$$\frac{\partial}{\partial u}(r + (\mu - r)u_s - \frac{1}{2}\sigma^2 u_s^2) = \mu - r - \sigma^2 u \tag{2.19}$$

$$\frac{\partial}{\partial^2 u}(r + (\mu - r)u_s - \frac{1}{2}\sigma^2 u_s^2) = -\sigma^2 \tag{2.20}$$

The negativity of the second derivative proves concavity and we can solve for u by setting the first derivative equal to zero. Therefore

$$u_t^* = \frac{\mu - r}{\sigma^2} \tag{2.21}$$

with a value of

$$J(w_o, u^*) = log(w_0) + \left(r + \frac{(\mu - r)^2}{2\sigma^2}\right)T \tag{2.22}$$

The final equation differs slightly from that derived by Tikosi, where I believe he may have made a small slip at the end.

In my future experiments, I will discretise the above for simulation, train and test purposes. The performance of the trained reinforcement learning agent will be compared to the Merton optimal agent given above.

2.4.2 Merton Solution - Power Utility

The second utility I shall use in my experiments is power utility. This is more general than log-utility and is given below:

$$U(x) = x^{\gamma}\,, 0 < \gamma < 1 \tag{2.23}$$

If the stochastic process is a geometric brownian motion then the Merton optimal investment is again a fixed proportion of one's wealth regardless of the level of the wealth or indeed the level of the GBM. By a similar derivation to that given above it can be shown that the Merton optimal investment is:

$$u^{*} = \frac{\mu - r}{\sigma^{2}(1-\gamma)} \tag{2.24}$$

The parameter γ, reflects the degree of risk aversion. In this work regardless of the type of utility curve, as long as the curve is upward sloping and the underlying stochastic process comes from a mean-variance equivalent distribution (which includes all elliptical distributions) then the optimal policy for the utility curve can be reformulated into an equivalent mean-variance one [35], the curves will not necessarily be the same, but they will have the same optimal policy. I use this to shape rewards for the RL agent.

Chapter 3

Machine Learning Background

3.1 Reinforcement Learning

I believe the best current exposition of Reinforcement Learning comes from 'Reinforcement Learning - An Introduction', by Sutton and Barto [49], as well as their new edition (currently on-line). My mathematical explanation of Reinforcement Learning closely follows [49] and the lectures by Deep Mind at UCL. In this section I will introduce the basics and justify the algorithms used.

3.1.1 The Reinforcement Learning setting

Unlike Supervised Learning, where we have a training set with training labels and attempt to predict labels on an unseen test set. And unlike Unsupervised Learning, where one has no labels and attempts to group similar data items together in some fashion. Reinforcement learning has a quite different problem setting.

In Reinforcement Learning we have an agent, who interacts with an environment and receives a reward in an online fashion. These rewards may be delayed, and noisy. The RL agent is able to interact with its environment by choosing an action, which results in a reward and a new state (not necessarily the one the agent hoped for as this also depends upon the environment). Learning in an RL sense comprises the agent learning a set of actions (a policy) appropriate to whatever state it finds itself in, in order to maximise its expected future rewards. I will formalise this with some mathematics, however the key intuition is that this is actually a very broad and ambitious goal for a machine learning algorithm and that it captures many real life situations, at least in principle.

Figure 3.1, from Sutton and Barto [49], shows the basic setup.

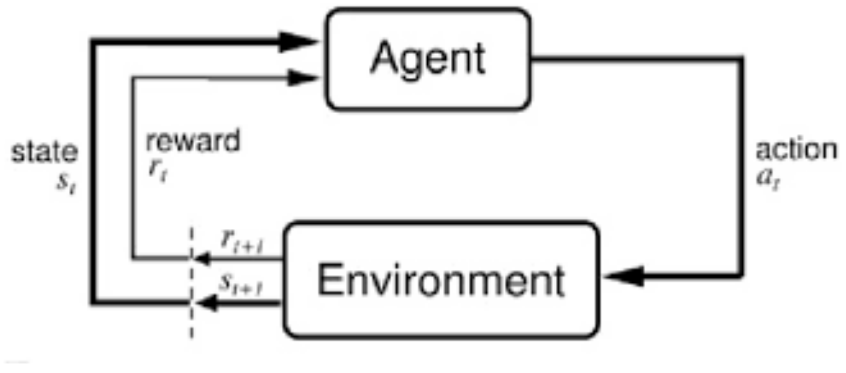


Figure 3.1: Basic RL framework

In a finance setting we may for example imagine the market itself is the environment our agent interacts with. States can be very general and may include for example price levels, but also any relevant information available at that time, such as valuation metrics and news items. The agent's wealth level will also be included in the state. For our purposes states will be asset prices and the wealth level of the agent. The agent's action at each time step will be to invest in a risky asset or a bond but where he may also use leverage or go short, the reward will be a risk adjusted change in wealth tailored for utilities.

In contrast to supervised learning, this is far broader than say regressing states against rewards, as rewards may be delayed and the states the agent may find himself within are themselves dependent upon his previous actions. It is also far more sophisticated than taking the best myopic action at each step as this often proves to be sub-optimal over multiple steps. If the last sentence reminds one of dynamic programming, then that is exactly the right way to think.

Reinforcement Learning thus involves elements of optimisation, online learning and has its own unique issues such as when to explore and when to exploit existing knowledge. As dimensionality increases problems may also require a function approximator such as a neural network to simplify a potentially intractable state, action space. It may also involve aspects of transfer learning, in moving from a simulated environment to a real world setting.

3.1.2 Markov Decision Processes

3.1.2.1 Markov Property

A state s has the Markov property if $\forall s' \in S$ and $\forall$ rewards $r \in R$ and all histories $S_1, \ldots, S_{t-1}$

$$\mathscr{P}(R_{t+1} = r, S_{t+1} = s' | S_t = s) = \mathscr{P}(R_{t+1} = r, S_{t+1} = s' | S_1, \ldots, S_{t-1}, S_t = s) \quad (3.1)$$

A market practitioner may say well this does not apply in reality, this is true. However, the fact that only the previous state matters is not as restrictive as first appears because one can add whatever prior information one wishes into our current state. The state-action space would then become more complex of course, however the point is that RL algorithms can use information prior to the current time step as long as this is included in the current state and so this formulation has proven to be quite general in practice.

3.1.2.2 State Transition Matrix

Given states with the markov property, then we can define the state transition probability as:

$$P_{ss'} = \mathscr{P}(S_{t+1} = s' | S_t = s) \quad (3.2)$$

If we have a finite set of states then the state transition probabilities form a matrix, which is part of the model of the environment. Bar the simplest settings such as dynamic programming, this model of the environment is likely to be hidden from the agent (and certainly is in this thesis, one reason why I do not use Dynamic

Programming). There are various ways an RL agent can deal with this, from learning its own model of the environment online, to simply learning optimal actions from rewards without caring about a model, to alternative combinations such as actor-critic.

3.1.2.3 Markov Process

A Markov process is now easy to define as a set of states together with a state transition matrix.

3.1.2.4 Markov Reward Process

A Markov reward process is simply a markov process with rewards. i.e we add a reward function. For both purposes of convengence (for non episodic environments) and also to alter the myopia of the agent, we may choose to discount rewards with a variable $\gamma \in (0,1)$.

$$\mathscr{R}_s = E(R_{t+1}|S_t = s) \tag{3.3}$$

3.1.2.5 Return

The return is the total discounted reward from a given time step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.4}$$

3.1.2.6 Markov Decision Process

In my view the core of RL lies around Markov Decision Processes and using the variations of the Bellman equation to find optimal control values. A Markov Decision Process (MDP) is a Markov Reward Process with decisions.

We adjust our state transition probability matrix and reward function as follows:

$$P^a_{ss'} = \mathscr{P}(S_{t+1} = s'|S_t = s, A_t = a) \tag{3.5}$$

$$\mathscr{R}_s^a = E(R_{t+1}|S_t = s, A_t = a) \tag{3.6}$$

or more concretely,

$$p(s', r|s, a) = \mathscr{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{3.7}$$

therefore,

$$P_{ss'}^a = \sum_{r \in R} p(s', r|s, a) \tag{3.8}$$

and

$$\mathscr{R}_s^a = \sum_{r \in R} r \sum_{s' \in S} p(s', s|s, a) \tag{3.9}$$

3.1.2.7 Policy

A policy is a distribution of actions over states. Notation hasn't quite settled down here yet, so following the Deep Mind lecturers...

$$\pi(s, a) = \pi(s|a) = \mathscr{P}(A_t = a|S_t = s) \tag{3.10}$$

3.1.2.8 State Value Function

The state value function $v^\pi(s)$ of a markov decision process is the expected return gained by beginning in a state s and following a policy π.

$$v^\pi(s) = E_\pi(G_t|S_t = s) \tag{3.11}$$

3.1.2.9 Action Value Function - Q

The action-value function $q^\pi(s, a)$ is the expected return from starting in a state s, taking an action a and then following the policy π.

$$q^\pi(s, a) = E_\pi(G_t|S_t = s, A_t = a) \tag{3.12}$$

3.1.3 The Bellman Expectation equation

For state value functions we can see that:

$$\begin{aligned}
v^{\pi}(s) &= E_{\pi}(G_t|S_t = s) && (3.13)\\
&= E_{\pi}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots|S_t = s) && (3.14)\\
&= E_{\pi}(R_{t+1} + \gamma(R_{t+2} + \gamma^2 R_{t+3} + \ldots|S_t = s) && (3.15)\\
&= E_{\pi}(R_{t+1} + \gamma G_{t+1}|S_t = s) && (3.16)\\
&= E_{\pi}(R_{t+1} + \gamma v^{\pi}(S_{t+1})|S_t = s) && (3.17)
\end{aligned}$$

using the law of iteration expectations.

Similarly for action value functions we can see that:

$$q^{\pi}(s,a) = E_{\pi}(R_{t+1} + \gamma v^{\pi}(S_{t+1}, A_{t+1})|S_t = s, A_t = a) \tag{3.18}$$

One can see the connection between state value functions and action-value functions as follows:

$$v^{\pi}(s) = \sum_{a \in A} \pi(s,a) q^{\pi}(s,a) \tag{3.19}$$

$$q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^{\pi}(s') \tag{3.20}$$

Therefore:

$$v^{\pi}(s) = \sum_{a \in A} \pi(s,a)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^{\pi}(s')) \tag{3.21}$$

and,

$$q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(s',a') q^{\pi}(s',a') \tag{3.22}$$

The bellman expectation equation thus gives us a method of finding the value of being in a particular state given a policy, or the value of being in a state, taking an action and then following the policy. This is nice, however we are more interested in finding the best policy.

3.1.3.1 Optimal State Value Function

The optimal state-value function $v^*(s)$ is the maximum value function over all possible policies.

$$v^*(s) = \max_{\pi} v^{\pi}(s) \tag{3.23}$$

3.1.3.2 Optimal Action Value Function

Similarly the optimal action-value function $q^*(s,a)$ is the maximum action value function over all possible policies.

$$q^*(s,a) = \max_{\pi} q^{\pi}(s,a) \tag{3.24}$$

3.1.3.3 Optimal Policies

We can define a partial ordering over policies $\pi \geq \pi^{'}$ if $v^{\pi} \geq v^{\pi^{'}}(s), \forall s$. It is known that for any markov decision process $\exists \pi^* \geq \pi \; \forall \pi$ and also that $v^{\pi^*}(s) = v^*(s)$ and $q^{\pi^*}(s,a) = q^*(s,a)$. In other words there is an optimal policy and all optimal policies achieve the same optimal state and action value function.

Therefore if we know $q^*(s,a)$, then we know our optimal policy immediately.

$$\pi^*(s,a) = 1 \text{ if } a = \arg\max_{a \in A} q^*(s,a) \text{ and } 0 \text{ otherwise} \tag{3.25}$$

3.1.4 The Bellman Optimality equation

Again we can use bellman this time on our optimal value function and action value function.

$$v^*(s) = \max_a q^*(s,a) \tag{3.26}$$

$$q^*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^*(s') \tag{3.27}$$

Therefore,

$$v^*(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^*(s') \tag{3.28}$$

and,

$$q^*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q^*(s',a') \tag{3.29}$$

However because of the maximum term in this equation there is in general no closed form solution. However there are various iterative solution methods such as policy and value iteration as well as Q-learning and Sarsa.

3.1.5 Learning algorithms - From Dynamic Programming to Dyna

We now have the Bellman optimality equation for both $q^*(s,a)$ and $v^*(s)$, the key question is what algorithms can we use to find this optimum? As previously mentioned there is in general no closed form solution for the bellman optimality equation, however there are iterative solution methods.

3.1.5.1 Dynamic programming methods

If we already have a policy π that we wish to evaluate then we know from Bellman that for every state:

$$v_\pi(s) = \mathbf{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s) \tag{3.30}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{3.31}$$

Using the above we can use an iterative method to derive successive approximations to the bellman equation where the final equation itself is a fixed point. Thus, if we guess some value for all states bar the terminal V_0 then successive approximations are derived for $k = 0, \ldots,$ and $\forall s \in$ by performing the following iteration:

$$v_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r'|s,a)[r + \gamma v_k(s')] \tag{3.32}$$

This is called policy evaluation, indeed after convergence we will have a new set of values $v(s)$ and for example by acting greedily with respect to these values we can derive a new policy. By performing policy evaluation on this new policy we again have a new set of values $v(s)$ and so on. It can be shown that this will converge to an optimal policy and is called policy iteration.

There are various adjustments to this which I won't go into (for example we don't necessarily need to go all the way to convergence when we do policy evaluation and can instead do value iteration), but for the purposes of this thesis I demonstrate this solely to highlight a somewhat obvious flaw for our financial setting.

To successfully perform this iteration we must already know the transition probabilities $p(s',r'|s,a)$, which implies that we already need to have a perfect model of the market. Thus, if we knew the model of the market we could perform this bootstrapping operation in a step-wise manner and solve the Merton problem. These are precisely the properties that Merton used in solving his problem (albeit in a continuous time setting using a reformulation of Bellman called the HJB equation), which is why Merton must know the formulation of the stochastic risky process in advance, as well as its parameters in order to solve his equation.

However we wish to learn the optimal investment strategy in a model free fashion, with minimal assumptions and purely through the agent's interactions with the market.

3.1.5.2 ε-Greedy exploration

Before taking a very brief look at Monte Carlo methods and why I chose not to use them I will briefly explain ε-greedy exploration.

If one imagines that an agent always takes the best action, then learning is compromised. The problem is that if for example the agent takes a random initial action, after the first positive update of the Q values then the agent will only ever take this specific action in this state. The agent will never try other potential actions that might be better and will therefore never fully explore the state-action space. To counter this and ensure convergence I use an ε-greedy policy. The agent takes the best action in any given state with probability $1-\varepsilon$ and takes a random action with probability ε. This ensures the complete exploration of the state-action space. The exploration-exploitation dilemma is unique to reinforcement learning and there are a variety of methods to overcome it, ε-greedy being just one.

Mathematically, if $A^* = \arg\max_a Q(S_t, a)$ then the action we take is that $\forall a \in A :,$

$$\pi(a|S_t) = 1 - \varepsilon + \frac{\varepsilon}{|A|} \text{ ,if } a = A^* \tag{3.33}$$

$$= \frac{\varepsilon}{|A|} \text{ ,if } a \neq A^* \tag{3.34}$$

3.1.5.3 Monte Carlo methods

Monte Carlo methods are capable of learning without a model of the environment, however they rely upon sampling experience in an episodic fashion. I did create an episodic environment for this particular problem (partly for test purposes), however I am not tied to it, the methods used in this thesis can be used in a continuing environment due to the use of step-wise rewards and the fact there is nothing special about reaching the end of an episode.

Monte carlo methods rely upon being able to reach an end state repeatedly and then update the q-values based upon their respective reward experience. Thus, although we now longer need a model of the environment, we have introduced a new issue in that the environment itself needs to be episodic in nature (for example millions of games of chess). Therefore, I chose not to use Monte Carlo methods.

For completeness I include the every visit Monte Carlo control algorithm below (algorithm 1), notice that this is an on policy (we update the q-values of the policy we are actually following) control algorithm using ε-greedy exploration and also note that it is usually more straightforward although memory intensive to work directly with q-values for control problems rather than v-values:

Algorithm 1 On Policy MC Control

```
 1: procedure MC CONTROL
 2:     Initialise:
 3:     $\pi \leftarrow$ some $\varepsilon$-soft policy
 4:     $Q(s,a) \in \mathscr{R}\ \forall s \in S, a \in A$
 5:     $Returns(s,a) \leftarrow$ an empty list $\forall s \in S, a \in A$
 6:     while True do                                   ▷ Loop forever for each episode
 7:         Generate an episode following $\pi$:$S_0,A_0,R_1,S_1,A_1,R_2,\dots,S_{T-1},A_{T-1},R_T$
 8:         $G \leftarrow 0$
 9:         while t $\leq T$ do   ▷ Loop for each step in episode $t = T-1, T-2, \dots, 0$
10:             $G \leftarrow G + R_{t+1}$
11:             Append G to $Returns(S_t,A_t)$
12:             $Q(S_t,A_t) \leftarrow average(Returns(S_t,A_t))$
13:             $A^* \leftarrow \arg\max_a Q(S_t,a)$
14:             $\forall a \in A$ :
15:             $\pi(a|S_t) \leftarrow 1 - \varepsilon + \frac{\varepsilon}{|A|}$ ,if $a = A^*$
16:             $\pi(a|S_t) \leftarrow \frac{\varepsilon}{|A|}$ ,if $a \neq A^*$
```

3.1.5.4 TD, and Q learning

Temporal difference methods are capable of learning without having to reach the end of an episode unlike Monte Carlo methods, but also without needing a model of the environment unlike dynamic programming methods. For my initial experiments I used the DoubleQ-learning algorithm [19] a variant on the Q-learning algorithm created by Watkins [55]. This is an off-policy TD variant in that the learned q values approximate the optimal q values independent of the policy being followed (I used ε-greedy policy exploration). The key update is:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{3.35}$$

So instead of our target being the return from the whole episode as in Monte Carlo, the target is bootstrapped from the next state as in dynamic programming. By choosing the $\max_a$ in our target update we are performing an off-policy update of maximal actions in each state. Q-learning is effectively learning from a guess, but one where there is slightly greater information available. For the Merton problem I considered Q-learning to be the simplest suitable model free RL algorithm.

Algorithm 2 Q-learning

```
1:  procedure Q LEARNING
2:      Initialise:
3:      Step size: $\alpha \in (0,1]$
4:      $Q(s,a) \in \mathscr{R}\ \forall s \in S, a \in A$
5:      while True do                                    ▷ Loop for each episode
6:          Initialise S
7:          while t $\leq T$ do          ▷ Loop for each step in episode $t = 0,\dots,T$
8:              Choose A from S, $\varepsilon$-greedy policy
9:              Take action A and receive from environment R,S'
10:             $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
11:             $S \leftarrow S'$
12:
```

Algorithm 2 shows the full Q-learning algorithm, α is our step-size and controls our learning rate and γ is our discount rate. On convergence the optimal policy π^* is found by acting greedily with respect to Q^*.

3.1.5.5 Double-Q learning and Double Sarsa

In Q-learning there is a known bias called the maximisation bias [49]. This comes about because we are maximising when creating our target policies but taking this maximum over estimated values. Sutton and Barto give a simple demonstration of this in their book.

To avoid maximisation bias I used Double-Q learning for the majority of my initial experiments. This is a simple adjustment where we maintain two separate sets of q-values and we randomly update one or the other at each step. When choosing my optimal policy I then acted greedily with respect to the average of the q-values. In the case of the Sarsa algorithm the only difference lies in the fact that the policy we follow is also our target policy (ϵ-greedy). From an algorithmic viewpoint the code is almost identical, but the effects in terms of agent behaviour can be quite different, one example being the windy gridworld example shown by Sutton and Barto. Algorithm 3 shows the Double-Q learning algorithm.

Algorithm 3 Double Q-learning

```
1: procedure DOUBLE Q LEARNING
2:     Initialise:
3:     Step size: $\alpha \in (0,1]$
4:     $Q_1(s,a), Q_2(s,a) \in \mathscr{R}\ \forall s \in S, a \in A$
5:     while True do                                    ▷ Loop for each episode
6:         Initialise S
7:         while t $\leq T$ do        ▷ Loop for each step in episode $t = 0,\dots,T$
8:             Choose A from S, $\epsilon$-greedy policy in $Q_1 + Q_2$
9:             Take action A and receive from environment R,S'
10:            if $p < 0.5$ then                    ▷ If p probability is 0.5 or less
11:                $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha[R + \gamma Q_2(S', \arg\max_a Q_1(S',a)) - Q_1(S,A)]$
12:                $S \leftarrow S'$
13:            else
14:                $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha[R + \gamma Q_1(S', \arg\max_a Q_2(S',a)) - Q_2(S,A)]$
15:                $S \leftarrow S'$
16:
```

3.1.5.6 Dyna

The final tabular RL agent I tested in my experiments is Dyna [48]. Although the aim of this thesis is to apply model free reinforcement learning methods, Dyna

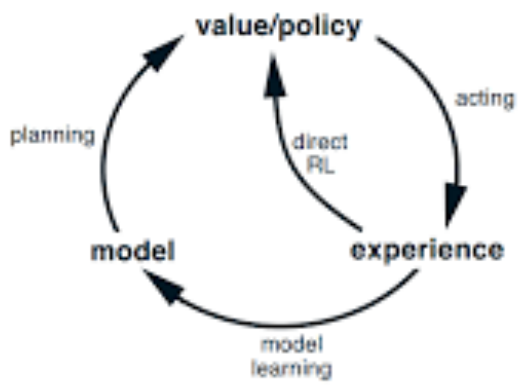


Figure 8.1: Relationships among learning, planning, and acting.

Figure 3.2: Dyna conceptual model - Sutton and Barto [49]

could be classified as being model based. However it is model free in terms of its requirements from the environment, the difference in philosophy is that it builds its own internal model as it gains experience, in order to predict how the environment will respond. Given this model it can also simulate its own experience (called experience replay) thus it learns by both interacting with the environment and thus gaining real experience as well as replaying its own simulated experience. Sutton and Barto [49] give a conceptual diagram showing the idea behind Dyna which is given in figure 3.2. The algorithm is shown in algorithm 4, notice the use of the model, as well as two sets of q-value updates, one from real experience and one from experience replay.

This completes the introduction to the basic tabular RL methods used in this thesis, from here I will illustrate more sophisticated methods utilising neural networks as function approximators used in later experiments.

3.2 Advanced Reinforcement Learning

3.2.1 Function Approximation

The reinforcement learning agents thus far have stored q-values in a tabular form. In this project that translates into placing wealth states or price states into a limited

Algorithm 4 Dyna

```
1: procedure DYNA
2:     Initialise:
3:     Step size: $\alpha \in (0,1]$
4:     $Q(s,a) \in \mathscr{R}$ and $Model(s,a)\ \forall s \in S, a \in A$
5:     while True do
6:         $S \leftarrow$ Current non-terminal state
7:         $A \leftarrow \varepsilon$ -greedy(S,Q
8:         Choose action A and receive reward R and state S'
9:         $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \max_a Q(S',a)) - Q(S,A)]$
10:        $Model(S,A) \leftarrow R, S'$
11:        while i in range(n) do        ▷ Loop for n, number of experience replays
12:            Choose A from S, $\varepsilon$-greedy policy in Q
13:            Take action A and receive from environment R,S'
14:            S ← random previous observed state
15:            A ← random action previously taken in S
16:            $R, S' \leftarrow Model(S,A)$
17:            $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \max_a Q_(S',a)) - Q(S,A)]$
18:
```

number of discrete 'buckets', this begs a number of questions, the most obvious being how should one choose these buckets, because the moment one chooses to bucket then the agent will be unable to differentiate intra-bucket. Also storing a table of discretised bucketted states combined with every action quickly becomes limiting or indeed intractable for a higher dimensional state space, both in terms of memory and computational cost. The solution is to use function approximation.

Instead of storing each v-value or q-value we now parameterise the value function, for example if $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$, then the value function is parameterised by $V(s, \theta^v)$. Basically we learn a mapping from the real number vector space of θ's to the value function space given input features, because the θ-space is of lower dimension than the original state space we thus circumvent the 'curse of dimensionality'. There is a trade off however. In the case of using a deep neural network for function approximation, one gains an enormous amount of flexibility due to it being a universal approximator, (Cybenko - 1989 [21]), however we also lose the convergence guarantees that exist for Q-learning.

3.2.2 Deep Learning

Neural nets have existed for many years, indeed the early history dates back to Rosenblatt's perceptron in 1958 [37]. For various reasons they have tended to go in and out of fashion, first due to the early models lack of expressibility and second due to difficulties in training. Over the years the research community has found better ways of initialising these models, better activation functions to aid gradient stability and much faster methods of training such as back-propogation, a subset of automatic differentiation.

The particular 'fan-in' shape of many of the problems in deep-learning (high dimensional feature space to low dimensional output space) makes back-prop particularly efficient. The hardware community has contributed, with massively increased compute power and memory, the largest current neural networks - usually found within machine vision, have many millions of hyper-parameters and may be trained on thousands of TPU's.

For space reasons I will not explain Deep Learning from the bottom up, my favourite reference and an excellent book for the basics is written by Ian Goodfellow (the creator of generative adversarial nets) [15]. At its heart training a neural net consists of a simple four stage process: Predict, Quantify Loss, Differentiate Loss, Learn (Alter weights).

Predict

Figure 3.3 shows a typical feedforward neural network. If one imagines data being pushed into the input layer then this data is multiplied (in a matrix vector sense) by the weights (lines) and in turn pushed to the next layer, at each layer we use a non-linear activation function, such as 'Relu' (this non-linearity is what enables the expressibility of the network). The final predictions are pushed into the output layer. In the case of regression this may be a linear layer, or in the case of a probability distribution (say over actions) then this could be a 'softmax' layer.

Quantify Loss

Feed forward neural networks are trained using targets, at first weights are set randomly and predictions are thus random, by adding correct training targets we may compare our predictions with these targets and quantify the quality of these predictions using a loss function (this itself is dependent upon the nature of the problem, with mean-squared error and cross-entropy being common 'go-to's for a regression or classification problem).

Differentiate Loss

In order to improve the predictions we differentiate the loss with respect to all of the weights, thus loss functions should be differentiable in order to use back-propogation. Non-differentiable loss functions require alternative methods to back-prop.

Alter weights / Learn

Weights are changed by moving in the opposite direction to the gradient, numerical optimisation is a whole subject in itself, but methods range from simple step-wise stochastic gradient descent, to more sophisticated adaptive methods such as Kingma's ADAM [22], or even new methods such as hyper-gradient gradient descent [38]. This thesis uses ADAM [22]. Changing weights in this manner comprises 'learning'.

Summary

In short a neural network and its training comprises predicting targets via data through non-linear function composition, quantifying the loss, shifting the weights to reduce that loss (basically via the chain rule) and repeating over many batches of data. Results in a number of fields but particularly vision have been outstanding, also the dirty secret of much of machine learning is extensive hand crafted 'feature engineering' requiring human intervention in order to create features that can be

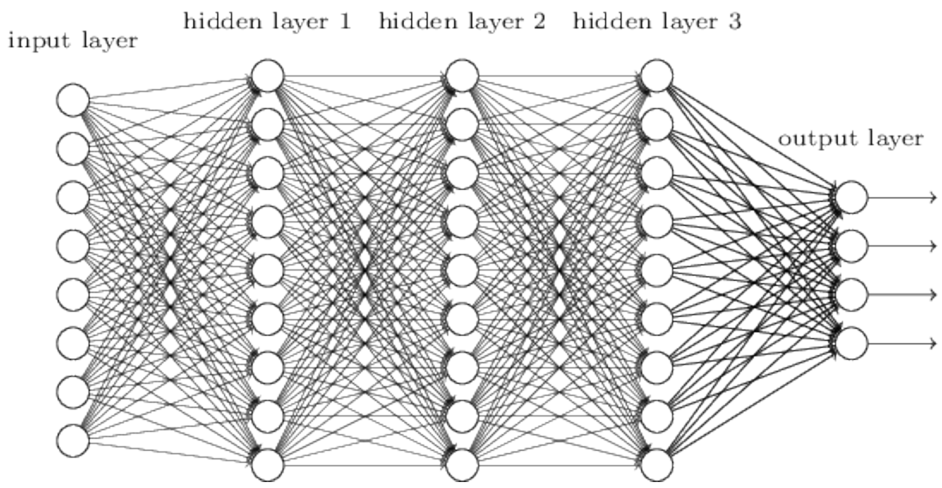


Figure 3.3: Feed forward Neural Net, (image from [34])

learned from. Neural nets work with the raw data itself and so in some sense 'learn' salient features themselves.

In this thesis the first deep neural networks used are relatively small fully connected feed-forward nets with 'Relu' activation functions. The inputs are price and wealth levels and the network predicts the q-values for a state (i.e. any price, wealth level). Because q-values are not limited in their range, linear activation functions are used for the output layer of the network (with its dimensionality being the number of possible actions for that state), for training purposes a mean-squared error loss function is used and the ADAM optimiser [22].

3.2.3 Deep and Double Deep-Q Learning

The presentation thus far has introduced basic tabular reinforcement learning agents and neural networks as a function approximator. A key paper which puts the two together and introduced Deep-Q learning is by Volodymyr Mnih et al. [52] from Deep Mind, a Deep-Q learner was trained on a high dimensional state space (Atari games) and became capable of a superhuman performance.

The challenges that RL present deep learning with are that first deep learning tends to require a large amount of labelled training data, however reinforcement learning algorithms do not learn in this manner, they learn instead from step-wise rewards which may be noisy, delayed and sparse. Also in deep learning we learn from batches of independent data samples, this is simply not the case in reinforcement learning where the norm is more likely a sequence of highly correlated states. Finally in deep learning it is assumed that the data has a fixed underlying distribution, however in reinforcement learning the data distribution will change as the agent learns (for example in this thesis the agent learns behaviours that improves the distribution of its utility of final wealth).

Minh [27] shows the following. If ε is our stochastic environment and we examine an equivalent form of the Bellman optimality equation:

$$Q^*(s,a) = \mathbf{E}_{s' \in \varepsilon}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{3.36}$$

If we now parameterise our Q action function by $Q(s,a;\theta) \approx Q^*(s,a)$, where the approximator is a neural network with weights θ, which Minh calls a Q-network, then the Q-network is trained by minimising a sequence of loss functions $L_i(\theta_i)$, where:

$$L_i(\theta_i) = \mathbf{E}_{s,a \in p(.)}[(y_i - Q(s,a;\theta_i))^2] \tag{3.37}$$

Here $y_i = \mathbf{E}_{s' \in \varepsilon}[r + \gamma \max_{a'} Q^*(s',a';\theta_{i-1})|s,a]$, this is the target for iteration i and $p(s,a)$ a probability distribution over sequences and actions. Parameters from the prior iteration are fixed when we optimise the loss and targets themselves depend upon the network weights, this is clearly different to the fixed targets used in standard neural network training. If we differentiate this loss we have that:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbf{E}_{s,a\in p(.),s'\in\varepsilon}[(r+\gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{3.38}$$

This can be solved iteratively through stochastic gradient descent, where we take samples from $p(.)$ and ε and is of the same familiar form necessary for standard off-policy Q-learning, thus solving the reward problem highlighted earlier. However we remain with the problem of correlated sequences of states, this was solved via experience replay.

3.2.3.1 Experience Replay

Here the agent's experiences are stored at each time step $\varepsilon_t = (s_t, a_t, r_t, s_{t+1})$ this is stored in a dataset called the replay memory $\mathscr{D} = (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N)$, when Q-learning updates are applied they are applied to mini-batches of experience drawn at random from $\mathscr{D}$ (I used a batch size of 32, as it seems obligatory to use powers of 2!). After experience replay in the same manner as before agent actions are selected via an ε-greedy policy, the only extra layer of sophistication being that here I also used a decay rate to control the agent's rate of exploration.

The benefit of this approach is that the sample correlation is broken helping the neural network to learn and also increased sample efficiency. Double deep-Q learning, which I also used in the experiments is exactly analogous to the double Q-learning variation introduced earlier, just using a deep neural network, it also aims to overcome the same bias affect. Details may be found in [19]. Algorithm 5 shows the implementation.

Algorithm 5 Deep-Q Learning with experience replay

```
 1: procedure DEEPQ
 2:     Initialise replay memory D to capacity N:
 3:     Initialise action value function Q with random weights
 4:     while episode in range(M) do
 5:         Initialise sequence $s_1$
 6:         while t in T do
 7:             Select an action $a_t$ using Q e.g. $\varepsilon$-greedy
 8:             Carry out action $a_t$ and observe $r_t$, $s_{t+1}$
 9:             Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathscr{D}$
10:             Sample transition batch $(s_j, a_j, r_j, s_{j+1})$ from replay buffer $\mathscr{D}$
11:             Create targets $y_j$
12:             if $s_{j+1}$ is terminal then
13:                 $y_j = r_j$
14:             else
15:                 $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a', \theta$
16:             Train the neural network using Equation 3.38
17:
```

3.2.4 Prioritised experience replay and Noisy Networks

The primary purpose in this thesis was to train an agent to solve Merton's problem in a model free manner, purely by interacting with the market. Along the way I found differences in the agents' sample efficiency as regards learning. As my final experiment I examined a different and more sophisticated exploration method.

3.2.4.1 Prioritised experience replay

Prioritised experience replay was introduced by Schaul et al. in 2016 [39]. In the previous section I introduced the idea of experience replay to break the correlation between states and also the improvement as regards sample efficiency. Experience replay uses uniform sampling, yet not all experiences are equal, the key idea behind [39] is that an RL agent can learn more effectively from some transitions compared to others, so we should prioritise them. If experience replay aids the agent in terms of not being forced to learn in the exact order in which experiences were experienced, then prioritised experience replay goes a step further by not forcing agents to learn in the exact frequency in which they were experienced.

The obvious question is how one should prioritise? As pointed out in [39] the ideal would be to do so by the amount an RL agent can learn via a transition from its current state, a proxy for which is the size of the transition's TD error δ, however as Schaul points out this is a very poor estimate when rewards are noisy, precisely the environment for the Merton problem. The author's solution was to introduce a stochastic sampling method interpolating between greedy δ prioritisation and a simple uniform sampling. They define the probability of sampling transition i as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.39}$$

with $p_i > 0$ being the priority of transition i and α being a parameter where we control the degree of prioritisation. I won't go into further details which can be found in [39]. My aim here is only to give a very high level introduction to the method used in the final experiment.

3.2.4.2 Noisy Networks

A second part to the final experiment is my use of noisy networks. This idea comes from a very recent paper by Fortunato et al. [13] in ICLR 2018. The idea here is to move beyond exploration as a random perturbation of the agent's policy (such as ε-greedy). Conceptually I consider this to be an interesting idea.

In the case of ε-greedy, decorrelated and state independent noise is added at each step. However the authors found that that a single change to the weight vector of a neural network can introduce complex, consistent and state dependent changes to policy over many time steps, thus the authors propose adding noise to the weights of the neural network, the clever wrinkle being that this noise is parameterised and those parameters themselves are being learned via gradient descent, thus creating a randomised value function from a neural network, which as the authors commment, was previously shown by Osband et al. [32] to be a provably efficient means of exploration.

I won't go into further details which again can be found in [13]. This is quite new and an area I am currently getting up to speed with. Indeed the final model I did not implement directly from scratch from the academic paper but instead used the excellent https://github.com/higgsfield/RL-Adventure, which contains some excellent implementations in Pytorch of advanced RL algorithms. So bar slightly adjusting and understanding the model, my only coding implementation here was to adapt my environment and code and test setting to be able to inherit from the OpenAi (https://openai.com/) reinforcement learning testbed. The goal was simply to try a very recent RL algorithm on the Merton problem which contained a novel method of exploration.

Chapter 4

Implementation

In this chapter I will highlight the design and implementation details. A significant part of this lay in shaping rewards for the RL agent. Also I shall lay out my test metrics, as mentioned earlier one of the issues for this particular problem is noise, resulting in even performance testing being non-trivial. For example over several episodes a completely random agent might out-perform a Merton optimal agent. I chose to create test distributions of utilities, rewards, wealth and Sharpe ratios, here we could see a clear differential between the performance of the different agents and successful tests would involve the trained RL agent closely matching the distribution of the Merton optimal one.

4.1 Reward Shaping

I need to reformulate the terminal utility of wealth at time T, into a series of step by step rewards. I began with a logarithmic utility curve because here things are simple, for example:

$$\begin{aligned}
U(W_T) &= log(W_T) && (4.1)\\
&= log(W_0(1+r_1)(1+r_2)\dots(1+r_T)) && (4.2)\\
&= log(W_0) + log(1+r_1) + \cdots + log(1+r_T) && (4.3)\\
&= log(W_0) + log(\frac{dW_1}{W_0}) + \cdots + log(\frac{dW_T}{W_{T-1}}) && (4.4)
\end{aligned}$$

Where dW_t is the change in wealth from period t-1, to t and r_t the return in the corresponding period. Notice that the final result in equation 4.1 is a purely step-wise breakdown of the original equation.

In reinforcement learning section we maximise the expected sum of (discounted rewards) i.e. $E(g_1 + \cdots + g_T)$. Therefore the step-wise reward for log-utility is simple:

$$g_t = log(\frac{dW_t}{W_{t-1}}) \tag{4.5}$$

Maximising the expected sum of these rewards is precisely equivalent to maximising the expected utility of terminal wealth $U(W_T)$ given above. These step-wise rewards signals for log-utility will therefore be rewards given to the RL agent to train towards end of episode utilities.

4.1.1 Discretisation - terminal utilities and step-wise rewards

Merton's formula exists in a continuous time world. By discretising I will move away from an exact continuous time setting to an approximation. For initial experiments involving tabular agents tests involved a simple low dimensional state action space. The farthest away from the pure Merton world is laid out below.

The simplest experiment comprised a geometric brownian motion with drift $\mu = 0.11$ and a very low of volatility $\sigma = 0.028$, I discretised to 20 time steps (the agent had to make investment decisions at each time step), each episode starting with a stock price $s_0 = 1$. Ten sample realisations of the GBM are shown in figure 4.1.

For log-utility the formula for the continuous-time optimal Merton ratio was derived earlier. That is $u^* = \frac{\mu - r}{\sigma^2}$. In this case this implies that the optimal solution involves levering up and investing 12.75x his wealth in the risky stock at every step.

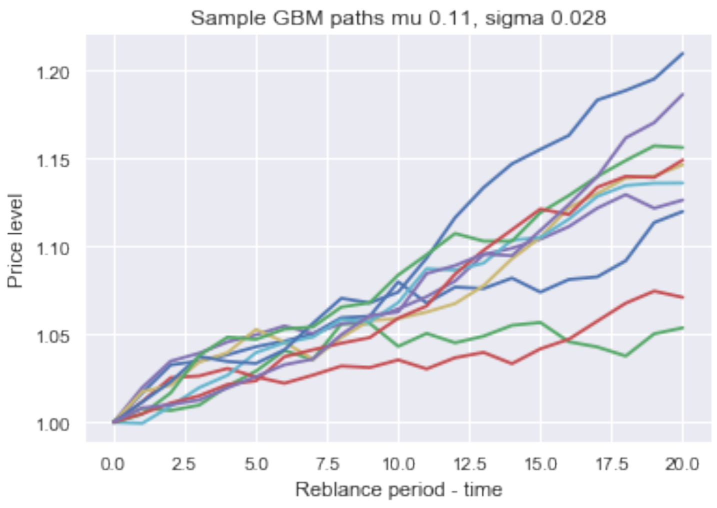


Figure 4.1: Low Volatility Geometric Brownian Motion Paths

To check the validity of the formula in this low dimensional discretised setting, I simulated various fixed ratio investment strategies and plotted the mean of their end of episode utilities. Figure 4.2, shows the results for a range of possible investments from -10x to 25x levered. Despite the discretisation, the simulation closely matches the analytic solution given by Merton, with the maximum average utilities achieved by an optimum investment ratio close to merton's analytic solution of 12.75x.

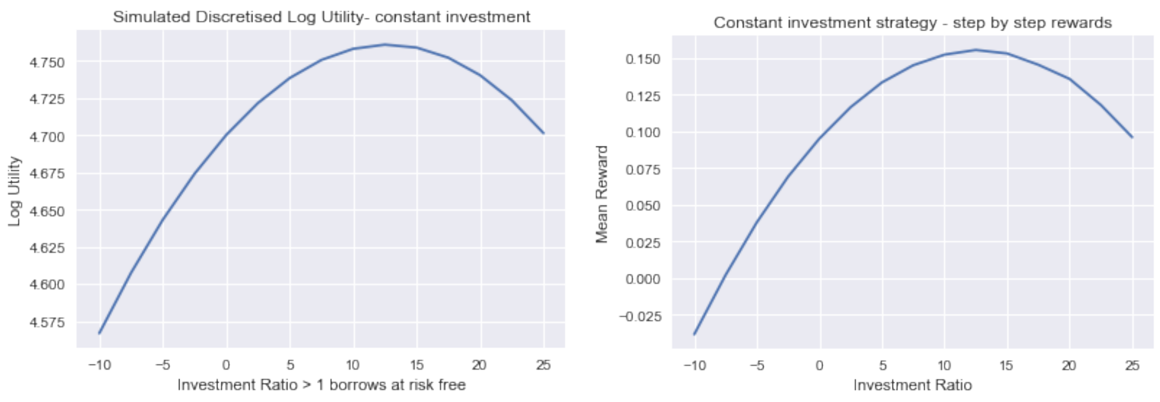


Figure 4.2: Discretised Log Utility versus Step-wise rewards

The right chart in figure 4.2 shows the same investment strategies now tested in the step-wise market environment using the rewards I derived earlier $r_t = log(\frac{dW_t}{W_{t-1}})$.

One would hope that the two charts match closely and that the optimum investment ratio for step-wise rewards closely matches that for end of episode utilities, figure 4.2 shows that happily this is the case.

4.1.2 Mean Variance Equivalent Distributions, Episodic

For most utility curves we will not be fortunate enough to move so smoothly from end of episode utilities to step-wise rewards. In Ritter's [35] work last year, he trained a Q-learning agent to learn to trade an Ornstein Uhlenbeck process with transaction costs, the technique he used was closely related to Chapter 2 where I describe mean-variance utilities. Thus in order to generalise beyond log-utilities, I decided to use Ritter's approximation and apply this to the Merton problem. This will be done in a two-stage process, in this subsection I shall convert utilities into an end of episode mean-variance form, in the next I shall take a step-wise approximation to this.

A random variable $\mathbf{r}$ follows a **mean-variance equivalent** distribution if it has a density $p(\mathbf{r})$, which has first and second moments and where for any increasing utility function u, there exists a constant $\kappa > 0$, such that the policy maximised by $\mathbf{E}(u(w_T))$ is also optimal for the simpler problem:

$$\max_{\pi}[\mathbf{E}(w_T) - \frac{\kappa}{2}\mathbf{V}(w_T)] \tag{4.6}$$

Given this equation one can see the connection with Markowitz's work, [24], and also mean-variance utility laid out in the finance section. The difference here being that we have added an extra parameter κ, a risk aversion parameter, this reshapes our distribution to approximate the utility curve. Note that although the optimal policy will match the optimal policy in terms of utilities exactly, the whole curve need not necessarily match as we are now using a second order approximation.

In figure 4.3, I again ran the simulation for various fixed investment ratios. However this time I use the above mentioned, mean-variance approximation. In this case my risk aversion parameter $\kappa = 0.006$, one can clearly see that the optimum investment ratio closely matches the curves in figure 4.2 for both utilities and rewards, however with the advantage that we are now no longer tied to log-utility.

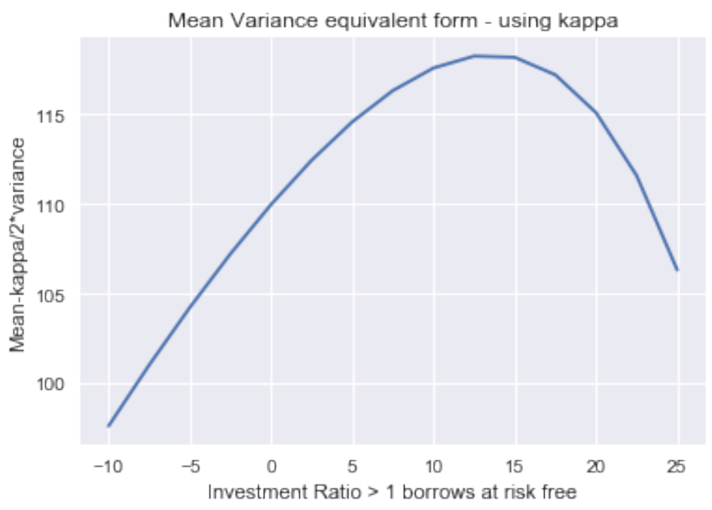


Figure 4.3: MV equivalent approx to Log-Utility, $\kappa = 0.006$

However this mean-variance equivalent approximation still leaves us in an episodic world with figure 4.3 illustrating end of episode results, so the final step is to enable step-wise rewards, here I follow Ritter [35], as illustrated below.

4.1.3 Mean-Variance equivalent distributions, Step-wise

To recap we have moved from Utility curves, to end of episode rewards, to step-wise rewards suitable for a myopic utility such as log-utility, to mean-variance equivalent distributions, this section illustrates how to turn this into an environment with step-wise rewards suitable for a reinforcement learning agent but for any utility curve.

Given the mean-variance equivalent form used by Ritter [35], we have that:

$$\max_{\pi}[\mathbf{E}(w_T) - \frac{\kappa}{2}\mathbf{V}(w_T)] \tag{4.7}$$

Thus we need to calculate $\mathbf{E}(w_T)$ and $\mathbf{V}(w_T)$, in a step-wise reward fashion. The expected value is straighforward:

$$\mathbf{E}(w_T) = w_0 + \sum_t \mathbf{E}(\delta w_t) \tag{4.8}$$

Where δw_t is the change in the agent's wealth in one time-step at time t. For the variance if δw_t is independent of δw_s when $t \neq s$, then we have that:

$$\mathbf{V}(w_T) = \sum_t \mathbf{V}(\delta w_t)$$

Thus the problem we need to solve has been reformulated as something far closer to something that can provide step-wise rewards, that is:

$$\max_{\pi} \sum_t [\mathbf{E}(\delta w_t) - \frac{\kappa}{2}\mathbf{V}(\delta w_t)] \tag{4.9}$$

If we define our reward as follows:

$$r_t = \delta w_t - \frac{\kappa}{2}(\delta w_t - \hat{\mu})^2 \tag{4.10}$$

where $\hat{\mu}$ is the estimate of the change in our wealth over one time-step. Then:

$$\mathbf{E}(r_t) = \mathbf{E}(\delta w_t) - \frac{\kappa}{2}\mathbf{V}(\delta w_t) \tag{4.11}$$

which is exactly what is required. However as Ritter [35] points out, there is a circularity in that we do not know $\hat{\mu}$. But clearly unless the Sharpe ratio of the strategy is very high and given small enough steps then we have a very close approximation:

$$\mathbf{E}(\delta w_t - \hat{\mu})^2 \approx \mathbf{E}((\delta w_t)^2) \tag{4.12}$$

The added benefit of this is that it is conservative as regards risk (because we are slightly over-estimating variance based upon a zero expected change in wealth). If one does wish to estimate $\hat{\mu}$, then we can train our reinforcement learning agent using the above formulation and after convergence, use the samples to estimate $\hat{\mu}$ and then simply retrain again including this estimate in our reward.

We have now solved the problem of creating step-wise rewards. Specifically if our reward at time t:

$$r_t = \delta w_t - \frac{\kappa}{2}(\delta w_t)^2 \tag{4.13}$$

Then in a reinforcement learning setting where we wish to maximise the sum of rewards, we find that:

$$\begin{aligned} \mathbf{E}(G_t) &= \mathbf{E}(r_{t+1} + r_{t+2}+, \ldots, r_T) && (4.14) \\ &= \sum_{s=t}^{T} \mathbf{E}(\delta w_s) - \frac{\kappa}{2}\mathbf{V}(\delta w_s) && (4.15) \end{aligned}$$

This is exactly what is needed for a step-wise reward formulation equivalent to maximising the terminal utility of wealth.

I again tested this implementation for a number of fixed investment ratio strategies and examined the mean rewards in order to check that this does indeed give a similar optimal investment to the original analytical Merton solution, figure 4.4 shows that this is indeed the case.

This step-wise approximation includes a risk aversion parameter κ which matches the risk aversion to the utility of the investor. We can adjust our κ to simulate different risk aversions and thus different utility curves. One extreme would be a risk neutral investor, that is $\kappa = 0$. We should hope to see a linear utility as the investor levers up. The simulation performed in Figure 4.5, shows that this is indeed the case.

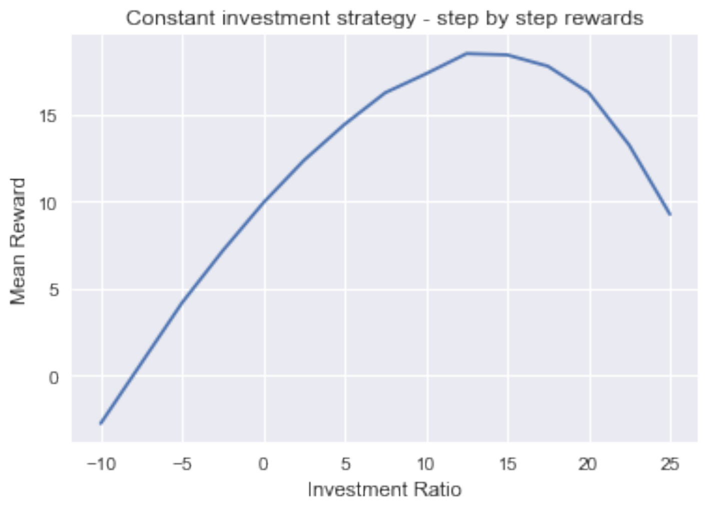


Figure 4.4: Mean Variance equivalent approximation to log-utility

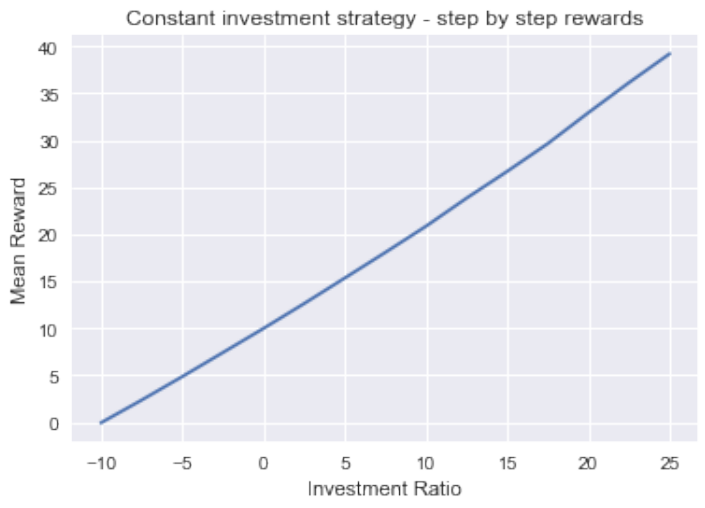


Figure 4.5: $\kappa = 0$, does this match risk neutrality?

4.1.4 Reward Shaping Conclusion

To conclude, we can now generate rewards from a market environment in a step-wise manner suitable for an RL agent approximating any utility curve. We first looked at end of episode utilities, then derived a simple step-wise reward for log utility, then generalised this into a mean-variance formulation albeit on an end of episode basis and finally created a mean variance approximation on a step-wise basis. These rewards should enable an RL agent to learn the optimal policy for a variety of stochastic processes and utility curves.

4.2 Design

In terms of design I created an episodic environment for training and test reporting purposes. Note that for the RL agent this is not actually necessary as the agent learns online step by step (I do not need to reach the end of an episode to learn, unlike for example Monte Carlo learning) thus the set-up can easily be reformulated into an a continuing rather than episodic environment. However by creating an episodic environment I can easily produce distributions of test results.

On an unseen test-set I examine the performance of a trained RL agent, against a Merton optimal agent and a random agent. Because we are dealing with stochastic processes, examining even several realisations of these time series will not give definitive answers. For example in any given episode a random agent can easily out-perform a Merton optimal agent, however by re-running millions of times then we can instead examine the distribution of test results.

From an RL viewpoint states comprise price and wealth levels. The agent is a price taker and so does not influence the future market price at all. The agent's action at each step is to choose a proportion of his wealth in the risky asset and similarly the risk-free bond (he also has the choice to lever up and borrow money and also to go 'short'). The agent will receive a 'reward' from the environment at each time step and find himself in a new state provided by the market environment. Clearly when I scale up using a neural network I am no longer forced to use a tabular setting (i.e. buckets of wealth or prices) and indeed in reality could easily add further features such as valuation metrics, news items or whatever else we might believe holds some predictive promise.

4.2.1 Market Environment

The market environment can be interacted with on a step-wise basis by an RL agent. It comprises two financial instruments. A risk-free bond and a risky asset (for example geometric brownian motion, ornstein-uhlenbeck etc.). The market environment is episodic, thus comprises multiple time periods T (a parameter we can of course vary). After each episode concludes we begin again with a new realisation of the stochastic process and bond and of we course reset the agent's wealth level. For the earlier tabular agents I discretise into 100-150 wealth or price levels, with 15 actions at each time step, with episodes lasting between 20 and 50 time periods. In my later implementations using neural nets states are simply raw inputs and can thus be of very high dimensionality, I also increased the number of actions to 25 at each time step. In my final experiments I found it useful to port my environment to be able to inherit from the OpenAI gym, which has useful generic features encoded.

4.2.2 Agent

The agent is of course a reinforcement learning algorithm. The agent at any given time is in a certain wealth state, depending on the information available (prices, and other features). The agent knows nothing about the underlying price process of the market, nor the type of stochastic process or any parameters of its parameters, similarly it knows no details about the risk-free bond.

The agent can only interact with the environment by providing an order, this order based on the agent's current wealth with be received by the market and executed, the order comprises an instruction to invest a certain amount of the agent's wealth in the risky asset and a certain amount in the risk-free bond. After receiving the order, the market will take a step, returning a new prices and a new wealth state, it also returns the reward described earlier. The agents goal will be to use these rewards to attempt to learn to trade close to Merton optimality.

The tabular reinforcement learning agents were: Q-Learning, Double-Q Learning, Double-Sarsa and Dyna. The initial goal was not to over-complicate the RL side of the implementation, where obvious extensions are easily added. My initial state space comprised 100-150 wealth buckets, with 15 actions covering a wide variety of investment possibilities at each time step from -10x short to 25x long the risky asset in the most extreme example. Thus, despite the reduced size, even the smallest state-action space still covers over 1500 possibilities, with multiple time steps. The number of time-steps for each episode was 20 steps, however 50, 100 or more time steps was also viable, but just slower to train. Even over 20 time steps there are 15^{20} possible policies for the agent to decide between, each episode, so plenty of chances to go wrong.

After scaling up, states are raw inputs, that is we can simply provide, prices, wealth states or whatever other features may be useful as inputs into the neural net. Here I tested DeepQ, DoubleDeepQ and Noisy nets. I used the Pytorch library and the Adam optimiser. It should be noted that I also used the excellent https://github.com/higgsfield/RL-Adventure, for some of the code. This is an advanced tutorial implementing several recent papers in Reinforcement Learning using Pytorch, in particular the implementation of Noisy nets came from here. In addition I also found it useful to port my environment to inherit from the OpenAI gym when scaling up.

4.3 Test Metrics

4.3.1 Noise intuitions - Merton and Random agents episodically

Before moving on to the test methodology, I will describe an initial issue and how it informed my test metrics. Note that all simulations in this section were performed with 200,000 simulations.

Perhaps naively, my first thought was that the Merton problem would be solved quite cleanly given Ritter's [35] formulation of the mean reverting trading problem for Q-Learning. I had slight concerns that the set up was part-way between a bandit setting and a multi-state RL setting. My concern was that the agent has little control of much of his environment, he is simply a price taker. However, the agent can potentially control the future distribution of his utility of wealth through his actions, so it didn't appear problematic.

In initial implementations the trained agent performed poorly. I therefore simplified the state space dimensionality to only 100 buckets of wealth and only 15 investment actions. I began with 1000 time steps for each episode (to minimise the impact of discretisation), and eventually reduced to 20 (due to training time on a 3Ghz laptop). Even after these simplifications there were learning problems, there are of course hyper-parameters to adjust and some time was wasted there - but the real issue was quite basic: noise.

I reduced the volatility of the geometric brownian motion. This has the disadvantage that we move further again from a real market problem and instead shift to the question of how much should I borrow to invest in this now not very risky stock. With a far lower volatility than one would find in an actual market the Q-learner successfully began to learn. The crux of the problem is shown in Figure 4.6. The left chart shows the distribution of terminal utilities, the right the distribution of terminal rewards. What is difficult to see is that I am actually showing rewards and utilities for two opposite strategies, Merton optimal and a random agent, i.e.

the distributions for the best performer and the worst are actually very similar on an episodic base. Given this fact it is unsurprising that an RL agent needed to use many training episodes to learn a strategy better than random.

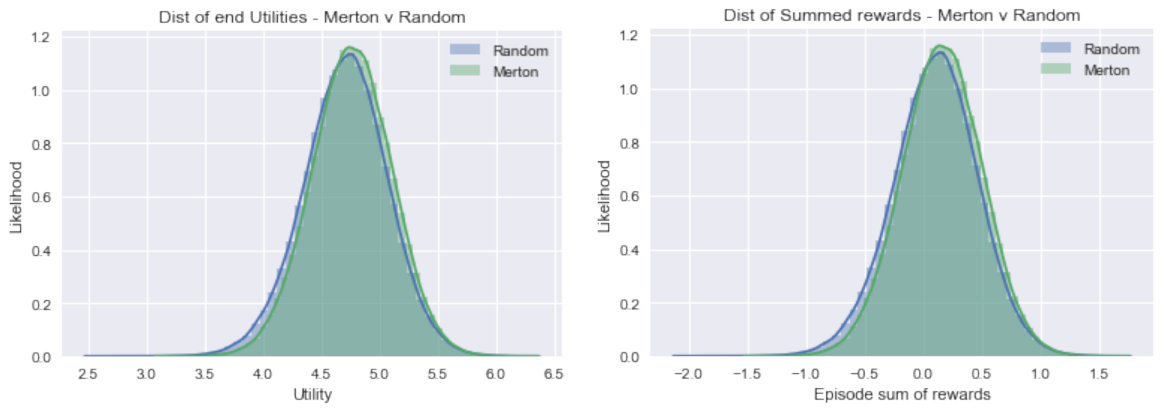


Figure 4.6: Episodic distribution of final utilities and rewards, Random v Merton

Recall that the agent does not actually receive the benefit of episodic rewards, the rewards are received on a step-wise basis. Figure 4.7 illustrates the actual step-wise rewards received by a random agent compared to a Merton optimal one, again the differences are minimal.

The training solution for tabular Q-learning was brute force - more episodes of training (more sophisticated methods such as experience replay, prioritised experience replay and noisy nets were introduced later), the RL agent was trained over several million episodes rather than several thousand. However the difficulty in differentiating performance on an episodic question also begs the question as to how to test performance.

4.3.2 Test Metric 1: Utilities

I therefore decided to test performance not on an episodic basis, but grouped over 1000 episodes. I would then batch another 1000 episodes (so they are independent) and repeated this process 3000 times. Thus 3,000,000 episodes in 3,000 batches.

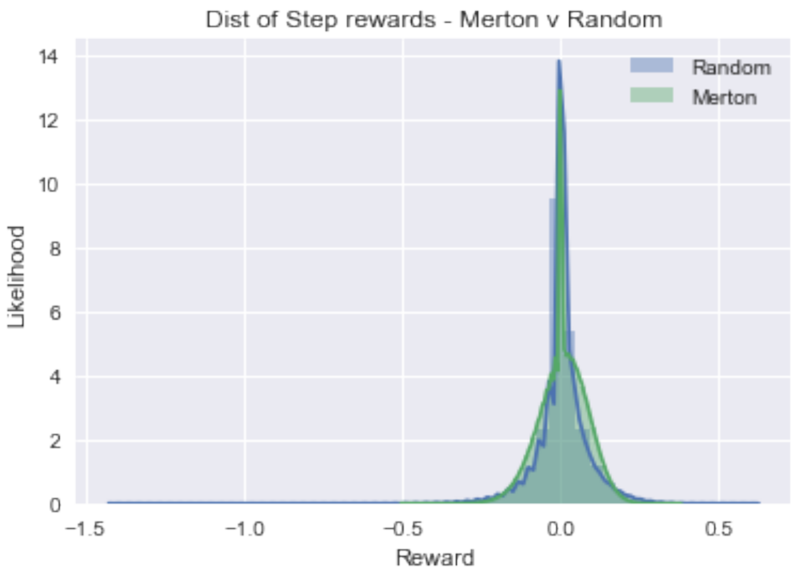


Figure 4.7: Stepwise rewards Merton optimal versus a Random agent

Figure 4.8 shows the difference in performance between a Merton optimal agent and a random agent using this revised test metric.

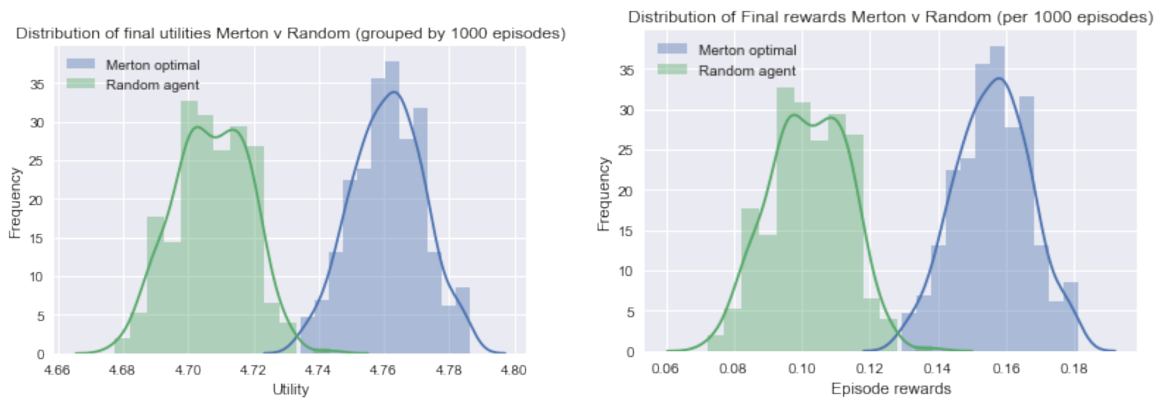


Figure 4.8: Batched utilities and rewards, random agent versus Merton

We can now see a clear differentiation in performance for both rewards and utilities between a random agent and a Merton optimal one. For test purposes the RL agents were compared to Merton optimal and Random using the above methodology with the agent acting greedily with respect to the trained Q-values.

4.3.3 Test Metric 2: Rewards

The right hand chart of figure 4.8, shows the rewards for a random agent and a Merton optimal. One would hope that this looks very similar to the utility chart if the rewards are matching utilities well, which in this case it does. However it can be the case that the mean-variance approximation is a poor approximator of utilities, in which case the agent is being trained towards an incorrect target. Therefore I also included rewards as a further test metric to shed further light on the performance of the RL agent.

4.3.4 Test Metric 3: Wealth

The question, 'Why not just show who made the most money?', is absolutely the wrong question. As an ex-fund manager, it is a slip I sometimes make. The object of this game is to learn to trade close to a Merton optimal manner, which involves learning a policy that maximises the expected utility of terminal wealth, not terminal wealth itself. The two are only the same if we are risk neutral (and thus have a linear utility curve). However examining the distribution of end of episode wealths remains interesting, particularly when we wish to examine a risk neutral verses a risk-averse agent to see if the agent improves its wealth distribution at the cost of risk.

4.3.5 Test Metric 4: Sharpe Ratio

The Sharpe ratio $\frac{\mathbf{E}(r-r_f)}{\sqrt{\mathbf{V}(r)}}$, where r is the return of our strategy and r_f and risk free rate is a well known metric in finance named after the Nobel Laureatte William Sharpe. I chose a simpler variation $\frac{\mathbf{E}(r)}{\sqrt{\mathbf{V}(r)}}$, where for each episode I calculated the episodic return and its standard deviation, in order to build a distribution of episodic 'Sharpe' ratios. The reason for reporting the distribution of Sharpe ratios was twofold. First this is a common and accepted metric in finance to examine the performance of strategies on a risk adjusted basis, and second because I wished to examine any cases where the agent actually did very well on a Sharpe ratio basis, but perhaps less so as regards final utilities, for example extreme risk aversion by hiding in the risk-free bond.

Chapter 5

Experiments

In this chapter I will describe the experiments performed and their results. There will be three groups of experiments.

Experiment Group 1

The first will be to attempt to train an RL agent with a performance level somewhere near a Merton optimal agent. The initial setting will be a discretised low-volatility market with log-utility. From here I will increase the volatility and generalise towards other utility functions. The step-wise reward formulation will be examined, one of the benefits of the step-wise approximation is that risk aversion can be tailored using κ, I will demonstrate this with an RL agent trained towards both risk neutrality and extreme risk aversion.

Experiment Group 2

The second experiments are designed to test if the agent can learn under more market realistic conditions, to this end I first calibrate the brownian motion to levels of drift and volatility exhibited by the S&P 500, and also lower the risk-free rate to match a more realistic long bond rate. From here I add very high levels of volatility and also introduce 'fat tails'. Here the agent comes into its own and more than matches a merton optimal agent.

Experiment Group 3

In the third group of experiments I add further realism by scaling up. Instead of using tabular RL agents and being forced to discretise the state space, I now use neural nets as a function approximator. DeepQ, Double DeepQ and Noisy nets were tested. Not only could price and wealth levels be input as raw data but clearly this setting has the capability of including other features. Experience replay and alternative exploration methods were also introduced here, with the benefit of learning through fewer episodes.

5.1 Experiment group 1: Learning to trade like Merton, Double-Q

In the following experiments I trained Q and Double-Q learners, using 3,000,000 training episodes with a step-size of 0.1 and ε- greedy exploration of 0.1. γ was set to 0.95. The test methodology was as described earlier using 3,000 batches of 1,000 episodes, thus comprising an unseen test-set of 3,000,000 episodes. T was set to 20 periods, purely for convenience as regards training time, (50, 100 or even more time periods had similar results but of course longer training times).

In each case test distributions will be produced showing where appropriate, utilities, rewards, wealth and sharpe ratios. The 'straw man' random agent will be compared to Merton optimal, as well as the trained RL agent.

5.1.1 Log-Utility

For the first test I used the exact myopic formula for step-wise rewards that I derived at the beginning of Chapter 3 on reward shaping (not yet the mean variance equivalent approximation used by Ritter [35]).

5.1.1.1 Log-utility: Description

The agent had 15 choices over a wide range for each of 20 time-steps, from investing -10x in the risky asset, to 25x leveraged. The agent could also choose how much of its current level of wealth to invest in a risk-free bond at each time step. The risky asset was a geometric brownian motion with a drift $\mu = 0.11$, and a low volatility $\sigma = 0.028$, the risk free bond had a rate of interest $r = 0.1$.

In this particular experiment we have a very small difference between the drift of the risk-free rate and the GBM. The GBM is also very low volatility, the solution from a Merton optimal viewpoint would be to lever up to 12.5x one's wealth at every time-step and invest all of this in the risky asset. The differences in the rewards between Merton optimal and random are also very small as mentioned above.

5.1.1.2 Log-utility: Results

Results are given in figure 5.1, the top row of charts shows the distributions of test utilities as well as a violin plot of the same. The Merton optimal agent as one would expect has a higher expected utility than the random agent, with the distribution further to the right. The double-Q learner is far better than random but not quite at the level of the Merton optimal agent. It has certainly learned to do way better than random, but is not perfect.

The second row shows the rewards, in this case the results are basically identical to that of utilities as I am using my exact formula for log-utility step-wise rewards and no approximation. The third row gives an interesting aside, the wealth distribution of the double-Q learner is higher even than the Merton optimal agent. If one examined results only from this naive viewpoint then one would argue that the Double-Q learner has outperformed even the Merton agent. However, this comes at a cost.The final row gives the distribution of Sharpe ratios for the three agents. The Merton optimal agent has the highest Sharpe ratios, thus showing that the strong final wealth results of the Double-Q learner came at the cost of taking on greater volatility, damaging both Sharpe ratios and final log-utilities.

5.1.1.3 Log-utility: Conclusion

Figure 5.2 shows the 10,000 episode moving average of the utilities on the test set. With tables 5.1, 5.2 showing the mean and standard deviations of the various test statistics.

In conclusion I would argue that the double-Q learner has not quite matched the performance of a Merton optimal agent in this particular case, but has still shown a very good performance, certainly way better than a random agent. The distribution of utilities and rewards is slightly worse than Merton optimal, with the slight surprise that the distribution of wealth is actually higher. Thus the agent has learned to generate more wealth even than the Merton optimal agent, but at the cost of higher volatility as evidenced by a lower sharpe ratio and log-utility.

Due to the stochastic nature of the risky asset as well as double-Q learning itself having stochasticity embedded within its exploration, it is important to note that experimental results can also vary. Notice also in table 5.1 that the utility distribution and rewards distribution are basically identical, indeed if we added the utility of the initial wealth, $log(w_0 = 100)$ to the sum of the rewards then they would be exactly the same. This is of course due to my using an exactly derived step-wise reward in the case of log-utility, rather than the mean-variance equivalent approximation as mentioned earlier.

My interpretation is that the results in this experiment are good, but not perfect. The agent still hasn't performed quite at the level of a Merton optimal agent utility-wise. Before generalising to mean-variance equivalence I shall now test the double-Q learner on a higher volatility problem, both to add greater realism and to add comfort that the performance on this particular problem was not some lucky coincidence.

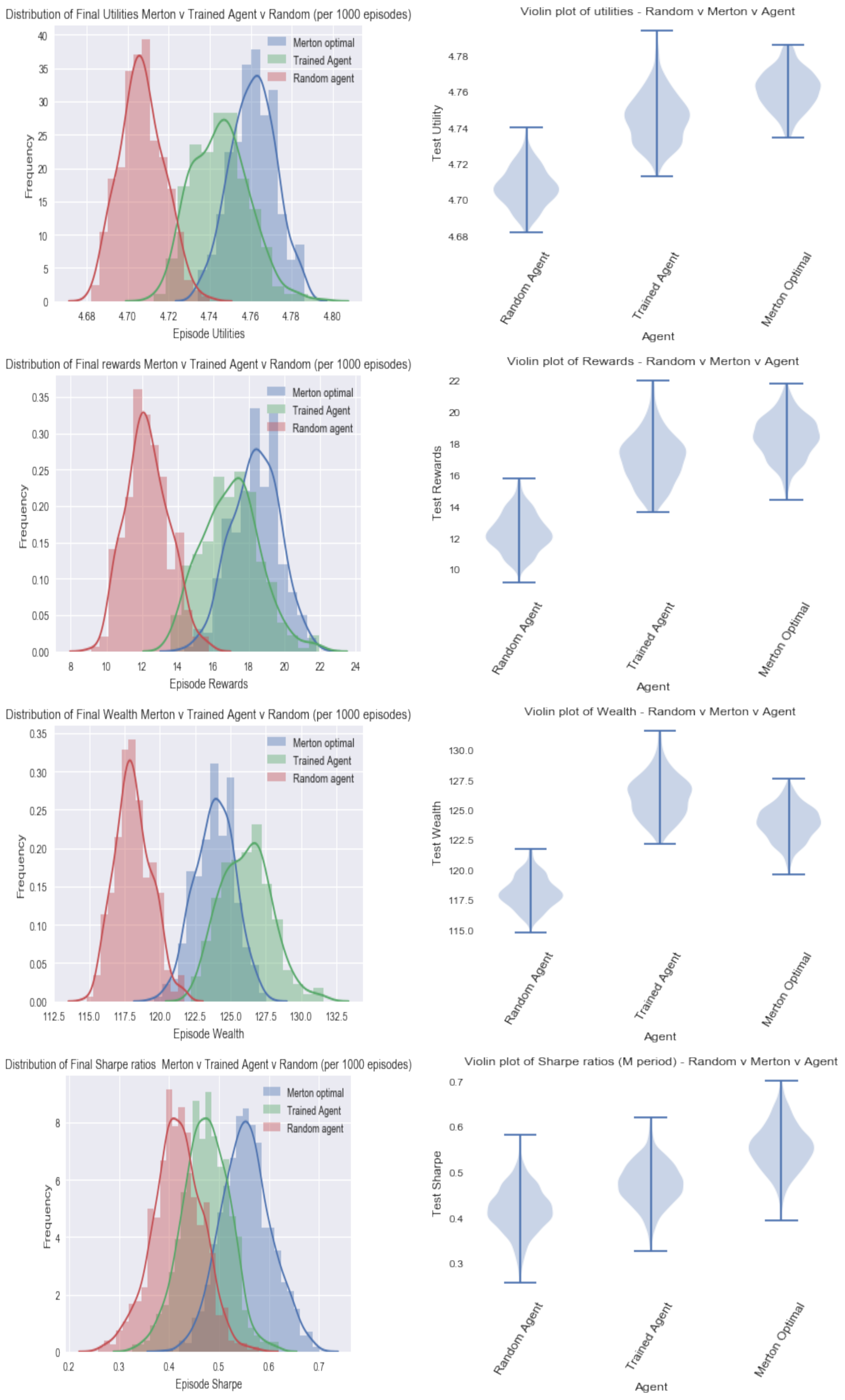


Figure 5.1: Double-Q learning: Utilities verses Merton and Random agent

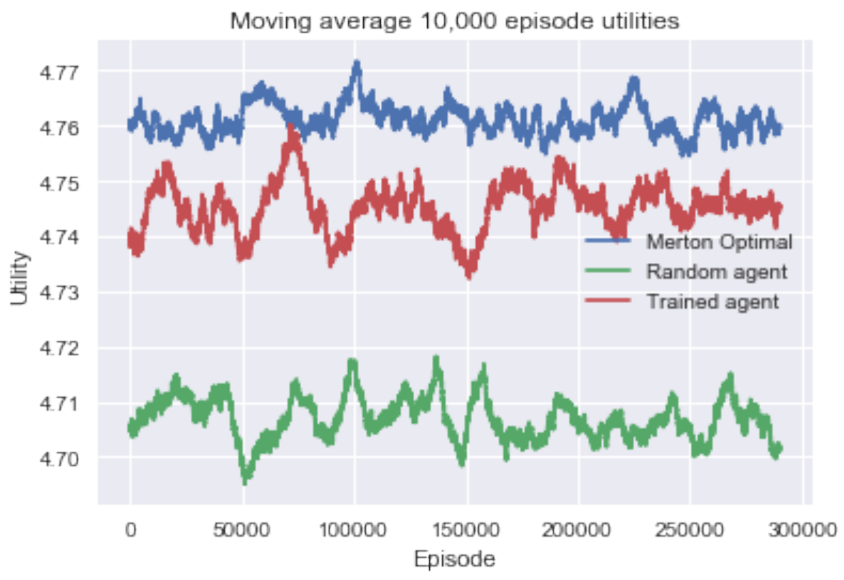


Figure 5.2: Moving average utilities Double-Q v Merton v Random

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	4.7062	4.7442	4.7611
Rewards Distribution	0.1010	0.1390	0.1559
Wealth Distribution	118.07	125.99	123.87
Sharpe Ratio Distribution	0.4145	0.4708	0.5544

Table 5.1: Double-Q, test performance v Merton - Mean

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	0.0118	0.0138	0.0108
Reward Distribution	0.0118	0.0138	0.0108
Wealth Distribution	1.3946	1.7390	1.4035
Sharpe Ratio Distribution	0.0493	0.0459	0.0501

Table 5.2: Double-Q, test performance v Merton - Std

5.1.2 Increasing the Brownian Motion volatility

5.1.2.1 Brownian Motion, medium volatility: Description

Every parameter in this new problem formulation is exactly the same bar one - volatility, for our risky asset we remain with a geometric brownian motion with a drift $\mu = 0.11$ and a risk-free asset with $r = 0.1$, the volatility of the GBM σ I now increase nearly fourfold, from 0.028, to 0.1. This will of course reduce the merton optimal investment ratio.$\frac{\mu - r}{\sigma^2}$. The solution now in Merton optimal terms is to remain fully invested with a constant 100% investment in the risky asset, but to not lever up further, not go short and not reduce risk by owning the risk-free bond.

Figure 5.3, illustrates some sample paths for the new stochastic process. Because the process is more volatile we are likely to experience a wider range of price states in the market and wealth states for our RL agent, however the agent remains with the action space given above, that is 15 possible investment actions at each time step. Because the merton optimal solution for the investment ratio has now reduced considerably (from 12.75x to 1x), I have changed the range of actions to being short 100%, to being long 2.5x. This is far closer to more realistic leverage ranges that many hedge funds might use and feels somewhat less contrived than the previous experiment.

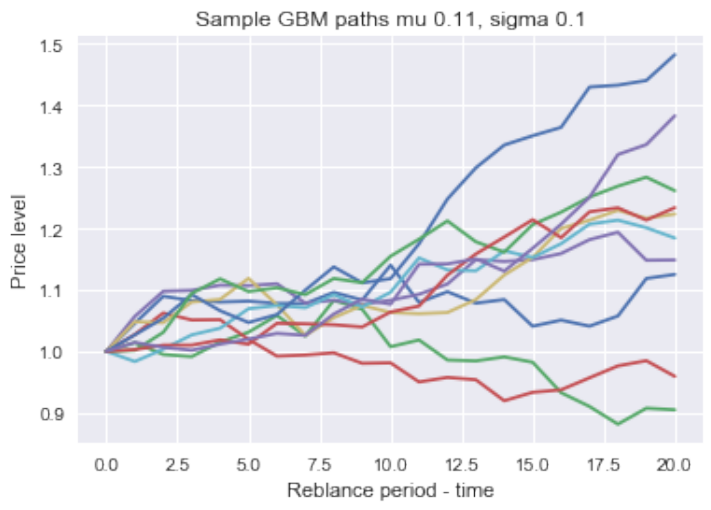


Figure 5.3: Medium Volatility Geometric Brownian Motion Paths

5.1.2.2 Simulated utilities

The environment has now changed, so I tested the step-wise implementation again. Figure 5.4, shows the new results achieved by simulating the environment 200,000 times for a range of constant investment strategies. The left chart shows that the peak episodic log-utility from simulation again matches the theoretical merton optimal ratio of investment (i.e. 1x). The right chart shows that the conversion of this into step-wise rewards run under the same simulation gives a very close match. Thus giving confidence that a reinforcement learning agent will again be able to learn from these step-wise rewards.

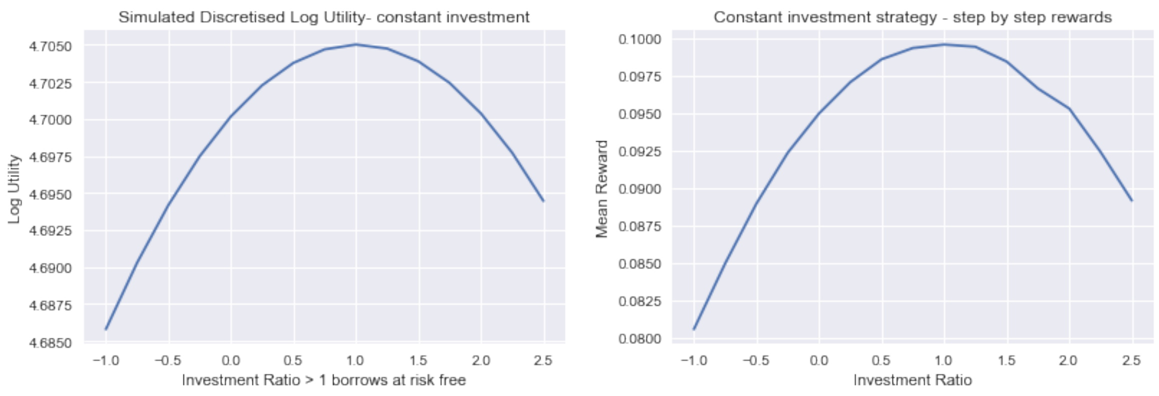
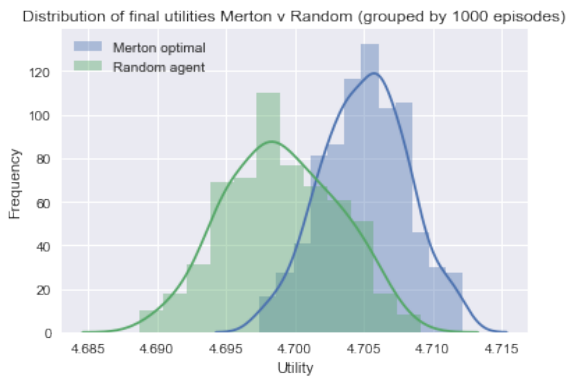


Figure 5.4: Discretised Log Utility versus Step-wise rewards - medium vol

5.1.2.3 Merton optimal v Random - rewards

In a similar fashion to the earlier low volatility problem, figure 5.5 shows the comparison of the actual rewards for a Merton optimal versus a random agent. Notice that compared to the previous example the higher volatility environment has now resulted in the two distributions overlapping somewhat, thus making this a potentially tougher environment for the RL agent to learn. Again we hope that when the RL agent is trained it will manage to approach the results of the Merton optimal test distribution.



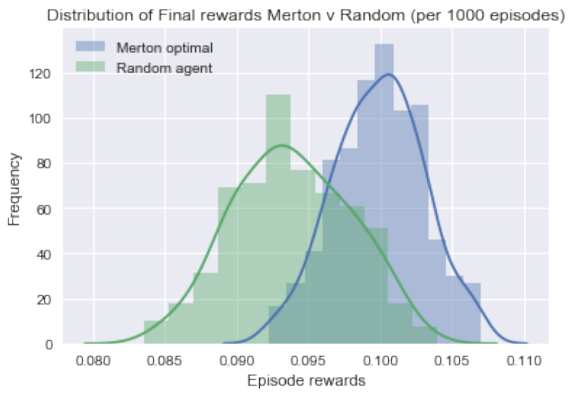


Figure 5.5: Distribution of batch Utilities Merton optimal v Random - medium vol

5.1.2.4 Brownian Motion, medium volatility: Conclusion

Figures 5.6, 5.7 and table 5.3 illustrate the results of the higher volatility experiment. I am again using the exactly derived log-utility rewards (not the mean-variance approximation) so I leave out the rewards results (due to being superfluous).

I think these results are excellent. The RL agent's test set distribution for utilities and wealth is very close indeed to that of the merton optimal agent. Similarly for the moving average of utilities. The performance of the double-Q agent is actually close to indistinguishable from the merton optimal agent bar the fact that it experienced a greater volatility of wealth on the test set resulting in a slightly lower sharpe ratio.

In this experiment the agent does appear to have successfully approximated the performance of a merton optimal agent in a reasonably difficult case where volatility is reasonably high and the drifts of the risk free and risky asset are very similar. I will now move away from the exact log-utility reward formula towards mean-variance approximations in order to generalise to a wider range of utility functions and introduce the risk aversion parameter κ.

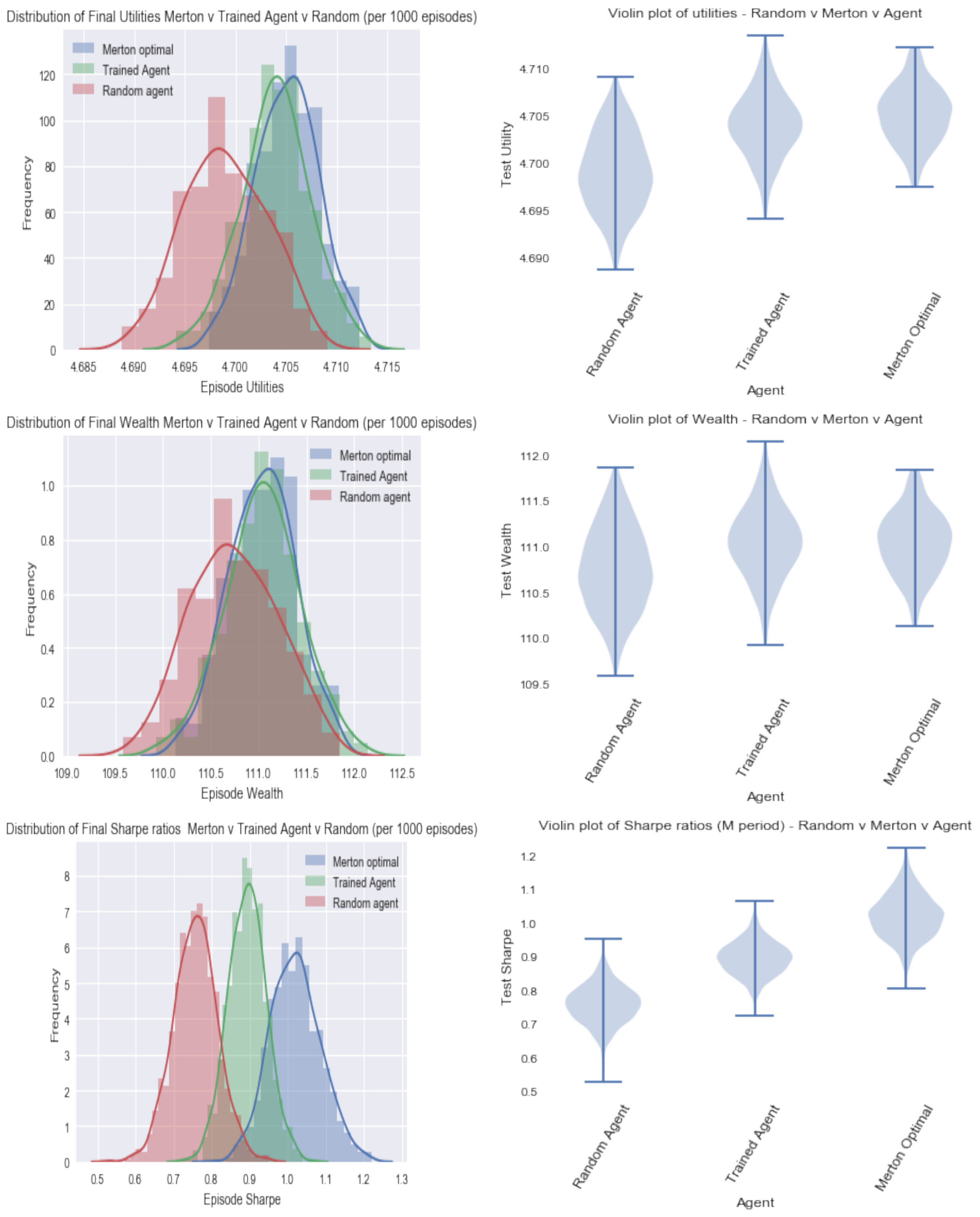


Figure 5.6: Double-Q learning: Utilities verses Merton -Medium Volatility

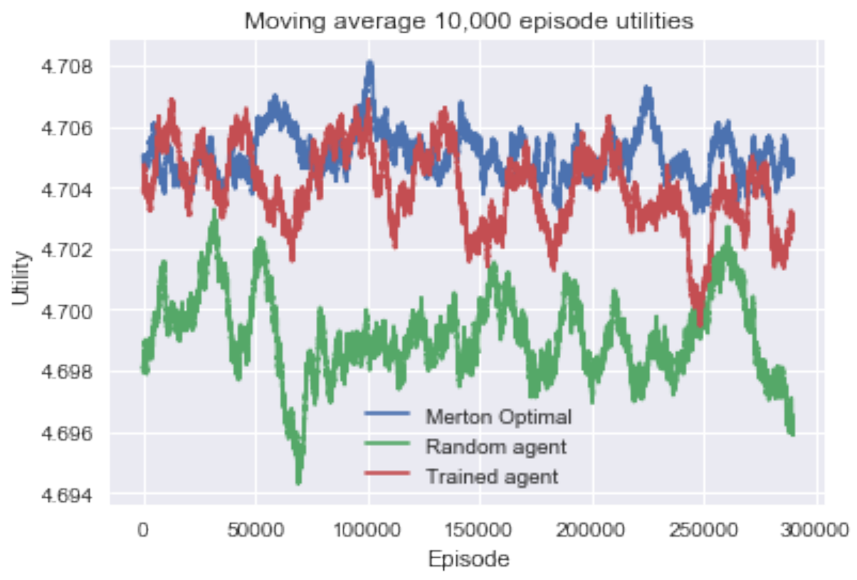


Figure 5.7: Moving average utilities Double-Q v Merton - medium volatility

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	4.6990	4.7043	4.7050
Wealth Distribution	110.74	111.09	111.03
Sharpe Ratio Distribution	0.7574	0.8961	1.0202

Table 5.3: Double-Q, test performance v Merton - Mean, Medium vol

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	0.0040	0.0034	0.0030
Wealth Distribution	0.4545	0.3963	0.3467
Sharpe Ratio Distribution	0.0581	0.0509	0.0665

Table 5.4: Double-Q, test performance v Merton - Std, Medium vol

5.1.3 Generalising to Mean-Variance rewards

5.1.3.1 Mean-Variance rewards: Description

For log-utilities I derived an exact step-wise reward function for the reinforcement learning agent, however in general this will not be possible. In this experiment I will still use log-utility and the higher volatility environment, but now introduce the mean-variance approximation derived in the implementation chapter and used by [35]. I will continue to use the double-Q learning agent.

This adjustment also introduces an additional risk aversion parameter $\kappa = 0.006$, parameterised earlier in this thesis. The key idea of κ being that as described by [35], the optimal policy of any upward sloping utility curve will match the optimal policy for some κ. From an investor viewpoint κ will also enable us to tailor our risk aversion and train the RL agent for differing risk preferences, which I shall test in a future experiment.

5.1.3.2 Mean-Variance rewards: Conclusion

Figures 5.8, 5.9 and table 5.5 shows the results using a mean-variance reward approximation for log-utility. Again these results are excellent. On the test set the agent has learned to closely match the performance of the merton optimal agent for utilities, and the final distribution of wealth.

In this particular experiment the double-Q agent has very closely matched the test utility performance of a merton optimal agent, but actually has a better sharpe ratio than the merton optimal due to its exhibiting a lower volatility of final wealth on the test set. I am unsure if this is simply due to my calibration of κ in that we are now training the agent to be risk averse in a step-wise variance sense as an approximation to utilities. However the test performance remains good. With this formulation we are now able to adjust our risk preferences κ and use this to train the RL agent towards various degrees of risk aversion, this I shall examine in a future experiment, but first I shall change the utility curve.

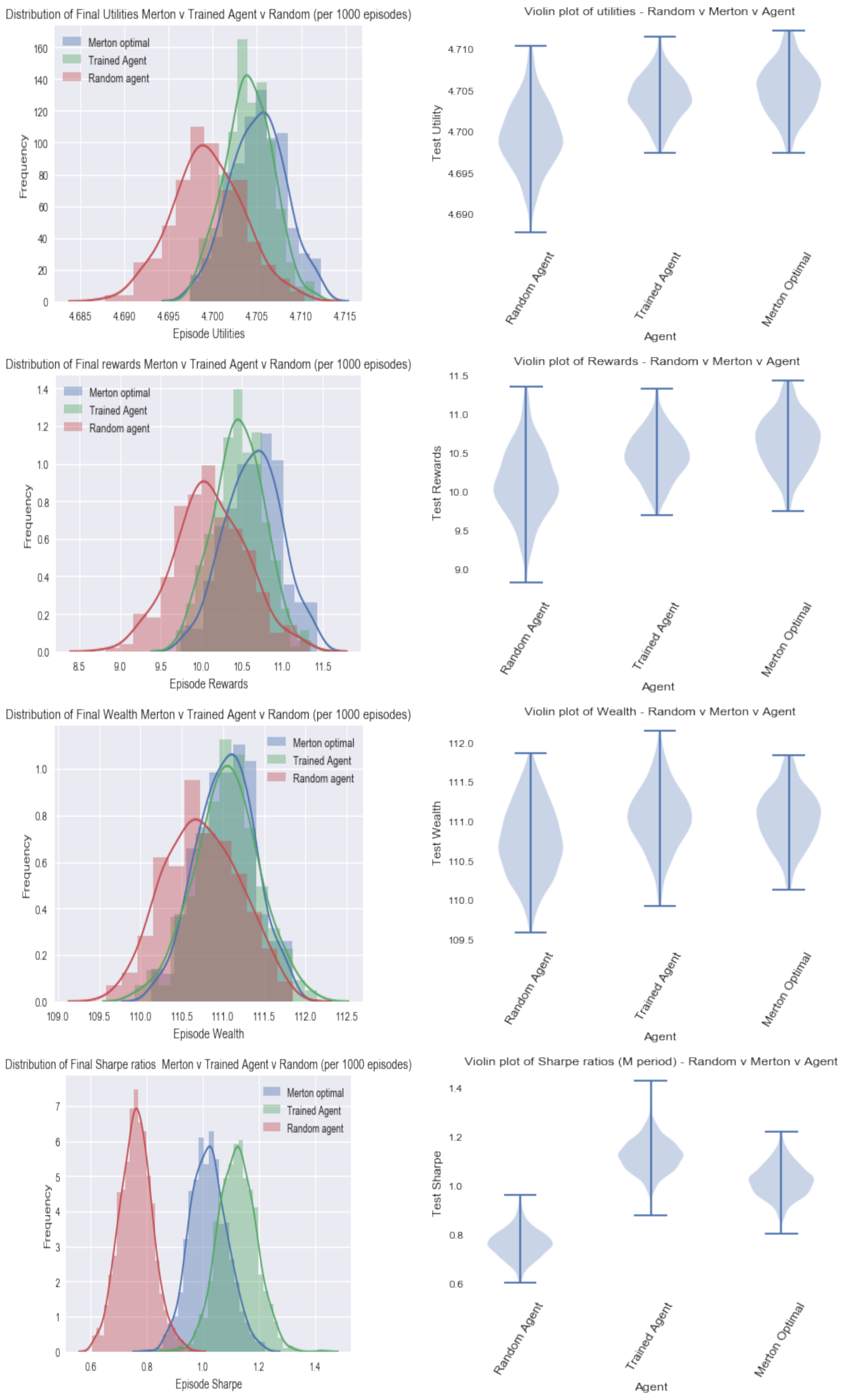


Figure 5.8: Double-Q learning: Utilities, MV-equivalent

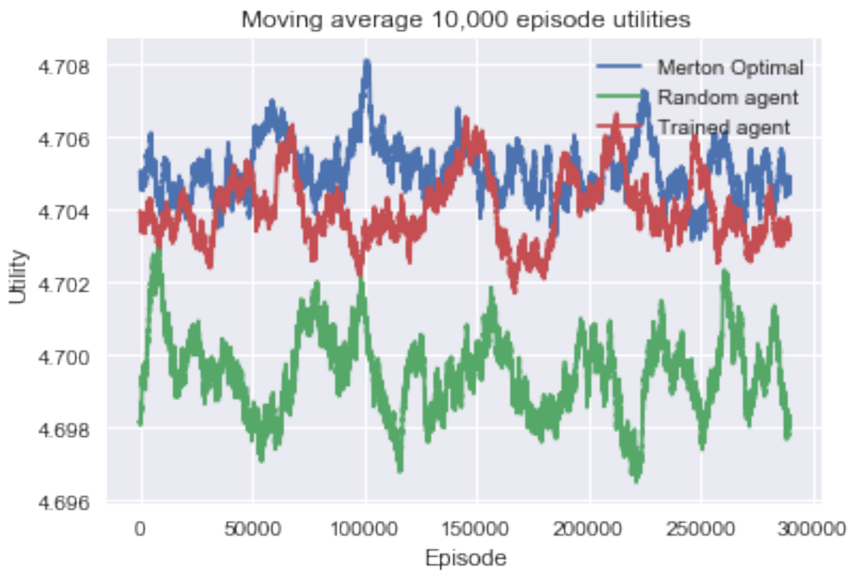


Figure 5.9: Moving average utilities Double-Q, MV-equivalent

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	4.6994	4.7040	4.7050
Rewards Distribution	10.11	10.46	10.63
Wealth Distribution	110.78	111.01	111.03
Sharpe Ratio Distribution	0.76	1.12	1.02

Table 5.5: Double-Q, MV equivalent, test performance - Mean

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	0.0040	0.0026	0.0030
Reward Distribution	0.4389	0.3070	0.3439
Wealth Distribution	0.4441	0.3147	0.3467
Sharpe Ratio Distribution	0.0586	0.0673	0.0665

Table 5.6: Double-Q, MV equivalent, test performance - Std

5.1.4 Power Utility

I introduced power utility in Chapter 2, and its merton optimal solution for a geometric brownian motion. The is more general than log-utility. The key intuition from the point of view of this thesis is that this utility could be one of any number of utility functions we can approximate through the mean-variance approach to merton's problem.

The equation and solution for power utility is given below:

$$U(x) = x^{\gamma}\,, 0 < \gamma < 1$$

$$u^{*} = \frac{\mu - r}{\sigma^{2}(1-\gamma)}$$

In experiment 5, the parameters of the experiment are the same as for my higher volatility environment. I set $\gamma = 0.1$ and approximate my rewards using mean-variance.

5.1.4.1 Power Utility conclusion

The results here are less conclusive and take some interpretation. First the agent definitely performed a lot better than a random agent and approached that of a merton optimal agent as regards utilities. However, the performances were noisy. In figure 5.11 we see the 10,000 episode moving average for utility performance. Even the merton optimal agent sometimes falls below a random agent, the trained RL in red appears to mostly match the performance of the Merton optimal agent but we can see that it is far more risk averse. The bottom two charts of figure 5.10 show that the agent has gained its higher utility performance at the expense of wealth by reducing risk (hence the high sharpe ratio). Clearly I could tune this if I wished, for example, train for more episodes as well as adjusting the risk aversion parameter κ to tune this, but for 3,000,000 episodes of training and the κ value used for other experiments, this is what the double-Q agent learned.

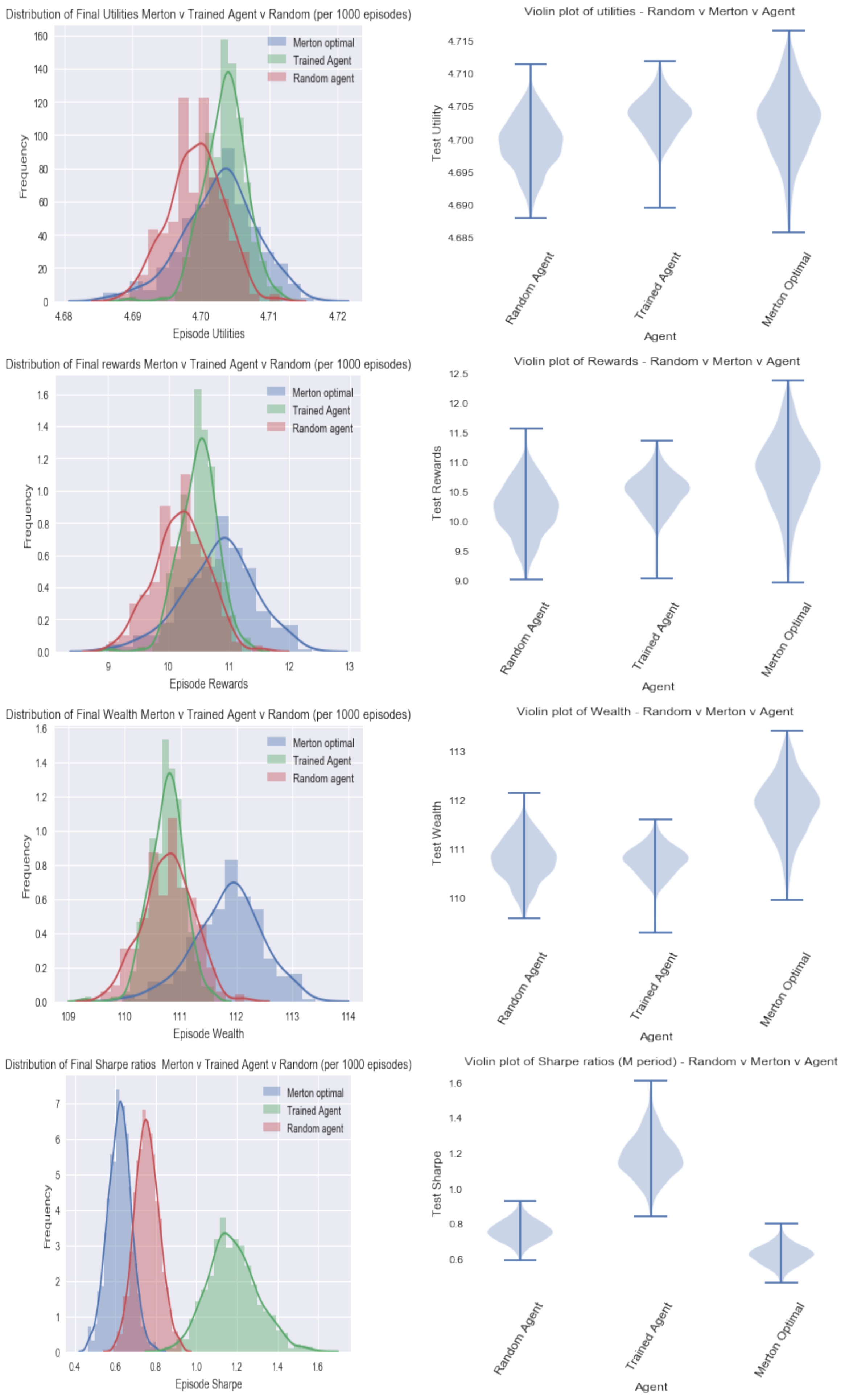


Figure 5.10: Double-Q learning: Utilities verses Merton, power utility

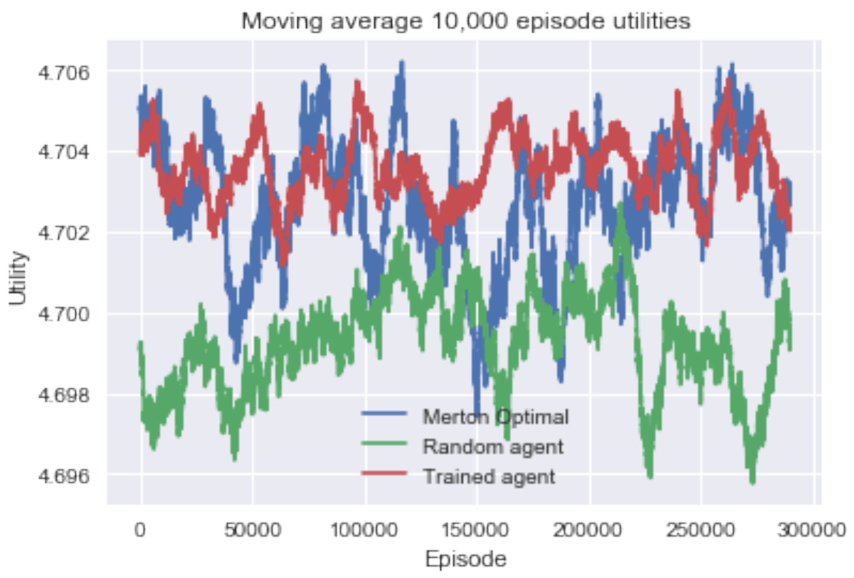


Figure 5.11: Moving average utilities Double-Q v Merton, power utility

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	4.6995	4.7038	4.7050
Rewards Distribution	10.22	10.52	10.69
Wealth Distribution	110.79	110.78	111.03
Sharpe Ratio Distribution	0.76	1.18	1.02

Table 5.7: Double-Q, test performance v Merton - Mean, power utility

	Random Agent	Double-Q	Merton Optimal
Utility Distribution	0.0118	0.0138	0.0108
Reward Distribution	0.0118	0.0138	0.0108
Wealth Distribution	1.3946	1.7390	1.4035
Sharpe Ratio Distribution	0.0493	0.0459	0.0501

Table 5.8: Double-Q, test performance v Merton - Std, power utility

5.1.5 Learning risk aversion κ

In this experiment I examine the effect of varying the risk aversion parameter κ, one would expect that by varying κ one can train the RL agent to exhibit different risk behaviours. I shall test the behaviour of the agent including two extremes - risk neutrality and extreme risk aversion. One would hope that in the case of risk neutrality that the agent becomes a pure wealth maximiser and indeed perhaps achieves a test distribution for wealth exceeding that of a merton optimal agent, albeit with a risk cost.

In the case of extreme risk aversion, one might expect that the agent shies away from either leverage or any investment in the risky asset, instead preferring to be 100% invested in the risk-free bond.

5.1.5.1 Learning risk aversion: Risk Neutrality

Figure 5.12 illustrates the utility gained by simulating various fixed investment ratios, in the case where $\kappa = 0$. As one might expect, the curve becomes linear and the maximum reward is gained by taking on as much leverage as possible and investing it all into the risky asset.

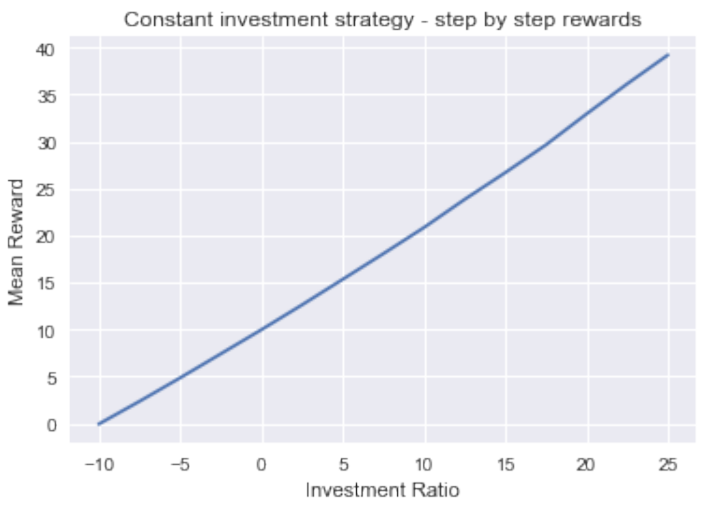


Figure 5.12: $\kappa = 0$, Risk neutrality

5.1.5.2 Risk Neutrality results

Figure 5.13 and tables 5.9, 5.10 illustrate the distributions of weath and sharpe ratios for the agent trained towards risk neutrality.

We can see that the trained double-Q agent indeed outperforms even the merton optimal agent as regards its wealth distribution on the test set, however this indeed comes at a considerable cost. The sharpe ratio is drastically reduced due to the higher variability of final wealth. Thus, the agent behaves exactly as intuition might suggest when trained towards risk neutrality.

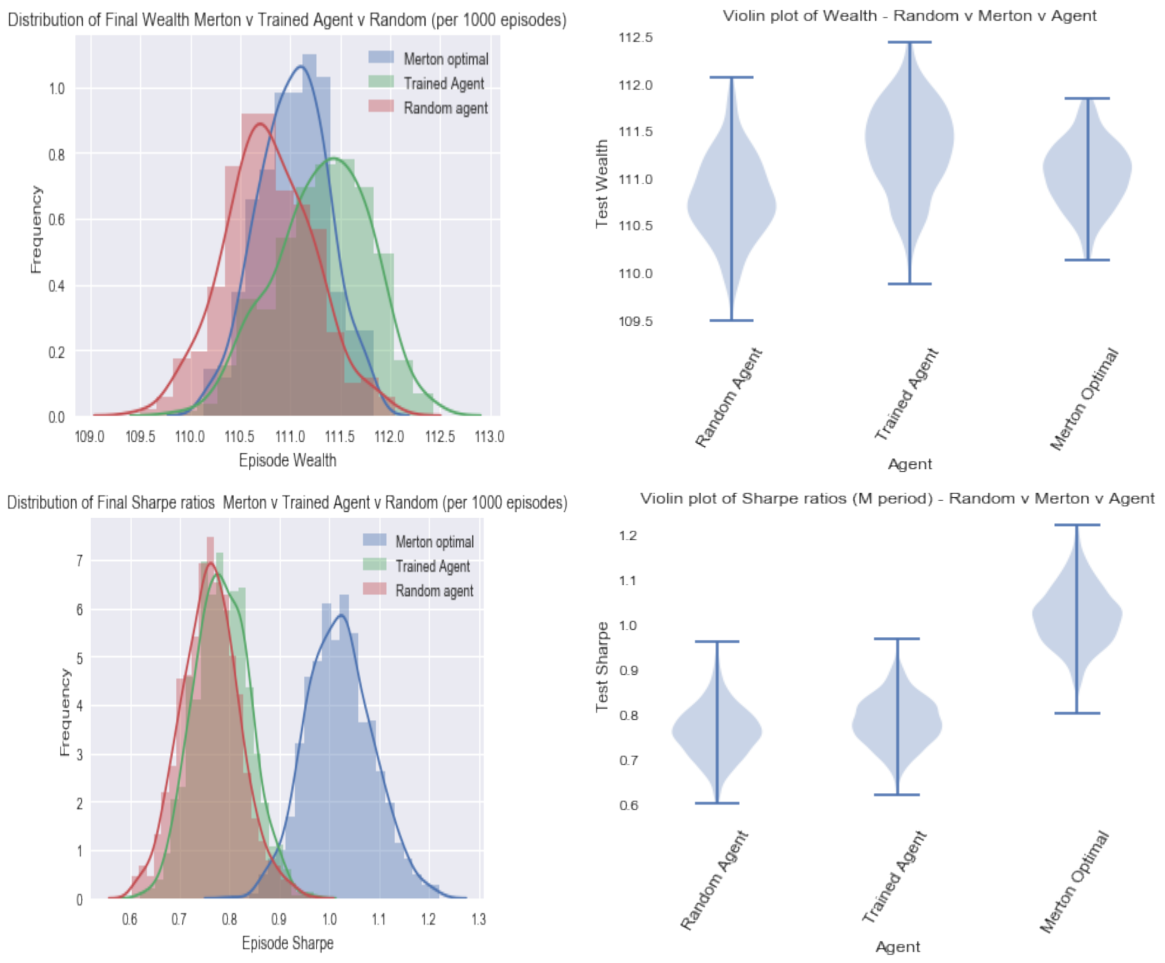


Figure 5.13: Double-Q learning: Wealth Distribution v Merton, Risk Neutral

	Random Agent	Double-Q	Merton Optimal
Wealth Distribution	110.78	111.32	111.03
Sharpe Ratio Distribution	0.7615	0.7842	1.0202

Table 5.9: Double-Q, Risk Neutral agent, test performance - Mean

	Random Agent	Double-Q	Merton Optimal
Wealth Distribution	0.4441	0.4683	0.3467
Sharpe Ratio Distribution	0.0586	0.0552	0.0665

Table 5.10: Double-Q, Risk Neutral agent, test performance - Std

5.1.5.3 Learning risk aversion: High Risk Aversion

I will now begin moving towards the other extreme, that is to increase κ in order to train the agent to first exhibit greater risk aversion than the utilities we have currently been using and then finally to train extreme risk aversion.

For a double-Q agent trained with a $\kappa = 0.013$, we see the test results in figure 5.14 and tables 5.11, 5.12. The trained agent now has a far lower final distribution of wealth than both the merton optimal agent and even a random agent. However the distribution of outcomes is also much narrower. This can be visualised easily when one examines the respective sharpe ratios. The risk averse agent has a much higher sharpe ratio than even the merton optimal (recall that my adjusted sharpe does not subtract the risk-free rate so we can see these effects more clearly).

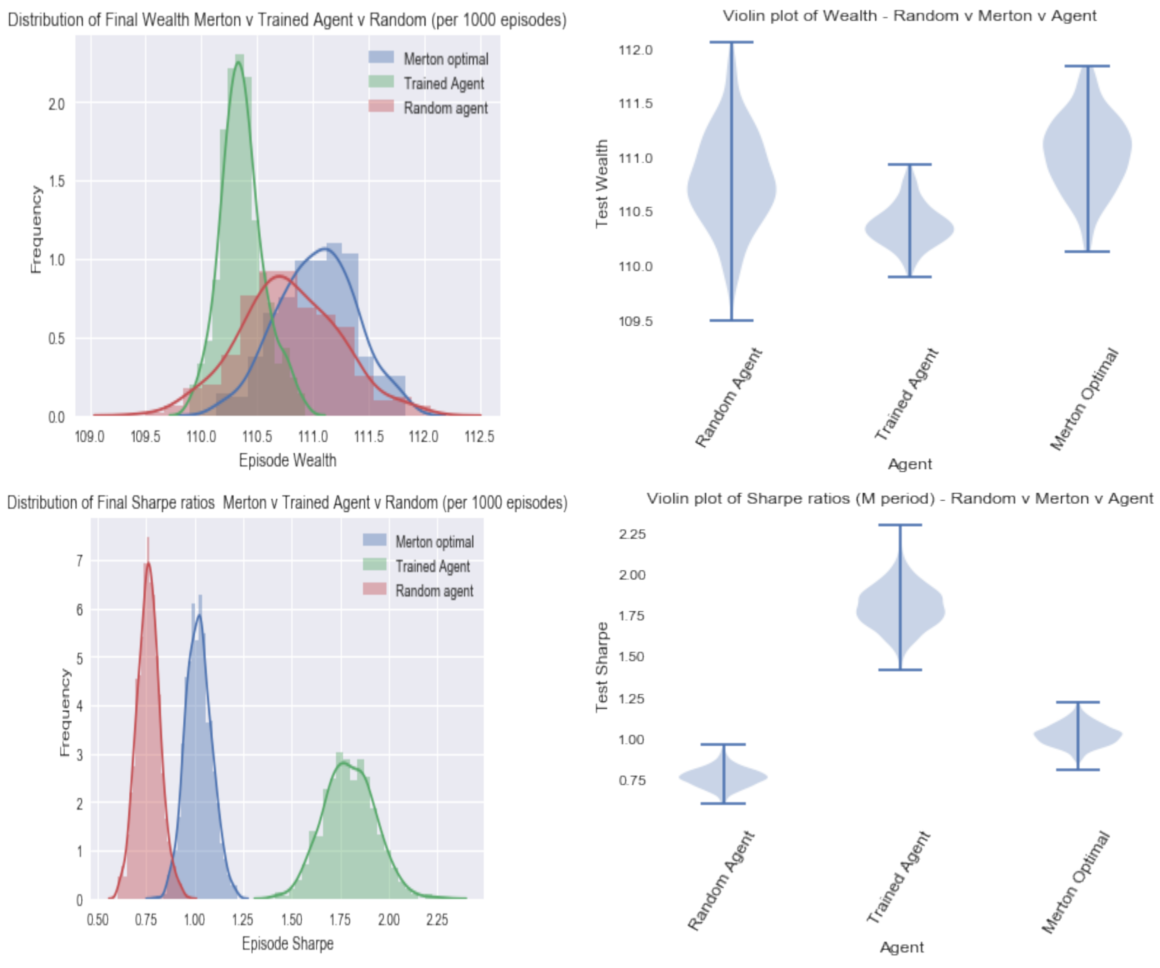


Figure 5.14: Double-Q learning: Wealth Distribution v Merton, Risk Aversion

	Random Agent	Double-Q	Merton Optimal
Wealth Distribution	110.78	110.36	111.03
Sharpe Ratio Distribution	0.7615	1.7959	1.0202

Table 5.11: Double-Q, Risk averse agent, test performance - Mean

	Random Agent	Double-Q	Merton Optimal
Wealth Distribution	0.4441	0.1923	0.3467
Sharpe Ratio Distribution	0.0586	0.1326	0.0665

Table 5.12: Double-Q, Risk averse agent, test performance - Std

5.1.5.4 Learning risk aversion: Extreme Risk Aversion

The story behind figure 5.15 hides a slightly painful anecdote. When I first implemented the code for step-wise mean-variance equivalence. I received the results in the right hand chart of figure 5.15. To my mind the flat red line showed that the double-Q agent had learned nothing. I interpreted this as a bug in my code and spent over a day testing and trying to find out where my error lay.

It was only when I simulated various constant investment strategies with $\kappa = 0.13$ and calculated the final utilities for each strategy (as shown in the left figure of 5.15) that I realised that at this level of κ the optimal utility would be gained by a 0% investment in the risky asset and 100% in the risk-free bond. A final check was to examine the greedy actions the reinforcement learning agent took (almost always action 4, which in this case corresponded to investing 100% in the risk free bond).

The answer of course was that I had calibrated my κ inadvertently to reflect extreme risk aversion and there was simply no distribution of wealth or utility, because there was no uncertainty! The scientific point however is that the RL agent successfully learned extreme risk aversion.

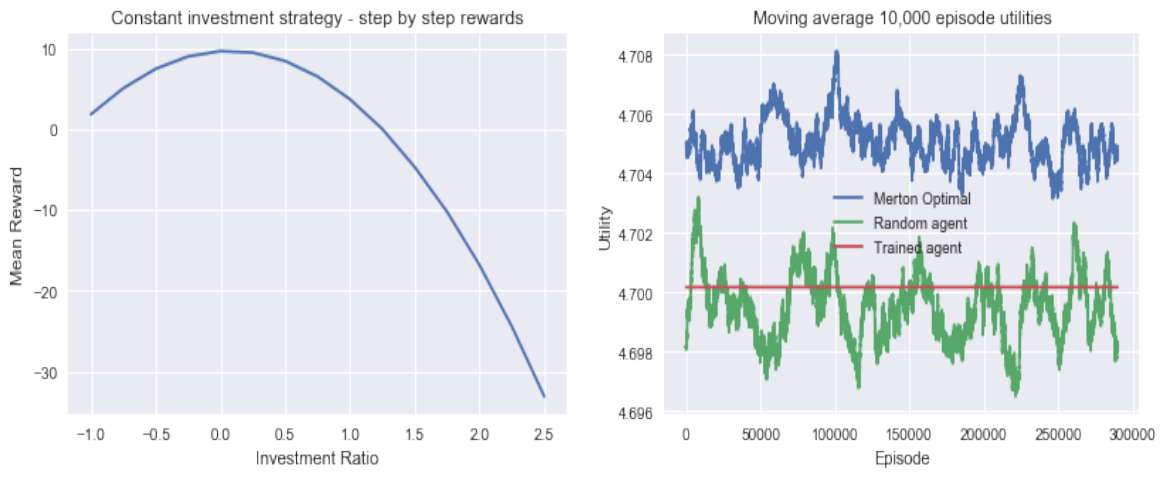


Figure 5.15: Double-Q learning: Utility curve for $\kappa = 0.13$, Extreme Risk Aversion

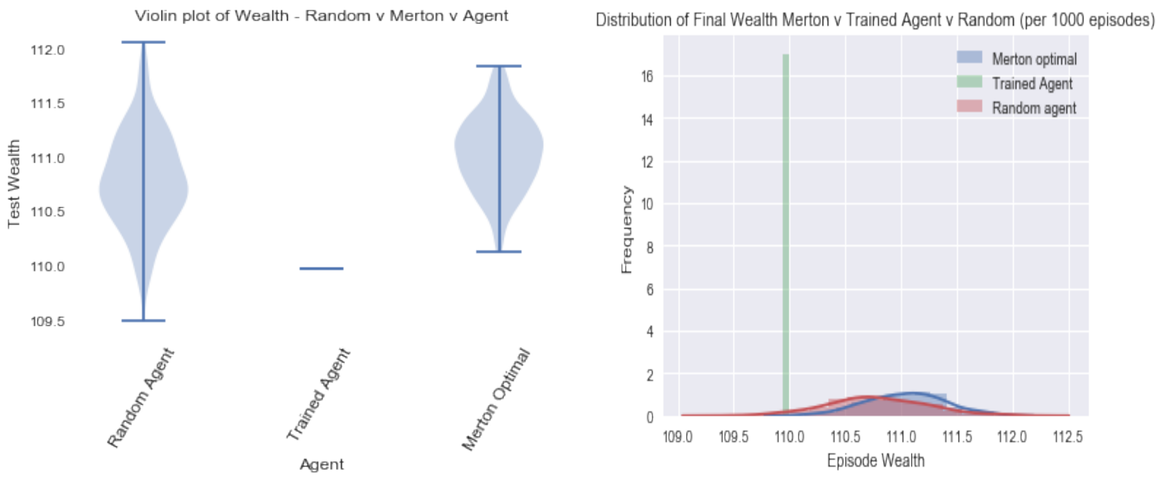


Figure 5.16: Double-Q learning: Wealth Distribution v Merton, Extreme Risk Aversion $\kappa = 0.13$

5.1.5.5 Learning risk aversion: Conclusion

This experiment showed the agent successfully learned a variety of behaviours, from extreme risk aversion to pure expected wealth maximisation. The results were successful and matched one's intuition about how one might expect the agent to behave in these circumstances.

5.2 Experiment group 2: Adding Realism

5.2.1 Realistic (S&P) volatility and risk free levels

In prior experiments we have shown that an RL agent (in this case a double-Q learner) can at least perform way better than a random agent and fairly close in performance to a merton optimal agent. This has so far been tested by using geometric brownian motion for our risky asset and low to mid-level volatility levels. Clearly as volatility increases then the reward signals for the agent can become more confusing. In this particular example I recallibrate the problem. The geometric brownian motion will now have levels of risk approximating that of the S&P 500, and the risk free rate will be far lower (approximating currently low long bond rates). Again the aim is to see if an RL can approximately match the performance of a merton optimal agent.

5.2.1.1 (S&P) volatility: Description

The stochastic process parameters are now changed. I used a drift $\mu = 0.1$, and a volatility $\sigma = 0.2$, the risk free rate $r = 0.02$. These numbers correspond more closely to a stochastic process akin to the long run returns of the S&P 500, with far higher volatility than the prior examples. The risk-free rate has been reduced, reflecting current low rates. A lack of realism remains in that we are still using a geometric brownian motion, but this does at least enable an analytical solution for the merton optimal ratio. Figure 5.17 illustrates some discretised sample paths in this new environment, because of the higher volatility we are likely to have a wider range of price and wealth states.

5.2.1.2 (S&P) volatility: Utilities and rewards

Again given the new environment I test the utilities and rewards with a variety of constant ratio investment strategies. Figure 5.18, illustrates the results. In this case the merton optimal investment would be approximately 2x leveraged, again this matches the simulated solution in the left chart of figure 5.18. In the right hand chart I have shown the same simulation using mean-variance approximated rewards. As a second order approximation there is no reason these curves should now be identical,

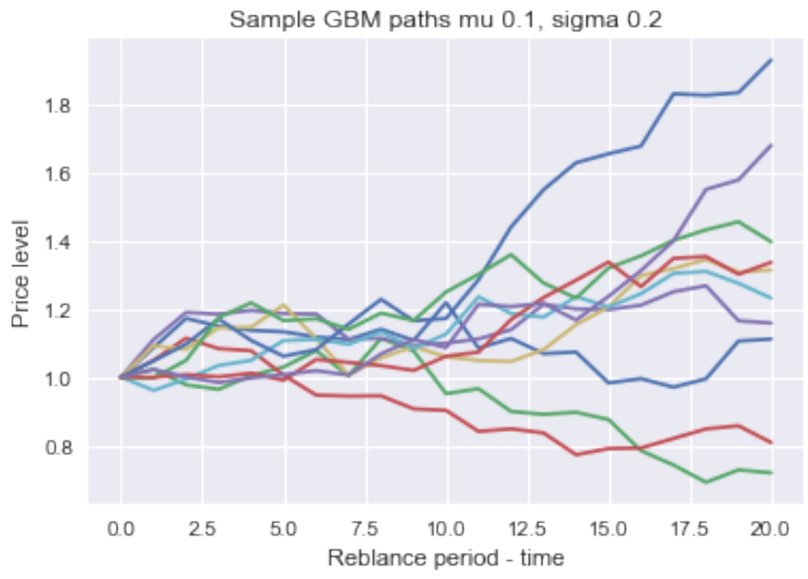


Figure 5.17: GBM sample paths, SP500 volatility levels

but κ should be such that the optimal solution for the step-wise rewards should be similar to the end of episode log-utilities. Here I have used $\kappa = 0.006$.

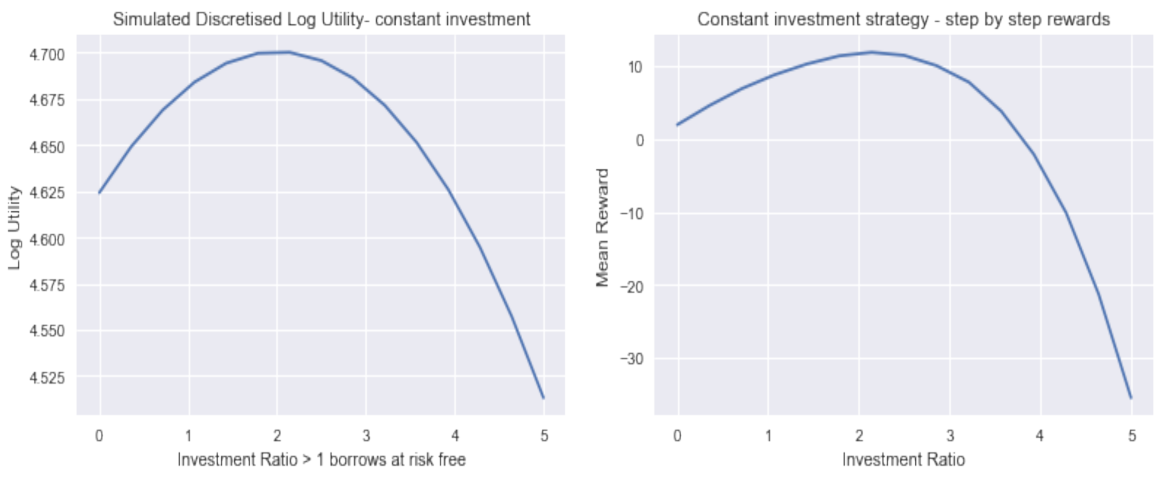


Figure 5.18: $\sigma = 0.2$, Log-utility, End episode utilities v Step-wise rewards $\kappa = 0.006$

Figure 5.19 shows the distribution of utilities and rewards for the merton optimal agent compared to a random agent. There remains an overlap in the two distributions due to the far higher level of volatility however this is somewhat mitigated by the increased difference in drift between the risky asset and the risk free. Notice that the rewards distribution as an approximation is less clean cut than for the final utilities as we are using the mean-variance approximation.

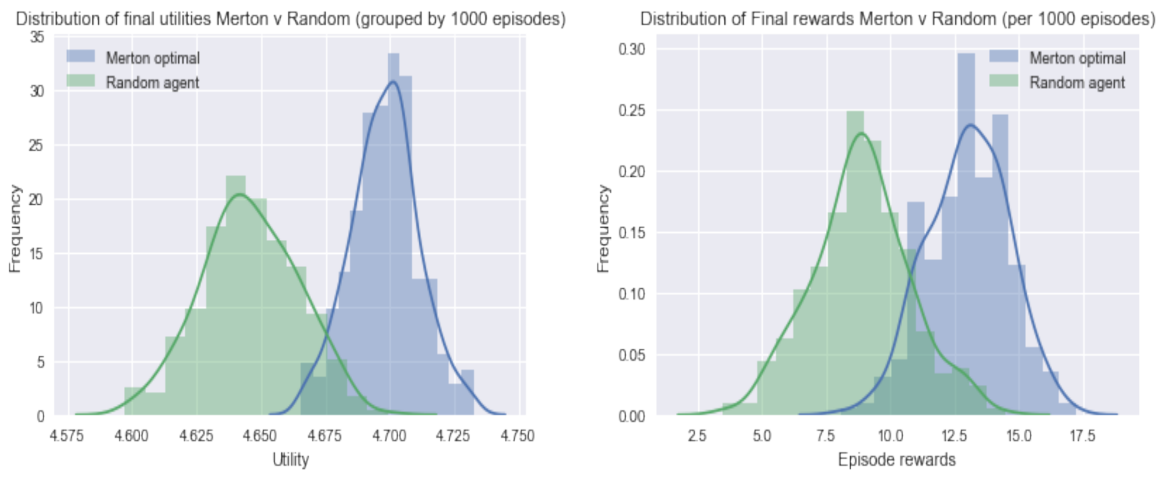


Figure 5.19: Batched Utilities and Reward distributions - Random v Merton Optimal

5.2.1.3 (S&P) volatility: Conclusion

I consider these results to be good. Recall that in this case the Q-learner knows nothing of the stochastic process or any parameters. However on the unseen test set the agent's distribution of utilities, rewards, wealth and sharpe ratios is similar in both size and variation to merton optimal. The Merton optimal agent remains a slightly stronger performer, but despite the higher volatility the Q-learner was able to learn to trade over multiple periods at a comparable level.

In case one is misled by the higher wealth performance of the random agent in the third chart of figure 5.20 one should realise that given the range of investment opportunities and leverage offered in this experiment, the random agent will on average take too much risk, the wealth levels are high (because of the higher drift in the risky asset), but at the cost of risk and thus rewards and utility.

Figure 5.21 shows the moving average of test utilities for the three agents, once again we see the double-Q learner shows a very close performance to the merton optimal agent and certainly way better than a random agent.

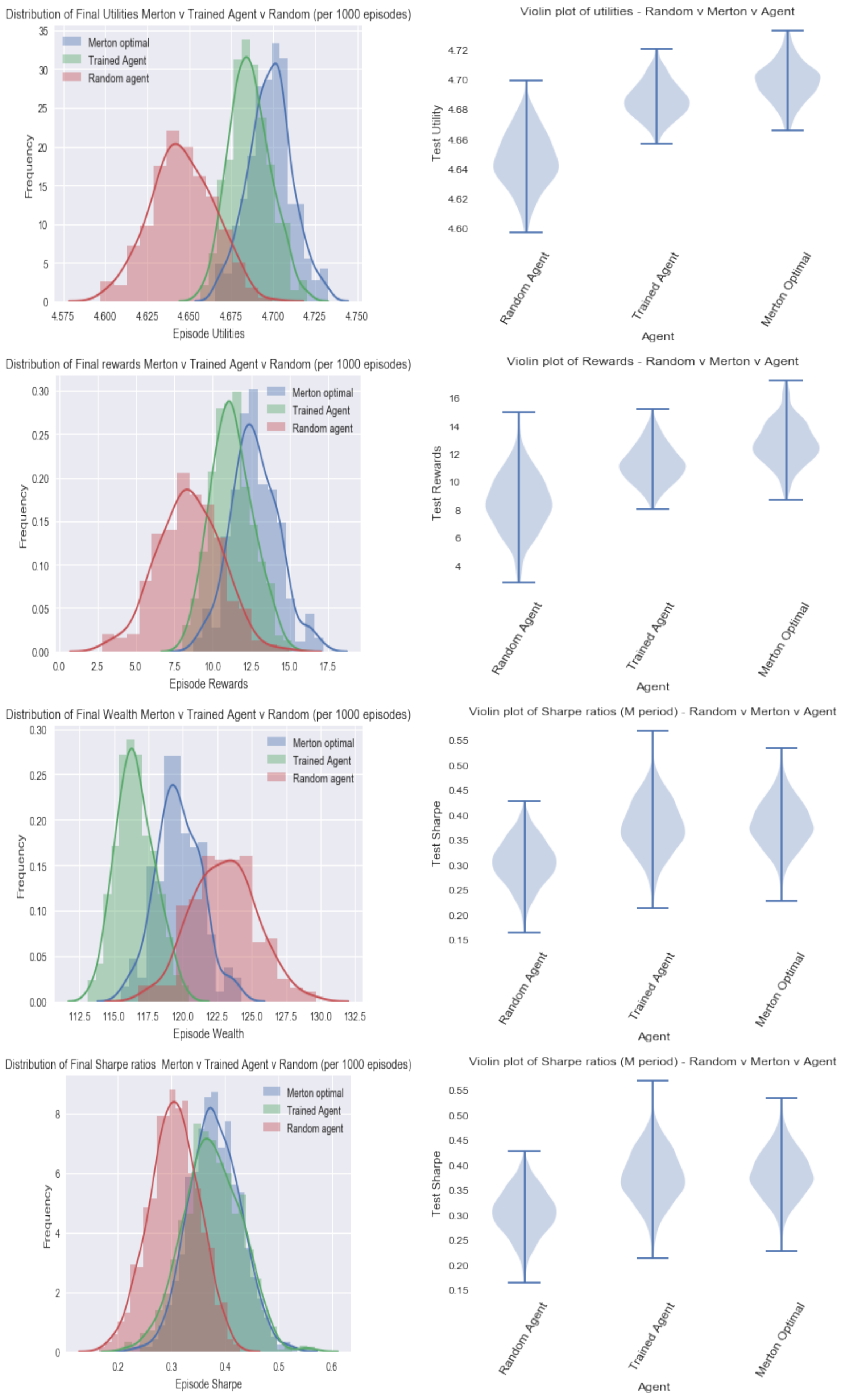


Figure 5.20: Q learning: Utilities verses Merton, $\sigma = 0.2$

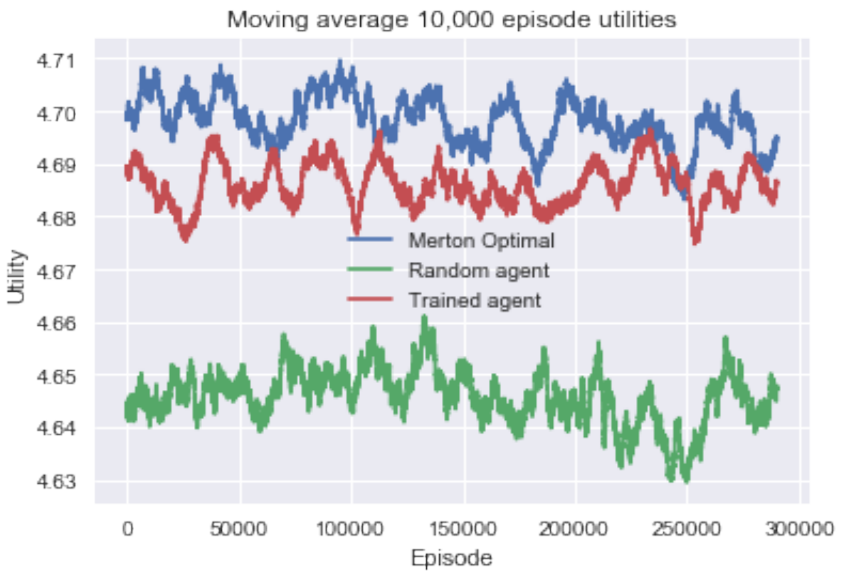


Figure 5.21: Moving average utilities Q v Merton, $\sigma = 0.2$

	Random Agent	Q	Merton Optimal
Utility Distribution	4.6457	4.6861	4.6981
Rewards Distribution	8.46	11.25	12.72
Wealth Distribution	122.97	116.54	119.68
Sharpe Ratio Distribution	0.30	0.37	0.38

Table 5.13: Q, test performance v Merton - Mean

	Random Agent	Q	Merton Optimal
Utility Distribution	0.0185	0.012	0.013
Reward Distribution	2.04	1.34	1.53
Wealth Distribution	2.34	1.39	1.64
Sharpe Ratio Distribution	0.04	0.05	0.04

Table 5.14: Q, test performance v Merton - Std

5.2.2 Fat tails

Real markets are not nearly as well behaved as geometric brownian motion. They have fat tails, jumps and are heteroscedastic. Thus far, the merton optimal agent has had a variety of advantages, the stochastic process is known and deliberately sourced to have an exact analytic solution, the parameters of this process also being known in advance. In real markets the underlying distribution is not known, and nor are the parameters stable. In these cases the merton agent becomes increasingly ill-specified, whereas a model free RL agent should be capable of learning from the data regardless of the underlying risky process.

5.2.2.1 Fat tails: Description

Here I make a mild shift towards including some real market effects. Instead of gaussian increments I add increments from a student-t distribution. Volatility $\sigma = 0.35$ is also very high. The effect of this is to create a less well behaved and slightly fat-tailed market environment. With 10 degrees of freedom for the student-t distribution the fat tails are still by no means extreme, but the idea behind the experiment is to test if the RL agent will begin to close the gap on the merton agent even with slightly fat tails. The underlying stochastic process is no longer a geometric brownian motion, and the merton agent is thus very slightly mis-specified for the first time.

Figure 5.22 illustrates some sample realisations of the risky process, notice the wider range of states the market can now reach. In order to accommodate these I slightly increased the range of state spaces for the RL agent to 150, I also increased the number of possible actions to 25 different investment choices at each time-step to make things more difficult. The right hand figure shows the distribution of end utilities between a merton optimal agent and a random agent. Clearly the random agent has a far broader range of utility outcomes.

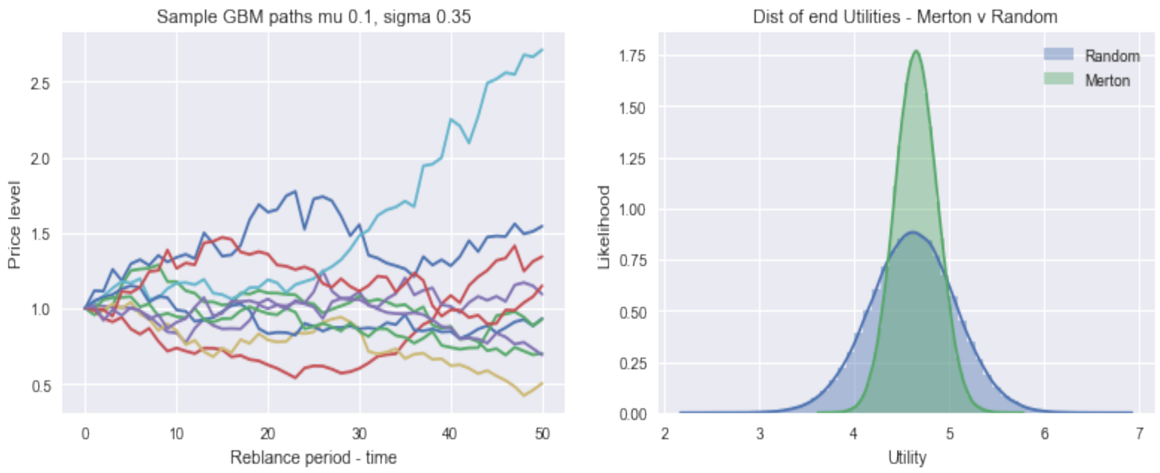


Figure 5.22: Sample fat tail paths/ Episodic distribution, Random agent v Merton

In figure 5.23 I re-examine our test statistics in this new environment. The charts show the difference in utility and reward distribution between the merton agent and a random agent with step-wise rewards and $\kappa = 0.006$.

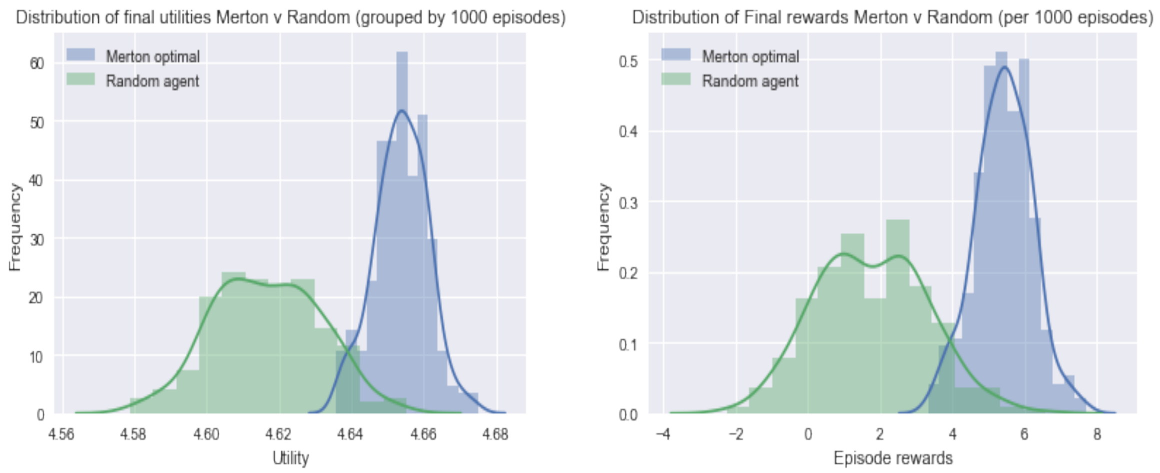


Figure 5.23: $\sigma = 0.35$ fat-tails, Log-utility, End episode utilities v Step-wise rewards $\kappa = 0.006$

5.2.2.2 Fat tails: Conclusion

Figures 5.24, 5.24 and tables 5.15, 5.16 illustrate the test results. Results are excellent. The introduction of very mild fat tails has resulted in the double-Q learner matching the performance of the merton agent for utilities, and resulted in a better performance as regards expected rewards and the expected sharpe ratio, also a favourable distribution of results in that it has achieved them with less variability than the merton optimal agent. This illustrates one of the potential advantages of a model free reinforcement learner...in the real world we do not know the exact underlying stochastic process, nor indeed its parameters, thus the merton model could be mis-specified. The RL agent will learn regardless. (Of course RL has other disadvantages such as needing a lot of data, however in practice one could compromise with semi-parametric approaches, where one sets a prior narrowing down the possible range of stochastic processes, then learns parameters from the real data and finally generates data for the RL agent to learn from - for example by using Gaussian [33] or t-processes [41]).

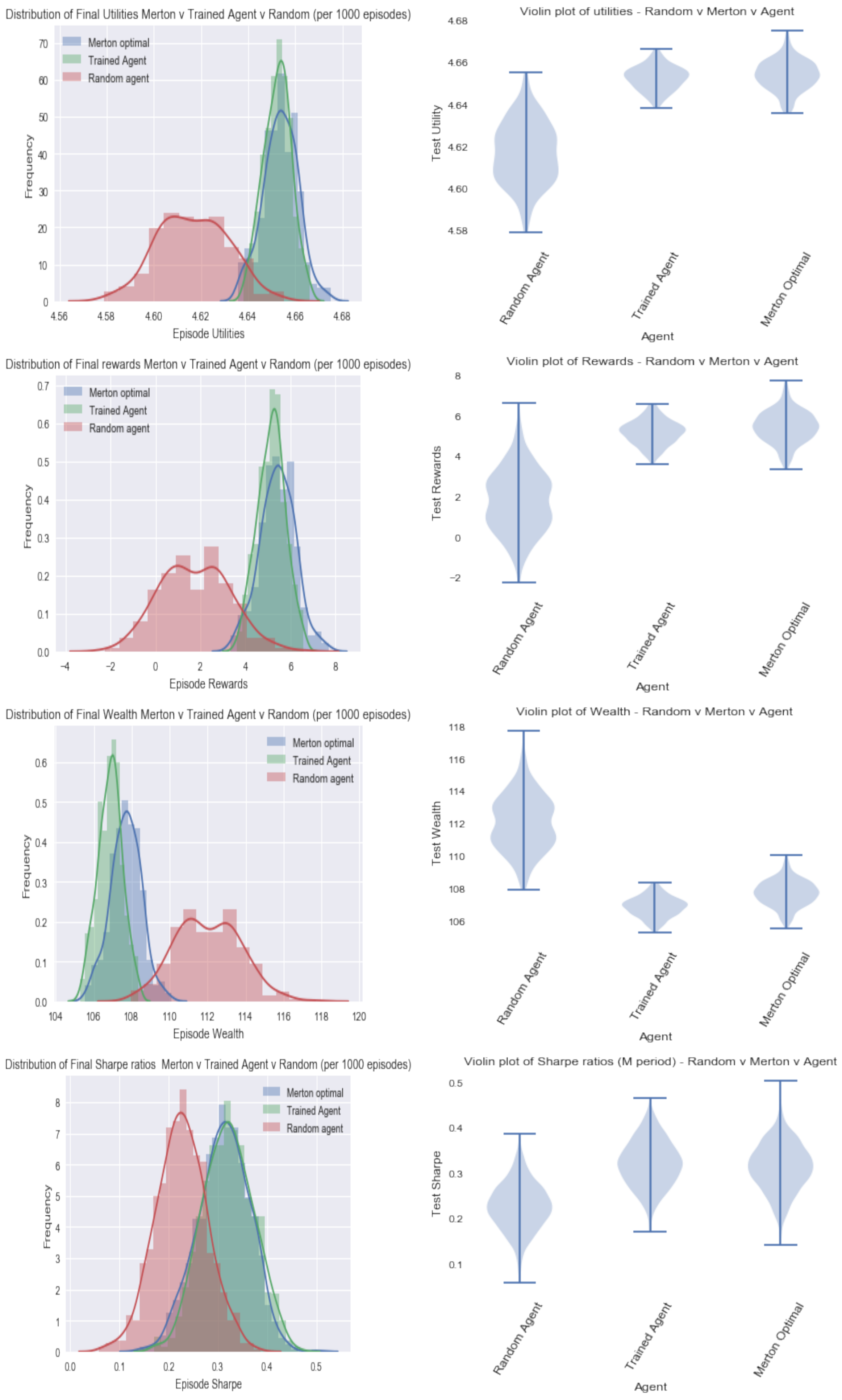


Figure 5.24: Q learning: Utilities verses Merton, $\sigma = 0.35$, fat tails

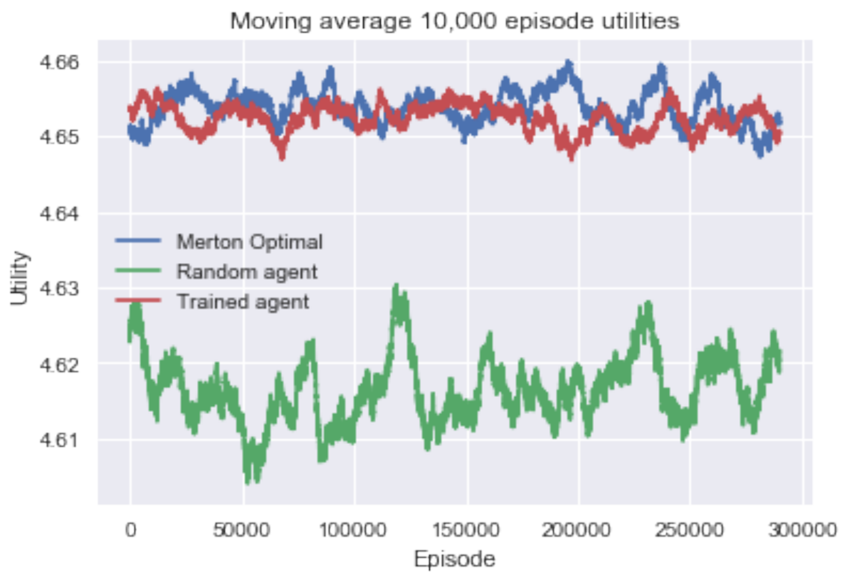


Figure 5.25: Moving average utilities Q v Merton, $\sigma = 0.35$, fat tails

	Random Agent	Q	Merton Optimal
Utility Distribution	4.6165	4.6523	4.6537
Rewards Distribution	1.7574	5.1524	5.4188
Wealth Distribution	112.02	106.87	107.69
Sharpe Ratio Distribution	0.2255	0.3201	0.3117

Table 5.15: Q, test performance v Merton - Mean, fat tails

	Random Agent	Q	Merton Optimal
Utility Distribution	0.01474	0.0058	0.0073
Reward Distribution	1.5255	0.6100	0.7847
Wealth Distribution	1.7025	0.6307	0.8041
Sharpe Ratio Distribution	0.0519	0.0515	0.0526

Table 5.16: Q, test performance v Merton - Std, fat tails

5.3 Experiment group 3: Adding power, alternative RL agents

5.3.1 Tabular agents Double Sarsa and Dyna

The experimental setting is exactly the same as before, that is a high volatility market environment with fat-tails.

5.3.1.1 Double Sarsa

Figures 5.26 and 5.26 show the distributions of utilities, rewards and the moving average of utilities for a random agent, compared to a merton optimal, the RL agent is a Double-Sarsa agent with an ε-greedy exploration parameter set to 0.1. Like Q-learning, Sarsa can also exhibit bias, thus my use of a Double-Sarsa agent. The key difference here is that Sarsa is an on-policy learning algorithm, that is the q-values reflect the expected reward from executing an ε-greedy policy, not a greedy one. Of course for test purposes the agent acts greedily with respect to the trained q-values.

Although the Double-Sarsa agent also performed well and was again far superior to a random agent, in general I found that the performance was weaker than the earlier double-Q agents. This is perhaps unsurprising and the exploration strategy could be improved, in particular one could begin with a high exploration rate and gradually anneal the rate of exploration with experience (this I did later when training the DeepQ agent).

It is certainly not always the case that Q-learning will always perform better than Sarsa with the classic example being the windy gridworld example given by Sutton and Barto [49]. The Q-learner chooses an optimal path close to the edge of the cliff, with no margin for error, but where the environment being windy can cause sharp losses to the agent's rewards. In contrast the Sarsa agent takes a safer wider path around the cliff edge (due to it learning from random exploration in training the dangers of being next to the edge of the cliff).

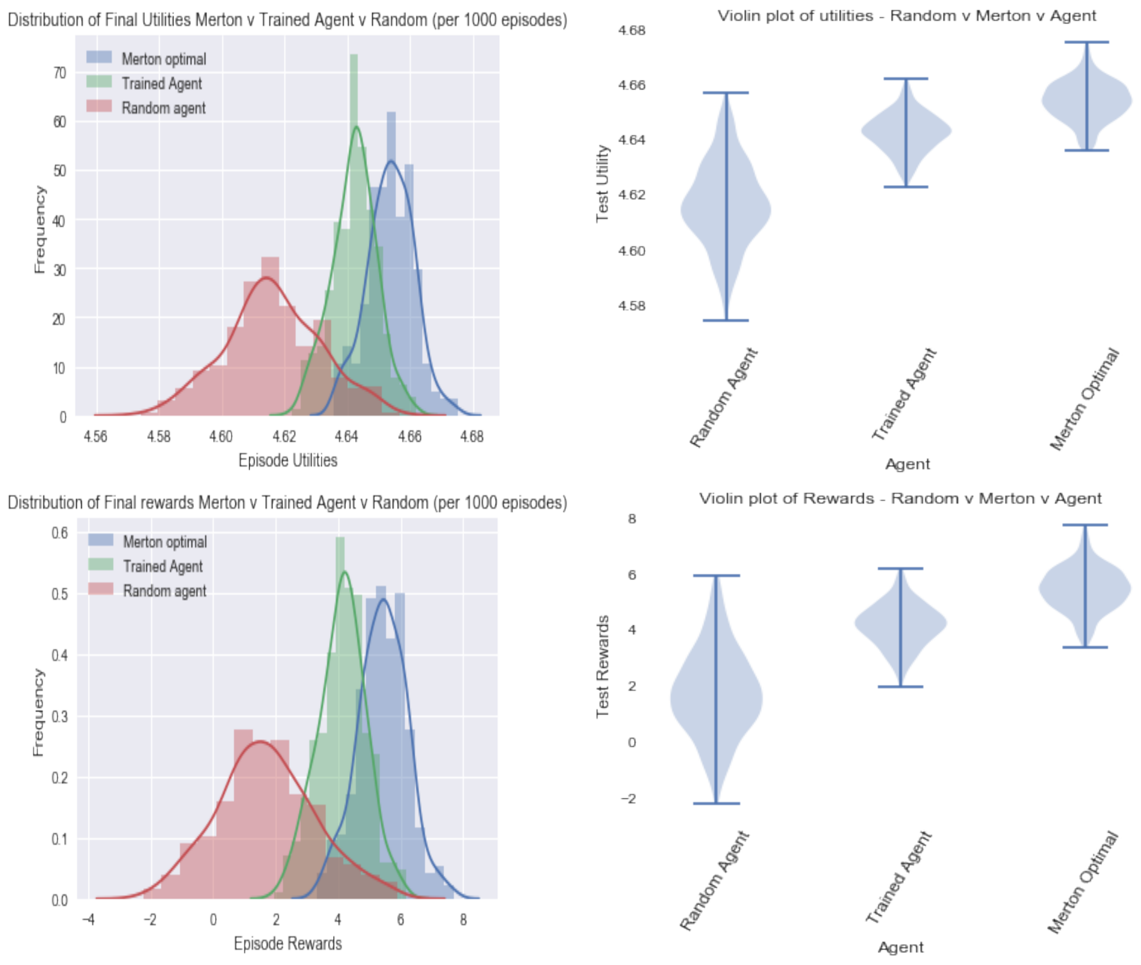


Figure 5.26: Double Sarsa learning: Utilities verses Merton, $\sigma = 0.35$, fat tails

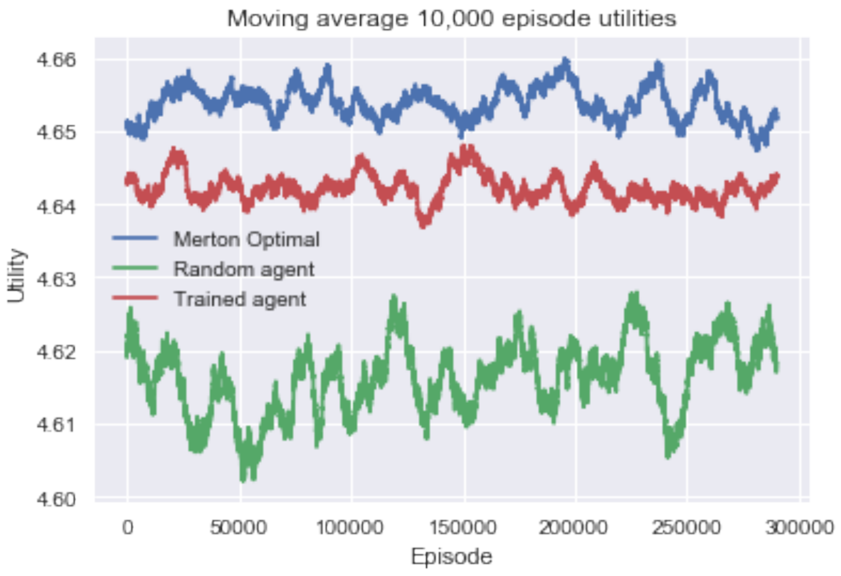


Figure 5.27: Moving average utilities Double Sarsa v Merton, $\sigma = 0.35$, fat tails

5.3.1.2 Dyna

The Dyna implementation used a very small amount of experience replay (I tried replays between 1 and 5, due to already executing 3 million episodes in training), as well as a tabular model (for Dyna to predict replays). A tabular setting is not entirely suitable for this environment due to its stochasticity, we will see more powerful models in the next section. The results in figures 5.28, 5.29 and tables 5.17, 5.18 demonstrate that Dyna was capable of learning within this environment again at close to optimal levels. Again I believe improvements can be gained by providing the Dyna models with a more sophisticated model. However results are still reasonable and despite the simplicity the experiment demonstrates that Dyna also learned to trade at performance levels close to the merton optimal agent.

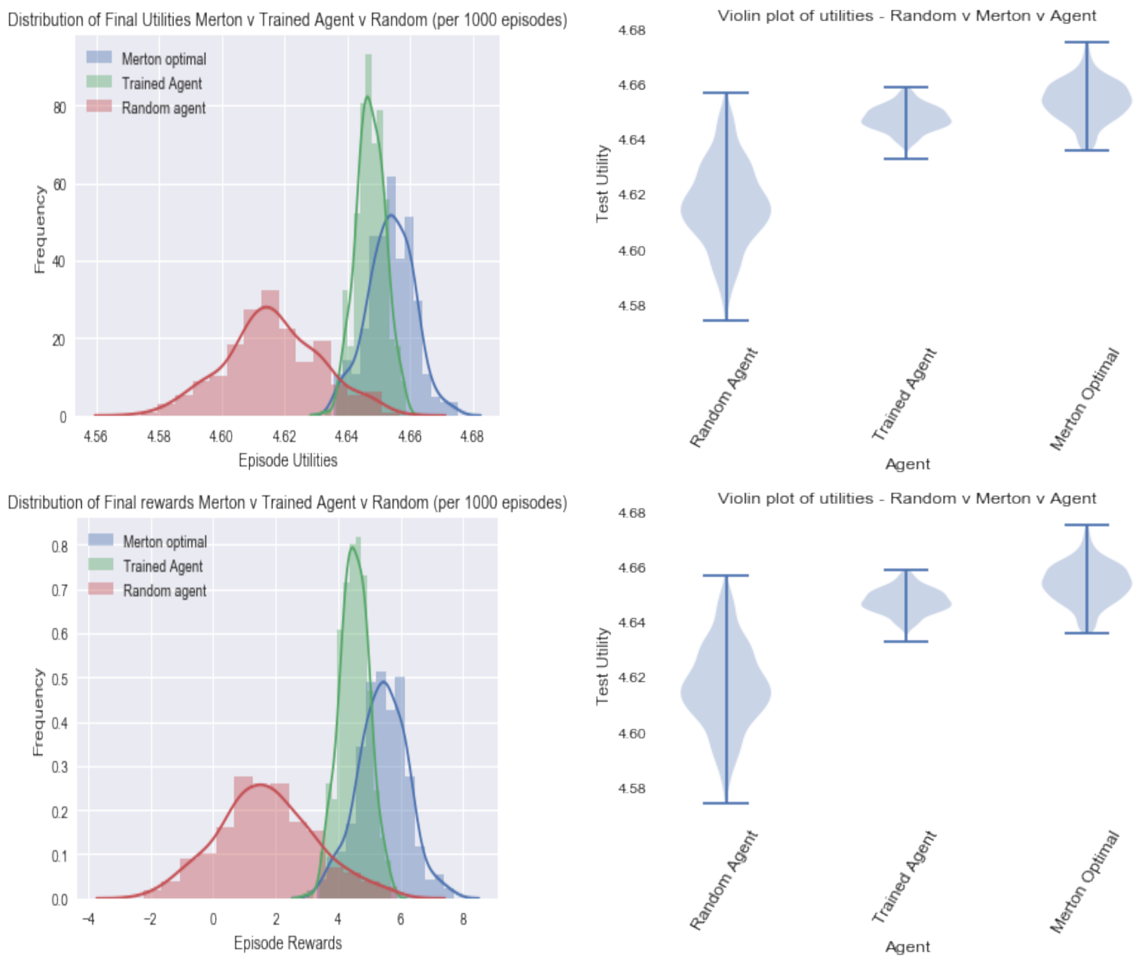


Figure 5.28: Dyna with experience replay: Utilities verses Merton, $\sigma = 0.35$, fat tails

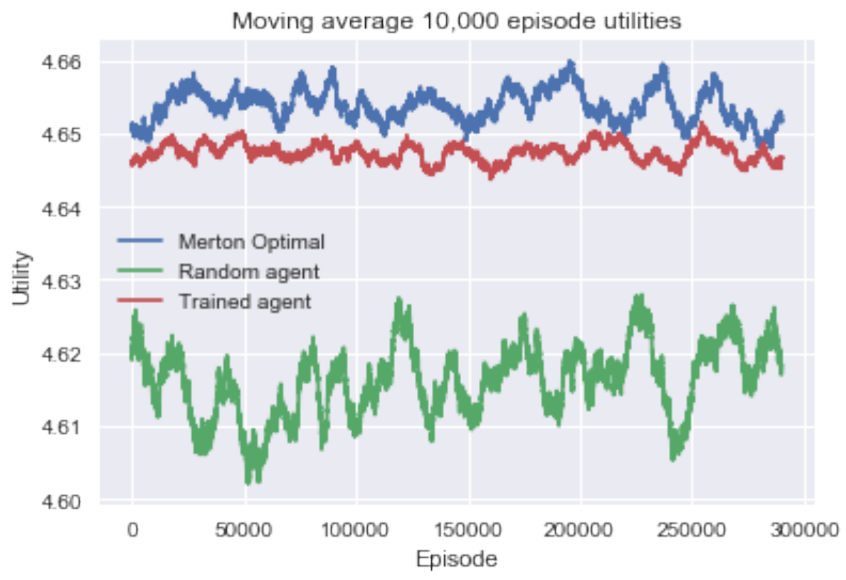


Figure 5.29: Moving average utilities Dyna with experience replay v Merton, $\sigma = 0.35$, fat tails

	Random Agent	Dyna	Merton Optimal
Utility Distribution	4.6162	4.6363	4.6537
Rewards Distribution	1.7101	3.9179	5.4188
Wealth Distribution	111.96	106.51	107.69
Sharpe Ratio Distribution	0.2246	0.2466	0.3117

Table 5.17: Dyna-experience replay, test performance v Merton - Mean, fat tails

	Random Agent	Dyna	Merton Optimal
Utility Distribution	0.0150	0.0083	0.0073
Reward Distribution	1.5482	0.8581	0.7847
Wealth Distribution	1.7195	0.8663	0.8041
Sharpe Ratio Distribution	0.0522	0.0601	0.0526

Table 5.18: Dyna-experience replay, test performance v Merton - Std, fat tails

5.3.2 Deep-Q learning

From this point the experimental setting changes. I no longer use tabular models, nor do I simplify the state space by bucketting. Inputs are raw prices and wealth levels, although inputs can now easily be extended to add many more features. The agent has 25 actions at each time step and for the environmental setting I use a volatility $\sigma = 0.2$. The risky asset is a geometric brownian motion and I use log-utility. The merton optimal agent is again the 'perfect' trader, where the analytic solution would be to lever up 2x and invest all of this in the risky asset at every time-step.

Sample path realisations for the risky asset and simulated utilities for various fixed ratio investment strategies are given in figure 5.30. In order to manage the increased dimensionalty I will no longer be representing each state-action tuple by an entry in a q-table. Instead I shall use function approximation, specifically a neural network will receive raw inputs and attempt to predict the q-value.

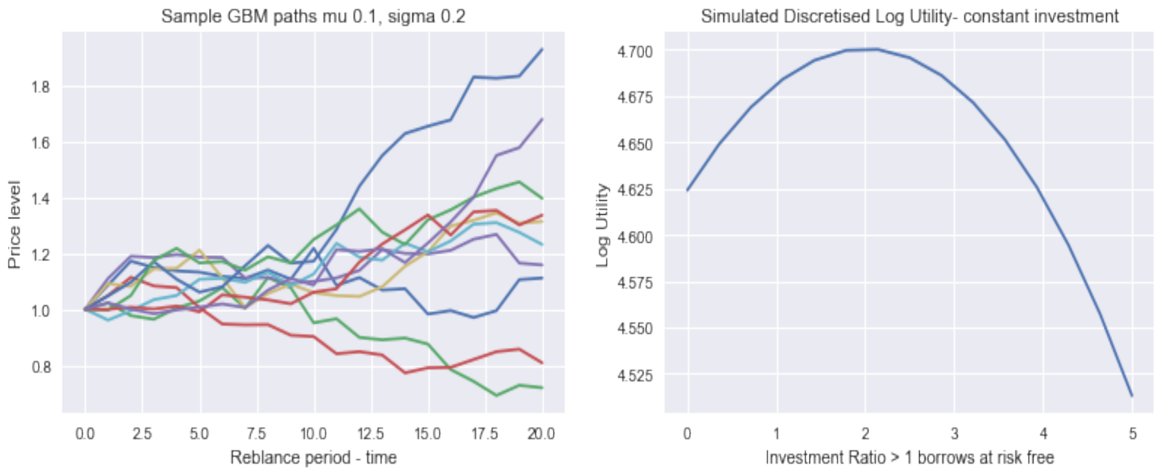


Figure 5.30: Sample paths and log-utilities - constant investment, $\sigma = 0.20$

5.3.2.1 Deep-Q learning: Description

For all of the deep reinforcement learning implementations I used the PyTorch library (https://pytorch.org/), the difference compared to Tensorflow is that it uses dynamic graphs rather than pre-specifying a static graph and compiling, both have their advantages. I also used code from the excellent advanced tutorial, https://github.com/higgsfield/RL-Adventure. DeepQ and Double DeepQ are easy enough to implement from scratch but this excellent website contains a number of clean implementations of reinforcement learning papers for the advanced student. Finally from an implementation point of view I also found it easier to modify my environment to be capable of inheriting from the OpenAI gym, this is a test bed of classic reinforcement learning environments, and again meant that I didn't have to reinvent the wheel and write everything from scratch.

The structure of the DeepQ network is given in table 5.19, a fairly small fully connected neural network using Relu activations. There are just under 19,000 parameters to train. Unlike tabular methods although experience is gained in an online fashion, training is now performed in batches (I used a batch size of 32), gathered through experience replay (as mentioned in my chapter on machine learning, this enables us to both break correlations between online states and also to aid sample efficiency in training). Finally the method of exploration was altered. ε-greedy was still used, but this time the initial exploration rate was 1.0, decaying to a final exploration rate equal to 0.01. The loss used was mean-squared error, together with the Adam optimiser.

Layer(type)	Output shape	Parameters
Linear-1	[-1, 128]	384
ReLU-2	[-1, 128]	0
Linear-3	[-1, 128]	16,512
ReLU-4	[-1, 128]	0
Linear-5	[-1, 15]	1,935
Input dimension:2		Total params: 18,831
(risky asset price and wealth)		Trainable params: 18,831
		Non-trainable params: 0

Table 5.19: Deep Q model specification

Figure 5.31 shows the neural network in training, the left chart shows the final rewards. As mentioned many times in this thesis, they are very noisy. The right chart shows the training loss. To reduce the noise in this case I have actually shown the moving average of the loss. The purpose of the experiment was to learn to trade, so I did not explore lots of different parameter settings, for example to the naked eye it appears the network reduced the vast majority of its training loss in far fewer episodes than used in this experiment, so I can probably be more sample efficient than I have been.

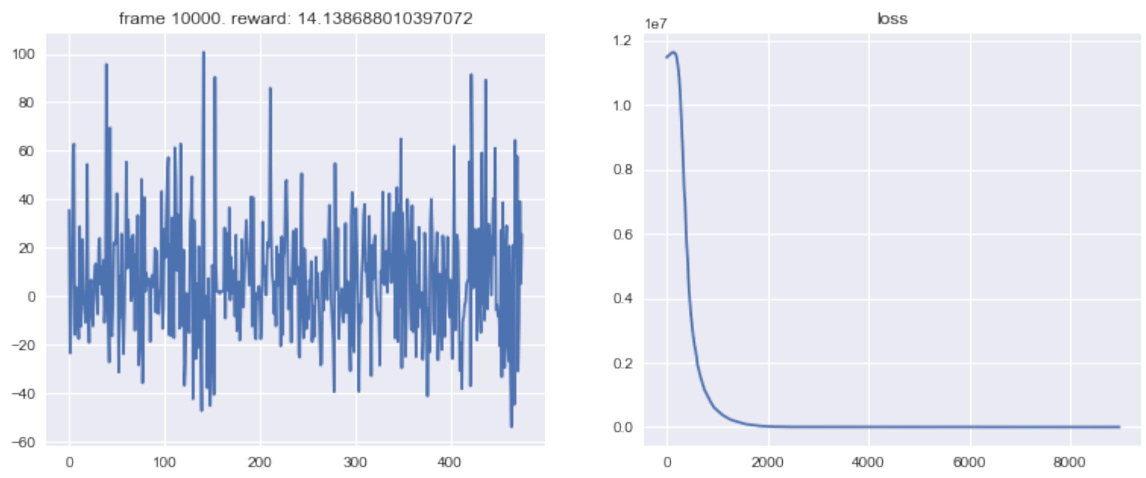


Figure 5.31: Final training rewards and loss function training DQN

5.3.2.2 Deep-Q learning: Conclusion

Figures 5.33, 5.34 and tables 5.20, 5.21 show the results. These results appear to be perfect, the agent has completely matched the distribution of the merton optimal agent for utilities, rewards, wealth and sharpe ratio. Figure 5.34 shows the moving average of utilities.

In order to test if the agent was close to merton optimal or actually acting as a merton optimal agent I performed a further test. I ran a new test with several hundred thousand episodes and created a histogram of the agents actions. Recall that a merton optimal agent will choose to invest 2x its wealth in the risky asset at every time step. The DeepQ agent has 25 different choices at each step. Figure 5.32, illustrates the histogram of actions chosen by the DeepQ agent, it always chose to lever up 2x and invest it all in the risky asset, exactly matching the the merton optimal agent and the peak of the curve in the left hand chart.

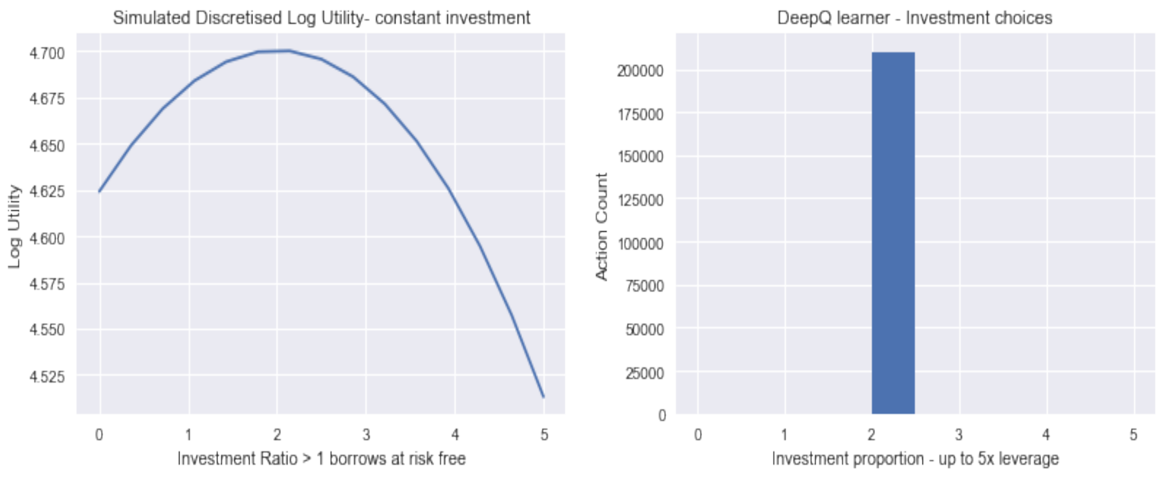


Figure 5.32: DQN investment choices for a variety of states

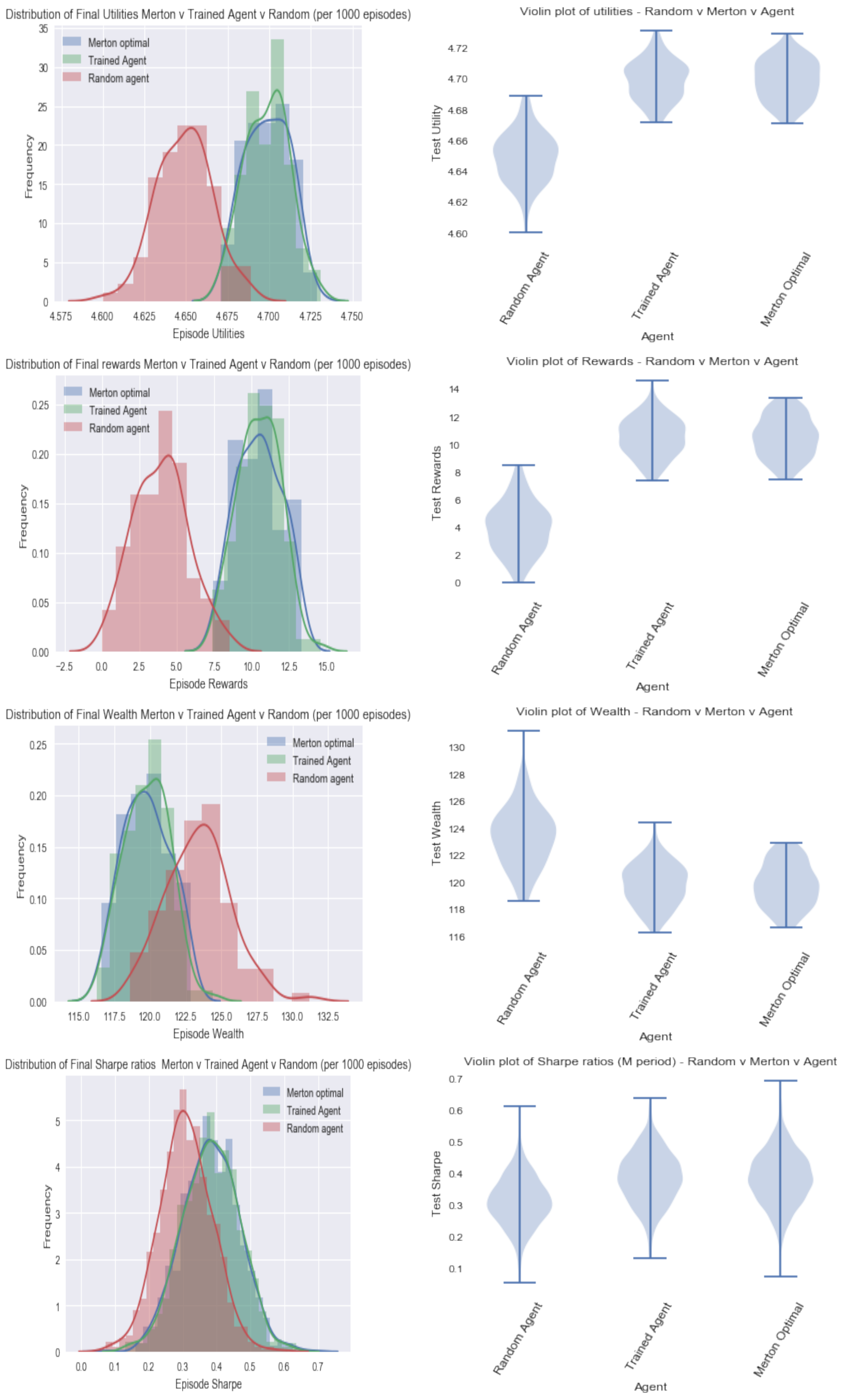


Figure 5.33: Deep Q learning: Utilities verses Merton, $\sigma = 0.20$

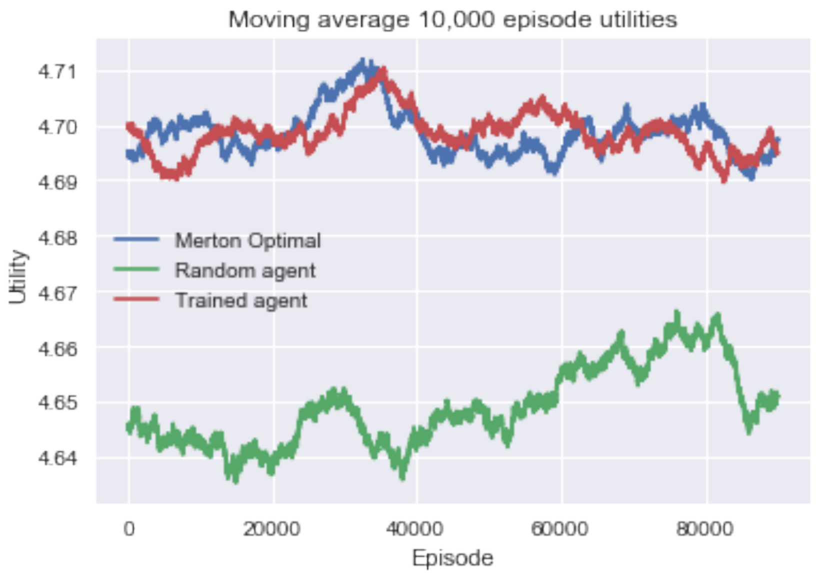


Figure 5.34: Moving average utilities Deep Q v Merton, $\sigma = 0.20$

	Random Agent	Deep Q	Merton Optimal
Utility Distribution	4.6489	4.6987	4.6987
Rewards Distribution	3.9576	10.4992	10.4936
Wealth Distribution	123.36	119.77	119.74
Sharpe Ratio Distribution	0.3098	0.3831	0.3837

Table 5.20: Deep Q, test performance v Merton - Mean

	Random Agent	DeepQ	Merton Optimal
Utility Distribution	0.0163	0.0130	0.0132
Reward Distribution	1.8240	1.4019	1.4837
Wealth Distribution	2.2410	1.5692	1.6197
Sharpe Ratio Distribution	0.0781	0.0833	0.0838

Table 5.21: DeepQ, test performance v Merton

5.3.3 Double Deep-Q learning

5.3.3.1 Double Deep-Q learning: Description

The double deep-Q learning implementation again used my OpenAI compatible market environment and borrowed heavily from https://github.com/higgsfield/RL-Adventure in terms of model implementation, double deep-Q [19] was described earlier in the thesis in the machine learning chapter. The only difference in this experiment (bar being a minor extension to the previous one by using double learning) was that I reduced the number of training episodes to 500,000 (recall that for tabular Q-learning with a much simpler state space I was using 3,000,000 episodes).

5.3.3.2 Double Deep-Q learning: Conclusion

Figures 5.36, 5.35, and tables 5.22, 5.23 again show the results. Perhaps it is unexpected given that the DeepQ learner appeared to give perfect test results that also in the case of double DeepQ learning the results appear to be perfect. The agent appears to match merton optimal behaviour exactly.

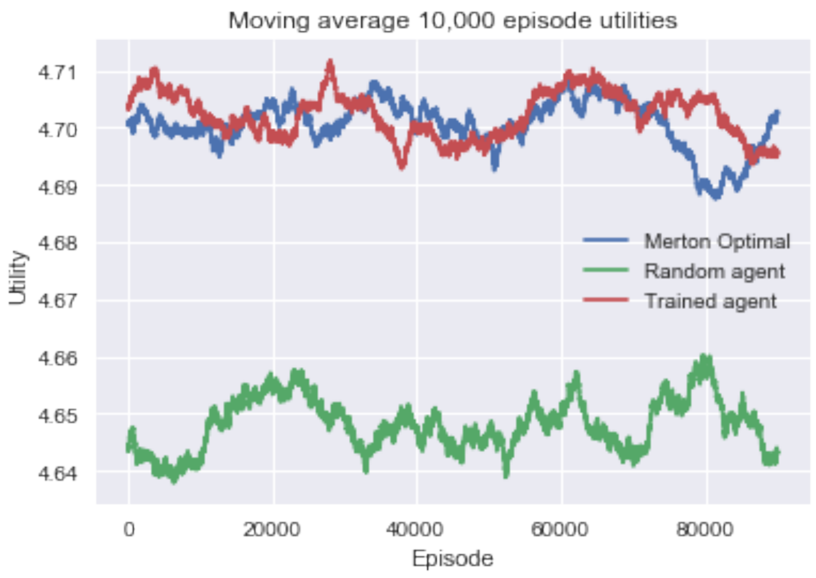


Figure 5.35: Moving average utilities Deep Q v Merton, $\sigma = 0.20$

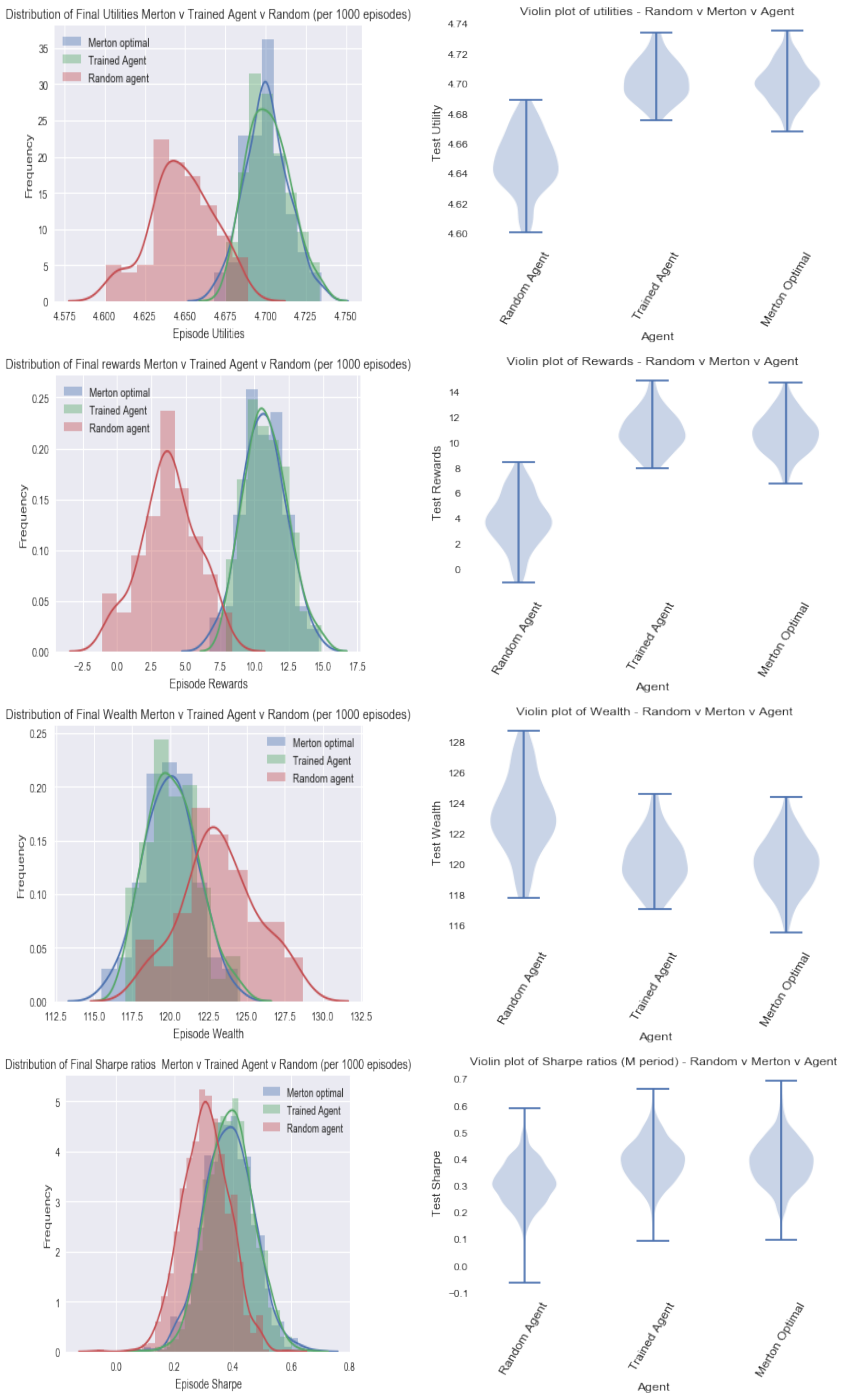


Figure 5.36: Double Deep Q learning: Utilities verses Merton, $\sigma = 0.20$

	Random Agent	Double Deep Q	Merton Optimal
Utility Distribution	4.6489	4.6987	4.6987
Rewards Distribution	3.9576	10.4992	10.4936
Wealth Distribution	123.36	119.77	119.74
Sharpe Ratio Distribution	0.3098	0.3831	0.3837

Table 5.22: Double Deep Q, test performance v Merton - Mean

	Random Agent	Double DeepQ	Merton Optimal
Utility Distribution	0.0163	0.0130	0.0132
Reward Distribution	1.8240	1.4019	1.4837
Wealth Distribution	2.2410	1.5692	1.6197
Sharpe Ratio Distribution	0.0781	0.0833	0.0838

Table 5.23: DeepQ, test performance v Merton

5.3.4 Noisy Net Deep-Q Networks

5.3.4.1 Noisy Net Deep-Q Networks: Description

The Noisy Net implementation again used my OpenAI compatible market environment and the model is built with code from https://github.com/higgsfield/RL-Adventure. This experiment examined a completely different method of exploration to ε-greedy. It is based on a recent paper [13], and uses both prioritised experience replay and 'noisy linear' layers for exploration. The high level idea is that noise is added to the neural net layers, but that this noise is itself parameterised, with parameters that can be learned via gradient descent.

As mentioned earlier in the case of ε-greedy, decorrelated and state independent noise is added at each step. However the authors found that a single change to the weight vector of a neural network can introduce complex, state dependent changes to the policy over multiple time steps. They are effectively creating a randomised value function from a neural network, which has previously been shown by Osband et al. [32] to be an efficient means of exploration. The number of training episodes was in this case reduced to 100,000. The training rewards and moving average of the training loss can be seen in figure 5.37.

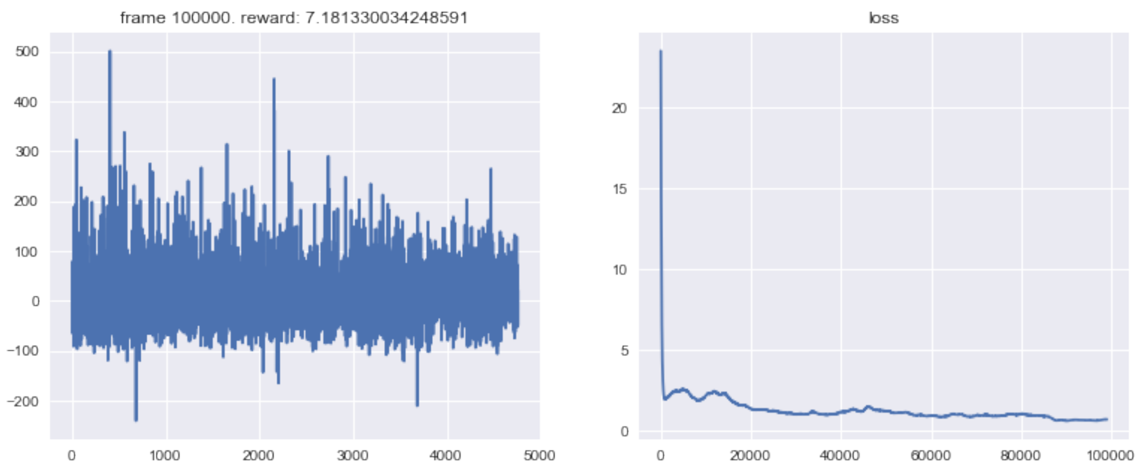


Figure 5.37: Final training rewards and loss function training DQN

5.3.4.2 Noisy Net Deep-Q Networks: Conclusion

Results can be seen in figures 5.39, 5.38, and tables 5.24, 5.25. Again the test results were successful with the noisy net agent learning to trade at merton optimal levels. In this case however training was achieved using an alternative method of exploration and for this one example at least did indeed appear to be faster. I would not draw a general conclusion based upon this single result, but clearly sample efficient learning is an important need for reinforcement learning (in particular for noisy environments).

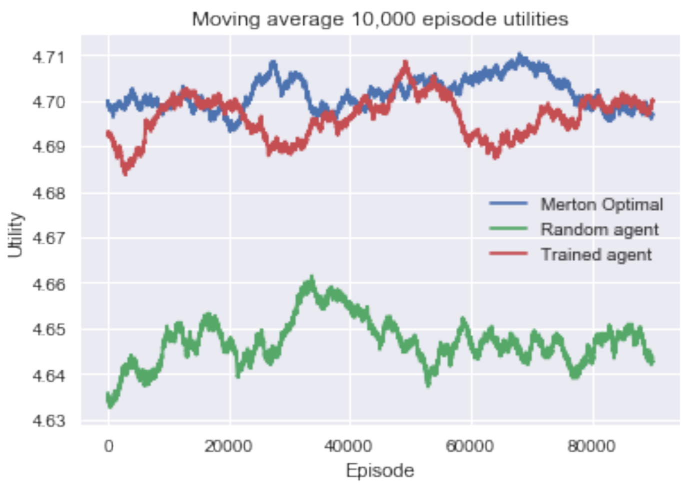


Figure 5.38: Moving average utilities Noisy Deep Q v Merton, $\sigma = 0.20$

	Random Agent	Noisy Deep Q	Merton Optimal
Utility Distribution	4.6462	4.6965	4.7008
Rewards Distribution	3.7337	11.2096	10.6893
Wealth Distribution	122.98	119.55	120.00
Sharpe Ratio Distribution	0.3064	0.4162	0.3881

Table 5.24: Noisy Deep Q, test performance v Merton - Mean

	Random Agent	Noisy Deep Q	Merton Optimal
Utility Distribution	0.0172	0.0140	0.0120
Reward Distribution	1.8445	1.4985	1.3590
Wealth Distribution	2.2585	1.5621	1.4887
Sharpe Ratio Distribution	0.0802	0.0952	0.0845

Table 5.25: Noisy DeepQ, test performance v Merton

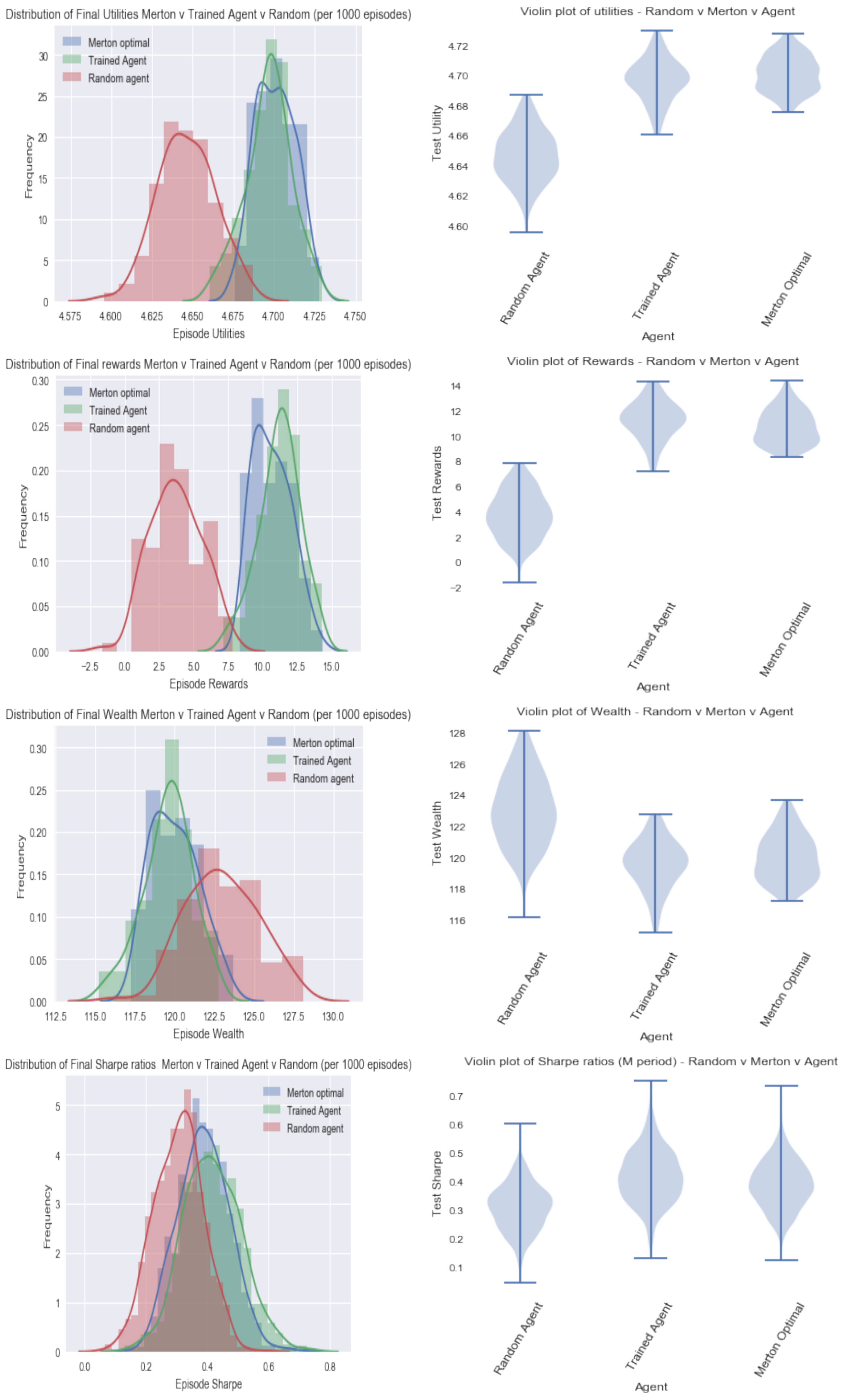


Figure 5.39: Noisy Deep Q learning: Utilities verses Merton, $\sigma = 0.20$

Chapter 6

General Conclusions

6.1 Main Conclusions

In this thesis I used model free reinforcement learning techniques and applied them to the Merton problem. An analytical solution of Merton's problem is only possible for specific known stochastic processes, with known parameters and then derived for a specific utility curve (in chapter 2 I show one of the derivations which involves techniques from stochastic optimal control). In contrast an RL agent was able to learn to trade purely from the data over multiple time periods closely matching the test performance of a Merton optimal agent.

A major part of solving an RL problem lies in creating the environment and agent. In this particular problem I had to reshape an end of period expected utility of wealth $\mathbf{E}[U(w_T)]$ into a reward suitable for an RL agent to learn from on a step-wise basis. I approached this first by choosing a particularly amenable utility curve such as log-utility and a 'friendly' stochastic process such as geometric brownian motion. Here some very basic mathematics which I show in chapter 4 enables an exact step-wise reward matching the required end of period expected utility.

From there I wished to generalise the RL to be applicable to any utility curve (and indeed a wider range of stochastic processes), here I used work by Ritter, 2017 [35] where he used a mean-variance approximation to formulate rewards in order to learn to trade an Ornstein-Uhlenbeck process with transaction costs. By applying this to the Merton problem I was able to move beyond log-utility and could also tune the RL agent's risk aversion.

A second issue which may perhaps be the key issue in financial markets is noise. Many recent RL successes have come from high dimensional environments (such as Atari), where the environment is relatively stable and rewards although perhaps delayed are not necessarily very noisy. Indeed the agent often also has considerable control not only of his future rewards, but indeed the next state he is likely to find himself within. These problems are often solved by reducing the dimensionality using a function approximator such as a neural network and indeed in my later experiments this is precisely the method I chose to handle higher dimensional state spaces.

The issue with rewards being very noisy is that the difference in the distribution of rewards and utilities for even a perfect Merton optimal agent and a random agent can quite difficult to ascertain unless one replays millions of episodes, and more difficult still as noise levels increase as the distributions increasingly overlap.

This problem resulted in my being keen to understand the actual test set distributions of rewards, wealth and sharpe ratios of my trained RL agent, compared to Merton and Random. The trained RL managed to achieve a far superior distribution of outcomes compared to the random agent, and for the later experiments using DeepQ and Double DeepQ learning and Noisy nets essentially traded with perfection, exactly matching a merton optimal agent simply using raw price and wealth levels as inputs.

6.2 Further Directions

I feel this thesis gives a proof of concept, but that there are various somewhat obvious extensions possible both on the machine learning side and the finance side.

On the finance side the problem setting really needs to be far more practical. Clearly by using deep neural nets in RL we are now in a position to easily increase the number of inputs into the state space by giving more features and information that could be relevant for future prices in practice. Real markets are not exact stochastic processes and the agent could learn to exploit this. Real markets also have costs, transaction costs such as commissions, spread and slippage, however this can be accommodated by shaping the reward for the RL agent. Another extension is of course not just using multiple features, but multiple assets, this is perhaps a more difficult problem but definitely necessary to move towards practical applicability.

Remaining on the finance side, I believe RL can be applied to a variety of problems and indeed the Merton problem may not actually be the most suitable one (despite a relatively small success in this thesis). Problems such as market making, optimal execution, liquidation and some areas of statistical arbitrage may be highly suitable. In particular I would look for problems where the agent not only learns from rewards, but where he is capable of having greater influence over the next state he finds himself in. In the Merton problem he is a 'price taker', he can learn actions affecting the future probability distribution of his wealth, but much of the state of this environment comprises the price process of the market and over this the agent has no control. In contrast if we consider an RL agent trained to optimise the utility of wealth through market making on the full limit order book, then the agent can directly affect that order book through his own trades and trade cancellations. In this game speed is of the essence and I believe current algorithms are relatively quick, but as far as I know from the industry do not believe that the use of RL algorithms is yet the norm and I believe they should be well suited to this problem.

The RL agent is also impractical because it needs millions of episodes in order to learn. From a finance perspective an immediate and practical option could be to attempt a semi-parametric approach. I have previously had some success using Gaussian processes in finance, which given a range a prior kernels were highly efficient at learning the underlying process as a posterior with relatively small amounts of data. The benefit here is that we are using a bayesian approach and can train a gaussian process (GP) on a small amount of real life data, but then use the trained GP to generate as much data as we need, which can in turn train the RL agent. Of course GP's being gaussian are not necessarily the best fit for real life markets, but there are extensions such as t-processes as shown by Shah and Gharamani at Cambridge University [41].

The semi-parametric approach however is a compromise. There are clearly more sophisticated model based reinforcement learning methods than I have used (where the agent learns his own model of the environment) which I think should be more sample efficient in this respect. And also although I briefly examined different methods of exploration, such as Noisy nets and prioritised experience replay, this merely touches the tip of the iceberg as regards exploration.

My final suggested directions come back to the heart of the problem in this thesis. If we wish to maximise only expected wealth then the reward formulation for an RL is immediate, if however we wish to maximise the expected future utility of terminal wealth it is certainly not. This requires some effort to shape rewards, but what we are actually doing in reality is risk-aware reinforcement learning of an end of period utility (which has risk aversion implicitly embedded into it) and this we wish to maximise. Risk-aware RL is an active research area which again may be applicable to these types of finance problems.

Finally, in RL setting we learn to maximise expected rewards (albeit where I have shaped those rewards to account for risk) which may not be the most efficient as it does not use all the information available. For this noisy problem setting I am actually more interested in the overall distribution of rewards, utilities and wealth - not their expected values. The RL agents used q-values which approximated an expected reward. However Bellemere et al. 2017 [2] recently derived algorithms for performing reinforcement learning from a distributional viewpoint, i.e. instead of learning state values, they learn the value distributions and indeed derive some state of the art results. To my mind this is appealing as one can immediately be able to examine not just q-values but the entire q-distributions and examine not only if the RL agent is learning, but also its certainty. In finance where there is a great deal of uncertainty this could prove useful to know.

Chapter 7

Colophon

For coding I used Python 3.6, together with the Anaconda environment as well as the PyTorch libraries. Some of the later code was either adapted or in the case of Noisy Nets the model created in the excellent https://github.com/higgsfield/RL-Adventure. My later market environment was also ported to be able to inherit features from the OpenAI gym, http://openai.com/.

Bibliography

[1] Ang, A. (2014). *Asset management: A systematic approach to factor investing*. Oxford University Press.

[2] Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*.

[3] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[4] Björk, T. (2009). *Arbitrage theory in continuous time*. Oxford university press.

[5] Borch, K. (1969). A note on uncertainty and indifference curves. *The Review of Economic Studies*, 36(1):1–4.

[6] Cartea, Á., Jaimungal, S., and Penalva, J. (2015). *Algorithmic and high-frequency trading*. Cambridge University Press.

[7] Chan, E. (2009). *Quantitative trading: how to build your own algorithmic trading business*, volume 430. John Wiley & Sons.

[8] Chan, E. (2013). *Algorithmic trading: winning strategies and their rationale*. John Wiley & Sons.

[9] Chapados, N. (2011). *Portfolio choice problems: An introductory survey of single and multiperiod models*. Springer Science & Business Media.

[10] Chapados, N. and Bengio, Y. (2007). Forecasting and trading commodity contract spreads with gaussian processes. In *13th International Conference on Computing in Economics and Finance*.

[11] Dickey, D. A. and Fuller, W. A. (1979). Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association*, 74(366a):427–431.

[12] Farrell, M. T. and Correa, A. (2007). Gaussian process regression models for predicting stock trends. *Relation*, 10:3414.

[13] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., et al. (2017). Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*.

[14] Gillespie, D. T. (1996). Exact numerical simulation of the ornstein-uhlenbeck process and its integral. *Physical review E*, 54(2):2084.

[15] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

[16] Grinold, R. C. and Kahn, R. N. (2000). Active portfolio management.

[17] Halperin, I. (2017). Qlbs: Q-learner in the black-scholes (-merton) worlds. *arXiv preprint arXiv:1712.04609*.

[18] Hamilton, J. D. (1994). *Time series analysis*, volume 2. Princeton university press Princeton.

[19] Hasselt, H. V. (2010). Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621.

[20] Hastie, T., Tibshirani, R., and Friedman, J. (2002). The elements of statistical learning: Data mining, inference, and prediction. *Biometrics*.

[21] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

[22] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[23] Levy, H. and Markowitz, H. M. (1979). Approximating expected utility by a function of mean and variance. *The American Economic Review*, 69(3):308–317.

[24] Markowitz, H. (1952). Portfolio selection. *The journal of finance*, 7(1):77–91.

[25] Merton, R. C. (1975). Optimum consumption and portfolio rules in a continuous-time model. In *Stochastic Optimization Models in Finance*, pages 621–661. Elsevier.

[26] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

[27] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[28] Moody, J. and Saffell, M. (2001). Learning to trade via direct reinforcement. *IEEE transactions on neural Networks*, 12(4):875–889.

[29] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

[30] Neuneier, R. (1996). Optimal asset allocation using adaptive dynamic programming. In *Advances in Neural Information Processing Systems*, pages 952–958.

[31] Øksendal, B. (2003). Stochastic differential equations. In *Stochastic differential equations*, pages 65–84. Springer.

[32] Osband, I., Van Roy, B., and Wen, Z. (2014). Generalization and exploration via randomized value functions. *arXiv preprint arXiv:1402.0635*.

[33] Rasmussen, C. E. and Williams, C. K. (2006). *Gaussian processes for machine learning*, volume 1. MIT press Cambridge.

[34] Razzak, M. I., Naz, S., and Zaib, A. (2018). Deep learning for medical image processing: Overview, challenges and the future. In *Classification in BioApps*, pages 323–350. Springer.

[35] Ritter, G. (2017). Machine learning for trading.

[36] Roncalli, T. (2013). *Introduction to risk parity and budgeting*. CRC Press.

[37] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

[38] Rubio, D. M. Convergence analysis of an adaptive method of gradient descent.

[39] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[40] Scholkopf, B. and Smola, A. J. (2001). *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press.

[41] Shah, A., Wilson, A., and Ghahramani, Z. (2014). Student-t processes as alternatives to gaussian processes. In *Artificial Intelligence and Statistics*, pages 877–885.

[42] Sharpe, W. F. (1964). Capital asset prices: A theory of market equilibrium under conditions of risk. *The journal of finance*, 19(3):425–442.

[43] Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel methods for pattern analysis*. Cambridge university press.

[44] Shreve, S. E. (2004). *Stochastic calculus for finance II: Continuous-time models*, volume 11. Springer Science & Business Media.

[45] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

[46] Smith, W. (2010). On the simulation and estimation of the mean-reverting ornstein-uhlenbeck process. *Commodities Markets and Modelling*.

[47] Spooner, T., Fearnley, J., Savani, R., and Koukorinis, A. (2018). Market making via reinforcement learning. *arXiv preprint arXiv:1804.04216*.

[48] Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163.

[49] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

[50] Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical review*, 36(5):823.

[51] van den Berg, T. Calibrating the ornstein-uhlenbeck (vasicek) model. `https://www.sitmo.com/?p=134`. Accessed: 2017-08-02.

[52] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ.

[53] Vasicek, O. (1977). An equilibrium characterization of the term structure. *Journal of financial economics*, 5(2):177–188.

[54] Von Neumann, J. and Morgenstern, O. (2007). *Theory of games and economic behavior (commemorative edition)*. Princeton university press.

[55] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.