HEC MONTRÉAL

Reinforcement Learning Algorithms for a Dynamic Goal-Based Wealth Management Problem

par

Maxence Prémont

Sciences de la gestion
(Option Ingénierie Financière)

Mémoire présenté en vue de l'obtention
du grade de maîtrise ès sciences
(M. Sc.)

Octobre 2021

À Maman et Papa,

Résumé

Ce mémoire explore l'utilisation d'algorithmes d'apprentissage par renforcement (RL) pour résoudre un problème de gestion de richesse par objectif (GBWM) sur plusieurs périodes. Le but de cette approche est de maximiser la probabilité d'atteindre l'objectif de richesse terminale d'un investisseur. Habituellement, un algorithme de programmation dynamique (DP) est utilisé pour résoudre ce problème d'allocation d'actifs. Cependant, l'emphase est placée sur l'implantation d'un algorithme Q-learning récemment proposé qui a été construit spécifiquement pour les problèmes GBWM. L'utilisation d'un algorithme de RL, même si celui-ci ne peut qu'approximer la solution, peut être avantageuse puisqu'il, contrairement à la DP, est capable de résoudre efficacement des problèmes de grandes dimensions et d'inclure facilement des contraintes dépendantes de la trajectoire de l'actif comme les coûts de transactions. Deux contributions qui améliorent la performance de l'algorithme sont proposées. Avec ces améliorations en place, le Q-learning est capable d'approximer la solution référence produite en utilisant la DP. Finalement, un autre type d'algorithme de RL, le Policy Gradient, est exploré. Bien qu'un problème simplifié soit résolu avec succès, l'algorithme n'est malheureusement pas capable d'approximer la solution de référence pour les problèmes GBWM plus complexes.

Mots-clés

Apprentissage par renforcement, Q-learning, programmation dynamique, gestion de richesse par objectif dynamique

Abstract

This M.Sc. thesis explores the use of reinforcement learning (RL) algorithms to solve a dynamic goal based wealth management (GBWM) problem, where the objective is to maximize an investor's probability of surpassing her terminal wealth goal. This asset allocation problem is usually solved with a dynamic programming algorithm. However, this thesis focuses on the implementation of a specific type of RL algorithm (Q-learning) specifically tailored for the GBWM problem examined here. Using RL, even if it can only approximate the solution, could be advantageous since, unlike dynamic programming, it can avoid the curse of dimensionality and easily integrate path-dependant constraints like transaction costs. Two contributions, which improve the algorithm's performance, are then proposed. Unlike the previously presented algorithm, the improved version is able to correctly approximate the benchmark solution obtained with dynamic programming. Finally, another type of RL algorithm, Policy Gradient, is explored. While it successfully solved simplified problems, it was unfortunately not able to accurately approximate the benchmark solution for the full version of the GBWM problem.

Keywords

Reinforcement learning, Q-learning, Policy Gradient, Dynamic Programming, Dynamic Goal-Based Wealth Management

Contents

List of Tables

List of Figures

Acknowledgements

I would like to sincerely thank my co-directors, Michel Denault and Jean-Guy Simonato, whose expertise and wise advices were invaluable in the realization of this thesis. Your generous availability, insightful feedbacks and guidance allowed me to overcome obstacles and constantly improve the quality of my research. I am deeply touched by the direct and indirect support received from my friends and family throughout this seemingly never-ending process. A special thanks goes to my mom, Line, and dad, Marc, for their unconditional love and support during the entirety of my studies. My academic, professional, and personal accomplishments would not be the same without you.

Chapter 1

Introduction

This M.Sc. thesis examines the use of reinforcement learning algorithms to solve a dynamic goal based wealth management problem. For decades, the financial industry's portfolio management decisions have been based on Modern Portfolio Theory (MPT) introduced by Markowitz (1952), which optimizes the risk-return trade-off by constructing efficient portfolios that maximize returns for a given level of risk, defined as the standard deviation of returns. In contrast, the objective of goal-based wealth management (GBWM) is to maximize an investor's probability of surpassing her terminal wealth goal. The risk is then defined as the likelihood of failing to reach the goal, which is a much more intuitive interpretation of risk for retail investors.

This relatively new investment approach is built around the ideas put forward in behavioural portfolio theory (BPT) from Shefrin and Statman (2000), which integrates two widely known and accepted decision-making biases and concepts, prospect theory and mental accounts (MA).Thus, it departs from the rationality assumption of expected utility theory embedded in MPT. Introduced by Kahneman and Tversky (1979), prospect theory describes the actual behaviour of investors by taking into account loss aversion, an asymmetric form of risk aversion in which potential gains and losses are assessed differently and depend on the specific situation. Additionally, MA, as described in Thaler (1985) and Thaler (1999), refers to the observation that investors tend to subdivide their wealth in

different underlying goals, each associated with a different risk aversion.

The first specific GBWM framework taking into account these ideas was proposed by Chhabra (2005), which as been widely followed by wealth management practitioners, as noted in Brunel (2015). Further developments combined the objectives of GBWM with portfolio volatility optimization in a static (one-period) model, making the approach fully consistent with MPT. This was first done for a MA framework in Das et al. (2010), and followed by a complete analysis in a GBWM setting from Das et al. (2018). Implementing this static model period by period in a dynamic (multi period) problem is not optimal, as it can lead to short-sighted decisions. The difference between dynamic and myopic asset allocation decisions was clearly illustrated in Campbell and Viceira (1999) using a CRRA utility function with predictable returns. Instead, the optimal solution of a dynamic GBWM problem is calculated using a dynamic programming algorithm specifically tailored for GBWM presented in Das et al. (2020).

While this type of backward recursion algorithm has been widely used to accurately solve other financial engineering problems, it can become increasingly inefficient, and even infeasible, as the dimension of the problem becomes greater and path dependent constraints, such as transaction fees, are included. Thus, another type of algorithms, reinforcement learning (RL), has gained interest in the financial industry, as it can avoid these problems. This led to the implementation of a tabular Q-learning algorithm (a specific type of RL algorithms) to solve a GBWM problem in Das and Varma (2020). This paper serves as the backbone of this thesis, which has two objectives. First, implement the tabular Q-learning algorithm, replicate the results presented in Das and Varma (2020), and improve its performance and convergence. Second, explore another type of RL algorithm by implementing the policy gradient algorithm and assess its ability to solve a dynamic GBWM problem.

The thesis is structured in the following way. Chapter 1 starts by presenting a brief literature review, and finishes with the elaboration of the dynamic goal-based wealth management problem. Afterwards, in chapter 2, the dynamic programming algorithm, used as a benchmark for the reinforcement learning algorithms, is briefly explained. This is followed by a detailed presentation of the Q-learning algorithm of Das and Varma (2020) and the improved version proposed in this thesis. Chapter 3 assesses the Q-learning model's ability to approximate the benchmark solution by presenting the numerical results. Finally, chapter 4 details the Policy Gradient algorithm, which is first tested with a simplified environment to gain intuition on the advantages and potential shortfalls. Then, by progressively increasing the complexity, the algorithm's ability to approximate the benchmark solution is evaluated.

1.1 Literature Review

Avoiding the curse of dimensionality in not unique to reinforcement learning algorithms. Other methods using regressions and simulations are widely documented in the multi-period portfolio optimization literature. Specifically, these methods were showed to be efficient in Brandt et al. (2005), Garlappi and Skoulakis (2010), and Van Binsbergen and Brandt (2007). These authors propose simulation and regression algorithms which use a Taylor expansion of the value function with a combination of Monte Carlo simulations and least-squares regression to approximate the expected values. Alternatively, Denault et al. (2017) propose a regression and simulation algorithm which avoids the Taylor series by regressing on both the state and decision variables. The resulting algorithm is fast and precise, while being able to handle continuous decision variables.

The first application of reinforcement learning algorithms to financial problems, done by Neuneier (1996) and Neuneier (1998), optimized dynamic multi-asset portfolio trading with simulated and real data using a Q-learning algorithm, which was able to handle transaction costs, risk preference, and allocation constraints. Other early work by Moody

et al. (1998) and Moody and Saffell (2001) focused on direct RL with recurrent reinforcement learning (RRL) algorithms to build single and multi asset trading strategies. In contrast to Q-learning, direct RL, which is a policy-based algorithm, ignores the value function by directly adjusting the parameters of a policy function. This method can thus completely avoid the curse of dimensionality. The results showed that using direct RL with RRL offers considerable advantages over regular Q-learning.

Since the publication of these seminal papers, much research has been conducted in the application of RL algorithms to various financial problems. Concerning direct learning and RRL, the same method used in Moody and Saffell (2001) was applied to a strategic asset allocation problem in Hens and Wöhrmann (2007). Using real data sampled from different international markets, it is shown that the method is able to actively time the market and consistently outperform it. Another deep direct learning RL algorithm, which proposes a task-aware back propagation method to deal with the gradient vanishing issue of deep learning, is implemented in Deng et al. (2016). The trading system was able to perform well with equities and commodities. In Dempster and Leemans (2006), a multi-layer system consisting of a RRL algorithm, a risk management module, and a dynamic utility optimization layer is introduced. By using the dynamic optimization layer to automatically adjust the RRL's hyperparameter, the algorithm is able to generate consistent Out-Of-Sample foreign exchange (FX) trading returns and avoid large draw-downs while limiting the need for human intervention and tuning. RRL was also applied in FX markets by Gold (2003), which assessed the impact of adding an extra hidden layer to the neural network. It was concluded that the one layer networks can outperform the two layer model, inferring that using simpler models might be advantageous with noisy financial data. Almahdi and Yang (2017) proposed a trading decision system based on RRL, which can handle multi-asset portfolio trading. Using an expected maximum drawdown risk objective function yielded superior returns than other measures such as the Sharpe ratio and the Sterling ratio.

Further research has also been made regarding Q-learning. In Casqueiro and Rodrigues (2006), a Q-learning algorithm is implemented and used to derive a single-asset trading strategy that maximizes the Sharpe ratio. Another Q-learning single-asset trading system was implemented in Bertoluzzo and Corazza (2012), this time using linear and kernel function approximations. Two RL algorithms, SARSA (On-policy) and Q-learning (Off-policy) were implemented and compared in Pendharkar and Cusatis (2018) in the context of a personal retirement portfolio management problem with discrete actions and states. Jeong and Kim (2019) proposed a deep Q-learning model with two branches, one that learns the action values and one that learns the number of shares to buy/sell to maximize the objective function. The model was able to vastly outperform a fixed-number trading system, which trades only a constant number of shares, on real data from different equity indices. In Park et al. (2020), another deep Q-learning trading agent was proposed. This model is able to determine the trading direction and size for a multi-asset portfolio. Additional constraints were also used to ensure practical applicability. A Deep Deterministic Policy Gradient algorithm, which is a mix of policy-based and value-based RL, was implemented in Jiang et al. (2017). Applied to a portfolio of cryptocurrencies, the model showed promising results by outperforming other trading strategies, even with high transaction cost.

More models have been proposed to take into account the particular nature of financial data. The ability to handle regime switching models was implemented in Maringer and Ramtohul (2012) with their smooth transition RRL algorithm. This new algorithm was able to outperform standard RRL algorithms when confronted with datasets composed of distinct regimes. Additionally, Bellmare et al. (2017) introduced distributional reinforcement learning, a method which learns the distribution of the value function, in contrast with traditional RL that only estimates the value function's mean. This technique was applied in Bossaerts et al. (2020) to deal with the leptokurtic nature of financial data. It is shown that distributional RL offers a significant improvement in convergence in an environment with leptokurtic rewards, even when the state and action space is small. Fur-

thermore, G-learning, a probabilistic extension to Q-learning which does not assume a data generation process and is suitable for noisy data, was applied to a GBWM problem by Dixon and Halperin (2020).

Besides trading systems and portfolio management, many financial applications of deep learning and RL algorithms have been proposed, such as hedging an option portfolio over multiple periods (see Buehler et al. (2019) and Cao et al. (2021)). More applications are discussed at length in Dixon et al. (2020), de Prado (2020), Guida (2018), Coqueret and Guida (2018), and Kolm and Ritter (2019) to name only a few good references.

1.2 Static Goal Based Wealth Management

This section presents a short introduction to the static GBWM problem, explaining how an investor would solve allocate her wealth according to this approach. As it was previously mentioned, the studied GBWM framework is fully consistent with MPT, which is centred around the concept of the mean-variance efficient frontier. Since an investor's objective, under MPT, is to maximize her risk-return trade-off, a mean-variance portfolio, constructed with a weight vector w, minimizes volatility for a given rate of return target r. This is calculated with the mean return vector $\bar{m}$ and covariance matrix S of the N available assets. The resulting optimization problem can be written as follows:

$$\begin{aligned} \min_{w} \; & w^{\top} \mathsf{S} w \\ \text{s.t. } & w^{\top} \bar{m} = r \\ & w^{\top} e = 1; \end{aligned}$$

where e is a vector of ones with the appropriate size. Additionally, as it is often the case in the literature, a constraint to prohibit short-selling is included. The solution of this optimization produces a single point on the frontier. The rest of the frontier is built by repeating the minimization for a different target rate. The portfolio selected by the

investor figures on the efficient frontier and is chosen according to her risk tolerance, which is expressed in terms of return volatility.

In contrast, goal-based wealth management proposes a completely different investment philosophy. It all starts with the investment objective, where an investor, rather than maximize, on a risk-adjusted basis, her terminal wealth W_T, focuses on maximizing the probability of reaching a terminal wealth goal G:

$$\max_{w} \mathbb{P}[W_T \quad G]$$

This objective leads to a new definition of risk: the probability of failing to reach a wealth goal $\mathbb{P}[W_T < G]$. This new interpretation of risk has important implications regarding portfolio management. Whereas decreasing the standard deviation of an underfunded portfolio is perceived as a risk-reducing action in the MPT framework, under GBWM such an action could actually increase the risk as it decreases the likelihood of attaining the goal. However, even if with a different risk interpretation, the goal-based framework is consistent with the mean-variance approach because it exclusively chooses portfolios located on the efficient frontier. Thus, to solve a static GBWM problem, an investor needs to select the mean-variance efficient portfolio that maximizes her success probability.

1.3 Dynamic Goal Based Wealth Management

This section details the dynamic GBWM problem studied throughout this thesis. Things get more complicated in a dynamic context, as it requires the investor to find the investment strategy that maximizes the overall success probability, rather than myopically optimizing period by period. This strategy dictates, for each wealth and time to maturity combination considered, the mean-variance portfolio that should be selected by the investor given her current wealth and the time remaining to reach her goal. Doing so requires a model to simulate the evolution of an investor's wealth and how it affected by her actions. Here, a Markov decision process is used, as it was done in Das et al. (2020). The

following sections define Markov decision processes and detail the specific characteristics, and the general rule s they follow, of the dynamic GBWM. To guarantee a reliable benchmark, the same specifications are used as in Das and Varma (2020).

1.3.1 Markov Decision Process

This section explains Markov Decision Processes (MDP), the class of models used to solve the dynamic (multi-period) goal-based wealth management optimization problem. These models formalize sequential decision-making, where future states and rewards are influenced by the current action taken. Indeed, in an MDP, the next state W_{t+1} is generated from the transition probabilities $\mathbb{P}[W_{t+1}|W_t; a_t]$, which depend on the current state W_t and the action taken a_t. From this new state, a reward R_t is received. By repeating the same process, the state W_{t+2} and reward R_{t+1} are generated from state W_{t+1} and action a_{t+1}, which depend on the initial state W and action a. Mathematically, an MDP is defined with discrete time-steps $t = 0; :::; T$ and a tuple $(\mathbb{W}; \mathbb{A}; \mathbb{T}; \mathbb{R})$, which is composed of the possible states, actions, transition probabilities, and rewards respectively.

The transition probability set $\mathbb{T}$ specifies, for given states $W_t; W_{t+1} \in \mathbb{W}$ and an action $a_t \in \mathbb{A}$, the probability $\mathbb{P}[W_{t+1}|W_t; a_t]$ of transitioning to state W_{t+1} given state W_t and action a_t. Even if the previous actions affect all the subsequent states and rewards, this probability, because it follows a Markovian process, only depends on the current state and action:

$$\mathbb{P}[W_{t+1}|W_t; a_t] = \mathbb{P}[W_{t+1}|W_t; W_{t-1}; :::; W_0; a_t; a_{t-1}; :::; a_0]:$$

To ensure that the transition probabilities are consistent probability measures, the total probability of transitioning from a given state W and action a must equal one:

$$\sum_{W_{t+1} \in \mathbb{W}} \mathbb{P}[W_{t+1}|W_t; a_t] = 1 \quad \forall W_t \in \mathbb{W};\ a_t \in \mathbb{A}:$$

By going through the MDP, a sequential choice of action based on the observed states will inevitably be made. These actions are chosen based on a given policy π, which is

simply the mapping $\pi(W) \rightarrow a$ from an observed state W to an action a. The amount of possible policies is virtually limitless. However, from all these policies, there exists at least one that is superior to the others. This optimal policy π^* identifies the best actions to take for every possible sequence of states accounted for by the model. The objective of an MDP is to find that optimal policy.

1.3.2 Numerical Application Specifics

This section defines the action-space $\mathbb{A}$, state-space $\mathbb{W}$, transition probabilities, and rewards of the MDP structure used to model the dynamic goal-based wealth management problem solved in this thesis. Specifically, the objective is to maximize the likelihood $\mathbb{P}[W_T \geq G]$ of observing a terminal wealth W_T greater or equal than the predetermined wealth goal $G = 200$ if the investor is initially worth $W_0 = 100$, has an investment horizon of $T = 10$ years, and can rebalance annually. Since the state and action spaces are discrete, the model is classified as a finite Markov Decision Process. Given an investible universe of N assets with a mean return vector $\bar{\mu}$ and a covariance matrix Σ, the investor will be allowed to choose a portfolio between a set of discretized and limited actions. Each action is defined as a pair $a = (\mu, \sigma)$ representing respectively the mean and standard deviation of the chosen portfolio. In the studied problem, the action space is based on the three assets available to the investor:

$$\bar{\mu} = \begin{bmatrix} 0.05 \\ 0.10 \\ 0.25 \end{bmatrix}; \Sigma = \begin{bmatrix} 0.0025 & 0 & 0 \\ 0 & 0.04 & 0.02 \\ 0 & 0.02 & 0.25 \end{bmatrix}.$$

Each portfolio is located on the mean-variance efficient frontier built with these three assets. The action space $\mathbb{A}$ is constructed by first specifying the minimal and maximal required rate of return. The minimal rate of return $\mu_{\min}$ is associated with the minimum-variance portfolio, which solves the previously described optimization problem. To avoid short-selling, the maximum return $\mu_{\max}$ can be set to, at most, the maximum of the return vector $\bar{\mu}$. The entire action space can then be calculated by minimizing the variance of the

rate of returns that are equally spaced on the range $[\mu_{\min}; \mu_{\max}]$. Using this method with the three available assets, a set of 15 portfolios composing the action space are calculated. It was shown in Das et al. (2020) that 15 actions are sufficient to maintain the accuracy of the results. The set of means and variances for each of these portfolios on the frontier is as follows:

	Portfolios							
	1	2	3	4	5	6	7	8
μ	0.0526	0.0552	0.0577	0.0603	0.0629	0.0655	0.0680	0.0706
σ	0.0485	0.0486	0.0493	0.0508	0.0529	0.0556	0.0587	0.0623

	Portfolios						
	9	10	11	12	13	14	15
μ	0.0732	0.0757	0.0783	0.0809	0.0835	0.0860	0.0886
σ	0.0662	0.0705	0.0749	0.0796	0.0844	0.0894	0.0945

The MDP simulates wealth trajectories with the geometric Brownian motion, a fairly basic stochastic process, given by:

$$W_{t+1} = W_t e^{(\mu_t - \frac{\sigma_t^2}{2})t + \sigma_t \sqrt{t} Z}; \text{ where } Z \sim N(0;1) \text{ and } t = t \quad (t \quad 1): \tag{1.1}$$

It is important to note that, while this MDP assumes that the portfolio returns are normally distributed, more complex or realistic distributions and stochastic processes could have easily been used instead. From this portfolio return model, the probability of transitioning from a given state W_t and action $a_t = (\mu_t; \sigma_t)$ to the state W_{t+1} can be calculated by first defining $f(z)$ as the value of the probability density function of the standard normal distribution such that $Z = z$:

$$\widetilde{\mathbb{P}}[W_{t+1} | W_t; a_t] = f\left(\frac{\ln(\frac{W_{t+1}}{W_t}) \quad (\mu_t \quad \frac{\sigma_t^2}{2})}{\sigma_t}\right): \tag{1.2}$$

As expected, these probabilities only depend on the wealth value at the previous timestep and the selected action a_t. Because equation (1.3) does not guarantee that the total probability of transitioning from a given state W_t and action a_t is equal to one, the proba-

bilities are normalized:

$$\mathbb{P}[W_{t+1}|W_t; a_t] = \frac{\widetilde{\mathbb{P}}[W_{t+1}|W_t; a_t]}{\sum_{W_{t+1} \in \mathbb{W}} \widetilde{\mathbb{P}}[W_{t+1}|W_t; a_t]}. \tag{1.3}$$

The chosen stochastic process is also used to define the discrete state space $\mathbb{W}$. Indeed, the set of wealth values $\mathbb{W}$ is constructed using three points, the starting value W_0, the minimum value $W_{\min}$, and the maximum value $W_{\max}$. The extremities of the grid are calculated with equation (1.1) using the lowest mean $\mu_{\min}$, the highest mean $\mu_{\max}$, and the highest volatility $\sigma_{\max}$ from the available portfolios, which provides the range of wealth values that are most likely to be observed:

$$[W_{\min}; W_{\max}] = [W_0 e^{(\mu_{\min} - \frac{1}{2}\sigma_{\max})T - 3\sigma_{\max}\sqrt{T}}; W_0 e^{(\mu_{\max} - \frac{1}{2}\sigma_{\max})T + 3\sigma_{\max}\sqrt{T}}].$$

Using the previously defined action space, the limits of the grid are set at $W_{max} = 568$ and $W_{min} = 66$. The wealth space is the same for all time-steps from $t = 0$ up to T. Because the time-steps are annual, the portfolio is rebalanced once a year, at the beginning of the time-step. To form a uniform grid, the wealth values are identical for all time-steps. The number of grid points will depend on the investment horizon and is set to $10T + 1$. These points are selected such that they all are equally spaced over the range $[\ln(W_{\min}); \ln(W_{\max})]$. To improve the stability of the results, which could significantly vary depending on the number of grid points, a midpoint technique is applied. This grid adjustment ensures that the point $\ln(G)$ is exactly halfway between the two closest grid points and can be done by simply shifting the entire grid up or down, which conserves the equally distanced property. The benefits of using such an adjustment compared to the grid used in Das and Varma (2020) is detailed in Denault and Simonato (2021). An investor's initial success probability is calculated with a simple linear interpolation if the initial wealth W_0 is not in the grid. The resulting state grid is assumed to be fully connected: transitioning from any given wealth level to any other wealth level is possible.

Considering GBWM's objective of maximizing the success probability, it would be wise to design the reward structure in a way which provides a direct interpretation of the

results. Consequently, no rewards are attributed in the intermediate time-steps. Instead, a single binary reward is assigned at maturity, which corresponds to the success probability. Either the investor attained or surpassed her goal G and receives a reward of 1, or she failed and receives nothing. Thus, the reward R_t received by the investor after transitioning to a wealth W_t follows this simple equation:

$$R_t = \begin{cases} 0 \text{ if } W_t < G \textbf{ or } t < T \\ 1 \text{ if } W_t \quad G \textbf{ and } t = T \end{cases} :$$

This definition concludes the characterization of the specific goal-based wealth management problem, which will be solved in chapter 3 with different algorithms detailed in the next chapter. Specifically, chapter 2 presents the benchmark algorithm, dynamic programming, as well as two versions of a Q-learning algorithm.

Chapter 2

Dynamic Programming and Q-Learning Algorithms

This chapter presents three algorithms used to solve the dynamic goal-based wealth management problem. First, the well known and commonly used dynamic programming algorithm, which will serve as a benchmark for the reinforcement learning algorithms, is detailed as it was done in Das et al. (2020). Two RL algorithms will also be discussed in this chapter. One of them, which will be referred as the base Q-Learning algorithm, directly comes from Das and Varma (2020). The other, also from the Q-Learning family of algorithms, proposes an improvement over the base case by implementing a decaying learning rate. The main objective of those two RL algorithms is to replicate the optimal solution produced by the dynamic programming algorithm. Doing so would open the door to the application of such RL algorithms in more complex and realistic wealth management problems for which the dynamic programming approach is unfeasible. In contrast to what was done in Das and Varma (2020), the algorithms' ability to replicate the optimal solution will be assessed with both the In-Sample and Out-Of-Sample performance.

2.1 Dynamic Programming

Introduced by Bellman (1952), dynamic programming is a backward recursion algorithm that maximizes an objective function, which is often referred to as the value function. This algorithm is commonly used to solve financial engineering problems like portfolio management and option pricing (see Bellman and Dreyfus (2015) and Bellman (2003) for more examples of applications). Because this approach assumes that the transition probabilities are known, it is considered as a model-based algorithm. In a Markov decision process, the value function is defined as the expected value of future rewards. To fit specific problems, the rewards can take multiple forms, such as a payoff function or even a utility function. Thus, in general terms, the value function $V^{\pi}(W_0)$ of the initial state W_0 following the policy π is defined by the expected sum of rewards $R(W_t; W_{t+1})$:

$$V^{\pi}(W_0) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{T-1} \gamma^t \, R(W_t; W_{t+1}) \mid W_0\right] \tag{2.1}$$

Where the term $\mathbb{E}^{\pi}[\,]$ is the expectation operator under the policy π and γ is a discount factor. With some simple manipulations and the Markov property, the value function $V^{\pi}(W_t)$ of a given state in equation (2.1) can be expressed as the expected value of the next reward and the next state's value function $V^{\pi}(W_{t+1})$, as shown below.

$$\begin{aligned} V^{\pi}(W_t) &= \mathbb{E}^{\pi}\left[R(W_t; W_{t+1}) + \sum_{i=1}^{T-t-1} \gamma^i \, R(W_{t+i}; W_{t+i+1}) \mid W_t\right] \\ &= \mathbb{E}^{\pi}\left[R(W_t; W_{t+1}) + \gamma \, \mathbb{E}^{\pi}[V^{\pi}(W_{t+1}) \mid W_{t+1}] \mid W_t\right] \\ &= \mathbb{E}^{\pi}\left[R(W_t; W_{t+1}) + \gamma \, V^{\pi}(W_{t+1}) \mid W_t\right] \end{aligned} \tag{2.2}$$

This relationship across the different states is the basis of the algorithm and is referred to as the Bellman Expectation Equation (BEE). While each policy π produces a different value function, at least one is superior to the others; the optimal policy π^*. The objective of DP is to find the value function $V^*(W_t)$ associated with the optimal policy. This value function satisfies the Bellman Optimality Equation (BOE) which simply finds the policy

π that maximizes $V^{\pi}(W_t)$ for all states:

$$V^{*}(W_t) = \max_{\pi} V^{\pi}(W_t) \; \forall W_t \in \mathbb{W} \tag{2.3}$$

To remain consistent with the interpretation of goal-based wealth management' objective, the discount factor γ is set to one. Because the reward is known at maturity, as shown in chapter 1, and the value function is defined as the sum of the expected future rewards, the terminal value function $V(W_T)$ for a given wealth W_T at maturity T is also known and equals the corresponding reward R_T. Furthermore, since only a fixed number of wealth levels are considered, the expectation term can be decomposed by simply taking the weighted average of the next period's value function with the corresponding transition probabilities defined in equation (1.3). This implies that, using the previously defined reward structure with equations (2.2) and (2.3), the optimal value function $V^{*}(W_t)$ for a given wealth W_t and time-step t can be calculated with the following equation:

$$\begin{aligned} V^{*}(W_t) &= \max_{\pi} \mathbb{E}^{\pi}\left[V^{*}(W_{t+1})\right] \\ &= \max_{a_t \in \mathbb{A}} \left[\sum_{W_{t+1} \in \mathbb{W}} \mathbb{P}[W_{t+1} | W_t; a_t] \, V^{*}(W_{t+1}) \right] \end{aligned} \tag{2.4}$$

From this equation, the optimal policy $\pi^{*}(W_t)$ can be easily defined as:

$$\pi^{*}(W_t) = \arg\max_{a_t \in \mathbb{A}} \left[\sum_{W_{t+1} \in \mathbb{W}} \mathbb{P}[W_{t+1} | W_t; a_t] \, V^{*}(W_{t+1}) \right] \tag{2.5}$$

The algorithm starts at $t = T - 1$, one time-step before the end of the investment horizon. The value function and the optimal policy for this time-step are then calculated using the known values of $V(W_T)$ with the equations (2.4) and (2.5). Once both functions are computed for every wealth point, the algorithm moves back one time-step to $t = T - 2$ and repeats the same process by using the previously calculated $V(W_{T-1})$. The same procedure is applied until the start of the investment horizon t_0 is reached, which effectively maps the value function and optimal policy of the specified state space. The pseudo-code of the algorithm is presented below.

Result : value function V and optimal policy π

set terminal conditions $V(W_T) \quad \mathbb{1}_{W \in G} \; \forall W \in \mathbb{W}$;

for $t = T - 1$ **to** 0 **do**

  for $W_t \in \mathbb{W}$ **do**

$$\pi(W_t) = \arg\max_{a_t \in \mathbb{A}} \left[\sum_{W_{t+1} \in \mathbb{W}} \mathbb{P}[W_{t+1} | W_t, a_t] \, V(W_{t+1}) \right];$$

$$V(W_t) \quad \sum_{W_{t+1} \in \mathbb{W}} \mathbb{P}[W_{t+1} | W_t, \pi(W_t)] \, V(W_{t+1});$$

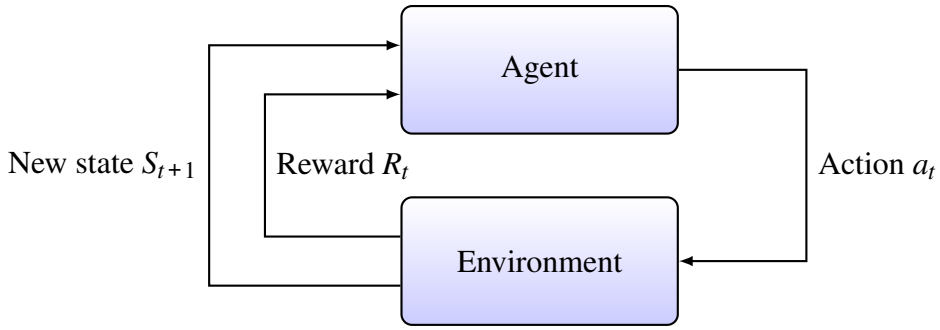
  end

end

Since the wealth values and the actions are limited and stay constant throughout the time-steps, the transition probabilities could be calculated once at the start of the algorithm, stored as a three dimension tensor, and referenced when needed. This additional step considerably improves the efficiency of the algorithm.

2.2 Q-Learning

First introduced in Watkins (1989) and Watkins and Dayan (1992), Q-learning rapidly became one of the most popular reinforcement learning (RL) algorithm. These iterative algorithms have been shown to converge for finite Markov Decision Processes under certain conditions. In RL terminology, an agent is defined as anything which interacts with its environment by taking autonomous actions to achieve a goal. They solve their environment by following a simple procedure shown in the flow chart below. First, the agent, in a given state S_t, chooses an action a_t. This action will generate, via the environment, a reward R_t and the next state S_{t+1}. The agent then uses the reward to adjust its strategy, effectively learning the best actions. By repeating this process multiple times, this simple algorithm is able to solve complex environments.



As noted in Dixon et al. (2020), three main factors differentiate reinforcement learning and the two other classes of machine learning algorithms. First, unlike supervised learning, the agent receives only partial feedback after taking an action. While a numerical reward is distributed, it does not explicitly indicate which action would produce the maximal reward. Second, the presence of a feedback loop, where the actions of an agent influence the environment which ultimately affects all the future actions, is unique to reinforcement learning. Such a feedback loop forces the agent to plan its actions to avoid selecting high local rewards that negatively affect the future environment. Third, since the agent doesn't know if it attained the maximal reward, it constantly explores the environment, by selecting random actions, to make sure that a higher reward doesn't exist. Thus, the agent is tasked with balancing exploitation (the use of its current knowledge of the environment to choose the best action) and exploration.

Q-learning is part of the Temporal Difference (TD) learning subcategory of value-based RL algorithms. As mentioned in Sutton and Barto (2018) and Dixon et al. (2020), TD learning shares similarities with DP, since it uses previously learned estimates and the Bellman optimality equations to updates its estimates. However, both approaches use these equations in a fundamentally different way. DP algorithms are designed to find the exact solution of the equations. To do so, they require a known model of the environment and a sufficiently low dimensionality, which is rarely the case for more complex and realistic problems. On the other hand, TD learning, relies directly on the data and experience without assuming a specific model of the environment's dynamics. Since it

relies on noisy data, TD methods do not seek to find the exact solution as it is done in DP. Instead, they focus on approximating the solution with fast and marginal improvements. This small change in objective gives TD algorithm enough flexibility to deal with the curse of dimensionality.

TD learning algorithms can also be separated in two classes, on-policy, such as SARSA, and off-policy algorithms like Q-learning. While the value function estimate of on-policy algorithms estimate relies on a specific policy, off-policy algorithms' value function estimate are completely independent of the followed policy. In technical terms, the on-policy algorithm assumes that the policy used to generate the path is an optimal policy, whereas the off-policy algorithm assumes that the path could've been generated by a random policy. Q-learning's ability to learn from off-policy data is extremely desirable in practice.

Overall, Q-learning possesses several characteristics which renders it a great candidate to solve the discussed dynamic goal-based wealth management problem. Besides being compatible with the previously described MDP and being model-free, Q-learning is a forward recursion algorithm, which allows path dependent rewards. These characteristics indicate that RL algorithms could potentially solve even more complex environments. The following sections present in details the two Q-learning algorithms that will be used to replicate the results of the dynamic programming algorithm. First, the base algorithm, as presented in Das and Varma (2020) is detailed. Then, some improvements related to the base algorithm are proposed.

2.2.1 Base Algorithm

This section first presents the general mathematical framework of the Q-learning algorithm. This general formulation is then applied to the studied problem by defining the specific characteristics of the algorithm. Considering the sequential nature of RL algorithms, where an agent in state W_t receives a reward $R(W_t; a_t; W_{t+1})$ after taking action a_t and transitioning into state W_{t+1}, the value function $V^p(W_t)$ for the policy p can be

defined as the expected total rewards starting at state W_t, time-step t, and following the policy π until the end of the episode T:

$$V^{\pi}(W_t) = \mathbb{E}^{\pi}\left[\sum_{i=0}^{T-t} \gamma^i R(W_{t+i}; a_{t+i}; W_{t+i+1}) \mid W_t\right]: \tag{2.6}$$

Because $V^{\pi}(W_t)$ strictly follows the policy π, it only differs for different states W_t, which is why it is referred to as the state-value function. In RL, it is relevant to also specify an action-value function $Q^{\pi}(W_t; a_t)$, which gives the value of being in state W_t, taking action a_t first, and then following the policy π:

$$Q_t^{\pi}(W_t; a_t) = \mathbb{E}_t^{\pi}\left[\sum_{i=0}^{T-t} \gamma^i R(W_{t+i}; a_{t+i}; W_{t+i+1}) \mid W_t; a_t\right]: \tag{2.7}$$

These two equations can be simplified by separating the first reward $R(W_t; a_t; W_{t+1})$ from the sum, as shown below. By doing so, the remaining sum is simply expressed as the state-value function of the next state $V^{\pi}(W_{t+1})$. Both simplifications yield the same equation because the action-value function is assumed to follow the policy π, just like the state-value function, after the first action:

$$Q^{\pi}(W_t; a_t) = \mathbb{E}^{\pi}[R(W_t; a_t; W_{t+1})] + \gamma\, \mathbb{E}^{\pi}[V^{\pi}(W_{t+1})]: \tag{2.8}$$

This is known as the Bellman equation of the action-value function. Similarly to DP, the RL algorithm's objective is to maximize the value function by developing the optimal policy π^*. This policy is superior to all other policies for every state:

$$V^*(W_t) = \max_{\pi} V^{\pi}(W_t) \; \forall W_t \in \mathbb{W}:$$

By definition, the optimal action-value function $Q^*(W_t; a_t)$ is maximized for every state $W_t \in \mathbb{W}$ given a_t as the first action. By maximizing this function over the first action a_t, extends the optimal policy π^* to the first action. Thus, it becomes equivalent to the optimal state-value function:

$$V^*(W_t) = \max_{a_t} Q^*(W_t; a_t):$$

Using this relationship, equation (2.7) can be expressed without the state-value function which yields the Bellman optimality equation of the action-value:

$$Q\ (W_t; a_t) = \mathbb{E}\ \left[R(W_t; a_t; W_{t+1}) + g \max_{a_{t+1}} Q\ (W_{t+1}; a_{t+1}) \right]: \tag{2.9}$$

Because Q-learning is a model-free algorithm, the transition probabilities are provided by the environment and are not known by the agent. This is in part why Q-learning uses the action-value function, as these probabilities are required by the sate-value function. Without a transition model, the expectation term in equation (2.9) can not be exactly calculated as it was done in the DP algorithm. Instead, TD methods such as Q-learning estimate the expectation by sampling a single observation. While this estimation method of computing the mean yields volatile results, it allows the algorithm to update its value function extremely fast. Over numerous training episodes, the average marginal improvement of the value function can produce an efficient algorithm.

By sampling over the next time-step, the agent approximates the expected term of equation (2.9) with the observed the reward R_t and next state W_{t+1}. With the expectation term now removed, it is possible to calculate the mismatch between the right-hand side and the left-hand side for the action-value function of a given policy $Q^p(W_t; a)$. Using this error term, the action-value function for state W_t and action a_t can be updated using a simple weighted average:

$$Q(W_t; a_t) \quad (1 \quad a)Q(W_t; a_t) + a \left[R_t + g \max_{a_{t+1}} Q(W_{t+1}; a_{t+1}) \right]: \tag{2.10}$$

The weight a refers to the learning rate of the agent and indicates the speed at which new information is absorbed in the value function. With this last step, the derivation of the general Q-learning formula is completed. Referring back to the RL flow chart, two major components of the RL algorithm still need to be detail: the action selection process and the rewards. While general rules are available, these components are often problem specific. The following specifications are made for the goal-based wealth management problem as presented in Das and Varma (2020).

Before receiving a reward and updating its value function, a RL agent needs to choose an action. For TD models, the exploration-exploitation trade-off is primordial. Whilst reaching the optimal solution obviously requires the agent to exploit its knowledge of the environment by selecting the "best" action, it is also dependent on a proper exploration of the environment to ensure that potentially optimal actions aren't ignored. In academia, the preferred way to properly control the exploration throughout the training phase, is to use the ϵ-greedy policy. This simple, yet effective, policy works by first specifying an exploration rate ϵ between zero and one, which correspond, respectively, to an exploitation-only and exploration-only policy. Then, by sampling and comparing a random number $u \sim U(0;1)$ to the exploration rate ϵ, the agent selects an action. If u is greater than ϵ, the optimal (greedy) action is chosen. Otherwise, the agent selects a random action, as shown by:

$$a = \begin{cases} \arg\max_{a \in \mathbb{A}} Q(W;t;a) & \text{if } u \geq \epsilon \\ \text{random action from } \mathbb{A} & \text{if } u < \epsilon \end{cases} :$$

As it will be shown in the following chapter, insufficient exploration will most likely result in a sub-optimal solution, as the agent gets stuck in a local optimum. On the other hand, excessive exploration, apart from slowing down the convergence, could hinder the agent's ability to fine-tune its policy which yields again a sub-optimal policy.

Even if no rewards are accorded in intermediate time-steps, the Q-Learning algorithm is still able to update its value function before the end of the episode by using the next step's Q-value estimate. With this reward scheme, which closely resembles the one used in the DP algorithm, the general Q-learning equation (2.10) can be modified to fit in the goal-based environment:

$$Q(W_t;a_t) \leftarrow \begin{cases} (1-\alpha)Q(W_t;a_t) + \alpha\,\gamma \max_{a_{t+1}} Q(W_{t+1};a_{t+1}) & \text{if } t < T \\ (1-\alpha)Q(W_t;a_t) + \alpha\,R_t & \text{if } t = T: \end{cases} \tag{2.11}$$

As it was done in Das and Varma (2020), the learning rate α is set to a small, constant number throughout the training phase. It is interesting to note that, as detailed in Sutton and Barto (2018), TD(0) algorithms, such as Q-Learning, with a sufficiently small and constant step-size α have been proved to converge in the mean under certain conditions. Also, to stay consistent with the interpretation of the Q-value as the success probabilities, the discount factor γ is set to one. Because Q-learning is a value-based RL algorithm which uses a value function to derive the optimal policy, such a policy for a given action-value function can be simply extracted for every state $(W; t)$ from each state's maximum Q-value:

$$\pi^*(W_t) = \arg\max_{a_t \in \mathbb{A}} Q(W_t; a_t) \quad \forall W_t \in \mathbb{W}:$$

To recapitulate, the RL learns the optimal policy of an environment by replaying the investment episode $N_{episodes}$ times. This is done by first selecting an action with the ϵ-greedy policy, which influences the next state. Then, a reward based on the next state and the proposed reward scheme is received and used to update the action-value function for the corresponding state and action. Once the training phase is over, the optimal policy is extracted. The complete algorithm is showed in a pseudocode format below.

Result : Q function Q and optimal policy π

for k 1 **to** $N_{episodes}$ **do**

assign starting wealth W_t W_0;

for t 0 **to** T **do**

generate random number u $U(0;1)$;

if $u < \epsilon$ **then**

choose action a_t randomly from $\mathbb{A}$;

else

choose best action a_t $\arg\max_{a_t \in \mathbb{A}} Q(W_t; a_t)$;

end

if $t < T$ **then**

transition to next state W_{t+1} with a_t;

$Q(W_t; a_t)$ $(1 \quad \alpha) Q(W_t; a_t) + \alpha \left[\gamma \max_{a_{t+1}} Q(W_{t+1}; a_{t+1}) \right]$;

update wealth W_t W_{t+1};

else

calculate reward R_t $\mathbb{1}_{W \quad G}$;

$Q(W_t; a_t)$ $(1 \quad \alpha) Q(W_t; a_t) + \alpha R_t$

end

end

end

extract optimal policy $\pi(W_t)$ $\arg\max_{a \in \mathbb{A}} Q(W_t; a) \ \forall W_t \in \mathbb{W}$

2.2.2 Improved Algorithm

This section presents the second Q-learning algorithm, which integrates an improvement over the base algorithm described in the previous section. The underlying structure and mathematical formulation stays exactly the same. In fact, only the learning rate a, which stays constant throughout the training phase in the base algorithm, is changed. Whilst a constant a has been proved to converge in the mean under certain conditions, most theoretical applications use a decreasing learning rate, as explained in Sutton and Barto (2018). As it will be shown in the following chapter, this adjustment greatly improves the convergence and the performance of the base Q-learning algorithm.

Intuitively, a decreasing learning rate allows the agent to reduce its sensitivity to shocks, potentially caused by white noise, as the training phase progresses. Stochastic approximation theory provides the theoretical background to learning rate decay. Indeed, it has been shown by Robbins and Monro (1951) that an iterative method of computing the mean, such as Q-Learning, will converge to the true value with probability 1, if the learning rate a_k respects the conditions (2.12). While the first condition guarantees that the initial steps are large enough to overcome random variance and initial sub-optimal values, the second condition guarantees that the steps will become small enough to ensure the convergence of the algorithm towards the true value. As mentioned in Sutton and Barto (2018), satisfying conditions (2.12) will often lead to a very slow convergence and require a lot of fine-tuning, which is why they are not always used in applications and empirical research.

$$\lim_{K \to \infty} \sum_{k=1}^{K} a_k = \infty \quad \lim_{K \to \infty} \sum_{k=1}^{K} a_k^2 < \infty \tag{2.12}$$

Multiple definitions of a_k are possible. The method implemented here follows Sutton and Barto (2018) and denotes a_k as the k^{th} selection of action a at the wealth level W_t and the time-step t, which allows Q-values to have different learning rates throughout the training process. States $\{W_t; t; a\}$ that are visited more often will have smaller step-sizes and thus require a larger number of episodes to diverges from the previous Q-values.

It is interesting to note that a relatively large constant step-size a, as used in Das and Varma (2020), does not satisfy the second condition in (2.12). However, this could be considered beneficial when the environment is non-stationary. Indeed, because the estimates never completely converge and continue varying with the new observations, the agent would be able to adapt to changes in the environment. This is not the case for the specific problem that is studied here, but could still be relevant for future work on the application of reinforcement learning algorithm to more complex financial settings such as regime switching environments.

While multiple other forms are possible, the learning rate decay function implemented follows an exponential function. One of the most basic forms is proposed in Sutton and Barto (2018) where the learning rate is given by $a_k = \frac{1}{k}$. However, this function provides little flexibility regarding the shape of the curves. Thus, the improvement proposed on the base Q-learning algorithm uses a more general and flexible decay function as proposed in Spall (2003), which respects convergence conditions (2.12):

$$a_k = \frac{a}{(k + A)^b}: \tag{2.13}$$

Since the interpretation of the parameters isn't necessarily intuitive at first sight, it is useful to redefine them as a function of parameters that can be easily understood, such as an upper bound a_{init} and a lower bound a_{end}. The initial learning rate a_1 ca be set to the upper bound. Similarly, since the lowest possible value of the learning rate is limited by the total number of episodes N, a_N is set to the lower bound. the original parameters can then be redefined:

$$A = \left(\frac{a}{a_{\text{init}}}\right)^{\frac{1}{b}} \quad 1 \text{ and } a = \left(\frac{N \quad 1}{a_{\text{end}}^{\frac{1}{b}} \quad a_{\text{init}}^{\frac{1}{b}}}\right)^b :$$

This way, the shape of the decay function can be controlled by three parameters; the starting value a_{init}, the ending value a_{end}, and the diminution speed b. The intuitive interpretation of the parameters facilitates the calibration of the decay function, as it provides

insight on the range of parameters to test. This is particularly important because the optimal choice of a learning rate sequence is often problem specific (Dixon et al. (2020)) and theoretical frameworks offer no clear general law (Sutton and Barto (2018)), which means that the step-size needs to be selected manually, by experimenting different decay functions.

While only one combination of parameters of this decay function will be compared to the base algorithm, multiple combinations were tested. Another type of decay function, a simple decreasing linear function, was also tested:

$$a_k = \frac{a_{\text{end}} \quad a_{\text{init}}}{N \quad 1}(k \quad 1) + a_{\text{init}}. \tag{2.14}$$

Furthermore, a second definition, referred to as the epoch method, of the learning rate a_k was explored. In this method, briefly introduced in Dixon et al. (2020), k is simply defined as the current episode number. Contrary to the previously defined state method, each Q-factor will have the same step-size and thus early exploration is extremely important. In total, four different decay functions were tested across multiple parameter combinations. While the tables reporting the results of these tests are available in the appendix (1, 2,3, and 4), a more detailed analysis will be made regarding the final choice of the decay function in the following chapter.

Concretely, this improvement requires a simple adjustment to the base algorithm. Besides the variable learning rate a_k which needs to be computed at each time-step for every episode, only one small modification is required. The Q-value updating formula now include the variable rate a_k and is given by:

$$Q(W_t; a_t) \quad \begin{cases} (1 \quad a_k)Q(W_t; a_t) + a_k g \max\limits_{a_{t+1}} Q(W_{t+1}; a_{t+1}) & \text{if } t < T \\ (1 \quad a_k)Q(W_t; a_t) + a_k [\mathbb{1}_{W_t \quad G}] & \text{if } t = T \end{cases}.$$

Whilst this adjustment might seem insignificant at first sight, the results presented in the next chapter will prove beyond any doubt that it is necessary to reach an acceptable performance compared to the benchmark.

Chapter 3

Numerical Results of the Q-Learning Replication

This chapter implements the three algorithms previously detailed to solve the same dynamic goal-based wealth management problem as in Das and Varma (2020), which has been described in chapter 1, and analyses the numerical results. In this setting, the investor starts with a wealth of $W_0 = 100$ and aims to maximize the likelihood of surpassing her terminal wealth goal of $G = 200$ after $T = 10$ years. To do so, the investor can choose, at each time-step, to invest in one of the 15 portfolios are available: The state space is com-

	Portfolios							
	1	2	3	4	5	6	7	8
m	0.0526	0.0552	0.0577	0.0603	0.0629	0.0655	0.0680	0.0706
s	0.0485	0.0486	0.0493	0.0508	0.0529	0.0556	0.0587	0.0623

	Portfolios						
	9	10	11	12	13	14	15
m	0.0732	0.0757	0.0783	0.0809	0.0835	0.0860	0.0886
s	0.0662	0.0705	0.0749	0.0796	0.0844	0.0894	0.0945

posed of 101 wealth values, which are based on the maximum $W_{\max} = 568$ and minimum $W_{\min} = 66$ wealth values. The following sections first discuss the numerical performance of the algorithm and present the Out-Of-Sample method used to test the results. Then, the benchmark solution is detailed and replicated by both Q-learning algorithms.

3.1 Numerical Performance: Python vs. Matlab

Beside personal preferences, the choice of a programming language can be influenced by the execution speed and by the ease at which the code can be implemented. Two higher level languages were assessed; Python and Matlab. Figure 3.1 presents a speed comparison of both programming languages running the same code 30 times. This histogram shows that Matlab is definitely able to run the algorithm faster than Python (about 4 times faster). Considering that the implementation of the algorithm is fairly easy in both languages, the speed was the determining factor in the final choice.

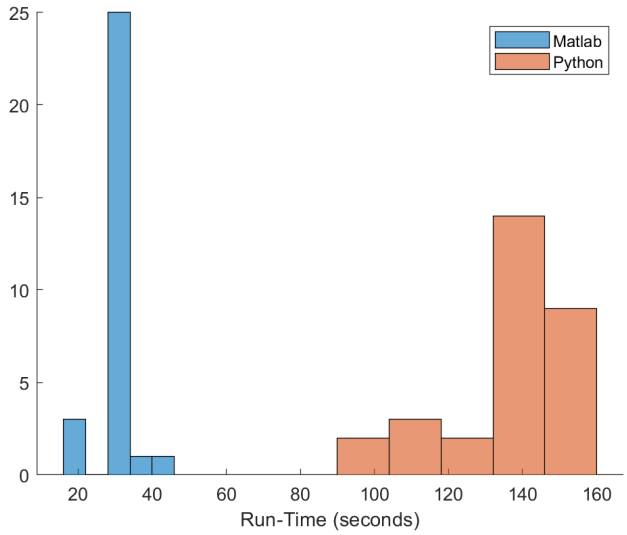


Figure 3.1: Speed comparison between Matlab and Python

Furthermore, the run-time for both languages could be significantly reduced by first compiling the code in C, which can be done by different modules in both languages. In Matlab, pre-compiling the code yielded an algorithm that is roughly 6 times faster (5 seconds vs. 30 seconds). Another time-saving addition, which proves especially valuable when replicating the results and adjusting the parameters of the Q-Learning algorithm, is the use of parallel computing. Overall, these adjustments exponentially reduced the computing require to produce the results presented in the following section.

3.2 Out-Of-Sample Testing

This section further explains the Out-Of-Sample (OOS) procedure used to test DP and RL policies. This testing method is the gold standard to measure the true performance of a model's investment strategy. Simply put, OOS, contrary to In-Sample (IS), tests a given model with data excluded from the sample used to fit the same model. While multiple types of cross-validation exist, the most relevant OOS test is simply using a model's policy to simulate numerous wealth trajectories. It is worth mentioning that Out-Of-Sample testing of the RL agent's optimal policy was not done in Das and Varma (2020), the paper on which this research is based. This contribution proved to be particularly insightful while assessing the convergence and performance of the Q-learning algorithm.

The trajectory simulation process of the OOS testing uses the same discrete state space and transition probabilities as the Markov Decision Process described in chapter 1. Unsurprisingly, it is the same MDP which is being solved by the DP and RL algorithms. At first sight, it may seem superfluous to test a policy in the same environment as it was trained. However, for both DP and RL algorithms, the In-Sample success rate is taken directly from the value function. The goal of the OOS tests is to verify that the success rate provided by the value function truly represents the likelihood of surpassing an investor's goal. For DP, the deviation between the OOS and IS success rates are negligible. This is not systematically the case for RL algorithms, where the magnitude of the deviation depends on whether the algorithms can correctly solve the environment and converge to a solution or not.

Concretely, simulating an Out-Of-Sample trajectory follows simple steps: select an action for given wealth and time-step using the optimal policy (RL agents do not use the ϵ-greedy policy to select an action), calculate the corresponding transition probabilities and randomly transition to the next state, and repeat the same process until the end of the investment horizon. After simulating numerous wealth trajectories, a success rate (num-

ber of successful trajectories divided by the total number of trajectories) can be calculated and compared with the value function's In-Sample result.

3.3 Benchmark Solution

This section presents the results of the benchmark; the DP algorithm. The resulting success probability and optimal policy are referred to as the optimal solution. Using the inputs from above, the DP algorithm suggests that an investor would have a **71**:**59**% chance of surpassing her goal at the end of a 10-year investment horizon if she follows the optimal policy. Figure 3.2 presents the resulting optimal policy and value function for a subset of the state space. Whereas the value function heatmap shows the success probability for a given wealth W and time-step t, the optimal policy heatmap displays the optimal action, represented as a number from 1 to 15 (1 being the safest action and 15 being the riskiest).

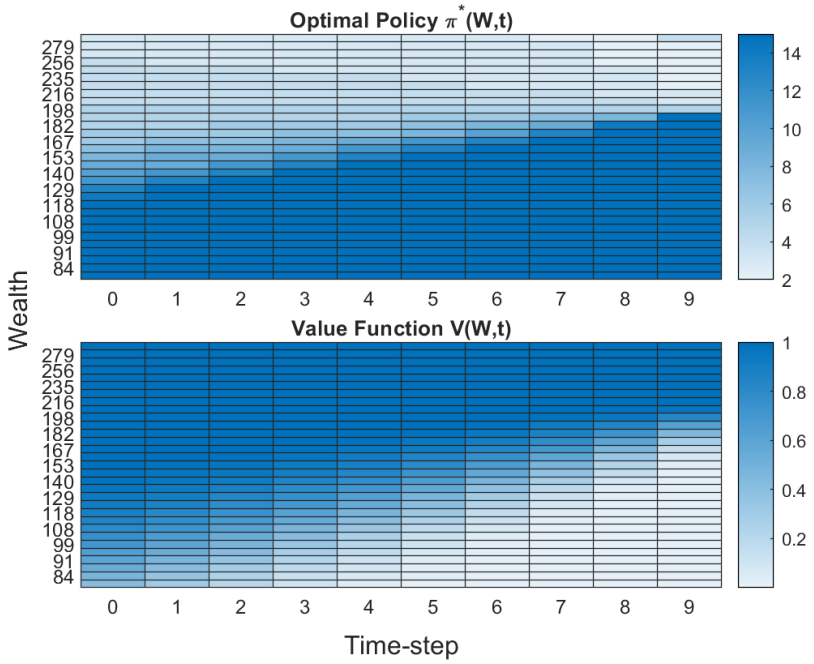


Figure 3.2: Dynamic programming optimal policy and value function

The insights provided by the solution follow an intuitive logic. Indeed, by looking at the policy matrix, the optimal strategy seems to choose a risky action, which increases the chances of larger wealth gains (and losses) when the current wealth level is below a certain threshold; and chooses a safe action, which essentially protects the previous gains, when the investor finds herself above the same decision threshold. As the investment episode progresses, and since the success probability for a wealth level $W < G$ decreases over time, as shown by the value function in figure 3.2, the investor increases her risk appetite, at the expanse of potentially incurring bigger losses, in the hopes of reaching her goal. Thus, the decision threshold, which indicates the critical wealth value where the investor seems to switch her risk preferences, will increase throughout the investment episode. The exact values of this threshold obviously depends on the available actions and the investment objective and horizon.

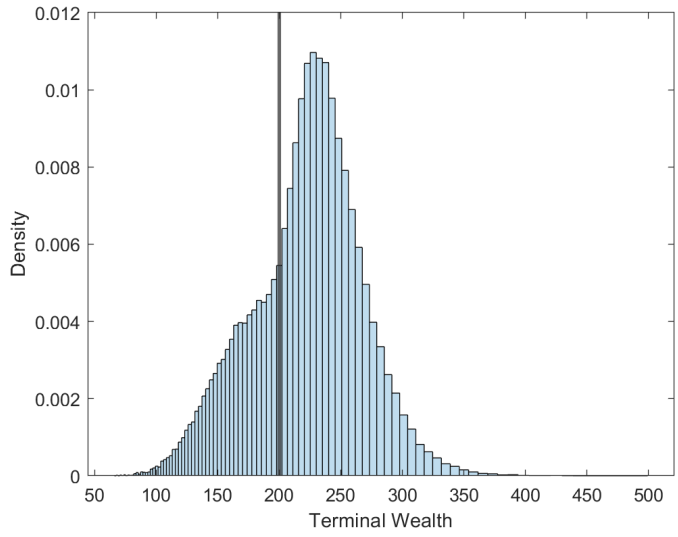


Figure 3.3: Dynamic programming empirical terminal wealth distribution

Unsurprisingly, using the optimal policy derived with the DP algorithm to simulate 100,000 trajectories yields the same success rate of 71:59%. Figure 3.3 displays the distribution of each trajectory's wealth at maturity. The distribution's particular shape of the curve reflects the risk-aversion shift that characterizes the optimal policy. Because the op-

timal policy favours risky portfolios when the wealth is below the goal, the distribution of the corresponding wealth values (to the left of the black vertical line) exhibits high variance. On the other hand, the distribution of wealth values to the right of the goal line are more concentrated, which is due to the optimal policy favouring safer portfolios. As expected, the simulated results of the DP solution are consistent to the IS results previously discussed.

The policy's logical interpretation is also part of the optimal solution, which is used to evaluate the ability of RL to solve goal-based wealth management problems. The objective of the RL algorithms is not only to reach the optimal success rate, but most importantly to produce a policy that follows the same logic. Since RL is a model-free algorithm, it is unrealistic to expect these algorithms to reproduce the exact optimal solution. Instead, RL algorithms are expected to correctly approximate the optimal solution. Beyond the In-Sample success rate provided by the value function, the robustness of the RL policies will also be assessed with Out-Of-Sample simulations.

3.4 Q-learning Replication

This section evaluates the ability of the implemented base Q-learning algorithm to approximate the optimal solution and to replicate the In-Sample results from Das and Varma (2020). The algorithm's success probabilities for different training parameters (exploration rate ϵ and the number of training episodes, also referred to as epochs) are presented in table 3.1. The column "DV Value" directly reports the results presented in Das and Varma (2020), which seem to have been obtained with only one simulation. It is important to remember that these results use a different wealth grid. Fortunately, using the grid described in chapter 1 did not significantly alter the results for both the DP and RL algorithms. The columns under "Simulations" present summary metrics of the 100 different simulations that were computed for each parameter combination. After training 100 different agents, descriptive metrics are calculated on the In-Sample success rates. These

metrics give the reader a deeper understanding of the training outcome variation compared to only reporting the results of a single training session. Here, a training session (or phase) is defined as going through the investment episode for each epoch.

			Simulations					
ϵ	Epochs	DV Value	Mean	Min	25%	Median	75%	Max
DP	1	72%	71.59%	-	-	-	-	-
0.10	50K	65%	60.55%	56.23%	58.58%	60.62%	61.78%	68.52%
0.10	100K	65%	63.48%	57.17%	61.79%	63.29%	64.76%	69.31%
0.20	50K	69%	64.80%	60.53%	63.26%	64.87%	66.01%	68.49%
0.20	100K	71%	67.28%	63.46%	65.71%	66.95%	69.03%	73.42%
0.25	100K	71%	68.59%	64.74%	66.92%	68.32%	70.01%	74.42%
0.30	50K	72%	68.42%	65.07%	67.20%	68.32%	69.83%	72.33%
0.30	100K	72%	70.25%	66.10%	69.05%	69.76%	71.57%	75.07%
0.40	50K	73%	71.36%	68.13%	70.40%	71.32%	72.22%	74.61%
0.40	100K	77%	73.46%	69.91%	72.04%	73.29%	74.59%	78.62%
0.40	200K	75%	74.31%	70.94%	72.64%	74.17%	75.43%	79.29%
0.40	500K	71%	74.34%	69.36%	72.72%	74.30%	76.31%	78.83%

Table 3.1: Q-Learning In-Sample replication results using a constant learning rate $\alpha = 0.1$, an initial wealth $W_0 = 100$, a terminal wealth goal $G = 200$, an investment horizon of $T = 10$ years, and 101 grid points.

Looking at the means of the simulations, the impact of the exploration rate ϵ can clearly be observed. Low values of ϵ failed to produce a single simulation where the success rate reached the expected level, which suggests that the lack of exploration limited the agent in its ability to find and try new policies. On the other hand, larger values produced results that are way higher than the expected success rate, which indicates that the agent would require additional epochs for its policy to truly converged. Additional epochs might be beneficial for the other ϵ values as well. Indeed, significant improvement in the success rate is noticeable when comparing the agents trained on 50k and 100k epochs. Considering that the results, even at 100k epochs, have not totally reached the expected success rate, training the agent on more epochs could be an interesting avenue to explore.

While the DV values don't exactly match the mean of the simulations, they all figure in the possible range. Most values are located in the fourth quartile, which indicates that the results presented in Das and Varma (2020) might have slightly overestimated the true In-Sample success rate of the algorithm. This small deviation isn't really concerning. However, what does raise a valid concern is the variance of the simulation results, which is much larger than one would have expected from an algorithm that has truly converged.

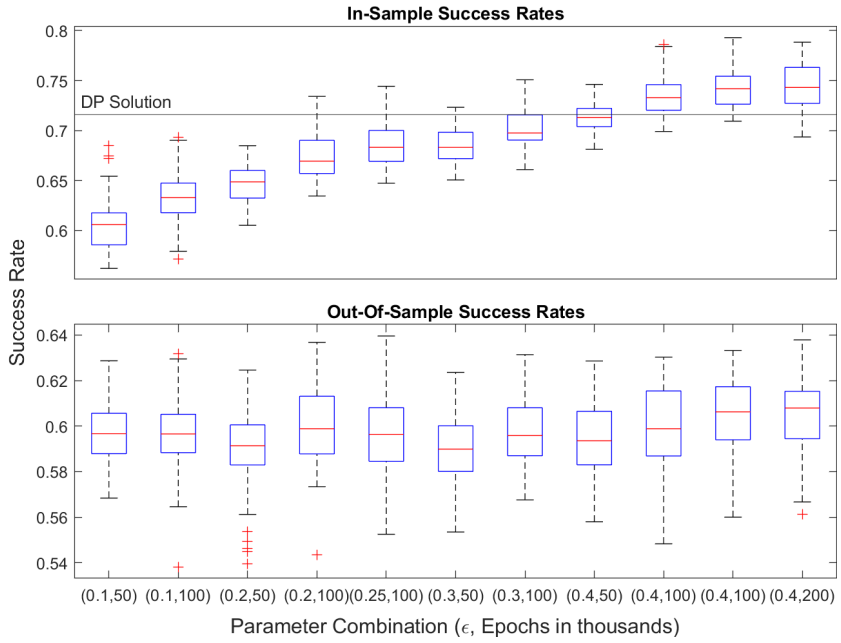


Figure 3.4: Distribution of In-Sample and Out-Of-Sample results for different parameter combinations of the base Q-Learning algorithm

Before confirming the Q-learning agent's ability to correctly approximate the optimal solution, it would be wise to test the results policies with Out-Of-Sample trajectories, as previously discussed. One could expect a converging RL algorithm to produce Out-Of-Sample results that closely resembles the In-Sample success rate. Additionally, to be considered a good alternative, the RL agent should produce success rates (In-Sample and Out-Of-Sample) that also match the ones from the dynamic programming algorithm. For a given parameter combination, the 100 agents that were previously trained each generated 10,000 trajectories using the previously described OOS procedure. The distribution of

these results are presented for every parameter combination considered in figure 3.4 in the form of a box plot. The labels on the x-axis refer to the tuple of parameters (ϵ, Epochs) used in the training phase.

Unfortunately, this figure clearly shows that the OOS success rates are nowhere near the expected performance. The Out-Of-Sample success rates are concentrated around 60%, with a maximum well below the expected 71.65% success rate. Furthermore, contrary to the IS distribution, little difference can be seen between the distribution of the different parameter combinations. From this figure, it seems that the OOS results are somewhat uncorrelated with the IS success rates. This observation is quite surprising and might suggest that the RL agents are not able to learn the optimal policy, or they simply have not yet converged, even though the In-Sample success rates seemed to converge towards the optimal solution at the end of the training phase.

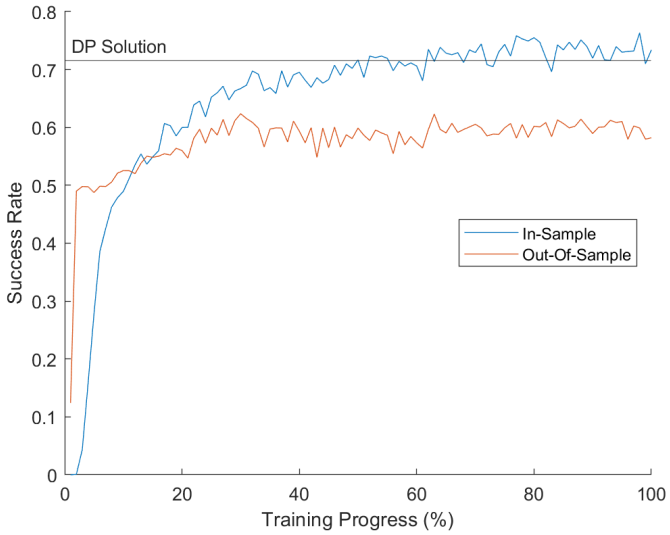


Figure 3.5: Sample IS and OOS success rate evolution for base Q-learning algorithm

Perhaps, further insight on the evolution of the In-Sample and Out-Of-Sample success rate throughout the training phase might help diagnose the problem. Figure 3.5 displays the IS and OOS success rates over the training phase of a single agent, which was trained with 100,000 episodes and an exploration rate $\epsilon = 0.3$. After each 1,000 epochs, the

agent's policy was evaluated with 10,000 OOS trajectories. Two main observations can be made from this figure. First, the In-Sample success rate, even towards the end of the training phase, is subject to a relatively high variance and is not totally centred around the DP solution. Second, the Out-Of-Sample success rate, despite initially increasing, stagnates quite early. Bizarrely, the improvements in the value function are not reflected in the policy. While the reason behind this discrepancy is still a mystery, it could potentially be fixed by the same adjustment.

While the In-Sample results of the Q-learning algorithm adequately approximate the optimal success rate, the Out-Of-Sample results, obtained using the procedure from section 3.2, proved that the algorithm did not learn the optimal policy and thus is not an appropriate alternative to dynamic programming. However, from the analyzed figures, it seems like this underwhelming performance is due to a lack of convergence rather than a fundamental problem with the model. The following section presents the results of the improved Q-learning algorithm, which was designed to offer better convergence.

3.5 Convergence Improvements

To improve the convergence of the Q-learning algorithm presented in Das and Varma (2020), another algorithm, which applies some small adjustments, was proposed in chapter 2. The main adjustment modifies the learning rate by decreasing it throughout the training phase via a decay function. Another possible adjustment consists of implementing a decaying exploration rate ϵ to gradually favour the exploitation of the agent's acquired knowledge throughout the training phase. However, since this modification did not significantly improve the results by its own, it was left out of the final model. This section presents the results of this algorithm and assesses if it truly offers an improvement on the base algorithm. Before analyzing its ability to replicate the optimal solution, the decay function needs to be calibrated.

3.5.1 Decay Function Calibration

As it was detailed in chapter 2, decay functions are problem specific and require a certain level of manual calibration. Thus, four decay functions were tested with multiple parameter combinations to find the one that best fits the studied problem. Specifically, the state-exponential, epoch-exponential, state-linear, and epoch-linear methods were trained with 1,000,000 episodes and tested Out-Of-Sample with 100,000 simulated wealth paths 30 times for each parameter combination. The parameter values tested were: $a_{\text{init}} \in \{1; 0.5; 0.1\}$ for the upper bound, $a_{\text{end}} \in \{0.01; 0.001; 0.0001;$ 1e-05; 1e-06$\}$ for the lower bound, and $b \in \{1; 0.75; 0.5\}$ for the decay speed. The tables presenting the calibration results, which are located in the appendix to avoid overcrowding the text, follow the same structure, where, for each parameter combination, the mean IS and OOS success rates are respectively presented on the first and second row.

Starting with the linear decay function for which the results are presented in table 1 for the state method and 2 for the epoch method. For the state method, the results clearly show that the linear decay function didn't improve the convergence of the algorithm. Even when the In-Sample success rate approaches the real optimal success rate, the Out-Of-Sample seems to be constrained at around 60%, which is comparable to the result obtained with a constant learning rate. The epoch method produces widely different results. For all the parameter combinations, the difference between the In-Sample and Out-Of-Sample success rates largely decreases. However, the optimal success rate was still not achieved as, even for the best parameter combination, the Out-of-Sample success rate didn't surpass 70%. The results of the exponential decay function are quite different. The epoch method (table 4) produces results that are at best comparable to the constant learning rate case, and at worst completely terrible. On the other hand, the state method (table 3) seems to yield IS and OOS results that approach the DP solution for all exploration rates. Even if the linear-epoch method produced on average better results for all the parameter combinations, none of them came as close to the optimal solution as the exponential-state method.

Overall, the state exponential decay function with parameters $a_{\text{init}} = 1$; $a_{\text{end}} = 1\text{e-}06$; $b = 0.75$ yields the best In-Sample and Out-Of-Sample results. This decay function will be used in the next section to replicate the optimal solution and assess if the proposed improved algorithm is able to correctly approximate the optimal solution.

3.5.2 Replication Results

With the improved Q-learning model now calibrated, it is time to provide further details on its ability to replicate the optimal solution. As it was previously done, the In-Sample results were obtained over multiple agents. Specifically, table 3.2 presents some descriptive metrics of the 30 agents trained over 1,000,000 episodes and tested on 100,000 simulated trajectories generated using the OOS procedure detailed in section 3.2 for the different exploration rates.

	In-Sample			Out-Of-Sample		
ϵ	Min	Mean	Max	Min	Mean	Max
0.10	69.87%	70.15%	70.58%	69.45%	70.04%	70.57%
0.20	70.67%	70.99%	71.22%	70.30%	70.74%	70.99%
0.30	71.08%	71.35%	71.63%	70.57%	70.92%	71.42%
0.40	71.28%	71.54%	71.73%	70.54%	70.94%	71.36%
0.50	71.51%	71.73%	71.88%	70.52%	70.86%	71.23%

Table 3.2: Improved Q-learning In-Sample and Out-Of-Sample replication results using a state exponential decaying learning rate, an initial wealth $W_0 = 100$, a terminal wealth goal $G = 200$, an investment horizon of $T = 10$ years, and 101 grid points.

The difference between these results and the ones presented in 3.1 are quite noticeable. Indeed, the In-Sample results of the improved algorithm almost completely eliminated the high variance produced by the base algorithm and are much closer to the optimal success rate of **71.59**%. This observation also translates to the Out-Of-Sample results, where the difference with the IS and DP success rates, as well as the overall variance, are much smaller. Furthermore, contrary to the base algorithm, both IS and OOS results are somewhat close to the DP results for all exploration rates.

Following those results, one could wonder whether the base algorithm was given a real chance to compete with the improved algorithm if the latter used 10 times more training episodes. In other words, did the decreasing learning rate really help the convergence, or was it only necessary to increase substantially the number of training episodes? The answer to this question is found in figure 3.6. The graph compares the success rate of both RL algorithms trained with 100K and 1M episodes, and tested 100 times throughout the training process with 10,000 simulated trajectories. It should be noted that, for the 100K case, the learning rate lower bound was adjusted to $a_{\text{end}} = 1\text{e-}06$ to better fit the training length. This adjustment does not affect the conclusions of this figure. Both algorithms use an exploration rate $e = 0.3$, which arguably provided the best replication results for both algorithms.

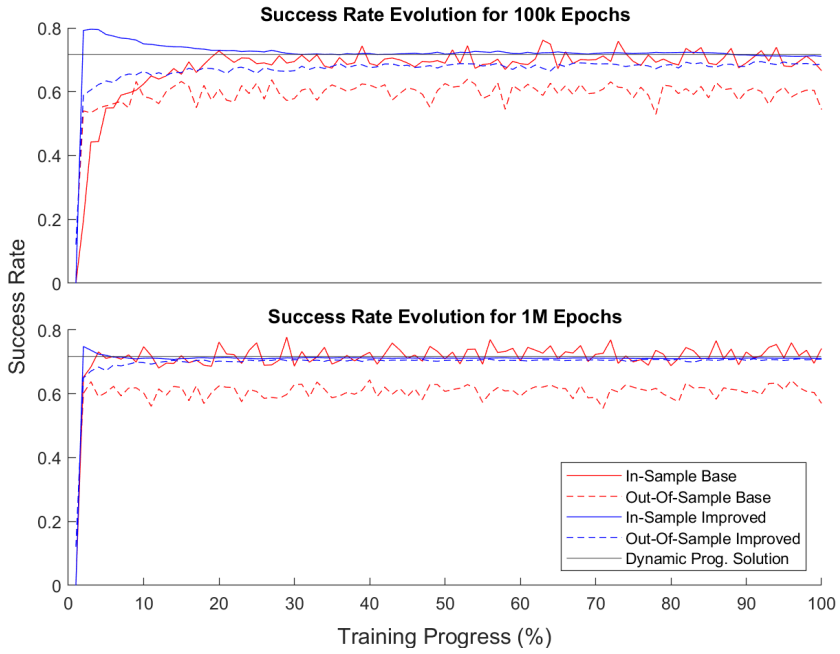


Figure 3.6: Base and improved Q-learning In-Sample and Out-Of-Sample success rate evolution comparison

Looking at the 100k case first, the improved algorithm's results, while inferior to the 1M case, are still way better than the base algorithm in terms of variance, In-Sample convergence, and Out-Of-Sample performance. Increasing the number of training episodes

to 1M did not improve the base algorithm's performance, as both IS and OOS success rates evolution follow the same pattern as before. Thus, while the increase in training episodes is necessary to reach a closer success rate, the key driver behind the convergence improvement is undoubtedly the decreasing learning rate. It is now possible to state that Q-learning is able to, at least, approximate an investor's likelihood of surpassing her terminal wealth goal.

3.6 Final Comparison

Before confirming that the Q-learning algorithm is able to fully reproduce the optimal solution, it is essential to compare the RL agent's optimal policy with DP's optimal solution and evaluate if it follows the same logic regarding dynamic risk-aversion changes. The optimal policy and its logical interpretation are as important as the optimal success rate, since, in practice, these elements of the optimal solution are what would dictate a goal-based investor's actions. The following results were obtained with one agent for each Q-learning algorithm. While the base Q-learning algorithm uses a constant learning rate $a = 0.1$ and 100K training episodes, the improved algorithm uses the calibrated exponential learning rate decay function and 1M epochs. Both algorithms use a constant exploration rate of $e = 0.3$. The In-Sample success rates for these parameters, presented in table 3.3, are close to the optimal success rate. While the base algorithm's success rate is lower, it is still somewhat similar to the DP solution.

	Dynamic Programming	Base Q-learning	Improved Q-learning
Mean	71.59%	70.29%	71.44%
Std Dev	-	1.93%	0.14%

Table 3.3: Dynamic programming and Q-learning In-Sample success rates comparison using an exploration rate $e = 0.3$, an initial wealth $W_0 = 100$, a terminal wealth goal $G = 200$, an investment horizon of $T = 10$ years, and 101 grid points.

Computation time wise, the Q-learning algorithm is definitively slower than dynamic programming, which is able to calculate the optimal solution is about 0.04 seconds using Matlab's C coder. On the other hand, the base Q-learning algorithms take about 3 seconds with 100K episodes and 30 seconds with 1M episodes. Similarly, the improved Q-learning algorithm, which is a bit slower, takes about 4 seconds with 100K episodes and 40 seconds with 1M episodes. Even if the Q-learning algorithm seem much slower than the benchmark algorithm, the runtime disadvantage could disappear as the state and action space is increases since, unlike dynamic programming, Q-learning is not subject to the curse of dimensionality

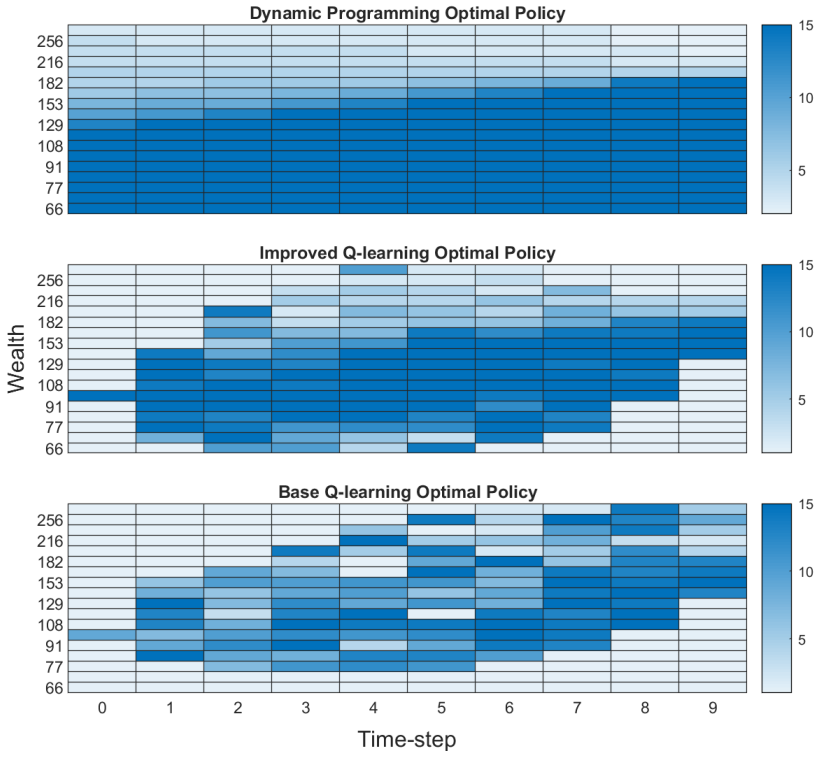


Figure 3.7: Dynamic programming and Q-learning optimal policies comparison

As it is shown in figure 3.7, which displays the optimal policy of all three algorithms for a subset of the state-space, the similarities between the three algorithms' IS success rates is not carried over to the optimal policies. Indeed, the base algorithm's optimal policy is surprisingly disorganized, creating significant differences with the benchmark.

Even if the policy of the base algorithm somewhat resembles DP's optimal policy towards the end of the investment horizon, the substantial randomness present in the previous time-steps considerably limits its interpretation. While this obviously sub-optimal policy can explain the low Out-Of-Sample results of the base algorithm, the cause behind the high In-Sample success rate is still a mystery.

In opposition, the improved algorithm is able to produce an optimal policy, which, at first sight, seems promising. While approximation errors are still noticeable, the overall structure of the policy closely resembles the optimal solution. Moreover, a clear interpretation can be made from this structure. Even if it is less flagrant than for DP's optimal policy, the increase in risk appetite over time for wealth values that are below the investment objective is visible. The investor's shift towards safer actions also follows approximately the same trend as the optimal solution. Practically speaking, the investment strategy proposed by the improved Q-learning algorithm is comparable to the one computed by the dynamic programming algorithm.

The improved algorithm's optimal policy is in no way perfect. However, it is important to remember that the objective of the algorithm is to produce an adequate approximation of the perfect solution. Even if some large sections of the heatmap (bottom right, top left) seem to display a discontinuity in the optimal policy, it does not affect the quality of the solution found by the algorithm. These irregularities are perfectly normal, as the corresponding states are unlikely to happen, which limited the number of visits by the agent. With more training episodes, the agent would be able to improve its estimate of the value function, which would increase even more the similarity to the benchmark.

Beyond the optimal policy, it might be interesting to compare the Out-Of-Sample performance of the three algorithms by using their respective optimal policies to generate trajectories, as described in section 3.2. Even though the results of each individual al-

gorithm were already discussed, it is still useful to compare them all at once. Table 3.4 presents the OOS success rates, obtained with 100,000 simulated trajectories, for the three principal algorithms, a risky policy, and a safe policy. Both the risky and safe policy operate in the same way by respectively selecting, at each time-step, the action with the highest and the lowest volatility from the action space. Including trivial policies provides a better appreciation of the optimal policy's value added for an investor.

Dynamic Prog.	Base QL	Improved QL	Safe Policy	Risky Policy
71.71%	59.98%	71.07%	12.10%	69.11%

Table 3.4: Dynamic programming and Q-learning Out-Of-Sample success rates comparison using an initial wealth $W_0 = 100$, a terminal wealth goal $G = 200$, an investment horizon of $T = 10$ years, and 101 grid points.

Without any surprise, the OOS results of the improved Q-learning algorithm are superior to all the other policies except the optimal solution. The difference between these two is small and acceptable as an approximation. In contrast, the base algorithm's underwhelming OOS performance is even lower than the trivial risky policy. While the risky policy seems to yield results that are somewhat close to the optimal solution, it is important to note that this policy will not systematically produce success rates that are comparable to the optimal solution.

Perhaps a deeper analysis of the OOS results could be made by assessing the terminal wealth distributions, which are presented in figure 3.8. The quality of the improved algorithm's approximation is again noticeable, as its distribution is almost identical to the optimal solution. Even if the risky policy produces a higher average terminal wealth than the improved algorithm's optimal policy, its high variance and lack of risk-aversion flexibility creates scenarios where the investor fails to reach her goal even though her wealth level was once above the objective. The particular shape of the optimal policy reflects well the risk-aversion shift included in the strategy. whilst the shape of the distribution is similar to the risky policy when the wealth is below the goal line, its shape, specifically its

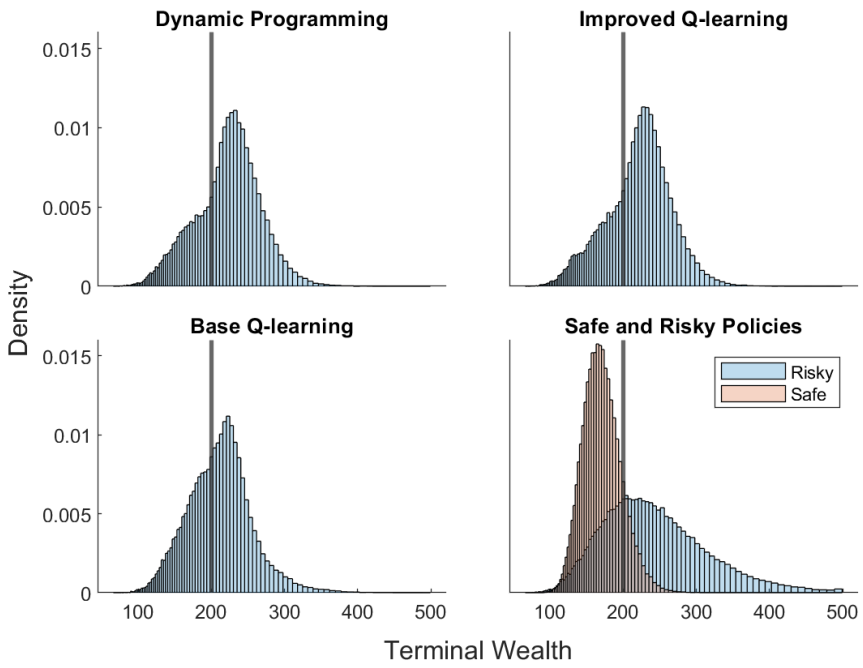


Figure 3.8: Dynamic programming and Q-learning terminal wealth distribution comparison

variance, matches the safe policy for wealth values that already surpass the objective. The same pattern can be partially observed in the base algorithm's distribution. This points back to its optimal policy, which is still exhibits signs of randomness.

From this analysis, it is fair to conclude that Q-learning, with an appropriate learning rate decay function, is fully capable of approximating the optimal solution of a dynamic goal-based wealth management problem modelled with a Markov decision process. This suggests that the RL algorithm could potentially solve similar problems with more complex and realistic constraints.

Chapter 4

Policy-Based Learning for Goal-Based Wealth Management

While the previous chapters focused on replicating the results from Das and Varma (2020) and improving the presented algorithm, this chapter introduces a policy gradient (PG) algorithm and attempts to solve the same dynamic goal-based wealth management problem, which has not been done yet in literature. In contrast with Q-learning, policy gradient methods are policy-based algorithms, meaning that they do not rely on the estimation of a value function to find the optimal policy. Instead, they directly estimate the policy as a parametric function, which can be advantageous for certain problems. Using a parametric policy function allows the investor to inject prior knowledge in the algorithm by specifying the appropriate form of the function. Moreover, for some environment, it might be the case where the policy function is simply easier to estimate than the value function, as it was shown in Şimşek et al. (2016) for Tetris.

Another key difference between the PG and the previously implemented Q-learning algorithms, is the use of a stochastic policy. In this chapter, the policy $p(a_t|W_t;q_t)$ is defined as the probability of selecting action a_t at time-step t given a state W_t and the policy function's parameters q_t. As long as the policy's gradient with respect to the parameters, $\tilde{N}_{q_t} p(a_t|W_t)$, exists, the investor can model the function as desired. Whereas value-based

policies can change significantly with a small variation in the value function, stochastic policies change smoothly as the parameters are updated. This produces stronger convergence guarantees, as mentioned in Sutton and Barto (2018). Using a stochastic policy removes the need to include an additional exploration parameter ϵ, since the random actions are already sampled from the policy. Also, selecting actions with arbitrary probabilities might be optimal for certain environment with imperfect information, such as bluffing in Poker.

Overall, the goal of PG algorithms is to maximize the expected reward by finding the optimal policy parameters. As it was the case for Q-learning and temporal difference methods, multiple policy gradient methods exist. However, only one was explored in this thesis, the REINFORCE algorithm from Sutton and Barto (2018). The following sections first detail the algorithm before testing it in a simplified environment. Lastly, an attempt to solve the complete GBWM environment from chapter 1 with this algorithm is made.

4.1 REINFORCE Algorithm

This section presents the REINFORCE algorithm. Unlike Q-learning, which uses the Bellman equation to update its action-value function, the policy gradient's algorithm uses a gradient ascent method to find the parameters θ that maximizes a policy's success probability (4.1). For a given trajectory t, the algorithm updates its parameters θ_t using the gradient of a performance measure $J(\theta_t)$, the expected value of the total reward of the trajectory $\mathbb{E}_{p_\theta}[R_t]$:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla} J(\theta_t): \tag{4.1}$$

Even with this definition of the performance measure, the evaluation of its gradient is still challenging. Luckily, using the policy gradient theorem (see Sutton and Barto (2018) for more details), equation (4.1) can be rewritten:

$$\theta_{t+1} = \theta_t + \alpha R_t \widehat{\nabla} \ln p(a|W; \theta_t): \tag{4.2}$$

While most policy gradient algorithms use this general equation, they most likely will have a different policy function $p(a|W;\theta)$. Here, a function $h(W;a;\theta)$, simply referred to as the preference function, is used, which returns a numerical preference for each state-action pair. In a given state, the actions with the higher preferences are assigned a higher selection probabilities than the actions with lower preferences. Using a preference function allows the algorithm to approach deterministic policies, which is not achievable by value-based algorithms that use ϵ-greedy action selection, such as Q-learning. A state-action pair's preference is converted into a probability measure using the softmax function:

$$p(a|W;\theta_t) = \frac{e^{h(W;a;\theta_t)}}{\sum_{b \in \mathbb{A}} e^{h(W;b;\theta_t)}}. \tag{4.3}$$

The specific form of $h(W;a;\theta)$ can be entirely customized to fit a given problem, ranging from simple linear combinations to deep neural networks. Selecting an appropriate form for $h(W;a;\theta)$ is not a trivial task. Using the knowledge acquired from the DP solution, it is possible to identify some key features that a given function should have in order to replicate the optimal strategies. When the wealth level is fixed, the function should be a concave quadratic with respect to the action. The parabolic shape ensures that only one maximum value, which can be located anywhere on the action space, is possible. Also, the apex of this curve should vary as the wealth level fluctuates. As the previous chapter showed, it is expected, from a goal-based wealth management perspective, to favour safe actions when the objective is already attained, and risky actions otherwise. Thus, as the wealth level approaches the set objective G, the apex of the quadratic action curve should move towards safer actions. While different functions which satisfy these characteristics were tested, only one will be presented:

$$\begin{aligned} h(W;a;\theta_t) &= \theta_t^{\top} x(W;a) \\ x(W;a) &= [1 \; a \; a^2 \; W \; W^2 \; W^3 \; aW \; aW^2 \; aW^3]. \end{aligned}$$

As it can be seen, this fairly simple function does not take into account the time-step. Instead, each time-step t uses a different set of parameters $\theta_{t;t}$ for each trajectory t, which

allows the preference function to be independent of the previous and subsequent time-steps. Neural networks could have been used to model the preference function. However, as it was done for the Q-learning algorithm, this research focuses on solutions that do not involve neural networks or other deep learning methods. Using this definition of the policy function $p(a|W;\theta)$ and some basic logarithmic and calculus rules, the gradient part of equation (4.2) can be developed as:

$$\begin{aligned}
\nabla_\theta \ln p(a|W;\theta_t) &= \nabla_\theta \ln \frac{e^{h(W;a;\theta_t)}}{\sum_{b\in\mathbb{A}} e^{h(W;b;\theta_t)}} \\
&= \nabla_\theta \ln e^{h(W;a;\theta_t)} \quad \nabla_\theta \ln \sum_{b\in\mathbb{A}} e^{h(W;b;\theta_t)} \\
&= \nabla_\theta h(W;a;\theta_t) \quad \frac{1}{\sum_{c\in\mathbb{A}} e^{h(W;c;\theta_t)}} \sum_{b\in\mathbb{A}} \nabla_\theta e^{h(W;b;\theta_t)} \\
&= \nabla_\theta h(W;a;\theta_t) \quad \frac{1}{\sum_{c\in\mathbb{A}} e^{h(W;c;\theta_t)}} \sum_{b\in\mathbb{A}} e^{h(W;b;\theta_t)} \nabla_\theta h(W;b;\theta_t) \\
&= \nabla_\theta h(W;a;\theta_t) \quad \sum_{b\in\mathbb{A}} \frac{e^{h(W;b;\theta_t)}}{\sum_{c\in\mathbb{A}} e^{h(W;c;\theta_t)}} \nabla_\theta h(W;b;\theta_t) \\
&= \nabla_\theta h(W;a;\theta_t) \quad \sum_{b\in\mathbb{A}} p(b|W;\theta_t) \nabla_\theta h(W;b;\theta_t)
\end{aligned} \tag{4.4}$$

Since the preference function $h(W;a;\theta_t)$ has been given a simple definition, it is possible to further develop equation (4.4) into a form that can be directly calculated:

$$\nabla_\theta \ln p(a|W;\theta_t) = x(W;a) \quad \sum_{b\in\mathbb{A}} p(b|W;\theta_t)\, x(W;b). \tag{4.5}$$

The REINFORCE algorithm, in contrast with Q-learning, goes through each simulated trajectory twice, as it requires the episode's cumulative final reward before it can update its parameters. Indeed, for each episode, the wealth trajectory is first simulated entirely using the current policy p. Then, after observing the episode's reward, the policy function's parameters θ_t are updated for each time-step with equations (4.2) and (4.5). This simple, yet effective algorithm is presented in a pseudo-code format below.

Result : Optimal policy p

for k 1 **to** $N_{episodes}$ **do**

  Generate trajectory $W_0; a_0; W_1; a_1 :::; W_T$ with $p(j; q_k)$;

  Calculate reward R_k $\mathbb{1}_{W_T \ G}$;

  for t 1 **to** T **do**

    $q_{k+1;t}$ $q_{k;t} + a \ R_k \left[x(W; a) \ \sum_{b2\mathbb{A}} p(bjW; q_t) \ x(W; b) \right]$;

  end

end

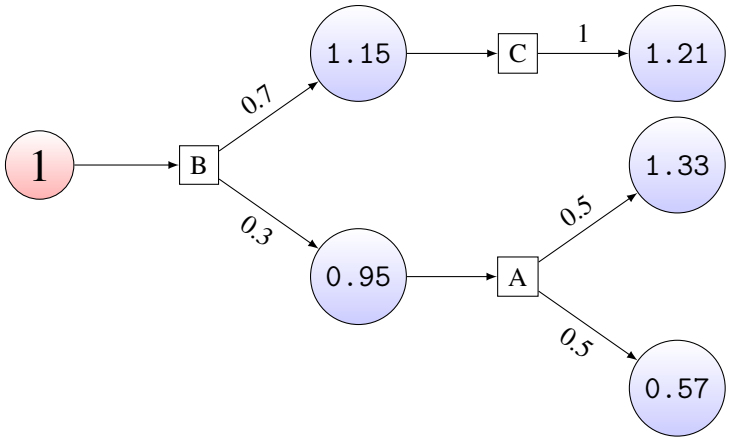
4.2 A Simple GBWM Case: Two Steps, Three Actions, and Discrete Returns

Before attempting to solve the GBWM problem, it is useful to get more familiar with the capabilities and limitations of the PG algorithm by implementing it in a simplified environment. Here, the investor starts with an initial wealth $W_0 = 1$ and has an objective of $G = 1{:}2$ with an investment horizon of $T = 2$. At each time-step, she can choose between two risky and one risk-free portfolio, as presented in table 4.1. To reduce the randomness of rewards and trajectories, the returns are discretized. Specifically, after choosing an action a_t, the investor faces two possible outcomes: either she receives return $R^1_{a_t}$ with probability $P^1_{a_t}$ or return $R^2_{a_t}$ with probability $P^2_{a_t}$. The new wealth is then calculated with the according realized return. For this simplified environment, the optimal policy and success rate were computed manually with a decision tree. The optimal policy, which yields a success rate of 85%, can be visualized in the decision tree below. The square nodes indicate the optimal actions taken for a given wealth level (round node). The transition probabilities associated with this action are indicated on the following line.

	$t = 0$		$t = 1$	
Action	Prob.	Return	Prob.	Return
A	0.5	25%	0.5	40%
	0.5	-10%	0.5	-40%
B	0.7	15%	0.5	25%
	0.3	-5%	0.5	-15%
C	1	5%	1	5%
	0	0%	0	0%

Table 4.1: Discrete actions returns and probabilities

To calculate the probability of ending in one of the three end nodes, simply multiply the probabilities of that particular trajectory. For example, the likelihood of being worth 1.33 at the end of the investment horizon is 0.3 0.5 = 15%. Similarly, the success probability is computed by adding the probabilities of the paths where the terminal wealth greater than the objective: 0.7 1 + 0.3 0.5 = 85%. The optimal policy of this environment is fairly simple. After choosing action B first, the investor should select the risk-free action C if she receives a positive return. Otherwise, she should choose the riskiest action A because it's the only one that would allow her to reach the objective.



Solving this simplified environment with the PG algorithm requires first the definition of three hyperparameters: the number of training episodes, the learning rate α, and the initial policy function parameters θ_0. These parameters significantly affect the perfor-

mance of the algorithm. Good results were obtained by training an agent with two million episodes, a learning rate $\alpha = 0.01$, and the starting preference function (where the first and second rows are associated with $t = 0$ and $t = 1$ respectively):

$$h(W; a; \theta_0) = \begin{bmatrix} 0 + 10a + 0a^2 + 0W & 10W^2 + 0W^3 & 20aW + 20aW^2 + 10aW^3 \\ 0 & 10a + 0a^2 + 0W + 0W^2 + 0W^3 + 10aW + 0aW^2 + 0aW^3 \end{bmatrix}.$$

Unfortunately, finding a good starting point θ_0 was not straight forward. In a perfect world, the algorithm would be able to reproduce the same results no matter the starting point. However, most optimization algorithms require a somewhat decent initial value. Finding the most general point for which the optimal solution is reached is no trivial task, especially for the implemented algorithm. Some key characteristics of a good policy function $\pi(a|W; \theta)$ can be extracted from the optimal decision tree previously presented. Specifically, the policy should favour the safest action C when the investor's wealth is above 1 and the riskiest action A when the wealth is below 1. Additionally, action B should be prioritized for the initial wealth $W_0 = 1$. With these characteristics in mind, a simple and tedious trial-and-error method was used to find a set of parameters θ_0 that produced an adequate policy curve $\pi(a|W; \theta_0)$. The quality of a given starting point θ_0 was assessed visually by examining the shape of the policy curve produced, and numerically by comparing the success rate achieved by a PG agent trained using this starting policy function with the success rate obtained by the optimal decision tree.

While the first row of figure 4.1 presents the policy function $\pi(a|W; \theta)$ found by the PG agent after its training phase, the second row shows the policy function produced by the initial parameters θ_0. Even if the difference between the initial and final policy curves for both time-steps is significant, especially for $t = 0$, the general shape is similar. It can be clearly seen that the agent was able to learn a more optimal solution, however, the extent to which the algorithm is able to approximate the benchmark policy greatly depends on the starting parameters θ_0. Further exploration would be needed to determine if the required starting point is too close to the optimal value to be considered for practical applications.

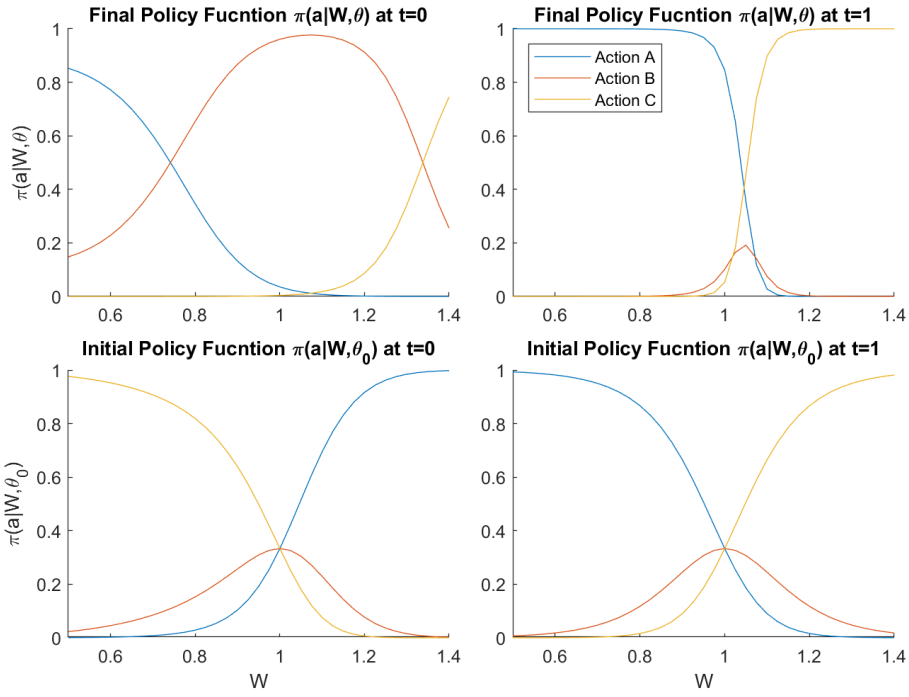


Figure 4.1: Policy curves for starting parameters q_0

Using these starting parameters q_0, the algorithm is able to find in about 40 seconds a policy which yields a success rate of 84.79%, which is close to the optimal success rate of 85%. The results of this optimization are presented in figure 4.2. The first panel presents the evolution of the success rate throughout the training phase. In contrast with the previous algorithms, the In-Sample success rate is not directly reported by PG. Instead, the policy is tested with 100,000 Out-Of-Sample simulated trajectories 100 times during the training phase. As it can be seen, the algorithm is able to find a "good enough" solution, which is presented in the second panel in the form of a heatmap which uses the expected actions (simple sum of the actions weighted by their respective probability where portfolios A, B, and C are respectively represented by 1,2,3). From this heatmap, it can observe that the policy follows well the optimal investment strategy, as it favours the same actions as the decision tree showed for the important breakpoints (0.95, 1, and 1.15). Specifically, the agent chooses action B as its initial action and favours action C if it reaches 1.15 and action A if it drops to 0.95 at $t = 1$. Thus, it's fair to say that the

algorithm successfully approximated the optimal solution of this environment.

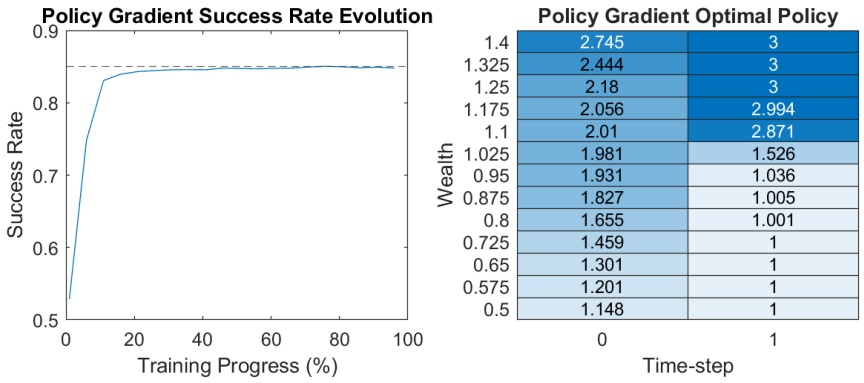


Figure 4.2: Policy Gradient simplified environment results

Although the results were promising, it is important to remember that they were obtained with good hyperparameters. No rule of thumb exists for a "good" set of hyperparameters, thus, multiple combinations of the number of episodes and the learning rate were tested. Table 4.2 presents, for each combination, the mean of the 10 Out-Of-Sample success rate calculated with 100,000 simulated trajectories each. The initial policy function parameters q_0 are kept the same. Considering the number of training episodes and the ability to generate a success rate comparable to the optimal solution of 85%, the best learning rate seems to be $a = 0{:}01$. However, a higher success rate could potentially be achieved with a lower learning rate, if the investor is willing to spend more time training the model.

This simplified environment helped unmask some key characteristics of the algorithm, such as its ability to model solve a discrete environment and the importance of the starting parameters. The need for a good starting point is definitively a limitation to keep in mind when assessing whether the algorithm could be reliably used as an approximation. However, since the algorithm was able to successfully approximate the optimal solution of the simplified problem, the next sections will attempt to solve progressively more complex environments, ending with the complete GBWM problem.

	α				
Episodes	0.1	0.01	0.001	0.0001	0.00001
100K	0.500	0.750	0.710	0.666	0.643
500K	0.499	0.796	0.742	0.693	0.660
1M	0.500	0.822	0.784	0.710	0.666
1.5M	0.499	0.828	0.822	0.720	0.671
2M	0.499	0.838	0.833	0.725	0.675
2.5M	0.501	0.840	0.839	0.730	0.679

Table 4.2: Policy Gradient parameter exploration for simplified environment

4.3 Complete GBWM Environment with Continuous Returns

To reduce the complexity gap between the simplified environment and the complete goal-based wealth management problem from the previous chapter, the returns go back to following a normal distribution, which, combined with a continuous state space, yields a limitless number of possible outcomes. Before assessing the algorithm's ability to approximate the benchmark solution for the complete GBWM problem, the impact of increasing the investment horizon and the number of available portfolios is evaluated.

4.3.1 Increasing the Investment Horizon

First, only the three portfolios presented below are considered, which facilitates the analysis of the effect of the number of training episodes and the investment horizon on the PG agent's success rate. Each action's distribution parameters $(\mu;\sigma)$ are the same for each time-step t. The mean Out-Of-Sample success rates of the policy gradient algorithm for different investment horizons (associated with an arbitrary wealth goal) and number of training episodes are presented in table 4.3. These results were obtained with a constant learning rate $\alpha = 0.001$ and 10 different OOS simulations of 100,000 trajectories. Since the policy function uses an exponential policy function $\pi(a|W;\theta)$, it is important to keep the values of the preference function $h(W;a;\theta)$ below a certain threshold, otherwise

NaN values are generated (numerically, e^x produces NaN values when x is approximately greater than 700). To do so, the wealth values are bounded between 0 and 3 with a starting value W_0 of 1 instead of 100. This wealth range allows the algorithm to explore the wealth values that will have a large impact on the agent's optimal policy and success rate, while avoiding larger values which could create a numerical error. Beyond the maximal wealth value of 3, it is assumed that the policy is the same.

	Portfolios		
	A	B	C
m	15%	10%	5%
s	30%	15%	7%

As it was mentioned in the previous section, the starting point has a large impact on the agent's performance. Here, the starting base function $h(a;W;q_0)$ used is:

$$h(W;a;q_0) = 0 \quad 100a \quad a^2 + 0W + 0W^2 + 0W^3 + 100aW \quad aW^2 \quad aW^3$$

which produces the policy curve $p(ajW;q_0)$ presented in the bottom left panel of figure 4.3. These parameters are different from the one used in the previous section simply because they produced better results, without being too close to the optimal solution. Since this starting point yielded good results for environments with a short investment horizons and 3 actions with continuous returns, it is plausible to suppose that its shape is somewhat appropriate. From table 4.3, it can be seen that, for short investment horizons, the algorithm is able to produce a policy that closely matches the benchmark's success rate obtained with the dynamic programming algorithm detailed in chapter 2 (reported under the DP column). However, as the investment horizon increases, the success rate difference becomes larger. Even with 10 million training episodes, which takes about 8 minutes to compute when $T = 10$, the difference between the benchmark and PG solution is greater than 1% when the investment horizon is greater than 5, reaching 3.2% for the case $T = 10$. This suggests that, even when only 3 actions are considered, the algorithm would need even more than 10 million epochs to converge. Although the results produced with $T = 2$ indicate that the agent is indeed able to find the good policy, whether or not

it would be able to replicate the optimal solution of more complex environments is still unknown.

		Policy Gradient			
T	DP	1M	2M	5M	10M
2	0.5900	0.5873	0.5880	0.5870	0.5862
5	0.6835	0.6733	0.6725	0.6744	0.6754
7	0.7248	0.6889	0.7074	0.7063	0.7117
10	0.7806	0.6585	0.7014	0.7349	0.749

Table 4.3: Policy Gradient performance for different investment horizons

Perhaps a better assessment of the PG agent's convergence when 10 million epochs are used could be done with the results presented in figure 4.3. As shown in the top left panel, which compares the relative success rate evolution (PG agent success divided by the DP benchmark), the policy doesn't significantly improve towards the end of the training phase when $T = 10$. For each investment horizon, the agent seems to reach a performance plateau at some point during training. This plateau is close to the DP success rate for short investment horizon and rapidly decreases with longer investment horizons. It could be possible though that, even if the success rate is not particularly close to the benchmark, the policy could follow a similar logic as the benchmark's optimal policy, where the investor's risk appetite increases over time if her wealth is not above her goal. This would indicate that the small policy approximation errors made at each time-step have a compounded effect on longer investment horizons.

The PG agent's optimal policy (bottom-right panel) can be assessed by comparing it to the benchmark policy in the top-right panel of figure 4.3. While a slight tendency towards the correct policy can be seen, the overall result is fairly disappointing. Even after 10 million episodes, a lot of noise is still present in the PG solution. The lack of consistency across the time-steps is definitively concerning. The risk appetite of the PG agent seems to arbitrarily change from one period to the other, offering no clear investment logic.

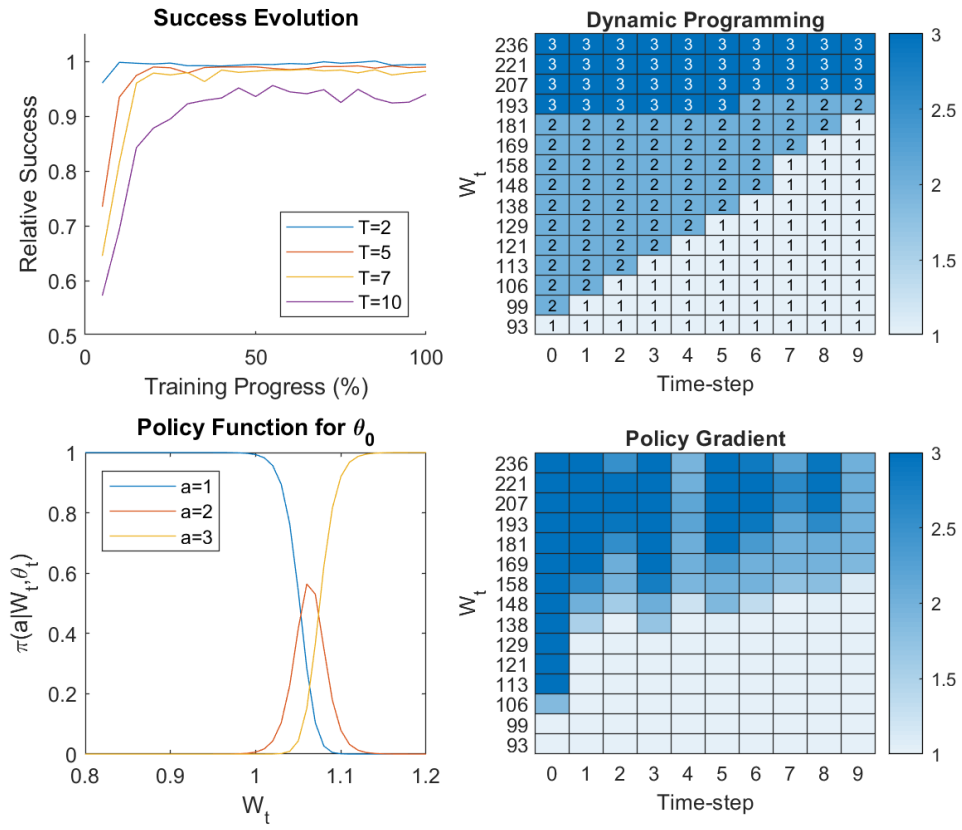


Figure 4.3: Policy Gradient results for 3 actions with continuous returns

Thus, it is fair to state that the agent was not able to replicate the optimal solution in this environment with 3 continuous actions and an investment horizon of 10 years.

4.3.2 Increasing the Number of Actions

The last step before reaching the complete goal-based wealth management problem's complexity is to increase the number of actions. From the previous mixed result, it is reasonable to expect the PG agent to have even more difficulty achieving the optimal solution, especially for longer investment horizons. Applying the same policy curve $p(ajW;q_0)$ based on the initial preference function $h(W;a;q_0)$ as presented in the previous section to cases with more actions is not as straightforward as it might seem. Indeed, if the policy function uses the same starting parameters with more actions, NaN will inevitably be generated for the same reasons previously discussed. To avoid this, a proper

set of parameters, which produces an adequate shape and generates valid numbers for the considered state space, is necessary. Since this task can be long and painful, the actions are normalized in order to use the same starting parameters without encountering a divergent gradient. This is done by dividing the possible actions (number from 1 to N) by the total number of actions N. Because the previously used starting point works well for cases with 3 actions, the normalized actions are multiplied by three $a = 3\frac{1}{N}; 3\frac{2}{N}; ...; 3$.

The impact of an increase in the number of available actions and the length of the investment horizon is presented in table 4.4. This table displays the difference between the benchmark success rate and the one obtained by the PG agent after 10 million training episodes. For example, with 15 actions and $T = 10$, the DP and PG success rates are, respectively, 71.59% and 47.36%, which corresponds to a 24.23% difference. The actions used are sampled from the portfolios presented in chapter one. The sampling is done by selecting N portfolios equally spaced across the specified action space. The runtime for the Policy Gradient algorithm with a ten-year investment horizon and 15 available actions trained over 10 million episodes is about 474 seconds (roughly 8 minutes). This is quite significantly slower than both the Q-learning and dynamic programming algorithm.

	Actions			
T	3	5	10	15
2	0.0052	0.0075	0.0085	0.0084
5	0.0077	0.0115	0.0137	0.0135
7	0.0574	0.1127	0.1151	0.1168
10	0.1809	0.2401	0.2415	0.2423

Table 4.4: Difference between Dynamic Programming and Policy Gradient success rates for different investment horizons and number of actions with continuous returns

The negative impact of using additional actions on the agent's performance can be observed across the different investment horizons. While this impact is not concerning for short maturities, it does significantly reduce the ability of the agent to adequately approximate the solution when $T > 5$. Even with only 5 actions, the results are noticeably worst.

At this point, it might be surprising to observe that the performance for longer investment horizons when three actions are considered is definitively lower than in table 4.3. This can be explained by the lack of difference between each portfolio's characteristics (m,s). Because the actions yield similar outcomes, it is harder for the agent to determine which action is the best. The impact of this similitude was certainly more significant than expected.

Unfortunately, considering the previously discussed shortfalls of the proposed PG algorithm, it is only logical to conclude that the agent is not able to solve the goal-based wealth management problem as presented in chapter one. Even if the algorithm was able to solve a simplified environment with shorter investment horizons, the complexity of the complete environment seemed to be too much for the current specifications. It is not impossible that with enough training episodes, the optimal solution could be reached. However, the large gap with the benchmark still observable with 10 million epochs indicates that the number of training needed to reach an acceptable solution is, practically speaking, a great limitation compared to the alternative algorithms (DP and Q-learning) with proven success.

It is important to note that the results presented in this chapter were obtained without the use of a baseline in the algorithm, even though baselines are often implemented as a way to reduce the variance of the estimators. This feature was ultimately not included in the final model because early tests did not show any improvements. However, a much more extensive exploration of the baseline's potential impact would be necessary before concluding definitively if it can help improve the convergence in a GBWM environment.

Although it ultimately failed, the exploration of a Policy Gradient algorithm to solve the GBWM problem was not in vain. Indeed, it was showed that the presented base function is able to handle short investment horizons, hinting that the algorithm, under the right specifications, would be able to accomplish the same for problems with longer maturities.

Finding the good form of the base function and an adequate set of starting parameters proved to be extremely challenging and required an enormous amount of trial and error. Thus, in an effort to improve the performance of the PG algorithm, it would be wiser to focus further research on the usage of deep neural networks since they would not require the investor to provide a base function with a specific form.

Chapter 5

Conclusion

This research presents the application of reinforcement learning algorithms to solve a dynamic goal-based wealth management problem. In contrast with traditional portfolio management, which maximizes the risk-adjusted return, the objective of a goal-based investor is to maximize the probability of surpassing her terminal wealth goal. The initial objective of this research was to implement the Q-learning algorithm presented in Das and Varma (2020), which was specifically tailored for the goal-based wealth management. Following the replication and analysis of its results, two contributions were proposed.

First, an Out-Of-Sample procedure was developed to test the reinforcement learning agent's true performance. This led to the realization that the presented Q-learning algorithm did not correctly approximate the benchmark solution, as its Out-Of-Sample performance was largely inferior. The second contribution, which proposes and implements an improvement to the base algorithm, specifically fixes this issue. It was shown that using a decreasing learning rate greatly facilitates the Q-learning algorithm's convergence, which also drastically improves its overall performance. With these contributions, it was concluded that the benchmark solution can be adequately approximated by the value based reinforcement learning algorithm.

Second, a policy based reinforcement learning algorithm, which directly estimates the policy function instead of approximating a value function first, was also explored. It was shown that a simplified environment with less possible actions and shorter investment horizons can be solved using this alternative algorithm. Unfortunately, the complete goal-based wealth management problem, as solved by the Q-learning algorithm, proved to be too complex for the specification of the presented Policy Gradient algorithm. To improve the performance, further research regarding this subject should focus on the implementation of a deep neural network as a base function, as it could potentially give the agent enough flexibility to find the right policy function without requiring the investor to provide its shape.

The positive results obtained in this thesis open the door for the addition of more realistic constraints to the goal-based wealth management problem, such as transactions costs, which are not trivial to implement with dynamic programming due to its backward induction nature. Furthermore, while the Q-learning algorithm does not need deep neural networks to converge, their usage could enable continuous state and action spaces. Because dynamic programming algorithms are not able to handle such environments, the successful implementation of a deep Q-learning algorithm would provide a clear advantage for practical applications.

Bibliography

Almahdi, S. and Yang, S. (2017). An adaptive portfolio trading system: A risk-return portfolio optimization using recurrent reinforcement learning with expected maximum drawdown. *Expert Systems with Applications*, 87:267–279.

Bellman, R. (1952). On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719.

Bellman, R. (2003). *Dynamic Programming*. Dover Publications, Inc., New York, NY, USA.

Bellman, R. and Dreyfus, S. (2015). *Applied Dynamic Programming*. Princeton University Press.

Bellmare, M., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. Paper presented at 34th International Conference on Machine Learning, Sydney, Australia, Aug.7, Vol.70, pp 449-458.

Bertoluzzo, F. and Corazza, M. (2012). Testing different reinforcement learning configurations for financial trading: Introduction and application. *Procedia Economics and Finance*, 3:68–77.

Bossaerts, P., Huang, S., and Yadav, N. (2020). Exploiting distributional temporal difference learning to deal with tail risk. *Risks*, 8(113).

Brandt, M. W., Goyal, A., Santa-Clara, P., and Stroud, J. R. (2005). A simulation approach to dynamic portfolio choice with an application to learning about return predictability. *The Review of Financial Studies*, 18(3).

Brunel, J. (2015). *Goal-Based Wealth Management: An Integrated and Practical Approach to Changing the Structure of Wealth Advisory Practices*. John Wiley & Sons, Ltd.

Buehler, H., l. Gonon, Teichmann, J., and Wood, B. (2019). Deep hedging. *Quantitative Finance*, 19:8:1271–1291.

Campbell, J. and Viceira, L. (1999). Consumption and portfolio decisions when returns are time varying. *Quarterly Journal of Economics*, 118:1449–1494.

Cao, J., Chen, J., Hull, J., and Poulos, Z. (2021). Deep hedging of derivatives using reinforcement learning. *Journal of Financial Data Science*, 3(1):10–27.

Casqueiro, P. and Rodrigues, A. (2006). Neuro-dynamic trading methods. *European Journal of Operational Research*, 175:1400–1412.

Chhabra, A. B. (2005). Beyond markowitz: A comprehensive wealth allocation framework for individual investors. *The Journal of Wealth Management*, 7(4):8–34.

Coqueret, G. and Guida, T. (2018). *Machine Learning for Factor Investing: R Version*. Chapman and Hall/CRC Financial Mathematics Series. CRC Press.

Das, S., Markowitz, H., Scheid, H., and Statman, M. (2010). Portfolio optimization with mental accounts. *Journal of Financial and Quantitative Analysis*, 45(2):311–344.

Das, S., Ostrov, D., Radhakrishnan, A., and Srivastav, D. (2018). Goals-based wealth management: A new approach. *Journal of Investment Management*, 16(3):1–27.

Das, S., Ostrov, D., Radhakrishnan, A., and Srivastav, D. (2020). A dynamic approach to goals-based wealth management. *Computational Management Science*, 17:613–640.

Das, S. and Varma, S. (2020). Dynamic goals-based wealth management using reinforcement learning. *Journal of Investment Management*, 18(2):37–56.

de Prado, M. (2020). *Machine Learning for Asset Managers*. Elements in Quantitative Finance. Cambridge University Press.

Dempster, M. and Leemans, V. (2006). An automated fx trading system using adaptive reinforcement learning. *Expert Systems with Applications*, 30:543–552.

Denault, M., Delage, E., and Simonato, J.-G. (2017). Dynamic portfolio choice: a simulation-and-regression approach. *Optimization and Engineering*, 18.

Denault, M. and Simonato, J.-G. (2021). A note on a dynamic goal-based wealth management problem. *Finance Research Letters*.

Deng, Y., Bao, F., Kong, Y., Ren, Z., and Dai, Q. (2016). Deep direct reinforcement learning for financial signal representation. *IEEE Transactions on Neural Networks and Learning Systems*, 28:1–12.

Dixon, M. and Halperin, I. (2020). G-learner and girl: Goal based wealth management with reinforcement learning. *Wealth Management eJournal*.

Dixon, M., Halperin, I., and Bilokon, P. (2020). *Machine Learning in Finance: From Theory to Practice*. Springer Nature Switzerland AG, Cham, Switzerland.

Garlappi, L. and Skoulakis, G. (2010). Solving consumption and portfolio choice problems: the state variable decomposition method. *The Review of Financial Studies*, 23:3346–3400.

Gold, C. (2003). FX trading via recurrent reinforcement learning. *IEEE/IAFE Conference on Computational Intelligence for Financial Engineering, Proceedings*, 2003-January:363–370.

Guida, T. (2018). *Big Data and Machine Learning in Quantitative Investment*. John Wiley & Sons, Ltd.

Hens, T. and Wöhrmann, P. (2007). Strategic asset allocation and market timing: A reinforcement learning approach. *Computational Economics*, 29:369–381.

Jeong, G. and Kim, H. (2019). Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 117:125–138.

Jiang, Z., Xu, D., and Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem.

Kahneman, D. and Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica*, 47:263–291.

Kolm, P. and Ritter, G. (2019). Modern perspectives on reinforcement learning in finance. *SSRN Electronic Journal*.

Maringer, D. and Ramtohul, T. (2012). Regime-switching recurrent reinforcement learning for investment decision making. *Computational Management Science*, 9:89–107.

Markowitz, H. (1952). Portfolio selection. *Journal of Finance*, 6(1):77–91.

Moody, J. and Saffell, M. (2001). Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12:875–889.

Moody, J., Wu, L., Liao, Y., and Saffell, M. (1998). Performance functions and reinforcement learning for trading systems and portfolios. *Science*, 17:441–470.

Neuneier, R. (1996). Optimal asset allocation using adaptive dynamic programming. *Advances in Neural Information Processing Systems*, 8.

Neuneier, R. (1998). Enhancing q-learning for optimal asset allocation. *Advances in neural information processing systems*, pages 936–942.

Park, H., Sim, M., and Choi, D. (2020). An intelligent financial portfolio trading strategy using deep q-learning. *Expert Systems with Applications*, 158.

Pendharkar, P. and Cusatis, P. (2018). Trading financial indices with reinforcement learning agents. *Expert Systems with Applications*, 103:1–13.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statistics*, 22:400–407.

Shefrin, H. and Statman, M. (2000). Behavioural portfolio theory. *Journal of Quantitative Analysis*, 35:127–151.

Spall, J. C. (2003). *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons, Inc., Hoboken, New Jersey.

Sutton, R. and Barto, A. (2018). *Reinforcement learning: An introduction*. The MIT Press, Cambridge, Massachusetts, 2 edition.

Thaler, R. (1985). Mental accounting and consumer choice. *Marketing Science*, 4:199–214.

Thaler, R. (1999). Mental accounting matters. *Journal of Behavioral Decision Making*, 12:183–206.

Van Binsbergen, J. and Brandt, M. (2007). Solving dynamic portfolio choice problems by recursing on optimized portfolio weights or on the value function? *Comput Econ*, 29:355–367.

Watkins, C. (1989). *Learning from delayed rewards*. PhD thesis, Kings College, Cambridge, England.

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:179–192.

Şimşek, O., Algorta, S., and Kothiyal, A. (2016). Why most decisions are easy in tetris—and perhaps in other sequential decision problems, as well. *Proceedings of the 33rd International Conference on Machine Learning (ICML 2016)*, pages 1757–1765.

Appendix A – Tables

		ϵ				
α_{init}	α_{end}	0.1	0.2	0.3	0.4	0.5
1	0.01	0.9676	0.9996	1	1	1
		0.5278	0.5231	0.5173	0.5176	0.5262
	0.001	0.9665	0.9997	1	1	1
		0.5226	0.5237	0.5305	0.531	0.5192
	0.0001	0.9911	0.9998	1	1	1
		0.5284	0.5242	0.5266	0.5183	0.5191
	1e-05	0.9872	0.9998	1	1	1
		0.5208	0.5189	0.5187	0.5203	0.5188
	1e-06	0.9908	0.9997	1	1	1
		0.526	0.5212	0.5219	0.52	0.5209
0.5	0.01	0.6637	0.7766	0.86	0.9197	0.9544
		0.528	0.5329	0.5226	0.5322	0.5319
	0.001	0.6754	0.7799	0.8703	0.9174	0.9552
		0.5193	0.5248	0.5284	0.5309	0.5382
	0.0001	0.6763	0.7884	0.8732	0.9253	0.959
		0.5181	0.527	0.5304	0.527	0.5368
	1e-05	0.6769	0.7962	0.8585	0.9166	0.9556
		0.525	0.5297	0.5357	0.5303	0.5427
	1e-06	0.6807	0.79	0.8622	0.9234	0.9522
		0.5281	0.5259	0.5278	0.536	0.5351
0.1	0.01	0.6508	0.6911	0.7071	0.7392	0.7753
		0.6056	0.6009	0.5996	0.6063	0.609
	0.001	0.6467	0.6856	0.7195	0.7475	0.7811
		0.5977	0.6056	0.6032	0.6072	0.6098
	0.0001	0.6513	0.6887	0.7203	0.7437	0.7717
		0.6017	0.6079	0.6084	0.6028	0.6131
	1e-05	0.6488	0.679	0.7126	0.7439	0.7729
		0.6036	0.6006	0.6084	0.6112	0.6106
	1e-06	0.6618	0.6819	0.7214	0.7486	0.7737
		0.6026	0.6045	0.6047	0.6081	0.6107

Table 1: Mean In-Sample (first rows) and mean Out-Of-Sample (second rows) success rates of the improved Q-learning algorithm using the state linear decay function with different parameter combinations. See sections 2.2.2 and 3.5.1 for more details.

		ϵ				
α_{init}	α_{end}	0.1	0.2	0.3	0.4	0.5
1	0.01	0.6625	0.6801	0.6964	0.7115	0.7267
		0.6661	0.6714	0.6719	0.6732	0.6716
	0.001	0.6653	0.6812	0.6941	0.7098	0.7253
		0.6656	0.6765	0.6793	0.6798	0.681
	0.0001	0.6659	0.6798	0.6939	0.7103	0.7245
		0.6667	0.6759	0.679	0.6822	0.6806
	1e-05	0.6656	0.6826	0.6949	0.7094	0.7252
		0.667	0.6769	0.6797	0.6804	0.6805
	1e-06	0.6675	0.6799	0.6938	0.7088	0.7236
		0.6689	0.6754	0.6792	0.6817	0.6802
0.5	0.01	0.674	0.6841	0.6955	0.7087	0.7224
		0.6716	0.675	0.6755	0.6776	0.6779
	0.001	0.6722	0.6847	0.697	0.7063	0.7178
		0.6736	0.6822	0.687	0.6867	0.6868
	0.0001	0.6741	0.6867	0.6952	0.7063	0.7178
		0.6755	0.6846	0.6847	0.6874	0.6874
	1e-05	0.6752	0.6862	0.6947	0.7069	0.7192
		0.675	0.6835	0.6851	0.6887	0.6875
	1e-06	0.6749	0.6857	0.6955	0.7059	0.7181
		0.6751	0.6829	0.6861	0.6875	0.6878
0.1	0.01	0.6863	0.6923	0.6999	0.706	0.7162
		0.6828	0.6828	0.6826	0.6815	0.6823
	0.001	0.6882	0.6946	0.7004	0.7062	0.7115
		0.6889	0.6957	0.6975	0.6973	0.6977
	0.0001	0.6885	0.6953	0.7	0.7053	0.7114
		0.689	0.6953	0.6973	0.6975	0.6982
	1e-05	0.6888	0.6946	0.7	0.7047	0.7115
		0.6896	0.6952	0.6972	0.6983	0.6984
	1e-06	0.6877	0.6938	0.6998	0.7058	0.7112
		0.689	0.6947	0.6964	0.6984	0.6984

Table 2: Mean In-Sample (first rows) and mean Out-Of-Sample (second rows) success rates of the improved Q-learning algorithm using the epoch linear decay function with different parameter combinations. See sections 2.2.2 and 3.5.1 for more details.

		e = 0:1			e = 0:2			e = 0:3			e = 0:4			e = 0:5		
a_{init}	a_{end}	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5
1	0.01	0.7669	0.9181	1	0.9022	0.9987	1	0.9583	1	1	0.9866	1	1	0.9962	1	1
		0.5292	0.5261	0.5192	0.5247	0.5153	0.5272	0.5282	0.527	0.5132	0.5381	0.5243	0.4946	0.5226	0.5235	0.4395
	0.001	0.6633	0.8077	0.9952	0.7299	0.9296	0.9998	0.7826	0.9748	1	0.8233	0.993	1	0.8601	0.9986	1
		0.5846	0.5395	0.5296	0.591	0.5333	0.5242	0.5833	0.5298	0.5272	0.5845	0.523	0.5213	0.5788	0.5201	0.515
	0.0001	0.6529	0.6677	0.839	0.7062	0.7216	0.952	0.7208	0.7706	0.9894	0.7305	0.8082	0.9979	0.7414	0.8454	0.9997
		0.6317	0.5933	0.5328	0.675	0.5916	0.5246	0.6759	0.5885	0.5203	0.6719	0.5813	0.5242	0.6682	0.5851	0.5219
	1e-05	0.6925	0.6779	0.6666	0.7074	0.7087	0.712	0.7145	0.7179	0.7492	0.7192	0.7262	0.787	0.7224	0.7342	0.8191
		0.6916	0.6632	0.5964	0.7027	0.6847	0.5953	0.707	0.683	0.5909	0.7087	0.6794	0.5948	0.7083	0.6754	0.5931
	1e-06	0.7177	0.7036	0.701	0.751	0.7119	0.7103	0.7755	0.7156	0.7152	0.7997	0.7182	0.7214	0.8197	0.7204	0.7272
		0.7052	0.7025	0.69	0.7011	0.7088	0.6899	0.6962	0.7103	0.6863	0.691	0.7108	0.6848	0.6823	0.7101	0.6817
0.5	0.01	0.6731	0.6923	0.694	0.7713	0.7802	0.823	0.8344	0.855	0.874	0.8858	0.9222	0.9214	0.9295	0.956	0.9571
		0.5376	0.5232	0.5225	0.5388	0.5277	0.5329	0.5431	0.5295	0.5328	0.541	0.5321	0.5412	0.5381	0.5336	0.5362
	0.001	0.6661	0.6683	0.6763	0.7192	0.7658	0.7815	0.7591	0.8383	0.878	0.7955	0.896	0.9242	0.8318	0.9382	0.959
		0.5992	0.5347	0.5241	0.5952	0.5385	0.5231	0.5925	0.5304	0.535	0.5966	0.543	0.5249	0.5879	0.535	0.5388
	0.0001	0.6717	0.6688	0.6773	0.7102	0.7129	0.7787	0.7211	0.7572	0.8649	0.7293	0.7919	0.9061	0.7401	0.8252	0.9453
		0.6532	0.6033	0.5298	0.6802	0.5968	0.5318	0.6795	0.5936	0.5408	0.6735	0.5908	0.5362	0.6692	0.5892	0.5429
	1e-05	0.6934	0.6832	0.6721	0.7071	0.7094	0.7062	0.7147	0.7187	0.7463	0.7189	0.7251	0.7794	0.7222	0.7339	0.8131
		0.693	0.6697	0.6061	0.7023	0.6859	0.5943	0.7078	0.6828	0.594	0.7082	0.6793	0.5888	0.7089	0.6764	0.5934
	1e-06	0.5397	0.7021	0.7016	0.5578	0.7111	0.7092	0.5652	0.7153	0.7149	0.5687	0.7177	0.7197	0.5652	0.7202	0.7261
		0.6795	0.7012	0.6891	0.6869	0.7081	0.6898	0.6902	0.7096	0.6868	0.6944	0.7101	0.6837	0.6952	0.7099	0.6808
0.1	0.01	0.6499	0.6508	0.6428	0.6822	0.6832	0.6853	0.7172	0.7118	0.716	0.744	0.7473	0.75	0.7705	0.7753	0.7743
		0.6107	0.6042	0.6016	0.6093	0.6021	0.6036	0.6063	0.6028	0.6032	0.613	0.6075	0.6094	0.612	0.6096	0.6074
	0.001	0.6721	0.6522	0.6407	0.6932	0.6849	0.6764	0.7149	0.7136	0.7177	0.7357	0.7454	0.7513	0.758	0.7715	0.7772
		0.6396	0.6059	0.5977	0.6341	0.6067	0.603	0.6351	0.6083	0.605	0.6341	0.6097	0.6114	0.6316	0.6105	0.6093
	0.0001	0.6877	0.6667	0.6511	0.7091	0.6925	0.6942	0.7183	0.7157	0.7149	0.724	0.7364	0.7456	0.7323	0.7599	0.7779
		0.6771	0.6316	0.6043	0.6885	0.6294	0.6063	0.685	0.6244	0.6098	0.6814	0.624	0.6075	0.6769	0.625	0.6097
	1e-05	0.6921	0.6904	0.6622	0.7064	0.7087	0.6923	0.713	0.7177	0.711	0.7175	0.7218	0.7399	0.7208	0.73	0.7638
		0.6921	0.6797	0.6183	0.7025	0.6883	0.6202	0.7062	0.685	0.6151	0.7087	0.6816	0.6211	0.7089	0.6767	0.6212
	1e-06	0.1977	0.696	0.699	0.2002	0.7073	0.7087	0.2173	0.7131	0.7135	0.2087	0.7158	0.7196	0.2139	0.718	0.7246
		0.6372	0.697	0.6871	0.6386	0.705	0.6897	0.6415	0.7084	0.6868	0.6426	0.7092	0.6837	0.6478	0.709	0.6807

Table 3: Mean In-Sample (first rows) and mean Out-Of-Sample (second rows) success rates of the improved Q-learning algorithm using the state exponential decay function with different parameter combinations. See sections 2.2.2 and 3.5.1 for more details.

		e = 0:1			e = 0:2			e = 0:3			e = 0:4			e = 0:5		
a_{init}	a_{end}	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5	b = 1	b = 0:75	b = 0:5
1	0.01	0.6891	0.6399	0.9617	0.6977	0.7158	0.9995	0.7059	0.778	1	0.7081	0.8299	1	0.7188	0.8706	1
		0.6837	0.5442	0.5118	0.6848	0.5497	0.5212	0.6836	0.5593	0.5163	0.6836	0.5643	0.5261	0.6837	0.5688	0.5294
	0.001	0.6116	0.6703	0.7473	0.6643	0.6898	0.8996	0.6819	0.698	0.9504	0.6895	0.7154	0.9805	0.6931	0.7331	0.9946
		0.6164	0.6536	0.5228	0.6694	0.6574	0.5191	0.6872	0.6523	0.5224	0.6947	0.6557	0.5294	0.6967	0.656	0.5269
	0.0001	0.4603	0.6524	0.6514	0.468	0.6873	0.6858	0.4671	0.6962	0.7105	0.4621	0.6993	0.7464	0.4551	0.7014	0.7721
		0.4924	0.6551	0.5991	0.5047	0.6874	0.6048	0.5085	0.6926	0.6058	0.5119	0.6945	0.6086	0.519	0.692	0.6089
	1e-05	0.0058	0.4789	0.6797	0.0032	0.4859	0.6921	0.0019	0.4804	0.6996	0.001	0.4757	0.7058	0.0004	0.4742	0.7102
		0.5004	0.5033	0.6758	0.5105	0.5162	0.6831	0.5045	0.516	0.6814	0.5015	0.5205	0.6805	0.5074	0.5297	0.6811
	1e-06	0	0.0087	0.5059	0	0.0044	0.5181	0	0.0023	0.5279	0	0.001	0.5378	0	0.0004	0.5509
		0.5011	0.4996	0.5249	0.5107	0.5091	0.5393	0.5024	0.5049	0.5554	0.5022	0.5051	0.5701	0.5128	0.5125	0.5914
0.5	0.01	0.6922	0.6431	0.682	0.6964	0.7147	0.8013	0.7051	0.7679	0.8741	0.7088	0.8073	0.9183	0.7139	0.8474	0.9561
		0.6846	0.5591	0.5255	0.6851	0.5659	0.525	0.6864	0.5661	0.5311	0.6826	0.5712	0.5401	0.6849	0.5746	0.5326
	0.001	0.5982	0.6714	0.6589	0.6537	0.6903	0.7656	0.6753	0.7042	0.8492	0.6839	0.7182	0.901	0.6871	0.7305	0.9351
		0.6038	0.6568	0.5291	0.6594	0.655	0.5316	0.6811	0.6558	0.5347	0.6901	0.6553	0.5345	0.6925	0.6575	0.5395
	0.0001	0.4577	0.6469	0.6447	0.4602	0.6867	0.683	0.4548	0.6952	0.7169	0.4441	0.697	0.7419	0.4298	0.6998	0.7765
		0.4945	0.6502	0.6018	0.5014	0.6871	0.6053	0.5044	0.6913	0.6134	0.5042	0.6929	0.6094	0.5087	0.6932	0.6099
	1e-05	0.0039	0.4744	0.6799	0.0021	0.4869	0.6942	0.0011	0.4847	0.6988	0.0005	0.4741	0.704	0.0002	0.4677	0.711
		0.4988	0.499	0.6748	0.5116	0.5166	0.6797	0.5042	0.5207	0.682	0.5005	0.5192	0.6802	0.5083	0.5261	0.6806
	1e-06	0	0.008	0.507	0	0.0041	0.5175	0	0.0022	0.5268	0	0.0009	0.5373	0	0.0003	0.55
		0.5002	0.5006	0.5254	0.5104	0.515	0.54	0.5026	0.5033	0.5528	0.5049	0.5078	0.5713	0.5123	0.5118	0.5915
0.1	0.01	0.686	0.6444	0.6414	0.6931	0.6854	0.6909	0.703	0.7123	0.7194	0.7097	0.7392	0.7463	0.715	0.7725	0.7792
		0.6809	0.6046	0.6007	0.6813	0.6108	0.6026	0.6841	0.6131	0.6059	0.682	0.6125	0.6077	0.6824	0.6185	0.6095
	0.001	0.5546	0.6762	0.6504	0.6112	0.6871	0.6896	0.6472	0.6984	0.7175	0.6654	0.711	0.7446	0.6715	0.728	0.7748
		0.5623	0.6584	0.6045	0.6212	0.6568	0.6004	0.6579	0.6602	0.6046	0.6762	0.654	0.6085	0.6829	0.6604	0.609
	0.0001	0.446	0.6177	0.654	0.4471	0.679	0.6802	0.4348	0.6896	0.7101	0.4143	0.6936	0.7261	0.3825	0.6972	0.7564
		0.5014	0.6215	0.6189	0.5119	0.6794	0.6202	0.5101	0.6891	0.624	0.5051	0.69	0.6213	0.5104	0.6912	0.6248
	1e-05	0.0015	0.4745	0.6826	0.0007	0.4842	0.6928	0.0003	0.4778	0.7014	0.0001	0.4714	0.7054	0	0.4628	0.7106
		0.5026	0.5021	0.6768	0.5081	0.5161	0.6821	0.5062	0.5157	0.6832	0.5026	0.5177	0.6816	0.5097	0.5231	0.6789
	1e-06	0	0.0069	0.507	0	0.0034	0.5151	0	0.0017	0.5282	0	0.0007	0.5399	0	0.0002	0.5488
		0.5014	0.4991	0.5264	0.513	0.5101	0.5374	0.5011	0.5084	0.5533	0.5052	0.5057	0.5745	0.5078	0.5131	0.5927

Table 4: Mean In-Sample (first rows) and mean Out-Of-Sample (second rows) success rates of the improved Q-learning algorithm using the epoch exponential decay function with different parameter combinations. See sections 2.2.2 and 3.5.1 for more details.