Journal Of Investment Management, Vol. 18, No. 2, (2020), pp. 37–56
© JOIM 2020

www.joim.com

DYNAMIC GOALS-BASED WEALTH MANAGEMENT USING REINFORCEMENT LEARNING

Sanjiv R. Das[a] *and Subir Varma*[a]

We present a reinforcement learning (RL) algorithm to solve for a dynamically optimal goal-based portfolio. The solution converges to that obtained from dynamic programming. Our approach is model-free and generates a solution that is based on forward simulation, whereas dynamic programming depends on backward recursion. This paper presents a brief overview of the various types of RL. Our example application illustrates how RL may be applied to problems with path-dependency and very large state spaces, which are often encountered in finance.

1 Introduction

Reinforcement learning (RL) has seen a resurgence of interest as the methodology has been combined with deep learning neural networks. Advances in hardware and software have enabled RL in achieving newsworthy successes, such as learning to surpass human-level performance in video games (Mnih *et al.*, 2013) and beating the world Go champion (Silver *et al.*, 2017). RL algorithms are particularly good at learning from experience and at optimizing the "explore–exploit" tradeoff inherent in dynamic optimization problems. This is why they can be fine-tuned to play dynamic games with superhuman levels of performance. The capacity of RL algorithms to learn from repeated trial episodes of games can be accelerated to a degree mankind cannot replicate. It is just not possible for anyone to play a million games over a weekend, whereas a machine can.

The dynamic optimization of portfolio wealth over long horizons is similar to optimal game play. Portfolio managers choose actions—that is, asset allocations, and hope to respond to market movements in an optimal manner with a view to maximizing long-run expected rewards. In this paper, we show how RL may be employed to solve a particular class of portfolio "game" known as goals-based wealth management (GBWM). GBWM has recently gained widespread acceptance by practitioners and the use of RL will inform the growth in this paradigm.

Since the seminal work of Markowitz (1952), there has been a vast literature on portfolio

[a]Santa Clara University

optimization, broadly defined as allocating money to a collection of assets (portfolio) with an optimal tradeoff between risk and return. In practice, return is defined as the weighted average mean return of the portfolio and risk is usually the standard deviation of this weighted return. Markowitz's early work detailed a static (one-period) optimization problem that forms the kernel for many dynamic optimization problems, where a statically optimal portfolio is chosen each period in a multiperiod model in order to maximize a reward function at horizon T. The reward function may be (1) based on a utility function, or (2) on whether the final value of the portfolio exceeds a desired threshold. The latter type of reward function underlies the broad class of goals-based wealth management (GBWM) models, see Chhabra (2005), Brunel (2015), and Das *et al.* (2018).

In GBWM, we seek to maximize the probability that an investor's portfolio value W will achieve a desired level of wealth H—that is, $W(T) \geq H$ at the horizon T for the goal. Starting from time $t = 0$, with wealth $W(0)$, every year, we will choose a portfolio that has a specific mean return (μ) and risk (denoted by the standard deviation of return σ) from a set of acceptable portfolios, so as to maximize the chance of reaching or exceeding threshold H. We may think of this as playing a video game where we see the portfolio move in random fashion, but we can modulate the randomness by choosing a (μ, σ) pair at every move in time. This game may be solved in two ways: (i) At time T, we consider all possible values of wealth $W(T)$ and assign high values to the outcomes where we exceed threshold H and low values to outcomes below H. We then work backwards from all these possible values of wealth to possible preceding values and work out which move (μ, σ) offers the highest possible expected outcome. This approach is widely used and is classic dynamic programming, see Das *et al.* (2019). It is known in the reinforcement learning (RL) literature as solving the "planning problem." We provide formal details on this briefly in Section 5.

Dynamic programming (DP) via backward recursion is oftentimes hard to implement because backward recursion is computationally costly, usually because (1) the number of states in the game is too large, or (2) the transition probabilities from one state to another are unknown and need to be estimated. Instead, we may resort to forward propagation game simulations, and improve our game actions by playing repeated games and learning from this play as to which actions are optimal for each state of the game we may be in. This approach, known as RL, has also been widely studied, see for example, Sutton and Barto (1998).[1] The video game analogy for RL has become popular since Mnih *et al.* (2013) used the approach to beat human-level performance at playing the Atari video game and many more. In this paper, we survey the various kinds of RL and show how we may solve a multiperiod retirement portfolio problem—that is, optimize GBWM.

This paper proceeds as follows. In Section 2 we review static mean–variance optimization and set up the dynamic multiperiod goal-based optimization problem as an example of game playing that may be solved using RL. In Section 3, we review how the set of efficient portfolios that comprise the action space in the model is computed, using the mean–variance solution of Markowitz (1952). Section 4 formulates the dynamic problem in terms of Markov Decision Processes (MDPs). Section 5 describes the various types of DP and RL algorithms that we may consider. Our taxonomy discusses (i) model-based versus model-free RL approaches, (ii) value iteration versus policy iteration as a solution approach, (iii) on-policy versus off-policy approaches, and (iv) discrete-state space solutions versus continuous-state space solutions that use deep learning neural nets. Section 6 presents the specific Q-Learning algorithm we use to solve the dynamic portfolio problem. This approach will be model-free, policy iterative, off-policy, and embedded in a discrete state space. Section 7 reports results of illustrative numerical experiments, and Section 8 offers concluding discussion.

2 Portfolio optimization: Statics and dynamics

In this section, we briefly recap static mean–variance optimization, define GBWM, recast dynamic portfolio optimization as a game, and introduce RL as a feasible solution approach.

2.1 Mean–variance optimization

The objective of Modern Portfolio Theory is to develop a diversified portfolio that minimizes risk—that is, the variance of portfolio return, for a specified level of expected (mean) return. This problem, known as mean–variance portfolio optimization, takes as input a vector of mean returns $M = [M_1, \dots, M_n]^{\mathrm{T}}$ and covariance matrix $\sum$ of returns of n assets. Once the investor specifies a required expected return μ for the portfolio, a set of asset weights $w = \{w_j\} = [w_1, \dots, w_j, \dots, w_n]^{\mathrm{T}}$, $1 \le j \le n$, is chosen to minimize portfolio return variance σ^2. The portfolio expected return and standard deviation are functions of the inputs—that is,

$$\mu = \sum_{j=1}^{n} w_j M_j = w^{\mathrm{T}} \cdot M; \quad \sigma = \sqrt{w^{\mathrm{T}} \cdot \sum \cdot w}$$

The asset weights are proportions of the amount invested in each asset, and these must add up to 1—that is, $\sum_{j=1}^{n} w_j = w^{\mathrm{T}} O = 1$, where $O = [1, 1, \dots, 1]^{\mathrm{T}} \in R^n$. Also, the portfolio must deliver the expected return μ. For every μ, there is a corresponding optimal σ, obtained by solving this portfolio problem for optimal weights vector w. This collection of (μ, σ) pairs is called the "Efficient Frontier" of portfolios.

An investor's wealth is statically managed by choosing the best portfolio from this "efficient set" at any point in time to dynamically meet her goals. This is the traditional approach to the portfolio optimization problem and it proceeds by choosing the portfolio that minimizes the overall portfolio risk σ, while achieving a given return μ. See Das *et al.* (2018) for the static optimization problem and Merton (1969, 1971), for the dynamic programming solution to the multiperiod problem in continuous time.

2.2 Goals-based wealth management

Recent practice is leaning towards an alternative formulation of the portfolio optimization problem that uses the framework of GBWM. In this formulation risk is understood as the probability of the investors not attaining their goals at the end of a time period, as opposed to the standard deviation of the portfolios. However, there is a mathematical mapping from the original mean–variance problem to the GBWM one, as explained in Das *et al.* (2010). Whereas this problem is a static one, the dynamic version of this GBWM problem is solved in Das *et al.* (2019), where DP is used to solve the long-horizon portfolio problem.

DP has been used to solve multiperiod problems in finance for several decades. The essence of the problem may be depicted as follows. At any point in time t, an investor's retirement account has a given level of wealth $W(t)$. The investor is interested in picking portfolios every year to reach a target level of wealth H at time T—that is, she wants that $W(T) \ge H$. Of course, this is not guaranteed, which means that the goal will only be met with a certain level of probability. The GBWM optimization problem is to reach the goal with as high a level of probability as possible—that is, with objective function:

$$\max_{w(t),\, t<T} Pr[W(T) > H]$$

This entails choosing a portfolio at each t, with a corresponding level of mean return and risk, $w(t) = (\mu, \sigma)_t$, where these portfolios are chosen from a select set of "efficient" portfolios, mentioned in Subsection 2.1 and described in the following Section 3. (Be careful not to mix up $w \in \mathcal{R}^n$, which is a vector of asset weights, and $W(t)$, which is a scalar level of wealth at period t.) Thus, we solve for an optimal "action" $w[W(t), t]$, a function of the "state" $[W(t), t]$. Here the state has two dimensions and the action chooses one of a collection of possible (μ, σ) pairs, which determines the range of outcomes of the wealth at the next period, $W(t + 1)$.

2.3 *Dynamic portfolio optimization as a game*

If you think of portfolio optimization as a game with a specific goal (with a corresponding payoff known as the "reward function"), then DP delivers a strategy telling the investor what risk–return pair portfolio to pick in every eventuality that may be encountered along the path of the portfolio, such that it maximizes the probability of reaching and exceeding the pre-specified threshold value H. The solution approach uses a method known as "backward recursion" on a grid, which is intuitive. Create a grid of wealth values W for all time periods—that is, $[W(t), t]$ (think of a matrix with time t on the columns and a range of wealth values W on the rows). The reward in column T is either 1 (if $W(T) \geq H$) or 0 otherwise, signifying the binary outcome of either meeting the goal or not. From every wealth grid point at time $T-1$, we can compute the probability of reaching all grid points at time T. We then pick the portfolio—that is, a (μ, σ) pair, that maximizes the expected reward values at time T for a single node at time $T-1$, and we do this for all nodes at time $T-1$. Computing the expected reward assumes that the transition probabilities from state i at time $T-1$ to state j at time T—that is, $Pr[W_j(T), T \mid W_j(T-1), T-1]$, are known (we will describe these probabilities in the next section). We have then found the optimal action—that is, $A = (\mu, \sigma)$ choice, for all nodes at $T-1$, and we can also calculate the expected final reward at each of the $T-1$ nodes. This expected final reward, determined at all states $[W(T-1), T-1]$, is known as the "value function" at each state. Then, proceed to do the same for all nodes at $T-2$, using the rewards at all nodes at time $T-1$. This gives the optimal action and expected terminal reward (value function) for each node at time $T-2$. Keep on recursing backwards until time $t = 0$. What we then have is the full solution to the GBWM problem at every node on the grid—that is, in one single backward pass. This is an extremely efficient problem-solving approach and gives the full solution to the problem in one pass. This problem setup is called the "planning problem" and the DP solution approach is easily implemented because the transition probabilities from one state to the next are known.

So far, DP via backward recursion has served its purpose well because the size of the problems has been small in terms of the number of state variables and action variables, and the transition probabilities are known. The computational complexity of the problem depends on the number of states—that is, the number of grid points we choose, which is tractable when the state comprises just wealth W and time t. Complexity also depends on the number of possible actions to choose from as each of these has to be explored. In the GBWM case, this depends on the number of choice portfolios we consider. Backward recursion considers all possible scenarios (states) that might be encountered in the portfolio optimization game and computes the best action for each state. It exhaustively enumerates all game positions and works well when the number of cases to be explored is limited. Compare this approach to building a program to play chess, where the goal is to solve for the best action for all possible board configurations, an almost impossible task even to describe the state space succinctly.

2.4 *Reinforcement learning solutions*

When the state and action space becomes large, and it becomes difficult to undertake exhaustive enumeration of the solution, then learning from experience through game playing is often a more efficient solution approach. In situations where the model of the environment is unknown—that is, transition probabilities in the large state space are not available *a priori*, then exploration through game playing is required to obtain an estimate of transition probabilities. Note that backward recursion is really an attempt to specify a solution to all possible game scenarios one may experience, which is a limit case of learning from limited experience through game playing.

In the portfolio optimization game, a player experiences a level of wealth W at each point in time and takes an action $A = (\mu, \sigma)$. A series of such state–action (S, A) pairs in sequence, through to horizon T, is a single game and results in a reward of 1 or 0. Clearly, we want to increase the likelihood of playing (S, A) pairs that lead to rewards of 1 and downplay those that lead to 0 rewards. That is, we "reinforce" good actions. RL is the process of training our model through repeated play of portfolio optimization game sequences. We note that in both DP and RL, actions are a function of the current state and not preceding states. This property of the sequence of actions characterizes it as a Markov decision process (MDP), which we will describe later in this paper.

Through the RL process the agent builds up a set of actions $A(W(t), t)$ to be taken in each state $[W(t), t]$. The set of actions is also known as the "policy," which is a function of the state space. As the agent plays more episodes, the policy is updated to maximize expected reward. With the right game playing setup, the policy will converge to the optimal in a stable manner.

Unlike DP, which is solved by backward recursion, RL is a forward iteration approach—that is, we play the game forward and assess the ultimate reward. RL does not visit each possible state, only a certain number, determined by how many games, known as "episodes" in the RL pantheon, we choose to play and the behavior of the random process governing the evolution of the system. The hope is that RL is efficient enough to approach the DP solution with far less computational and algorithmic complexity. In this paper, we solve the GBWM dynamic portfolio problem using a variant of RL, known as "Q-Learning" (QL). We cross check that the solution obtained by DP is also attained by QL, affirming that RL works as intended for dynamic portfolio optimization.

3 Candidate portfolios

Our problem is restated as follows: Assume that the portfolios have to be chosen at fixed intervals ($h = 1$ year) and discrete periods $t = 0, 1, 2, \ldots, T$ and the amount of wealth at time t is given by $W(t)$, $0 \le t \le T$. The threshold return is denoted as H, and the dynamic GBWM game is to choose the action $A[W(t), t] = (\mu, \sigma)_{W(t),t}$ driven by portfolio weights for wealth W_j at time period t, $w_j(t)$, at each time period $w_j(t)$, $1 \le j \le n$, $0 \le t \le T-1$, such that the probability at the final time T of the total wealth exceeding H, given by $P(W(T) > H)$, is maximized.

Each period, we restrict our portfolios to a discrete set of portfolios—that is, (μ, σ) pairs, that lie along what is known as the "efficient frontier," described in the previous section as the solution to a problem where we find a locus of points in

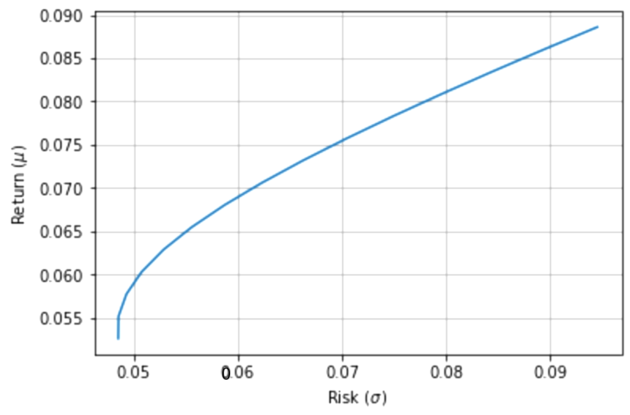

Figure 1 The efficient frontier of (σ, μ) pairs, which are possible portfolio choices available to the investor each year.

(σ, μ) space (see Figure 1) such that for each μ we have the portfolio w that minimizes variance $\sigma^2 = w^{\mathrm{T}} \sum w$. The mathematics in Das *et al.* (2018) shows that the equation for the efficient frontier is as follows:

$$\sigma = \sqrt{a\mu^2 + b\mu + c}$$

This curve traces out a hyperbola, as shown in Figure 1. The values a, b, c are given by $a = h^T \sum h$, $b = 2g^T \sum h$, $c = g^T \sum g$, where the vectors g and h are defined by $g = \frac{l \sum^{-1} O - k \sum^{-1} M}{lm - k^2}$, $h = \frac{m \sum^{-1} M - k \sum^{-1} O}{lm - k^2}$, and the scalars k, l, m are defined by $k = M^T \sum^{-1} O$, $l = M^T \sum^{-1} M$, $m = O^T \sum^{-1} O$. In these equations $M = [M_1, \ldots, M_n]^{\mathrm{T}}$ is the vector of the n expected returns of the portfolio assets, O is the vector of n ones, and $\sum$ is the covariance matrix of the n assets.

In the rest of this paper we restrict the portfolio choice at all instants of time $0 \leq t \leq T$ to one of a set of K evenly spaced portfolios that lie along the Efficient Frontier curve, given by $(\mu_1, \sigma_1), \ldots, (\mu_K, \sigma_K)$. We note that the solution here is a one-period solution that delivers the best locus of portfolios that are inputs into the action space of the RL algorithm. The optimal one-period choice is applied each period in dynamic manner leading to an optimal dynamic programming solution. Note that choosing the portfolio with the best return may not be the optimal policy, since higher returns also come with higher variance, which may cause degradation of the objective function.

4 Formulation as a Markov decision processes (MDP)

We now move from the static problem to considering the dynamic portfolio game. At any given state—that is, level of wealth $W(t)$, we choose an action $w(t) \equiv A(t) \equiv [\mu(t), \sigma(t)]$. The next step in the game is to generate the next period state $W(t+h)$, which is stochastic. The time between transitions is h. For illustrative purposes, we choose a stochastic process that transitions $W(t)$ to $W(t+h)$, and we choose geometric Brownian motion (GBM), as in Das *et al.* (2019):

$$W(t+h) = W(t) \exp\left\{\left[\mu(t) - \frac{1}{2}\sigma^2(t)\right]h + \sigma(t)\sqrt{h} \cdot Z\right\} \quad (1)$$

where $Z \sim N(0,1)$. Therefore, randomness is injected using the standard normal random variable Z. GBM is one of the most popular choices used in financial modeling and therefore, we employ it here. However, the choice of Z as Gaussian is not strict and we may use any other distribution without loss of generality.

We are now ready to formulate the portfolio optimization problem as an MDP. The MDP state is defined as $(W(t), t)$, where $W(t)$ is the value of the portfolio at time t. For example the portfolio value is $W(0)$ at the start of the MDP, and the MDP terminates at $t = T$ when the portfolio value is $W(T)$. RL is then used to find the optimal policy for the MDP through playing repeated episodes. A *policy* is a mapping from the state $\{W(t), t\}$ to

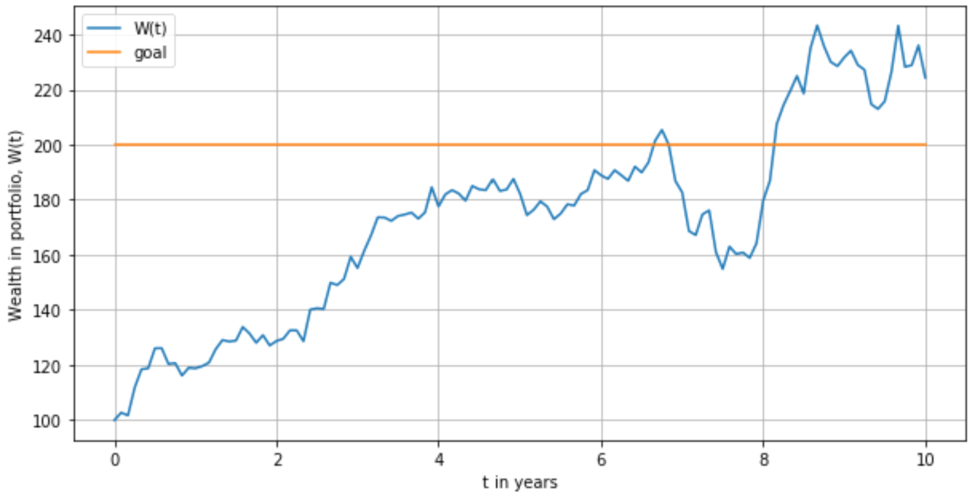

Figure 2 Evolution of the total portfolio value with time. Choose the portfolios at $t = T - 1$ so as to maximize the probability of $W(T)$ being greater than the goal threshold.

action $A = \{\mu, \sigma\}$. Each episode will proceed as follows:

1. Starting with an initial wealth of $W(0)$ at time $t = 0$, we implement one of the $k = 1 \ldots K$ portfolios, denoted by $[\mu(0), \sigma(0)]_k$. Which portfolio (action) is chosen will depend on the current policy in place. (At the outset, we may initialize the policy to be a guess, possibly random, or some predetermined k.)
2. Using the Geometric Brownian Motion model, the wealth $W(t + h)$ at time $t + h$ is a random variable, given by the transition Equation (1). Using this formula, we can sample from the random variable Z to generate the next wealth value $W(t + h)$ at time $t + h$, conditional on choosing an action $[\mu(t), \sigma(t)]$. We then once again use our policy to choose the next portfolio $[\mu(t + h), \sigma(t + h)]$. The sequence of these portfolios is an MDP. In our example, we have chosen the transition equation explicitly to generate the next state, but we are going to solve the problem as if we do not know Equation (1).
3. Set $t \rightarrow t + h$. If $t = T$ then stop, otherwise go back to step 2.

The system described above evolves in discrete time and action space, but in continuous wealth space (see Figure 2). For implementing the Q-learning solution later we will restrict the wealth space to be discrete as well. This approach may be generalized to continuous spaces by using function approximations based on neural networks.

4.1 State space

Our approach is to define a large range of

$$[W_{\min}, W_{\max}] = [W(0)\exp(-3\sigma_{\max}\sqrt{T}),\\ W(0)\exp(+3\sigma_{\max}\sqrt{T})]$$

at the end of the time horizon T, which will be an array of final values of $W(T)$, suitably discretized on a grid. Here $\sigma_{\max}$ is taken to be the highest possible standard deviation of return across all candidate portfolios from Section 3. The number of grid points on the wealth grid is taken to be $(10T + 1)$, so if $T = 10$, then we have 101 grid points. The grid points are equally spaced in log-space—that is, equally over the range $[\ln(W_{\min}), \ln(W_{\max})]$. From the initial wealth $W(0)$, we assume that it is possible to transition to any wealth value on the grid at the end of the period ($t = h$), though the probability of reaching extreme values is exceedingly small. From each wealth value at $t = h$ we assume transition is possible to all the wealth grid values at $t = 2h$, and so on.

Hence, the grid is "fully connected." Transition probabilities from a node i at t to node j at $t + h$ are described in the next subsection.

4.2 *Transition probability*

Using Equation (1) that describes the evolution of $W(t)$, we can write down the following equation for the transition probability from state $W_i(t)$ at time t to state $W_j(t+h)$ at time $t+h$ (a transition from node i to node j):

$$Pr[W_j(t+h) \mid W_i(t)] = \phi\left[\frac{\ln\left(\frac{W_j(t+h)}{W_i(t)}\right) - (\mu - 0.5\sigma^2)h}{\sigma\sqrt{h}}\right]$$

where $\phi(\cdot)$ is the normal probability density function (pdf) because the Z in Equation (1) is Gaussian. In order to fully define the MDP, we also need to specify the reward function. In this model, the reward R is only specified for the final states $W_0(T), \ldots, W_{10T}(T)$, and is as follows: $R = 1$ if $W_j(T) \geq H$, and 0 otherwise. This is akin to a video game reward—that is, you get 1 if you win the game and 0 if you lose.

In the case where the investor is allowed to make infusions $I(t)$ into the portfolio over time, transition probabilities are adjusted to account for these additional cash flows coming into the portfolio. The revised transition probabilities are as follows:

$$Pr[W_j(t+h) \mid W_i(t), I(t)] = \phi\left[\frac{\ln\left(\frac{W_j(t+h)}{W_i(t)+I(t)}\right) - (\mu - 0.5\sigma^2)h}{\sigma\sqrt{h}}\right]$$

Because the transition probabilities are known, but we solve the problem through forward simulation, we are not explicitly using the transition probabilities in determining the optimal actions. We only use the transition probabilities to generate the next state with the correct probabilities. The analogy here to video gaming (for this portfolio game) is that the game designer has to use some transition probabilities with which states are generated in the game, but these are not given to the game player. So, as problem designer, we define the system and its transition probabilities using Equation (1) but we do not use these explicitly to discover the solution, i.e., the optimal policy. Therefore, the RL solution approach we employ in this paper is an example of *model-free* RL. The distinction between model-based and model-free RL is discussed next and we describe a broad taxonomy of RL approaches.

5 RL taxonomy: Methods for solving the MDP

A "policy" $\pi(s)$ is defined as a mapping from the state s to one of the portfolios in the set A of actions $\{(\mu_1, \sigma_1), \ldots, (\mu_K, \sigma_K)\}$. The optimal policy $\pi_*(s)$ is the one that maximizes the total expected reward, which in this case is the probability of the final value $W_i(T)$ exceeding the threshold H. Solving the MDP is the process of identifying this optimal policy.

MDP solution methods can be classified into the following categories:

1. *Model-based algorithms*: These algorithms assume that the state transition probabilities are known. For a given policy π, they are based on the concept of a State Value Function $V_\pi(s)$, $s \in [W(t), t]$, and the State–Action Value Function $Q_\pi(s, a)$, $a \in A$. The Value Function $V_\pi(s)$ is the expectation of total reward starting from state s under policy π, while the State–Action Value Function $Q_\pi(s, a)$ is

the expectation of total reward starting from state s and using a as the first action, and policy π thereafter. Similarly, $V_*(s)$ and $Q_*(s, a)$ are the corresponding value functions that are attained when using the optimal policy π_*. The value function under the optimal policy satisfies the Bellman Optimality Equation (BOE), see Bellman (1952, 2003) and Bellman and Dreyfus (2015), which is a formal statement of the backward recursion procedure described in Section 2.3:

$$\begin{aligned} V_*(s_t) &= \max_{a_t} E[V_*(s_{t+1}) \mid s_t] \\ &= \max_{a_t} E\left\{ \max_{a_{t+1}} E[V_*(s_{t+2}) \mid s_{t+1}] \mid s_t \right\} \\ &= \max_{a_{t+h},\dots,a_{T-h} \in \pi(s)} E[V_*(s_T) \mid s_t] \end{aligned} \tag{2}$$

where the last equation follows from the law of iterated expectations and the Markov property. This is a simple version of the Bellman equation because the reward is only received at maturity and there are no intermediate rewards. The same equation without the "maximization" is simply the Bellman Expectation Equation (BEE) and gives the value of any policy, which may not be optimal. A similar equation holds for the State–Action Value function, also known as the Q function (for "quality"):

$$Q_*(s_t, a_t) = \max_{a_t, a_{t+h},\dots,a_{T-h} \in \pi(s)} E[Q_*(s_T) \mid s_t] \tag{3}$$

Once we know either $V_*(s)$ or $Q_*(s, a)$, we can readily compute the optimal policy. To arrive at these optimal functions, one of the following algorithms is used:

- Value Function Iteration.
- Policy Function Iteration.

Both of these are iterative algorithms that work by updating value and/or policy functions through episodes of game playing. Value iteration iterates on the BOE, whereas policy iteration iterates on the BEE.

RL may be implemented on a discrete or continuous state and action space. If both state and action spaces are discrete and finite (and of small size), then it is feasible to maintain grids (tensors of any dimension) for state and action, and solve for $V_*(s)$ and $Q_*(s, a)$ at each point on the grid. This is known as "tabular" RL. In our GBWM problem, the state space has two dimensions $W(t)$ and t, and the action space has one (K portfolios), so the tabular grid will be of dimension three—that is, $V_*(s)$ and $Q_*(s, a)$ will be values on a 3D tensor.
To get some quick intuition about the approach, we define the following components of the algorithm.

- State $s(t)$: The current value of variables on which the decision (action) is based. In our example, this is the level of wealth $W(t)$ and time t, and is represented by a node on the grid.
- Action $a(s(t))$: Define the action a as an element in the index set $\{1, 2, \dots, K\}$, and the policy $\pi(s)$ as a mapping from the state s to one of the elements in the action set. In our case, this is the portfolio chosen until the next state is realized, at which time another action will be taken. The series of actions is often denoted as a "plan" and hence, learning is analogous to solving a "planning" problem, the result of which is a policy—that is, resulting in a series of actions $(a_0, \dots, a_{T-h})$.
- Reward $r(s(t), a(t))$: At each state, the agent may or may not receive a reward for the action taken. In our example, rewards are only received at the final horizon T of the problem.

- Transition probability $p[s(t+h) \mid \{s(t), a(t)\}]$: This defines the likelihood of moving to a probabilistic state the following period, conditional on the current state and action.

The value function $V(s(t))$ is defined over the same grid. The solution procedure for this problem consists of starting from time T and populating the value function in the last section of the grid—that is, $V(s(T))$. For our problem, the value is binary—that is, if $W(T) \geq H$, then $V(s(T)) = 1$, else it is equal to 0. Once we have populated the value function at time T, we can proceed to populate the value function at time $(T-h)$, using backward iteration based on the Bellman equation.

$$\begin{aligned} V(s(T-h)) &= \max_{a(s(T-h)} E[V(s(T))|s(T-h)] \\ &= \max_{a(s(T-h))} \Bigg[\sum_{i=0}^{m} \{p(s(T) \mid s(T-h) = i, \\ &\qquad a(T-h))\} \cdot V(s(T)) \Bigg] \qquad (4) \end{aligned}$$

where an expectation has been taken over values $V(T)$ in all m states in the next period using the transition probability function $p(s(T) \mid s(T-h), a(T-h))$. This equation embodies the "backward recursion" solution procedure, because the same equation may be applied for all periods from $t = T-h$ to $t = 0$. This procedure is value iteration and we first solved the GBWM problem this way using DP to determine the optimal value function.

In policy iteration, we choose a random policy, and then find the value function of that policy (policy evaluation step). Find a new and better policy based on the previous value function, and so on, until no further improvement is possible. During each iteration of the algorithm, the BEE can be used to compute the value function for the current policy. Here the policy is explicitly chosen, starting from an initial functional guess. Standard DP is almost always amenable to policy iteration, and we have solved the portfolio problem using RL that way.

When all the components of the problem s, a, r, p (state, action, reward, and transition probability) are known, the algorithm is denoted "model-based." Often, one or both of the r, p functionals are not known in advance, and have to be learned while solving the problem, usually through repetitive play. This is denoted as "model-free" learning.

2. *Model-free algorithms*: If the state transition probabilities are not known in advance, then the MDP is solved by collecting sample-paths of the state transitions, which are generated by the "environment" (latent transition probabilities), and the corresponding rewards, and then estimating the optimal State–Action Value Function $Q_*(s, a)$ using statistical averaging. Note that the state value function $V_*(s)$ is no longer useful in the model-free case, since even if it were known, the calculation of the optimal policy still requires knowledge of the state dynamics. On the other hand, once $Q_*(s, a)$ is known, the optimal policy can be obtained by doing a simple maximization $\pi_*(s) = \arg\max_a Q_*(s, a)$. The two main classes of Model Free algorithms are:

   - Monte Carlo (MC) Learning: These algorithms work for cases when the MDP sample paths terminate, and proceed by estimating $Q_\pi(s, a)$ by averaging the total future rewards seen whenever the system is in state s and the agent takes action a. This results in an unbiased estimate of $Q_\pi(s, a)$, however it is subject to a large variance, as a result of which a large number of sample paths are needed to get a good estimate.

- Temporal Difference (TD) Learning: These algorithms work even for nonterminating MDPs, and are lower variance and thus more sample efficient than Monte Carlo methods. They use a one-step version of the BEE given by the following iteration:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]. \qquad (5)$$

The reward R, state s, and action a are seen at time t, and the state and action next period are denoted s' and a', respectively. The parameter α proxies for the "learning rate" and is usually chosen to be a small value. The "discount rate" is $\gamma \leq 1$ and it suffices to tradeoff later rewards against earlier ones in an episode. It also helps to set a horizon on the importance of rewards when episodes do not terminate in a short horizon.

We seek to obtain estimates of the optimal State-Action Value Function $Q^*(s,a)$ using the generated sample paths. When in state s, the action a is generated according to what is known as an "epsilon-greedy" policy π. Under this ϵ-greedy approach, the current action a is chosen based on the current policy with probability $(1-\epsilon)$ but with probability ϵ, a random action is chosen. Using the current policy is known as "exploitation" and using the random policy generates "exploration" behavior. Exploration is a key ingredient in RL, because it enables better coverage of the state space. This idea optimally implements the *exploitation–exploration tradeoff.* Staying on the beaten track (exploitation) may not lead to the best solution and some wandering (exploration) often leads to discovering better outcomes. Note that for our specific portfolio problem, $R = 0$ except at time T, when it takes a value in $\{0, 1\}$. The equation above can therefore, also be written as

$$Q(s,a) \leftarrow Q(s,a)(1-\alpha) + \alpha(\gamma \cdot Q(s',a')) \qquad (6)$$

where we note that this update equation sets the new value of $Q(s,a)$ to a weighted average of the current value function and the value function in the next period, and when α is small, the learning is of course slow, but convergence is more stable.

This formula in Equation (5) uses the following sample path transitions: Start from state s and take action a (using the ϵ-greedy policy) to generate reward R, followed by a probabilistic transition to state s' from where the action a' is taken, again under the ϵ-greedy policy. This is followed by a version of the policy iteration algorithm, to progressively refine the policy, until it converges to the optimal policy π_*. TD Learning comes in two flavors:

(a) SARSA-Learning: This is an "on-policy" version of TD Learning, in which the policy π being followed to generate the sample paths is the same as the current iteration of the optimal policy. Note that the current policy may not be optimal unless it has converged. In Equation (6), both a and a' are chosen using the current policy. Therefore, it is called "on- policy" learning.

(b) Q-Learning: This is an "off-policy" version of TD Learning, in which the policy being used to generate the sample paths (called the "behavior policy") may not be the same as the current iteration of the optimal policy (called the "target policy"). This is a very beneficial property to have for two reasons: (1) The behavior policy can be designed to explore more states and actions, thus improving the Q-estimates. Using the optimal target policy instead to generate sample paths leads to the problem that not all states and actions will be fully explored. (2) Due to the off-policy nature of Q-learning, state transitions can be stored and used multiple times in order to improve the Q estimates. In contrast on-policy methods need to generate new sample paths every time the policy changes. The iteration in Q-Learning is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \max_{a'} \times \gamma Q(s',a') - Q(s,a)]$$

In this equation action a is chosen according to the behavior policy while a' is chosen according to the target policy. See that in Q-Learning the policy a' is chosen optimally from highest value function in state s', unlike in the case of SARSA, where it is chosen based on the current policy function. In this paper, we implement Q-Learning on a 3D tensor—that is, we implement tabular RL.

3. *Algorithms that use function approximators*: The reinforcement learning algorithms described so far are tabular in nature, since they work only for discrete values of states and actions. If this assumption is not satisfied, or if the number of states (or actions) is extremely large, then these methods do not work any longer. In their place we have a range of methods that use a function approximator, such as a neural network, rather than a table, to represent the value function. This results in the following two classes of algorithms:
   - Deep Q-Learning: In this case a neural network is used to approximate the state–action value function $Q_\pi(s,a)$. The neural network is trained in a supervised fashion, by using training sample paths from the MDP to generate the ground-truth values for Q. Some recent successes of RL, such as the Atari game playing system developed by DeepMind, were based on the deep Q-Learning algorithm. These are known as DQNs or deep Q nets, see Mnih *et al.* (2013).
   - Policy gradients: This is an alternative approach to RL, in which the policy is optimized directly (as opposed to indirectly obtaining the policy by first estimating value functions). In order to do so, policies are represented using neural networks, and the policy optimization proceeds by using well-known techniques such as stochastic gradient ascent. Policy gradient methods work even for cases when the action space is continuous and can also accommodate randomized polices. Applications of RL to areas such as robotics and finance often use policy gradients.

Next, we describe the specifics of our algorithm for the GBWM problem.

6 Our algorithm

We first solve the problem using classical DP based on Equation (2). This gives us solutions against which we may check our RL algorithm. We do not provide further details regarding the standard Bellman approach for DP as it is well known.

The algorithm we use solve the MDP is the type of RL algorithm called tabular Q- Learning and is stated below:

- Define a quality function Q that maps to each of the states and actions in the MDP and initialize it to 0—that is, $Q(W(t), t, a_k) = 0, 0 \leq t \leq T, 1 \leq k \leq K$. Note that action a_k corresponds to using the efficient frontier pair (μ_k, σ_k).

- Set $W(0)$ to a constant corresponding to the initial wealth at time $t = 0$.
- Initialize time to $t = 0$ and repeat the following steps in a loop for M episodes (or training epochs):

1. Choose action a as the one that maximizes $Q(W(t), t; a_k), 1 \leq k \leq K$, as modified by the ϵ-greedy algorithm. The ϵ-greedy approach is one that arbitrarily chooses a random strategy with probability ϵ to implement the "explore versus exploit" tradeoff. We describe the exact specification of the ϵ-greedy choice in the next section.
2. Transition to the next state $(W(t + 1), t + 1)$, where $W(t+1)$ is sampled using the MDP state transition probability values. While we know the exact transition function, we operate as if this is generated by the environment and is not known to the agent.
3. Choose the next action a' in state $(W(t + 1), t + 1)$, as the one that *maximizes* $Q(W(t + 1), t + 1, a_k), 1 \leq k \leq K$
4. Update the Q value of the original state $(W(t), t)$ and action a, using
$$Q(W(t), t, a) \leftarrow Q(W(t), t, a) + \alpha[0 + \gamma Q(W(t + 1), t + 1, a') - Q(W(t), t, a)].$$
Note that rewards are 0 for intermediate states $t < T$.
5. $t \to t + 1$.
6. If $t = T$, then this is the end of the episode. Update the Q values for the final state $(W(T), T)$:
$$Q(W(T), T, a) \leftarrow Q(W(T), T, a) + \alpha[1 - Q(W(T), T, a)] \quad \text{if } W(T) \geq H$$
$$Q(W(T), T, a) \leftarrow Q(W(T), T, a) + \alpha[0 - Q(W(T), T, a)] \quad \text{if } W(T) < H$$
Increase the number of episodes by 1, set $t = 0$ and re-initialize the state to $(W(0), 0)$ to start a new episode.
7. If $t < T$ then go back to step 1 to continue the current episode.

Both the DP (Planning) algorithm and the Q-Learning algorithm were implemented in the Python programming language. The first component of the algorithm was to decide the grid for portfolio wealth outcomes. Below we display only some snippets of code in order to make the programming of the algorithm clearer.[2] Think of these snippets as pseudo-code.

1. Create the wealth grid. The code below creates an equally spaced grid in log wealth, which is then translated back into wealth by exponentiation (line 8). `W0` is initial wealth. The number of periods is T (line 4). The infusions at each time t are `I(t)` (lines 5, 6). The grid size was set to 101 nodes (line 7).

```
1 lnW = log(W0)
2 lnw_min = lnW
3 lnw_max = lnW
4 for t in range(1,T+1):
5     lnw_min = log(exp(lnw_min)+I[t]) + (mu_min - 0.5*sig*sig)*h
    - 3*sig*sqrt(h)
6     lnw_max = log(exp(lnw_max)+I[t]) + (mu_max - 0.5*sig*sig)*h
    + 3*sig*sqrt(h)
7 lnw_array = linspace(lnw_min,lnw_max,101)
8 w_array = exp(lnw_array)
```

2. Construct a blank 3D tensor that combines the 2D state space and the 1D action space. This will hold the tabular $Q(S, A)$ function values. The first dimension of the tensor is wealth, the second one is time (from 0 to T), and the third is the action space, where `NP` $= K$ is the number of portfolios available to choose from.

```
Q = zeros((len(w_array), TT+1, NP))
```

3. Initialize 3D reward tensor in $\{W, t, a\}$. We see that rewards are only attained at maturity in this problem if the final wealth value is greater than goal level H. We see that the `Q` and `R` tensors are of the same dimension.

```
R = zeros((len(w_array), TT+1, NP))
for j in range(maxlenW):
    if W[TT][j]>H:
        R[j,TT,:] = 1.0
```

4. State transition under the policy. Suppose we are at node $W_i(t)$ at time t and transition to a node at time $t + 1$, denoted $W_j(t + 1)$. Which node we transition to depends on the environment (transition probabilities), but these in turn depend on the action taken—that is, $a_k = (\mu_k, \sigma_k)$. We create a separate function to generate state transitions—that is, to mimic the behavior of the portfolio's wealth from the underlying environment. Given current scalar `w0`, `t0`, and action `a0`, sample a transition to wealth vector `w1` at time `t1` (line 1). Infusions are denoted by variable I (lines 1, 4). Action `a0` involves the choice of a pair mu, sigma (lines 2, 3). These are drawn from a set of possible pairs of mean return from list `EF_mu` and standard deviation of return from list `EF_sig`. We have to normalize probabilities from line 4 in line 5. The probabilistic transition under the policy is then selected in line 6. We return the grid index of wealth in line 7.

```
1 def sampleWidx(w0,w1,I,a0): #to give the next state
2     mu = EF_mu[a0]
3     sig = EF_sig[a0]
4     p1 = norm.pdf((log(w1/(w0+I))-(mu-0.5*sig**2)*h)/(sig*sqrt(h
    )))
5     p1 = p1/sum(p1)  #normalize probabilities
6     idx = where(rand() > p1.cumsum())[0]
7     return len(idx)  #gives index of the wealth node w1 at t1
```

We may also easily replace the normal distribution with a t-distribution (or any other). For example, line 4 above would be replace with

```
p1 = t.pdf((log(w1/(w0+I))-(mu-0.5*sig**2)*h)/(sig*sqrt(h)),5)
```

where we see that the function `norm.pdf` is replaced with `t.pdf`—that is, a t-distribution with 5 degrees of freedom.

5. Temporal difference update at a single node. When we arrive at a node in the state space (indexed by `idx0`, `t0` in line 1 below), we then have to pick an action a0, which we do using the epsilon-greedy approach (lines 2–9). Under that action we will then call the preceding function to ascertain the next state `idx1,t1`. We then update the State–Action Value Function `Q[idx0,t0,a0]` in lines

10–16 if we are at $t < T$; or lines 17–19 if we are at $t = T$. Note that, in line 16, we choose the optimal policy at `t1`, as you can see the element `Q[idx1,t1,:].max()` in the code. In `TD` Learning, we update at every step in an episode, so it is easy to build all the update logic into a single generic function for one node, which we call `doOneNode(idx0,t0)` here.

```
1  def doOneNode(idx0,t0):  #idx0: index on the wealth axis, t0: index on the time axis
2      #Pick optimal action a0 using epsilon greedy approach
3      if rand() < epsilon:
4          a0 = randint(0,NP)    #index of action; or plug in best action from last step
5      else:
6          q = Q[idx0,t0,:]
7          a0 = where(q==q.max())[0] #Choose optimal Behavior policy
8          if len(a0)>1:
9              a0 = random.choice(a0) #randint(0,NP)    #pick randomly from multiple maximizing actions
10     #Generate next state S' at t+1, given S at t and action a0, and update State-Action Value Function Q(S,A)
11     t1 = t0 + 1
12     if t0<TT:  #at t<T
13         w0 = W[t0][idx0]  #scalar
14         w1 = W[t1]        #vector
15         idx1 = sampleWidx(w0,w1,infusions[t0],a0)   #Model-free transition
16         Q[idx0,t0,a0] = Q[idx0,t0,a0] + alpha*(R[idx0,t0,a0] + gamma*Q[idx1,t1,:].max() - Q[idx0,t0,a0])
17     else:  #at T
18         Q[idx0,t0,a0] = (1-alpha)*Q[idx0,t0,a0] + alpha*R[idx0,t0,a0]
19         idx1 = idx0
20     return [idx1,t1]  #gives back next state (index of W and t)
```

6. String together a sequence of calls to the previous function to generate updates through one episode, moving forward in time. At the beginning we set idx equal to the wealth index for initial wealth `W0`. The kernel of the code for one episode is just this. At every point in the episode, whichever state is visited experiences an update, and the entire `Q` table evolves into a new policy.

```
for t in range(TT+1):
    [idx,t] = doOneNode(idx,t)
```

7. We choose the number of episodes (epochs) as 105,000. Other parameters chosen are $\alpha = 0.1$, $\gamma = 1$, and $\epsilon = 0.3$. We initialize the `Q` tensor to zeros and then begin processing episode after episode. In order to examine if the algorithm is converging to a stable policy, we compute the sum of squared differences between Q tensors from consecutive episodes. At close to 50,000 epochs this metric becomes very small and stabilizes. Still, we run 55,000 more epochs to be assured of convergence.

In the next section, we present illustrative results from a numerical implementation of the Q-Learning algorithm.

7 Experimental results

We present some experimental results from running the Q-Learning algorithm in Table 1. The table shows the inputs to the problem, which are the mean vector of returns and the covariance matrix of returns. These are then used to compute the $K = 15$ portfolio that is available in the action space. The model outputs are the algorithm used, the number of training epochs, and the final value function outcome. We also offer extensions and observations. We recap some algorithm details and note the following:

1. Discussion of the epsilon-greedy algorithm: Q-Learning uses the epsilon-greedy algorithm in order to choose the action (or portfolio in this case), from the state $(W(t), t)$ at each step in the episode. This algorithm is as follows:

   - Sample x from the Bernoulli Distribution $B(\epsilon, 1 - e)$.
   - If $x < \epsilon$: Choose action a_k, $1 \leq k \leq K$ with probability $\frac{1}{K}$, *else:* Choose action $a = \arg\max_{1 \leq k \leq K} Q(W(t), t, a_k)$

   Most of the time this algorithm chooses the action that maximizes the Q value for small ϵ. However every once in a while it chooses an action uniformly from the set of available actions. This allows the Q-Learning algorithm to explore states and actions that it otherwise would not if it were to strictly follow the optimal policy. The value of ϵ has a significant effect on the working of the algorithm, and has to be at least 0.30 in order to get good results. This shows that without a sufficient amount of exploration, the algorithm may get stuck in states and actions that cause it to under estimate the Q values. Also note that the epsilon-greedy policy is being used here in an off-policy fashion, so that larger values of epsilon do not affect the accuracy of the Q values being estimated.
2. The results of Q-Learning algorithm are verified by comparing the value function at $t = 0$, given by $V(W(0), 0)$, with that computed using regular DP. Note that we can obtain V readily from Q by using the formula:

$$V(W(0), 0) = \max_{1 \leq k \leq K} Q(W(0), 0, a_k)$$

   By definition $V(W(0), 0)$ is the maximum expected reward when starting with an initial wealth of $W(0)$ at $t = 0$. In this case the expected reward corresponds to the probability of the final expected wealth value exceeding H. Applying regular Dynamic Programming to this problem yields $V(W(0), 0) = 0.72$, and we can see that Q-Learning also gives this answer after training for 100 K episodes, provided the value of the epsilon-greedy parameter is at least 0.30. Larger values of epsilon lead to greater exploration of the state space, which ultimately improves the accuracy of the Q values. However this comes at the cost of slower convergence, since the algorithm wanders over a larger number of states and actions. This can be seen in Table 1 for $\epsilon = 0.4$. In this case the algorithm converges to a good estimate of the optimal Q, but takes a larger number of iterations to do so. When the RL algorithm is run to a very high number of epochs, say 500 K, then it converges to the DP result, as seen in the last line of Table 1.
3. Choice of parameters (α, γ): The parameter γ is used in the Q-Learning algorithm as a discount factor for future rewards. Since the reward used in the GBWM problem formulation does not require any discounting, we set $\gamma = 1$. The parameter α is used to control the window of Q values that are averaged together.

Table 1 Results from the Q-Learning Algorithm. The parameters for these runs are as follows. The initial portfolio wealth is $W(0) = 100$; target portfolio goal = 200; horizon is $T = 10$ years. A total of 15 portfolios are used and these are generated from a mean vector of returns M and a covariance matrix of returns $\sum$ shown below, along with the mean and standard deviation of the portfolios' returns derived from M and $\sum$. The RL algorithm used the following parameters: $\alpha = 0.10$; $y = 1$. We assume zero infusions. The run time for 50 K epochs is ~1.5 minutes and for 100 K epochs is ~3 minutes. Dynamic programming, of course, takes 0.5 seconds. And the solution is provided in the top row of the bottom panel below.

MODEL INPUTS

$$M = \begin{bmatrix} 0.05 \\ 0.10 \\ 0.25 \end{bmatrix}; \quad \sum = \begin{bmatrix} 0.0025 & 0 & 0 \\ 0 & 0.04 & 0.02 \\ 0 & 0.02 & 0.25 \end{bmatrix}$$

	Portfolios							
	0	1	2	3	4	5	6	7
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							
	8	9	10	11	12	13	14	
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	

MODEL OUTPUTS

ϵ	No. of epochs	$V[W(0), t = 0]$
DP solution	1	0.72
0.10	50K	0.65
0.10	100K	0.65
0.20	50K	0.69
0.20	100K	0.71
0.25	100K	0.71
0.30	50K	0.72
0.30	100K	0.72
0.40	50K	0.73
0.40	100K	0.77
0.40	200K	0.75
0.40	500K	0.71

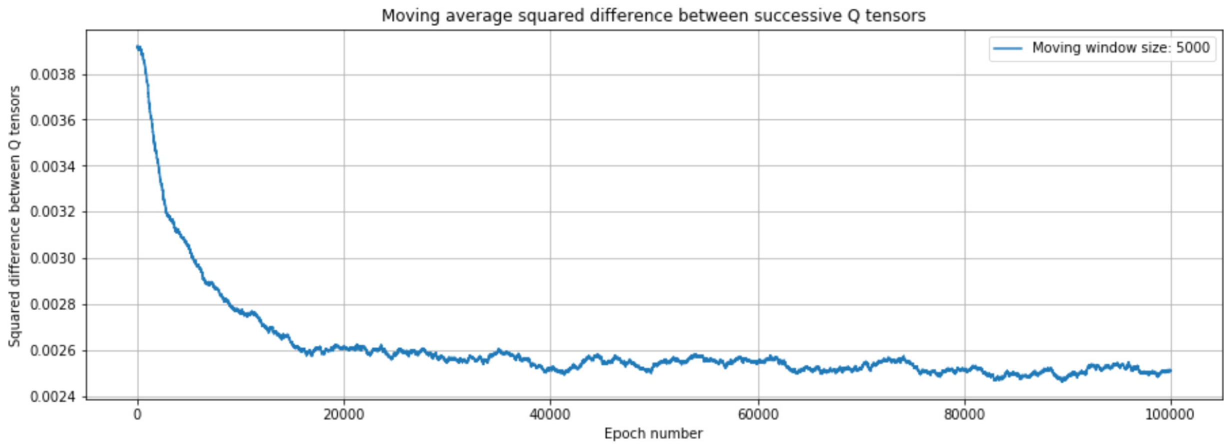

Figure 3 Convergence of the algorithm over successive epochs. The solution is reached in approximately 20,000 epochs.

Experimentally we observed that $\alpha = 0.1$ works quite well, which corresponds to a moving average over the last 10 Q values.

4. In order to measure the convergence of the algorithm, we plot the moving average squared difference between the Q-tensor from successive epochs. Figure 3 shows that the algorithm stabilizes in about 20,000 epochs.
5. Solving GBWM with other algorithms: There are a number of other algorithms that can be used to solve the GBWM problem. Since the state transition dynamics are specified to follow the geometric Brownian motion model, we can apply classical DP algorithms to this problem, as shown in Das *et al.* (2019). RL algorithms are needed for the following cases in which Dynamic Programming is not applicable:

   - The state transition dynamics are not known: In this case DP can no longer be used, however Q-Learning is still applicable provided there is a collection of sample paths that can be used for training.
   - The state space is not discretized: DP is difficult to implement numerically if we do not discretize the state space, and unfortunately the tabular Q-Learning algorithm does not work either. However, continuous states can be handled by deep Q-Learning using function approximators, or by the policy gradients algorithm. Likewise, continuous-time, continuous-space versions of the Bellman (1952) approach may be used for dynamic programming as in Merton (1971).
   - The results with the t-distribution are not much different than the normal. This suggests that the dynamic portfolio solution is robust to different distributional choices.

8 Concluding comments

DP may be used to solve multiperiod portfolio problems to reach desired goals with the highest possible probability. This is known as GBWM. This paper introduced RL as an alternate approach to solving the GBWM problem. In addition to providing a brief taxonomy of RL solution approaches, we also implemented one such approach, Q-Learning, and showed that we get the same results as DP. Our goal is to provide a quick introduction to how dynamic portfolios may be modeled using RL. The RL approach is highly extensible to larger state and action spaces. For example, if the action space (portfolios that may be chosen) varies based on whether the economy is in normal times or in a recession, then it adds the state of the economy as an additional dimension to the problem. This can be easily handled with RL. RL especially shines in comparison to DP when the problem becomes path-dependent, such as in the case of multiperiod portfolio optimization with taxes, when keeping track of the cost basis across portfolio holdings is required and this tax basis depends on the path of the portfolio, so that classic DP via backward recursion becomes computationally expensive from an explosion in the state space.

The recent advances in hardware and software for RL suggest great potential for finance applications that depend on dynamic optimization in stochastic environments whose transition probabilities are hard to estimate, such as high-frequency trading (HFT). HFT has been one of the areas of early investigation of RL in finance. There is a long history of papers implementing RL models for trading, such as Moody and Saffell (2001), Dempster and Leemans (2006), Li *et al.* (2007), Lu (2017), Du *et al.* (2018), and Zarkias *et al.* (2019). Additional areas in which RL may be used are option pricing (Halperin, 2017), optimal hedging of derivatives (Halperin, 2018), market-making agents (Halperin and Feldshteyn (2018), Zarkias *et al.* (2019), cryptocurrencies, optimal trade execution, etc.

Notes

[1] The latest (2018) version of this book is available here: http://incompleteideas.net/book/the-book-2nd.html.

[2] If you wish to implement the code, you will need to wrap these code ideas into a full Python program.

References

Bellman, R. (1952). "On the Theory of Dynamic Programming," *Proceedings of the National Academy of Sciences* **38**(8), 716–719.

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

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

Brunel, J. (2015). *Goals-Based Wealth Management: An Integrated and Practical Approach to Changing the Structure of Wealth Advisory Practices* (New York: Wiley).

Chhabra, A. B. (2005). "Beyond Markowitz: A Comprehensive Wealth Allocation Framework for Individual Investors," *The Journal of Wealth Management* **7**(4), 8–34.

Das, S. R., Markowitz, H., Scheid, H. J., and Statman, M. (2010). "Portfolio Optimization with Mental Accounts," *Journal of Financial and Quantitative Analysis* **45**(2), 311–334.

Das, S. R., 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. R., Ostrov, D., Radhakrishnan, A. and Srivastav, D. (2019). "A Dynamic Approach to Goals-Based Wealth Management," *Computational Management Science*, https://doi.org/10.1007/s10287-019-06351-7.

Dempster, M. A. H. and Leemans, V. (2006). "An Automated FX Trading System Using Adaptive Reinforcement Learning," *Expert Systems with Applications* **30**(3), 543–552.

Du, X., Zhai, J. and Koupin, L. (2018). "Algorithm Trading Using Q-Learning and Recurrent Reinforcement Learning," *Working Paper, Stanford University*.

Halperin, I. (2017). QLBS: Q-Learner in the Black-Scholes (-Merton) Worlds, SSRN Scholarly Paper ID 3087076, Social Science Research Network, Rochester, NY.

Halperin, I. (2018). The QLBS Q-Learner Goes NuQLear: Fitted Q Iteration, Inverse RL, and Option Portfolios, SSRN Scholarly Paper ID 3102707, Social Science Research Network, Rochester, NY.

Halperin, I. and Feldshteyn, I. (2018). Market Self-learning of Signals, Impact and Optimal Trading: Invisible Hand Inference with Free Energy (Or, How We Learned to Stop Worrying and Love Bounded Rationality). SSRN Scholarly Paper ID 3174498, Social Science Research Network, Rochester, NY.

Li, H., Dagli, C. H., and Enke, D. (2007). "Short-term Stock Market Timing Prediction under Reinforcement Learning Schemes," In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pp. 233–240.

Lu, D. W. (2017). "Agent Inspired Trading Using Recurrent Reinforcement Learning and LSTM Neural Networks," arXiv:1707.07338 [q-fin].

Markowitz, H. H. (1952). “Portfolio Selection,” *Journal of Finance* **6**, 77–91.

Merton, R. (1969). “Lifetime Portfolio Selection under Uncertainty: The Continuous-Time Case,” *The Review of Economics and Statistics* **51**(3), 247–257.

Merton, R. C. (1971). “Optimum Consumption and Portfolio Rules in a Continuous-Time Model,” *Journal of Economic Theory* **3**(4), 373–413.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). “Playing Atari with Deep Reinforcement Learning,” arXiv:1312.5602 [cs].

Moody, J. and Saffell, M. (2001). “Learning to Trade via Direct Reinforcement,” *IEEE Transactions on Neural Networks* **12**(4), 875–889.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). “Mastering the Game of Go without Human Knowledge,” *Nature* **550**(7676), 354–359.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction* (second edition edition ed.). Cambridge, Mass: A Bradford Book.

Zarkias, K. S., Passalis, N., Tsantekidis, A. and Tefas, A. (2019). “Deep Reinforcement Learning for Financial Trading Using Price Trailing,” In *ICASSP 2019—2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3067–3071.

Keywords: Reinforcement learning; goals; dynamic optimization.