Chapter16-Reinforcement-Learning

2023 年 11 月 21 日

1 Reinforcement Learning

Due to its increasing popularity within the Machine Learning community, we dedicate a chapter to reinforcement learning (RL). In 2022 only, more than 148 papers dedicated to RL have been submitted to (or updated on) arXiv under the **q:fin** (quantitative finance) classification. Moreover, an early survey of RL-based portfolios is compiled in Sato (2019) (see also Zhang, Zohren, and Roberts (2020)) and general financial applications are discussed in Kolm and Ritter (2019b), Meng and Khushi (2019), Charpentier, Elie, and Remlinger (2020) and Mosavi et al. (2020). This shows that RL has recently gained traction among the quantitative finance community.

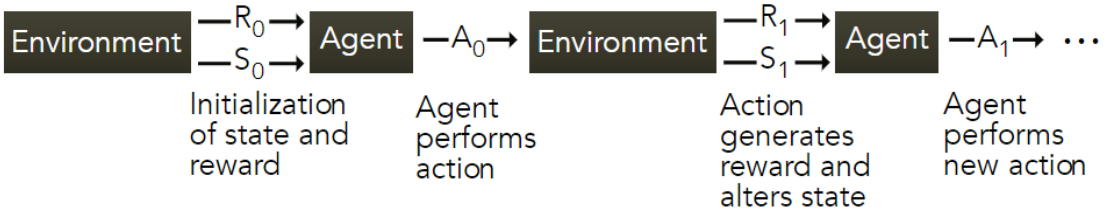
While RL is a framework much more than a particular algorithm, its efficient application in portfolio management is not straightforward, as we will show.

1.1 Theoretical Layout

1.1.1 General Framework

In this section, we introduce the core concepts of RL and follow relatively closely the notations (and layout) of Sutton and Barto (2018), which is widely considered as a solid reference in the field, along with Bertsekas (2017). One central tool in the field is called the **Markov Decision Process** (**MDP**, see Chapter 3 in Sutton and Barto (2018)).

MDPs, like all RL frameworks, involve the interaction between an **agent** (e.g., a trader or portfolio manager) and an **environment** (e.g., a financial market). The agent performs **actions** that may alter the state of environment and gets a reward (possibly negative) for each action. This short sequence can be repeated an arbitrary number of times, as is shown in Figure 16.1.



Given initialized values for the state of the environments (S_0) and reward (usually $R_0 = 0$), the agent performs an action (e.g., invests in some assets). This generates a reward R_1 (e.g., returns, profits, Sharpe Ratio) and also a future state of the environment (S_1). Based on that, the agent performs a new action and the sequence continues. When the sets of states, actions and rewards are finite, the MDP is logically called *finite*. In a financial framework, this is somewhat unrealistic and we discuss this issue later on. It nevertheless is not hard to think of simplified and discretized financial problems. For instance, the reward can be binary: win money versus lose money. In the case of only one asset, the action can also be dual: investing versus not investing. When the number of assets is sufficiently small, it is possible to set fixed proportions that lead to a reasonable number of combinations of portfolio choices, etc.

We pursue our exposé with *finite* MDPs; they are the most common in the literature and their formal treatment is simpler. The relative simplicity of MDPs helps grasp the concepts that are common to other RL techniques. As is often the case with Markovian objects, the key notion is that of transition probability:

$$p(s', r|s, a) = \mathbb{P}\left[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right]$$

which is the probability of reaching state s' and reward r and time t, conditionally on being in state s and performing action a at time $t-1$. The finite sets of states and actions will be denoted with $\mathcal{S}$ and $\mathcal{A}$ henceforth. Sometimes, this probability is averaged over the set of rewards which gives the following decomposition:

$$\sum_r rp(s', r|s, a) = \mathcal{P}^a_{ss}\mathcal{R}^a_{ss}, \quad \text{where}$$

$$\mathcal{P}^a_{ss} = \mathbb{P}[S_t = s'|S_{t-1} = s, A_{t-1} = a], \quad \text{and}$$
$$\mathcal{R}^a_{ss} = \mathbb{E}[R_t|S_{t-1} = s, S_t = s', A_{t-1} = a].$$

The goal of the agent is to maximize some function of the stream of rewards. This gain is usually defined as

$$\begin{aligned} G_t &= \sum_{k=0}^{T} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned}$$

i.e., it is a *discounted* version of the reward, where the discount factor is $\gamma \in (0, 1]$. The horizon T may be infinite, which is why γ was originally introduced. Assuming the rewards are bounded, the infinite sum may diverge for $\gamma = 1$. This is the case if rewards don't decrease with time and there is no reason why they should. When $\gamma < 1$ and rewards are bounded, convergence is assured. When T is finite, the task is called *episodic* and, otherwise, it is said to be *continuous*.

In RL, the focal unknown to be optimized or learned is the **policy** π, which drives the actions of the agent. More precisely, $\pi(a, s) = \mathbb{P}[A_t = a|S_t = s]$, that is π equals the probability of taking action a if the state of the environment is s. This means that actions are subject to randomness, just like for *mixed strategies* in game theory. While this may seem disappointing because an investor would want to be sure to take the best action, it is also a good reminder that the best way to face random outcomes may well be to randomize actions as well.

Finally, in order to tty to deterimine the *best* policy, one key indicator is the so-called *value* function:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

where the time index t is not very relevent and omitted in the notation of the function. The index π under the expectation operator $\mathbb{E}[\cdot]$ simply indicates that the average is taken when the policy π is enforced. The value function is simply equal to the average gain conditionally on the state being equal to s. In financial terms, this is equivalent to the average profit if the agent takes actions driven by π when the market environment is s. More generally, it is also possible to condition not only on the state, but also on the action taken. We thus introduce the q_π *action-value function*:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

The q_π function is highly important because it gives the average gain when the state and action are fixed. Hence, if the current state is known, then one obvious choice is to select the action for which $q_\pi(s, \cdot)$ is the highest. Of course, this is the best solution if the optimal value of q_π is known, which is not always the case in practice. The value function can easily be accessed via $q_\pi : v_\pi(s) = \sum_a \pi(a, s) q_\pi(s, a)$.

The optimal v_π and q_π are straightforwardly defined as

$$v_*(s) = \max_\pi v_\pi(s), \ \forall s \in \mathcal{S}, \quad \text{and} \quad q_*(s, a) = \max_\pi q_\pi(s, a), \ \forall (s, a) \in \mathcal{S} \times \mathcal{A}.$$

If only $v_*(s)$ is known, then the agent must span the set of actions and find those that yield the maximum value for any given state s.

Finding these optimal values is a very complicated task and many articles are dedicated to solving this challenge. One reason why finding the best $q_\pi(s, a)$ is difficult is because it depends on two elements (s and a) on one side and π on the other. Usually, for a fixed policy π, it can be time

consuming to evaluate $q_\pi(s,a)$ for a given stream of actions, states and rewards. Once $q_\pi(s,a)$ is estimated, then a new policy π' must be tested and evaluated to determine if it is better than the original one. Thus, this iterative search for a good policy can take long. For more details on policy improvement and value function updating, we recommend chapter 4 of Sutton and Barto (2018) which is dedicated to dynamic programming.

1.1.2 Q-learning

An interesting shortcut to the problem of finding $v_*(s)$ and $q_*(s,a)$ is to remove the dependence on the policy. Consequently, there is then of course no need to iteratively improve it. The central relationship that is required to do this is the so-called *Bellman equation* that is satisfied by $q_\pi(s,a)$. We detail its derivation below. First of all, we recall that

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a], \end{aligned}$$

The expression $\mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a]$ can be further decomposed. Since the expectation runs over π, we need to sum over all possible actions a' and states s' and resort to $\pi(a', s')$. In addition, the sum on the s' and r arguments of the probability $p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]$ gives access to the distribution of the random couple (S_{t+1}, R_{t+1}) so that in the end $\mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a] = \sum_{a',r,s'} \pi(a', s')p(s', r|s, a)r$. A similar reasoning applies to the second portion of q_π and:

$$\begin{aligned} q_\pi(s,a) &= \sum_{a',r,s'} \pi(a', s')p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s', A_t = a']\right] \\ &= \sum_{a',r,s'} \pi(a', s')p(s', r|s, a) \left[r + \gamma q_\pi(s', a')\right]. \end{aligned}$$

This equation links $q_\pi(s,a)$ to the future $q_\pi(s', a')$ from the states and actions (s', a') that are accessible from (s,a).

Notably, the previous equation is also true for the optimal action-value function $q_* = \max_\pi q_\pi(s,a)$:

$$\begin{aligned} q_*(s,a) &= \max_{a'} \sum_{r,s'} p(s', r|s, a) \left[r + \gamma q_*(s', a')\right], \\ &= \mathbb{E}_{\pi^*}[r|s,a] + \gamma \sum_{r,s'} p(s', r|s, a) \left(\max_{a'} q_*(s', a')\right) \end{aligned}$$

because one optimal policy is one that maximizes $q_\pi(s,a)$, for a given state s and over all possible actions a. This expression is central to a cornerstone algorithm in RL called Q-learning (the formal proof of convergence is outlined in Watkins and Dayan (1992)). In Q-learning, the state-action function no longer depends on the policy and is written with capital Q. The process is the following:

Initialize values $Q(s,a)$ for all states s and actions a. For each episode:

$$(\mathbf{QL}) \quad \begin{cases} 0.\ \text{Initialize state } S_0 \text{ and for each iteration } i \text{ until the end of the episode;} \\ 1.\ \text{observe state } s_i; \\ 2.\ \text{perform action } a_i \text{(depending on } Q\text{)}; \\ 3.\ \text{receive reward } r_{i+1} \text{ and observe state } s_{i+1}; \\ 4.\ \text{Update } Q \text{ as follows:} \end{cases}$$

$$Q_{i+1}(s_i,a_i) \longleftarrow Q_i(s_i,a_i) + \eta \left(\underbrace{r_{i+1} + \gamma \max_a Q_i(s_{i+1},a)}_{\text{echo of (16.7)}} - Q_i(s_i,a_i) \right)$$

The underlying reason this update rule works can be linked to fixed point theorems of contraction mappings. If a function f satisfies $|f(x) - f(y)| < \delta|x-y|$ (Lipshitz continuity), then a fixed point z satisfying $f(z) = z$ can be iteratively obtained via $z \leftarrow f(z)$. This updating rule converges to the fixed point. The previous equation can be solved using a similar principle, except that a learning rate η slows the learning process but also technically ensures convergence under technical assumptions.

More generally, (16.8) has a form that is widespread in reinforcement learning that is summarized in Equation (2.4) of Sutton and Barto (2018):

$$\text{New estimate} \leftarrow \text{Old estimate} + \text{Step size } (i.e.,\ \text{learning rate}) \times (\text{Target - Old estimate})$$

where the last part can be viewed as an error term. Starting from the old estimate, the new estimate therefore goes in the 'right' (or sought) direction, modulo a discount term that makes sure that the magnitude of this direction is not too large. The update rule is often referred to as '*temporal difference*' learning because it is driven by the improvement yielded by estimates that are known at time $t+1$ (target) versus those known at time t.

One important step of the Q-learning sequence (**QL**) is the second one where the action a_i is picked. In RL, the best algorithms combine two features: **exploitation** and **exploration**. Exploitation is when the machine uses the current information at its disposal to choose the next action. In this case, for a given state s_i, it chooses the action a_i that maximizes the expected reward $Q_i(s_i,a_i)$. While obvious, this choice is not optimal if the current function Q_i is relatively far from the *true* Q. Repeating the locally optimal strategy is likely to favor a limited number of actions, which will narrowly improve the accuracy of the Q function.

In order to gather new information stemming from actions that have not been tested much (but that can potentially generate higher rewards), **exploration** is needed. This is when an action a_i is chosen randomly. The most common way to combine these two concepts is called ϵ-greedy

exploration. The action a_i is assigned according to:

$$a_i = \begin{cases} \underset{a}{\text{argmax}}\, Q_i(s_i, a) & \text{with probability } 1-\epsilon \\ \text{randomly (uniformly) over } \mathcal{A} & \text{with probability } \epsilon \end{cases}$$

Thus, with probability ϵ, the algorithm explores and with probability $1-\epsilon$, it exploits the current knowledge of the expected reward and picks the best action. Because all actions have a non-zero probability of being chosen, the policy is called "soft". Indeed, the best action has a probability of selection equal to $1-\epsilon(1-\text{card}(\mathcal{A})^{-1})$, while all other actions are picked with probability $\epsilon/\text{card}(\mathcal{A})$.

1.1.3 SARSA

In Q-learning, the algorithm seeks to find the action value function of the optimal policy. Thus, the policy that is followed to pick actions is different from the one that is learned (via Q). Such algorithms are called *off-policy*. *On-policy* algorithms seek to improve the estimation by continuously acting *according to the policy* π. One canonical example of on-policy learning is the SARSA method which requires two consecutive states and actions **SARSA**. The way the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ is processed is presented below.

The main difference between Q-learning and SARSA is the update rule. In SARSA, it is given by

$$Q_{i+1}(s_i, a_i) \longleftarrow Q_i(s_i, a_i) + \eta \left(r_{i+1} + \gamma\, Q_i(s_{i+1}, a_{i+1}) - Q_i(s_i, a_i) \right)$$

The improvement comes only from the **local** point $Q_i(s_{i+1}, a_{i+1})$ that is based on the new states and actions (s_{i+1}, a_{i+1}), whereas in Q-learning, it comes from all possible actions which only the best is retained $\max_a Q_i(s_{i+1}, a)$.

A more robust but also more computationally demanding version of SARSA is *expected* SARSA in which the target Q function is averaged over all actions:

$$Q_{i+1}(s_i, a_i) \longleftarrow Q_i(s_i, a_i) + \eta \left(r_{i+1} + \gamma \sum_a \pi(a, s_{i+1}) Q_i(s_{i+1}, a) - Q_i(s_i, a_i) \right)$$

Expected SARSA is *less volatile* than SARS because the latter is strongly impacted by the random choice of a_{i+1}. In expected SARSA, the average smoothes the learning process.

1.2 The Curse of Dimensionality

Let us first recall that reinforcement learning is a framework that is not linked to a particular algorithm. In fact, different tools can very well co-exist in a RL task (AlphaGo combined both tree methods and neural networks, see Silver et al. (2016)). Nonetheless, any RL attempt will always rely on the three key concepts: the states, actions and rewards. In factor investing, they are fairly easy to identify, though there is always room for interpretation: - Actions are evidently

defined by *portfolio compositions*. - The states can be viewed as the current values that describe the economy: as a first-order approximation, it can be assumed that the *feature* levels fulfill this role (possibly conditioned or complemented with macro-economic data). - The rewards are even more straightforward. *Returns* or any relevant performance metric can account for rewards.

A major problem lies in the dimensionality of both states and actions. Assuming an absence of leverage (no negative weights), the actions take values on the simplex

$$\mathbb{S}_N = \left\{ \mathbf{x} \in \mathbb{R}^N \middle| \sum_{n=1}^{N} x_n = 1,\ x_n \geq 0,\ \forall n = 1, \dots, N \right\}$$

and assuming that all features have been uniformized, their space is $[0,1]^{NK}$. Needless to say, the dimensions of both spaces are numerically impractical.

A simple solution to this problem is *discretization*: each space is divided into a small number of categories. Some authors do take this route. In Yang, Yu, and Almahdi (2018), the state space is discretized into three values depending on volatility, and actions are also split into three categories. Bertoluzzo and Corazza (2012) and Xiong et al. (2018) also choose three possible actions (buy, hold, sell). In Almahdi and Yang (2019), the learner is expected to yield binary signals for buying or shorting. García-Galicia, Carsteanu, and Clempner (2019) consider a larger state space (8 elements) but restrict the action set to 3 options. In terms of the state space, all articles assume that the state of the economy is determined by prices (or returns).

One strong limitation of these approaches is the marked simplification they imply. Realistic discretizations are numerically intractable when investing in multiple assets. Indeed, splitting the unit interval in h points yields h^{NK} possibilities for feature values. The number of options for weight combinations is exponentially increasing N. As an example: just 10 possible values for 10 features of 10 stocks yield 10^{100} permutations.

The problems mentioned above are of course not restricted to portfolio construction. Many solutions have been proposed to solve Markov Decision Processes in continuous spaces. We refer for instance to Section 4 in Powell and Ma (2011) for a review of early methods (outside finance).

This curse of dimensionality is accompanied by the fundamental question of training data. Two options are conceivable: *market data versus simulations*. Under a given controlled generator of samples, it is hard to imagine that the algorithm will beat the solution that maximizes a given utility function. If anything, it should converge towards the static optimal solution under a stationary data generating process (see, e.g., Chaouki et al. (2020) for trading tasks), which is by the way a very strong modelling assumption.

This leaves market data as a preferred solution but even with large datasets, there is little chance to cover all the (actions, states) combinations mentioned above. Characteristics-based datasets

have depths that run through a few decades of monthly data, which means several hundreds of time-stamps at most. This is by far too limited to allow for a reliable learning process. It is always possible to generate synthetic data (as in Yu et al. (2019)), but it is unclear that this will solidly improve the performance of the algorithm.

1.3 Policy Gradient

1.3.1 Principle

Beyond the discretization of action and state spaces, a powerful trick is **parametrization**. When a and s can take discrete values, action-value functions must be computed for all pairs (a, s), which can be prohibitively cumbersome. An elegant way to circumvent this problem is to assume that the policy is driven by a relatively modest number of *parameters*. The learning process is then focused on optimizing this set of parameters $\boldsymbol{\theta}$. We then write $\pi_{\boldsymbol{\theta}}(a, s)$ for the probability of choosing action a in state s. One intuitive way to define $\pi_{\boldsymbol{\theta}}(a, s)$ is to resort to a *softmax* form:

$$\pi_{\boldsymbol{\theta}}(a, s) = \frac{e^{\boldsymbol{\theta}' \mathbf{h}(a,s)}}{\sum_b e^{\boldsymbol{\theta}' \mathbf{h}(b,s)}}$$

where the output of function $\mathbf{h}(a, s)$, which has the same dimension as $\boldsymbol{\theta}$ is called a feature vector representing the pair (a, s). Typically, $\mathbf{h}$ can very well be a simple neural network with two input units and an output dimension equal to the length of $\boldsymbol{\theta}$.

One desired property for $\pi_{\boldsymbol{\theta}}$ is that it can be *differentiable* w.r.t. $\boldsymbol{\theta}$ so that $\boldsymbol{\theta}$ can be improved via some gradient method. The most simple and intuitive results about policy gradients are known in the case of episodic tasks (finite horizon) for which it is sought to maximize the average gain $\mathbb{E}_{\boldsymbol{\theta}}[G_t]$ where the gain is defined as $G_t = R_{t+1} + \gamma G_{t+1}$. The expectation is computed according to a particular policy that depends on $\boldsymbol{\theta}$, this is why we use a simple subscript. One central result is the so-called policy gradient theorem which states that

$$\nabla \mathbb{E}_{\boldsymbol{\theta}}[G_t] = \mathbb{E}_{\boldsymbol{\theta}} \left[G_t \frac{\nabla \pi_{\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}}} \right]$$

This result can then be used for **gradient ascent**: when seeking to maximize a quantity, the parameter change must go in the upward direction:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla \mathbb{E}_{\boldsymbol{\theta}}[G_t]$$

This simple update rule is known as the **REINFORCE** algorithm. One improvement of this simple idea is to add a baseline, and we refer to section 13.4 of Sutton and Barto (2018) for a detailed account on this topic.

1.3.2 Extensions

A popular extension of REINFORCE is the so-called **actor-critic** (AC) method which combines policy gradient with Q- or v-learning. The AC algorithm can be viewed as some kind of mix between policy gradient and SARSA. A central requirement is that the state-value function $v(\cdot)$ be a differentiable functioni of some parameter vector $\boldsymbol{w}$ (often taken to be a neural network). The update rule is then

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \left(R_{t+1} + \gamma v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}) \right) \frac{\nabla \pi_{\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}}}$$

but the trick is that the vector $\mathbf{w}$ must also be updated. The actor is the policy side which is what drives decision making. The critic side is the value function that evaluates the actor's performance. As learning progresses (each time both sets of parameters are updated), both sides improve. The exact algorithmic formulation is a bit long and we refer to Section 13.5 in Sutton and Barto (2018) for the precise sequence of steps of AC.

Another interesting application of parametric policies is outlined in Aboussalah and Lee (2020). In their article, the authors define a trading policy that is based on a recurrent neural network. Thus, the parameter $\boldsymbol{\theta}$ in this case encompasses all weights and biases in the network.

Another favorable feature of parametric policies is that they are compatible with continuous sets of actions. Beyond the softmax form, there are other ways to shape $\pi_{\boldsymbol{\theta}}$. If $\mathcal{A}$ is a subset of $\mathbb{R}$, and $f_{\boldsymbol{\Omega}}$ us a density function with parameters $\boldsymbol{\Omega}$, then a candidate form for $\pi_{\boldsymbol{\theta}}$ is

$$\pi_{\boldsymbol{\theta}} = f_{\boldsymbol{\Omega}(s,\boldsymbol{\theta})}(a)$$

in which the parameters $\boldsymbol{\Omega}$ are in turn functions of the states and of the underlying (second order) parameters $\boldsymbol{\theta}$.

While the *Gaussian distribution* (see section 13.7 in Sutton and Barto (2018)) is often a preferred choice, they would require some processing to lie inside the unit interval. One easy way to obtain such values is to apply the normal *cumulative distribution function* to the output. In Wang and Zhou (2019), the multivariate Gaussian policy is theoretically explored, but it assumes no constraint on weights.

Some natural parametric distributions emerge as alternatives. If only one asset is traded, then the *Bernoulli* distribution can be used to determine whether or not to buy the asset. If a riskless asset is available, the beta distribution offers more flexibility because the values for the proportion invested in the risky asset span the whole interval; the remainder can be invested into the safe asset. When many assets are traded, things become more complicated because of the budget constraint. One ideal candidate is the *Dirichlet* distribution because it is defined on a simplex:

$$f_{\boldsymbol{\alpha}}(w_1, \ldots, w_n) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{n=1}^{N} w_n^{\alpha_n - 1}$$

where $B(\boldsymbol{\alpha})$ is the multinomial beta function:

$$B(\boldsymbol{\alpha}) = \frac{\prod_{n=1}^{N} \Gamma(\alpha_n)}{\Gamma\left(\sum_{n=1}^{N} \alpha_n\right)}$$

If we set $\pi = \pi_{\boldsymbol{\alpha}} = f_{\boldsymbol{\alpha}}$, the link with factors or characteristics can be coded through $\boldsymbol{\alpha}$ via a linear form:

$$(\mathbf{F1}) \quad \alpha_{n,t} = \theta_{0,t} + \sum_{k=1}^{K} \theta_t^{(k)} x_{t,n}^{(k)}$$

which is highly tractable, but may violate the condition that $\alpha_{n,t} > 0$ for some values of $\theta_{k,t}$. Indeed, during the learning process, an update in $\boldsymbol{\theta}$ might yield values that are out of the feasible set of $\boldsymbol{\alpha}_t$. In this case, it is possible to resort to a trick that is widely used in *online learning*. The idea is simply to find the acceptable solution that is closest to the suggestion from the algorithm. If we call $\boldsymbol{\theta}^*$ the result of an update rule from a given algorithm, then the closest feasible vector is

$$\boldsymbol{\theta} = \min_{\mathbf{z} \in \Theta(\mathbf{x}_t)} ||\boldsymbol{\theta}^* - \mathbf{z}||^2$$

where $||\cdot||$ is the Euclidean norm and $\Theta(\mathbf{x}_t)$ is the feasible set, that is, the set of vectors $\boldsymbol{\theta}$ such that $\alpha_{n,t} = \theta_{0,t} + \sum_{k=1}^{K} \theta_t^{(k)} x_{t,n}^{(k)}$ are all non-negative.

A second option for the form of the policy, $\pi^2_{\boldsymbol{\theta}_t}$, is slightly more complex but remains always valid (i.e., has positive $\alpha_{n,t}$ values):

$$(\mathbf{F2}) \quad \alpha_{n,t} = \exp\left(\theta_{0,t} + \sum_{k=1}^{K} \theta_t^{(k)} x_{t,n}^{(k)}\right)$$

which is simply the exponential of the first version. With some algebra, it is possible to derive the policy gradients. The policies $\pi^j_{\boldsymbol{\theta}_t}$ are defined by the Equations (**Fj**) above. Let denote the *digamma* function. Let 1 denote the $\mathbb{R}^N$ vector of all ones. We have

$$\frac{\nabla_{\boldsymbol{\theta}_t} \pi^1_{\boldsymbol{\theta}_t}}{\pi^1_{\boldsymbol{\theta}_t}} = \sum_{n=1}^{N} \left(\left(\mathbf{1}' \mathbf{X}_t \boldsymbol{\theta}_t\right) - \left(\mathbf{x}_{t,n} \boldsymbol{\theta}_t\right) + \ln w_n \right) \mathbf{x}'_{t,n}$$

$$\frac{\nabla_{\boldsymbol{\theta}_t} \pi^2_{\boldsymbol{\theta}_t}}{\pi^2_{\boldsymbol{\theta}_t}} = \sum_{n=1}^{N} \left(\left(\mathbf{1}' e^{\mathbf{X}_t \boldsymbol{\theta}_t}\right) - \left(e^{\mathbf{x}_{t,n} \boldsymbol{\theta}_t}\right) + \ln w_n \right) e^{\mathbf{x}_{t,n} \boldsymbol{\theta}_t} \mathbf{x}'_{t,n}$$

where $e^{\mathbf{X}}$ is the element-wise exponential of a matrix $\mathbf{X}$.

The allocation can then either be made by direct sampling, or using the mean of the distribution $(1'\boldsymbol{\alpha})^{-1}\boldsymbol{\alpha}$. Lastly, a technical note: Dirichlet distributions can only be used for *small* portfolios because the scaling constant in the density becomes numerically intractable for large values of N (e.g., above 50).

1.4 Simple Examples

1.4.1 Q-learning with Simulations

To illustrate the gist of the problems mentioned above, we propose two implementations of Q-learning. For simplicity, the first one is based on simulations. This helps understand the learning process in a simplified framework.

We consider two assets: one risky and one riskless, with return equal to zero. The returns for the risky process follow an autoregressive model of order one (AR(1)): $r_{t+1} = a + \rho r_t + \epsilon_{t+1}$ with $|\rho| < 1$ and ϵ following a standard white noise with variance σ^2. In practice, individual (monthly) returns are seldom autocorrelated, but adjusting the autocorrelation helps understand if the algorithm learns correctly (see exercise below).

The environment consists only in observing the past return r_t. Since we seek to estimate the Q function, we need to discretize this state variable. The simplest choice is to resort to a binary variable: equal to -1 (negative) if $r_t < 0$ and to +1 (positive) if $r_t \geq 0$. The actions are summarized by the quantity invested in the risky asset. It can take 5 values: 0 (risk-free portfolio), 0.25, 0.5, 0.75 and 1 (fully invested in the risky asset). This is for instance the same choice as in Pendharkar and Cusatis (2018).

We will hard-code the simulation and the Q-learning process in Python. It requires a dataset with the usual inputs: state, action, reward and subsequent state. We start by simulating the returns: they drive the states and the rewards (portfolio returns). The actions are sampled randomly. The data is built in the chunk below.

[ ]:
```
from statsmodels.tsa.arima_process import ArmaProcess
import pandas as pd
import numpy as np

n_sample = int(1e5)
rho = 0.8
sd = 0.4
a = 0.06 * rho

ar1_process = ArmaProcess(ar=np.array([1, -rho]))
returns = a / rho + ar1_process.generate_sample(nsample=n_sample, scale=sd)
action = np.round(np.random.uniform(0, 1, n_sample) * 4) / 4

data_RL = pd.DataFrame([returns, action], index=['returns', 'action']).T
```

```
data_RL['new_state'] = data_RL['returns'].apply(lambda x: 'neg' if x < 0 else␣
↪'pos')
data_RL['reward'] = data_RL['returns'] * data_RL['action']
data_RL['state'] = data_RL['new_state'].shift(1)
data_RL = data_RL.dropna() # remove the last sample where the new_state is N/A
data_RL
```

[ ]:

	returns	action	new_state	reward	state
1	-0.375426	0.50	neg	-0.187713	pos
2	-0.975589	0.75	neg	-0.731691	neg
3	0.144108	0.50	pos	0.072054	neg
4	0.663847	0.00	pos	0.000000	pos
5	-0.081382	0.25	neg	-0.020346	pos
…	…	…	…	…	…
99995	-0.546845	0.00	neg	-0.000000	neg
99996	-1.126224	0.50	neg	-0.563112	neg
99997	-1.185559	0.50	neg	-0.592780	neg
99998	-1.016677	0.75	neg	-0.762508	neg
99999	-0.859250	0.00	neg	-0.000000	neg

[99999 rows x 5 columns]

There are 3 parameters in the implementation of the Q-learning algorithms:

- η, which is the learning rate in the Q-learning process.
- γ, the discounting rate for the rewards;
- and ϵ, which controls the rate of exploration versus exploitation.

[ ]:
```
def q_learning_sim(eta, gamma, epsilon, data_RL: pd.DataFrame):

    records = data_RL.to_dict(orient='records')
    actions = data_RL.action.unique()
    states = data_RL.state.unique()
    fit_RL = {state: {action: 0 for action in actions} for state in states} #␣
↪Initialization of Q
    r_final = 0

    for record in records:
```

```
        cur_state, cur_action = record['state'], record['action'] # s_i, a_i
        new_state = record['new_state'] # s_{i+1}
        if(np.random.random() <= epsilon): # exploration
            best_action = np.random.choice(actions) # random choice over A
            best_Q = fit_RL[new_state][best_action]
        else: # exploitation
            best_Q = max(fit_RL[new_state].values()) # max Q_i(s_{i+1}, a)

        fit_RL[cur_state][cur_action] += eta * (record['reward'] + gamma *␣
↪best_Q - fit_RL[cur_state][cur_action]) # Q-learning
        r_final += record['reward']

    return fit_RL, r_final

eta = 0.1
gamma = 0.7
epsilon = 0.1
fit_RL, r_final = q_learning_sim(eta, gamma, epsilon, data_RL)
Q_func = pd.DataFrame.from_dict(fit_RL).T.sort_index(axis=1)
display(Q_func)
print(f'Reward (last iteration): {r_final}')
```

```
         0.00      0.25      0.50      0.75      1.00
pos  0.658828  0.739241  0.999412  0.763612  1.072064
neg  0.397690  0.286704  0.072400  0.094711 -0.057815

Reward (last iteration): 2257.339198401797
```

The output shows the Q function, which depends naturally both on states and actions. When the state is negative, large risky positions (action equal to 0.75 or 1.00) are associated with the smallest average rewards (even negative). When the state is positive, the average rewards are the highest for the largest allocations. The rewards in both cases are almost a monotonic function of the proportion invested in the risky asset. Thus, the recommendation of the algorithm (i.e., the policy) is to be fully invested in a positive state and to refrain from investing in a negative state. Given the positive autocorrelation of the underlying process, this does make sense.

Basically, the algorithm has simply learned that positive (*resp.* negative) returns are more likely to follow positive (*resp.* negative) returns. While this is somewhat reassuring, it is by no means impressive, and much simpler tools would yield similar conclusions and guidance.

1.4.2 Q-learning with Market Data

The second application is based on the financial dataset. To reduce the dimensionality of the problem, we will assume that:

- Only *one* feature (price-to-book ratio) captures the state of the environment. This feature is processed so that is has only a limited number of possible values;
- Actions take values over a discrete set consisting of three positions: +1 (buy the market), -1 (sell the market) and 0 (hold no risky positions);
- Only *two* assets are traded: those with `stock_id` 3 and 4 - they both have 245 days of trading data.

The construction of the dataset is unelegantly coded below.

[ ]:
```
data_ml = pd.read_pickle('./data/data_ml.pkl')

return_3 = data_ml[data_ml['stock_id'] == 3]['R1M_Usd'].values
return_4 = data_ml[data_ml['stock_id'] == 4]['R1M_Usd'].values
pb_3 = data_ml[data_ml['stock_id'] == 3]['Pb'].values
pb_4 = data_ml[data_ml['stock_id'] == 4]['Pb'].values
action_3 = np.floor(np.random.uniform(0, 1, len(pb_3)) * 3) - 1
action_4 = np.floor(np.random.uniform(0, 1, len(pb_4)) * 3) - 1

RL_data = pd.DataFrame([return_3, return_4, pb_3, pb_4, action_3, action_4],␣
 ↪index=['return_3', 'return_4', 'pb_3', 'pb_4', 'action_3', 'action_4']).T
RL_data['action'] = RL_data['action_3'].astype('int64').apply(str) + " " +␣
 ↪RL_data['action_4'].astype('int64').apply(str)
RL_data['pb_3'] = np.round(5 * RL_data['pb_3']) # simplifying
RL_data['pb_4'] = np.round(5 * RL_data['pb_4']) # simplifying
RL_data['state'] = RL_data['pb_3'].astype('int64').apply(str) + " " +␣
 ↪RL_data['pb_4'].astype('int64').apply(str)
RL_data['reward'] = RL_data['action_3'] * RL_data['return_3'] +␣
 ↪RL_data['action_4'] * RL_data['return_4']
RL_data['new_state'] = RL_data['state'].shift(-1)

RL_data = RL_data[['action', 'state', 'reward', 'new_state']][:-1]
RL_data
```

[ ]:

	action	state	reward	new_state
0	-1 1	1 1	-0.093	1 1
1	1 1	1 1	-0.024	1 1
2	1 -1	1 1	-0.135	1 1
3	1 -1	1 1	0.038	1 1
4	1 -1	1 1	0.036	1 1
..	…	…	…	…
222	1 -1	1 2	0.037	1 2
223	-1 0	1 2	0.049	1 2
224	-1 -1	1 2	0.095	1 2
225	0 0	1 2	0.000	1 2
226	1 0	1 2	-0.070	1 3

[227 rows x 4 columns]

Actions and states have to be merged to yield all possible combinations. To simplify the states, we round 5 times the price-to-book ratios.

We keep the same hyperparameters as in the previous example. Columns below stand for actions: the first (*resp.* second) number notes the position in the first (*resp.* second) asset. The rows correspond to states.

[ ]:
```
fit_RL, r_final = q_learning_sim(eta, gamma, epsilon, RL_data)
Q_func = pd.DataFrame.from_dict(fit_RL).T.sort_index(axis=1)
# display(Q_func.style.format(precision=4))
print(Q_func.to_string(float_format=lambda x: '{:.4f}'.format(x)))
print(f'Reward (last iteration): {r_final}')
```

	-1 -1	-1 0	-1 1	0 -1	0 0	0 1	1 -1	1 0	1 1
1 1	0.0071	0.0346	0.0371	-0.0017	0.0126	0.0154	0.0078	0.0182	0.0461
2 1	-0.0231	0.0027	0.0399	0.0077	0.0046	0.0120	0.0051	0.0025	-0.0082
2 2	-0.0001	0.0116	0.0000	0.0000	0.0000	0.0039	-0.0001	-0.0010	-0.0238
3 1	-0.0006	0.0052	0.0000	0.0000	0.0004	0.0000	0.0000	0.0000	0.0000
1 2	0.0137	0.0017	0.0422	0.0118	0.0140	0.0241	0.0212	0.0290	0.0073
2 3	0.0326	0.0000	0.0000	-0.0042	0.0000	0.0000	0.0000	0.0000	0.0000
1 3	0.0000	-0.0041	0.0094	0.0040	0.0015	0.0004	0.0000	0.0086	0.0081
0 3	0.0000	0.0000	0.0000	0.0093	0.0000	0.0000	0.0119	0.0000	0.0000
0 2	0.0000	0.0000	-0.0110	0.0000	0.0000	-0.0041	0.0000	0.0000	0.0000

Reward (last iteration): 2.141999999999999

The output shows that there are many combinations of states and actions that are not spanned by the data: basically, the Q function has a zero and it is likely that the combination has not been explored. Some states seem to be more often represented(`1 1`, `1 2` and `1 1`), others, less(`3 1`, `2 3`). It is hard to make any sense of the recommendations. Some states close (like `1 1` and `1 2`) but the outcomes are very different. Moreoever, there is no coherence and no monotonicity in actions with respect to individual state values: low values of states can be associated to very different actions.

One reason why these conclusions do not appear trustworthy pertains to the data size. With only 200+ time points and 99 state-action pairs (11 times 9), this yields on average only two data points to compute the Q function. This could be improved by testing more random actions, but the limits of the sample size would eventually (rapidly) be reached anyway. This is left as an exercise (see below).

1.5 Concluding Remarks

Reinforcement learning has been applied to financial problems for a long time. Early contributions in the late 1990s include Neuneier (1996), Moody and Wu (1997), Moody et al. (1998) and Neuneier (1998). Since then, many researchers in the computer science field have sought to apply RL techniques to portfolio problems. The *advent of massive datasets* and the *increase in dimensionality* make it hard for RL tools to adapt well to very rich environments that are encountered in factor investing.

Recently, some approaches seek to adapt RL to *continuous action spaces* (Wang and Zhou (2019), Aboussalah and Lee (2020)) but not to high-dimensional state spaces. These spaces are those required in factor investing because all firms yield hundreds of data points characterizing their economic situation. In addition, applications of RL in financial frameworks have a particularity compared to many typical RL tasks: in financial markets, actions of agents have **no impact on the environment** (unless the agent is able to perform massive trades, which is rare and ill-advised because it pushes prices in the wrong direction). This lack of impact of actions may possibly *mitigate the efficiency* of traditional RL approaches.

Those are challenges that will need to be solved in order for RL to become competitive with alternative (supervised) methods. Nevertheless, the progressive (online-like) way RL works seems suitable for *non-stationary* environments: the algorithm slowly shifts paradigms as new data arrives. In stationary environments, it has been shown that RL manages to converge to optimal solutions (Kong et al. (2019), Chaouki et al. (2020)). Therefore, in non-stationary markets, RL could be a recourse to build dynamic predictions that *adapt to changing macroeconomic conditions*. More research needs to be carried out in this field on large dimensional datasets.

We end this chapter by underlining that reinforcement learning has also been used to *estimate com-*

plex theoretical models (Halperin and Feldshteyn (2018), García-Galicia, Carsteanu, and Clempner (2019)). The research in the field is incredibly diversified and is orientated towards many directions. It is likely that captivating work will be published in the near future.

1.6 Exercises

1. Test what happens if the process for generating returns has a negative autocorrelation. What is the impact on the Q function and the policy.

We will re-conduct the process in Section 16.4.1 with a negative ρ.

[ ]:
```
rho = -0.8 # NEGATIVE
sd = 0.4
a = 0.06 * rho

ar1_process = ArmaProcess(ar=np.array([1, -rho]))
returns = a / rho + ar1_process.generate_sample(nsample=n_sample, scale=sd)
action = np.round(np.random.uniform(0, 1, n_sample) * 4) / 4

data_RL = pd.DataFrame([returns, action], index=['returns', 'action']).T
data_RL['new_state'] = data_RL['returns'].apply(lambda x: 'neg' if x < 0 else␣
 ↪'pos')
data_RL['reward'] = data_RL['returns'] * data_RL['action']
data_RL['state'] = data_RL['new_state'].shift(1)
data_RL = data_RL.dropna()
```

[ ]:
```
eta = 0.1
gamma = 0.7
epsilon = 0.1
fit_RL, r_final = q_learning_sim(eta, gamma, epsilon, data_RL)
Q_func = pd.DataFrame.from_dict(fit_RL).T.sort_index(axis=1)
display(Q_func)
print(f'Reward (last iteration): {r_final}')
```

```
         0.00      0.25      0.50      0.75      1.00
neg  0.437242  0.508467  0.636388  0.808101  1.046243
pos  0.556519  0.509264  0.521704  0.354364  0.263849

Reward (last iteration): 2882.6919890304275
```

In this case, the negative state is associated with large profits when the portfolio is fully invested, while the positive state has the best average reward when the agent refrains from investing.

2. Keeping the same 2 assets as in Section 16.4.2, increases the size of RL_data by testing **all possible action combinations** for each original data point. Re-run the Q-learning function and see what happens.

[ ]:
```
import itertools
pos_3 = [-1, 0, 1]
pos_4 = [-1, 0, 1]
pos = itertools.product(pos_3, pos_4)
pos_combinations = pd.DataFrame(pos, columns=['pos_3', 'pos_4'])

RL_data = pd.DataFrame([return_3, return_4, pb_3, pb_4], index=['return_3',␣
 ↪'return_4', 'pb_3', 'pb_4']).T
# NOTE: repeat the original data for 9 times to match all the combinations
RL_data = pd.concat([RL_data] * len(pos_combinations), ignore_index=True)
# NOTE: repeat the combinations for N (the length of records) times
pos_combinations = pd.DataFrame(np.repeat(pos_combinations, len(return_3),␣
 ↪axis=0), columns=['pos_3', 'pos_4'])
RL_data = pd.concat([RL_data, pos_combinations], axis=1)

# Processing
RL_data['action'] = RL_data['pos_3'].astype('int64').apply(str) + " " +␣
 ↪RL_data['pos_4'].astype('int64').apply(str)
RL_data['pb_3'] = np.round(5 * RL_data['pb_3']) # simplifying
RL_data['pb_4'] = np.round(5 * RL_data['pb_4']) # simplifying
RL_data['state'] = RL_data['pb_3'].astype('int64').apply(str) + " " +␣
 ↪RL_data['pb_4'].astype('int64').apply(str)
RL_data['reward'] = RL_data['pos_3'] * RL_data['return_3'] + RL_data['pos_4'] *␣
 ↪RL_data['return_4']
RL_data['new_state'] = RL_data['state'].shift(-1)

RL_data = RL_data[['action', 'state', 'reward', 'new_state']][:-1]
RL_data
```

[ ]:

	action	state	reward	new_state
0	-1 -1	1 1	-0.061	1 1
1	-1 -1	1 1	0.024	1 1
2	-1 -1	1 1	-0.171	1 1
3	-1 -1	1 1	-0.016	1 1
4	-1 -1	1 1	-0.064	1 1
…	…	…	…	…
2046	1 1	1 2	0.029	1 2
2047	1 1	1 2	-0.013	1 2
2048	1 1	1 2	-0.095	1 2
2049	1 1	1 2	0.036	1 2
2050	1 1	1 2	-0.158	1 3

[2051 rows x 4 columns]

[ ]:
```
fit_RL, r_final = q_learning_sim(eta, gamma, epsilon, RL_data)
Q_func = pd.DataFrame.from_dict(fit_RL).T.sort_index(axis=1)
# display(Q_func.style.format(precision=4))
print(Q_func.to_string(float_format=lambda x: '{:.4f}'.format(x)))
print(f'Reward (last iteration): {r_final}')
```

	-1 -1	-1 0	-1 1	0 -1	0 0	0 1	1 -1	1 0	1 1
1 1	-0.0309	-0.0059	0.0366	0.0021	0.0196	0.0474	0.0179	0.0412	0.0670
2 1	0.0034	0.0136	0.0372	0.0170	0.0203	0.0294	0.0098	0.0161	0.0367
2 2	0.0377	0.0460	0.0442	0.0248	0.0191	0.0111	-0.0074	-0.0127	-0.0196
3 1	0.0020	0.0020	0.0029	0.0028	0.0030	0.0027	0.0017	0.0016	-0.0001
1 2	0.0148	0.0222	0.0252	0.0167	0.0157	0.0176	0.0081	0.0108	0.0178
2 3	0.0538	0.0569	0.0529	0.0079	0.0068	0.0060	-0.0413	-0.0411	-0.0443
1 3	-0.0362	-0.0269	-0.0173	-0.0023	0.0043	0.0175	0.0367	0.0480	0.0573
0 3	-0.0128	-0.0194	-0.0261	0.0066	-0.0003	-0.0084	0.0307	0.0243	0.0207
0 2	-0.0166	-0.0184	-0.0202	0.0018	0.0003	-0.0004	0.0234	0.0248	0.0239

Reward (last iteration): -0.07300000000000206

The matrix is less sparse; we have covered much more ground! The change occurs for the states for which only a few points were available in the first trial. With more data, the decision is altered.

1.7 Takeaways

- Theoretical Layout

  - **Markov Decision Process** (MDP)
  - Elements
    * Agent (e.g., a trader or portfolio manager)
    * Environment (e.g., a financial market)
    * Actions (positions)
    * State
    * Reward (returns, Sharpe ratio, etc.)
    * Transition probability $p(s', r|s, a)$
    * Gain and discount rate γ
    * The focal to be optimized: **policy** $\pi = \pi(a, s) = \mathbb{P}[A_t = a|S_t = s]$
    * Value function: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$
    * Action-value function $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$
    * Find the optimal v_π and q_π: challenging
  - Q-learning
    * Remove the dependence on policy -> no iteration needed
    * $q_*(s, a) = \mathbb{E}_{\pi^*}[r|s, a] + \gamma \sum_{r,s'} p(s', r|s, a) \left(\max_{a'} q_*(s', a') \right)$
    * A itertion process with Q updated as $Q_{i+1}(s_i, a_i) \longleftarrow Q_i(s_i, a_i) + \eta \left(r_{i+1} + \gamma \max_a Q_i(s_{i+1}, a) - Q_i(s_i, a_i) \right)$
    * ϵ-greedy (exploitation VS exploration): "explore" with probability ϵ to gather new information when Q_i is relatively far from the *true* Q
  - SARSA
    * On-policy learning (Q-learning as off-policy)
    * Similar to Q-learning with improvement only from the **local** point $Q_i(s_{i+1}, a_{i+1})$
    * Expected SARSA: the target Q from is averaged over all actions, less impacted by the random choice a_{i+1} thus less volatile
- The curse of dimensionality
  - The action take values on a simplex with space $[0, 1]^{NK}$
  - Numerically impractical dimensions
  - A simple solution: discretization, divide the space into a small number of categories
    * Could over-simplify
    * Realistic discretizations are numerically intractable when investing in multiple assets
  - Training data: simulated data not preferred since a simple utility function maximization would be suffice, but market data can be small in scale
- Policy Gradient
  - **Parametrization** for $\pi_{\boldsymbol{\theta}}(a, s)$
    * One form: soft-max form
    * Policy gradient theorem and gradient ascent: REINFORCE algorithm

  - Extensions
    - Actor-critic (AC) method that mixes policy gradient and SARSA
    - Updated by $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \left(R_{t+1} + \gamma v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}) \right) \frac{\nabla \pi_{\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}}}$
    - Other forms for $\pi_{\boldsymbol{\theta}} = f_{\boldsymbol{\Omega}(s,\boldsymbol{\theta})}(a)$: a Dirichlet distribution for a small number of assets
- Concluding Remarks
  - Main issue for RL in finance: massive datasets and increase in dimensionality
  - A particularity: actions of agents typically have no impact on the environment which may possibly mitigate the efficiency of traditional RL approaches
  - Possible contributions: suitable for non-stationary markets
  - Other applications: estimation for complex theoretical models