Reinforcement Learning For Systematic Trading

Pierpaolo G. Necchi
Mathematical Engineering
Politecnico di Milano
Milano, IT 20123
pierpaolo.necchi@gmail.com

Abstract

TODO

1 Introduction

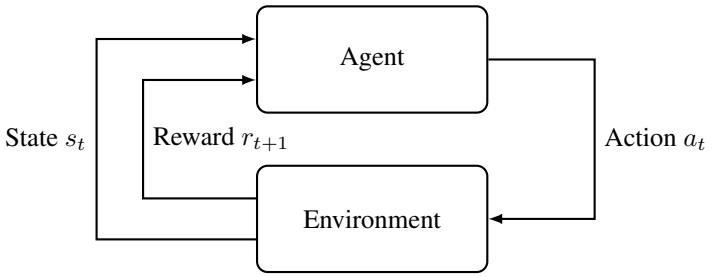

Figure 1: Agent-environment interaction in sequential decision problems.

2 Basics of Reinforcement Learning

Reinforcement Learning (RL) is a general class of algorithms in the field of *Machine Learning* (ML) that allows an agent to learn how to behave in a stochastic and possibly unknown environment, where the only feedback consists of a scalar reward signal [1]. The goal of the agent is to learn by trial-and-error which actions maximize his long-run rewards. However, since the environment evolves stochastically and may be influenced by the actions chosen, the agent must balance his desire to obtain a large immediate reward by acting greedily and the opportunities that will be available in the future. Thus, RL algorithms can be seen as computational methods to solve sequential decision problems by directly interacting with the environment.

2.1 Markov Decision Processes

Sequential decision problems are typically formalized using *Markov Decision Processes* (MDP). An MDP is a stochastic dynamical system specified by the tuple $< \mathbb{S}, \mathbb{A}, \mathcal{P}, \mathcal{R}, \gamma >$, where $(\mathbb{S}, \mathcal{S})$ is a measurable state space, $(\mathbb{A}, \mathcal{A})$ is a measurable action space, $\mathcal{P} : \mathbb{S} \times \mathbb{A} \times \mathcal{S} \to \mathbb{R}$ is a Markov transition kernel, $\mathcal{R} : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ is a reward function and $0 < \gamma < 1$ is the discount factor. Suppose that at time t the system is in state $S_t = s$ and that the agent takes action $A_t = a$, then, regardless of the previous history of the system, the probability to find the system in a state belonging to $B \in \mathcal{S}$ at time $t + 1$ is given by

$$\mathcal{P}(s, a, B) = \mathbb{P}\left(S_{t+1} \in B | S_t = s, A_t = a\right) \tag{1}$$

Following this random transition, the agent receives a stochastic reward R_{t+1}. The reward function $\mathcal{R}(s, a)$ gives the expected reward obtained when action a is taken in state s, i.e.

$$\mathcal{R}(s, a) = \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right] \tag{2}$$

This feedback mechanism between the environment and the agent is illustrated in Figure 1. At any time step, the agent selects his actions according to a certain policy $\pi : \mathbb{S} \times \mathcal{A} \to \mathbb{R}$ such that for every $s \in \mathbb{S}$, $C \mapsto \pi(s, C)$ is a probability distribution over $(\mathbb{A}, \mathcal{A})$. Hence, a policy π and an initial state $s_0 \in \mathbb{S}$ determine a random state-action-reward sequence $\{(S_t, A_t, R_{t+1})\}_{t \geq 0}$ with values on $\mathbb{S} \times \mathbb{A} \times \mathbb{R}$. In an infinite horizon task, the agent's performance is typically measured as the total discounted reward obtained following a specific policy

$$G_t = \sum_{t=0}^{\infty} \gamma^t R_{t+k+1} \tag{3}$$

Since this gain is stochastic, the agent considers its expected value, which is typically called *state-value function*

$$V_\pi(s) = \mathbb{E}_\pi\left[G_t | S_t = s\right] \tag{4}$$

where the subscript in $\mathbb{E}_\pi$ indicates that all the actions are selected according to policy π. The state-value function measures how good it is for the agent to be in a given state and follow a certain policy. Similarly, we introduce the *action-value function*

$$Q_\pi(s, a) = \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] \tag{5}$$

We have the following relationship between V_π and Q_π

$$V_\pi(s) = \int_{\mathbb{A}} \pi(s,a) Q_\pi(s,a) da \tag{6}$$

Almost all reinforcement learning algorithms are designed to estimate these value functions and are typically based on the Bellman equations.

$$V_\pi(s) = \mathcal{R}_\pi(s) + \gamma T_\pi V_\pi(s) \tag{7}$$

$$Q_\pi(s,a) = \mathcal{R}(s,a) + \gamma T_a V_\pi(s) \tag{8}$$

where we denoted by T_a (resp. T_π) the transition operator for action a (resp. for policy π)

$$T_a F(s) = \mathbb{E}\left[F(S_{t+1}) | S_t = s, A_t = a\right] = \int_{\mathbb{S}} \mathcal{P}(s,a,s') F(s') ds' \tag{9}$$

$$T_\pi F(s) = \mathbb{E}_\pi\left[F(S_{t+1}) | S_t = s\right] = \int_{\mathbb{A}} \pi(s,a) \int_{\mathbb{S}} \mathcal{P}(s,a,s') F(s') ds' da \tag{10}$$

These equations can be rewritten as fixed-point equations which, under some formal assumptions on the reward functions, admit a unique solution by the contraction mapping theorem. The agent's goal is to select a policy π_* that maximizes his expected return in all possible states. Such a policy is called *optimal* and the corrisponding value functions are called *Optimal State-Value Function*

$$V_*(s) = \sup_\pi V_\pi(s) \tag{11}$$

and *Optimal Action-Value Function*

$$Q_*(s,a) = \sup_\pi Q_\pi(s,a) \tag{12}$$

The optimal value functions satisfy the following Bellman equations.

$$V_*(s) = \sup_a Q_*(s,a) = \sup_a \left\{\mathcal{R}(s,a) + \gamma T_a V_*(s)\right\} \tag{13}$$

$$\begin{aligned} Q_*(s,a) &= \mathcal{R}(s,a) + \gamma T_a V_*(s) \\ &= \mathcal{R}(s,a) + \gamma \int_{\mathbb{S}} \mathcal{P}(s,a,s') \sup_{a'} Q_*(s',a') ds' \end{aligned} \tag{14}$$

Again, these are fixed-point equations for which the existence and uniqueness of a solution is guaranteed by the contraction mapping theorem. Given the optimal action-value function Q_*, an optimal policy is obtained by selecting in each state the action with maximizes Q_*

$$a_* = \arg\sup_a Q_*(s,a) \tag{15}$$

This greedy policy is deterministic and only depends on the current state of the system.

2.2 Policy Gradient Methods

The standard way to solve MDPs is through dynamic programming, which simply consists in solving the Bellman fixed-point equations discussed in the previous chapter. Following this approach, the problem of finding the optimal policy is transformed into the problem of finding the optimal value function. However, apart from the simplest cases where the MDP has a limited number of states and actions, dynamic programming becomes computationally infeasible. Moreover, this approach requires complete knowledge of the Markov transition kernel and of the reward function, which in many real-world applications might be unknown or too complex to use. *Reinforcement Learning* (RL) is a subfield of Machine Learning which aims to turn the infeasible dynamic programming methods into practical algorithms that can be applied to large-scale problems. RL algorithms are based on two key ideas: the first is to use samples to compactly represent the unknown dynamics of the controlled system. The second idea is to use powerful function approximation methods to compactly estimate value functions and policies in high-dimensional state and action spaces. In this section we will only focus on a particular class of algorithms called *Policy Gradient Methods*, which have proved successful in many applications. For a more complete introduction to RL, the reader may consult [1], [2] or [3].

In *policy gradient methods* [4], the optimal policy is approximated using a parametrized policy $\pi : \mathbb{S} \times \mathcal{A} \times \Theta \to \mathbb{R}$ such that, given a parameter vector $\theta \in \Theta \subseteq \mathbb{R}^{D_\theta}$, $\pi(s, B; \theta) = \pi_\theta(s, B)$ gives the probability of selecting an action in $B \in \mathcal{A}$ when the system is in state $s \in \mathbb{S}$. The general goal of policy optimization in reinforcement learning is to optimize the policy parameters $\theta \in \Theta$ so as to maximize a certain objective function $J : \Theta \to \mathbb{R}$

$$\theta^* = \arg\max_{\theta \in \Theta} J(\theta) \tag{16}$$

In the following, we will focus on gradient-based and model-free methods that exploit the sequential structure of the the reinforcement learning problem. The idea of policy gradient algorithms is to update the policy parameters using the gradient ascent direction of the objective function

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta J\left(\theta_k\right) \tag{17}$$

where $\{\alpha_k\}_{k \geq 0}$ is a sequence of learning rates. Typically, the gradient of the objective function is not known and its approximation is the key component of every policy gradient algorithm. It is a well-know result from stochastic optimization [5] that, if the gradient estimate is unbiased and the learning rates satisfy the *Robbins-Monro conditions*

$$\sum_{k=0}^{\infty} \alpha_k = \infty \qquad \sum_{k=0}^{\infty} \alpha_k^2 < \infty \tag{18}$$

the learning process is guaranteed to converge at least to a local optimum of the objective function. In an episodic environment where the system always starts from an initial state s_0, the typical objective function is the start value.

$$J_{\text{start}}(\theta) = V_{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}\left[G_0 | S_0 = s_0\right] \tag{19}$$

In a continuing environment, where no terminal state exists and the task might go on forever, it is common to use either the average value

$$J_{\text{avV}}(\theta) = \mathbb{E}_{S \sim d^\theta}\left[V_{\pi_\theta}(S)\right] = \int_{\mathbb{S}} d^\theta(s) V_{\pi_\theta}(s) ds \tag{20}$$

where d^θ is the stationary distribution of the Markov chain induced by π_θ. Alternatively, one may use the average reward per time step

$$J_{\text{avR}}(\theta) = \rho(\theta) = \mathbb{E}_{\substack{S \sim d^\theta \\ A \sim \pi_\theta}}\left[\mathcal{R}(S, A)\right] = \int_{\mathbb{S}} d^\theta(s) \int_{\mathbb{A}} \pi_\theta(s, a) \mathcal{R}(s, a) da ds \tag{21}$$

Luckily, the same methods apply with minor changes to the three objective functions.

2.2.1 Policy Gradient Theorem

The *policy gradient theorem* [6] shows that the gradient can be rewritten in a form suitable for estimation from experience aided by an approximate action-value or advantage function.

Theorem 2.1 (Policy Gradient)**.** *Let π_θ be a differentiable policy. The policy gradient for the average reward formulation is given by*

$$\nabla_\theta \rho(\theta) = \mathbb{E}_{\substack{S \sim d^\theta \\ A \sim \pi_\theta}}\left[\nabla_\theta \log \pi_\theta(S, A) Q_\theta(S, A)\right] \tag{22}$$

where d^θ is the stationary distribution of the Markov chain induced by π_θ. The policy gradient for the start value formulation is given by

$$\nabla_\theta J_{start}(\theta) = \mathbb{E}_{\substack{S \sim d_\gamma^\theta(s_0, \cdot) \\ A \sim \pi_\theta}}\left[\nabla_\theta \log \pi_\theta(S, A) Q_\theta(S, A)\right] \tag{23}$$

where $d_\gamma^\theta(s_0, \cdot)$ is the γ-discounted visiting distribution over states starting from the initial state s_0 and following policy π_θ

$$d_\gamma^\theta(s, x) = \sum_{k=0}^{\infty} \gamma^k \mathcal{P}_\theta^{(k)}(s, x) \tag{24}$$

Algorithm 1 GPOMDP

Input:

- Initial policy parameters $\theta_0 = (\theta_0^1, \dots, \theta_0^{D_\theta})^T$
- Learning rate $\{\alpha_k\}$
- Number of trajectories M

Output: Approximation of the optimal policy $\pi_{\theta^*} \approx \pi_*$

1: Initialize $k = 0$
2: **repeat**
3: Sample M trajectories $h^{(m)} = \{(s_t^{(m)}, a_t^{(m)}, r_{t+1}^{(m)}\}_{t=0}^{T^{(m)}}$ of the MDP under policy π_{θ_k}
4: Compute the optimal baseline

$$\widehat{b}_k^n = \frac{\sum_{m=1}^{M} \left[\sum_{i=0}^{T^{(m)}} \partial_{\theta_k} \log \pi_\theta \left(s_i^{(m)}, a_i^{(m)}\right)\right]^2 \sum_{j=0}^{T^{(m)}} \gamma^j r_{j+1}^{(m)}}{\sum_{m=1}^{M} \left[\sum_{i=0}^{T^{(m)}} \partial_{\theta_k} \log \pi_\theta \left(s_i^{(m)}, a_i^{(m)}\right)\right]^2} \tag{27}$$

5: Approximate policy gradient

$$\frac{\partial}{\partial \theta^n} J_{\text{start}}(\theta_k) \approx \widehat{g}_k^n = \frac{1}{M} \sum_{m=1}^{M} \sum_{i=0}^{T^{(m)}} \frac{\partial}{\partial \theta^n} \log \pi_{\theta_k} \left(s_i^{(m)}, a_i^{(m)}\right) \left(\sum_{j=i}^{T^{(m)}} \gamma^j r_{j+1}^{(m)} - \widehat{b}_k^n\right) \tag{28}$$

6: Update actor parameters $\theta_{k+1} = \theta_k + \alpha_k \widehat{g}_k$.
7: $k \leftarrow k + 1$
8: **until** converged

Let us notice that we can subtract a state-dependent baseline from the action-value function without changing the value of the expectation, indeed

$$\begin{aligned}
\mathbb{E}_{\substack{S\sim d^\theta \\ A\sim\pi_\theta}} \left[\nabla_\theta \log \pi_\theta(S, A) B_\theta(S)\right] &= \int_{\mathbb{S}} d^\theta(s) \int_{\mathbb{A}} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) B_\theta(s) da ds \\
&= \int_{\mathbb{S}} d^\theta(s) B_\theta(s) \int_{\mathbb{A}} \nabla_\theta \pi_\theta(s, a) da ds \\
&= \int_{\mathbb{S}} d^\theta(s) B_\theta(s) \nabla_\theta \underbrace{\int_{\mathbb{A}} \pi_\theta(s, a) da}_{=1} ds = 0
\end{aligned}$$

Hence, the policy gradient theorem can be rewritten as

$$\nabla_\theta \rho(\theta) = \mathbb{E}_{\substack{S\sim d^\theta \\ A\sim\pi_\theta}} \left[\nabla_\theta \log \pi_\theta(S, A) \left(Q_{\pi_\theta}(S, A) - B_\theta(S)\right)\right] \tag{25}$$

The baseline can be chosen so as to minimize the variance of the gradient estimate which can prove beneficial for the algorithm convergence [4]. This result can be used as the starting point to derive several policy gradient methods that use different approximation of the action-value function, which is typically unknown. For instance, in an episodic MDP the action-value function can be estimated with the total return obtained on a sample trajectory

$$Q_\theta(s_0, a_0) \approx \sum_{t=0}^{T^{(m)}} \gamma^t r_{t+1}^{(m)} \tag{26}$$

Combining this remark with a Monte Carlo approximation of Eq. (25), we obtain the *Monte Carlo Policy Gradient* algorithm [7] (also known as GPOMDP) for which the pseudocode is reported in Algorithm 1.

2.2.2 Parameter-Based Policy Gradient Methods

In Monte Carlo Policy Gradient, trajectories are generated by sampling at each time step an action according to a stochastic policy π_θ and the objective function gradient is estimated by differentiating

the policy with respect to the parameters. However, sampling an action from the policy at each time step leads to a large variance in the sampled histories and therefore in the gradient estimate, which can in turn slow down the convergence of the learning process. To address this issue, the *policy gradient with parameter-based exploration* (PGPE) method [8] replaces the search in the policy space with a direct search in the model parameter space. Given an episodic MDP, PGPE considers a deterministic controller $F : \mathbb{S} \times \Theta \to \mathbb{A}$ that, given a set of parameters $\theta \in \Theta \subseteq \mathbb{R}^{D_\theta}$, maps a state $s \in \mathbb{S}$ to an action $a = F(s;\theta) = F_\theta(s) \in \mathbb{A}$. The policy parameters are drawn from a probability distribution p_ξ, with hyper-parameters $\xi \in \Xi \subseteq \mathbb{R}^{D_\xi}$. Combining these two hypotheses, the agent follows a stochastic policy π_ξ defined by

$$\forall B \in \mathcal{A},\ \pi_\xi(s, B) = \pi(s, B; \xi) = \int_\Theta p_\xi(\theta) \mathbb{1}_{F_\theta(s) \in B} d\theta \tag{29}$$

In this setting, the policy gradient theorem can be reformulated in the following way

Theorem 2.2 (Parameter-Based Policy Gradient)**.** *Let p_ξ be differentiable with respect to ξ, then the gradient of the average reward is given by*

$$\nabla_\xi J(\xi) = \mathbb{E}_{\substack{S \sim d^\xi \\ \theta \sim p_\xi}} \left[\nabla_\xi \log p_\xi(\theta) Q_{\pi_\xi}(S, \theta) \right] \tag{30}$$

where we denoted $Q_\xi(S, \theta) = Q_\xi(S, F_\theta(S))$.

This expression is very similar to the original policy gradient theorem, but the expectation is taken over the controller parameters instead of the action space and we have the likelihood score of the controller parameters distribution instead of that of the stochastic policy. Thus, we might interpret this result as if the agent directly selected the parameters θ according to a policy p_ξ, which then lead to an action through the deterministic mapping F_θ. Therefore, it is as if the agent's policy was in the parameters space and not in the control space. As in the standard policy gradient methods, we can subtract a state-dependent baseline $B_\xi(S)$ to the gradient without increasing the bias

$$\nabla_\xi J(\xi) = \mathbb{E} \left[\nabla_\xi \log p_\xi(\theta) \left(Q_{\pi_\xi}(S, \theta) - B_\xi(S) \right) \right] \tag{31}$$

The PGPE algorithm, which is outlined in Algorithm 2, employs a Monte Carlo approximation of this gradient, where the action-value function is estimated using the returns on a sampled trajectory of the MDP. The benefit of this approach is that the controller is deterministic and therefore the actions do not need to be sampled at each time step, with a consequent reduction of the gradient estimate variance. Indeed, It is sufficient to sample the parameters θ once at the beginning of the episode and then generate an entire trajectory following the deterministic policy F_θ. As an additional benefit, the parameter gradient is estimated by direct parameter perturbations, without having to backpropagate any derivatives, which allows to use non-differentiable controllers. Again the baseline can be chosen so as to minimize the gradient estimate variance [9].

3 Reinforcement Learning for Systematic Trading

Many financial problems can be seen as sequential decision problems which naturally fall in the stochastic optimal control framework introduced above. In this section we discuss how the reinforcement learning algorithms can be applied to the asset allocation problem, where an agent invests his capital on various assets available in the market.

3.1 Asset Allocation With Transaction Costs

The asset allocation problem consists of determining how to dynamically invest the available capital in a portfolio of different assets in order to maximize the expected total return or another relevant performance measure. Let us consider a financial market consisting of $I + 1$ different stocks that are traded only at discrete times $t \in \{0, 1, 2, \ldots\}$ and denote by $Z_t = (Z_t^0, Z_t^1, \ldots, Z_t^I)^T$ their prices at time t. Typically, Z_t^0 refers to a riskless asset whose dynamic is given by $Z_t^0 = (1 + X)^t$ where X is the deterministic risk-free interest rate. The investment process works as follows: at time t, the investor observes the state of the market S_t, consisting for example of the past asset prices and other relevant economic variables, and subsequently chooses how to rebalance his portfolio, by specifying the units of each stock $n_t = (n_t^0, n_t^1, \ldots, n_t^I)^T$ to be held between t and $t + 1$. In doing so, he needs

Algorithm 2 Episodic PGPE algorithm

Input:

- Initial hyper-parameters $\xi_0 = (\xi_0^1, \ldots, \xi_0^{D_\xi})^T$
- Learning rate $\{\alpha_k\}$
- Number of trajectories M

Output: Approximation of the optimal policy $F_{\xi^*} \approx \pi_*$

Initialize $k = 0$
repeat
  for $m = 1, \ldots, M$ **do**
    Sample controller parameters $\theta^{(m)} \sim p_{\xi_k}$
    Sample trajectory $h^{(m)} = \{(s_t^{(m)}, a_t^{(m)}, r_{t+1}^{(m)}\}_{t=0}^{T^{(m)}}$ under policy $F_{\theta^{(m)}}$
  end for
  Compute optimal baseline

$$\widehat{b}_k^n = \frac{\sum_{m=1}^M \left[\partial_{\xi^n} \log p_{\xi_k}\left(\theta^{(m)}\right)\right]^2 \sum_{j=0}^{T^{(m)}} \gamma^j r_{j+1}^{(m)}}{\sum_{m=1}^M \left[\partial_{\xi^n} \log p_{\xi_k}\left(\theta^{(m)}\right)\right]^2} \tag{32}$$

  Approximate policy gradient

$$\frac{\partial}{\partial \xi^n} J_{\text{start}}(\xi_k) \approx \widehat{g}_k^n = \frac{1}{M} \sum_{m=1}^M \frac{\partial}{\partial \xi^n} \log p_{\xi_k}\left(\theta^{(m)}\right) \left(\sum_{j=i}^{T^{(m)}} \gamma^j r_{j+1}^{(m)} - \widehat{b}_k^n \right) \tag{33}$$

  Update hyperparameters using gradient ascent $\xi_{k+1} = \xi_k + \alpha_k \widehat{g}_k^n$
  $k \leftarrow k + 1$
until converged

to take into account the transaction costs that he has to pay to the broker to change his position. At time $t+1$, the investor realizes a profit or a loss from his investment due to the stochastic variation of the stock values. The investor's goal is to maximize a given performance measure. Let W_t denote the wealth of the investor at time t. The profit realized between t and $t+1$ is simply given by the difference between the trading results and the transaction costs payed to the broker. More formally

$$\Delta W_{t+1} = W_{t+1} - W_t = \text{PNL}_{t+1} - \text{TC}_t$$

where PNL_{t+1} denotes the profit due to the variation of the portfolio asset prices between t and $t+1$

$$\text{PNL}_{t+1} = n_t \cdot \Delta Z_{t+1} = \sum_{i=0}^{I} n_t^i (Z_{t+1}^i - Z_t^i)$$

and TC_t denotes the fees payed to the broker to change the portfolio allocation and on the short positions

$$\text{TC}_t = \sum_{i=0}^{I} \delta_p^i \left| n_t^i - n_{t-1}^i \right| Z_t^i - \delta_f W_t \mathbb{1}_{n_t \neq n_{t-1}} - \sum_{i=0}^{I} \delta_s^i (n_t^i)^- Z_t^i$$

The transaction costs consist of three different components. The first term represent a transaction cost that is proportional to the change in value of the position in each asset. The second term is a fixed fraction of the total value of the portfolio which is payed only if the allocation is changed. The last term represents the fees payed to the broker for the shares borrowed to build a short position. The portfolio return between t and $t+1$ is thus given by

$$X_{t+1} = \frac{\Delta W_{t+1}}{W_t} = \sum_{i=0}^{I} \left[a_t^i X_{t+1}^i - \delta_i \left| a_t^i - \tilde{a}_t^i \right| - \delta_s (a_t^i)^- \right] - \delta_f \mathbb{1}_{a_t \neq \tilde{a}_{t-1}} \tag{34}$$

where

$$X_{t+1}^i = \frac{\Delta Z_{t+1}^i}{Z_t^i}$$

is the return of the i-th stock between t and $t+1$,

$$a_t^i = \frac{n_t^i Z_t^i}{W_t}$$

is the fraction of wealth invested in the i-th stock between time t and $t+1$ and finally

$$\tilde{a}_t^i = \frac{n_{t-1}^i Z_t^i}{W_t} = \frac{a_{t-1}^i (1+X_t^i)}{1+X_t}$$

is the fraction of wealth invested in the i-th stock just before the reallocation. We assume that the agent invests all his wealth at each step, so that W_t can be also interpreted as the value of his portfolio. This assumption leads to the following constraint on the portfolio weights

$$\sum_{i=0}^{I} a_t^i = 1 \qquad \forall t \in \{0, 1, 2, \ldots\} \tag{35}$$

We notice that we are neglecting the typical margin requirements on the short positions, which would reduce the available capital at time t. Considering margin requirements would lead to a more complex constraint on the portfolio weights which would be difficult to treat in the reinforcement learning framework. Plugging this constraint into Eq. (34), we obtain

$$X_{t+1} = X + \sum_{i=1}^{I} a_t^i (X_{t+1}^i - X) - \sum_{i=0}^{I} \left[\delta_i \left| a_t^i - \tilde{a}_t^i \right| - \delta_s^i (a_t^i)^- \right] - \delta_f \mathbb{1}_{a_t \neq \tilde{a}_{t-1}} \tag{36}$$

which highlights the role of the risk-free asset as a benchmark for the portfolio returns. The total profit realized by the investor between $t=0$ and T is

$$\Pi_T = W_T - W_0 = \sum_{t=1}^{T} \Delta W_t = \sum_{t=1}^{T} W_t X_t$$

The portfolio return between $t=0$ and T is given by

$$X_{0,T} = \frac{W_T}{W_0} - 1 = \prod_{t=1}^{T} (1+X_t) - 1$$

In order to cast the asset allocation problem in the reinforcement learning framework, we consider the log-return of the portfolio between $t=0$ and T

$$R_{0,T} = \log \frac{W_T}{W_0} = \sum_{t=1}^{T} \log(1+X_t) = \sum_{t=1}^{T} R_t \tag{37}$$

where R_{t+1} is the log-return of the portfolio between t and $t+1$

$$R_{t+1} = \log \left\{ 1 + \sum_{i=0}^{I} \left[a_t^i X_{t+1}^i - \delta_i \left| a_t^i - \tilde{a}_t^i \right| - \delta_s (a_t^i)^- \right] - \delta_f \mathbb{1}_{a_t \neq \tilde{a}_{t-1}} \right\} \tag{38}$$

The portfolio return and log-return can be used as the reward function of a RL algorithm, either in a offline or in an online approach.

3.2 Reinforcement Learning Application

In the previous section we derived the reward function for the asset allocation problem with transaction costs. In order to apply the policy gradient algorithms discussed in the previous sections we still need to define the state space, the action space and the agent's policy. For simplicity, we limit ourselves to the case of a single risky asset, i.e. $I=1$, but the discussion could be generalized to the multi-asset case.
We assume that at each time step the agent considers the $P+1$ past returns of the risky asset, i.e. $\{X_t, X_{t-1}, \ldots, X_{t-P}\}$. In order to properly incorporate the effects of transaction costs into his

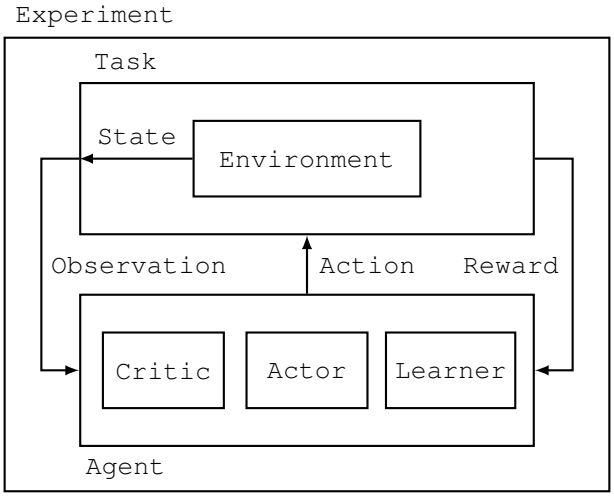

Figure 2: PyBrain standard architecture for an RL problem.

decision process, the agent must keep track of its current position $\tilde{a}_t$. The state of the system is thus given by

$$S_t = \{X_t, X_{t-1}, \dots, X_{t-P}, \tilde{a}_t\} \tag{39}$$

We might also include some external variables Y_t that may be relevant to the trader, such as the common technical indicator used in practice. Furthermore, these input variables may be used to construct more complex features for example using some deep learning techniques, such as a deep auto-encoder.

The agent, or trading system, specifies the portfolio weights $a_t = (a_t^0, a_t^1)^T$ according to a long-short strategy, i.e. the agent may be long ($a_t^1 = +1$) or short ($a_t^1 = -1$) on the risky-asset while $a_t^0 = 1 - a_t^1$ since the agents invests all the available capital at each time step. In the GPOMDP framework we assume that the agent selects a_t^1 according to a Boltzmann policy, i.e.

$$\pi_\theta(s, +1) = \frac{e^{\theta^T s}}{1 + e^{\theta^T s}} \qquad \pi_\theta(s, -1) = \frac{1}{1 + e^{\theta^T s}} \tag{40}$$

where we included a bias term in the parameters and in the state. In the parameter-based formulation, we assume that agent selects actions according to the binary controller

$$F_\theta(s) = \text{sign}(\theta^T s) \tag{41}$$

where the controller parameters are normally distributed $\theta \sim \mathcal{N}(\mu, \text{diag}(\sigma))$.

4 Python Prototype

In this section, we start discussing the implementation details of this project. The first step of this project has been to implement a prototype in Python, a high-level, general-purpose, interpreted, dynamic programming language which is gaining a widespread popularity both in the academic world and in the industry. Python natively supports the object-oriented paradigm which makes it perfect to quickly develop a prototype of the class architecture, which can then be translated in C++. Moreover, thanks to external libraries such as Numpy, Scipy and Pandas, Python offers an open-source alternative to Matlab for scientific computing applications.

For the basic RL algorithms we exploited PyBrain[1], a modular ML library for Python whose goal is to offer flexible, easy-to-use yet still powerful algorithms for ML tasks and a variety of predefined environments to test and compare different algorithms [10]. An RL task in PyBrain always consists of an `Environment`, an `Agent`, a `Task` and an `Experiment` interacting with each other as illustrated in Figure 2.

The `Environment` is the world in which the `Agent` acts and is characterized by a state

[1] http://pybrain.org/

which can be accessed through the `getSensors()` method. The `Agent` receives an observation of this state through the `integrateObservation()` method and selects an action through the `getAction()` method. This action is applied to the `Environment` with the `performAction()` method. However, the interactions between the `Environment` and the `Agent` are not direct but are mediated by the `Task`. The `Task` specifies what the goal is in an `Environment` and how the agent is rewarded for its actions. Hence, the composition of an `Environment` and a `Task` fully defines the MDP. An `Agent` always contains an `Actor`, which represents the policy used to select actions. Based on the rewards that the `Agent` receives via the `getReward()` method, the `Learner` improves the policy via a `learn()` procedure. In this step, an `Actor` may be used to evaluate a state with the goal of reducing the variance of the learning process. This entire learning process is controlled by an `Experiment` object.
This structure is quite standard for a RL problem and can be easily adapted to the problem at hand and extended to the learning algorithms developed in this thesis. Based on this architecture, we thus developed a fully-working Python prototype of the asset allocation problem. This prototype yielded some interesting results both on simulated data and on historical data, in particular for the PGPE algorithm. However, the learning process resulted too slow to be run systematically for a large number of time-steps and training epochs. By consequent, we quickly decided to pass to C++.

5 C++ Implementation

6 Execution Pipeline

7 Numerical Results

8 Conclusion

Acknowledgments

TODO

References

References

[1] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.

[2] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[3] Marco Wiering and Martijn Van Otterlo. *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*. Springer, 1 edition, 2012.

[4] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[5] Harold Kushner and G George Yin. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

[6] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[7] Jonathan Baxter and Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[8] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In *Artificial Neural Networks-ICANN 2008*, pages 387–396. Springer, 2008.

[9] Tingting Zhao, Hirotaka Hachiya, Gang Niu, and Masashi Sugiyama. Analysis and improvement of policy gradient estimation. In *Advances in Neural Information Processing Systems*, pages 262–270, 2011.

[10] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.