Prof. Alfio Ferrara

Lab of Reinforcement Learning

Coding classes

Introduction

The coding lessons of the reinforcement learning lab aim to complement the theory lessons with a more practical part in which we will learn to get our hands on the main models and algorithms of the theory.

General informations:

- The programming language is `Python 3.x`
- We will see some examples with the **Gym**) library (**Gymnasium**), but the code presented in class will almost exclusively be implemented from scratch using basic features of the Python language. The aim is not to hide the elements of complexity inherent in modeling reinforcement learning problems
- The code presented does not have the ambition to be the best possible implementation and favors the didactic purpose. Students are indeed invited to modify and improve the proposed code
- All material is available as a **GitHub repository** at https://github.com/afflint/rlcoding
- During the last lesson of the course **project outlines** will be presented for the final assignment which will consist of some theory questions and a short presentation of a project. Although not recommended, the project can be done by two people as long as the amount of work is adequate. The exam will be individual in all cases.

Class 1: MDP model and dynamic programming

The reference model that we need to implement is MDP, defined as

$$MDP = \langle S, A, p(\cdot \mid s, a) \rangle \tag{1}$$

A state $s \in S$ represents a state of the agent within the environment.

We then have:

States: $S_s \subseteq S$ as the set of initial states; $S_G \subset S$ and the set of golden states or final states.

Actions: In theory, we may assume that all the actions A are available from any state s. In practice, we model the action space as a function $A(s) \rightarrow A_s \subseteq A$.

Transition probability: $P(\cdot \mid s, a)$ is modeled as a transition function depending from the starting state s, the action a and the final state s', such that $T(a, a, s') \rightarrow p \in \mathbb{R}\ with\ 0 \leq p \leq 1 \wedge \sum_{s_i \in S} T(s, a, s_i) = 1$.

Moreover, we have:

Reward: We model the reward as the outcome of ending up in a state s' given that we have chosen action a from state s. This will be a function of the starting state, the action and the ending state: $R(s, a, s')$

Discount: We may not have any final state. However, in an infinite horizon MDP, it's hard to compute the infinite sum of rewards because it may not converge to a finite value. We can address this issue by the intuition of discounting the value of the future reward in proportion to how far it is in time. To this end we will have also a discount factor $\gamma : 0 \leq \gamma < 1$. Remember that:

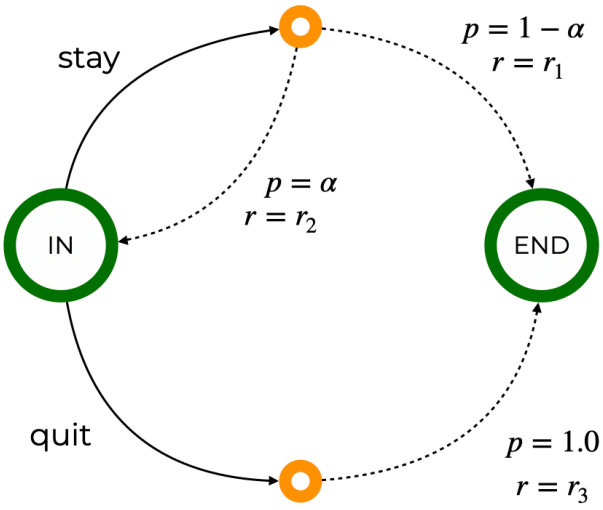
$$R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \dots \tag{2}$$

Goals of RL

Given and MDP, we may have to deal with two different situations:

1. We **actually know** $p(\cdot \mid s, a)$ and $r(s, a, s')$: still, it's not trivial to understand which is the optimal behavior in the environment (i.e., how to maximize the expected return) $\rightarrow$ evaluate the value of a policy and find the optimal policy (**this is the content of class 1**)
2. We **do not know** $p(\cdot \mid s, a)$ and $r(s, a, s')$: we can only interact with the unknown MDP and observe the reward in order to:
    1. Learn $p(\cdot \mid s, a)$ and $r(s, a, s')$ and use them to find the optimal polity (**model-based RL**) (**not in this course**)
    2. Learn directly how to maximize the reward (**model-free RL**) (**this is the content of class 2**)

Example: State-Quit Game



Exercise: implement this MDP.

```
class StayQuitMDP(object):
    def __init__(self, alpha, r1, r2, r3, gamma=.9):
        self.alpha, self.r1, self.r2, self.r3 = alpha, r1, r2, r3
        self.gamma = gamma
        self.P = {
            ('IN', 'stay', 'IN'): self.alpha, ('IN', 'stay', 'END'): 1 - self.alpha,
            ('IN', 'quit', 'END'): 1.
        }
        self.R = {
            ('IN', 'stay', 'IN'): self.r2, ('IN', 'stay', 'END'): 1 - self.r1,
```

```
            ('IN', 'quit', 'END'): self.r3
        }
    def states(self):
        return {'IN', 'END'}
    def actions(self, s):
        A = set()
        if s == 'IN':
            A = {'stay', 'quit'}
        return A
    def transition(self, s, a, s_prime):
        p = 0.
        try:
            p = self.P[(s, a, s_prime)]
        except KeyError:
            pass
        return p
    def reward(self, s, a, s_prime):
        r = 0.
        try:
            r = self.R[(s, a, s_prime)]
        except KeyError:
            pass
        return r
    def successors(self, s, a):
        outcomes = []
        for s_prime in self.states():
            p = self.transition(s, a, s_prime)
            if p > 0:
                outcomes.append((s_prime, p, self.reward(s, a, s_prime)))
        return outcomes
    def start_states(self):
        return {'IN'}
    def is_gold(s):
        return s == 'END'
```

(See the implementation of this and the abstract class)

Policy, policy evaluation and optimal policy

A policy π is a mapping from states to actions. We may have **deterministic policies** and **stochastic policies**. Here, we will implement deterministic policies.

Motivating example

Lets try to play `stay-quit` under the assumption that you can only choose the stationary policies `stay` or `quit` and with:

$$\alpha = \frac{2}{3} \quad | \quad r_1 = r_2 = 4 \quad | \quad r_3 = 10 \tag{3}$$

```
s = list(mdp.start_states())[0]
message = 'choose and action: '
R = 0
while not mdp.is_gold(s):
    a = input(message)
    clear_output(wait=True)
    if a in mdp.actions(s):
        message = 'choose and action'
        options = mdp.successors(s, a)
        i = np.random.choice(range(len(options)), p=[p for _, p, _ in options])
        s, _, r = options[i]
        R += r
        print('Total reward: {}'.format(R))
    else:
        message = 'not a valid action, retry: '
```

Now, observe that when I'm wondering what's best to do, what I really want to know is how much I can **expect to gain** by choosing to do a when I'm in s. In other terms, what is the expected reward of choosing a in s, also called the **action-value** of (s, a). This depends on the **immediate** reward plus the (**eventually discounted**) value of future actions.

$$Q(s,a) = \sum_{s'} p(s' \mid s,a) \left[r(s,a,s') + \gamma V(s') \right], \tag{4}$$

Where $V(s')$ is the expected value of the next state s'.

Policy evaluation

The key point here is that when we want to evaluate a policy, we always know which action we should choose in any given state, because the policy is a mapping from state to actions. Thus, we can evaluate the value $V_\pi(s)$ of each state according to the policy π as:

$$V_\pi(s) = \begin{cases} 0 & if\ s \in S_G \\ Q(s, \pi(s)) & otherwise \end{cases} \tag{5}$$

Now what we need is an iterative algorithm for implementing the state-value for each state with policy π.

```
Init V(s) = 0 for all s in S
Init th as a small threhsold
Delta = True
while Delta:
  new_v = {}
  for s in S:
    if s in S_G:
      new_v[s] = 0
    else:
      new_v[s] = Q(s, pi(s))
  if max(|new_v[s] - V[s]| for all s in S) < th:
    Delta = False
  V = new_v
```

(see the implementation of this and how to compute the values)

Policy improvement

When we have a policy π, for all states s, we can find a greedy policy π' by:

$$\pi'(s) = \arg\max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma V_\pi(s') \right] \tag{6}$$

Note: suppose that π' is not better than the old policy π. Then $V_\pi = V_{\pi'}$. Thus we can rewrite:

$$V_{\pi'}(s) = \max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma V_{\pi'}(s') \right] \tag{7}$$

Now, since (7) is the **Bellman optimality equation** $V_{\pi'}$ must be V^* and both π and π' must be optimal policies. Thus policy improvement **always gives us a strictly better policy except when the policy is already optimal**.

Policy Iteration

Now, we can combine policy evaluation and policy improvement to obtain the optimal policy.

```
## Initialize the policy randomly
    pi = randomPolicy()

    Loop:
        # Policy evaluation

        optimal_policy = True # we assume this is the optimal policy
        V = policy_evaluation(pi)

        # Now we run policy update
        for s in S:
            current_value = V(s)
            for a in A(s):
```

```
                new_value = sum(
                  p(s, a, s_prime) * [r(s, a, s_prime) + gamma * V(s_prime)] for
                                s_prime in S)
                if new_value > current_value and
                    pi(s) != a: # We have found a new better action
                      pi_prime[s] = a
                      current_value = new_value
                      optimal_policy = False
        pi = pi_prime
        if optimal_policy: # nothing has changed, we can stop
            break
```

Value Iteration

Another strategy that we can use to find the optimal policy comes directly from the Bellmann optimality equation and states:

$$V^*(s) = \max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma V^*(s') \right] \tag{8}$$

We can turn this into an update rule:

$$V_{t+1}(s) = \max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma V_t(s') \right] \tag{9}$$

The idea is to improve the state-value through and iterative process. Let us define:

$$Q_t^*(s, a) = \sum_{s'} p(s' \mid s, a) [r(s, a, s') + \gamma V_t^*(s') \tag{10}$$

```
# Initialize V*(s) = 0 for all s in S
Loop:
  new_values = {}
  for s in S:
    if s in S_G:
      new_values[s] = 0.
    else:
      new_values[state] = max(Q*(state, a) for a in A(s))
    if max(|new_values[s] - V*(s)) for s in S) <= epsilon:
      break
  V* = new_values

# Collect policy actions
pi[s] = argmax(Q*(s, a) for a in A(s)
```

Exercise: From Conservatorio 7 to Celoria 18

Suppose to model the road from Via Conservatorio 7 to via Celoria 18 as a sequence $\langle 1, 2, \ldots, n-1, n \rangle$ locations.

For each location you can choose to take one of this options:

- `walk`: when you walk you always spend 2 minutes to go to $i + 1$;
- `bus`: if you take a bus you spend 1 minute to go to $i + 2$, but with probability α you need to wait the bus. There's not bus traveling to j if $j > n$;
- `train` if you take the train you spend 1 minute to go to $2i$, but with probability β you need to wait the train. There's not train traveling to j if $j > N$;

Your goal is to reach n as fast as possible.