Exercises: Reinforcement learning

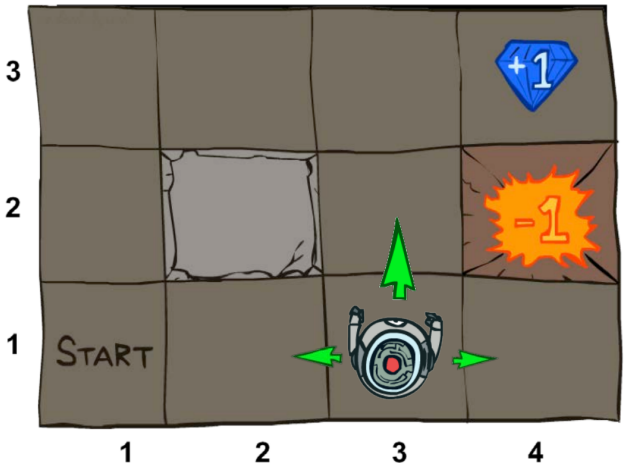
Statistical Planning and Reinforcement Learning

The purpose of these exercises is to help you understand fundamental concepts in reinforcement learning.

1 Practical exercises

1. Implement the grid world environment described during the lecture (Fig. 1). The environment has 12 states. Every state corresponds to a position in a 3×4 grid, except for the *absorbing state*. The grid world has one *wall* (not represented by a state), one *goal state*, and one *trap state*. The agent has four actions, which correspond to moving up, down, left, or right. An invalid action (for instance, an action directed at a wall or outside the grid) leaves the state unchanged. The agent receives reward 1 upon taking an action at the goal state, and reward -1 upon taking an action at the trap state. In every other case, the agent receives zero reward. Note that the agent does not receive a reward upon moving *into* a goal state or trap state. Upon taking an action at a goal state or a trap state, the agent moves to the absorbing state. Every action taken at the absorbing state leads to the absorbing state, which also does not provide rewards.

Figure 1: Grid world. Image from [Klein et al., 2019]



Using your favourite programming language, try to mimic the Python interface presented in Listing 1.

Listing 1: Reinforcement learning environment.

```
class EnvironmentModel:
    def __init__(self, n_states, n_actions, seed=None):
        self.n_states = n_states
        self.n_actions = n_actions

        self.random_state = np.random.RandomState(seed)

    def p(self, next_state, state, action):
        raise NotImplementedError()

    def r(self, next_state, state, action):
        raise NotImplementedError()

    def draw(self, state, action):
        p = [self.p(ns, state, action) for ns in range(self.n_states)]
```

```
        next_state = self.random_state.choice(self.n_states, p=p)
        reward = self.r(next_state, state, action)

        return next_state, reward


class Environment(EnvironmentModel):
    def __init__(self, n_states, n_actions, max_steps, dist, seed=None):
        EnvironmentModel.__init__(self, n_states, n_actions, seed)

        self.max_steps = max_steps

        self.dist = dist
        if self.dist is None:
            self.dist = np.full(n_states, 1./n_states)

    def reset(self):
        self.n_steps = 0
        self.state = self.random_state.choice(self.n_states, p=self.dist)

        return self.state

    def step(self, action):
        if action < 0 or action >= self.n_actions:
            raise Exception('Invalid action.')

        self.n_steps += 1
        done = (self.n_steps >= self.max_steps)

        self.state, reward = self.draw(self.state, action)

        return self.state, reward, done
```

The class *EnvironmentModel* represents a model of an environment. The constructor of this class receives a number of states, a number of actions, and a seed that controls the pseudorandom number generator. Its subclasses must implement two methods: p and r. The method p returns the probability of transitioning from *state* to *next_state* given *action*. The method r returns the expected reward in having transitioned from *state* to *next_state* given *action*. The method *draw* receives a pair of state and action and returns a state drawn according to p together with the corresponding expected reward. Note that states and actions are represented by integers starting at zero. We highly recommend that you follow the same convention, since this will facilitate immensely the implementation of reinforcement learning algorithms. You can use a Python dictionary (or equivalent data structure) to map (from and to) integers to a more convenient representation when necessary.

The class *Environment* represents an interactive environment and inherits from *EnvironmentModel*. The constructor of this class receives a number of states, a number of actions, a maximum number of steps for interaction, a probability distribution over initial states, and a seed that controls the pseudorandom number generator. Its subclasses must implement two methods: p and r, which were already explained above. This class has two new methods: *reset* and *step*. The method *reset* restarts the interaction between the agent and the environment by setting the number of time steps to zero and drawing a state according to the probability distribution over initial states. This state is stored by the class. The method *step* receives an action and returns a next state drawn according to p, the corresponding expected reward, and a flag variable. The new state is stored by the class. This method also keeps track of how many steps have been taken. Once the number of steps matches or exceeds the pre-defined maximum number of steps, the flag variable indicates that the interaction should end.

The grid world environment may be implemented as a subclass of the class *Environment*, which should implement the methods p and r. These methods are required by the reinforcement learning algorithms that we will cover in the next lecture.

Note that, in general, agents may receive rewards drawn probabilistically by an environment, which is not supported in this simplified implementation.

2. Implement a method called *render* that is able to represent the current state of your new environment (pictorially or textually) and a function that allows interacting with this environment using keyboard inputs. Try to mimic the Python interface presented in Listing 2.

Listing 2: Interacting with a reinforcement learning environment.

```
actions = ['8', '2', '4', '6'] # Numpad directions
state = env.reset()
env.render()

done = False

while not done:
    c = input('\nMove: ')
    if c not in actions:
        raise Exception('Invalid action')

    state, r, done = env.step(actions.index(c))
    env.render()
```

3. Implement a simple game of your choice using the same interface that you used to implement the grid world environment in the previous exercise. After the next lecture, you will be able to find an optimal policy for this game.

2 Theoretical exercises

The exercises marked by asterisks are significantly more challenging.

1. Consider the task of developing a reinforcement learning agent that plays chess against a fixed opponent. Explain how you would model this reinforcement learning problem in terms of states, actions, state transition model, reward model, and discount factor. Note that the state that follows an action of the agent should contain the move made by the opponent, since that is when your agent will make decisions. For the purposes of this exercise, do not worry about whether an optimal policy can be found efficiently.

2. Consider the grid world environment that you implemented earlier, and suppose that the agent must now obtain a *key* from some specific grid position in order to obtain a positive reward upon taking an action at the goal state. How should the states be changed in order for them to possess the Markov property?

3. Show that adding a constant to all the rewards in any given reinforcement learning problem simply adds a constant to the value of each state

4. (*) Show the recursivity of the action value function (lecture slides, page 32)

5. (*) Show the relationship between V^π and Q^π (lecture slides, page 33)