COMP3411/COMP9814 23T1
Artificial Intelligence

Assignment 2 - Reward-based learning agents

Due: Week 7, Thursday, 30 March 2023, 10 pm

1 Activities

In this assignment, you are asked to implement a modified version of temporal-difference Q-learning and SARSA algorithms. The modified algorithms will use a modified version of the ϵ-greedy action selection method.

You will use another version of gridworld. **The new code can be found here**.

The modification of the method includes the following two aspects:

- Random numbers will be obtaining sequentially from a file.
- When exploring, the worst action will be taken instead of a random one.

The random numbers are available in the file `random_numbers.txt`. The file contains 500k random numbers between 0 and 1 with seed = 999 created with numpy.random.random as follows:

```
import numpy as np

np.random.seed(999)
random_numbers=np.random.random(500000)
np.savetxt('random_numbers.txt', random_numbers)
```

1.1 Part 1 (5 marks)

Modifying TODO sections in Qlearning.py, create an action selection method that receives a state as an argument and returns the action. Consider the following:

- The method must use sequentially the random number in the provided file.
- In case of a random number $rnd < \epsilon$ the method return an exploratory action which in this case will be the worst action, i.e., the one with the lowest Q-value. Otherwise, the method returns the greedy action, i.e., the one with the greatest Q-value.
- If more than one action shares the lowest or greatest Q-value, the first occurrence should be considered.
- To read random numbers you could either load the numbers into an array (or similar) structure and read each position keeping an index or read the file line by line.
- The defined method will be used in do_step method when selecting actions.

1.2 Part 2 (5 marks)

Modifying TODO sections in SARSA.py, create the on-policy SARSA method. Consider the following for the implementation:

- Use the same modified version of ϵ-greedy action selection method as in Q-learning.
- In this case, you should start reading random numbers from the beginning, however, take into account that SARSA will use two random numbers at each step.
- Implement the method do_step to perform the update of state-action pairs using SARSA.

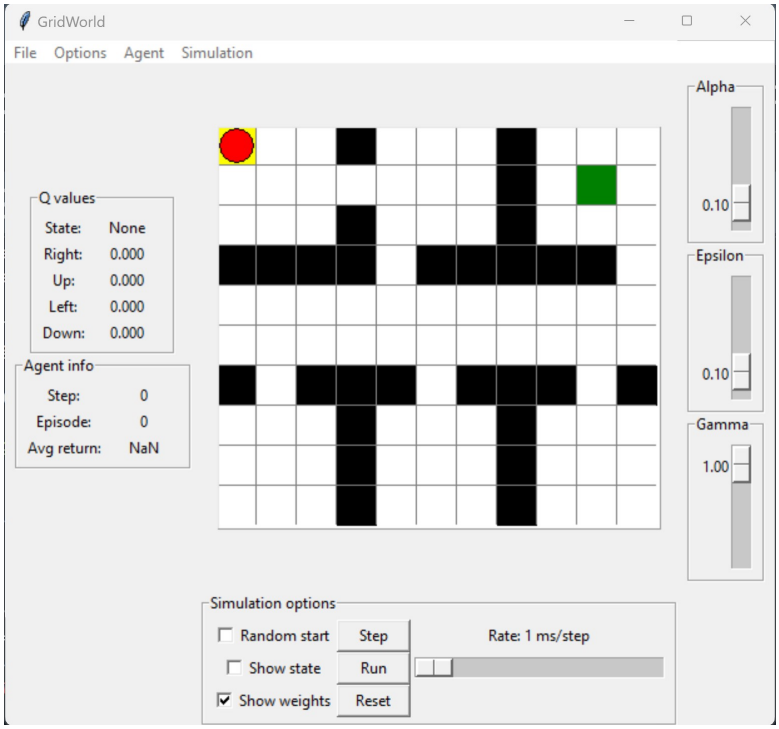

Figure 1: Gridworld used to simulate 50k steps with both Q-learning and SARSA.

1.3 Testing your code

Every time you run Q-learning or SARSA the code will create (or overwrite) a file called `stepsperepisode.txt`. This file contains the number of actions taken in each episode, i.e., the actions needed to reach the goal.

For instance, if you want to test your code, you can use the gridworld shown in Fig. 1. The agent always starts at the top-left corner.

This gridworld is saved in the file `worlds/rlworld.gwd`. The file `stepsperepisodeQLearning.txt` contains the simulation results for 50k steps with Q-learning, while `stepsperepisodeSARSA.txt` contains the results for 50k steps with SARSA. In both cases, the parameters used are: learning rate $\alpha = 0.1$, discount factor $\gamma = 1.0$, and the modified ϵ-greedy action selection method with $\epsilon = 0.1$. You can use `diff` to compare your output.

In case you want to see a graphical comparison, you could plot the learning

curves as shown in Fig. 2 for the first 400 episodes. The plot was created with the following script:

```
import matplotlib.pyplot as plt
import numpy as np

stepsQLearning=np.loadtxt("stepsperepisodeQLearning.txt")
stepsSARSA=np.loadtxt("stepsperepisodeSARSA.txt")

limit = 400
plt.plot(stepsQLearning[0:limit], label="Q-learning")
plt.plot(stepsSARSA[0:limit], label="SARSA")

plt.xlim(0,limit)
plt.legend(loc="best",prop={"size":10})
plt.title("Steps performed by Q-learning and SARSA")
plt.xlabel("Episodes")
plt.ylabel("Steps")
plt.grid()
plt.show()
```

To mark your submission, different gridworlds and learning parameters will be used as test cases.

2 Submitting your assignment

Make sure you only modify TODO sections of Qlearning.py and SARSA.py. Please do not modify anything outside the TODO sections. Keep methods or other files on the project the same.

Your submission will consist of two files Qlearning.py and SARSA.py which should implemented in Python what is requested in Parts 1 and 2.

To hand in, log in a School of CSE Linux workstation or server, make sure that your files are in the current working directory, and use the Unix command:

```
> give cs3411 assign2 Qlearning.py SARSA.py
```

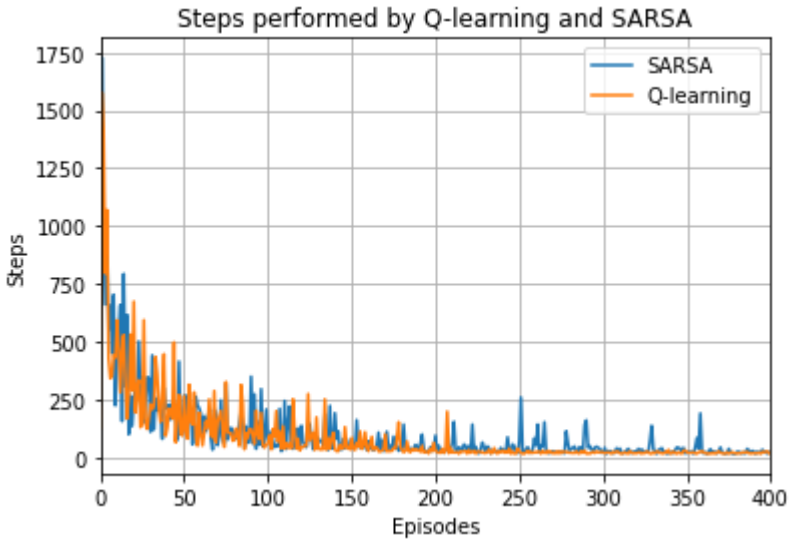

Figure 2: Steps performed each learning episode with both Q-learning and SARSA.

Please make sure your code works on CSE's Linux machines and generates no warnings. Remove all test code from your submissions.

Once give has been enabled, you can submit as many times as you like – later submission will overwrite earlier ones. When give accepts your submission please read the give acceptance receipt carefully and take a screenshot of it for future references.

Late submission penalty: UNSW has a standard late submission penalty of 5% per day, capped at five days (120 hours) from the assessment deadline, after that students cannot submit the assignment.

3 Deadline and questions

Deadline: Week 7, Thursday 30th of March 10:00pm. Please use the forum on the course website to ask questions related to the project. You should send your questions to cs3411@unsw.edu.au only in the case that they might show the answer to others and, therefore, they cannot be asked in public.

4 Plagiarism policy

Group submissions are not allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for any similar projects from previous years) and serious penalties will be applied, particularly in the case of repeat offences.

Do not copy from others. Do not allow anyone to see your code. Please refer to the UNSW Policy on Academic Honesty and Plagiarism if you require further clarification on this matter.

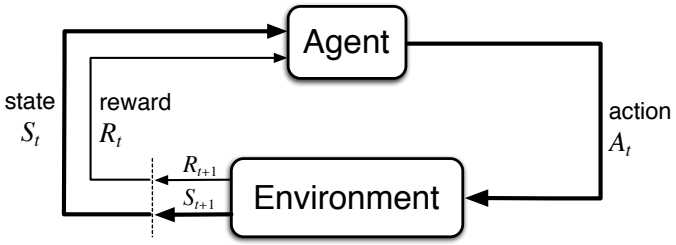

Figure 3: Reinforcement learning loop between the agent and the environment [1]. At each time-step the RL agent from s_t selects an action a_t to perform and receives from the environment the state s_{t+1} and a reward signal r_{t+1}.

5 Additional material

5.1 Reinforcement learning framework

Reinforcement learning (RL) [1] is a machine learning method based on behavioral psychology. It allows an agent to learn how to perform new tasks by exploring the environment and observing state modifications and possible rewards. The method is oriented toward goals in which an agent, either human or robotic, tries to maximize the long-term cumulative reward by iterative interactions with the environment. Figure 3 shows the classic interaction loop between an RL agent and the environment.

An RL problem is comprised of:

- A policy: that defines how an agent selects an action aiming to maximize the reward signal obtained.
- A reward signal: that establishes a definition of positive and/or negative events, i.e., the aims to be achieved by the RL agent.
- A value function: that specifies how good the reward signal might be over time from one state or a state-action pair.
- Optionally, a model of the environment: that allows the agent to infer what will be the next state and reward, given any state and action.

Based on this model, an RL agent is able to learn the optimal policy that allows selecting the action leading to the highest cumulative reward given the value function.

5.2 Markov decision processes

Markov decision processes (MDPs) are the base of RL tasks. In an MDP, transitions and rewards depend only on the current state and the selected action by the agent [2]. In other words, a Markov state contains all the information related to the dynamics of a task, i.e., once the current state is known, the history of transitions that led the agent to that position is irrelevant in terms of the decision-making problem.

An MDP is characterized by the 4-tuple $< S, A, \delta, r >$ where:

- S is a finite set of states,
- A is a set of actions,
- δ is the transition function $\delta : S \times A \to S$, and,
- r is the reward function $r : S \times A \to \mathbb{R}$.

At each time t, the agent perceives the current state $s_t \in S$ and selects the action $a_t \in A$ to perform it. The environment returns the reward $r_t = r(s_t, a_t)$ and the agent transits to the state $s_{t+1} = \delta(s_t, a_t)$. The functions r and δ depend only on the current state and action, i.e., it is a process with no memory.

To formalize the problem we should consider that the agent wants to learn the policy $\pi : S \to A$ which, from a state s_t, produces the greatest accumulated reward over time. Therefore, the value can be computed as follows:

$$r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i \cdot r_{t+1} = V^{\pi}(s_t) \tag{1}$$

where $V^{\pi}(s_t)$ is the accumulated reward by following the policy π from an initial state s_t and γ is a constant ($0 \leq \gamma < 1$) which determines the relative importance of immediate rewards with respect to the future rewards.

5.3 Temporal-difference learning

Actions are selected according to a policy π, which in psychology is called a set of stimulus-response rules or associations [3]. Thus, the value of taking

an action a in a state s under a policy π is denoted $q^{\pi}(s,a)$ which is also called the action-value function for a policy π.

In essence, to solve an RL problem means to find a policy that collects the highest reward possible over the long run. If there exists at least one policy which is better or equal than all others this is called an optimal policy. Optimal policies are denoted by π^* and share the same optimal action-value function which is denoted by q^* and defined as:

$$q^*(s,a) = \max_{\pi} q^{\pi}(s,a) \tag{2}$$

This optimal action-value function can be solved through the Bellman optimality equation for q^* as follows:

$$q^*(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} q^*(s',a')] \tag{3}$$

where s is the current state, a is the taken action, s' is the next state reached by performing action a in the state s, and a' are possible actions that could be taken in s'. In the equation, p represents the probability of reaching the state s' given that the current state is s and the selected action is a, and r is the received reward for performing action a in the state s to reach the state s'.

For solving Eq. (3) diverse learning methods exist, e.g., SARSA and Q-learning. The on-policy method SARSA [4] solves the Eq. (3) considering transitions from state-action pair to state-action pair as shown in Eq. (4). In the Q-learning method, state-action values are updated according to the Eq. (5) [5,6]. Algorithm 1 shows a general learning method with an iterative update of $Q(s,a)$ based on temporal-difference learning [1].

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \tag{4}$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A(s_{t+1})} Q(s_{t+1},a) - Q(s_t,a_t)] \tag{5}$$

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

Algorithm 1 General algorithm of temporal-difference learning.

Initialize $Q(s, a)$ arbitrarily
for (each episode) **do**
    Choose an action a_t
    repeat
        Take action a_t
        Observe reward r_{t+1} and next state s_{t+1}
        Choose an action a_{t+1}
        Update $Q(s_t, a_t)$
        $s_t \leftarrow s_{t+1}$
        $a_t \leftarrow a_{t+1}$
    until s is terminal
end for

[2] M. L. Puterman, *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[3] S. Kornblum, T. Hasbroucq, and A. Osman, "Dimensional overlap: cognitive basis for stimulus-response compatibility–a model and taxonomy.," *Psychological review*, vol. 97, no. 2, p. 253, 1990.

[4] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[5] C. J. Watkins, *Learning from Delayed Rewards*. Doctoral dissertation, University of Cambridge, 1989.

[6] P. Dayan and C. Watkins, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.