VRIJE UNIVERSITEIT BRUSSEL

Computational Game Theory
Assignment 3

Deadline: 10 August 2022, 23:59

Submission Instructions

- We have organized a Github classroom for this assignment. Use this link `https://classroom.github.com/a/T-q-ZnXd` to accept the assignment and to create inside the classroom your own private repository. Commit there your code and report (in PDF form), before the deadline.
- Follow this naming convention for your PDF report: *Lastname_Firstname_studentNo_affiliation.pdf*.
- Late submissions will be penalized (-1 point/day).
- This is an **individual** assignment, sharing code is considered plagiarism and will be punished. Any other online sources used should be declared in the report. Plagiarism will not be tolerated.
- A template for the code is available once you accept the assignment. Read the `README.md` file. You can create additional classes, methods or functions. However, you need to complete and use the existing class/methods/functions without renaming them. Failing to do so will be penalized.
- If you have any questions send an email to: `andries.rosseau@ai.vub.ac.be` and `raphael.avalos@vub.be` with the subject `[CGT2021][Assignment 3]`.

1 Single-agent RL: Frozen Lake

1.1 The Frozen Lake environment

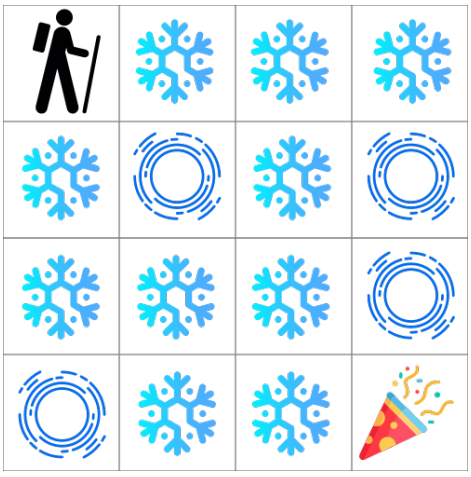

Figure 1: Representation of the Frozen Lake environment. The goal of this game is to go from the starting state (top left) to the goal state (bottom right) by walking only on frozen tiles and avoiding holes. There are four directions you can move to, however, the ice is slippery, so you won't always move in the direction you intend to; this is a *stochastic* environment!

For this exercise, you will have to implement a Reinforcement Learning agent that plays Frozen Lake. We use the implementation of the FrozenLake environment from OpenAI's Gym, where it is described as follows:

> `Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the`

```
ice has melted. If you step into one of those holes, you'll fall into the
freezing water. At this time, there's an international frisbee shortage, so
it's absolutely imperative that you navigate across the lake and retrieve
the disc. However, the ice is slippery, so you won't always move in the
direction you intend.
The surface is described using a grid like the following

    SFFF
    FHFH
    FFFH
    HFFG

S : starting point, safe
F : frozen surface, safe
H : hole, fall to your doom
G : goal, where the frisbee is located

The episode ends when you reach the goal or fall in a hole.
You receive a reward of 1 if you reach the goal, and zero otherwise.
```

The agent can move in the 4 directions (0: Left; 1: Down; 2: Right; 3: Up). However, the action has only a probability of 1/3 to succeed! With probability 2/3 the agent will move in one of the two 'perpendicular' directions (1/3 each). The agent does not know the randomness of the environment (it needs to discover this by interaction).

To give you an example, if your agent selects action 0, the environment implements action 'LEFT', but since the environment is also stochastic, there is a 33% chance of actually going left, a 33% chance of going up, and a 33% chance of going down. There is 0% chance of going in the reverse direction, in this case RIGHT. If you run into a wall, you stay in the same place.

1.2 The SARSA algorithm

For this exercise, you will implement one of the most iconic Reinforcement Learning algorithms of all time: SARSA [2], or *State-Action-Reward-State-Action*. Take a look in the course material or anywhere online to write this algorithm yourself. Do not forget to update the exploration rate at the end of every episode:

```
epsilon = max(epsilon * decay_rate, epsilon_min)
```

You start off with a completely random agent. Because it is random, the agent will only explore its environment, and learn from its interactions with it. After a while, the agent learns that certain actions in certain states are better than others. The exploration rate decreases over time, to allow the agent to gradually exploit the learned policy more often, until it converges to a policy that approximately maximizes the expected return. Keep in mind that the stochasticity of the environment might lead to a policy that does not always succeed.

1.3 Your part

To complete this exercise, we expect you to write the functions in the `frozen_lake` folder. A large part of the code has already been set up, you mostly have to write the agent parts. After completing the code, run it and train an agent that performs well. Give your algorithm `30 000` episodes to converge. Fine-tune your hyperparameters, such as the learning rate and your exploration parameters (ϵ_{max}, ϵ_{min}, ϵ_{decay}). You can set a discount factor of 0.99. You should mention your final parameters in your report.

The policy should be evaluated over 32 evaluation episodes every 1000 training episodes. A plot containing the return of the agent during training and the return at all evaluation times should be included in the report. This plot should be averaged over 20 full training runs with standard deviations as error bars.

After that, you should write code that lets you inspect a trained policy in human-readable form (e.g., a grid world with arrows), and provide the (interesting) policies in the report. Inspect your trained policies to find potentially different final strategies. Discuss some interesting strategies that you notice the agent has learned.

2 Multi-agent RL: stochastic hill-climbing game

For this exercise, two agents will play the stochastic hill-climbing game (see lectures). You will compare the performance of two independent learning agents against two agents using commitment sequences [1]. You will use one of the stochastic hill-climbing games presented here. To know which one, take the last number of your student number. If that number is 0, 3, 6 or 9 you have Table 1. If it is 1, 4 or 7, you have Table 2. If it is 2, 5 or 8, you have Table 3.

	b_1	b_2	b_3
a_1	$(0, 2)$	$(-5, -75)$	$(-2, -18)$
a_2	$(-5, -75)$	$(4, -10)$	$(2, -10)$
a_3	$(-5, -15)$	$(-5, -15)$	$(0, -10)$

Table 1: Stochastic Climbing Game 1: (x, y) means a 50% chance of getting either x or y as a payoff.

	b_1	b_2	b_3
a_1	$(6, 12, -15)$	$(-6, -4, -110)$	$(0, 10, -20)$
a_2	$(-6, -4, -110)$	$(15, -10, -14)$	$(0, -5, -7)$
a_3	$(-2, 2, -30)$	$(0, 10, -40)$	$(-6, -5, -4)$

Table 2: Stochastic Climbing Game 2: (x, y, z) means a 1/3 chance of getting either x, y or z as a payoff.

	b_1	b_2	b_3
a_1	$(0.4 : -3.5, 0.6 : 4)$	$(0.25 : -46, 0.75 : -38)$	$(0.6 : -6, 0.4 : -16)$
a_2	$(0.25 : -46, 0.75 : -38)$	$(0.8 : -5, 0.2 : 5)$	$(0.8 : -5, 0.2 : 0)$
a_3	$(0.7 : -4, 0.3 : -17)$	$(0.6 : -6, 0.4 : -16)$	$(0.8 : -6, 0.2 : -1)$

Table 3: Stochastic Climbing Game 3: $(a : x, b : y)$ means a chance of getting x as a payoff, and b chance of getting y as payoff.

2.1 Commitment sequences

A commitment sequence is some list of time slots $(t_0; t_1; :::)$ for which an agent is committed to taking the same action. A few conditions need to be met for this technique:

- An exponentially increasing time interval between successive time slots for the same action pair;
- A common global clock to time the actions;
- A shared protocol for defining the commitment sequences.

If all the agents have the same protocol for defining these sequences, then they are committed to selecting the same joint-action for every time point in the sequence. Although each agent does not know the action choices of the other agents, it can be certain that observed rewards will be statistically stationary and represent unbiased samples for the reward distribution of the same joint action. In order to allow an arbitrarily high number of joint actions and consequently commitment sequences to be considered as the agent learns, it is necessary that the sequences are finite or have an exponentially increasing time interval $\delta_i = t_{i+1} - t_i$ between successive time slots. A sufficient condition is that $\gamma\delta_{i+1} \geq \delta_i$ where $\gamma > 1$ for all $i > i_0$ for some predefined constant i_0. An example of creating successive increments would be:

$$\delta_{i+1} = \lfloor \frac{c\delta_i + c - 2}{c - 1} \rfloor$$

where $c > 1$ is the increment factor and $\lfloor X \rfloor$ indicates rounding down to an integer value. In this exercise we will use an increment factor of 6 and therefore the following rule with $\delta_{-1} = -1$ and the first sequence starting at time step 0.

$$\delta_{i+1} = \lfloor \frac{6\delta_i + 4}{5} \rfloor \tag{1}$$

The first such sequence starts at time-step 0 and contains (0; 2; 5; 9; 14; 20; 28; 38 ...). The second sequence therefore starts at time slot 1. To prevent any 2 sequences from selecting the same time slot, each sequence excludes the existing ones from counting. As time-step 1 does not belong to a sequence, a new sequence is created, so the second sequence starts at time-step 1 and contains (1; 4; 8; 13; 19; 26; 35; 46; ...).
The value of a joint action can be estimated using the average reward received during the part of the sequence that has been completed so far. Longer sequences provide more reliable estimates. You will evaluate the simplest approach possible where the agents maintain a running reward average for all the active commitment sequences.

At the beginning of each sequence, the agents need to choose an action. With probability p the agents will choose to explore by randomly and uniformly selecting an action, and with the probability $1 - p$ the agents will exploit by selecting the action of the sequence currently considered optimal. To be considered reliable, and therefore a candidate for exploitation, a reward estimate needs to be computed from at least N_{min} samples. Finally, to ensure sufficient exploration at the beginning the first N_{init} sequences will start by selecting a random action.

At the end of the training, the actions that are chosen for the final (joint) policy are both agents' exploitative actions. This action for each agent is the one associated with the sequence with the highest estimated reward.

2.2 Your part

You will compare independent learners versus commitment learners, and report probabilities of converging to the optimal joint action after $t_{max} = 1000$ steps. You will approximate this probability by repeating each experiment at least 100 times.

First, implement two independent learners to play the game. Look at the `iql` files in the `commitment` folder of the template. You can reuse some code of the first exercise. You should implement the independent agents according to the code documentation provided. You should also implement the runner which trains your agents. Do not forget that there is a README file to help you as well and you can reuse parts of the first exercise.

Instead of SARSA or Q-learners which use a learning rate in their update step, you will use agents that simply take the average of the sampled rewards per action to find an estimate of the Q-value. This is to more closely follow the original work from Kapetanakis et al. [1] on commitment sequences for the second part of this exercise. Note that using averages is not that different from using the SARSA or Q-learning update rule in a single-state (bandit) setting, since the sample average after k samples can be written as:

$$\begin{aligned}
\hat{Q}_k &\equiv \frac{1}{k}\sum_{i=1}^{k} R_i \\
&= \frac{1}{k}\left(\sum_{i=1}^{k-1} R_i + R_k\right) \\
&= \frac{1}{k}\left((k-1)\hat{Q}_{k-1} + R_k\right) \\
&= (1 - 1/k)\hat{Q}_{k-1} + R_k/k \\
&= \hat{Q}_{k-1} + \frac{1}{k}(R_k - \hat{Q}_{k-1})
\end{aligned} \tag{2}$$

where at the end you recognize the SARSA or Q-learning update rule, with learning rate equal to $1/k$ (there is no difference between SARSA or Q-learning anymore, because there is no future state and therefore no max or next Q-sample to include). So taking the average is basically applying Q-learning with a diminishing learning rate per sample (and also a slightly different initialization). You can use this derivation in your code also to efficiently calculate averages with each new sample.

Tune the hyper-parameters reasonably and evaluate how Independent Learners perform by reporting the probability of converging to the optimal joint action. Explain your results.

Second, implement two agents with commitment sequences (see above, the lectures, or Kapetanakis et al. [1]). You should implement the independent agents according to the code documentation provided. You should also implement the runner which trains your agents. Use the following parameter values:

- $t_{max} = 1000$;
- $N_{min} = 10$;
- $N_{init} = 10$;
- $p = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$;

where you should try all values of p that are given in brackets. Plot the probability that your agents converge to the optimal joint action across the different values of p. Explain your results.

Report how the strategies of the independent agents and the commitment agents (here only for $p = 0.9$) evolve through time, i.e., how they 'climb' the hill throughout their training. To do so, let's assume you need 1000 episodes for your algorithms to converge, then for every 100 episodes, save the joint action that would be produced by the 'exploit' strategy for the commitment learners and the independent learners. At the end, report the most frequently occurring joint-action per 100 episodes (and their percentage of occurrence) to give you an indication of how the independent and commitment agents move through the reward table. Discuss the differences of commitment learning compared to the independent agents, and write down your thoughts on why and how these differences occur.

References

[1] S. Kapetanakis, D. Kudenko, and M. Strens. Learning of coordination in cooperative multi-agent systems using commitment sequences. *Artificial Intelligence and the Simulation of Behavior*, 1(5), 2004.

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