COMP 4010A
Introduction to Reinforcement Learning
Assignment 2

Instructor: Junfeng Wen (junfeng.wen [AT] carleton.ca)

Fall 2025
School of Computer Science
Carleton University

Deadline: 11:59 pm, Monday, Oct. 27, 2025

Instructions: **Submit the following three files** to Brightspace for marking

- A Python file `A2codes.py` that includes all your implementations of the required functions
- A pip requirements file named `requirements.txt` that specifies the running environment including a list of Python libraries/packages and their versions required to run your codes. The current template uses Python=3.10.
- A PDF file `A2report.pdf` that includes all your answers to the written questions. It should also specify your name and student ID. Please clearly specify question/sub-question numbers in your submitted PDF report so TAs can see which question you are answering.

Do not submit a compressed (zip) file, or it may result in a mark deduction. We recommend trying your code using Colab or Anaconda/Virtualenv before submission.

Rubrics: This assignment is worth 20% of the final grade. Your codes and report will be evaluated based on their scientific qualities including but not limited to: Are the implementations correct? Is the analysis rigorous and thorough? Are the codes efficient and easily understandable (with comments)? Is the report well-organized and clear?

Policies:

- You must write your answers **individually**.
- You may consult others (classmates/TAs) about general ideas but don't share codes/answers. Please specify in the PDF file any individuals you consult for the assignment. Any student found to cheat or violate this policy will receive a score of 0 for this assignment.
- You can **only** ask large language models (LLMs) general functionality questions. For example, "How do I set all negative numbers in a NumPy array to zeros?" If you used LLMs, clearly show us how you use them (either include the prompts and replies in the report, or include URL links showing chat history in the report). Any student found to cheat or violate this policy will receive a score of 0 for this assignment.

- Remember that you have **three** excused days *throughout the term* (rounded up to the nearest day), after which no late submission will be accepted.
- Specifically for this assignment, you can use libraries with general utilities, such as matplotlib, numpy/scipy, and pandas. **However, you must implement everything by yourselves without using any pre-existing implementations of the algorithms or any functions from an RL/ML library**. The goal is for you to really understand, step by step, how the algorithms work.

Question 1 (5%) Monte-Carlo Methods

(a) Suppose we have three states A, B, C in an MDP and the discount factor is $\gamma = 0.5$. The agent with a fixed policy collects the following experiences from several episodes with various lengths (states followed by immediate rewards)

- $B, 1, A, 4$
- $A, 0, B, 2, A, 4$
- $C, 2, A, 2$
- $C, 1, B, 2, B, 2$
- $A, 1, C, 0, C, 2$

(a.1) (1%) When estimating the state-value of the agent using **every-visit** Monte-Carlo prediction, what are the estimated values $V(A), V(B), V(C)$? Show your calculation.

(a.2) (1%) When estimating the state-value of the agent using **first-visit** Monte-Carlo prediction, what are the estimated values $V(A), V(B), V(C)$? Show your calculation.

(b) (1%) Write a block of pseudocode that estimates the **action value function** $q_\pi(s, a)$ of a given/fixed policy π using **first-visit** Monte-Carlo method.

(c) Consider the following MDP where every state transition receives a reward of -1 and discount factor is $\gamma = 1$:

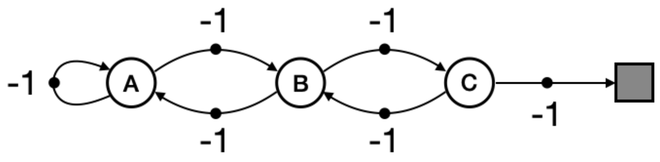


Figure 1: A simple MDP with 2 actions

Then $-v_\pi(X)$ of any state X essentially represents the expected number of steps a policy π needs to reach the terminal state from state X. The goal is to maximize $v_\pi(X)$, which is equivalent to minimize the number of steps before reaching the terminal state.

(c.1) (0.5%) What is the expected number of steps a equiprobable random policy (taking all allowed actions equally likely) needs to reach the terminal state from state A? (In other words, what is $-v_\pi(A)$ when π is a uniform policy?)

(c.2) (0.5%) Describe the optimal ϵ-soft policy for this MDP when ϵ is a small positive number.

(c.3) (1%) When $\epsilon = 0.1$, what is the expected number of steps the optimal ϵ-soft policy needs to reach the terminal state from state A? (In other words, what is $-v_\pi(A)$ when π the optimal 0.1-soft policy?)

Question 2 (4%) TD Learning

(a) (1%) Suppose we have three states A, B, C in an MDP and the discount factor is $\gamma = 1$. The agent with a fixed policy collects the following experiences from several episodes with various lengths (states followed by immediate rewards)

- $A, 1, B, 2$
- $A, 0, B, 2, B, 0$
- $B, 1$
- $C, 1, C, 0, A, 4$

Suppose we estimate the state-values using TD(0) and the initial values are all zeros. When the step size $\alpha = 0.5$, what are the estimated values $V(A), V(B), V(C)$ after these episodes? Show your calculation.

(b) (1%) Consider the following MDP where every non-terminal state has two actions denoted by colors red and blue. The start state is always X and the terminal state is denoted as T. The numbers indicate the rewards of the transitions and the discount actor is $\gamma = 1$.

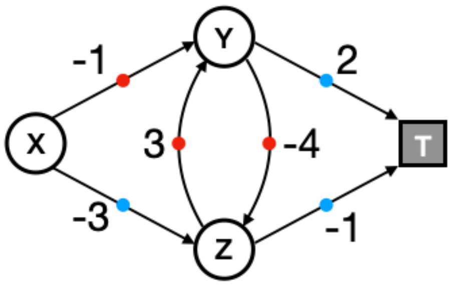


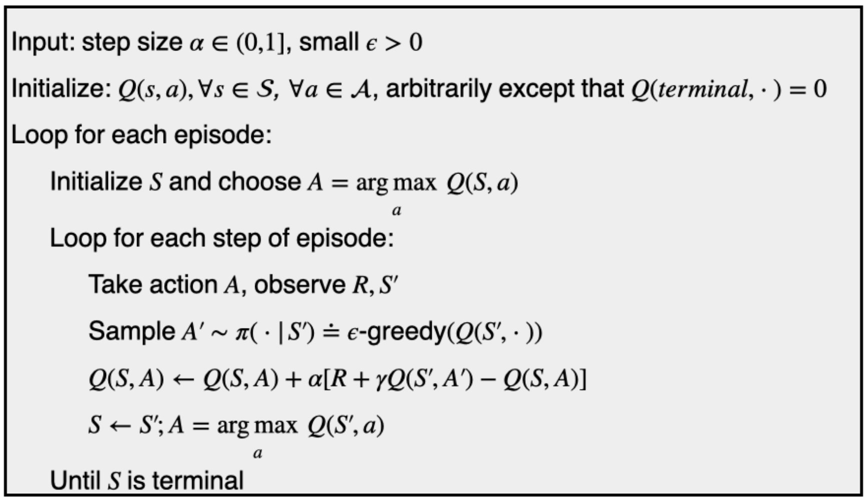
Figure 2: A simple MDP with 2 actions per non-terminal state

Suppose we apply the SARSA control algorithm to this MDP and observe the transitions shown as the first column of the following table. For example, "X, R, -1, Y" means the agent starts from state X and takes action red (R), then observes a reward -1 and transitions to state Y.

State	X		Y		Z	
Action	Red	Blue	Red	Blue	Red	Blue
Initial	0	0	0	0	0	0
X, R, -1, Y						
Y, R, -4, Z						
Z, B, -1, T						
X, B, -3, Z						
Z, B, -1, T						

All initial action-values are zeros and the learning rate is $\alpha = 0.5$. Complete the table by filling the updated values after each SARSA update.

(c) Consider the following algorithm



```
Input: step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialize: $Q(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize $S$ and choose $A = \arg\max_a Q(S,a)$
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        Sample $A' \sim \pi(\cdot \mid S') \doteq \epsilon$-greedy($Q(S',\cdot)$)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'; A = \arg\max_a Q(S',a)$
    Until $S$ is terminal
```

Figure 3: A modified TD algorithm

(c.1) (1%) Is this algorithm on-policy or off-policy? Why?

(c.2) (1%) Will this algorithm converge to the optimal action-values q_*? Why or why not?

Question 3 (5%) Q-Learning

In this question, you will implement the Q-learning algorithm, and evaluate its results on a simple gridworld environment (see Fig. 4 with optimal values), which is an **episodic** task with zero reward everywhere except for a reward of +1 when reaching on the goal state/cell. Once the goal is reached, the episode is terminated.

Your implementation must be able to handle arbitrary numbers of (discrete) states $|\mathcal{S}| > 0$ and actions $|\mathcal{A}| > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (3%) Implement a Python function

```
Pi, q = QLearning(env, gamma, step_size, epsilon, max_episode)
```

that takes the following arguments

- An environment `env`, which has properties like number of states/actions and the goal state. It (roughly) follows the API of the gymnasium.Env (check `A2helpers.py` for more details). It can be reset using the `env.reset()` function and proceed to next state with the `env.step()` function
- A scalar `gamma` (i.e., the discount factor $\gamma \in [0, 1]$)
- A scalar `step_size` (i.e., $\alpha \in (0, 1]$) for updating the action-values
- A small positive number `epsilon` (i.e., $\epsilon \in (0, 1)$) specifying the behavior policy
- An integer `max_episode` indicating the maximum number of episodes before the algorithm terminates

It returns an expanded (diagonalized) policy matrix `Pi` (i.e., $\Pi \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|}$) of an (approximately) optimal policy and a $|\mathcal{S}||\mathcal{A}| \times 1$ vector `q` representing the action-values of the policy. Note that the policy is derived from the action-values **after** finishing `max_episode` episodes of Q-learning.

Hints:

- Even though, in theory, the initial values can be set arbitrarily, it is a bad idea to set them all to the same value (e.g., all zeros). Think about what the program will do in the first episode when initialize the values this way.
- `env.step()` will return a tuple of *five* elements. Please pay extra attention to `terminated` and `truncated` and how they should be used.
- When the goal is set to state 0 and $\gamma = 0.9$, you should get a similar, but not necessarily identical, plot as Fig. 4 when calling the `plot_grid_world` function with your returned policy and action-values. You can also play with the parameters (`step_size`, `max_episode` and `epsilon`) to see different learning outcomes.

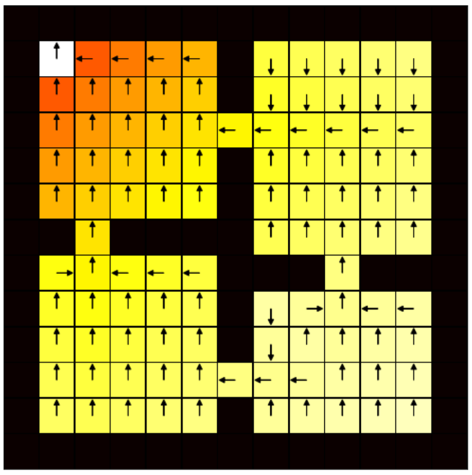


Figure 4: Four room gridworld with goal state at the top-left.

(b) (1%) In this part, you will evaluate your implementation with different parameters. Complete and submit the Python function

`step_size_results, epsilon_results, max_episode_results = runQLExperiments(env)`

as shown below:

```python
def runQLExperiments(env):

    def repeatExperiments(gamma=0.9, step_size=0.1, epsilon=0.1, max_episode=500):
        n_runs = 5
        RMSE = np.zeros([n_runs])
        for r in range(n_runs):
            Pi, q = QLearning(env, gamma, step_size, epsilon, max_episode)
            RMSE[r] = # TODO: compute RMSE(q, q_star)

        # TODO: compute and return the *average* RMSE over runs

    step_size_list = [0.1, 0.2, 0.5, 0.9]
    epsilon_list = [0.05, 0.1, 0.5, 0.9]
    max_episode_list = [50, 100, 500, 1000]

    step_size_results = np.zeros([len(step_size_list)])
    epsilon_results = np.zeros([len(epsilon_list)])
    max_episode_results = np.zeros([len(max_episode_list)])

    q_star = np.load('optimal_q.npy')

    # TODO: Set the following random seed to your *student ID*
    np.random.seed(_)

    # TODO: Call repeatExperiments() with different step_size in the step_size_list,
    #       *while fixing others as default*. Save the results to step_size_results.

    # TODO: Call repeatExperiments() with different epsilon in the epsilon_list,
    #       *while fixing others as default*. Save the results to epsilon_results.

    # TODO: Call repeatExperiments() with different max_episode in the max_episode_list,
    #       *while fixing others as default*. Save the results to max_episode_results.

    return step_size_results, epsilon_results, max_episode_results
```

It loops through different parameters and for each parameter configuration, repeats 5 runs of Q-learning. In each run, the returned **q** vector is compared with the true optimal action-values $\mathbf{q}_*$ (loaded from the `optimal_q.npy` file), based on root-mean-square error (RMSE).

Once completed, call the function with the following environment

```
env = FourRoom(goal=0)
```

whose optimal action-values $\mathbf{q}_*$ are given in the provided `optimal_q.npy` file. Then report the results in the following tables

Table 1: RMSE for different step size α

α	0.1	0.2	0.5	0.9
RMSE				

Table 2: RMSE for different ϵ

ϵ	0.05	0.1	0.5	0.9
RMSE				

Table 3: RMSE for different number of episodes

Max Episode	50	100	500	1000
RMSE				

Runtime. For efficient implementation, this function can be finished within one minute.

(c) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

Question 4 (6%) Planning

In this question, you will implement the Dyna-Q algorithm, and evaluate its results on a simple gridworld environment (see Fig. 4 with optimal values), which is an **episodic** task with zero reward everywhere except for a reward of +1 when landing on the goal state/cell. Once the goal is reached, the episode is terminated.

Your implementation must be able to handle arbitrary numbers of (discrete) states $|\mathcal{S}| > 0$ and actions $|\mathcal{A}| > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (4%) Implement a Python function

```
Pi, q = DynaQ(env, gamma, step_size, epsilon, max_episode, max_model_step)
```

that takes the following arguments

- An environment `env`, which has properties like number of states/actions and the goal state. It (roughly) follows the API of the gymnasium.Env (check `A2helpers.py` for more details). It can be reset using the `env.reset()` function and proceed to next state with the `env.step()` function
- A scalar `gamma` (i.e., the discount factor $\gamma \in [0, 1]$)
- A scalar `step_size` (i.e., $\alpha \in (0, 1]$) for updating the action-values
- A small positive number `epsilon` (i.e., $\epsilon \in (0, 1)$) specifying the ϵ-greedy behavior policy
- An integer `max_episode` indicating the maximum number of episodes before the algorithm terminates
- An integer `max_model_step` indicating the maximum number of planning update steps

It returns an expanded (diagonalized) policy matrix `Pi` (i.e., $\Pi \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|}$) of an (approximately) optimal policy and a $|\mathcal{S}||\mathcal{A}| \times 1$ vector `q` representing the action-values of the policy. Note that the policy is derived from the action-values **after** finishing `max_episode` episodes.

(b) (1%) In this part, you will evaluate your implementation with different parameters. Complete and submit the Python function

```
results = runDynaQExperiments(env)
```

as shown below:

```
def runDynaQExperiments(env):

    def repeatExperiments(gamma=0.9,
                          step_size=0.1,
                          epsilon=0.5,
                          max_episode=100,
                          max_model_step=10):
        n_runs = 5
        RMSE = np.zeros([n_runs])
        for r in range(n_runs):
            Pi, q = DynaQ(env, gamma, step_size, epsilon, max_episode, max_model_step)
            RMSE[r] = # TODO: compute RMSE(q, q_star)

        # TODO: compute and return the average RMSE over runs

    max_episode_list = [10, 30, 50]
    max_model_step_list = [1, 5, 10, 50]
    results = np.zeros([len(max_episode_list),
                        len(max_model_step_list)])

    q_star = np.load('optimal_q.npy')

    # TODO: Set the following random seed to your *student ID*
    np.random.seed(_)

    # TODO: Call repeatExperiments() with different max_episode in
    #       the max_episode_list and max_model_step in the max_model_step_list
    #       *while fixing others as default*. Save the results to the results array.

    return results
```

It loops through different parameters and for each parameter configuration, repeats 5 runs of Dyna-Q. In each run, the returned $\mathbf{q}$ vector is compared with the true optimal action-values $\mathbf{q}_*$ (loaded from the `optimal_q.npy` file), based on root-mean-square error (RMSE).

Once completed, call the function with the following environment

```
env = FourRoom(goal=0)
```

whose optimal action-values $\mathbf{q}_*$ are given in the provided `optimal_q.npy` file. Then report the results in the following table

Table 4: RMSE for different parameters

Model Step / Episode	1	5	10	50
10				
30				
50				

Runtime. For efficient implementation, this function can be finished within one minute.

(c) (1%) Looking at your table from above, analyze the results and discuss any findings you may have and the possible reason behind them.