Assignment # 1 - Q-learning, SARSA, Experiment Loops, and ϵ-greedy Exploration

Author: Prabhat Nagarajan CMPUT 628, Deep RL: Winter 2025

Instructions and General Notes

- **Due Date**: January 24, 2025.
- **Late Policy**: See syllabus.
- **Scoring**: There will be 100 total marks.

Collaboration and Cheating

Do not cheat. You are graduate students. You can discuss the assignment with other students, but do not share code with one another. Do not use language models to produce your solutions.

General Advice

We highly recommend **starting the assignments early**. In machine learning and reinforcement learning, training agents or learning systems requires a different skillset from traditional programming and software engineering. Oftentimes the process of debugging is far more time-consuming, as code may be only "somewhat" wrong and so discovering why your agent is underperforming is not a straightforward task. Moreover, debugging often involves having agents run for periods of time, and training can be both expensive in terms of both time and compute. Building agents is also not merely validating program correctness, but involves a lot of trial and error on the programmer's part (not just on the part of the agents).

Software and Dependencies

This assignment will use:

- Python 3.11
- numpy
- matplotlib
- gymnasium

The files contain all the required imports. You should not need to import any additional libraries or packages. You are welcome to import additional libraries for debugging purposes, but your final submission should not include any new imports.

Submission

Please submit a zipped folder titled `a1_<ccid>.zip`, where you replace <ccid> with (can you guess?) your ccid. E.g., a1_machado.zip. The unzipped folder should be `a1_<ccid>`. *Failure to do so will cost you 5 marks.* This zip file should contain:

- pdf titled a1_<ccid>.pdf
- All the python files from `a1` containing your solutions in `a1_<ccid>.zip`
- Do not include the test code in your submission.

Assignment

In this assignment, you will learn many of the basics of reinforcement learning implementation. In particular, in this assignment you will implement:

- **The agent-environment interface**: You will implement the interaction between the agent and the environment. You will query the agent for actions and feed those to the environment and send the consequent rewards, next states, terminal signals, etc. back to the agent for performing parameter updates. You will also be responsible for managing environment resets.
- **ϵ-greedy exploration**. ϵ-greedy exploration is one of the most basic and perhaps the most often used exploration mechanisms in value-based reinforcement learning.
- **Temporal difference learning algorithms**: You will implement two basic temporal difference learning algorithms: SARSA and Q-learning.

Tests

- Testing your own code and functionality is an an integral part of building a research codebase. *Nonetheless, as the deeply generous people we are, we will provide you with a binary (without the source code) for all the tests that you will need to pass, along with a sample of the source code of some of the actual tests...*
- We have included *some* of the tests in `a1_partial_mark`. The remaining tests can be run from the binary. To run these tests:
  - Copy your files e.g., `cp a1_machado/*.py a1_partial_mark`
  - Run `a1_partial_mark/a1_tests.py` passing in your ccid
- To run all the tests, run the provided binary file, pointing it to the directory with all your code. E.g. `./a1_tests /Users/machado/assignments/a1/a1_machado/`

General Interface for the Course

Environments We will be using Farama Foundation's Gymnasium interface for the environment. In this interface, the agent executes an action (an integer typically for discrete action environments) and the environment returns the next observation, the reward, whether the episode terminates, whether the episode truncates, and other additional information. Truncation refers to when the agent does not transition to a terminal state, but rather the episode has run on long enough and we want to reset the environment. You are responsible for looking up this documentation and learning this API yourself.

Agents It implements two core methods: `act` and `process_transition`. The `act` method takes in an `obs` (observation) and then produces an action (likely) based on the observation. The `process_transition` method takes in the consequences of the previous action: `obs` (the next observation), `reward`, `terminated` (whether the episode terminated), `truncated` (whether the episode was cut off due to time limits) and performs the relevant processing of the transition (including updating the Q-functions or agent parameters). **These should be the only two methods that are called in your agent-environment loops.** In this assignment, we will implement both SARSA and Q-learning on the CliffWorld environment (see Example 6.6 of Sutton and Barto's book).

Agent Environment Interface: `agent_environment.py` [20 marks]

This portion of the assignment will be re-used in future assignments, so it is important to write this correctly.

In RL, we often train agents for a specific number of training episodes. Alternatively, we may train agents for a specific number of timesteps. Both options are used, and so we will implement both kinds of training loops in this assignment.

You will implement two functions:

- `def agent_environment_episode_loop` **[10 marks]**: This function takes as input an `agent`, an `env` (a Gymnasium environment), and *the number of episodes* for which you want to run. Your function should:
  - Complete the required numbers of episodes.
  - Accurately compute the episodic returns.
  - Pass the correct information to the agent (the `act` and `process_transition` methods).
  - Reset the environment at the appropriate times.

  These will all be tested in a single test.
- `def agent_environment_step_loop` **[10 marks]**: This function takes as input an `agent`, an `env` (a Gymnasium environment), and *the number of steps* for which you want to run. *If the final episode does not terminate or truncate once the number of training steps*

has been completed, the final unfinished (i.e., non-terminated and non-truncated) episode should not be included in your list of episode returns, as this is premature truncation. Your function should:

- Complete the required number of timesteps.
- Accurately compute the episodic returns.
- Pass the correct information to the agent (the `act` and `process_transition` methods).
- Reset the environment at the appropriate times.

These will all be tested in a single test.

Note that both truncation and termination are considered a complete episode for the purposes of this function. These functions then return a list of the cumulative undiscounted episode rewards. The only `agent` methods that your agent-environment loops can call are the `agent.act` and the `agent.process_transition` methods. No other methods from the agents should be called. You should call these methods each once for every `env.step` call.

ϵ-greedy exploration: `epsilon_greedy_explorers.py` [20 marks]

This portion of the assignment will be re-used in future assignments, so it is important to write this correctly.

You will implement one function:

- `def compute_epsilon_greedy_action_probs`: This function takes as input `q_vals` and an `epsilon`. `q_vals` is a `numpy` array of n action-values. The function should return another numpy array of n values summing to 1 where the ith value corresponds to the probability of taking action i under ϵ-greedy action selection. In ϵ-greedy action-selection, with probability ϵ the agent selects an action uniformly at random. With probability $1-\epsilon$ the agent selects amongst the *greedy* actions (i.e., the action or actions that has the highest action-value) uniformly at random. There will be 4 tests:
  - Can you accurately compute the action probabilities in the standard case of a single value-maximizing action? **[5 marks]**
  - Can you accurately compute the action probabilities when there are multiple value-maximizing actions? **[5 marks]**
  - Can you accurately compute the action probabilities when $\epsilon = 1.0$? **[5 marks]**
  - Can you accurately compute the action probabilities when $\epsilon = 0$ (pure exploitation)? **[5 marks]**

SARSA and Q-learning: `sarsa.py` and `q_learning.py` [20 marks]

Both Q-learning and SARSA follow the general update of:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\text{target} - Q(s,a)].$$

Given a transition (s,a,r,s'), the Q-learning target is:

$$\text{target} = r + \gamma \max_{a'} Q(s',a').$$

Given a transition and next-action (s,a,r,s',a'), the SARSA target is:

$$\text{target} = r + \gamma Q(s',a').$$

If s' is the terminal state, then the target for both SARSA and Q-learning is:

$$\text{target} = r,$$

as $Q(s',a')$ should be zero for the terminal state. Note that some of these cases may or may not be covered by the Cliffworld example from this assignment.

Your implementations should:

- Perform the correct updates under terminal and nonterminal transitions.

In both files you should implement the `act`, `update_q`, and the `process_transition` functions. `act` should, among other things, use the explorer to produce the agent's action in a state. The `update_q` method should actually update the `self.q` estimate. The `process_transition` method should process the consequences of the previous decision, including calling the `update_q` method with the appropriate parameters. You are free to make modifications/introduce variables within the scope specified for your code. The tests include:

- A standard SARSA update with bootstrapping (`update_q`) **[5 marks]**
- A standard Q-learning update with bootstrapping (`update_q`) **[5 marks]**
- A SARSA update with a transition to a terminal state (`update_q`) **[5 marks]**
- A Q-learning update with a transition to a terminal state (`update_q`) **[5 marks]**

Cliffworld results: `cliffworld_learner.py` [40 marks]

One the previous files have been completed correctly, you should be able to run `cliffworld_learner.py` successfully, which should produce 3 files:

- q_learning_cliff.png
- sarsa_cliff.png
- sarsa_q_learning_cliff.png

You should not need to modify this file, other than replacing the `CCID` variable with your ccid. Include these three files in a single pdf called `a1_<ccid>.pdf` and submit this pdf alongside your code (where you use your ccid).