Summer 2 2024, Khoury College, Northeastern University

Programming Assignment 2

Instructions

- If you discuss this assignment with one or more classmates, all parties must declare collaborators in their individual submissions within a code comment. Such discussions must be kept at a conceptual level, and no sharing of actual code is permitted.
- You may use any generative AI tool available to you, as long as it is appropriately cited in a code comment. I recommend using AI tools for debugging or conceptual understanding rather than to produce actual functions.

Deadlines

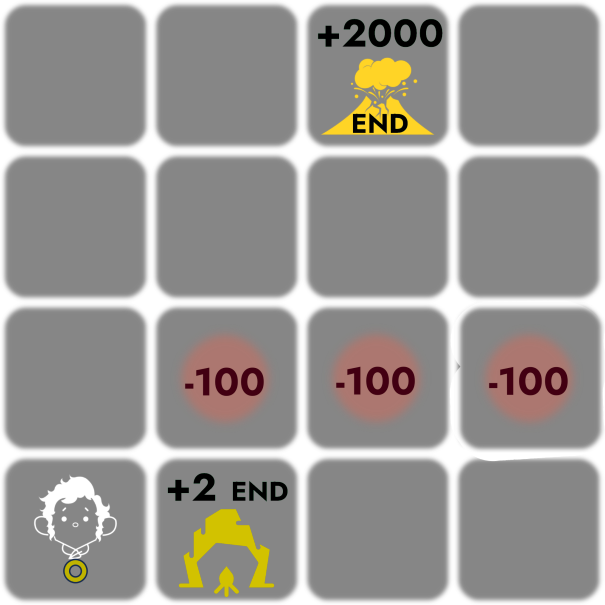
- Submissions should be uploaded to Gradescope by 6:00 PM on 8/4/2024.
- Gradescope will show a 'late' deadline of 8/7. This is intended solely for any students who may wish to invoke the freebie, outlined in the course policies.
- Any submissions received after 6:00 PM on 8/4 will be considered late, and will automatically invoke the use of your freebie. If you have used your freebie on a previous assignment, your submission will not be accepted for credit.
- Regrade requests must be submitted on Gradescope within 1 week of receiving your grade, after which no further requests will be entertained.

Reach Out!

If at any point you feel stuck with the assignment, please reach out to the TAs or the instructor, and do so early on! This lets us guide you in the right direction in a timely fashion and will help you make the most of your assignment.

A Clumsy Frodo in Mordor

This assignment introduces students to reinforcement learning in a discrete state-space setting. You will write code that helps our agent, Frodo the hobbit, figure out an optimal policy in a grid-based environment. The environment is divided into 16 cells, arranged in a 4×4 grid, as follows:



Frodo, our agent, starts in the bottom left cell at the beginning of any episode. His goal in this world is to reach the volcano, Mount Doom - the third cell in the top row - in order to destroy a ring, which awards the agent +2000 points. Mount Doom is a terminal state.

To the right of Frodo's starting position is a cave, which is a safe-haven. The agent, upon entering this cave, receives a small reward of +2 points, and the episode ends (i.e., the cave is a terminal state), since we assume that the agent never wishes to leave the safety of the cave.

En route to Mount Doom, Frodo must also avoid the gaze of the evil sorcerer, Sauron. If Frodo is seen stepping into any of the three red cells, he receives a penalty of -100, but the game does **not** end, i.e., these cells are not terminal states.

From each cell, the agent may attempt one of four actions - corresponding to the four directions the agent may move in. The agent is not allowed to leave the grid (e.g., from the start state, only 'UP' and 'RT' are valid actions, and so on). To make things more interesting, in this world, Frodo is clumsy - and once in a while ends up in a cell other than the one he was heading towards. This is modeled as Frodo having a 'slipping' probability, and is supplied by the autograder. A slipping probability of 0.1 means that the agent's action (if valid) succeeds in moving the agent in the desired direction with a 90% probability,

and moves the agent in one of the remaining valid directions with 10% probability. The remaining legal directions should be considered equally likely if an action fails.

The grid-world is already set up for you as a Gymnasium environment. Interacting with this environment follows the same syntax as lab 2.

Your Task

Your task is to complete two functions in the `mordor.py` file provided to you - namely the `q_learning()` function and the `plot_heatmaps()` function.

1. The `q_learning()` function takes four arguments - the number of episodes, n to simulate, and a list of checkpoint episode numbers, the discount factor γ, and the starting value of ϵ, and returns the following:

- A Q-value table (numpy array) of shape (num. states$\times$ num. actions)
- The optimal policy, as a numpy array of size (num. states), ordered by state index
- Checkpointed values for V_{opt} as a list of k numpy arrays, where k is the number of checkpoints supplied. Each array contains the optimal value function for every state at the respective checkpoint, and should be of shape (num. states).

In this function, simulate n episodes, using the `env.step()` function to interact with the environment. Calling `env.step()` at any cell gives you a reward and a new state. States are mapped to the integer range [0, 15], where state 0 represents the top left cell (i.e. (0, 0)), state 4 represents the cell indexed by (1, 0) (the cell below state 0), and so on, until state 15 represents (3, 3). Actions are mapped to the integers 0-3 in the order ['up', 'down', 'left', 'right'].

The agent should also use the ϵ-greedy policy to explore the environment (see below). For each observed (s, a, r, s'), the Q-value estimate, $Q_{opt}(s, a)$ for the state-action pair (s, a) should be updated as follows:

$$\eta = \frac{1}{1 + \text{number of updates to } \hat{Q}_{opt}(s,a)}$$

$$\text{Estimate, } \hat{Q}^{t}_{opt}(s,a) = (1-\eta)\hat{Q}^{(t-1)}_{opt}(s,a) + \eta[R(s,a,s') + \gamma \hat{V}^{(t-1)}_{opt}(s')]$$

$$\text{where } \hat{V}_{opt}(s) = \max_{a' \in A} \hat{Q}^{(t-1)}_{opt}(s', a')$$

At each time step, t, the action the agent plays from state s is chosen as follows:

$$\pi_{act}(s) = \begin{cases} \text{random}(a \in A); \text{with probability } P = \epsilon \\ \arg\max_{a \in A} \hat{Q}_{opt}(s,a); \text{with probability } P = (1-\epsilon) \end{cases}$$

In case an action chosen by the ϵ-greedy strategy is not a legal move, generate a different random legal action. The episode terminates when the agent reaches one of two terminal states. The Q-table is initialized to all zeros. The value of η is unique for every (s, a) pair, and should be updated as 1/(1 + number of updates to $\hat{Q}_{opt}(s, a)$). The number of updates to $\hat{Q}_{opt}(s, a)$ should be stored in a matrix of shape (num. states, num. actions), initialized to zeros, and updated such that executing $num\_updates[s, a]$ gives you the number of times $\hat{Q}_{opt}(s, a)$ has been updated. You can then calculate η using the formula given above. The value of epsilon should be decayed to (0.9999 * epsilon) at the end of each episode.

Finally, the optimal policy should be a numpy array, where the i^{th} entry in the array is an integer corresponding to the optimal action for state i. For end states, the optimal action should be encoded as -1. Actions for end states may be hard coded at the end of the function before the return statement.

2. The `plot_heatmaps()` function takes two arguments - an array of shape (num. states) containing per-state $V_{opt}(s)$ values for any one given checkpoint, and a corresponding filename. This function should plot a 4x4 heatmap of the optimal value function, with states appropriately mapped to (and in the same location as the corresponding) cells in the map of Mordor, with the given filename in the current directory. Make sure your heatmap is accompanied by a legend (colorbar) showing the range of the plotted V_{opt} values. **Do not use plt.show(), or your code will time out on the autograder.**

Submission

Submit your completed `mordor.py` file and the three plots generated by the `plot_heatmaps()` function, using the default parameters, to Gradescope. Your `q_learning()` function will be executed with various parameters by the autograder to check for correctness.

Grading is based on a combination of factors in your implementation - including the shapes and data-types of the Q-table and the optimal policy, whether the expected optimal value for various states is within expectation, correct generation of heatmaps, proper checks for whether actions are valid before executing `env.step()`, and whether the optimal policy leads the agent to the goal (within reasonable expectation accounting for the slipping probability).