**Homework 2: Reinforcement Learning**

**Due: Monday, October 28 at 1:35 PM**

In this assignment, you will be implementing the Q-learning algorithm, using the “clumsy Frodo” example we discussed in class. The goal of this assignment is to increase your familiarity with reinforcement learning in general and Q-learning in particular, as well as to give you more practice working with the Gymnasium package.

**The Problem**

A screenshot of a game

Description automatically generated

Our agent, Frodo, is traveling through Mordor. His goal is to reach Mt. Doom (to throw a ring in a volcano) and to avoid Sauron, who can see the squares marked in red on the map. Additionally, Frodo has the option to enter a cave, which will grant him a small reward and end his travels.

We will represent Frodo’s travels using the grid above. This is a 4x4 grid, consisting of 16 states. 2 of these states, the cave (which gives a reward of +2) and Mt. Doom (which gives a reward of +2000), are terminal; as soon as Frodo enters one (and receives his reward), the game ends. The 3 states in Sauron’s range each provide a reward of -100 and do **not** end the game. All other states do not provide any reward and do not end the game.

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. Occasionally, up in a cell other than the one he was heading towards. This is modeled as Frodo having a “slipping” probability, which by default is set to 0.1. This means that 90% of the time, Frodo will move in the intended direction. The remaining 10% of the time, Frodo will move in a legal unintended direction. If there are multiple legal unintended directions, one will be chosen at random. (Please note that these probabilities are different from the examples we did in class with a similar problem.)

This problem is already set up for you as a Gymnasium environment in the file mordor.py. Interacting with this environment follows the same syntax as lab 2.

**Learning**

In this assignment, you will be implementing Q-learning to help maximize Frodo’s reward. You will be doing this in the provided file mordor.py. While the relevant Gymnasium environment has been provided for you, you will need to update two functions

**q\_learning**

The primary function you will be implementing is q\_learning. This function takes in 4 arguments:

* num\_episodes: the number of episodes to simulate
* checkpoints: a list of numbers representing “checkpoint” episodes whose Q-values will be saved
* gamma: the discounting factor
* epsilon: the starting value of epsilon for epsilon-greedy exploration

This function also returns three values:

* Q: the final values of the Q-table, which should be represented as a numpy array with dimensions num. states x num. actions
* optimal\_policy: the final optimal policy, represented as a one-dimensional numpy array (containing actions) of size num. states, ordered by the states’ indices described below
* V\_opt\_checkpoint\_values: a list of *k* numpy arrays, where *k* is the number of checkpoints. Each array contains, the value of V\_opt(s) for each state s at the specified checkpoint. We define V\_opt(s) as the maximum value of Q(s, a’) across all actions a’.

In this function, simulate the specified number of 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], working left to right, and then top to bottom. This means 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’].

During training, we will use the epsilon-greedy approach to choose actions. This means with probability epsilon, we will choose a random legal action. With probability 1-epsilon, we will choose the optimal action: the action a that maximizes Q(s,a). If multiple actions have the same Q-value, we will select the one with the lower index. The starting value of epsilon is 0.9, and at the end of each episode of training, we will lower it by multiplying it by 0.999.

While an episode of training does not end until Frodo reaches one of the two terminal states, since Q-learning is a temporal difference algorithm, we will update our Q values immediately after each action using the equation discussed in class:

Q(s, a) ← Q(s,a) + α(R(s, a, s’) + γ \* max\_a’ Q(s’,a’) - Q(s, a))

To account for the fact that different states will get explored different numbers of times during training, we will use a different value of α for each state-action pair and we will lower these α values over time. In particular, define:

α = 1/(1 + #of previous updates to Q(s,a))

This means that the first time Q(s, a) is updated, α will equal 1, and α will then decrease as more updates occur. In order to keep track of this, you will want 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 Q(s, a) has been updated. Using this matrix, you can calculate α directly before each update.

When running q\_learning, you will specify several “checkpoint” episodes. At the end of a checkpoint episode, you will need to store the optimal Q values for each state. In particular, this will come in the form of a numpy array whose ith entry corresponds to the maximum value of Q(s, a) for state i (using the numbering system described below.

Finally, the optimal policy should be a numpy array, where the ith entry in the array is an integer corresponding to the optimal action for state i. For terminal states, the optimal action should be encoded as -1. Actions for terminal states may be hard coded at the end of the function before the return statement. For all nonterminal states, the optimal action should be a direction, as represented by a number from 0 to 3.

**plot\_heatmaps**

The second function you need to write creates heatmaps to visualize the optimal Q values for each state at each of the checkpoints, using matplotlib. This function takes in two arguments:

* V\_opt: array of shape (num. states) containing per-state V\_opt(s) values for any one given checkpoint. (Recall that we define V\_opt(s) as the maximum value of Q(s, a) over all a.)
* filename: a filename to save your heatmap to

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. Make sure your heatmap is accompanied by a legend (colorbar) showing the range of the plotted V\_opt values. Your function should save your heatmap to the specified file and should **not** use plt.show() (as this can interfere with autograding capabilities).

**Wrapping Up**

Turn in your mordor.py file, along with all three heatmaps provided by the checkpoint values in mordor.py. You do **not** need to turn in a written report or any other files.