CSCI 3202, Spring 2023

Homework 6

Due: Monday, May 1 at 9:00 pm

In this assignment, I'm providing the solution and asking you to reflect on the Q-learning portion of the solution. The code is setup to run 3 trials, 10 steps each, and is hard-coded to start at the same location on each trial. It won't find the landing pad in only 3 trials, you will need to change that number if you want to see the code produce a valid solution. I added a few print statements in the Q-learning algorithm to help illustrate the code. Feel free to add as many additional print statements that you need to help you understand how its working.

The questions are listed in Part 2 of this notebook. This homework is worth 50 points

Your name: Julia Troni

In [ ]:

```
import pandas as pd
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
from collections import defaultdict

# added packages
import heapq
from matplotlib import colors
```

Part 1: Reinforcement learning

Consider a **cube** state space defined by $0 \leq x, y, z \leq L$. Suppose you are piloting/programming a drone to learn how to land on a platform at the center of the $z = 0$ surface (the bottom). Some assumptions:

- In this discrete world, if I say the drone is at (x, y, z) I mean that it is in the box centered at (x, y, z). And there are boxes (states) centered at (x, y, z) for all $0 \leq x, y, z \leq L$. Each state is a 1 unit cube. So when $L = 2$ (for example), there are cubes centered at each $x = 0, 1, 2$, $y = 0, 1, 2$ and so on, for a total state space size of $3^3 = 27$ states.
- All of the states with $z = 0$ are terminal states.

- The state at the center of the bottom of the cubic state space is the landing pad. So, for example, when $L = 4$, the landing pad is at $(x, y, z) = (2, 2, 0)$.
- All terminal states ***except*** the landing pad have a reward of -1. The landing pad has a reward of +1.
- All non-terminal states have a reward of -0.01.
- The drone takes up exactly 1 cubic unit, and begins in a random non-terminal state.
- The available actions in non-terminal states include moving exactly 1 unit Up (+z), Down (-z), North (+y), South (-y), East (+x) or West (-x). In a terminal state, the training episode should end.

Part A

Write a class `MDPLanding` to represent the Markov decision process for this drone. Include methods for:

1. `actions(state)` , which should return a list of all actions available from the given state
2. `reward(state)` , which should return the reward for the given state
3. `result(state, action)` , which should return the resulting state of doing the given action in the given state

and attributes for:

1. `states` , which is just a list of all the states in the state space, where each state is represented as an (x, y, z) tuple
2. `terminal_states` , a dictionary where keys are the terminal state tuples and the values are the rewards associated with those terminal states
3. `default_reward` , which is a scalar for the reward associated with non-terminal states
4. `all_actions` , a list of all possible actions (Up, Down, North, South, East, West)
5. `discount` , the discount factor (use $\gamma = 0.999$ for this entire problem)

How you feed arguments/information into the class constructor is up to you.

Note that actions are *deterministic* here. The drone does not need to learn transition probabilities for outcomes of particular actions. What the drone does need to learn, however, is where the heck that landing pad is, and how to get there from any initial state.

In [ ]:
```
# Solution:

class MDPLanding:
    def __init__(self, size, default_reward, actions, discount):
        self.size = size
        self.lz = (int(size/2), int(size/2), 0)
        self.states = [(x,y,z) for x in range(size) for y in range(size) for z in range
        self.terminal_states = {(x,y,0) : -1 for x in range(size) for y in range(size)}
        self.terminal_states[self.lz] = +1
        self.default_reward = default_reward
        self.all_actions = actions
        self.discount = discount

    def actions(self, state):
        '''all are available, unless you are in a terminal state
        (the drone might bump into wall in some cases though)'''
        if state in self.terminal_states:
            return [None]
        else:
            return self.all_actions

    def reward(self, state):
        return self.terminal_states[state] if state in self.terminal_states.keys() else

    def result(self, state, action):
        '''result of doing `action` in `state`, deterministic.
        `action` is one of N, S, E, W, U, D (as string)'''

        assert action in self.actions(state), 'Error: action needs to be available in t
        assert state in self.states, 'Error: invalid state'

        if action=='N':
            new_state = (state[0], state[1]+1, state[2])
        elif action=='S':
            new_state = (state[0], state[1]-1, state[2])
        elif action=='E':
            new_state = (state[0]+1, state[1], state[2])
        elif action=='W':
            new_state = (state[0]-1, state[1], state[2])
        elif action=='U':
            new_state = (state[0], state[1], state[2]+1)
        elif action=='D':
            new_state = (state[0], state[1], state[2]-1)
        elif action is None:
            return state
        return new_state if new_state in self.states else state
```

Part B

Write a function to implement **policy iteration** for this drone landing MDP. Create an MDP environment to represent the $L = 4$ case (so 125 total states).

Use your function to find an optimal policy for your new MDP environment. Check (by printing to screen) that the policy for the following states are what you expect, and comment on the results:

1. $(2, 2, 1)$
2. $(0, 2, 1)$
3. $(2, 0, 1)$

In [ ]:

```
def policy_iteration(mdp):
    # initilize utility for all states
    utility = {s : 0 for s in mdp.states}
    # initialize a policy for each state, being a random action
    policy = {s: np.random.choice(mdp.actions(s)) for s in mdp.states}
    while True:
        # policy evaluation step
        utility = policy_evaluation(policy, utility, mdp)
        # policy improvement step
        unchanged = True
        for s in mdp.states:
            best = (-999, None)
            for a in mdp.actions(s):
                new_util = utility[mdp.result(s,a)]
                if new_util > best[0]:
                    best = (new_util, a)
            if best[1] != policy[s]:
                policy[s] = best[1]
                unchanged = False
        if unchanged:
            return policy

def policy_evaluation(policy, utility, mdp, n_iter=50):
    '''Do a handful (n_iter) of value iterations to update the
    utility of each state, under the given policy'''
    for i in range(n_iter):
        for s in mdp.states:
            if s in mdp.terminal_states:
                utility[s] = mdp.reward(s)
            else:
                utility[s] = mdp.reward(s) + mdp.discount * utility[mdp.result(s, polic
    return utility

size = 5
default_reward = -0.01
discount = 0.999
actions = ['N','S','E','W','U','D']
mdp = MDPLanding(size, default_reward, actions, discount)

policy = policy_iteration(mdp)
print('Policy(2,2,1) = '+policy[(2,2,1)])
print('Policy(0,2,1) = '+policy[(0,2,1)])
print('Policy(2,0,1) = '+policy[(2,0,1)])
```

Solution:

Makes sense, because (2,2,1) is right above the landing pad in the 5x5x5 environment, so the drone should just go straight down. Similarly, (0,2,1) is West of the landing pad (lower x-value), so the drone needs to head toward +x, or East, and (2,0,1) is South of the landing pad (lower y-value), so the drone needs to head toward +y, or North.

Part C

Code up a **Q-learning** agent/algorithm to learn how to land the drone. You can do this however you like, as long as you use the MDP class structure defined above.

Your code should include some kind of a wrapper to run many trials to train the agent and learn the Q values (see Section 22.3 in the textbook - page 803 might be of particular interest). You also do not need to have a separate function for the actual "agent"; your code can just be a "for" loop within which you are refining your estimate of the Q values.

From each training trial, save the cumulative discounted reward (utility) over the course of that episode. That is, add up all of $\gamma^t R(s_t)$ where the drone is in state s_t during time step t, for the entire sequence. I refer to this as "cumulative reward" because we usually refer to "utility" as the utility *under an optimal policy*.

Some guidelines:

- The drone should initialize in a random non-terminal state for each new training episode.
- The training episodes should be limited to 50 time steps, even if the drone has not yet landed. If the drone lands (in a terminal state), the training episode is over.
- You may use whatever learning rate α you decide is appropriate, and gives good results.
- There are many forms of Q-learning. You can use whatever you would like, subject to the reliability targets in Part D below.
- Your code should return:
  - The learned Q values associated with each state-action pair.
  - The cumulative reward for each training trial.
  - Anything else that might be useful in the ensuing analysis.

In [ ]:

```
# Solution:

def run_many_trials(mdp, n_trials, n_steps=None):

    Q = defaultdict(float)   # state-action matrix
    Nsa = defaultdict(float) # state-action visit counter (exploration)
    rewards = []
    epsilon = 0.1 # for epsilon-greedy action
    alpha = lambda n: 1./(1+n) # learning rate related to number of visits to this stat
    policy = {state : np.random.choice(mdp.all_actions) for state in mdp.states} # rand

    for k in range(n_trials):

        #s = mdp.states[np.random.randint(len(mdp.states))] # random initial location
        s = mdp.states[148] #seems like a nice starting place in the middle of the stat
        print('mdp.states[148]:')
        print(s)
        print('trial: ', k)
        while s in mdp.terminal_states: #if its a terminal state
            s = mdp.states[np.random.randint(len(mdp.states))] # re-sample a new state
        r = mdp.reward(s) #get initial reward
        cumulative_reward = 0 #track cumulative reward for the trial

        for t in range(n_steps):

            print("Step: ", t)

            if s in mdp.terminal_states: #if terminal state
                Q[s, None] = r #update value of terminal state
                policy[s] = None # update the policy to none
                print('Terminal found:')
                print(s)
                cumulative_reward += (mdp.discount**t)*r
                break
            else:
                # pick a new action and state using greedy policy
                print('policy: ' + policy[s])
                a = return_epsilon_greedy_action(policy, s, mdp, epsilon)
                print("action: " + a)
                print(a)
                s1 = mdp.result(s, a) #new state
                print("s1:")
                print(s1)
                r1 = mdp.reward(s1) #new reward
                print("r1:")
                print(r1)
                Nsa[s, a] += 1 # update the state-action "matrix"
                print("Q[s, a] before for s and a:", Q[s,a], s, a)
                Q[s, a] += alpha(Nsa[s, a]) * (r + mdp.discount * max(Q[s1, a1] for a1
                print("Q[s, a] for s and a:", Q[s,a], s, a)
                for a1 in mdp.actions(s1):
                    print("Q for a1, s1", Q[s1, a1], s1, a1)
                #                Q[s, a] += 0.001 * (r + mdp.discount * max(Q[s1, a1] f
                # update the policy
                best = (-999, None)
                for a1 in mdp.actions(s):
                    new = Q[s, a1]
                    if new > best[0]:
                        best = (new, a1) #choose action with highest value so far
                policy[s] = best[1] #update policy
                cumulative_reward += (mdp.discount**t)*r #accumulate reward
                s = s1 #transition to new state
                r = r1 #update reward

        rewards.append(cumulative_reward)

    return Q, Nsa, rewards

def return_epsilon_greedy_action(policy, state, mdp, epsilon=0.1):
    '''Return a random action or the one with highest utility (so far)'''
    if np.random.uniform(0, 1) <= epsilon:
        action = np.random.choice(mdp.actions(state))
    else:
        action = policy[state]
    return action
```

Part D

Initialize the $L = 10$ environment (so that the landing pad is at $(5, 5, 0)$). Run some number of training trials to train the drone.

How do I know if my drone is learned enough? If you take the mean cumulative reward across the last 5000 training trials, it should be around 0.80. This means at least about 10,000 (but probably more) training episodes will be necessary. It will take a few seconds on your computer, so start small to test your codes.

Then: Compute block means of cumulative reward from all of your training trials. Use blocks of 500 training trials. This means you need to create some kind of array-like structure such that its first element is the mean of the first 500 trials' cumulative rewards; its second element is the mean of the 501-1000th trials' cumulative rewards; and so on. Make a plot of the block mean rewards as the training progresses. It should increase from about -0.5 initially to somewhere around +0.8.

And: Print to the screen the mean of the last 5000 trials' cumulative rewards, to verify that it is indeed about 0.80.

In [ ]:

```
# Solution Part D:
```

```
size = 11
default_reward = -0.01
discount = 0.999
actions = ['N','S','E','W','U','D']
mdp = MDPLanding(size, default_reward, actions, discount)
n_trials = 3 #homework 3, part 2
Q,Nsa,rewards = run_many_trials(mdp, n_trials, n_steps=10)
#Q,Nsa,rewards = run_many_trials(mdp, n_trials=1, n_steps=50)


#di = 500
#rewards_block = list(np.mean(np.asarray(rewards).reshape(-1, di),1))
#plt.plot(list(range(0,len(rewards),di)), rewards_block)
#plt.xlabel('Training episode')
#plt.ylabel('Cumulative reward')
#plt.show()

#print('Last 5000 trials mean cumulative reward = {:0.4f}'.format(np.mean(rewards[-5000
```

Part 2 - Homework 3 questions.

These questions refer to the Q-learning implementation in Parts C and D of this notebook.

1: (5 pts) This code in Part D is set up to run 3 trials, 10 steps each, starting from (1,2,5). The values for state, reward, policy, and Q[s,a] print on each step.

- What is the first action taken and what state does the agent move to?
- The values for policy and action are printed. What's the difference in the code?
- What are the lines of code that implement the action and determine the new state?

2: (5 pts) How is the s1 variable used in the code, including in the Q[s,a] calculation?

3: (5 pts) What are the initial values for the Q-matrix, the discount factor, learning rate, and the policy matrix?

4: (10 pts) What is the purpose of the *max(Q[s1, a1] for a1 in mdp.actions(s1))* code in the Q[s,a] calculation? How does it influence the Q[s,a] value?

5: (10 pts) Where does the code store the policy for each state? How is it updated and where is it used in the code?

6: (15 pts) Write a paragram explaining how the algorithm learns in this example. What are the limitations of this method? <<note: I am guessing you mean write a paragraph not paragram??>>

1. For trial 1 the first policy is East and action taken is UP. The agent moves from (1,2,5) to (1,2,6).

For trial 2 the first policy is Down and action taken is Down. The agent moves from (1,2,5) to (1,2,4).

For trial 3 the first policy is North action taken is North. The agent moves from (1,2,5) to (1,3,5).

The policy variable is a dictionary that maps each state in the MDP to an action. It represents the current policy of the agent. The action variable is the actual action that the agent takes in a given state.

The lines of code that implement the action and determine the new state are: `a = return_epsilon_greedy_action(policy, s, mdp, epsilon)`
`s1 = mdp.result(s, a) </code>`

```
The first line uses an epsilon-greedy strategy to choose an action based on the
current policy, the current state, and exploration parameter epsilon. The second
line applies the chosen action a to the current state s to obtain the new state
s1. The result() method of the MDP class returns the new state that results from
applying a given action to a given state.
```

1. The s1 variable is used to hold the new state that the agent moves to. This is calculated here `s1 = mdp.result(s, a)  </code>.  It is then used to calculate the new reward r1 = mdp.reward(s1) </code>`

   ```
   In the Q[s,a] calculation, s1 is used to calculate the maximum Q-value over
   all actions a1 that can be taken from the next state s1. This is done using
   the code max(Q[s1, a1] for a1 in mdp.actions(s1)), which returns the maximum
   Q-value among all actions that can be taken in the next state s1
   ```

1. The initial values for the Q-matrix are set to zero with Q=defaultdict(float), the discount factor is 0.9, learning rate is initially 1 since n=0 and the learning rate is calculated as 1/1+n where n is the number of steps, and the policy matrix is initialized to a random initial policy for each state

1. The purpose of `max(Q[s1, a1] for a1 in mdp.actions(s1))</code>, is to estimate the value of the best action a1 that can be taken from the new state s1. This code selects the maximum value of the Q-value estimates for all actions that can be taken from state s1, which is an approximation of the value of the optimal policy for state s1.`

   ```
   it is used to obtain an estimate of the expected return of taking action a
   in state s and following the optimal policy thereafter. This value is then
   subtracted from the current estimate of Q[s,a] and the result is used to
   update the value of Q[s,a] for the current trial.
   ```

1. The policy is updated in this for loop:

```
<code/>  # update the policy
    for a1 in mdp.actions(s):
        new = Q[s, a1]
        if new > best[0]:
            best = (new, a1) #choose action with highest value so far
        policy[s] = best[1] #The policy is updated here
 </code>
```

   Here, the code loops over all possible actions for the current state s, and selects the action with the highest value according to the current estimate of the state-action values stored in the Q matrix. This action is then set as the new policy for the state s.

   This policy is used to pick a new action and state using greedy policy `a = return_epsilon_greedy_action(policy, s, mdp, epsilon)</code> which returns either a random action (with probability epsilon), or the action with the highest estimated value according to the current policy.`

1. In this example the algorithm used is Q-learning, which is a type of model-free reinforcement learning algorithm that tries to learn the optimal policy for an environment by iteratively updating the Q-value of state-action pairs. The Q-value is the expected total reward of taking a particular action in a particular state and following the optimal policy thereafter.

   Initially, the Q-matrix is initialized to zero for all state-action pairs, and the policy matrix is initialized randomly. The algorithm then interacts with the environment by selecting actions based on an epsilon-greedy policy, which chooses the action with the highest Q-value with probability 1-epsilon and a random action with probability epsilon.

   After each action, the algorithm observes the resulting state and the reward obtained and updates the Q-value for the previous state-action pair. The update rule involves a learning rate alpha that determines how much the new information should be weighted against the old information, and a discount factor gamma that weights future rewards less than immediate rewards. Continue in this way to update the Q-values and the policy matrix based on the observed rewards until it converges to an optimal policy, which maximizes the expected total reward for any given state. The rewards obtained during the learning process are accumulated over multiple trials and used to evaluate the performance of the algorithm.

One limitation of Q-learning and other model-free reinforcement learning algorithms is that it is extremely comutationally expensive and takes lots of time and this can be inefficient for large and complex environments and may require a large number of iterations to converge to the optimal policy.

```
In [ ]:
```