hw4-Q2

April 5, 2021

1 2. Reinforcement Learning

There are 3 files: 1. `maze.py`: defines the `MazeEnv` class, the simulation environment which the Q-learning agent will interact in. 2. `qlearning.py`: defines the `qlearn` function which you will implement, along with several helper functions. Follow the instructions in the file. 3. `plotting_utils.py`: defines several plotting and visualization utilities. In particular, you will use `plot_steps_vs_iters`, `plot_several_steps_vs_iters`, `plot_policy_from_q`

[ ]:
```
#from qlearning import qlearn
#from maze import MazeEnv, ProbabilisticMazeEnv
#from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters,␣
↪plot_policy_from_q
```

1.1 qlearning.py

[ ]:
```
import numpy as np
import math
import copy

def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps,␣
↪use_softmax_policy, init_beta=None, k_exp_sched=None):
    """ Runs tabular Q learning algorithm for stochastic environment.

    Args:
        env: instance of environment object
        num_iters (int): Number of episodes to run Q-learning algorithm
        alpha (float): The learning rate between [0,1]
        gamma (float): Discount factor, between [0,1)
        epsilon (float): Probability in [0,1] that the agent selects a random␣
↪move instead of
                selecting greedily from Q value
        max_steps (int): Maximum number of steps in the environment per episode
        use_softmax_policy (bool): Whether to use softmax policy (True) or␣
↪Epsilon-Greedy (False)
        init_beta (float): If using stochastic policy, sets the initial beta as␣
↪the parameter for the softmax
```

```
        k_exp_sched (float): If using stochastic policy, sets hyperparameter␣
↪for exponential schedule
            on beta

    Returns:
        q_hat: A Q-value table shaped [num_states, num_actions] for environment␣
↪with with num_states
            number of states (e.g. num rows * num columns for grid) and␣
↪num_actions number of possible
            actions (e.g. 4 actions up/down/left/right)
        steps_vs_iters: An array of size num_iters. Each element denotes the␣
↪number
            of steps in the environment that the agent took to get to the goal
            (capped to max_steps)
    """
    action_space_size = env.num_actions
    state_space_size = env.num_states
    q_hat = np.zeros(shape=(state_space_size, action_space_size))
    steps_vs_iters = np.zeros(num_iters)

    for i in range(num_iters):
        # TODO: Initialize current state by resetting the environment
        curr_state = env.reset()
        num_steps = 0
        done = False

        # TODO: Keep looping while environment isn't done and less than maximum␣
↪steps
        while (num_steps <= max_steps) and (not done):
            num_steps += 1

            # Choose an action using policy derived from either softmax Q-value
            # or epsilon greedy
            if use_softmax_policy:
                assert(init_beta is not None)
                assert(k_exp_sched is not None)
                # TODO: Boltzmann stochastic policy (softmax policy)
                beta = beta_exp_schedule(init_beta, i, k = k_exp_sched)
                action = softmax_policy(q_hat, beta, curr_state)
            else:
                # TODO: Epsilon-greedy
                action = epsilon_greedy(q_hat, epsilon, curr_state,␣
↪action_space_size)

            # TODO: Execute action in the environment and observe the next␣
↪state, reward, and done flag
```

```
            next_state, reward, done = env.step(action)

            # TODO: Update Q_value
            if next_state != curr_state:
                new_value = reward + gamma * np.max(q_hat[next_state, :]) -␣
↪q_hat[curr_state, action]
                # TODO: Use Q-learning rule to update q_hat for the curr_state␣
↪and action:
                # i.e., Q(s,a) <- Q(s,a) + alpha*[reward + gamma *␣
↪max_a'(Q(s',a')) - Q(s,a)]
                q_hat[curr_state, action] = q_hat[curr_state, action] + alpha *␣
↪new_value
                # TODO: Update the current staet to be the next state
                curr_state = next_state

        steps_vs_iters[i] = num_steps

    return q_hat, steps_vs_iters


def epsilon_greedy(q_hat, epsilon, state, action_space_size):
    """ Chooses a random action with p_rand_move probability,
    otherwise choose the action with highest Q value for
    current observation

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns and␣
↪num_actions
            number of possible actions
        epsilon (float): Probability in [0,1] that the agent selects a random
            move instead of selecting greedily from Q value
        state: A 2-element array with integer element denoting the row and␣
↪column
            that the agent is in
        action_space_size (int): number of possible actions

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: Sample from a uniform distribution and check if the sample is below
    # a certain threshold
    # ...
    q_state = q_hat[state]
```

```python
    if np.all(q_state == 0) or np.random.uniform(0, 1) < epsilon:
        act = np.random.randint(action_space_size)
        return act
    else:
        optim = np.argmax(q_state)
        return optim


def softmax_policy(q_hat, beta, state):
    """ Choose action using policy derived from Q, using
    softmax of the Q values divided by the temperature.

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns
        beta (float): Parameter for controlling the stochasticity of the action
        obs: A 2-element array with integer element denoting the row and column
            that the agent is in

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: use the stable_softmax function defined below
    # ...
    q_state = q_hat[state]
    softmax_q_state = stable_softmax(beta * q_state, axis = 0)
    return np.random.choice(4, p = softmax_q_state)

def beta_exp_schedule(init_beta, iteration, k = 0.1):
    beta = init_beta * np.exp(k * iteration)
    return beta

def stable_softmax(x, axis=2):
    """ Numerically stable softmax:
    softmax(x) = e^x /(sum(e^x))
               = e^x / (e^max(x) * sum(e^x/e^max(x)))

    Args:
        x: An N-dimensional array of floats
        axis: The axis for normalizing over.

    Returns:
        output: softmax(x) along the specified dimension
    """
```

```
    max_x = np.max(x, axis, keepdims = True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims = True)

    return output
```

1.2 plotting_utlis.py

[ ]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
# from qlearning import *
# from maze import *

#  UTILITY FUNCTIONS



color_cycle = ['#377eb8', '#ff7f00', '#a65628',
               '#f781bf','#4daf4a',  '#984ea3',
               '#999999', '#e41a1c', '#dede00']

def plot_steps_vs_iters(steps_vs_iters, block_size=10):
    num_iters = len(steps_vs_iters)
    block_size = 10
    num_blocks = num_iters // block_size
    smooted_data = np.zeros(shape=(num_blocks, 1))
    for i in range(num_blocks):
        lower = i * block_size
        upper = lower + 9
        smooted_data[i] = np.mean(steps_vs_iters[lower:upper])

    plt.figure()
    plt.title("Steps to goal vs episodes")
    plt.ylabel("Steps to goal")
    plt.xlabel("Episodes")
    plt.plot(np.arange(1,num_iters,block_size), smooted_data,␣
 ↪color=color_cycle[0])

    return

def plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10):
    smooted_data_list = []
    for steps_vs_iters in steps_vs_iters_list:
        num_iters = len(steps_vs_iters)
        block_size = 10
        num_blocks = num_iters // block_size
        smooted_data = np.zeros(shape=(num_blocks, 1))
```

```python
        for i in range(num_blocks):
            lower = i * block_size
            upper = lower + 9
            smooted_data[i] = np.mean(steps_vs_iters[lower:upper])
        smooted_data_list.append(smooted_data)

    plt.figure()
    plt.title("Steps to goal vs episodes")
    plt.ylabel("Steps to goal")
    plt.xlabel("Episodes")
    index = 0
    for label, smooted_data in zip(label_list, smooted_data_list):
        plt.plot(np.arange(1,num_iters,block_size), smooted_data, label=label,␣
↪color=color_cycle[index])
        index += 1
    plt.legend()

    return


# this function sets color values for
# Q table cells depending on expected reward value
def get_color(value, min_val, max_val):

    switcher={
                0:'gray',
                1:'indigo',
                2:'darkmagenta',
                3:'orchid',
                4:'lightpink',
             }

    step = (max_val-min_val)/5
    i = 0
    color='lightpink'

    for limit in np.arange(min_val, max_val, step):
        if limit <= value < limit+step:
            color = switcher.get(i)
        i+=1
    return color



# get first cell out of the start state
def get_next_cell(x1,x2,heatmap,policy_table,xlim=9,ylim=9):
    up_reward=-10000
```

```
    down_reward=-10000
    left_reward=-10000
    right_reward=-10000

    if (x1<ylim):
        if (policy_table[x1-1][x2]!=3):
            up_reward = heatmap[x1-1][x2]
    else:
        up_reward = -1000

    if (x1>0):
        if (policy_table[x1+1][x2]!=0):
            down_reward = heatmap[x1+1][x2]
    else:
        down_reward = -1000

    if (x2>0):
        if (policy_table[x1][x2-1]!=1):
            left_reward = heatmap[x1][x2-1]

    else:
        left_reward = -1000

    if (x2<xlim):
        if (policy_table[x1][x2+1]!=2):
            right_reward = heatmap[x1][x2+1]

    else:
        right_reward = -1000

    rewards = np.array([up_reward, down_reward, left_reward, right_reward])
    idx = np.argmax(rewards)
    next_cell = [(x1-1,x2), (x1+1,x2), (x1,x2-1), (x1,x2+1)][idx]
    choice = ['up', 'down', 'left', 'right']
    #print ('picking ',choice[idx])
    return next_cell




# get coordinates of the cells
# on the way from the start to goal state
def get_path(x1,x2, policy_table):
    x_coords = [x1]
    y_coords = [x2]
    x1_new = x1
    x2_new = x2
```

```python
    i=0
    num_steps = 0
    total_cells = len(policy_table)*len(policy_table[0])
    while (policy_table[x1][x2]!='G') and num_steps < total_cells:
        if (policy_table[x1][x2]==1): # right
            x2_new=x2+1
            #print(i, ' - moving right')

        elif (policy_table[x1][x2]==0):
            x1_new=x1-1
            #print(i, ' - moving up')

        elif (policy_table[x1][x2]==3):
            x1_new=x1+1
            #print(i, ' - moving down')

        elif (policy_table[x1][x2]==2):
            x2_new=x2-1
            #print(i, ' - moving left')

        x1 = x1_new
        x2 = x2_new
        x_coords.append(x1)
        y_coords.append(x2)
        num_steps += 1
    return x_coords, y_coords



# plot Q table
# optimal path is highlighted and cells colored by their values
def plot_table(env, table_data, heatmap, goal_states, start_state, max_val,␣
 ↪min_val, x_coords, y_coords):
    fig = plt.figure(dpi=80)
    ax = fig.add_subplot(1,1,1)
    plt.figure(figsize=(10,10))

    width = len(table_data[0])
    height = len(table_data)

    new_table = []

    for i in range(height):
        new_row = []

        for j in range(width):
```

```
            if env.map[i][j] == 0:
                new_row.append('')
            else:
                digit = table_data[i][j]
                if (digit==0):
                    new_row.append('\u2191') # up
                elif (digit==1):
                    new_row.append('\u2192') # right
                elif (digit==2):
                    new_row.append('\u2190') # left
                elif (digit==3):
                    new_row.append('\u2193') # down
                elif (digit=='G'):
                    new_row.append('G') # goal state
                elif (digit=='S'):
                    new_row.append('S') # goal state
                elif (digit==-1):
                    new_row.append('+') # All four directions
                else:
                    new_row.append('x') # unknown

        new_table.append(new_row)

    table = ax.table(cellText=new_table, loc='center',cellLoc='center')

    table.scale(1,2)

    for i in range(height):
        new_row = []

        for j in range(width):
            if new_table[i][j] == '':
                table[i, j].set_facecolor('black')
            else:
                table[i, j].
↪set_facecolor(get_color(heatmap[i][j],min_val,max_val))

    for goal_state in goal_states:
        table[(goal_state[0], goal_state[1])].set_facecolor("limegreen")
    table[(start_state[0], start_state[1])].set_facecolor("yellow")
    ax.axis('off')
    table.set_fontsize(16)

    for i in range(len(x_coords)):
        table[(x_coords[i], y_coords[i])].get_text().set_color('red')
    plt.show()
```

```
# this function takes 3D Q table as an input
# and outputs optimal trajectory table (policy table)
# and corresponding excpected reward values of different cells (heatmap)
def get_policy_table(q_hat_3D, start_state, goal_states):
    policy_table = []
    heatmap = []

    for i in range(q_hat_3D.shape[0]):
        row = []
        heatmap_row = []
        for j in range(q_hat_3D.shape[1]):

            heatmap_row.append(np.max(q_hat_3D[i,j,:]))

            for goal_state in goal_states:
                if (goal_state[0]==i) and (goal_state[1]==j):
                    row.append('G')

            if (start_state[0]==i) and (start_state[1]==j):
                row.append('S')
            else:
                if np.max(q_hat_3D[i,j,:]) == 0:
                    row.append(-1) # All zeros
                else:
                    row.append(np.argmax(q_hat_3D[i,j,:]))
        policy_table.append(row)
        heatmap.append(heatmap_row)

    return policy_table, heatmap

def plot_policy_from_q(q_hat, env):
    q_hat_3D = np.reshape(q_hat, (env.m_size, env.m_size, env.num_actions))
    max_val = q_hat_3D.max()
    min_val = q_hat_3D.min()
    start_state = env.get_coords_from_state(env._get_start_state)
    goal_states = env._get_goal_state
    goal_states = [env.get_coords_from_state(goal_state) for goal_state in␣
 ↪goal_states]
    policy_table, heatmap = get_policy_table(q_hat_3D, start_state, goal_states)
    x,y = get_next_cell(start_state[0],start_state[1],heatmap,policy_table)
    x_coords, y_coords = get_path(x,y,policy_table)
    plot_table(env, policy_table, heatmap, goal_states,␣
 ↪start_state,max_val,min_val, x_coords, y_coords)

    return
```

1.3 maze.py

[ ]:
```
import numpy as np
import copy
import math

ACTION_MEANING = {
    0: "UP",
    1: "RIGHT",
    2: "LEFT",
    3: "DOWN",
}

SPACE_MEANING = {
    1: "ROAD",
    0: "BARRIER",
    -1: "GOAL",
}


class MazeEnv:

    def __init__(self, start=[6,3], goals=[[1, 8]]):
        """Deterministic Maze Environment"""

        self.m_size = 10
        self.reward = 10
        self.num_actions = 4
        self.num_states = self.m_size * self.m_size

        self.map = np.ones((self.m_size, self.m_size))
        self.map[3, 4:9] = 0
        self.map[4:8, 4] = 0
        self.map[5, 2:4] = 0

        for goal in goals:
            self.map[goal[0], goal[1]] = -1

        self.start = start
        self.goals = goals
        self.obs = self.start

    def step(self, a):
        """ Perform a action on the environment

            Args:
                a (int): action integer
```

```
        Returns:
            obs (list): observation list
            reward (int): reward for such action
            done (int): whether the goal is reached
    """
    done, reward = False, 0.0
    next_obs = copy.copy(self.obs)

    if a == 0:
        next_obs[0] = next_obs[0] - 1
    elif a == 1:
        next_obs[1] = next_obs[1] + 1
    elif a == 2:
        next_obs[1] = next_obs[1] - 1
    elif a == 3:
        next_obs[0] = next_obs[0] + 1
    else:
        raise Exception("Action is Not Valid")

    if self.is_valid_obs(next_obs):
        self.obs = next_obs

    if self.map[self.obs[0], self.obs[1]] == -1:
        reward = self.reward
        done = True

    state = self.get_state_from_coords(self.obs[0], self.obs[1])

    return state, reward, done

def is_valid_obs(self, obs):
    """ Check whether the observation is valid

        Args:
            obs (list): observation [x, y]

        Returns:
            is_valid (bool)
    """

    if obs[0] >= self.m_size or obs[0] < 0:
        return False

    if obs[1] >= self.m_size or obs[1] < 0:
        return False
```

```python
        if self.map[obs[0], obs[1]] == 0:
            return False

        return True

    @property
    def _get_obs(self):
        """ Get current observation
        """
        return self.obs

    @property
    def _get_state(self):
        """ Get current observation
        """
        return self.get_state_from_coords(self.obs[0], self.obs[1])

    @property
    def _get_start_state(self):
        """ Get the start state
        """
        return self.get_state_from_coords(self.start[0], self.start[1])

    @property
    def _get_goal_state(self):
        """ Get the start state
        """
        goals = []
        for goal in self.goals:
            goals.append(self.get_state_from_coords(goal[0], goal[1]))
        return goals

    def reset(self):
        """ Reset the observation into starting point
        """
        self.obs = self.start
        state = self.get_state_from_coords(self.obs[0], self.obs[1])
        return state

    def get_state_from_coords(self, row, col):
        state = row * self.m_size + col
        return state

    def get_coords_from_state(self, state):
        row = math.floor(state/self.m_size)
        col = state % self.m_size
        return row, col
```

1.4 1. Basic Q Learning experiments

(a) Run your algorithm several times on the given environment. Use the following hyperparameters:

1. Number of episodes = 200
2. Alpha (α) learning rate = 1.0
3. Maximum number of steps per episode = 100. An episode ends when the agent reaches a goal state, or uses the maximum number of steps per episode
4. Gamma (γ) discount factor = 0.9
5. Epsilon (ϵ) for ϵ-greedy = 0.1 (10% of the time). Note that we should "break-ties" when the Q-values are zero for all the actions (happens initially) by essentially choosing uniformly from the action. So now you have two conditions to act randomly: for epsilon amount of the time, or if the Q values are all zero.

[ ]:
```
# TODO: Fill this in
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: Instantiate the MazeEnv environment with default arguments
env = MazeEnv()

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
↪max_steps, use_softmax_policy)
```

Plot the steps to goal vs training iterations (episodes):

[ ]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```

Visualize the learned greedy policy from the Q values:

[ ]:
```
# TODO: plot the policy from the Q value
plot_policy_from_q(q_hat, env)
```

(b) Run your algorithm by passing in a list of 2 goal locations: (1,8) and (5,6). Note: we are using 0-indexing, where (0,0) is top left corner. Report on the results.

[ ]:
```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False
```

```
# TODO: Set the goal
goal_locs = [[1, 8], [5, 6]]
env = MazeEnv(start = [6, 3], goals = goal_locs)

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
↪max_steps, use_softmax_policy)
```

Plot the steps to goal vs training iterations (episodes):

[ ]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```

Plot the steps to goal vs training iterations (episodes):

[ ]:
```
# TODO: plot the policy from the Q values
plot_policy_from_q(q_hat, env)
```

1.5 2. Experiment with the exploration strategy, in the original environment

(a) Try different ϵ values in ϵ-greedy exploration: We asked you to use a rate of ϵ=10%, but try also 50% and 1%. Graph the results (for 3 epsilon values) and discuss the costs and benefits of higher and lower exploration rates.

[ ]:
```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: set the epsilon lists in increasing order:
epsilon_list = [0.01, 0.1, 0.5]

env = MazeEnv()

steps_vs_iters_list = []
for epsilon in epsilon_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
↪max_steps, use_softmax_policy)
    steps_vs_iters_list.append(steps_vs_iters)
```

[ ]:
```
# TODO: Plot the results
label_list = ["epsilon={}".format(eps) for eps in epsilon_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

(b) Try exploring with policy derived from **softmax of Q-values** described in the Q learning lecture. Use the values of $\beta \in \{1, 3, 6\}$ for your experiment, keeping β fixed throughout the training.

[ ]:
```
# TODO: Fill this in for Static Beta with softmax of Q-values
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100

# TODO: Set the beta
beta_list = [1, 3, 6]
use_softmax_policy = True
k_exp_schedule = 0

env = MazeEnv()
steps_vs_iters_list = []
for beta in beta_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
 ↪max_steps, use_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

[ ]:
```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

(c) Instead of fixing the $\beta = \beta_0$ to the initial value, we will increase the value of β as the number of episodes t increase:

$$\beta(t) = \beta_0 e^{kt}$$

That is, the β value is fixed for a particular episode. Run the training again for different values of $k \in \{0.05, 0.1, 0.25, 0.5\}$, keeping $\beta_0 = 1.0$. Compare the results obtained with this approach to those obtained with a static β value.

[ ]:
```
# TODO: Fill this in for Dynamic Beta
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100

# TODO: Set the beta
beta = 1.0
use_softmax_policy = True
k_exp_schedule_list = [0.05, 0.1, 0.25, 0.5]
env = MazeEnv()
```

```
steps_vs_iters_list = []
for k_exp_schedule in k_exp_schedule_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
 ↪max_steps, use_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

[ ]:
```
# TODO: Plot the steps vs iterations
label_list = ["k={}".format(k_exp_schedule) for k_exp_schedule in␣
 ↪k_exp_schedule_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

1.6 3. Stochastic Environments

(a) Make the environment stochastic (uncertain), such that the agent only has a 95% chance of moving in the chosen direction, and has a 5% chance of moving in some random direction.

[ ]:
```
# TODO: Implement ProbabilisticMazeEnv in maze.py
class ProbabilisticMazeEnv(MazeEnv):
    """ (Q2.3) Hints: you can refer the implementation in MazeEnv
    """

    def __init__(self, goals=[[2, 8]], p_random=0.05):
        """ Probabilistic Maze Environment

            Args:
                goals (list): list of goals coordinates
                p_random (float): random action rate
        """
        super(ProbabilisticMazeEnv, self).__init__(goals=goals)
        self.p_random = p_random


    def step(self, a):
        """ Perform a action on the environment

            Args:
                a (int): action integer

            Returns:
                obs (list): observation list
                reward (int): reward for such action
                done (int): whether the goal is reached
        """
        done, reward = False, 0.0
        next_obs = copy.copy(self.obs)

        if np.random.uniform(0, 1) < self.p_random:
```

```
            a = np.random.randint(self.num_actions)

        if a == 0:
            next_obs[0] = next_obs[0] - 1
        elif a == 1:
            next_obs[1] = next_obs[1] + 1
        elif a == 2:
            next_obs[1] = next_obs[1] - 1
        elif a == 3:
            next_obs[0] = next_obs[0] + 1
        else:
            raise Exception("Action is Not Valid")

        if self.is_valid_obs(next_obs):
            self.obs = next_obs

        if self.map[self.obs[0], self.obs[1]] == -1:
            reward = self.reward
            done = True

        state = self.get_state_from_coords(self.obs[0], self.obs[1])

        return state, reward, done
```

[ ]:
```
num_iters = 200
alpha = 1
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# Set the environment probability of random
env_p_rand_list = [0.05]

steps_vs_iters_list = []
for env_p_rand in env_p_rand_list:
    # Instantiate with ProbabilisticMazeEnv
    env = ProbabilisticMazeEnv(p_random = env_p_rand)

    # Note: We will repeat for several runs of the algorithm to make the result␣
 ↪less noisy
    avg_steps_vs_iters = np.zeros(num_iters)
    for i in range(10):
        q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
 ↪max_steps, use_softmax_policy)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
```

```
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

[ ]:
```
label_list = ["env_random={}".format(env_p_rand) for env_p_rand in␣
↪env_p_rand_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

(b) Change the learning rule to handle the non-determinism, and experiment with different probability of environment performing random action $p_{rand} \in \{0.05, 0.1, 0.25, 0.5\}$ in this new rule. How does performance vary as the environment becomes more stochastic?

Use the same parameters as in first part, except change the alpha (α) value to be **less than 1**, e.g. 0.5.

[ ]:
```
# TODO: Use the same parameters as in the first part, except change alpha
num_iters = 200
alpha = 0.5
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# Set the environment probability of random
env_p_rand_list = [0.05, 0.1, 0.25, 0.5]

steps_vs_iters_list = []
for env_p_rand in env_p_rand_list:
    # Instantiate with ProbabilisticMazeEnv
    env = ProbabilisticMazeEnv(p_random = env_p_rand)

    # Note: We will repeat for several runs of the algorithm to make the result␣
↪less noisy
    avg_steps_vs_iters = np.zeros(num_iters)
    for i in range(10):
        q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
↪max_steps, use_softmax_policy)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

[ ]:
```
label_list = ["env_random={}".format(env_p_rand) for env_p_rand in␣
↪env_p_rand_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

2 3. Did you complete the course evaluation?

[ ]:
```
# Answer: yes
```

[ ]:

```

```