CS6700_PA1.ipynb

March 9, 2022

[ ]:
```
!pip install numpy matplotlib tqdm scipy
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(1.21.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
packages (3.2.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(4.63.0)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages
(1.4.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (3.0.7)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
packages (from matplotlib) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7
/dist-packages (from matplotlib) (1.3.2)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7
/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-
packages (from python-dateutil>=2.1->matplotlib) (1.15.0)
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
%matplotlib inline
```

[ ]:
```
DOWN = 0
UP = 1
LEFT = 2
RIGHT = 3
actions = [DOWN, UP, LEFT, RIGHT]
```

[ ]:
```
from math import floor
import numpy as np
```

```python
def row_col_to_seq(row_col, num_cols):  #Converts state number to row_column format
    return row_col[:,0] * num_cols + row_col[:,1]

def seq_to_col_row(seq, num_cols): #Converts row_column format to state number
    r = floor(seq / num_cols)
    c = seq - r * num_cols
    return np.array([[r, c]])
class GridWorld:
    """
    Creates a gridworld object to pass to an RL algorithm.
    Parameters
    ----------
    num_rows : int
        The number of rows in the gridworld.
    num_cols : int
        The number of cols in the gridworld.
    start_state : numpy array of shape (1, 2), np.array([[row, col]])
        The start state of the gridworld (can only be one start state)
    goal_states : numpy arrany of shape (n, 2)
        The goal states for the gridworld where n is the number of goal
        states.
    """
    def __init__(self, num_rows, num_cols, start_state, goal_states, wind = False):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self.start_state = start_state
        self.goal_states = goal_states
        self.obs_states = None
        self.bad_states = None
        self.num_bad_states = 0
        self.p_good_trans = None
        self.bias = None
        self.r_step = None
        self.r_goal = None
        self.r_dead = None
        self.gamma = 1 # default is no discounting
        self.wind = wind

    def add_obstructions(self, obstructed_states=None, bad_states=None, restart_states=None):

        self.obs_states = obstructed_states
        self.bad_states = bad_states
        if bad_states is not None:
            self.num_bad_states = bad_states.shape[0]
```

```
        else:
            self.num_bad_states = 0
        self.restart_states = restart_states
        if restart_states is not None:
            self.num_restart_states = restart_states.shape[0]
        else:
            self.num_restart_states = 0

    def add_transition_probability(self, p_good_transition, bias):

        self.p_good_trans = p_good_transition
        self.bias = bias

    def add_rewards(self, step_reward, goal_reward, bad_state_reward=None,␣
↪restart_state_reward = None):

        self.r_step = step_reward
        self.r_goal = goal_reward
        self.r_bad = bad_state_reward
        self.r_restart = restart_state_reward


    def create_gridworld(self):

        self.num_actions = 4
        self.num_states = self.num_cols * self.num_rows# +1
        self.start_state_seq = row_col_to_seq(self.start_state, self.num_cols)
        self.goal_states_seq = row_col_to_seq(self.goal_states, self.num_cols)

        # rewards structure
        self.R = self.r_step * np.ones((self.num_states, 1))
        #self.R[self.num_states-1] = 0
        self.R[self.goal_states_seq] = self.r_goal

        for i in range(self.num_bad_states):
            if self.r_bad is None:
                raise Exception("Bad state specified but no reward is given")
            bad_state = row_col_to_seq(self.bad_states[i,:].reshape(1,-1), self.
↪num_cols)
            #print("bad states", bad_state)
            self.R[bad_state, :] = self.r_bad
        for i in range(self.num_restart_states):
            if self.r_restart is None:
                raise Exception("Restart state specified but no reward is␣
↪given")
            restart_state = row_col_to_seq(self.restart_states[i,:].
↪reshape(1,-1), self.num_cols)
```

```python
            #print("restart_state", restart_state)
            self.R[restart_state, :] = self.r_restart

        # probability model
        if self.p_good_trans == None:
            raise Exception("Must assign probability and bias terms via the␣
↪add_transition_probability method.")

        self.P = np.zeros((self.num_states,self.num_states,self.num_actions))
        for action in range(self.num_actions):
            for state in range(self.num_states):


                # check if the state is the goal state or an obstructed state -␣
↪transition to end
                row_col = seq_to_col_row(state, self.num_cols)
                if self.obs_states is not None:
                    end_states = np.vstack((self.obs_states, self.goal_states))
                else:
                    end_states = self.goal_states

                if any(np.sum(np.abs(end_states-row_col), 1) == 0):
                    self.P[state, state, action] = 1

                # else consider stochastic effects of action
                else:
                    for dir in range(-1,2,1):

                        direction = self._get_direction(action, dir)
                        next_state = self._get_state(state, direction)
                        if dir == 0:
                            prob = self.p_good_trans
                        elif dir == -1:
                            prob = (1 - self.p_good_trans)*(self.bias)
                        elif dir == 1:
                            prob = (1 - self.p_good_trans)*(1-self.bias)

                        self.P[state, next_state, action] += prob

                # make restart states transition back to the start state with
                # probability 1
                if self.restart_states is not None:
                    if any(np.sum(np.abs(self.restart_states-row_col),1)==0):
                        next_state = row_col_to_seq(self.start_state, self.
↪num_cols)
                        self.P[state,:,:] = 0
                        self.P[state,next_state,:] = 1
```

```python
        return self

    def _get_direction(self, action, direction):

        left = [2,3,1,0]
        right = [3,2,0,1]
        if direction == 0:
            new_direction = action
        elif direction == -1:
            new_direction = left[action]
        elif direction == 1:
            new_direction = right[action]
        else:
            raise Exception("getDir received an unspecified case")
        return new_direction

    def _get_state(self, state, direction):

        row_change = [-1,1,0,0]
        col_change = [0,0,-1,1]
        row_col = seq_to_col_row(state, self.num_cols)
        row_col[0,0] += row_change[direction]
        row_col[0,1] += col_change[direction]

        # check for invalid states
        if self.obs_states is not None:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1) or
                np.any(np.sum(abs(self.obs_states - row_col), 1)==0)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]
        else:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]

        return next_state

    def reset(self):
      return int(self.start_state_seq)

    def step(self, state, action):
```

```python
        p, r = 0, np.random.random()
        for next_state in range(self.num_states):

            p += self.P[state, next_state, action]

            if r <= p:
                break

        if(self.wind and np.random.random() < 0.4):

          arr = self.P[next_state, :, 3]
          next_next = np.where(arr == np.amax(arr))
          next_next = next_next[0][0]
          return next_next, self.R[next_next]
        else:
          return next_state, self.R[next_state]
```

[ ]:
```python
# specify world parameters
num_cols = 10
num_rows = 10
obstructions = np.array([[0,7],[1,1],[1,2],[1,3],[1,7],[2,1],[2,3],
                         [2,7],[3,1],[3,3],[3,5],[4,3],[4,5],[4,7],
                         [5,3],[5,7],[5,9],[6,3],[6,9],[7,1],[7,6],
                         [7,7],[7,8],[7,9],[8,1],[8,5],[8,6],[9,1]])
bad_states = np.array([[1,9],[4,2],[4,4],[7,5],[9,9]])
restart_states = np.array([[3,7],[8,2]])
start_state = np.array([[3,6]])
#start_state = np.array([[0,4]])
goal_states = np.array([[0,9],[2,2],[8,7]])

# create model
gw = GridWorld(num_rows=num_rows,
               num_cols=num_cols,
               start_state=start_state,
               goal_states=goal_states, wind = False)
gw.add_obstructions(obstructed_states=obstructions,
                    bad_states=bad_states,
                    restart_states=restart_states)
gw.add_rewards(step_reward=-1,
               goal_reward=10,
               bad_state_reward=-6,
               restart_state_reward=-100)
gw.add_transition_probability(p_good_transition=0.7,
                              bias=0.5)
env = gw.create_gridworld()
```

[ ]:
```python
print("Number of actions", env.num_actions) #0 -> UP, 1-> DOWN, 2 -> LEFT, 3->␣
↪RIGHT
```

```python
print("Number of states", env.num_states)
print("start state", env.start_state_seq)
print("goal state(s)", env.goal_states_seq)
```

```
Number of actions 4
Number of states 100
start state [36]
goal state(s) [ 9 22 87]
```

[ ]:
```python
plt.figure(figsize=(10, 10))
# Go UP
start=row_col_to_seq(start_state, num_cols)
env.step(start,UP)
#env.render(ax=plt, render_agent=True)
```

[ ]:
```
(46, array([-1.]))
```

```
<Figure size 720x720 with 0 Axes>
```

[ ]:
```python
Q = np.zeros((num_rows*num_cols, len(actions)))
```

[ ]:
```python
env.P[0,:,0]
```

[ ]:
```
array([0.85, 0.15, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  ])
```

[ ]:
```python
def plot_Q(Q, message = "Q plot"):

    plt.figure(figsize=(10,10))
    plt.title(message)
    plt.pcolor(Q.max(-1), edgecolors='k', linewidths=2)
    plt.colorbar()
    def x_direct(a):
        if a in [UP, DOWN]:
            return 0
        return 1 if a == RIGHT else -1
    def y_direct(a):
        if a in [RIGHT, LEFT]:
            return 0
        return 1 if a == UP else -1
```

```
    policy = Q.argmax(-1)
    policyx = np.vectorize(x_direct)(policy)
    policyy = np.vectorize(y_direct)(policy)
    idx = np.indices(policy.shape)
    plt.quiver(idx[1].ravel()+0.5, idx[0].ravel()+0.5, policyx.ravel(), policyy.
 ↪ravel(), pivot="middle", color='red')
    plt.show()


def plot_Qstep(stepsVisitCount, message = "Steps plot"):

    plt.figure(figsize=(10,10))
    plt.title(message)
    plt.pcolor(stepsVisitCount, edgecolors='k', linewidths=1)
    plt.colorbar()
    for (x, y),z  in np.ndenumerate(stepsVisitCount.T):
        plt.text(x+.5, y+.5, f"{z:}", va="top", ha="center")
    plt.show()
```

[ ]:
```
from scipy.special import softmax
seed = 23
rg = np.random.RandomState(seed)

#EPSILON GREEDY
def choose_action_epsilon(Q, state, epsilon, rg=rg):
  if not Q[state].any() or rg.rand() < epsilon:
    return rg.choice(Q.shape[-1])
  else:
    return np.argmax(Q[state])

#SOFTMAX
def choose_action_softmax(Q, state, beta, rg=rg):
  return rg.choice(Q.shape[-1],p=softmax(beta*Q[state]))

def modifyQ(Q):
  Q_new = np.zeros((10, 10, 4))
  for i in range(100):
    row_col = seq_to_col_row(i, 10)[0]
    Q_new[row_col[0], row_col[1]] = Q[i]

  return Q_new
```

[ ]:
```
alpha0 = 0.57
gamma0 = 0.94
episodes = 20000
epsilon0 = 0.08
beta0 = 5
```

0.0.1 Q Learning

[ ]:
```
print_freq = 100
stepsVisitCount = np.zeros((10, 10))

def qLearning(env, Q, gamma, plot_heat = False, choose_action =␣
 ↪choose_action_softmax):

    Q_new = np.zeros((10, 10, 4))

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q_new)
    epsilon = epsilon0
    alpha = alpha0

    current_episode = 1
    goal_states_seq = row_col_to_seq(goal_states, num_cols)
    for current_episode in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state, epsilon)

        #while steps <= 100 and np.any((goal_states_seq) != state):
        while steps <= 100 and state not in goal_states_seq:
                state_next, reward = env.step(state, action)
                seq_next = seq_to_col_row(state_next, num_cols)[0]
                stepsVisitCount[seq_next[0], seq_next[1]]+=1
                action_next = choose_action(Q, state_next, epsilon)

                #update equation
                best_next_action = np.argmax(Q[state])

                Q[state, action] += alpha * (reward+gamma*Q[state_next,␣
 ↪best_next_action]-Q[state, action])

                tot_reward += reward
                steps += 1

                state, action = state_next, action_next

        episode_rewards[current_episode-1] = tot_reward
        steps_to_completion[current_episode-1] = steps-1
```

```
        current_episode += 1
        Q_new = modifyQ(Q)

        if (current_episode+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q_new, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax:␣
↪%.2f, Qmin: %.2f"%(current_episode+1, np.
↪mean(episode_rewards[current_episode-print_freq+1:current_episode]),
                                                                                   np.
↪mean(steps_to_completion[current_episode-print_freq+1:current_episode]),
                                                                                  ␣
↪Q_new.max(), Q_new.min()))
        episode_rewards[current_episode-1] = episode_rewards[current_episode-2]
        steps_to_completion[current_episode-1] =␣
↪steps_to_completion[current_episode-2]

    return Q, episode_rewards, steps_to_completion
```

0.0.2 Running 3 independent experiments

[ ]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print()
    print()
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_rows*num_cols, len(actions)))
    rg = np.random.RandomState(i)
    Q, rewards, steps = qLearning(env, Q, gamma = gamma0, plot_heat = True,␣
↪choose_action = choose_action_epsilon)
    #print(steps)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

[ ]:
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs, 0))
plt.show()
```

0.0.3 SAARSA

```
[ ]: from unicodedata import bidirectional
print_freq = 100

stepsVisitCount = np.zeros((10, 10))
def saarsa(env, Q, gamma, plot_heat = False, choose_action =␣
 ↪choose_action_softmax):

    Q_new = np.zeros((10, 10, 4))

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)

    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q_new)

    epsilon = epsilon0
    alpha = alpha0

    current_episode = 1
    goal_states_seq = row_col_to_seq(goal_states, num_cols)
    for current_episode in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state, beta0)
        #while steps <= 100 and np.any((goal_states_seq) != state):
        while steps <= 100 and state not in goal_states_seq:
                state_next, reward = env.step(state, action)
                seq_next = seq_to_col_row(state_next, num_cols)[0]
                stepsVisitCount[seq_next[0], seq_next[1]]+=1
                action_next = choose_action(Q, state_next, beta0)

                # update equation
                Q[state, action] += alpha*(reward + gamma*Q[state_next,␣
 ↪action_next] - Q[state, action])

                tot_reward += reward
                steps += 1

                state, action = state_next, action_next

        episode_rewards[current_episode-1] = tot_reward
        steps_to_completion[current_episode-1] = steps-1
```

```
        current_episode += 1

        Q_new = modifyQ(Q)

        if (current_episode+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q_new, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax:␣
↪%.2f, Qmin: %.2f"%(current_episode+1, np.
↪mean(episode_rewards[current_episode-print_freq+1:current_episode]),
                                                                                    np.
↪mean(steps_to_completion[current_episode-print_freq+1:current_episode]),
                                                                                   ␣
↪Q_new.max(), Q_new.min()))

        episode_rewards[current_episode-1] = episode_rewards[current_episode-2]
        steps_to_completion[current_episode-1] =␣
↪steps_to_completion[current_episode-2]

    return Q, episode_rewards, steps_to_completion
```

0.0.4 Running 3 independent experiments

[ ]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_rows*num_cols, len(actions)))
    rg = np.random.RandomState(i)
    Q, rewards, steps = saarsa(env, Q, gamma = gamma0, plot_heat = True,␣
↪choose_action = choose_action_softmax)
    print(steps)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```