Task_1.2C

March 20, 2021

[1]:
```python
import numpy as np
import gym

from gym import Env, spaces
from gym.utils import seeding


def categorical_sample(prob_n, np_random):
    """
    Sample from categorical distribution
    Each row specifies class probabilities
    """
    prob_n = np.asarray(prob_n)
    csprob_n = np.cumsum(prob_n)
    return (csprob_n > np_random.rand()).argmax()


class DiscreteEnv(Env):

    """
    Has the following members
    - nS: number of states
    - nA: number of actions
    - P: transitions (*)
    - isd: initial state distribution (**)
    (*) dictionary of lists, where
      P[s][a] == [(probability, nextstate, reward, done), ...]
    (**) list or array of length nS
    """
    def __init__(self, nS, nA, P, isd):
        self.P = P
        self.isd = isd
        self.lastaction = None  # for rendering
        self.nS = nS
        self.nA = nA

        self.action_space = spaces.Discrete(self.nA)
```

```python
        self.observation_space = spaces.Discrete(self.nS)

        self.seed()
        self.s = categorical_sample(self.isd, self.np_random)

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def reset(self):
        self.s = categorical_sample(self.isd, self.np_random)
        self.lastaction = None
        return int(self.s)

    def step(self, a):
        transitions = self.P[self.s][a]
        i = categorical_sample([t[0] for t in transitions], self.np_random)
        p, s, r, d = transitions[i]
        self.s = s
        self.lastaction = a
        return (int(s), r, d, {"prob": p})
```

[2]:

```python
import sys
from contextlib import closing
from io import StringIO
from gym import utils
from gym.envs.toy_text import discrete
import numpy as np



MAP = [
    "+-------------------+",
    "| :A| : :B: : | :C| |",
    "| : | : | : | : | : |",
    "| : | : | : | : | : |",
    "| : : : | : : : : : |",
    "| : | : : : | : | : |",
    "| : | : | : | : | : |",
    "| : | : : : | : | : |",
    "| | : : | : : | : : |",
    "| :D| : :E: : : |F| |",
    "| | : : | : | | : : |",
    "+-------------------+",
]

class TaxiEnv(discrete.DiscreteEnv):
```

```
    """
    Description:
    There are 8 designated locations in the grid world indicated by A, B, C, D,␣
↪E, F . When the episode starts, the taxi starts off at a random square and␣
↪the passenger is at a random location. The taxi drives to the passenger's␣
↪location, picks up the passenger, drives to the passenger's destination␣
↪(another one of the four specified locations), and then drops off the␣
↪passenger. Once the passenger is dropped off, the episode ends.

    Observations:
    There are 4200 discrete states since there are 100 taxi positions, 7␣
↪possible locations of the passenger (including the case when the passenger␣
↪is in the taxi), and 6 destination locations.

    Passenger locations:
    - 0: A
    - 1: B
    - 2: C
    - 3: D
    - 4: E
    - 5: F
    - 6: in taxi

    Destinations:
    - 0: A
    - 1: B
    - 2: C
    - 3: D
    - 4: E
    - 5: F


    Actions:
    There are 6 discrete deterministic actions:
    - 0: move up
    - 1: move down
    - 2: move left
    - 3: move right
    - 4: pickup passenger
    - 5: drop off passenger

    Rewards:
    There is a default per-step reward of -1,
    except for delivering the passenger, which is +10,
    or executing "pickup" and "drop-off" actions illegally, which is -5.

    Rendering:
```

```
    - blue: passenger
    - magenta: destination
    - red: empty taxi
    - green: full taxi
    - other letters (A, B, C, D, E, F, G, H): locations for passengers and␣
↪destinations
    state space is represented by:
        (taxi_row, taxi_col, passenger_location, destination)
    """
    metadata = {'render.modes': ['human', 'ansi']}

    def __init__(self):
        self.desc = np.asarray(MAP, dtype='c')

        self.locs = locs = [(0, 1), (0, 4), (0, 8), (8, 1), (8, 4), (8, 8)]

        num_states = 4200
        num_rows = 10
        num_columns = 10
        max_row = num_rows - 1
        max_col = num_columns - 1
        initial_state_distrib = np.zeros(num_states)
        num_actions = 6
        P = {state: {action: []
                     for action in range(num_actions)} for state in␣
↪range(num_states)}
        for row in range(num_rows):

            for col in range(num_columns):

                for pass_idx in range(len(locs) + 1):  # +1 for being inside␣
↪taxi

                    for dest_idx in range(len(locs)):
                        state = self.encode(row, col, pass_idx, dest_idx)

                        if pass_idx < 6 and pass_idx != dest_idx:
                            initial_state_distrib[state] += 1

                        for action in range(num_actions):
                            # defaults
                            new_row, new_col, new_pass_idx = row, col, pass_idx
                            reward = -1  # default reward when there is no␣
↪pickup/dropoff

                            done = False
                            taxi_loc = (row, col)
```

```
                            if action == 0:
                                new_row = min(row + 1, max_row)

                            elif action == 1:
                                new_row = max(row - 1, 0)

                            if action == 2 and self.desc[1 + row, 2 * col + 2] == b":":

                                new_col = min(col + 1, max_col)

                            elif action == 3 and self.desc[1 + row, 2 * col] == b":":

                                new_col = max(col - 1, 0)

                            elif action == 4:  # pickup

                                if (pass_idx < 6 and taxi_loc == locs[pass_idx]):

                                    new_pass_idx = 6

                                else:  # passenger not at location
                                    reward = -3

                            elif action == 5:  # dropoff

                                # if the taxi decides to drop off and it reaches the correct destination after exploring all the destination, reward is 10 and the experiment stop
                                if (taxi_loc == locs[dest_idx] and pass_idx == 6):

                                    new_pass_idx = dest_idx
                                    done = True
                                    reward = 10

                                # if the taxi decides to drop off and hasnt reach the correct destination, reward is -5
                                elif (taxi_loc in locs):
                                    new_pass_idx = locs.index(taxi_loc)

                                    if (new_pass_idx != dest_idx):
                                        done = False
                                        reward = -5
```

```
                            new_state = self.encode(new_row, new_col,␣
↪new_pass_idx, dest_idx)

                            P[state][action].append((1.0, new_state, reward,␣
↪done))

        initial_state_distrib /= initial_state_distrib.sum()

        discrete.DiscreteEnv.__init__(
            self, num_states, num_actions, P, initial_state_distrib)

    def encode(self, taxi_row, taxi_col, pass_loc, dest_idx):
        # (10) 10, 7, 6
        i = taxi_row
        i *= 10
        i += taxi_col
        i *= 7
        i += pass_loc
        i *= 6
        i += dest_idx
        return i

    def decode(self, i):
        out = []
        out.append(i % 6)
        i = i // 6
        out.append(i % 7)
        i = i // 7
        out.append(i % 10)
        i = i // 10
        out.append(i)
        assert 0 <= i < 10
        return reversed(out)

    def render(self, mode='human'):
        outfile = StringIO() if mode == 'ansi' else sys.stdout

        out = self.desc.copy().tolist()
        out = [[c.decode('utf-8') for c in line] for line in out]

        taxi_row, taxi_col, pass_idx, dest_idx = self.decode(self.s)


        #print("taxi_row:{}, taxi_col: {}, pass_idx: {}, dest_idx: {}".
↪format(taxi_row, taxi_col, pass_idx, dest_idx))
```

```
        def ul(x): return "_" if x == " " else x

        if pass_idx < 8:
            #print("pass_idx: {}".format(pass_idx))
            #print("[1 + taxi_row]: {}, [2 * taxi_col + 1]: {}".format([1 +␣
↪taxi_row],[2 * taxi_col + 1]))
            out[1 + taxi_row][2 * taxi_col + 1] = utils.colorize(
                out[1 + taxi_row][2 * taxi_col + 1], 'red', highlight=True)
            #print("out[1 + taxi_row][2 * taxi_col + 1]: {}".format(out[1 +␣
↪taxi_row][2 * taxi_col + 1]))

            pi, pj = self.locs[pass_idx]
            #print("\npi: {}, pj: {}".format(pi, pj))
            #print("[1 + pi]: {}, [2 * pj + 1]: {}".format([1 + pi],[2 * pj +␣
↪1]))
            out[1 + pi][2 * pj + 1] = utils.colorize(
                out[1 + pi][2 * pj + 1], 'blue', bold=True)
            #print("out[1 + pi][2 * pj + 1]: {}".format(out[1 + pi][2 * pj +␣
↪1]))

        else:  # passenger in taxi
            #print("[1 + taxi_row]: {}, [2 * taxi_col + 1]: {}".format([1 +␣
↪taxi_row],[2 * taxi_col + 1]))
            out[1 + taxi_row][2 * taxi_col + 1] = utils.colorize(
                ul(out[1 + taxi_row][2 * taxi_col + 1]), 'green',␣
↪highlight=True)
            #print("out[1 + taxi_row][2 * taxi_col + 1]: {}".format(out[1 +␣
↪taxi_row][2 * taxi_col + 1]))

        di, dj = self.locs[dest_idx]
        #print("\ndi: {}, dj: {}".format(di, dj))
        #print("[1 + di]: {}, [2 * dj + 1]: {}".format([1 + di],[2 * dj + 1]))
        out[1 + di][2 * dj + 1] = utils.colorize(out[1 + di][2 * dj + 1],␣
↪'magenta')
        #print("out[1 + di][2 * dj + 1]: {}".format(out[1 + di][2 * dj + 1]))


        outfile.write("\n".join(["".join(row) for row in out]) + "\n")
        if self.lastaction is not None:
            outfile.write("  ({})\n".format(["Down", "Up", "Right", "Left",␣
↪"Pickup", "Dropoff"][self.lastaction]))
        else:
            outfile.write("\n")

        # No need to return anything for human
        if mode != 'human':
```

```
            with closing(outfile):
                return outfile.getvalue()
```

[3]:
```
from gym.envs.registration import register

register(
    id='smart_cab-v2',
    entry_point='smart_cab.envs:TaxiEnv')
```

[4]:
```
import gym

# core gym interface is env
env = gym.make('smart_cab:smart_cab-v2')

env.render()
```

```
+--------------------+
| :A| : :B: : | :C| |
| : | : | : | : | : |
| : | : | : | : | : |
| : : : | : : : : : |
| : | : : : | : | : |
| : | : | : | : | : |
| : | : : : | : | : |
| | : : | : : | : : |
| :D| : :E: : : |F| |
| | : : | : | | : : |
+--------------------+
```

[5]:
```
# env.reset(): Resets the environment and returns a random initial state.
env.reset()

# env.render(): Renders one frame of the environment (helpful in visualizing␣
 ↪the environment)
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))

text = """
The filled square represents the taxi, which is yellow without a passenger and␣
 ↪green with a passenger.

The pipe ("|") represents a wall which the taxi cannot cross.

A, B, C, D, E, F are the possible pickup and destination locations.
```

```
The blue letter represents the current passenger pick-up location.

The pink letter is the current drop-off location.
"""

print(text)
```

```
+-------------------+
| :A| : :B: : | :C| |
| : | : | : | : | : |
| : | : | : | : | : |
| : : : | : : : : : |
| : | : : : | : | : |
| : | : | : | : | : |
| : | : : : | : | : |
| | : : | : : | : : |
| :D| : :E: : : |F| |
| | : : | : | | : : |
+-------------------+

Action Space Discrete(6)
State Space Discrete(4200)

The filled square represents the taxi, which is yellow without a passenger and
green with a passenger.

The pipe ("|") represents a wall which the taxi cannot cross.

A, B, C, D, E, F are the possible pickup and destination locations.

The blue letter represents the current passenger pick-up location.

The pink letter is the current drop-off location.
```

[6]:
```
# (taxi row, taxi column, passenger location index, drop-off location index)
# Pick-up/Drop-off --> A - 0, B - 1, C - 2, D - 3, E - 4, F - 5
# Manually set the state and  give it to the environment
state = env.encode(0, 1, 2, 3)
print("State:", state)

# A number is generated corresponding to a state between 0 and 4200, which
 ↪turns out to be 57.

env.s = state
env.render()
```

```
State: 57
+---------------------+
| :A| : :B: : | :C| |
| : | : | : | : | : |
| : | : | : | : | : |
| : : : | : : : : : |
| : | : : : | : | : |
| : | : | : | : | : |
| : | : : : | : | : |
| | : : | : : | : : |
| :D| : :E: : : |F| |
| | : : | : | | : : |
+---------------------+
```

[7]:

```
# Reward Table




text = """
Output is default reward values assigned to each state.

This dictionary has the structure {action: [(probability, nextstate, reward,␣
 ↪done)]}.

The 0-5 corresponds to the actions (south, north, east, west, pickup, dropoff)␣
 ↪the taxi can perform at our current state in the illustration.

In this env, probability is always 1.0.

The nextstate is the state we would be in if we take the action at this index␣
 ↪of the dict

All the movement actions have a -1 reward and the pickup/dropoff actions have␣
 ↪-5 reward in this particular state.

If we are in a state where the taxi has a passenger and is on top of the right␣
 ↪destination, we would see a reward of 10 at the dropoff action (5)

""done"" is used to tell us when we have successfully dropped off a passenger␣
 ↪in the right location. Each successfull dropoff is the end of an episode

If the taxi hits the wall, it will accumulate a -1 as well and this will affect␣
 ↪a the long-term reward.
"""

print(text)
```

```
env.P[57]
```

```
Output is default reward values assigned to each state.

This dictionary has the structure {action: [(probability, nextstate, reward,
done)]}.

The 0-5 corresponds to the actions (south, north, east, west, pickup, dropoff)
the taxi can perform at our current state in the illustration.

In this env, probability is always 1.0.

The nextstate is the state we would be in if we take the action at this index of
the dict

All the movement actions have a -1 reward and the pickup/dropoff actions have -5
reward in this particular state.

If we are in a state where the taxi has a passenger and is on top of the right
destination, we would see a reward of 10 at the dropoff action (5)

""done"" is used to tell us when we have successfully dropped off a passenger in
the right location. Each successfull dropoff is the end of an episode

If the taxi hits the wall, it will accumulate a -1 as well and this will affect
a the long-term reward.
```

[7]:
```
{0: [(1.0, 477, -1, False)],
 1: [(1.0, 57, -1, False)],
 2: [(1.0, 57, -1, False)],
 3: [(1.0, 15, -1, False)],
 4: [(1.0, 57, -3, False)],
 5: [(1.0, 45, -5, False)]}
```

Without Reinforcement Learning

[8]:
```
env.s = 57  # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False
```

```python
# create an infinite loop which runs until one passenger reaches one␣
↪destination (one episode), or in other words, when the received reward is 10
while not done:
  # take the next action
  action = env.action_space.sample()
  state, reward, done, info = env.step(action)

  if reward == -5 or reward == -3:
      penalties += 1

  epochs += 1


print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))

text = """
The agent takes thousands of timesteps and makes lots of wrong drop offs to␣
↪deliver just one passenger to the right destination.

This is because we aren't learning from past experience.

It can run this over and over, and it will never optimize as the agent has no␣
↪memory of which action was best for each state.
"""

print(text)
```

```
Timesteps taken: 391
Penalties incurred: 63

The agent takes thousands of timesteps and makes lots of wrong drop offs to
deliver just one passenger to the right destination.

This is because we aren't learning from past experience.

It can run this over and over, and it will never optimize as the agent has no
memory of which action was best for each state.
```

1 With Reinforcement Learning

[9]:
```
# train the agent

import numpy as np
q_table = np.zeros([env.observation_space.n, env.action_space.n])

print(q_table)
```

```
[[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.]]
```

[10]:
```
%%time
"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters
alpha = 0.1
gamma = 0.6
epsilon = 0.1

# For plotting metrics
all_epochs = []
all_penalties = []

for i in range(1, 200001):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

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

        old_value = q_table[state, action]
```

```
        next_max = np.max(q_table[next_state])

        new_value = (1 - alpha) * old_value + alpha * (reward + gamma *␣
↪next_max)
        q_table[state, action] = new_value

        if reward == -5 or reward == -3:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 10000 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```

```
Episode: 200000
Training finished.

CPU times: user 1min 13s, sys: 351 ms, total: 1min 13s
Wall time: 1min 13s
```

[11]:

```
text = """
0 = down
1 = up
2 = right
3 = left
4 = pickup
5 = dropoff
"""

print(text)

action = ['down', 'up', 'right', 'left', 'pick-up', 'drop-off']

index_action = list(q_table[57]).index(np.max(q_table[57]))

text_2 = """
The max Q-value is {}, which is '{}', showing that Q-learning has effectively␣
↪learned
the best action to take in that current state!""".format(np.max(q_table[57]),␣
↪action[index_action])

print("q_table[57]: {}".format(q_table[57]))
print(text_2)
```

```
0 = down
1 = up
2 = right
3 = left
4 = pickup
5 = dropoff

q_table[57]: [-2.39607006 -2.40324751 -2.4001698  -2.40455825 -3.12968968
-3.95714678]

The max Q-value is -2.3960700613978037, which is 'down', showing that Q-learning
has effectively learned
the best action to take in that current state!
```

[12]:
```python
"""Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 10

# For all 10 successful passenger drop-off
for i in range(episodes):

    # Resets the environment and returns a random initial state
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        # use the current q table with its current state to do the next action
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        if reward == -5 or reward == -3:
            penalties += 1

        epochs += 1


    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
# 0 penalties incurred after Reinforcement Learning
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
Results after 10 episodes:
Average timesteps per episode: 11.2
Average penalties per episode: 0.0
```

[ ]:

```

```