

13.1.OMIMS-RL

December 17, 2019

1 Reinforcement Learning

Gym provides different game environments. We will use the Gym environment called Taxi-V2.

1.1 The Taxi Problem

from “Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition” by Tom Dietterich

Description:

There are four designated locations in the grid world indicated by R, B, G, and Y. When the episode starts, the taxi starts off at a random square and the passenger is at a random location (R, b, G, or Y). The taxi drive to the passenger’s location, pick up the passenger, drive to the passenger’s destination (another one of the four specified locations), and then drop off the passenger. Once the passenger is dropped off, the episode ends.

Observations:

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is the taxi), and 4 destination locations.

Actions: There are 6 discrete deterministic actions: - 0: move south - 1: move north - 2: move east - 3: move west - 4: pickup passenger - 5: dropoff passenger

Rewards:

There is a reward of -1 for each action and an additional reward of +20 for delivering the passenger. There is a reward of -10 for executing actions “pickup” and “dropoff” illegally.

Rendering:

- blue: passenger
- magenta: destination
- yellow: empty taxi
- green: full taxi
- other letters: location

1.2 gym

The core gym interface is *env*, which is the unified environment interface. The following are the env methods:

- *env.reset* : resets the environment and returns a random initial state.
- *env.step(action)* : step the environment by one timestep. Returns
- observation: observations of the environment


- reward : if your action was beneficial or not
- done : indicates if we have successfully picked up and dropped off a passenger, also called one episode
- info : additional info such as performance and latency for debugging purposes
- `env.render` : renders one frame of the environment (helpful in visualizing the environment)

```
In [1]: # pip install --no-index -f https://github.com/Kojoley/atari-py/releases at
        # pip install gym
```

```
import gym
```

```
env = gym.make("Taxi-v2").env
```

```
env.render()
```


```
+-----+
|R: | :  :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+-----+
```

- 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.
- R, G, Y, B are the possible pickup and destination locations. The blue letter represents the current passenger pick-up location, and the purple letter is the current destination.

```
In [2]: env.reset() # reset environment to a new, random state
        env.render()
```

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

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

```
+-----+
|B: | : :G|
| : : :  : |
| : : : : |
| | : | : |
|Y| : |B: |
+-----+
```

```
Action Space Discrete(6)
```

```
State Space Discrete(500)
```

We have an Action Space of size 6 and a State Space of size 500.

Recall that the 500 states correspond to a encoding of the taxi's location, the passenger's location (R,Y,G,B + "in the taxi"), and the destination location.

1.3 Initialization

The rows are numbered from 0 to 4 and the columns from 0 to 4.

We have the taxi at row 3, column 1, our passenger is at location 2 (Y), and our destination is location 0 (R).

In short, the taxi has to pick up the passenger at Y and drop him off at R. During the taxi ride, the taxi will be in green.

Our current state is 328

```
In [3]: state = env.encode(3, 1, 2, 0) # (taxi row, taxi column, passenger index, destination index)
        print("State:", state)

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

```
State: 328
+-----+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+-----+
```

1.4 The Reward Table

When the Taxi environment is created, there is an initial Reward table that's also created, called P. We can think of it like a matrix that has the number of states as rows and number of actions as columns, i.e. a states \times actions matrix.

```
In [4]: env.P[328]

Out[4]: {0: [(1.0, 428, -1, False)],
         1: [(1.0, 228, -1, False)],
         2: [(1.0, 348, -1, False)],
         3: [(1.0, 328, -1, False)],
         4: [(1.0, 328, -10, False)],
         5: [(1.0, 328, -10, False)]}
```

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

A few things to note:

- 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.
- `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 -10 reward in this particular state.
- `done` is used to tell us when we have successfully dropped off a passenger in the right location. Each successful dropoff is the end of an episode

1.4.1 Solving the environment without Reinforcement Learning

Let's see what would happen if we try to **brute-force** our way to solving the problem without RL. Since we have our P table for default rewards in each state, we can try to have our taxi navigate just using that.

We'll create an infinite loop which runs until one passenger reaches one destination (one episode), or in other words, when the received reward is 20.

The `env.action_space.sample()` method automatically selects one random action from set of all possible actions.

The code below generate random actions until one passenger reaches one destination. We record all the actions in the `frames` object

```
In [5]: env.s = 328 # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False

while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if reward == -10:
        penalties += 1

    # Put each rendered frame into dict for animation
    frames.append({
        'frame': env.render(mode='ansi'),
        'state': state,
        'action': action,
        'reward': reward
    })

epochs += 1
```

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

```
Timesteps taken: 2949
Penalties incurred: 919
```

Let's have a look at the actions performed to reach the destination...

```
In [ ]: from IPython.display import clear_output
        from time import sleep

        def print_frames(frames, sleep_param):
            for i, frame in enumerate(frames):
                clear_output(wait=True)
                print(frame['frame'].getvalue())
                print(f"Timestep: {i + 1}")
                print(f"State: {frame['state']}")
                print(f"Action: {frame['action']}")
                print(f"Reward: {frame['reward']}")
                sleep(sleep_param)

        print_frames(frames, 0.01)
```

```
+-----+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+-----+
(Dropoff)
```

```
Timestep: 851
State: 188
Action: 5
Reward: -10
```

Poor performance. Our 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. We can run this over and over, and it will never optimize. The agent has no memory of which action was best for each state, which is exactly what Reinforcement Learning will do for us.

1.4.2 Reinforcement Learning with Q-learning

First, we'll initialize the Q-table to a 500×6 matrix of zeros.

```
In [7]: import numpy as np
        q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

We can now create the training algorithm that will update this *Q-table* as the agent explores the environment over thousands of episodes

We will use the *ϵ -greedy algorithm*. This is done simply by using the epsilon value and comparing it to the *random.uniform(0, 1)* function, which returns an arbitrary number between 0 and 1.

We execute the chosen action in the environment to obtain the *next_state* and the reward from performing the action. After that, we calculate the maximum Q-value for the actions corresponding to the *next_state*, and with that, we can easily update our *Q-value*.

```
In [8]: %%time
        #time spent on the cell execution

        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, 100001):
            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 == -10:
```

```

        penalties += 1

    state = next_state
    epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

    print("Training finished.\n")

```

```

Episode: 100000
Training finished.

```

```

Wall time: 1min 6s

```

Now that the Q-table has been established over 100,000 episodes, let's see what the Q-values are at our illustration's state:

```

In [9]: q_table[328]

Out[9]: array([-2.40379161, -2.27325184, -2.41354915, -2.35749303,
               -9.94674854, -10.68594347])

```

The *max Q-value* is `q_table[1]` corresponding to "north", so it looks like Q-learning has effectively learned the best action to take in our illustration's state

1.4.3 Performance Evaluation

We generate 20 episodes. For each of them, the moves of the taxi are generated by selecting the action with the highest *Q-value*.

```

In [10]: total_epochs, total_penalties = 0, 0
         episodes = 20

         frames = [] # for animation

         for _ in range(episodes):
             state = env.reset()
             epochs, penalties, reward = 0, 0, 0

             done = False

             while not done:
                 action = np.argmax(q_table[state])
                 state, reward, done, info = env.step(action)

                 if reward == -10:

```

```

        penalties += 1
        # Put each rendered frame into dict for animation
        frames.append({
            'frame': env.render(mode='ansi'),
            'state': state,
            'action': action,
            'reward': reward
        })

    epochs += 1

    total_penalties += penalties
    total_epochs += epochs

    print(f"Results after {episodes} episodes:")
    print(f"Average timesteps per episode: {total_epochs / episodes}")
    print(f"Average penalties per episode: {total_penalties / episodes}")

```

Results after 20 episodes:
Average timesteps per episode: 12.4
Average penalties per episode: 0.0

Display of the path for each of the 20 episodes...

```
In [11]: print_frames(frames, 0.3)
```

```

+-----+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+-----+
(Dropoff)

```

```

Timestep: 248
State: 475
Action: 5
Reward: 20

```

1.5 A Shortest-Path Problem with the Weightings Set to 1

We would like to determine the shortest path between node 0 and node 4. All the weights are set to 1!

This does not work if all the weights are not equal !

Let's first create the graph.


```
In [12]: import numpy as np
import pylab as plt
```

```
# map cell to cell, add circular cell to goal point
points_list = [(0,1), (1,2), (0,2), (1,3), (2,3), (1,4), (3,4)]
```

```
In [13]: goal = 4
```

```
import networkx as nx
```

```
G=nx.Graph()
```

```
G.add_edges_from(points_list)
```

```
pos = nx.spring_layout(G)
```

```
print(pos)
```

```
nx.draw_networkx_nodes(G,pos)
```

```
nx.draw_networkx_edges(G,pos)
```

```
nx.draw_networkx_labels(G,pos)
```

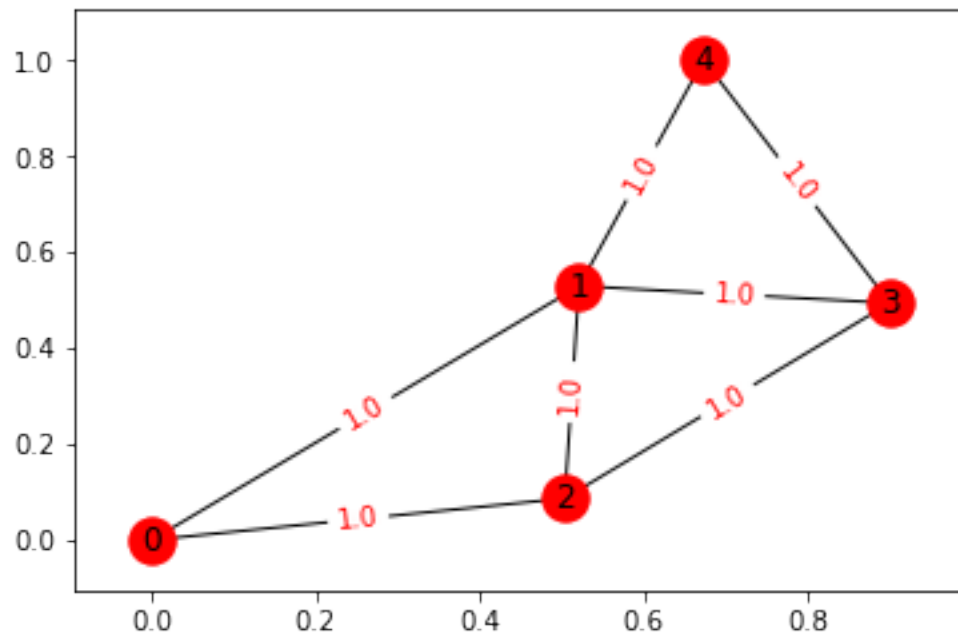
```
weights = (len(points_list)) * [1.0]
```

```
edge_labels = dict(zip(points_list, weights))
```

```
nx.draw_networkx_edge_labels(G,pos,edge_labels, font_color= 'red')
```

```
plt.show()
```

```
{0: array([ 0.,  0.]), 1: array([ 0.52098332,  0.52922075]), 2: array([ 0.50338244,
```



We then create the rewards matrix. Several choices are possible. We initialize this matrix with -10 except for *possible* actions/moves where the reward is set to 0 and 100 if we reach the goal.

```
In [14]: # how many points in graph
MATRIX_SIZE = 5

# create matrix x*y
R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))
R *= -10.0

for i, point in enumerate(points_list):
    if point[1] == goal:
        R[point] = 100
    else:
        R[point] = 0

    if point[0] == goal:
        R[point[::-1]] = 100
    else:
        # reverse of point
        R[point[::-1]] = 0

# add goal point round trip
R[goal,goal]= 100

R
```

```
Out[14]: matrix([[ -10.,    0.,    0.,  -10.,  -10.],
 [    0.,  -10.,    0.,    0.,  100.],
 [    0.,    0.,  -10.,    0.,  -10.],
 [ -10.,    0.,    0.,  -10.,  100.],
 [ -10.,    0.,  -10.,    0.,  100.]])
```

To read the above matrix, the y-axis is the state or where your bot is currently located, and the x-axis is your possible next actions.

We then build our Q-learning matrix which will hold all the lessons learned from our bot.

```
In [15]: # Q matrix

Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

# Gamma learning parameter
gamma = 0.8

# Initial state. (Usually to be chosen at random)
initial_state = 0
```

```

# This function returns all available actions in the state given as an arg
def available_actions(state):
    current_state_row = R[state,]
    av_act = np.where(current_state_row >= -10)[1]
    return av_act

# Get available actions in the current state
available_act = available_actions(initial_state)

# This function chooses at random which action to be performed within the
# of all the available actions.
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_act,1))
    return next_action

# Sample next action to be performed
action = sample_next_action(available_act)

# This function updates the Q matrix according to the path selected and the
# learning algorithm
def update(current_state, action, gamma):

    max_index = np.where(Q[action,] == np.max(Q[action,]))[1]

    if max_index.shape[0] > 1:
        max_index = int(np.random.choice(max_index, size = 1))
    else:
        max_index = int(max_index)
    max_value = Q[action, max_index]

    # Q learning formula

    Q[current_state, action] = R[current_state, action] + gamma * max_value
    # print('max_value', R[current_state, action] + gamma * max_value)

# Update Q matrix
update(initial_state, action, gamma)

```

We can now create the training algorithm that will update this Q-table over 100 episodes

```
In [16]: np.set_printoptions(precision=2)
```

```
episodes = 100
```

```

scores = [] # for animation

for _ in range(epochs):

    done = False

    while not done:
        current_state = np.random.randint(0, int(Q.shape[0]))
        available_act = available_actions(current_state)
        action = sample_next_action(available_act)
        score = update(current_state, action, gamma)
        scores.append(score)
        if action == goal :
            done = True

    print(f"Q-table after {epochs} episodes:")

    print(100.*Q/(np.max(Q)))

```

Q-table after 100 episodes:

```

[[ 61.98  80.    63.85  75.89  77.8 ]
 [ 64.    77.98  64.    80.    100. ]
 [ 64.    78.93  61.83  80.    77.8 ]
 [ 61.98  80.    64.    77.98  100. ]
 [ 61.98  78.93  61.98  80.    99.77]]

```

Display of the shortest path following the actions given by the largest Q-values

```

In [17]: # Determining the shortest path
current_state = 0
steps = [current_state]

while current_state != goal:

    next_step_index = np.where(Q[current_state,] == np.max(Q[current_state,]))

    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)

    steps.append(next_step_index)
    current_state = next_step_index

print("Shortest path:")
print(steps)

```

Shortest path:

```
[0, 1, 4]
```

```
In [ ]:
```