Department of Computer Science and Engineering (Data Science)

Subject: Reinforcement Learning

AY: 2022 - 23

Experiment 8

Q Learning Algorithm

AIM :
To implement the Q Learning algorithm in the Grid World environment

THEORY:

Q Learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations. Q-learning is another type of TD method. The ‘q’ in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward. The difference between SARSA and Q-learning is that SARSA is an on-policy model while Q-learning is off-policy.

In the Q-Learning algorithm, the goal is to iteratively learn the optimal Q-value function using the Bellman Optimality Equation. To do so, we store all the Q-values in a table that we will update at each time step using the Q-Learning iteration:.

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

where α is the learning rate, an important hyperparameter that we need to tune since it controls the convergence.

Off-Policy learning:

Off-Policy learning algorithms evaluate and improve a policy that is different from Policy that is used for action selection. In short, [Target Policy != Behavior Policy]. This helps speed up the convergence i.e learning can be fast.

ALGORITHM:

Set values for learning rate α、 discount rate γ、 reward matrix R

Initialize $Q(s,a)$ to zeros

Repeat for each episode,do

    Select state s randomly

    Repeat for each step of episode,do

        Choose a from s using ε-greedy policy or Boltzmann policy

        Take action a obtain reward r from R, and next state s'

        Update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

        Set $s = s'$

        Until s is the terminal state

    End do

End do

LAB ASSIGNMENT TO DO:

1. Initialize the Grid World environment and implement the Q Learning algorithm
2. Display the initial and final Q-tables
3. Plot the learning curve for different values of alpha(learning rate), gamma(discount factor) and draw your conclusions

Name : Sarvagya Singh
SAP ID : 60009200030
Batch : K1

In [ ]:

```
import numpy as np
import operator
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:

```
class GridWorld:
    ## Initialise starting data
    def __init__(self):
        # Set information about the gridworld
        self.height = 5
        self.width = 5
        self.grid = np.zeros(( self.height, self.width)) - 1

        # Set random start location for the agent
        self.current_location = ( 4, np.random.randint(0,5))

        # Set locations for the bomb and the gold
        self.bomb_location = (1,3)
        self.gold_location = (0,3)
        self.terminal_states = [ self.bomb_location, self.gold_location]

        # Set grid rewards for special cells
        self.grid[ self.bomb_location[0], self.bomb_location[1]] = -10
        self.grid[ self.gold_location[0], self.gold_location[1]] = 10

        # Set available actions
        self.actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']


    ## Put methods here:
    def get_available_actions(self):
        """Returns possible actions"""
        return self.actions

    def agent_on_map(self):
        """Prints out current location of the agent on the grid (used for debugging)"""
        grid = np.zeros(( self.height, self.width))
        grid[ self.current_location[0], self.current_location[1]] = 1
        return grid

    def get_reward(self, new_location):
        """Returns the reward for an input position"""
        return self.grid[ new_location[0], new_location[1]]


    def make_step(self, action):
        """Moves the agent in the specified direction. If agent is at a border, agent sta
ys still
        but takes negative reward. Function returns the reward for the move."""
        # Store previous location
        last_location = self.current_location

        # UP
        if action == 'UP':
            # If agent is at the top, stay still, collect reward
            if last_location[0] == 0:
                reward = self.get_reward(last_location)
            else:
                self.current_location = ( self.current_location[0] - 1, self.current_loc
```

```
ation[1])
                reward = self.get_reward(self.current_location)

        # DOWN
        elif action == 'DOWN':
            # If agent is at bottom, stay still, collect reward
            if last_location[0] == self.height - 1:
                reward = self.get_reward(last_location)
            else:
                self.current_location = ( self.current_location[0] + 1, self.current_loc
ation[1])
                reward = self.get_reward(self.current_location)

        # LEFT
        elif action == 'LEFT':
            # If agent is at the left, stay still, collect reward
            if last_location[1] == 0:
                reward = self.get_reward(last_location)
            else:
                self.current_location = ( self.current_location[0], self.current_locatio
n[1] - 1)
                reward = self.get_reward(self.current_location)

        # RIGHT
        elif action == 'RIGHT':
            # If agent is at the right, stay still, collect reward
            if last_location[1] == self.width - 1:
                reward = self.get_reward(last_location)
            else:
                self.current_location = ( self.current_location[0], self.current_locatio
n[1] + 1)
                reward = self.get_reward(self.current_location)

        return reward

    def check_state(self):
        """Check if the agent is in a terminal state (gold or bomb), if so return 'TERMIN
AL'"""
        if self.current_location in self.terminal_states:
            return 'TERMINAL'
```

In [ ]:

```
class RandomAgent():
    # Choose a random action
    def choose_action(self, available_actions):
        """Returns a random choice of the available actions"""
        return np.random.choice(available_actions)
```

In [ ]:

```
class Q_Agent():
    # Intialise
    def __init__(self, environment, epsilon=0.05, alpha=0.1, gamma=1):
        self.environment = environment
        self.q_table = dict() # Store all Q-values in dictionary of dictionaries
        for x in range(environment.height): # Loop through all possible grid spaces, cre
ate sub-dictionary for each
            for y in range(environment.width):
                self.q_table[(x,y)] = {'UP':0, 'DOWN':0, 'LEFT':0, 'RIGHT':0} # Populate
sub-dictionary with zero values for possible moves

        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma

    def choose_action(self, available_actions):
        """Returns the optimal action from Q-Value table. If multiple optimal actions, ch
ooses random choice.
        Will make an exploratory random action dependent on epsilon."""
        if np.random.uniform(0,1) < self.epsilon:
```

```
            action = available_actions[np.random.randint(0, len(available_actions))]
        else:
            q_values_of_state = self.q_table[self.environment.current_location]
            maxValue = max(q_values_of_state.values())
            action = np.random.choice([k for k, v in q_values_of_state.items() if v == m
axValue])

        return action

    def learn(self, old_state, reward, new_state, action):
        """Updates the Q-value table using Q-learning"""
        q_values_of_state = self.q_table[new_state]
        max_q_value_in_new_state = max(q_values_of_state.values())
        current_q_value = self.q_table[old_state][action]

        self.q_table[old_state][action] = (1 - self.alpha) * current_q_value + self.alph
a * (reward + self.gamma * max_q_value_in_new_state)
```

In [ ]:

```
def play(environment, agent, trials=500, max_steps_per_episode=1000, learn=False):
    """The play function runs iterations and updates Q-values if desired."""
    reward_per_episode = [] # Initialise performance log

    for trial in range(trials): # Run trials
        cumulative_reward = 0 # Initialise values of each game
        step = 0
        game_over = False
        while step < max_steps_per_episode and game_over != True: # Run until max steps
or until game is finished
            old_state = environment.current_location
            action = agent.choose_action(environment.actions)
            reward = environment.make_step(action)
            new_state = environment.current_location

            if learn == True: # Update Q-values if learning is specified
                agent.learn(old_state, reward, new_state, action)

            cumulative_reward += reward
            step += 1

            if environment.check_state() == 'TERMINAL': # If game is in terminal state,
game over and start next trial
                environment.__init__()
                game_over = True

        reward_per_episode.append(cumulative_reward) # Append reward for current trial t
o performance log

    return reward_per_episode # Return performance log
```

In [ ]:

```
env = GridWorld()
agent = RandomAgent()

print("Current position of the agent =", env.current_location)
print(env.agent_on_map())
available_actions = env.get_available_actions()
print("Available actions =", available_actions)
chosen_action = agent.choose_action(available_actions)
print("Randomly chosen action =", chosen_action)
reward = env.make_step(chosen_action)
print("Reward obtained =", reward)
print("Current position of the agent =", env.current_location)
print(env.agent_on_map())
```

```
Current position of the agent = (4, 0)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
```

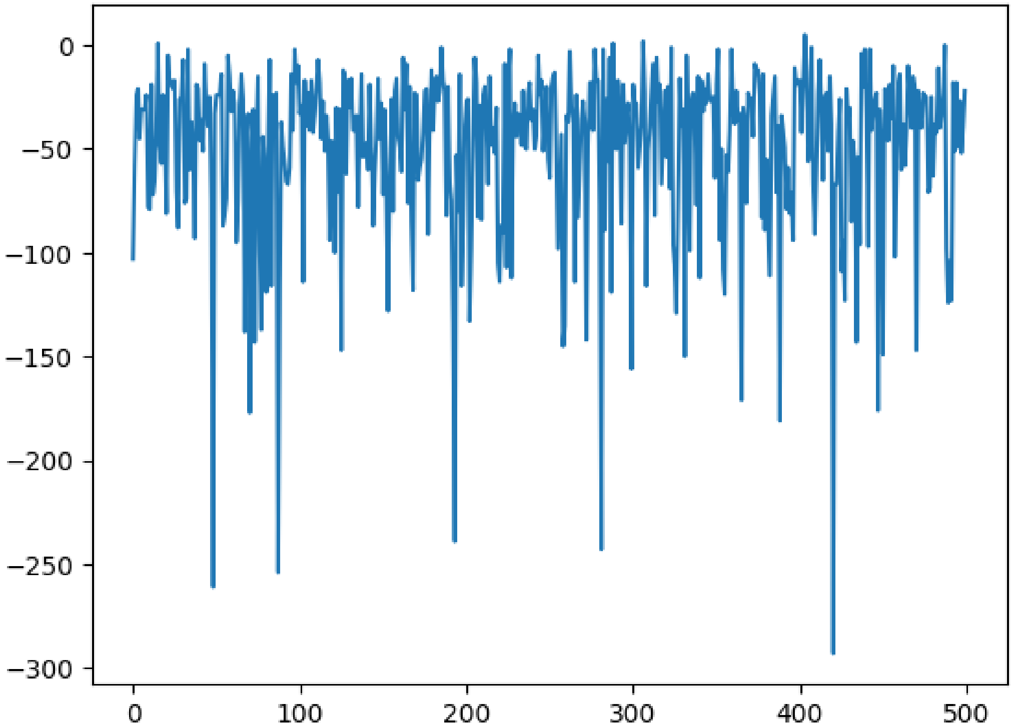
```
 [1. 0. 0. 0. 0.]]
Available_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
Randomly chosen action = UP
Reward obtained = -1.0
Current position of the agent = (3, 0)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

In [ ]:

```
# Initialize environment and agent
environment = GridWorld()
random_agent = RandomAgent()

reward_per_episode = play(environment, random_agent, trials=500)

# Simple learning curve
plt.plot(reward_per_episode)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7fd9841995b0>]
```

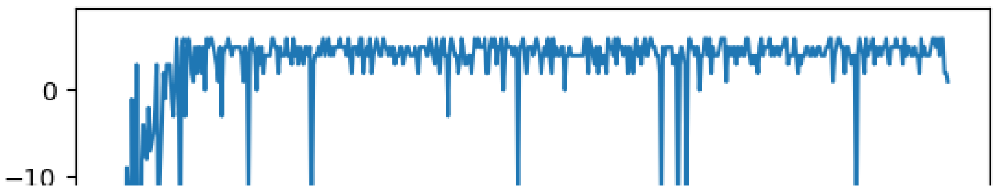


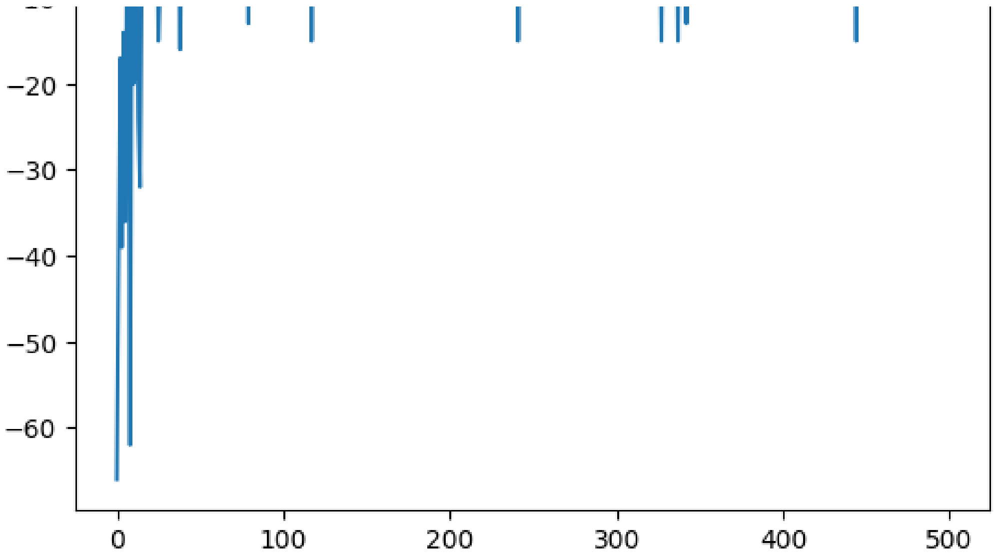
In [ ]:

```
environment = GridWorld()
agentQ = Q_Agent(environment)

# Note the learn=True argument!
reward_per_episode = play(environment, agentQ, trials=500, learn=True)

# Simple learning curve
plt.plot(reward_per_episode)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7fd98409cb80>]
```





In [ ]:

```
def pretty(d, indent=0):
    for key, value in d.items():
        print('\t' * indent + str(key))
        if isinstance(value, dict):
            pretty(value, indent+1)
        else:
            print('\t' * (indent+1) + str(value))


pretty(agentQ.q_table)
```

```
(0, 0)
 UP
  -0.30000000000000004
 DOWN
  -0.32801
 LEFT
  -0.30000000000000004
 RIGHT
  -0.24470569000000003
(0, 1)
 UP
  -0.1
 DOWN
  -0.11000000000000001
 LEFT
  -0.1
 RIGHT
  4.966569362650776
(0, 2)
 UP
  5.04778493723522
 DOWN
  3.222830533165051
 LEFT
  1.0597648529877604
 RIGHT
  9.999999999999995
(0, 3)
 UP
  0
 DOWN
  0
 LEFT
  0
 RIGHT
  0
(0, 4)
 UP
  -0.1
 DOWN
```

```
 DOWN
  -0.1
 LEFT
  4.68559
 RIGHT
  -0.1
(1, 0)
 UP
  -0.5331710000000001
 DOWN
  -0.5863922100000001
 LEFT
  -0.5000000000000001
 RIGHT
  -0.4580881154092101
(1, 1)
 UP
  -0.2629
 DOWN
  -0.3294807
 LEFT
  -0.30810000000000004
 RIGHT
  4.624162710093041
(1, 2)
 UP
  8.999999999999982
 DOWN
  3.2209220863047547
 LEFT
  0.943164529877979
 RIGHT
  -6.12579511
(1, 3)
 UP
  0
 DOWN
  0
 LEFT
  0
 RIGHT
  0
(1, 4)
 UP
  0.6209469000000001
 DOWN
  -0.22800000000000004
 LEFT
  -1.0
 RIGHT
  -0.2
(2, 0)
 UP
  -0.830459111
 DOWN
  -0.9405838065889002
 LEFT
  -0.9000000000000004
 RIGHT
  -0.8172038988892565
(2, 1)
 UP
  -0.594578
 DOWN
  -0.42619088194539334
 LEFT
  -0.5991900000000001
 RIGHT
  2.924340103425687
(2, 2)
 UP
  7.999999999999978
 DOWN
```

```
 2.5605317777155023
LEFT
 -0.1301817136962649
RIGHT
 -0.2119340849715119
(2, 3)
UP
 -2.71
DOWN
 -0.56590452
LEFT
 2.5347111091216044
RIGHT
 -0.5840458000000001
(2, 4)
UP
 -0.609088321
DOWN
 -0.7147164782936901
LEFT
 -0.6453092159486822
RIGHT
 -0.7000000000000002
(3, 0)
UP
 -1.2554412854561792
DOWN
 -1.3193254070836904
LEFT
 -1.2988908100000005
RIGHT
 -1.010951622960419
(3, 1)
UP
 -0.9957038246244667
DOWN
 -1.0172985262510004
LEFT
 -1.1271079692492323
RIGHT
 4.153335534709361
(3, 2)
UP
 6.9999999999999725
DOWN
 2.3203255890058925
LEFT
 -0.16459308439539613
RIGHT
 2.468888728605326
(3, 3)
UP
 -0.911348061074724
DOWN
 -0.4006625517967772
LEFT
 5.999999847039558
RIGHT
 -0.8894672955152446
(3, 4)
UP
 -1.1052780330788894
DOWN
 -1.1656104054385736
LEFT
 1.4028050750228915
RIGHT
 -1.1000000000000005
(4, 0)
UP
 -1.8001472739018642
DOWN
```

```
 DOWN
  -0.8888236115023643
 LEFT
  -1.7970910000000009
 RIGHT
  3.998932814292887
(4, 1)
 UP
  -0.18287118816545522
 DOWN
  -0.6088863428351093
 LEFT
  -1.2113682846931946
 RIGHT
  4.999999896242859
(4, 2)
 UP
  5.999999999996087
 DOWN
  0.21548925167439897
 LEFT
  0.6744480329153673
 RIGHT
  -0.4753800313463207
(4, 3)
 UP
  4.999995228505165
 DOWN
  -0.8741135925627628
 LEFT
  1.2330673870866098
 RIGHT
  -1.3088987416564297
(4, 4)
 UP
  -1.4561131628986512
 DOWN
  -1.6072073416883088
 LEFT
  3.9949496699897127
 RIGHT
  -0.7369232157860539
```

In [ ]:

```

```