Assignment 2 – Q Learning

CSE4007 2021-2학기 (13347)

기계공학부 ********** 손상현

1. 코드 설명

Python 파일 최상단부에는 bonus reward, decay factor가 상수로 선언되어 있습니다. 다음 개발의 편의를 위해 각 지점 키워드 (S, G, T, B P)에 대응하는 정수형 id 및 reward 값들과, 상하좌우 이동에 대응하는 q-table index를 dictionary 자료형 형태로 선언해 두었습니다.

```
5   BONUS = 1
6   DECAY = 0.9
7
8   MAP_DICT = { "S": 1, "G": 2, "T": 9, "B": -1, "P": 0 }
9   INV_MAP_DICT = { 1: "S", 2: "G", 9: "T", -1: "B", 0: "P" }
10  REWARD_DICT = { "S": 0, "G": 100, "T": BONUS, "B": -100, "P": 0 }
11  DIR_DICT = { "R": 0, "B": 1, "L": 2, "T": 3 }
12  INV_DIR_DICT = { 0:"R", 1:"B", 2:"L", 3:"T"}
```

A. 실행 구문 & 입출력 함수

```
197 MAP = read_input()
198 qlearner = QLearner(MAP, DECAY)
199 qlearner.train()
200 qlearner.define_policy()
201 path = qlearner.get_optimum_path()
202 val = qlearner.get_start_qvalue()
203 write_output(path, val)
204 #write_qtable_output(qlearner.q_table)
```

**********_assignment_2.py 파일에서 실행되는 main 구문입니다. `read_input` 함수는 `MAP`을 읽어들이고, `MAP`과 위에서 선언한 `DECAY` 값으로 `QLearner` 클래스 인스턴스를 생성합니다.

`qlearner`는 `train`, `define_policy`를 순차적으로 실행해야 하며, 학습이 완료된 후 `get_optimum_path`, `get_start_qvalue` 함수를 호출해 경로와 시작위치에서의 qvalue를 얻어옵니다. 이후 `write_output` 함수는 `output.txt` 파일로 지정된 양식을 출력합니다.
`write_qtable_output` 함수는 q-table을 직접 조사할 때 사용합니다.

* **read_input**

```python
def read_input():
    with open('./input.txt', 'r') as f:
        lines = f.read().splitlines()
        map_ = []

        for line in lines:
            map_row = []
            for c in line:
                map_row.append(MAP_DICT[c])
            map_.append(map_row)

    return map_
```

`input.txt`로부터 맵 텍스트를 읽어 들여 각 키워드에 대응되는 정수형 id를 원소로 가지는 2차원 list 지도로 분해 후, 반환합니다.

* **write_output**

```python
def write_output(path, qval):
    f_name = "output.txt"
    with open(f_name, 'w') as f:
        for v in path:
            f.write(f"{v} ")
        f.write("\n")
        f.write(f"{qval}")
```

학습이 모두 끝난 후 구한 path와 시작점의 최대 q-value를 같은 경로의 `output.txt` 파일로 출력합니다.

* **write_qtable_output**

```python
def write_qtable_output(q_table):
    f_name = "qtable_output.txt"
    with open(f_name, 'w') as f:
        for i in range(5):
            for k in range(4):
                f.write(f"{INV_DIR_DICT[k]}) ")
                for j in range(5):
                    f.write(f"{q_table[i][j][k]:-20.10f}")
                f.write("\n")
            f.write("\n")
```

q-table의 값을 직접 조사할 때 사용하는 함수입니다. `q_table` 값을 입력 받아 같은 경로의 `qtable_output.txt`로 출력합니다. 제출되는 코드에는 주석으로 실행부분이 비활성화 되어 있으며, 마지막 결과분석에 함수를 사용하였습니다.

B. QLearner 클래스

* __init__

```python
class QLearner:

    def __init__(self, MAP, DECAY):
        self.DECAY = DECAY
        self.MAP = MAP

        # init starting point
        for i, row in enumerate(MAP):
            if MAP_DICT['S'] in row:
                self.START = (i, row.index(MAP_DICT['S']))

        self.q_table = [[[0, 0, 0, 0] for j in range(5)] for i in range(5)] # i x j x 4 size q-table
        self.reward_table = [[REWARD_DICT[INV_MAP_DICT[MAP[i][j]]] for j in range(5)] for i in range(5)]
        self.policy_table = [[-1 for j in range(5)] for i in range(5)]
```

`QLearner` 클래스의 생성자입니다. 클래스 인스턴스의 생성시, decay factor와 map을 입력 받아 필드로 저장합니다. 그리고 map에서 시작지점을 찾아 `START` 필드로 초기화한후, 5x5x4 크기의 `q_table`을 0으로 초기화, map에 대응하는 `reward_table` 값을 초기화, 그리고 학습이 끝나고 작성될 `policy_table`을 -1으로 초기화 합니다.

* train

```python
    def train(self):
        # repeat until training satisfies termination condition
        for iter in range(1000):
            # start training at random point in the map
            state = [randrange(0, 5), randrange(0, 5)]

            # training iteration
            while not self._isBomb(state) and not self._isGoal(state):
                # 이동 가능한 방향중 랜덤 선택.
                next_state, action = self._random_walk(state)
                i, j = state
                i_, j_ = next_state
                # q-table update
                self.q_table[i][j][DIR_DICT[action]] = self.reward_table[i_][j_] + self.DECAY*self._delayed_reward(next_state)

                # visualization in terminal
                self._visualize(state, iter, action)

                # 폭탄, 골을 만나면 다음 while loop시, q-table을 update하지 않고 종료
                state = next_state

            print("Iteration Terminated!")
```

`train` 함수에서는 1000번의 학습 iteration을 실행합니다. 1000은 수렴을 위해 임의로 설정한 횟수입니다. 각 iteration은 지도상의 랜덤한 위치에서 시작하여, 폭탄, goal에 도달하면 종료됩니다. 학습과정에서는 `_random_walk` 함수를 호출해서 위치를 랜덤한 방향으로 움직이며, 매 이동시, q-learning 공식에 따라 q-table을 매번 업데이트합니다. Q-learning의 공식은 다음과 같습니다.

$$\mathrm{Q}(s,a) = \mathrm{r}(s,a) + \gamma \max_{a'} Q(s',a')$$

여기서 $r(s,a)$부분은 `next_state` 위치에서의 reward를 대입하는 것으로 간략화 할 수 있었습니다. $\max_{a'} Q(s',a')$ 부분은 `next_state`에서 가능한 action중 가장 큰 q_value를 반환하도록 함수 `_delayed_reward`를 선언하여 구할 수 있도록 하였습니다.

학습이 끝나면 따로 값을 반환하지 않고 클래스가 학습된 q-table을 저장합니다.

* **define_policy**

```
85  def define_policy(self):
86      for i in range(5):
87          for j in range(5):
88              state = [i, j]
89              # no policy for bomb and goal point
90              if not self._isBomb(state) and not self._isGoal(state):
91                  # find optimum direction
92                  optimum_val = 0
93                  optimum_dir = -1
94                  for k in range(4):
95                      if self.q_table[i][j][k] > optimum_val:
96                          optimum_val = self.q_table[i][j][k]
97                          optimum_dir = INV_DIR_DICT[k]
98                  # if max value is 0, no policy is established for that cell. (-1)
99                  self.policy_table[i][j] = optimum_dir
```

Q-table의 학습이 끝난 이후 호출하는 함수입니다. 종료지점인 bomb, goal을 제외한 지도상의 모든 각 위치에서 q-table 값이 가장 높은 action을 policy로 지정합니다. 이때, q-value가 동일한 두 개 이상의 action이 있을 경우(복수 경로가 가능할 경우), 위의 dictionary에 선언된 순서에 따라서, Right>Bottom>Left>Top 순으로 이동의 우선순위를 주었습니다.

모든 action의 qvalue가 0 이하인 위치에서는 (학습이 되지 않았거나, goal, bonus에 연결되지 않음) policy가 지정되지 않고, -1로 표시됩니다.

정의가 끝나면 따로 값을 반환하지 않고 클래스가 policy table을 저장합니다.

* **get_optimum_path**

```
100 def get_optimum_path(self):
101     i, j = self.START
102     path = [i*5+j]
103
104     while self.MAP[i][j] != MAP_DICT["G"]:
105         policy = self.policy_table[i][j]
106         i, j = self._move_state_action([i, j], policy)
107         path.append(i*5+j)
108
109     return path
```

Policy의 정의가 완료된 후 호출하는 함수입니다. 시작 위치로부터 policy를 따라 경로를 추적합니다. goal위치에 도달하면 지나온 위치들을 list로 반환합니다. 단, 학습이 충분히 이루어지지 않아 policy가 없는 곳을 지나거나, reward 세팅에 의해 goal 위치가 나오기 전에 policy의 루프 구간이 있는 경우, 함수가 무한 루프에 빠지거나 오류가 날 수 있습니다.

* **get_start_qvalue**

```
112 def get_start_qvalue(self):
113     i, j = self.START
114     max_q_val = max(self.q_table[i][j])
115
116     return max_q_val
```

학습이 끝난 이후 호출되는 함수입니다. 시작 지점에서의 action q_value중 가장 큰 값을 찾아 반환합니다. 시작 위치는 클래스 인스턴스가 생성될 때 START 필드로 저장하고 있습니다.

* **_random_walk**

```
118     def _random_walk(self, state):
119         possible_action = self._get_possible_action(state)
120         walk = randrange(0, len(possible_action))
121         action = possible_action[walk]
122         next_state = self._move_state_action(state, action)
123 
124         return next_state, action
```

학습과정에서 랜덤한 방향으로 위치를 움직이며 학습을 진행하는 과정을 구현한 함수입니다. 현재 위치 `state`를 입력 받아, 상하좌우 중 맵 밖으로 나가지 않는 `possible_action` 중 한 개를 랜덤으로 선택해서 `next_state`로 이동합니다. 이동한 다음 상태인 `next_state`와 이동에 사용된 `action`을 반환합니다.

가능한 경우의 수 `possible_action`과 이동 후의 `next_state`를 구하는 과정은 다음 설명될 `_get_possible_action`과 `_move_state_action` 함수로 분리하였습니다.

* **_get_possible_action**

```
126     def _get_possible_action(self, state):
127         possible_action = ["R", "B", "L", "T"]
128         if state[0] == 0: # i index minimum
129             possible_action.remove("T")
130         if state[0] == 4: # i index maximum
131             possible_action.remove("B")
132         if state[1] == 0: # j index minimum
133             possible_action.remove("L")
134         if state[1] == 4: # j index maximum
135             possible_action.remove("R")
136 
137         return possible_action
```

상하좌우 이동중 맵 밖으로 나가지 않는 유효한 action들을 구하는 함수입니다. 현재 `state` 위치 입력 받아, 위치가 맵 index의 양 끝단에 도달해 있는 경우, 가능한 action 리스트에서 밖으로 나가는 action을 제거합니다. 가능한 action들을 담고 있는 list를 반환합니다.

* **_move_state_action**

```
139     def _move_state_action(self, state, action):
140         next_state = list(state) #copy
141         if action == "R":
142             next_state[1] += 1 # move j+1
143         elif action == "B":
144             next_state[0] += 1 # move i+1
145         elif action == "L":
146             next_state[1] -= 1 # move j-1
147         elif action == "T":
148             next_state[0] -= 1 # move i-1
149 
150         return next_state
```

현재 위치 `state`와 `action`을 입력 받아 `action`을 해석하고 다음 위치로 이동해 `next_state`를 구하는 함수입니다. 다음 위치 `next_state`를 반환합니다.

* **_delayed_reward**

```
152     def _delayed_reward(self, next_state):
153         possible_action = self._get_possible_action(next_state)
154 
155         max = 0
156         for action in possible_action:
157             i_, j_ = next_state
158             reward = self.q_table[i_][j_][DIR_DICT[action]]
159             if max < reward:
160                 max = reward
161 
162         return max
```

학습과정의 q-value 업데이트 공식 중 $\max_{a'} Q(s', a')$ 부분을 따로 분리한 함수입니다. 다음 위치(s') `next_state`를 입력 받고, 다음 위치에서 가능한 action중 현재 `q_table`에서 가장 큰 q-value를 찾아 반환합니다.

가능한 action이라 함은 맵 밖으로 나가지 않는 action을 뜻합니다. `_get_possible_action` 함수가 여기서도 사용됩니다.

* **_isBomb, _isGoal**

```
164     def _isBomb(self, state):
165         if self.MAP[state[0]][state[1]] == MAP_DICT["B"]:
166             return True
167         else:
168             return False
169 
170     def _isGoal(self, state):
171         if self.MAP[state[0]][state[1]] == MAP_DICT["G"]:
172             return True
173         else:
174             return False
```

현재 위치 `state`를 입력 받아, `MAP`에서 그 위치가 bomb, goal에 해당하는지 해석하고, 해당하는 경우 `True` 아니면 `False`를 반환합니다. `train` 함수에서 학습 iteration `while` 루프가 termination 컨디션을 판단할 때 호출합니다.

* **_visualize**

```
176     def _visualize(self, state, iter, action):
177         i_, j_ = state
178 
179         os.system('cls')
180         print(f'{iter}) Q-Table update: [{i_}][{j_}][{action}] = {self.q_table[i_][j_][DIR_DICT[action]]:15.10f}')
181 
182         for i in range(5):
183             for j in range(5):
184                 if state == [i, j]:
185                     print("◎", end='')
186                 else:
187                     if self.MAP[i][j] == MAP_DICT["S"]:
188                         print("S", end="")
189                     elif self.MAP[i][j] == MAP_DICT["G"]:
190                         print("G", end='')
191                     elif self.MAP[i][j] == MAP_DICT["T"]:
192                         print("★", end='')
193                     elif self.MAP[i][j] == MAP_DICT["B"]:
194                         print("※", end='')
195                     elif self.MAP[i][j] == MAP_DICT["P"]:
196                         print(" ", end='')
197                 print('|', end='')
198             print("")
```

학습과정에서 지도에서 현재 위치를 작성하고 q-table이 업데이트되는 것을 terminal에 출력합니다. 제출되는 코드에서는 비활성화되어 있습니다.

2. 함수 설명

모든 함수들은 **********_assignment_1.py 파일 한곳에 작성되었습니다.

A. 입출력 함수

함수 명	리턴 값	설명
read_input()	map (2d list)	input.txt로부터 맵 텍스트를 읽어 들여 각 키워드에 대응되는 정수형 id를 원소로 가지는 2차원 list 지도로 분해 후, 반환합니다.
write_output(path, qval)	none	강화학습으로 생성된 path와 시작점에서 가능한 action의 최대 q값을 입력 받아 output.txt 파일로 작성합니다.
write_qtable_output(q_table)	none	실행 결과를 분석하고, 디버깅을 위해 학습된 q_table을 qtable_output.txt파일로 내보냅니다. 제출된 파일에는 실행부분이 주석으로 제외되어 있습니다.

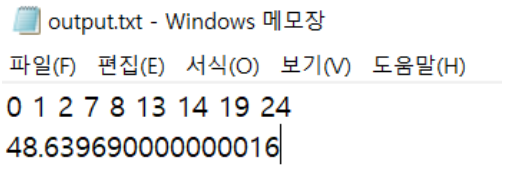
B. QLearner 클래스

함수 명	리턴 값	설명
__init__(self, MAP, DECAY)	none	QLearner 클래스의 생성자입니다. Decay factor와 map을 입력 받아 필드로 저장합니다. START 지점, q_table, reward_table, policy_table을 초기화 합니다. 자세한 알고리즘은 위의 코드설명에서 서술합니다.
train()	none	train 함수에서는 1000번의 학습 iteration을 실행합니다. 각 iteration은 폭탄, goal에 도달하면 종료됩니다. 학습과정에서는 랜덤한 방향으로 움직이며 q-learning 공식에 따라 q-table을 매번 업데이트합니다. 자세한 알고리즘은 위의 코드설명에서 서술합니다.
define_policy()	none	학습이 끝난 후 폭탄, goal/bomb 위치를 제외한 맵의 모든 위치에서 q-table을 바탕으로 가장 q-value 값이 높은 action으로 policy를 지정합니다. 자세한 알고리즘은 위의 코드설명에서 서술합니다.
get_optimum_path()	path (1d list)	시작 위치로부터 policy를 따라 경로를 추적합니다. goal위치에 도달하면 지나온 위치들을 list로 반환합니다. 만약 reward 세팅에 의해 goal 위치가 나오기 전에 policy의 루프 구간이 있는 경우, 함수가 무한

		루프에 빠질 수 있습니다. 자세한 내용은 실험결과 분석에서 서술합니다.
get_start_qvalue()	max_q_val (int)	시작지점에서의 action q_value중 가장 큰 값을 찾아 반환합니다.
_random_walk(state)	next_state (list), action (int)	입력 받은 state 위치에서 맵 밖으로 나가지 않고 이동 가능한 다음 위치를 랜덤하게 선택합니다. 랜덤하게 이동한 다음 위치와, 이동을 위해 현재 위치에서 행한 action을 반환합니다.
_get_possible_action(state)	possible_action (list)	입력 받은 위치에서 맵 밖으로 나가지 않고 이동가능한 action들을 구하는 함수입니다. 이동가능한 방향들은 list로 반환됩니다.
_move_state_action(state, action)	next_state (list)	현재 위치에서 지정된 action을 따라 다음 위치를 구하는 함수입니다. 다음 상태를 반환합니다.
_delayed_reward(next_state)	max (int)	q-learning의 q-table 업데이트 공식에서, delayed reward에 해당하는 항입니다. 현재 위치에서 action을 취한 다음 위치(s')을 입력 받아, 다음 위치의 여러 action에서 max q값을 반환합니다.
_isBomb(state)	Boolean	현재 위치가 bomb 위치이면 true, 아니면 false를 반환합니다.
_isGoal(state)	Boolean	현재 위치가 goal 위치이면 true, 아니면 false를 반환합니다.
_visualize(state)	none	학습과정을 terminal에서 시각화 하는 함수입니다. 지도와 현재 위치를 terminal에 출력합니다.

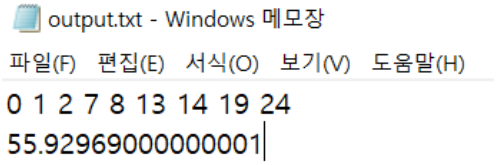
3. 실험 결과 및 분석

A. Bonus reward=1, γ=0.9



(Goal 도달)

B. Bonus reward=10, γ=0.9



(Goal 도달)

Bonus reward가 1, 10인 경우에는 모두 적절한 경로를 따라 goal에 도착함을 알 수 있었습니다. 다만 bonus reward가 커짐에 따라 start 위치에서 최대 q-value가 커지는 것을 확인할 수 있었습니다. Q값의 전파과정을 확인하기 위해 아래처럼 **0~9** 위치에서 q-table을 출력하였습니다.

이 결과 8번 위치에서 max-q 값은 아래 방향인 72.9로 동일하나, 처음 지나는 bonus 지점인 7번 위치에서의 reward가 start 위치로 전파되는 것을 알 수 있었습니다. 두 경우 모두 17번 위치의 bonus 포인트는 지나지 않으므로, 8번 위치 이후에서 전파되어오는 max q 값은 goal reward에 의해서만 전파됨을 확인할 수 있었습니다.

또한, 4 방향에서 동일 Q값이 있는 경우 policy 지정 우선순위에 따라서 0 1 2 7 8 13 18 23 24 같은 복수의 경로 역시 성립 가능했습니다.

```
R)      48.6396900000      54.0441000000    -100.0000000000       0.0000000000       0.0000000000
B)    -100.0000000000      54.0441000000      60.0490000000       0.0000000000    -100.0000000000
L)       0.0000000000      43.7757210000      48.6396900000       0.0000000000    -100.0000000000
T)       0.0000000000       0.0000000000       0.0000000000       0.0000000000       0.0000000000

R)       0.0000000000      60.0490000000      65.6100000000    -100.0000000000       0.0000000000
B)       0.0000000000      48.6396900000    -100.0000000000      72.9000000000       0.0000000000
L)       0.0000000000    -100.0000000000      54.0441000000      60.0490000000       0.0000000000
T)       0.0000000000      48.6396900000      54.0441000000    -100.0000000000       0.0000000000
```

```
R)      55.9296900000      62.1441000000    -100.0000000000       0.0000000000       0.0000000000
B)    -100.0000000000      62.1441000000      69.0490000000       0.0000000000    -100.0000000000
L)       0.0000000000      50.3367210000      55.9296900000       0.0000000000    -100.0000000000
T)       0.0000000000       0.0000000000       0.0000000000       0.0000000000       0.0000000000

R)       0.0000000000      69.0490000000      65.6100000000    -100.0000000000       0.0000000000
B)       0.0000000000      55.9296900000    -100.0000000000      72.9000000000       0.0000000000
L)       0.0000000000    -100.0000000000      62.1441000000      69.0490000000       0.0000000000
T)       0.0000000000      55.9296900000      62.1441000000    -100.0000000000       0.0000000000
```

C. Bonus reward=20, γ=0.9

Goal에 도달하지 못하고 무한 루프에 빠집니다. 이때 Q-table의 값을 자세히 보면, 0 -> 1 -> 2 -> 7 -> 8 까지의 경로는 같으나, 이후 경로를 보면, 7 -> 8 -> 7 -> 8 … 으로 7, 8 구간에서 무한루프가 발생함을 볼 수 있습니다. 즉, 중간 bonus reward의 값이 너무 큰 경우, goal에 의해서 전파된 reward보다 더 영향력이 커짐으로 인해서 goal에 도달하지 못하고, local 해에 수렴해버리는 경우가 발생함을 알 수 있습니다. 즉 적절한 reward의 설정이 필요합니다.