지능시스템소프트웨어 Term Project 설명 문서

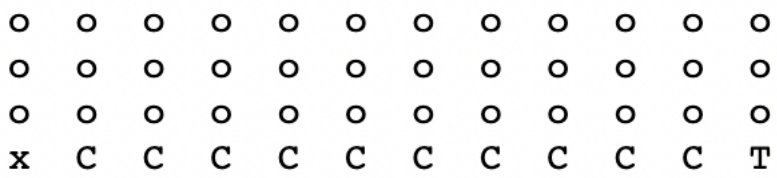
1. 사용한 환경 버전

기존 실습 환경에 추가로 설치가 필요한 패키지가 없습니다. 실습에서 사용한 docker image의 기본 환경을 이용하였으나 colab 환경에서도 아무 문제없이 실행됨을 확인하였습니다.

2. openAI gym 환경(cliffwalking) 소스코드

open AI gym environment 중에서 cliffwalking 환경을 선택하였습니다. 다음 github 주소에서 환경의 전체 소스코드를 확인하실 수 있습니다.

https://github.com/openai/gym/blob/master/gym/envs/toy_text/cliffwalking.py



cliffwalking 환경은 4*12 배열에서 (3,0) 지점을 시작점, (3,11) 지점을 도착점으로 합니다. 시작점과 도착점을 제외한 4번째 열은 모두 cliff입니다. Cliff를 밟게 된다면 시작점으로 돌아가며 목적지에 도착하면 episode가 종료됩니다.

UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

환경은 위와 같이 구성되어 있습니다. 한 스텝이 지날 때마다 reward는 -1씩 더해지며 cliff를 밟게 되면 -100의 reward를 더합니다. 최종 목표는 reward를 가장 크게 하면서 즉, 시작점에서 도착점까지 가장 빠르게 가는 루트를 찾는 것입니다.

```
### Description
The board is a 4x12 matrix, with (using NumPy matrix indexing):
- [3, 0] as the start at bottom-left
- [3, 11] as the goal at bottom-right
- [3, 1..10] as the cliff at bottom-center

If the agent steps on the cliff it returns to the start.
An episode terminates when the agent reaches the goal.

### Actions
There are 4 discrete deterministic actions:
- 0: move up
- 1: move right
- 2: move down
- 3: move left

### Observations
There are 3x12 + 1 possible states. In fact, the agent cannot be at the cliff, nor at the goal
(as this results the end of episode). They remain all the positions of the first 3 rows plus the bottom-left cell.
The observation is simply the current position encoded as
[flattened index](https://numpy.org/doc/stable/reference/generated/numpy.unravel_index.html).

### Reward
Each time step incurs -1 reward, and stepping into the cliff incurs -100 reward.
```

1) init 함수

```
def __init__(self):
    self.shape = (4, 12)
    self.start_state_index = np.ravel_multi_index((3, 0), self.shape)

    self.nS = np.prod(self.shape)
    self.nA = 4

    # Cliff Location
    self._cliff = np.zeros(self.shape, dtype=bool)
    self._cliff[3, 1:-1] = True

    # Calculate transition probabilities and rewards
    self.P = {}
    for s in range(self.nS):
        position = np.unravel_index(s, self.shape)
        self.P[s] = {a: [] for a in range(self.nA)}
        self.P[s][UP] = self._calculate_transition_prob(position, [-1, 0])
        self.P[s][RIGHT] = self._calculate_transition_prob(position, [0, 1])
        self.P[s][DOWN] = self._calculate_transition_prob(position, [1, 0])
        self.P[s][LEFT] = self._calculate_transition_prob(position, [0, -1])

    # Calculate initial state distribution
    # We always start in state (3, 0)
    self.initial_state_distrib = np.zeros(self.nS)
    self.initial_state_distrib[self.start_state_index] = 1.0

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

환경을 초기화하는 함수입니다. 환경이 4*12 배열로 이루어져 있기 때문에 shape을 (4,12)로 정의합니다. 시작점이 (3,0)로 고정되어 있어 시작점의 좌표를 (3,0)로 저장합니다. state 개수가 4*12로 48이므로 현재 배열을 product하여 state 개수를 nS에 저장합니다. 그리고 action의 개수는 상하좌우로 움직일 수 있어 4개이므로 4를 action 개수 nA에 저장합니다.

_cliff는 cliff를 1, 아닌 곳을 0으로 저장한 배열입니다.

P는 transition probability를 저장하는 딕셔너리입니다. 모든 state를 돌면서 각 action을 선택했을 때의 transition probability를 저장합니다.

Initial_state_distribution은 state 개수만큼 0으로 초기화하며 시작점은 1.0으로 초기화합니다.

Observation_space는 state 개수만큼의 이산적인 값들로 정의합니다. action_space는 action 개수만큼의 이산적인 값들로 정의합니다.

2) _limit_coordinates 함수

```
def _limit_coordinates(self, coord: np.ndarray) -> np.ndarray:
    """Prevent the agent from falling out of the grid world."""
    coord[0] = min(coord[0], self.shape[0] - 1)
    coord[0] = max(coord[0], 0)
    coord[1] = min(coord[1], self.shape[1] - 1)
    coord[1] = max(coord[1], 0)
    return coord
```

agent가 정해진 범위를 넘어가지 않게 조정하는 함수입니다. 4*12 배열로 3과 현재 y좌표(행 값)을 비교하여 더 작은 값을 y좌표로 바꿉니다. 만약 3을 넘어가는 값이라면 3이 선택되어 최대 3의 값을 가지며 3보다 작은 값이라면 현재 값을 가집니다. 0과 현재 y좌표를 비교하여 만약 0보다 작은 값이라면 0의 값을 가지며 0보다 큰 값이라면 현재 값을 가집니다. x좌표도 y좌표와 동일하게 동작합니다. 이렇게 바뀐 좌표를 반환합니다.

3) _calculate_transition_prob 함수

```
def _calculate_transition_prob(self, current, delta):
    """Determine the outcome for an action. Transition Prob is always 1.0.

    Args:
        current: Current position on the grid as (row, col)
        delta: Change in position for transition

    Returns:
        Tuple of ``(1.0, new_state, reward, done)``
    """
    new_position = np.array(current) + np.array(delta)
    new_position = self._limit_coordinates(new_position).astype(int)
    new_state = np.ravel_multi_index(tuple(new_position), self.shape)
    if self._cliff[tuple(new_position)]:
        return [(1.0, self.start_state_index, -100, False)]

    terminal_state = (self.shape[0] - 1, self.shape[1] - 1)
    is_done = tuple(new_position) == terminal_state
    return [(1.0, new_state, -1, is_done)]
```

transition probability를 계산하는 함수입니다. 리턴값은 (1.0, new_state, reward, done)입니다. new_position은 current와 delta를 더한 값인데 current는 (row, col)의 현재 위치, delta는 각 action마다 해당 방향으로 1씩 이동하는 좌표 변화입니다. 즉, 현재 위치에 변화된 위치를 더해 새로운 위치를 정합니다. 이 위치에 _limit_coordinates 함수를 실행하여 주어진 배열 안으로 좌표를 제한합니다. new_state는 새로운 위치를 인덱스로 하여 shape을 저장합니다. 만약 이 새로운 위치가 _cliff로 정의한 위치에 도달한다면 cliff를 밟았기 때문에 -100의 reward를 가지며 시작점이 다음 state가 됩니다. 그리고 도착점에 아직 도달하지 않아 False를 리턴합니다. transition probability는 항상 1.0이므로 (1.0, self.start_state_index, -100, False)을 리턴합니다.

terminal_state는 4*12 배열에서 (3,11) 위치이므로, (shape[0]-1, shape[1]-1)의 값으로 정의합니다. is_done은 새로운 위치가 terminal_state이면 1을, 아니면 0을 가집니다. cliff를 밟고 있지 않은 경우 이동할 때마다 -1의 reward를 가지며 다음 state가 new_state가 됩니다. 따라서 (1.0, new_state, -1, is_done)이 리턴됩니다.

4) step 함수

```
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})
```

timestep마다 수행하는 함수입니다. transitions는 현재 state에서 action을 취했을 때의 transition probability 값을 저장합니다. i는 모든 transitions의 transition probability 값을 넣어 categorical sample한 결과의 index를 저장합니다. p, s, r, d에는 i번째 transition 즉, categorical_sample한 transition의 값을 저장합니다. p는 transition probability, s는 next state, r은 reward, d는 done입니다. 현재 state는 next state로 lastaction은 현재 action으로 갱신합니다. 그리고 (state, return, done, transition probability)을 리턴합니다.

5) reset 함수

```
def reset(
    self,
    *,
    seed: Optional[int] = None,
    return_info: bool = False,
    options: Optional[dict] = None
):
    super().reset(seed=seed)
    self.s = categorical_sample(self.initial_state_distrib, self.np_random)
    self.lastaction = None
    if not return_info:
        return int(self.s)
    else:
        return int(self.s), {"prob": 1}
```

부모 클래스의 reset 함수를 실행합니다. 이 부모 클래스의 reset은 환경을 처음 state로 리셋하며 처음 observation을 리턴합니다. state에 initial_state_distrib로 categorical_sample한 결과를 저장합니다. 초기 distribution은 시작점만 1.0을 가지며 나머지는 모두 0의 값을 가지기 때문에 categorical_sample을 하게 되면 시작점을 리턴하게 된다. 즉, state를 시작점으로 만드는 부분이다. 리셋하게 되면 action은 아무것도 선택되지 않은 상태이므로 lastaction을 None으로 저장합니다. 그리고 만약 return_info가 False라면 state를 반환하고 true라면 state와 transition probability를 1로 하여 반환합니다. 기본적으로 return_info는 False 값을 갖기 때문에 따로 변수 값을 넣어주지 않는다면 state만, return_info 값을 정의해준 경우는 transition probability를 1로 설정하여 리턴해주는 것입니다.

6) render 함수

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

    for s in range(self.nS):
        position = np.unravel_index(s, self.shape)
        if self.s == s:
            output = " x "
        # Print terminal state
        elif position == (3, 11):
            output = " T "
        elif self._cliff[position]:
            output = " C "
        else:
            output = " o "

        if position[1] == 0:
            output = output.lstrip()
        if position[1] == self.shape[1] - 1:
            output = output.rstrip()
            output += "\n"

        outfile.write(output)
    outfile.write("\n")

    # No need to return anything for human
    if mode != "human":
        with closing(outfile):
            return outfile.getvalue()
```

환경을 불러와 렌더링하는 함수입니다. StringIO는 문자열을 파일처럼 처리하는 객체로 mode를 ansi로 넣어줄 경우 해당 부분을 실행하며 문자열을 파일로 저장하지만 그게 아니라면 표준출력으로 보여줍니다.

그리고 모든 state를 돌면서 position을 state로 인덱스 처리합니다. 현재 state와 변수로 들어온 state가 같다면 사람이 있는 곳을 의미하여 문자열에 x를 넣어줍니다. 그리고 position이 (3,11)이라면 도착점이므로 T를 넣어줍니다. 그리고 _cliff[position]이 1이라는 것은 cliff의 값이라는 의미로 C를 넣어줍니다. 그리고 나머지는 o를 넣어줍니다. 만약 x가 0이라면 제일 왼쪽 열이므로 lstrip으로 왼쪽 공백을 지워주며, shape[1] – 1이라면 제일 오른쪽 열이므로 rstrip으로 오른쪽 공백을 지우고 다음 행으로 넘겨줍니다. outfile에 문자열을 출력하고 for문을 모두 돈 후에 다음 행으로 넘겨주는 작업을 통해 이어서 출력하게 될 경우를 고려해줍니다.

그리고 만약 mode가 human이면 어떤 값도 리턴할 필요가 없지만 ansi면 mode에 다른 값을 넣어준 것입니다. outfile을 파일로 처리한 것으로 outfile의 getvalue 값을 리턴합니다.

3. 적용한 강화학습 알고리즘

1) 알고리즘 선택 과정

Cliffwalking 환경은 4*12 배열로 state의 개수가 48, action의 개수가 4개로 적은 개수의 state, action을 가집니다. 이걸 학습시키기 위해서는 agent가 매 state에서 최적의 결과를 내는 action을 선택해야만 합니다. 절벽을 밟게 되면 -100, 절벽이 아닌 곳으로 한 칸 이동하게 되면 -1의 리워드를 갖기 때문에 action을 선택할 때 가장 value가 높은 것을 선택하면 됩니다. Value는 현재의 reward만이 아니라 미래의 reward도 고려하기 때문에 최종 reward를 최대화하는 것을 목표로 학습해야 합니다.

위 방법만 보고 처음에는 MDP나 DP를 이용하여 해결하는 방법을 생각하였습니다. 하지만 Frozenlake를 구현했던 것과 다르게 위 환경에서는 initial_state_distrib가 처음부터 정해져 있는 것이 아니었습니다. 따라서 모든 모델을 dynamics를 알지 못한다는 것을 깨닫고 MDP를 이용하는 데 무리가 있다는 것을 깨달았습니다. Cliffwalking문제는 유한한 개수의 state와 action을 가지고 있지만 finite MDP가 아니기 때문에 DP를 이용하여 iterative한 방식도 어렵다는 것을 느끼고 MDP가 아닌 완전히 다른 알고리즘을 생각했습니다.

그 다음 떠올린 알고리즘은 MC 였습니다. 왜냐하면 모델의 모든 dynamics를 완벽히 알지 못하는 경우 사용할 수 있는 방법이 바로 MC 방법이라고 배웠기 때문입니다. 하지만 모델 MC로 every visit, first visit 모두 코드를 실행하여 확인한 결과, 해당 알고리즘으로는 절벽을 피해 가장 빠르게 도착한다는 목표와 달리 너무 많은 step을 필요로 하였습니다. MC 방식은 항상 최적의 결과를 내는 것이 아니었습니다. 이 방법을 해결하기 위해 에피소드의 길이를 더 길게 하여 학습시킬 수도 있지만 에피소드를 500번만 해도 꽤나 학습을 기다리는데 시간이 꽤나 소요되었으며 성능이 시간 대비 좋지 않았습니다. 그래서 MC 방식도 아닌 다른 알고리즘을 적용하고자 했습니다.

위의 조건을 다시 살펴보면서 모델의 dynamics를 모두 알지 못하더라도 학습이 가능하지만 episode의 길이가 너무 길거나 continuous task일 때 사용할 수 있는 알고리즘을 떠올렸습니다. 이 상황에서 가장 적합한 것은 TD 방식이었습니다. TD 방식은 non-episodic task에도 적용 가능하며 모델의 dynamics를 몰라도 적용 가능했기 때문에 앞서 해결하지 못한 부분을 분명 해결할 수 있을 것이라고 생각했습니다.

2) 선택 알고리즘 설명

Sarsa 방식과 Q learning 2가지의 TD 방식을 수업 중에 다뤘는데 이 두 방식의 차이를 직접 비교하기 위해 2가지 모두 코드를 구현하여 실행해봤습니다. TD 방식이란 step 별로 차이를 계속 업데이트하는 방식을 말합니다. 현재 step에서 얻은 reward R_{t+1}와 현재 step에서 action을 취해서 넘어간 다음 step에서 value $V(S_{t+1})$값을 추정하여 예측값을 구하고 실제 현재값을 예측값에서 빼서 값을 계속 업데이트하는 과정을 거칩니다. 이렇게 업데이트되는 값이 converge되면 업데이트를 그만하고 종료합니다. Prediction에서는 value function을 사용하지만 control에서는 Q function을 사용합니다. 앞으로 설명할 Sarsa와 Q-learning은 TD control 방식이며 따라서 Q value가 업데이트됩니다. 두 방식의 차이는 on-policy, off-policy로 실제로 행동하는 policy와 업데이트하는 policy가 서로 같은가 다른가의 차이입니다. Policy에 대한 자세한 설명은 아래에서 자세히 하도록 하겠습니다.

A) Sarsa

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Sarsa는 on-policy 방식의 TD control 방법 중 하나입니다. 모든 에피소드에 대해서 루프를 돌면서 현재 Q value를 기반으로 하여 현재의 state에서 action을 epsilon-greedy 방식으로 선택합니다. 그리고 매 step마다 루프를 도는데 action을 취해 reward와 next_state를 얻습니다. 그리고 다시 epsilon-greedy 방식으로 다음 timestep을 보아 next_state에서 next_action을 선택합니다. 그리고 Q value를 업데이트 하고 state와 action도 업데이트합니다. 이 방식이 on-policy인 이유는 agent가 실제로 학습하기 위한 policy가 같은 epsilon greedy 방식으로 선택이 되는 같은 policy이기 때문입니다.

B) Q-learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Q-learning은 off-policy 방식의 TD control 방법 중 하나입니다. 기본적인 것들은 Sarsa와 유사합니다. 같은 TD 방식이기 때문에 우선 모든 에피소드에 대해서 루프를 돌면서 매 step마다 다시 루프를 돕니다. 이때 현재 Q value를 기반으로 하여 현재의 state에서 action을 epsilon-greedy 방식으로 선택합니다. 그리고 도는데 action을 취해 reward와 next_state를 얻습니다. 구한 값들을 이용해 Q value를 업데이트하는데 Q value를 업데이트하는 부분에서 Sarsa와 큰 차이를 갖습니다. 이 Q value에서 실제로 업데이트 하는 부분을 보면 기존의 Q와는 다른 $\max_a$Q(next_state, a) 값을 가짐을 알 수 있습니다. Sarsa에서는 똑같이 epsilon-greedy를 사용하여 같은 policy를 이용해 Q 값을 업데이트 했지만, Q-learning은 실제로 행동을 하는 Q와 달리 업데이트를 하는 부분이 다릅니다. Q 값들 중 가장 큰 값을 가지는 action을 선택하는 방식으로 사실상 greedy 방식의 전혀 다른 policy를 사용합니다. 이렇게 실제로 행동하는 policy와 업데이트하는 policy가 다른 방식을 off-policy라고 합니다.

4. 소스코드 설명

1) Sarsa

```
In [1]: import random
        import gym
        %matplotlib inline
        env = gym.make('CliffWalking-v0')
```

필요한 라이브러리를 import 합니다. 그리고 사용할 openAI gym 환경을 가져옵니다. 사용할 환경은 CliffWalking-v0입니다.

```
In [2]: env.render()
        print(env.observation_space.n)
        print(env.action_space.n)
```

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  C  C  C  C  C  C  C  C  C  C  T

48
4
```

환경을 불러옵니다. 4*12 배열의 환경으로 좌상단이 (0,0)이라고 할 때, (3,0)는 시작점, (3,11)이 도착점, 시작점과 도착점을 제외한 모든 3행의 값은 절벽입니다. 나머지는 이동 가능한 부분입니다. 가능한 state의 개수는 4*12로 48, action의 개수는 상하좌우로 4입니다.

```
In [3]: Q = {}
        for s in range(env.observation_space.n):
            for a in range(env.action_space.n):
                Q[(s,a)] = 0.0
```

Sarsa 방식은 TD 방식이므로 state-action value인 Q function을 이용합니다. 이 Q값을 담을 Q를 정의해줍니다. 그리고 모든 state와 action 조합에 대해 0.0 값을 넣어줌으로 (state, action)의 Q값을 초기화하는 과정을 거칩니다.

```
In [4]: def epsilon_greedy(state, epsilon):
            if random.uniform(0,1) < epsilon:
                return env.action_space.sample()
            else:
                return max(list(range(env.action_space.n)), key = lambda x: Q[(state,x)])
```

위의 알고리즘 설명에서 보았듯 실제 Q값과 업데이트하는 Q값을 다른 policy를 사용하는데epsilon greedy 방식을 이용하여 실제 Q값을 결정하기 때문에 epsilon greedy 함수를 정의합니다. Epsilon greedy 방식은 epsilon이라는 매우 작은 확률로 exploration 하여 다른 값을 랜덤으로 선택하고 1 – epsilon의 확률로 exploitation하여 현재의 Q value들 중에서 가장 높은 값을 greedy하게 선택하는 방식입니다. 따라서 코드를 구현할 때 0부터 1 사이의 랜덤한 값이 epsilon이라는 값보다 작다면 가능한 action 중에서 하나를 랜덤으로 반환하며, 그게 아니라면 모든 Q 값을 돌면서 max 값을 가지는 action을 반환하도록 합니다.

```
In [5]: alpha = 0.4
        gamma = 0.8
        epsilon = 0.01
```

하이퍼파라미터를 정의합니다. 필요한 하이퍼파라미터는 Q값을 업데이트할 때 쓰이는 alpha, gamma, 그리고 epsilon greedy에서 중요한 기준이 되는 epsilon이 있습니다.

alpha 값은 learning rate로 학습을 하는데 있어 매우 중요한 요소입니다. 위에 Sarsa의 알고리즘을 보면 알 수 있듯이 예측값과 현재값의 차이인 에러를 얼마만큼의 비중을 둘 것인지를 결정합니다. 이 alpha의 값이 커지게 된다면 step을 지날 때마다 값의 변화가 크기 때문에 수렴하지 않고 발산하게 됩니다. 하지만 alpha 값이 너무 작아지게 된다면 step을 지날 때마다 값의 변화가 너무 작아 원하는 값을 얻기까지 학습에 오랜 시간이 필요하게 될 수 있습니다. 따라서 충분히 작지만 너무 작지 않은 값을 선택해야 합니다. 먼저 0.5의 값으로 학습하자 너무 작은 리워드를 갖는 것을 확인하여 더 작은 값인 0.4를 넣어보았습니다. 0.4일 때는 학습의 결과를 보자 -10000이 넘는

너무 작은 리워드를 갖는 경우가 거의 없어 발산하지 않으면서도 충분히 큰 값이라고 판단하여 0.4로 설정하였습니다.

gamma는 discount factor로 현재와 미래 중에 어떤 것을 더 중요하게 여길지 컨트롤할 수 있는 요소입니다. gamma 값이 클수록 예측값을 즉, 미래를 더 중요하게 여기며, gamma 값이 작을수록 현재의 값을 더 중요하게 여기게 됩니다. 우리의 최종 목표는 도착점에 도착하는 것입니다. 물론 절벽을 피하기 위해 현재의 값도 중요하지만 최종 목표에 도달해야만 에피소드가 종료되기 때문에 미래를 더 중요하게 여기고자 했습니다. 따라서 0.8로 설정하였습니다.

epsilon은 epsilon greedy 방식에서 exploration과 exploitation을 조절하는 요소입니다. epsilon은 매우 작은 값을 가져야 합니다. 따라서 우선 0.1보다는 작은 값을 넣어 주기로 결정하였습니다. 이 값을 조정하면서 결과를 관찰한 결과 0.05의 값을 가질 때는 -17의 리워드를 반환하는 경우가 많았으며 다른 하이퍼파라미터들을 조정하면 아예 발산하여 끝나지 않는 경우도 생겼습니다. 따라서 더 작은 값인 0.01을 넣어주자 -17의 리워드로 최적의 리워드는 아니지만 충분히 안전하게 잘 도착하는 결과를 이끌어 낼 수 있었습니다. 따라서 0.01 로 설정하였습니다.

```
In [6]: for i in range(4000):
            total_reward = 0
            state = env.reset()
            action = epsilon_greedy(state,epsilon)

            while True:
                next_state, reward, done, _ = env.step(action)
                next_action = epsilon_greedy(next_state,epsilon)

                Q[(state,action)] += alpha * (reward + gamma * Q[(next_state,next_action)]-Q[(state,action)])

                action = next_action
                state = next_state

                total_reward += reward

                if done:
                    break

            print("total reward: ", total_reward)
```

다음은 학습하는 부분입니다. 우선 모든 에피소드를 돌아야 하는데 4000번의 에피소드를 돌도록 하여 학습을 진행합니다. 에피소드마다 각 step에서의 리워드를 더한 cumulative 값이 최종 리워드가 되기 때문에 이 값을 total_reward로 정의해줍니다. 그리고 모든 에피소드는 독립적이기 state의 값을 환경의 reset 함수를 이용해서 리셋합니다. 그리고 처음 action을 epsilon_greedy 함수를 이용하여 정합니다.

이렇게 에피소드를 위한 준비를 했다면 에피소드의 각 step을 돌아야 합니다. 이 step을 도는 과정은 도착점에 도착했을 때만 종료되기 때문에 while True를 통해 무한루프를 생성하고 마지막에 if done일 때 이 루프를 빠져나가도록 코드를 작성해줍니다. 환경에는 step 함수가 있어 action을 넣었을 때의 timestep을 수행해주기 때문에 action을 넣어줍니다. 그리고 위에 환경의 소스코드 설명에도 작성하였듯이 이 리턴값은 (state, return, done, transition probability)입니다. 따라서 next_state, reward, done 값에 step 함수를 실행한 리턴값을 넣어줍니다. 여기서 transition probability 값은 이 과정에서 사용되지 않으며 또한 환경에서는 이 값을 1.0으로 고정시켜뒀기 때문에 따로 값을 넣어주지 않았습니다. 그리고 next_action은 다음 step에서의 state에서 어떤 action을 취할지를 담아둘 변수로 Sarsa 알고리즘은 on-policy이기 때문에 현재의 action을 결정했던 것과 똑같이 epsilon_greedy 함수를 사용하여 값을 넣어줍니다. 그럼 이제 현재의 state S, 현재의 action A, 현재 step의 reward R', 다음 step의 state S', 다음 step의 action A'을 모두 알고 있으며 alpha, gamma 값은 이미 정해둔 하이퍼파라미터이므로 Q 값을 업데이트 시킬 준비가 끝났습니다. 위에서 알고리즘 설명에 있던 수식과 같이 Q 값을 업데이트 시켜줍니다. reward + gamma * Q[(next_state,next_action)]

부분이 미래의 예측값으로 이 값에서 현재의 값인 Q[(state,action)]를 빼서 예측값과 현재값 간의 차이, 즉 TD error를 구합니다. 이 값에 alpha를 곱해 learning rate를 고려해주고 이 값을 현재의 Q 값에 더해준다면 Q value로 policy를 업데이트 할 수 있게 됩니다. 이제 action과 state를 다음 step의 action과 state로 갱신해 준 뒤, total_reward에 현재의 reward를 더합니다. 이 step을 끝냈을 때 done 상태라면 루프를 빠져나가 해당 에피소드가 종료되며 done이 아니라면 done 상태가 될 때까지 이 과정을 계속 반복합니다. 이렇게 학습은 종료됩니다.

```
In [11]: state = env.reset()
         total_reward = 0

         while True:
             env.render()

             action = max(list(range(env.action_space.n)), key = lambda x: Q[(state,x)])

             state, reward, done, _ = env.step(action)
             total_reward += reward
             print('Episode Reward= ', total_reward)

             if done:
                 break

         env.close()
```

다음은 이 학습의 결과를 확인해봅니다. 먼저 reset 함수를 통해 state를 다시 리셋해주고 total_reward를 0으로 만듭니다. 그리고 이번에는 테스트로 한 번의 에피소드만 돌릴 것이기 때문에 에피소드를 돌리는 루프가 필요없이 step에 관한 루프 하나만 작성해주면 됩니다. 각 step의 결과를 시각적으로 볼 수 있게끔 매 step마다 환경을 렌더링하는 render 함수를 사용하였습니다. 이제 결과를 확인하기 위해 state, action, reward, done 의 값을 넣어줘야 하는데 이 부분은 학습이 아니라 테스트이기 때문에 완성된 Q table이 있는 것입니다. 따라서 epsilon greedy와 같은 함수를 사용하는 것이 아니라 현재의 가장 큰 Q 값으로 action을 선택해주면 됩니다. 따라서 action을 현재의 state에서 모든 Q 값을 확인하고 그 중 가장 큰 값으로 넣어줍니다. 그리고 state, reward, done은 위의 학습하던 코드와 동일하게 해당 action을 했을 때 step함수의 리턴값으로 넣어줍니다. total_reward는 동일하게 계속 매 step의 reward를 cumulative하게 더해줍니다. 이 최종 리워드를 출력하고 만약 done이 true라면 도착점에 도달한 것으로 루프를 빠져나옵니다. 그리고 이제 코드에서 더는 환경을 사용할 부분이 없기 때문에 close 함수를 이용해 객체 내 모든 상태를 소멸합니다.

2) Q-learning

Q-learning은 Sarsa와 비슷한 부분이 많기 때문에 위의 설명과 중복되는 부분은 생략하고 추가되거나 달라진 부분만 설명하도록 하겠습니다.

5번 cell까지 모든 코드가 동일합니다.

```
In [6]: def update_Q_table(state, action, reward, next_state, alpha, gamma):
            Q_next = max([Q[(next_state, a)] for a in range(env.action_space.n)])
            Q[(state,action)] += alpha * (reward + gamma * Q_next - Q[(state,action)])
```

Q-learning은 Sarsa와 다르게 Q-table을 업데이트하는 action을 선택하는 Policy가 epsilon-greedy 방식이 아닌 greedy 방식이었습니다. 이 Q-table을 업데이트하는 부분을 함수로 뺐습니다. Q-table을 업데이트하기 위해 Q-learning에서 필요한 것들은 현재 step에서의 state S, action A, reward R', 다음 step의 state S', 그리고 미리 정의한 하이퍼파라미터 값인 alpha, gamma입니다. 따라서 이 변수들을 모두 인자로 받고 Q-table을 업데이트 합니다. Q-learning에서 가장 키포인트는 다음 step의 action을 업데이트하는 policy가 현재 step의 action을 선택하는 policy와 다르다는 것입니다. 이 다음 step에서의 action은 다음 step의 state에서 가능한 모든 action 중에서 가장 큰 값을 갖는 action으로 하며 그때의 Q 값을 Q_next라는 변수에 넣어줍니다. 이제 현재의 Q값을 업데이트합니다. Sarsa와의

차이는 epsilon-greedy 방식이 아닌 greedy 방식으로 업데이트한 next_action의 Q 값이 고려되었다는 것으로 Q[(next_state,next_action)] 대신 Q_next를 사용했다는 것 외에 나머지는 모두 같습니다.

```
In [7]: for i in range(4000):
            total_reward = 0
            state = env.reset()

            while True:
                action = epsilon_greedy(state, epsilon)
                next_state, reward, done, _ = env.step(action)

                update_Q_table(state, action, reward, next_state, alpha, gamma)

                state = next_state

                total_reward += reward

                if done:
                    break

            print("total reward: ", total_reward)
```

이제 학습을 위한 모든 준비가 끝나 학습을 시킬 차례입니다. Policy를 업데이트 하는 부분을 제외하고 다른 조건을 모두 Sarsa와 동일하게 맞추기 위해 4000번의 에피소드를 돌립니다. 모든 에피소드는 서로 독립적이라 total_reward는 처음에 0의 값을 가지며 state는 reset 함수를 통해 초기화합니다. Q-learning과 Sarsa 알고리즘 설명 부분에 있는 알고리즘 이미지를 보면 Q-learning은 action을 선택하는 부분을 step 반복문 안에서 하고 있습니다. 이는 action을 업데이트하는 policy가 현재 action을 선택한 policy와 같은 policy를 사용하는 게 아니라 매번 달라지기 때문에 계속 달라지는 action을 고려하기 위해 에피소드 안인 while문 내부에서 정의한 것입니다. 이 action은 epsilon_greedy 함수를 적용시켰을 때의 action을 넣어줍니다. 그리고 환경의 step 함수를 통해 현재 action의 step 결과를 next_state, reward, done에 넣어줍니다. 그리고 Q-learning의 특징인 update_Q_table 함수를 실행합니다. 이 과정을 통해 Q 값이 epsilon greedy가 아닌 greedy policy를 사용하여 업데이트 됩니다. action은 업데이트 된 action이 아니라 다음 step의 state에서 epsilon_greedy policy로 선택한 action이 계속 갱신될 것으로 action을 업데이트하는 부분이 없다는 차이가 있습니다.
결과를 확인하는 부분의 함수는 동일하여 생략합니다.

5. 최종 결과에 대한 설명

1) Sarsa

Sarsa 알고리즘을 4000번의 에피소드를 학습했을 때의 최종 리워드는 -17로 향하는 추세를 보입니다. 해당 결과를 보게 되면 가장 안전한 방향으로 우회하여 17번의 이동을 통해 도착점에 도달한다는 것을 알 수 있습니다. 하지만 이 환경에서의 목표는 가장 안전하게가 아닌 안전하면서도 가장 빠르게 도달하는 것이었습니다. Sarsa가 최적의 루트를 찾지 못 한 이유를 분석해보았습니다. Sarsa는 epsilon-greedy 방식으로 현재 action과 업데이트할 action을 선택합니다. epsilon이 아무리 작은 확률이라도 바로 옆에 절벽이 있어 큰 패널티를 받을 확률이 있기 때문에 가급적 절벽으로부터 멀리 떨어져서 이동하고자 하는 경향을 보인 것으로 생각됩니다.
결과는 다음과 같습니다.

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -1
```

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -2
o  o  o  o  o  o  o  o  o  o  o  o
x  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -3
x  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -4
o  x  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -5
o  o  x  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -6
o  o  o  x  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -7
o  o  o  o  x  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -8
o  o  o  o  o  x  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -9
o  o  o  o  o  o  x  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -10
o  o  o  o  o  o  o  x  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -11
o  o  o  o  o  o  o  o  x  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
        Episode Reward=  -12
```

```
o  o  o  o  o  o  o  o  o  x  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -13
o  o  o  o  o  o  o  o  o  o  x  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -14
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  x  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -15
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  x
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -16
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  x
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -17
```

2) Q-learning

Q-learning 알고리즘을 4000번의 에피소드를 학습했을 때의 최종 리워드는 -13을 향하는 추세를 보입니다. 해당 결과를 보게 되면 가장 최적의 루트로 13번의 이동을 통해 도착점에 도달한다는 것을 알 수 있습니다. 목표에 적합한 결과를 출력합니다. Q-learning이 Sarsa와 다르게 최적의 루트를 찾은 이유를 분석해보았습니다. Sarsa와 달리 Q-learning은 action을 업데이트할 때 epsilon-greedy가 아닌 max 값을 반영하는 greedy-policy를 이용합니다. 즉, 아무리 바로 옆에 절벽이 있어 큰 패널티를 받을 확률이 있더라도 greedy policy에 의해서 해당 위험은 모두 무시됩니다. 따라서 Q-learning은 Sarsa와 달리 최적의 루트를 찾아낸 것으로 생각됩니다.
결과는 다음과 같습니다.

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -1
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -2
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  x  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -3
```

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  x  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -4
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  x  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -5
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  x  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -6
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  x  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -7
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  x  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -8
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  x  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -9
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  x  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -10
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  x  o  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -11
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  x  o
o  C  C  C  C  C  C  C  C  C  C  T
       Episode Reward=  -12
```

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  x
o  C  C  C  C  C  C  C  C  C  C  T
      Episode Reward=  -13
```

3) 두 알고리즘에 대한 차이 비교와 분석 정리

현재 action과 업데이트할 action을 선택하는 policy가 epsilon-greedy로 같은 Sarsa는 항상 모든 action에 대한 Q 값이 고려됩니다. 따라서 근처에 절벽이 있다면 가능한 그 state로 가는 action을 피하고자 합니다. 그러므로 이 cliffwalking 환경 외에도 최적의 결과보다 가장 안전한 결과를 도출해내기를 원한다면 Sarsa 알고리즘을 사용하는 것이 적절할 것입니다. 해당 환경에서는 하이퍼파라미터를 더 조정하거나 더 많은 에피소드를 학습시키는 방법을 통해 조금이나마 더 최적의 루트에 가까운 결과를 얻을 수 있습니다. 하이퍼파라미터 조정 중 -15의 리워드를 갖는 결과를 출력하는 것까지 관찰할 수 있었습니다.

현재 action은 epsilon-greedy policy로 업데이트할 action은 greedy policy로 선택하는 Q-learning은 모든 Q가 아닌 가장 큰 Q 값만으로 다음 값을 업데이트 합니다. 따라서 근처에 절벽이 있더라도 그 확률은 모두 무시됩니다. 그러므로 이 cliffwalking 환경 외에도 최적의 결과를 원한다면 Q-learning 알고리즘을 사용하는 것이 적절할 것입니다. 반면 이 알고리즘은 주변의 위험은 고려되지 않으면서 Q 값을 업데이트 하기 때문에 가장 안전한 결과를 도출해내기 원하는 환경에서는 목표에 맞지 않는 결과를 도출할 수도 있습니다. 이런 경우에는 Sarsa 알고리즘을 사용하는 것을 권장합니다.