

먼저 환경을 Non-deterministic grid-world로 프로그램을 수정하였습니다.

Transition Probability는 환경이 주는 값이기 때문에 environment 파일을 수정해 줍니다.

```
WIDTH = 5 # 그리드월드 가로
Transition_prob = 0.8

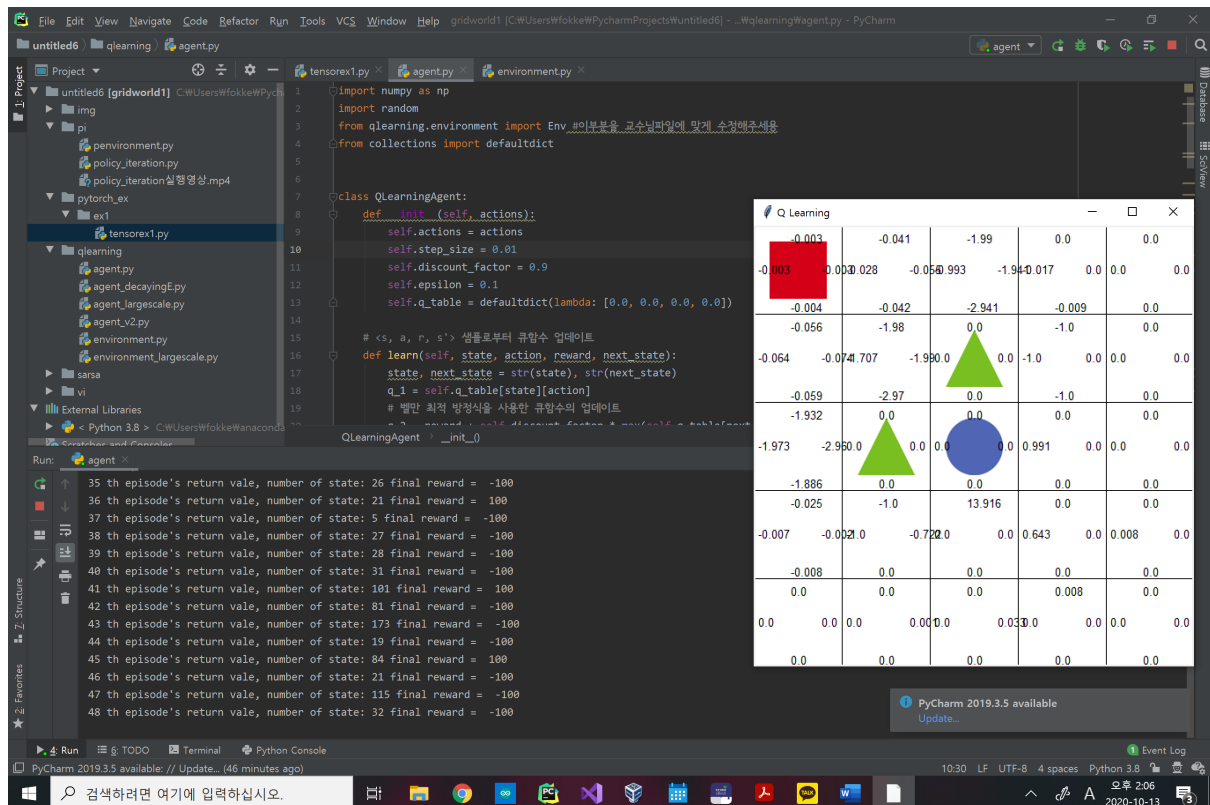
class Env(tk.Tk):
    def __init__(self):
        super(Env, self).__init__()
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.title('Q Learning')
        self.geometry('{0}x{1}'.format(HEIGHT * UNIT, HEIGHT * UNIT))
        self.shapes = self.load_images()
        self.canvas = self._build_canvas()
        self.texts = []
        self.transition_prob = Transition_prob
```

Env 클래스에 Transition_prob변수를 선언, 초기화 해주었습니다.

```
def step(self, action):
    state = self.canvas.coords(self.rectangle)
    base_action = np.array([0, 0])
    if np.random.rand() > self.transition_prob:
        action = np.random.randint(4)
    self.render()
    if action == 0: # 상
        if state[1] > UNIT:
            base_action[1] -= UNIT
    elif action == 1: # 하
        if state[1] < (HEIGHT - 1) * UNIT:
            base_action[1] += UNIT
    elif action == 2: # 좌
        if state[0] > UNIT:
            base_action[0] -= UNIT
    elif action == 3: # 우
        if state[0] < (WIDTH - 1) * UNIT:
            base_action[0] += UNIT
```

이후 step 함수에서 작은확률에 의해 매개변수로 들어온 action이 상,하,좌,우 중 하나의 random 한 행동으로 바뀌게 구현해 주었습니다.

이렇게 환경을 바꾸고 프로그램을 실행시켜 보았습니다.



그랬더니 random한 행동으로 인해 q함수값이 아주 이상하게 변해 경로를 찾아가지 못하는 현상을 발견하였습니다.

따라서수정이 필요함을 느껴 수정하였습니다.

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

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a')]$$

큐함수를 다음 큐함수에 의해 한번에 업데이트 하는 방식이 아닌 learning rate에 따라 조금씩 현재 큐함수를 업데이트 해나가도록, 즉 현재 큐함수가 다음 큐함수보다 dominant한 term이 되도록 수정해주었습니다. 이와 같은 수식으로 큐함수를 바꾸어서 다시 실행해 보았습니다. α (learning rate)값은 0.1로 주었습니다.

```
# <s, a, r, s'> 샘플로부터 큐함수 업데이트
def learn(self, state, action, reward, next_state):
    state, next_state = str(state), str(next_state)
    q_1 = self.q_table[state][action]
    # 벨만 최적 방정식을 사용한 큐함수의 업데이트
    q_2 = (1-self.learning_rate)*q_1+self.learning_rate*(reward + self.discount_factor * max(self.q_table[next_state]))
    self.q_table[state][action] += self.step_size * (q_2 - q_1)
```

The image shows a PyCharm IDE with a project named 'qlearning'. The main editor displays the 'agent_stochastic.py' file, which defines the 'QLearningAgent' class. The class has an 'init' method that sets parameters like 'actions', 'step_size', 'discount_factor', 'epsilon', 'q_table', 'learning_rate', and 'iteration'. The 'learn' method implements the Bellman optimality equation for Q-learning. The 'Run' console shows the results of 120 episodes, indicating that the agent has learned to reach the goal state (reward 100) in most episodes.

On the right side of the IDE, there is a 'Q Learning' window showing a 5x5 grid environment. The grid contains various obstacles (red squares, green triangles, blue circles) and a red path indicating the agent's trajectory. The grid cells are labeled with numerical values representing the Q-values for different actions.

그랬더니 큐함수의 값이 stochastic한 환경에 의해 잘못 계산되어도 그 잘못판단한 것을 현재 큐함수에 조금 반영하여 이후 차차 다시 잘 변하는 것을 확인할 수 있었습니다. 결국 여러 번의 episode진행이후 빨간 경로를 따라 잘 학습이 된 것을 확인했습니다.

아래는 최종 실행 영상입니다.