Reinforcement Learning Homework 3: Mountain Car Problem

Kichang Lee*
Department of Integrated Technology
College of Computing
Yonsei University
kichang.lee@yonsei.ac.kr

1 Introduction

Homework 3에서는 OpenAI에서 제공하는 gym library를 활용하여 Mountain Car Problem을 다룬다. Mountain Car Problem을 다룸에 있어 본 report에서는 'Grid-Tiling Method'를 기반으로 한 binary feature mapping 방식과 'Semi-Gradient-SARSA', 'Semi-Gradient-n-step-SARSA' 두가지의 학습 방식을 다룬다. 본 report는 Introduction 이후 Mountain Car Problem을 구체적으로 서술하는 Section 2과 문제 해결에 사용된 알고리즘들에 대한 설명과 구현을 다루는 Section 3, 실험 결과와 분석을 다루는 Section 4, 마지막으로 report를 마무리하는 Section 5로 구성되어 있다.

Table 1에는 이번 과제에 사용한 하드웨어 환경과 언어, 라이브러리 등에 대한 정보를 명시해 두었다. **모든 실험의 경우 재현가능성을 보장하기 위하여 numpy package의 random seed를 본인의 학번인 2022314416으로 통일 및 고정하여 진행하였다.**

	Specification
OS	Microsoft Windows 10 Education
OS Version	10.0.19042
CPU	Intel(R) Core(TM) i9-10850K CPU 3.60GHz, 3600Mhz
RAM	64GB
GPU	NVIDIA RTX 3090
Language	Python(=3.11.1)
Library	numpy(=1.23.5), gymnasium(=0.26.2)

Table 1: Environment Specification

2 Mountain Car Problem

이번 과제에서 다루는 Mountain Car Problem은 두 언덕 사이에 있는 자동차를 언덕 위의 목표 지점까지 옮기는 task이다. 해당 환경에서 자동차는 우측과 좌측으로 가속 혹은 아무것도 하지 않는 action을 취할 수 있다. 즉, 주어진 문제의 action space는 $\mathcal{A} \in \{Left, Nothing, Right\}$와 같이 구성된다. 제약 조건으로 단순한 가속 action만으로는 언덕을 오를 수 없다. Mountain Car에서의 state는 현재 자동차의 위치와 현재 자동차의 속도 두 가지의 변수로 구성된다. 시간 $t+1$에서 자동차의 위치 x와 자동차의 속도 v는 다음과 같은 관계를 가지는 수식으로 modeling 된다.

$$v_{t+1} = bound[v_t + F \times (A_t - 1) - G \times cos(3x_t)]$$
$$x_{t+1} = bound[x_t + v_{t+1}]$$

*Embedded Intelligent System Laboratory(EIS LAB)

위 수식에서 *bound* 연산은 자동차가 이동할 수 있는 범위를 제약하는 역할과, 앞서 언급한 자동차의 가속 조건을 제약하는 역할을 한다. 따라서 임의의 시간 t에 대해 $x_t \in [-1.2, 0.6]$과 $v_t \in [-0.07, 0.07]$를 만족하도록 제약된다[1]. 앞선 수식의 F와 G는 각각 force와 gravity에 대응되는데, 각각 0.001, 0.0025로 설정된다. 마지막으로 A_t의 경우 시간 t에 취하는 action을 의미하며 $\mathcal{A}(\{Left, Nothing, Right\}) \rightarrow \{0, 1, 2\}$와 같이 대응된다.

3 Algorithms and Implementation

알고리즘들의 설명과 구현에 대해서 설명하기에 앞서 report의 가독성을 위하여 가장 핵심적인 코드를 제외한 나머지 code는 https://github.com/leekichang/IIT6051$_R$L/tree/main/HW3/src에서 확인할 수 있도록 하였다.

3.1 Grid Tiling

구체적인 학습 알고리즘에 대해서 이야기 하기 전에 continuous한 position과 velocity를 state로 하는 환경을 discrete하게 매핑하기 위한 수단으로 본 report에서는 교과서에서 사용한 grid-tiling을 통한 coding을 사용하였다[2]. Grid-tiling method는 연속적인 다차원 공간에서 사용되는 coarse coding 방법 중 하나로, 연속적인 2D state space를 여러개의 overlapping 되는 tiling을 통해 특정 state가 각 tiling의 어떤 receptive field에 속하는지를 통해 discrete하게 mapping하는 역할을 수행한다. 기본적으로 해당 report에서 특별한 언급이 없는 경우 tiling의 수는 8개로 설정하였다.

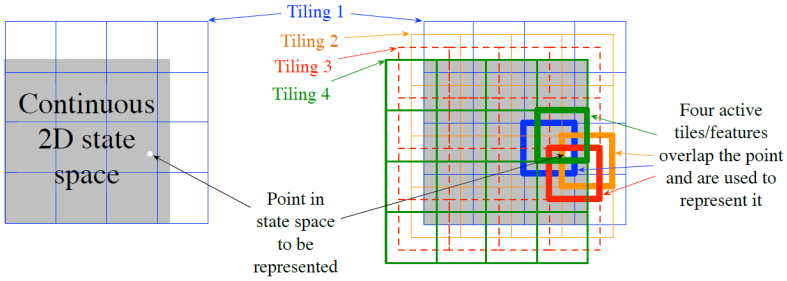


Figure 1: Grid Tiling Method

3.2 Semi-gradient n-step Sarsa

해당 report에서는 approximated action-value function $\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T\mathbf{x}(s, a)$를 구하기 위해 **Semi-gradient n-step Sarsa**를 사용한다. 해당 알고리즘에 대해서 설명하기에 앞서, 해당 report에서 특별한 명시가 없는 경우 $n - step = 1$로 설정하였으며, 이는 semi-gradient n-step Sarsa 알고리즘의 가장 간단하며 특수한 케이스로 semi-gradient sarsa와 동일하다. 조금 더 간략한 경우인 semi-gradient sarsa estimation을 우선적으로 설명하겠다.

Algorithm 1 Semi-gradient Sarsa

for each **EPISODE do**
  $S, A \leftarrow init()$
  for each **STEPS do**
    $A \leftarrow getAction(S)$
    $R, S' \leftarrow observe(Env(A))$
    if $S' isterminal$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$
      $Break$
    end if
    $A' \leftarrow \epsilon - greedy(\hat{q}(S', \cdot, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$
    $S, A \leftarrow S', A'$
  end for
end for

[1] 이는 교과서의 조건과 차이가 있으며 본 report에서는 gym 라이브러리에서 제공하는 조건을 명시하였으며 이를 따른다.

[2] 구현의 경우 교과서에서 제공한 http://incompleteideas.net/tiles/tiles3.html 코드를 그대로 사용하였다.

Algorithm 1는 semi-gradient Sarsa의 pseudocode이다. Semi-gradient Sarsa 알고리즘은 기존의 Sarsa 알고리즘에 Function approximation을 적용한 것으로, approximated action-value function에 해당되는 $\hat{q}(S, A, \mathbf{w})$를 구성하는 $\mathbf{w}$를 업데이트 하는 과정이 일반 Sarsa 알고리즘과의 가장 핵심적인 차이이다. Semi-gradient *n-step* Sarsa로의 확장 또한 기존 Sarsa에서 n-step Sarsa 알고리즘으로의 확장과 유사하다. Semi-gradient Sarsa가 weight를 update하기 위해서 바로 다음 state의 상황을 bootstrapping하고, semi-gradient n-step Sarsa의 경우에는 그보다 먼 미래의 action-value를 bootstrap 한다. 해당 알고리즘의 Algorithm 1의 update 부분에서 target에 해당되는 $R + \hat{q}(S', A', \mathbf{w})$를 return G로 대체하면 된다. Report의 가독성을 위하여 구체적인 pseudocode는 생략하니 교과서의 chapter 10.2를 참고하라. Episodic semi-gradient n-step sarsa 구현은 code **??**를 따른다. 해당 코드의 *agent.learn*() 함수가 내부적으로 weight를 update하는 역할을 담당한다. Grid Tiling에서 state가 매핑되는 tile에 해당되는 weight에 대해서만 update를 수행하여 semi-gradient 기반 update를 수행할 수 있도록 한다. 이번 과제에서는 discount factor γ의 값은 1로 고정하였으며, reward는 gym 라이브러리의 환경에서 반환해주는 값을 그대로 사용하였으며 이 또한 과제의 조건과 동일하게 terminal state가 아닌 경우 -1을 얻게 된다.

Listing 1 Semi-gradient n-step Sarsa

```
def sarsa(self):
  time, is_terminal, T     = 0, False, float('inf')
  self.currState, _        = self.env.reset()
  action                   = self.agent.getAction(self.currState)
  states, actions, rewards = [self.currState], [action], [0]
  while True:
    time += 1
    if time < T:
      next_state, next_action, reward, is_terminal = self.step(action)
      states.append(next_state)
      actions.append(next_action)
      rewards.append(reward)

      if is_terminal:
        T = time

    update_time = time - self.n_step
    if time >= self.n_step:
      returns = 0.0
      for t in range(update_time + 1, \
                min(T, update_time + self.n_step) + 1):
        returns += rewards[t]

      if time <= T:
        returns += self.agent.getValue(states[update_time+self.n_step],
                                       actions[update_time+self.n_step])

      if not is_terminal:
        self.agent.learn(states[update_time],\
                         actions[update_time], returns)

    if update_time == T-1:
      break
    self.currState, action = next_state, next_action
  return time
```

Listing 2 Update function

```
def learn(self, state, action, target):
  active_tiles = self.getActiveTiles(state, action)
  estimation   = np.sum(self.weight[active_tiles])
  delta        = self.alpha * (target - estimation)
  for tile in active_tiles:
    self.weight[tile] += delta
```

1) Draw cost-to-go function (max$_a$ ^q(s; a;w)) learned during one run.

A. You must explain learning method you applied to training $\boldsymbol{w}$ parameters of action-value function ^q($\boldsymbol{s}$; $\boldsymbol{a}$; $\boldsymbol{w}$) . *(30 points)*

B. Draw for 10, 100, 1000 d iterations, and final shape after convergence (*20 points).*

Figure 2: Question 1

4 Experiments

4.1 Question 1

첫 질문의 경우 parameter **w**가 학습되는 알고리즘에 대해서 설명하고, cost-to-go function($-max_a(\hat{q}(S, A, \mathbf{w}))$)의 형태를 3차원상에 그리는 것이다. 첫번째 질문의 경우 Section 3에서 설명을 하였다. Figure 3은 cost-to-go function을 각각 10, 100, 1000 episode에 대해서 plot한 것이다. Episode가 많이 진행됨에 따라 교과서의 Figure 10.1과 유사한 형태를 보이는 것을 확인할 수 있었으며, 값들도 커지는 것을 확인할 수 있으며 terminal state에 해당되는 position= 0.6에 가까워지며, cost값이 작아지는 경향성도 확인할 수 있다.

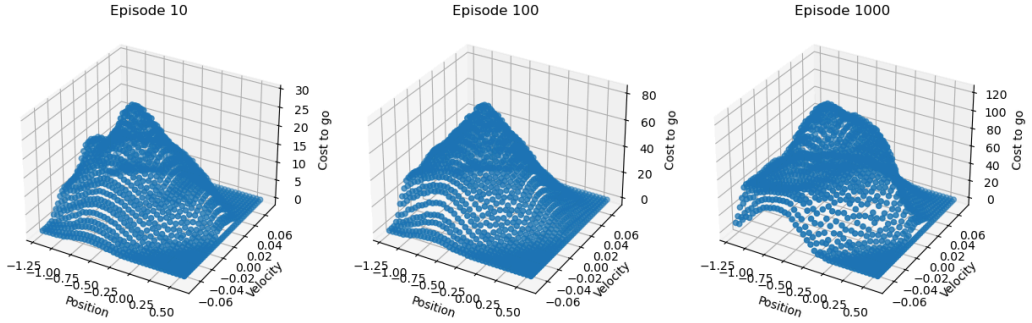


Figure 3: Cost-to-go function after 10, 100, 1000 episodes.

4.2 Question 2

두번째 질문의 경우 주어진 설정에서 action-value function이 적절하게 학습되어 converge하는데 까지 얼마나 많은 episode가 필요한지를 묻고 있다. 이 과정을 위해 학습의 convergence를 어떻게 판단할가에 대한 이야기가 선행되어야 한다. 본 report에서는 교과서의 Figure 10.2와 같이 에피소드당 요구되는 step 수를 통해 이를 판단하고자 한다. 가령 한 episode내에서 소요되는 step 수가 줄어든다면, 정상적으로 학습이 진행되고 있다고 판단할 수 있으며, 해당 step 수의 변화가 유의미하지 않아지는 시점이 온다면, converge가 된 것이라고 판단하고자 한다[3]. 실험의 경우 교과서에 주어진 것과 동일하게 총 500 episode를 수행하였으며, α 값 또한 $0.1/8, 0.2/8, 0.5/8$ 3가지에 대해서 실험을 진행하였다. Figure 5의 경우 그 결과를 보여주고 있다. α 값과 무관하게 1000번 이상의 step이 필요했던 것과 달리 200 300 episode 시점부터는 평균적으로 100 200 step 안에 episode를 마칠 수 있었음을 확인할 수 있다. Convergence를 확인하기 위하여 RMS error 혹은 policy의 변화 유무를 확인할 수도 있겠지만 해당 방식들은 생략하도록 하겠다. 학습된 model의 simulation animation은 **https://youtu.be/WMrDFAbMHuA**에서 확인할 수 있다.

[3]해당 해석은 homework2에서 windy grid world의 수렴을 확인하는 과정에서도 활용하였었다.

2) How many episodes do we need until convergence. *(20 points).*
Must Include simulated animations for several steps like this video (https://www.youtube.com/watch?v=RIkjCwA6DvA)

Figure 4: Question 2

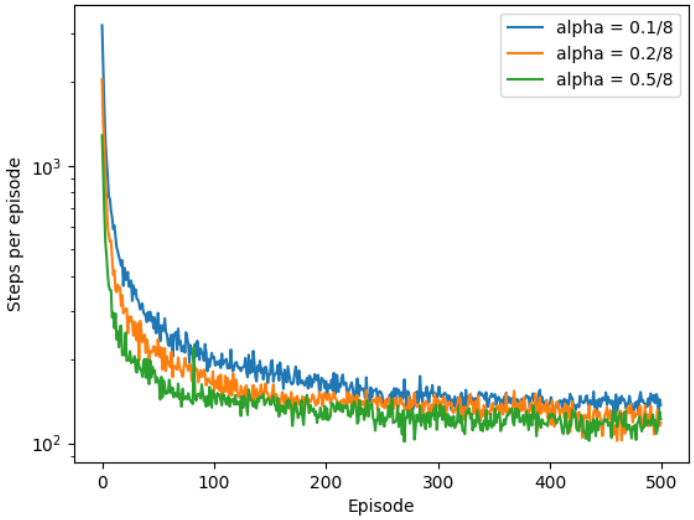
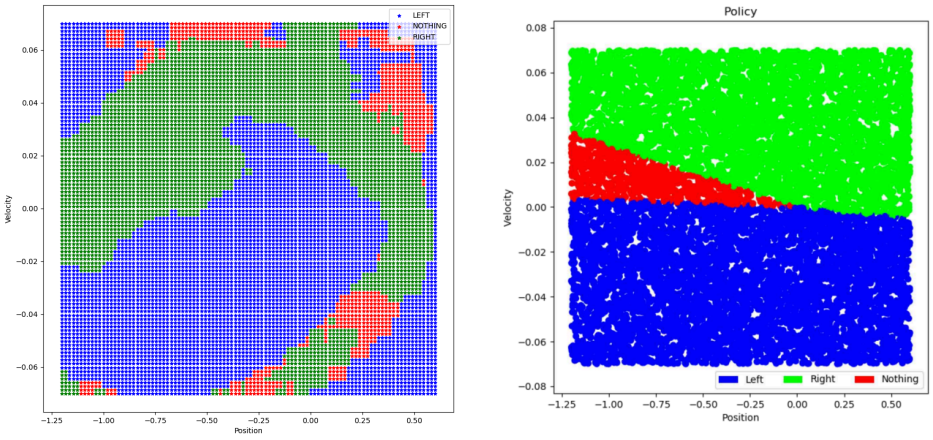


Figure 5: Question 2

4.3 Question 3

마지막 질문은 optimal policy를 cost-to-go function을 통해 얻어내는 것이다. Figure 6의 좌측 그림은 cost-to-go function상에서 max value를 산출하는 action을 2D 공간상에 plot한 것으로 푸른색은 왼쪽으로 가속, 녹색은 오른쪽으로 가속, 빨간색은 아무것도 하지 않는 것을 의미한다. 반면 우측 그림은 과제 소개에 첨부되어 있는 policy plot이다. 해당 두 그림을 비교해보았을 때, semi-gradient Sarsa algorithm을 통해서 구한 optimal policy가 과제에서 주어진 그림과 유사한 경향을 보이지 않는다는 것을 확인할 수 있다.

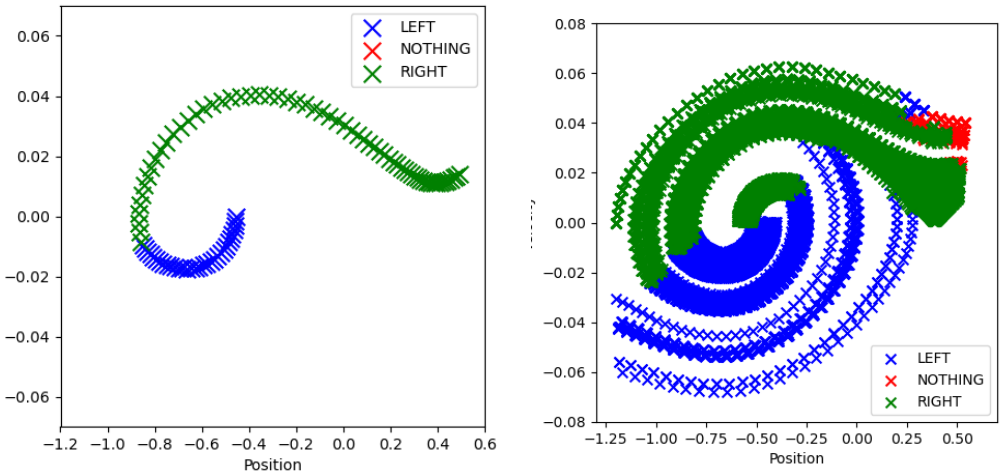


(a) Empirical result (b) From assignment specification

Figure 6: Optimal policy plot

실험을 통해 구해진 좌측의 optimal policy가 정상적으로 잘 구해진 것인지 확인하기 위하여 9000 episode를 거치며 학습된 action-value function을 적용하여 1번, 100번의 episode를 수행하며 실제로 방문한 state에서 선택한 action을 plot해보았다.

각 episode를 시작하는 initial state는 random하게 initialize 되었다. Figure 7 (a)를 보면 최초 시작 지점인 -0.4 부근에서 좌측 경사를 활용하기 위해 왼쪽으로 가속하다 특정 지점에 도달하면 우측으로 가속하여 최종 목적지에 도달한다. 해당 결과는 꽤나 직관적으로 과제



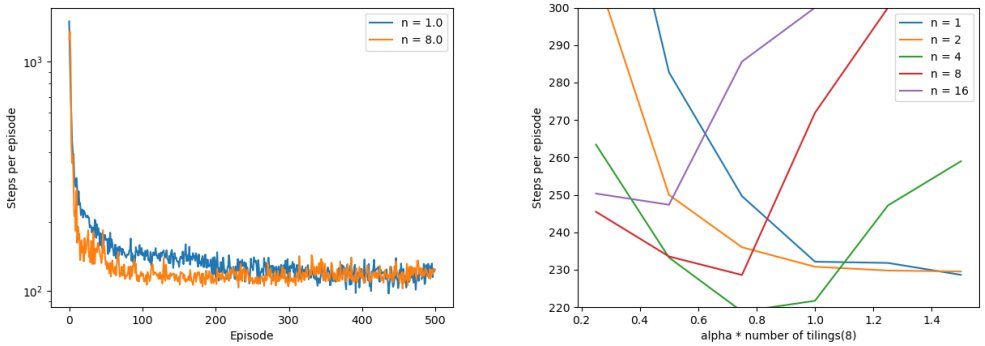
(a) After single episode (b) After 100 episodes

Figure 7: Actual actions from two different number of episodes

소개에 주어진 그림이 optimal하지 않다는 것을 보여주는 반례이다. 가령 position이 -0.8이고 velocity가 -0.01인 지점은 좌측 경사로에서 오른쪽 방향으로의 가속을 시작하기 좋은 지점임에도 우측의 그림에는 오른쪽 방향으로의 가속이 optimal action이 아니라고 표현되고 있다. 물론 실험을 통해 얻은 cost-to-go function에서의 그림에서도 value function이 제대로 학습하지 못한 state에 대한 오류가 있을 수 있지만, 100번의 episode를 통해 볼 때, optimal policy의 전체적인 개형자체는 실험을 통해 얻은 좌측에 가까울 것으로 사료된다.

4.4 Exploration

앞서 다룬 내용들을 통해 과제에서 요구한 질문들에 대하여 논의했다. 해당 과제에서 더 많은 변수들이 존재하여 해당 부분들에 대해서도 교과서에서 보여주는 실험들을 추가적으로 진행해보았다. 추가적인 실험은 총 두가지로 교과서의 Figure 10.3과 Figure 10.4를 다루며 그 내용은 n-step sarsa algorithm에서 n-step 변수와 α 변수가 학습에 어떤 영향을 미치는지를 분석한다.



(a) Effect of n-step parameter (b) Effect of n-step and α parameter

Figure 8: Actual actions from two different number of episodes

Figure 8은 해당 실험들의 결과로 Figure 8(a)는 n-step을 1과 8로 나누어 semi-gradient sarsa와 semi-gradient n-step (8-step) sarsa의 성능을 비교한다. 앞선 실험에서 언급했듯, 나타내는 값은 episode를 완료하는데 소요된 step수를 여러 시행에 대해 평균낸 것으로 그 소요된 step수가 줄어든다는 것은 곧 수렴이 되고 있다는 의미로 해석할 수 있다. 실험 결과에 따르면, single step을 적용한 sarsa 알고리즘보다는 8-step을 보는 알고리즘이 조금 더 빠른 수렴 속도를 보여줬다. 이는 미래의 정보를 bootstrapping 하며 학습을 진행하는 n-step sarsa 방식이 mountain car problem에서도 유효함을 확인할 수 있었다. Figure 8(b)의 경우 α의 값을 변화

시키며 여러 n-step 수에 대해서 episode당 평균 소요 step 수를 나타낸 것으로, 앞선 실험에서 나올 수 있는 질문인 'n-step이 single step보다 좋은 성능을 보여준다면, 그 n 값을 어떻게 설정하고, alpha 값은 어떻게 설정하는 것이 좋은가?'에 대한 해답이라고 할 수 있다. 경향성의 경우 1, 2, 4 순으로 step 수를 증가시켜본 결과 조금 더 큰 step 수에서 수렴 속도가 빠름을 확인할 수 있었다. 그러나 이는 α 값과도 관련이 있었으며 $n = 4$인 경우에서 α 값이 특정 값을 초과하는 순간 앞선 1, 2의 경우보다 소요되는 step의 수가 증가하는 것을 확인하였다. 또한 단순히 step 수를 늘리는 것이 높은 성능을 보장하지는 않았다. $n = 8, 16$의 경우 $n = 1, 2$보다 좋지 못한 성능을 보이는 것 또한 확인할 수 있었다. 최적의 parameter는 4-step에서 $\alpha = 0.75$로 설정한 경우였다.

5 Conclusion

이를 통해 주어진 문제들과 추가적인 실험에 대한 내용을 모두 다루었다. 이번 과제를 통해 Open AI에서 제공하는 gym library의 classic control의 대표적인 문제인 Mountain Car Problem에 function approximation을 통한 학습을 진행하였으며, 이 과정에서 tiling을 통한 continuous state의 mapping 방식과 episodic semi-gradient n-step sarsa algorithm 등에 대하여 다루었다. 이후 cost-to-go function의 형태와 이를 기반으로 한 optimal policy를 확인하고, 다양한 변수에 대한 수렴속도와 수렴성에 대해서 실험적으로 확인하는 절차를 거쳤다. 또한 추가적인 실험을 통해 step의 숫자와 α 파라미터가 학습에 끼치는 영향에 대한 실험 또한 진행하여 앞서 다룬 semi-gradient n-step Sarsa 알고리즘에 대한 이해도를 높일 수 있었다.