Reinforcement Learning Homework 2

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

1 Introduction

Homework 2에서는 Temporal Difference (TD)와 Monte Carlo (MC) method를 통해서 총 3가지의 강화학습 예제를 풀어본다. 첫번째 문제는 5 state random walk 상황에서 TD와 MC method를 적용하여 RMS (Root Mean Square) error의 개형을 확인하는 것을 목표로 한다. 두번째 문제는 19-state random walk에서 step의 크기를 변동시키며 평균 RMS error의 개형을 확인한다. 마지막 문제는 windy grid world에서 timestep에 따른 episode 수를 측정한다.

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	macOS
OS Version	Ventura 13.2.1
Language	Python(=3.9.7)
Library	numpy(=1.20.3), matplotlib(=3.4.3)

Table 1: Environment Specification

2 Problem 1: Random Walk

2.1 Problem Statement and Code Implementation

첫번째 문제의 경우 양쪽 끝에 2개의 terminal state를 포함한 총 7개의 state가 한줄로 나열되어 있는 Markov Decision Process (MDP)에서 random walk를 수행하는 경우 각 state의 value 값들을 확인하는 문제이다. 해당 MDP의 구조는 Figure 1과 같다. 문제의 조건은 다음과 같다. Reward의 경우 reward +1을 주는 경우인 state E에서 우측에 있는 terminal state로의 transition을 제외한 모든 transition에 대해서 0의 reward를 제공한다. 시작하는 state의 경우 state C로 하며, Temporal Difference를 적용하는 경우 n-step이 1인 TD(0)을 사용한다. Step-size parameter에 해당되는 α 값는 0.1로 설정하며 discount factor γ는 1로 state value의 초기값은 0.5로 설정하였다.

Figure 1: A small Markov process for generating random walks.

*Embedded Intelligent System Laboratory(EIS LAB)

해당 random walk 환경을 구현하기 위해 python 환경에서 RandomWalk class를 생성했으며 해당 코드는 아래와 같다.

Listing 1 Random Walk Class: __init__()

```
    def __init__(self, N=5, n_step=1,GAMMA=1.0, is_Q1=True):
        self.N = N
        self.is_Q1 = is_Q1

        if is_Q1:
            self.init_values            = np.zeros(N + 2)
            self.init_values[1:N+1]     = 0.5
            # use this True value for Q1
            self.TRUE_VALUE         = np.zeros(N + 2)
            self.TRUE_VALUE[1:N+1] = np.arange(1, N+1)/(N+1)
            self.TRUE_VALUE[N+1]    = 1
        else:
            # use this True value for Q2
            self.init_values         = np.zeros(N+2)
            self.TRUE_VALUE          = np.arange(-1*(N+1), N+3, 2) / (N+1)
            self.TRUE_VALUE[0], self.TRUE_VALUE[-1] = 0, 0


        self.actions            = {"LEFT":0, "RIGHT":1}
        self.term_state         = [0, N+1]
        self.GAMMA              = GAMMA
        self.n_step             = n_step
```

해당 문제와 두번째 문제의 구조적 유사성을 고려하여 첫 문제와 두번째 문제에 모두 사용할 수 있도록 객체를 작성하였다. N의 경우 state의 개수를 의미하며 terminal state는 제외된 숫자이다. 이후 ground truth에 해당하는 value의 경우 문제 설명에 명시된 것과 같이 $y = x$ 꼴을 가지도록 설정하였다. Action의 경우는 왼쪽 혹은 오른쪽 state로만 넘어갈 수 있도록 설정하였다.

Listing 2 Random Walk Class: random_walk()

```
class RandomWalk:
    def random_walk(self):
        return -1 if np.random.binomial(1, 0.5) == self.actions["LEFT"] else 1
```

Code 2의 `random_walk` 함수의 경우 numpy package에 포함된 binomial 함수를 통해 난수를 생성하여 random하게 action을 선택하여 random walk이 이루어질 수 있도록 구현한 것이다.

Listing 3 Random Walk Class: MonteCarlo()

```
class RandomWalk:
    def MonteCarlo(self, values, init_state=3, alpha=0.1):
        state = init_state if init_state != None else (self.N+2)//2
        trace = [init_state]
        while True:
            state += self.random_walk()
            trace.append(state)

            returns = 1.0 if state == self.N+1 else 0.0
            if state in self.term_state:
                break

        for state_ in trace[:-1]:
            values[state_] += alpha * (returns - values[state_])
```

Code 3의 `MonteCarlo()` 함수는 MC method를 구현한 것으로, terminal state에 도달할 때까지 반복하여 random walk을 수행하며 terminal state에 도달하는 경우 최종적인 return을 계산하도록 하였는데, $\gamma = 1.0$인 경우를 가정하고 있으며 우측에 존재하는 terminal state가 아닌 경우 모든 reward가 0인 상태이므로 상기한 것과 같이 단순하게 구현하였다.

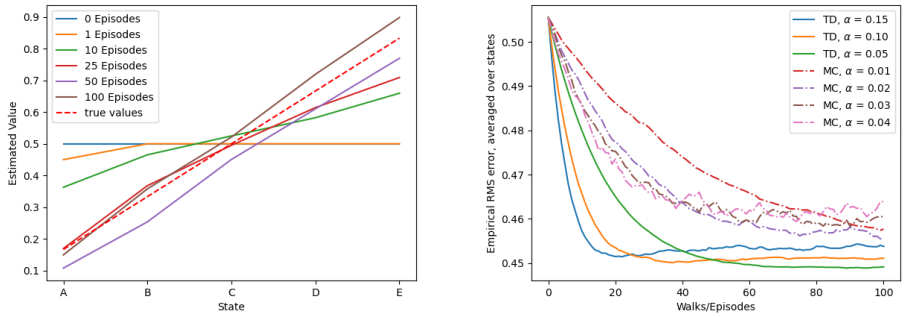
Listing 4 Random Walk Class: TD()

```
class RandomWalk:
    def TD(self, value, n=1, alpha=0.1, init_state=None):
        state = init_state if init_state!=None else (self.N+2)//2
        states, rewards = [state], [0]
        time = 0
        T = np.inf
        while True:
            time += 1
            if time < T:
                next_state = state + self.random_walk()
                if self.is_Q1:
                    # use this reward for Q1
                    reward = 1 if next_state == self.N+1 else 0
                else:
                    # Use this reward for Q2
                    if next_state == self.N+1:
                        reward = 1
                    elif next_state == 0:
                        reward = -1
                    else:
                        reward = 0
                states.append(next_state)
                rewards.append(reward)
                if next_state in self.term_state:
                    T = time
            update_time = time - n
            if time >= n:
                returns = 0.0
                for t in range(update_time + 1, \
                               min(T, update_time + n) + 1):
                    returns += pow(self.GAMMA, \
                                   t - update_time - 1) * rewards[t]
                if time <= T:
                    returns += pow(self.GAMMA, n)\
                          * value[states[(time)]]
                update_state = states[update_time]
                if not update_state in self.term_state:
                    value[update_state] += alpha * \
                        (returns - value[update_state])
            if update_time == T - 1:
                break
            state = next_state
```

Code 4의 `TD()`는 n-step TD method를 구현한 것으로, 첫번째 문제에 제시된 TD(0)의 상황을 위해서는 n 값을 1로 설정하면 된다. Timestep에 해당되는 `time` 변수가 설정한 n 값보다 커지게 되는 시점, 즉 $TD(n)$을 계산할 수 있는 시점부터 update를 시작하며, 각 timestep 별로 discounting factor인 γ를 적용하여 연산을 진행한다. 전체 while loop의 마지막에 위치한 for loop 내부에서 $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^n V(S_{t+n})$에서 $R_{t+1} + \gamma R_{t+2} + ...$ 부분에 해당하는 return들을 계산하고, 아래 if문에서 마지막 $\gamma^n V(S_{t+n})$ 부분을 계산한다. 최종적으로 TD method의 update 형식인 $V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t)$가 그 아래 if문에서 이루어지며, terminal state의 경우에는 update되지 않도록 예외 처리를 하였다.

2.2 Evaluation

해당 코드를 통해 주어진 parameter들을 기반으로 확인한 결과는 아래와 같다.



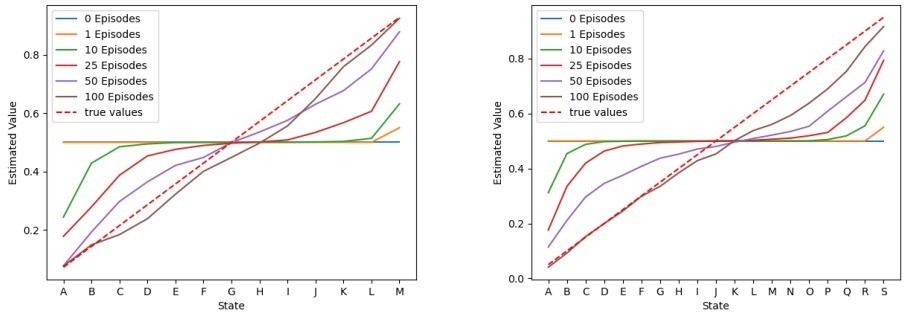
(a) Values learned by TD(0) after various numbers of episodes.

(b) Learning curves for TD(0) and constant- MC methods, for various values of , on the prediction problem for the random walk.

Figure 2: Problem 1 Evaluation

Figure 2 (a)는 이번 과제에서 명시적으로 확인할 필요는 없는 자료이지만, episode를 많이 학습하며 model이 점점 ground truth와 유사한 value 값들을 가지게 되는 것을 확인할 수 있다. 해당 결과는 textbook에 명시된 것과 같이 TD(0)의 경우를 대상으로 산출된 결과임을 명시한다. Figure 2 (b)는 MC방식과 TD방식을 다양한 α 값에 대하여 RMS error를 산출한 것이다. 실제 assignment specification에 첨부되어 있는 그래프와 유사한 개형을 가지고 있음을 보여준다. 강의 시간에 다룬것과 유사하게 해당 문제에 있어서는 TD method가 MC method에 비해서 좀 더 효율적으로 학습되는 것을 확인할 수 있으며, 적절한 α 값을 설정함을 통해 RMS error를 최소화할 수 있다는 것도 확인할 수 있다.

추가적으로 state가 더 많아진 경우, n-step의 크기를 키운 경우 결과가 달라지는 것을 관찰해 보기 위하여 해당 값들을 변경시키며 추가적인 실험을 진행하였다. Figure 3의 경우 동일하게 TD(0) 방식을 통해 13-state, 19-state에 대하여 value를 산출한 결과이다. Figure 2 (a)와 비교



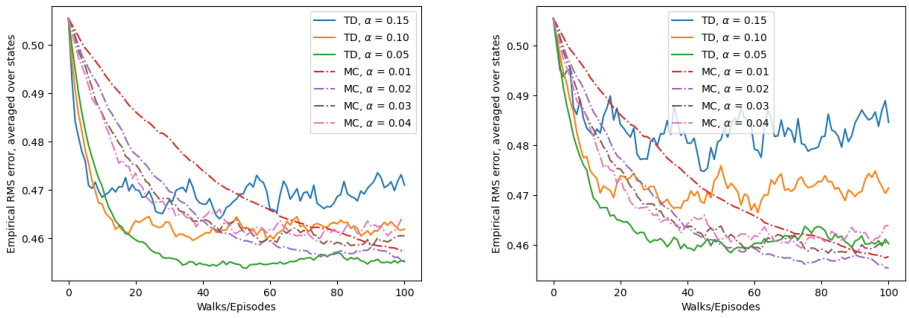
(a) Values learned by TD(0) after various numbers of episodes in 13-state MDP.

(b) Values learned by TD(0) after various numbers of episodes in 19-state MDP.

Figure 3: Problem 1 Evaluation: More states

할 때, state의 수가 적은 경우에는 빠르게 ground truth value와의 차이를 줄여나가며 개형을 따라잡지만, state의 수가 늘어난 경우 model이 예측하는 value의 값이 ground truth의 값을 따라가는 속도가 현저하게 느려지는 것을 확인할 수 있다.

Figure 4의 경우 문제의 조건에서 n-step의 크기를 5, 10으로 증가시킨 경우 RMS error curve가 학습과정에서 어떻게 변화하는지를 관찰한 것이다. n-step의 크기가 증가한 모든 경우에서 TD(0)과 비교하여 RMS error가 증가하는 모습을 보였으며, 특히 $\alpha = 0.15, 0.10$의 경우에서는 학습 과정이 불안정해지는 모습도 관찰할 수 있었다.



(a) Learning curves for TD(5) and constant-MC methods, for various values of , on the prediction problem for the random walk.

(b) Learning curves for TD(10) and constant-MC methods, for various values of , on the prediction problem for the random walk.

Figure 4: Problem 1 Evaluation: Larger n-step

3 Problem 2: n-Step TD

3.1 Problem Statement and Code Implementation

두번째 문제의 경우 첫번째 문제와 거의 동일한 설정이었으나 주어진 문제와 동일한 결과가 쉽게 산출되지 않아 textbook의 설정을 참고하여 코드를 작성하였다. 달라진 점은 다음과 같다. 기존의 random walk에서는 우측 terminal state를 제외한 모든 state의 reward가 0이었지만, 두번째 문제에서는 왼쪽 terminal state를 택하는 경우 -1의 penalty를 부여하며 value들의 초기값 또한 0.5가 아닌 0으로 초기화하도록 한다. 또한 state의 수가 5에서 19로 증가하며, 특정 α 값에 대하여 여러번(=100) RMS error를 측정하여 그 평균을 취한다. TD method의 구현은 Code 4를 참조하면 된다.

3.2 Evaluation

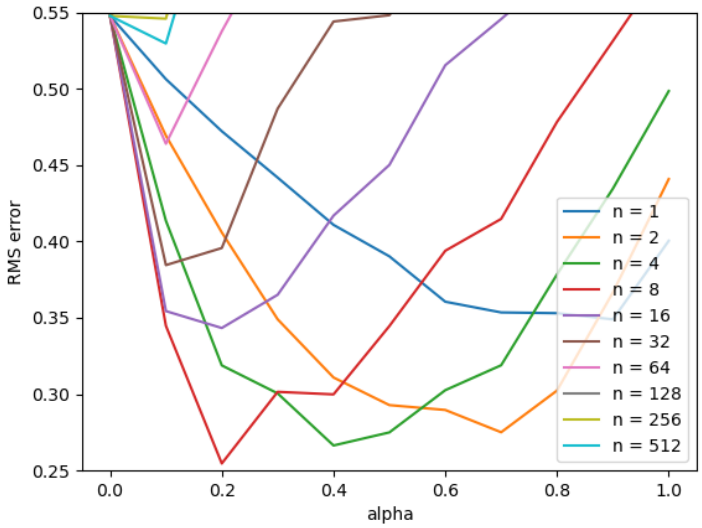


Figure 5: Performance of n-step TD methods as a function of α, for various values of n, on a 19-state random walk task.

Figure 5는 주어진 조건에서 진행한 실험의 결과로 주어진 그림과 유사한 개형을 얻을 수 있었다. 특히 $n = 8, \alpha = 0.2$의 설정에서 가장 좋은 성능을 확인할 수 있었으며, 너무 큰 n 값을 설정하는 것이 학습에 오히려 좋지 않은 영향을 주는 것 또한 확인할 수 있었다.

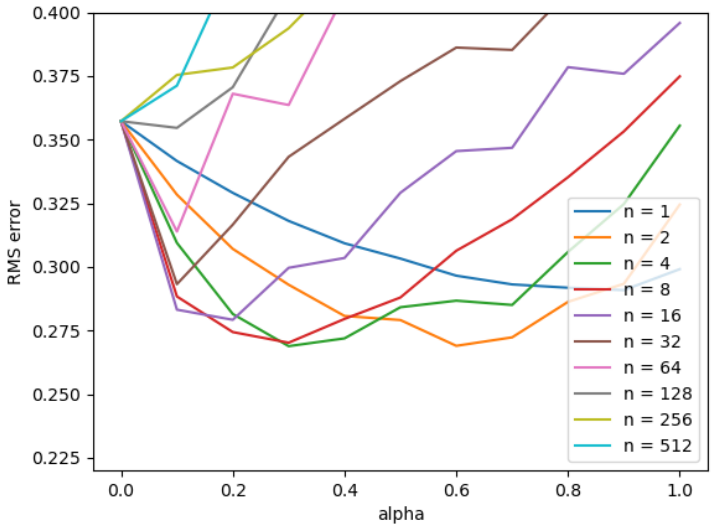


Figure 6: Performance of n-step TD methods as a function of α, for various values of n, on a 19-state random walk task.

Figure 6은 첫 문제와 동일한 설정 아래에서 state의 수만 19개로 증가시켜 해당 실험을 진행한 것으로 Figure 5과 거의 유사한 결과가 도출되었다. 하지만 RMS error의 범위에는 차이가 있었는데 이는 value값의 초기값의 설정의 차이로 보이며, 첫 문제의 경우 모든 value를 모두 0.5로 초기화하였으며 ground truth value와 실제 value간의 차이가 크지 않기 때문에 발생하는 것이지 근본적인 차이가 존재하는 것으로 보이지 않는다.

4 Problem 3: Windy Grid World

4.1 Problem Statement and Code Implementation

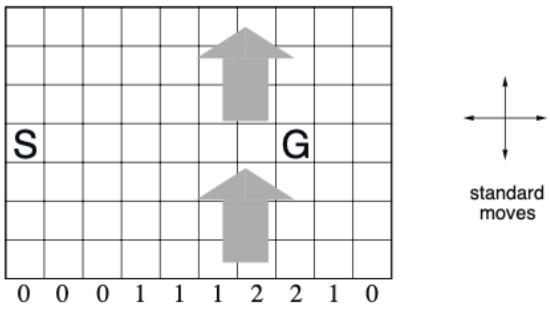


Figure 7: Gridworld in which movement is altered by a location-dependent, upward "wind."

마지막 문제의 경우 7×10 windy grid world에서 목적지까지 도달하는 과정을 학습하는 것이다. Windy grid world의 경우 기존과 달리 location-dependent한 영향이 있는 상태로 특정 state에 위치하게 되면 다음 action이 해당 영향을 받게된다. 주어진 상황에서는 4 7 그리고 9번째 열에서 1칸, 8과 9번째 열에서는 두칸 위로 이동하는 영향을 받는다.

Code 5는 windy grid world를 위한 class를 정의하는 부분으로 world의 크기, 바람이 부는 위치, `eps`는 travel에 대한 확률이며, `alpha`는 step size에 해당된다.

Code 6의 경우 특정 action이 결정된 경우 해당 action을 통해 다음 위치, 즉 state를 찾는 함수이다. 해당 부분을 구현함에 있어서 주의할 점은 바람의 영향으로 grid 밖으로 나가게 되는 상황 혹은 grid 벽에 붙어 더 갈려고 시도하는 상황을 주의하여야 하는데, 해당 부분은 `max` 함수와 `min` 함수를 통해 그런 경우 grid world의 범위를 벗어나지 않도록 하였다.

Listing 5 WindyGridWorld Class: __init__()

```
class WindyGridWorld:
    def __init__(self,
                 H,
                 W,
                 WIND,
                 eps=0.1,
                 alpha=0.5,
                 reward=-1.0,
                 init_state=[3,0],
                 goal=[3,7]
                 ):
        self.H, self.W   = H,W
        self.wind        = WIND
        self.actions     = {"UP":0,"DOWN":1,"LEFT":2,"RIGHT":3}
        self.actions_lst = [0, 1, 2, 3]
        self.eps         = eps
        self.alpha       = alpha
        self.reward      = reward
        self.init_state  = init_state
        self.goal        = goal
```

Listing 6 WindyGridWorld Class: move()

```
class WindyGridWorld:
    def move(self, state, action):
        i, j = state
        if action == self.actions["UP"]:
            return [max(i-1-self.wind[j], 0), j]
        elif action == self.actions["DOWN"]:
            return [max(min(i+1-self.wind[j], self.H - 1), 0), j]
        elif action == self.actions["LEFT"]:
            return [max(i - self.wind[j], 0), max(j - 1, 0)]
        elif action == self.actions["RIGHT"]:
            return [max(i - self.wind[j], 0), min(j + 1, self.W - 1)]
```

Code 7는 한 번의 episode를 전체적으로 수행하는 과정으로, 세번째 문제에서 가장 핵심적인 ϵ-greedy Sarsa를 수행하는 부분이다. 간략하게 설명하자면, 랜덤한 확률($=\epsilon$)로 action중 하나를 선택하여 수행한다. 그렇지 않은 경우에는 greedy한 방식으로 action을 선택한다. 이후 $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$를 통해 update를 수행한다. 해당 과정을 목적지(=terminal state)에 도달할 때까지 반복한다.

4.2 Evaluation

Figure 8은 해당 문제에서 요구한 Time-steps v.s. Episode에 대한 결과로 초반 2000 step 가량은 성공적으로 episode를 마치지 못했으나, 이후 학습이 이루어지면서 linear하게 episode가 증가하는 모습을 관찰할 수 있었다. Monte Carlo 기반의 방식이 windy grid world에서는 효율적인 학습 방법이 아님을 확인하는 실험을 추가적으로 진행해보고자 했으나 시간이 부족하여 해당 실험은 진행하지 못하였다.

Listing 7 WindyGridWorld Class: episode()

```
class WindyGridWorld:
    def episode(self, q_value):
        time = 0
        state = self.init_state
        if np.random.binomial(1, self.eps) == 1:
            action = np.random.choice(self.actions_lst)
        else:
            values_ = q_value[state[0], state[1], :]
            action  = np.random.choice([action_ for action_, value_ \
                                        in enumerate(values_)\
                                        if value_ == np.max(values_)])
        while state != self.goal:
            next_state = self.move(state, action)
            if np.random.binomial(1, self.eps) == 1:
                next_action = np.random.choice(self.actions_lst)
            else:
                values_ = q_value[next_state[0], next_state[1], :]
                next_action = np.random.choice([action_ for action_,\
                                                value_ in enumerate(values_)\
                                                 if value_ == np.max(values_)])
            q_value[state[0], state[1], action] += \
                self.alpha * (self.reward + q_value[next_state[0],\
                            next_state[1], next_action] -\
                            q_value[state[0], state[1], action])
            state = next_state
            action = next_action
            time += 1
        return time
```

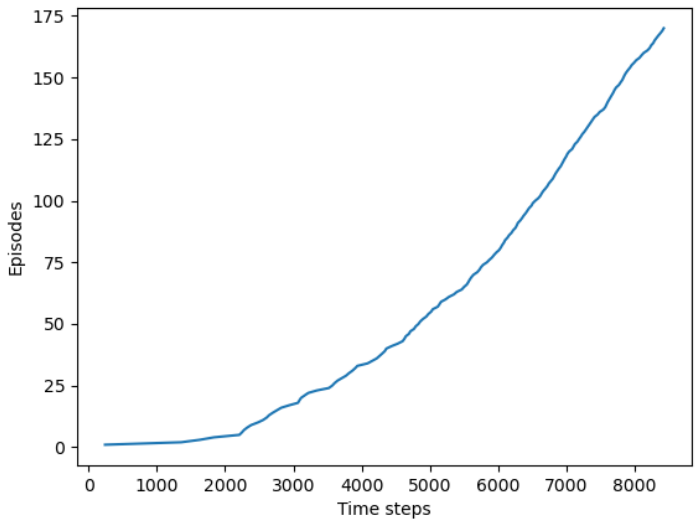


Figure 8: Results of Sarsa applied to the windy grid world.