김범수

Model Free Control

Model Free Control

By POSCAT

Contents

지난 내용 복습

- 지난 강의에서는 MDP와 이를 해결하는 방법 중 하나인 DP에 대해 학습하였습니다. 이 중, reward와 total reward에 대한 개념은 중요하므로 한 번 다시 복습하고 넘어가겠습니다.
- 시간 t 시점에 state s에서 받을 보상을 R_{t+1}이라 합니다. 이 때, R_{t+1}의 평균값을 $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$라고 표기합니다.
- 시간 t부터 게임이 끝날 때 까지 받는 총 보상(total reward)을 $G_t = R_{t+1} + R_{t+2} + \cdots$ 와 같이 표기할 수 있습니다. 그러나, 이와 같이 할 경우 G_t가 무한으로 발산할 위험이 있어 discount factor γ를 도입하여 $G_t = R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + \cdots$와 같이 표기합니다. 이 때, γ는 0이상 1이하의 값으로, 0에 가까울수록 현재의 보상을 중요시하고, 1에 가까울수록 미래의 보상을 중요시합니다.
- value function은 state를 input으로 받고, total reward의 평균값을 반환하는 함수입니다. 즉, $v(s) = \mathbb{E}[G_t|S_t = s]$로 정의합니다.

지난 내용 복습

- 지난 강의에서 우리의 목표는 $v(s)$를 찾는 것이었습니다. $v(s)$는 점화식의 형태로 정의할 수 있습니다.
- $v(s) = \mathbb{E}[G_t|S_t = s]$

  $= \mathbb{E}[R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + \cdots |S_t = s]$

  $= \mathbb{E}[R_{t+1}|S_t = s] + \gamma * \mathbb{E}[R_{t+2} + \gamma * R_{t+3} + \cdots |S_t = s]$

  $= \mathcal{R}_s + \gamma * \sum s' \in \mathcal{S}\, \mathbb{P}[S_{t+1} = s'|S_t = s]\mathbb{E}[R_{t+2} + \gamma * R_{t+3} + \cdots |S_{t+1} = s']$

  $= \mathcal{R}_s + \gamma * \sum s' \in \mathcal{S}\, \mathcal{P}_{ss'} v(s')$
- 우리는 위 식을 방정식의 형태로 직접 풀거나, DP처럼 점화식으로 사용해서 value function을 구하는 법을 배웠습니다.

Model Free Control

- 지금까지는 $\mathcal{R}$과 $\mathcal{P}$를 모두 안다고 가정하고 문제를 해결하였습니다. 즉, model을 완전히 알고 있다고 가정하고 문제를 풀었습니다. 하지만 대부분의 경우는 model의 전체 구조를 파악하지 못하는 경우가 대부분입니다.
- 바둑의 경우 10^160개의 state를 가지고 있다고 합니다. 이처럼 state의 수가 너무 많은 경우, 우리는 model을 완전히 파악할 수 없습니다.
- 슈퍼마리오 게임을 처음 플레이할 때, 물음표 상자에서 무슨 아이템이 나올 지 알 수 없습니다. 100% 확률로 코인이 나올 지, 100% 확률로 아이템이 나올 지, 아니면 반반 확률로 코인과 아이템이 나올 지 알 수 없습니다. 이처럼 state의 전이 확률 $\mathcal{P}$을 정확히 파악할 수 없는 경우가 발생할 수 있습니다.
- 처음 보는 룰렛 게임을 할 때, 내가 받을 수 있는 보상의 기댓값을 알 수 없습니다. 즉, $\mathcal{R}$를 정확히 파악할 수 없는 경우가 발생할 수 있습니다.
- 이외에도 우리가 접하는 많은 문제는 model을 완전히 파악하지 못하는 경우가 대부분입니다.

Model Free Control

- 저번 강의에서 배운 DP를 사용하기 위해서는 model을 완전히 파악해야 합니다. 따라서 DP는 model-based control이라 합니다. 이번 강의에 배울 것은 model-free control, 즉 model 전체를 완전히 파악하지 않고 optimal policy를 계산하는 방법에 대해 배워볼 것입니다.

- DP에서는 value function $v(s)$를 사용하여 아래와 같이 policy를 improve 하였습니다.

  $\pi(s) = argmax_{a\in\mathcal{A}}\mathcal{R}_s^a + \mathcal{P}_{ss'}^a v(s')$

  즉, 미래의 보상을 최대화하는 action을 선택하는 것으로 policy를 정의했습니다. 그러나, model free control에서는 $\mathcal{P}, \mathcal{R}$을 알지 못하므로 위 식을 사용하지 못합니다.

- 위 문제를 해결하기 위해서 우리는 value function을 다시 정의합니다.

  $q(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a]$, 즉 state s에서 action a를 취하였을 때의 total reward로 정의합니다.

  그렇다면, policy는 아래와 같이 간단하게 정의할 수 있습니다.

  $\pi(s) = argmax_{a\in\mathcal{A}}q(s,a)$

Model Free Control

- 오늘 배울 Model Free Control은 두 과정을 반복하며 진행됩니다.
  1. 먼저 실제로 주어진 환경에서 게임을 진행합니다. 그 과정에서 받는 state s, action a, 그리고 reward r을 기록합니다.
  2. 그 후에는 기록한 결과를 이용하여 $q(s,a)$의 값을 계산합니다.
- 게임을 하는 방법은 크게 2가지로 나뉩니다. 지금까지 본 것 중에서 가장 좋은 action을 취하는 방법과 새로운 action을 취하는 방법으로 나뉩니다. 지금까지의 정보로 가장 좋은 action을 취하는 것을 exploitation, 새로운 action을 취하는 것을 exploration이라 나뉩니다.
- 만약 당신의 앞에 슬롯머신 2개가 주어져 있다고 가정합시다. 1번 슬롯머신은 1000~10000원을 균일한 확률로 주고, 2번 슬롯머신은 2000원~3000원을 균일한 확률로 줍니다. 만약 두 슬롯머신을 돌려서 각각 1000원, 3000원을 얻었다고 가정해보죠. 만약 exploitation만 한다면, 당신은 더 많은 보상을 얻었던 2번 슬롯머신만 돌리게 될 것입니다. 그에 반해 exploration만 한다면 1번 슬롯머신과 2번 슬롯머신을 둘 다 계속 돌리게 될 것입니다. 따라서 exploration과 exploitation을 적절히 조화하여 여러 번 각 슬롯머신을 돌려서 1번 슬롯머신이 돈을 더 준다는 것을 확인하고, 1번 슬롯머신만 돌리는 전략을 취해야 합니다.

Model Free Control

- 지금까지 우리는 greedy policy, 즉 단순히 가장 좋은 action을 선택하는 policy를 사용하였습니다. 하지만, 이는 exploitation만 사용하게 된다는 단점이 있습니다. 따라서 우리는 ϵ- greedy policy를 사용할 것입니다.
- ϵ- greedy policy는 ϵ의 확률로 random한 action을 선택하고, 1- ϵ의 확률로 greedy action을 선택하는 policy 입니다. m개의 action이 존재할 때, policy를 수식으로 표현하면 아래와 같습니다.

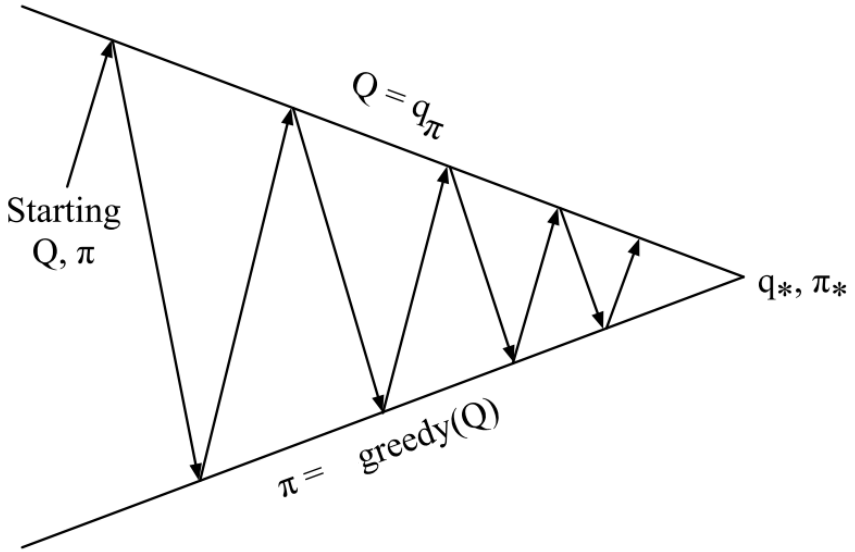
$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon \; if \; a = \; argmax_{a' \in \mathcal{A}} \, q(s, a') \\ \frac{\epsilon}{m} \qquad\qquad otherwise \end{cases}$$

- ϵ값은 상수로 지정할 수도 있고, 게임이 진행될수록 점점 감소하는 값으로 지정할 수도 있습니다.

Model Free Control

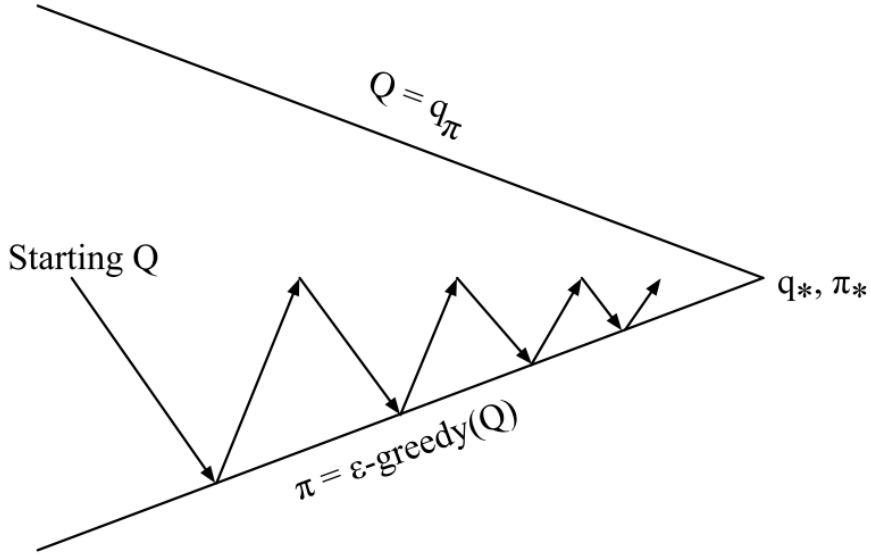
- DP에서는 policy evaluation 과정에서 점화식을 매우 많이 반복하여 value function에 매우 가까운 값을 구하는 것이 목표였습니다. 그 후, value function을 이용하여 새로운 policy를 도출하고, 그 policy를 이용하여 새로운 value function을 구하였습니다. 이 과정을 그림으로 표현하면 아래처럼 표현할 수 있습니다.

- 위쪽으로 향하는 화살표는 policy를 이용해 value function을 구하는 과정을 의미하고, 아래쪽으로 향하는 화살표는 value function을 이용해 새로운 policy를 도출하는 과정을 의미합니다. 두 과정을 계속 반복하여, optimal value function과 optimal policy를 구하는 과정을 표현하고 있습니다.



Model Free Control

- Model Free control에서도 마찬가지로, 게임을 무한히 반복하여 value function을 구하고, value function을 이용하여 새로운 policy를 도출할 수 있습니다. 하지만, 보다 빠르게 계산하기 위해 정확한 value function을 구하지 않고, 게임 한 번 진행할 때마다 새로운 policy를 도출합니다.
- 게임을 1번 진행합니다. 이 때 게임은 ϵ- greedy policy 를 따라서 진행합니다. 게임 1회는 (s1, a1, r2, s2, a2, …) 으로 구성되어 있습니다.
- 게임 진행 과정을 이용하여 value function을 이용합니다. 오른쪽 그림에서 위쪽 화살표에 해당합니다.
- 새로운 ϵ- greedy policy 를 도출합니다. 오른쪽 그림에서 아래쪽 화살표에 해당합니다.



Monte-Carlo

- 그렇다면 value function은 어떻게 계산할 수 있을까요? 가장 간단한 방법은 게임을 통해 얻은 total reward의 평균을 계산해서 value function을 어림잡는 법입니다. 이러한 방법을 Monte-Carlo(MC) algorithm이라 합니다.
- 지금까지 state s에서 action a를 취한 횟수를 N(s, a), 그리고 value function을 Q(s, a)라고 하겠습니다.
- 이번 게임에서 state s에 방문하고, action a를 또 한 번 취하고 그 결과 G의 total reward를 받았다고 가정합니다. 그렇다면 새로운 평균은 아래와 같이 구할 수 있습니다.
- 먼저 방문 횟수를 1회 증가시킵니다. N(s, a) <- N(s, a) + 1
- 이번 게임을 제외한 total reward의 총합은 (N(s, a) – 1) * Q(s, a)가 됩니다. 따라서 이번 게임을 포함한 total reward의 총합은 (N(s, a) – 1) * Q(s, a) + G가 되고, 새로운 total reward의 평균은
  ((N(s, a) – 1) * Q(s, a) + G) / N(s, a) = Q(s, a) + (G – (Q(s, a)) / N(s, a) 가 됩니다.
- 게임을 무한히 진행하면, 우리가 계산한 Q(s, a)는 실제 G(s, a)값에 가까워집니다.

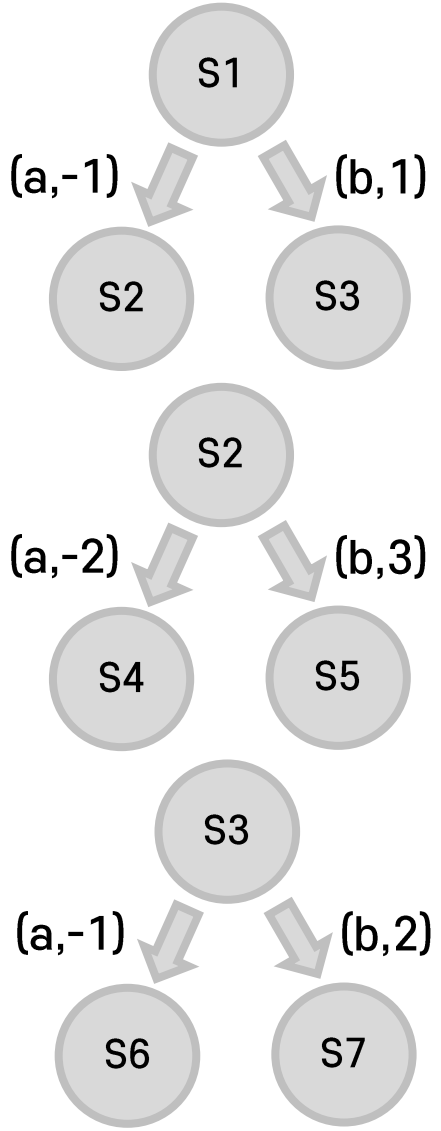
Monte-Carlo

Code Explanation

```
Initialize N(s, q) = 0 for all s∈ 𝒮 and a∈ 𝒜
Repeat (for each episode):
    Initialize state S and memory N = {S}
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e-greedy)
        Take action A and observe R, S'
        Add A, R, S' to memory N
        S <- S'
    until S is terminal
# N = {S1, A1, R2, S2, … ST}
    for each state St and At in the episode:
        Gt = Rt+1 + r * Rt+2 + r^2 * Rt+3 + …
        N(St, At) <- N(St, At) + 1
        Q(St, At) <- Q(St, At) + (Gt – Q(St, At)) / N(St, At)
```

Monte-Carlo

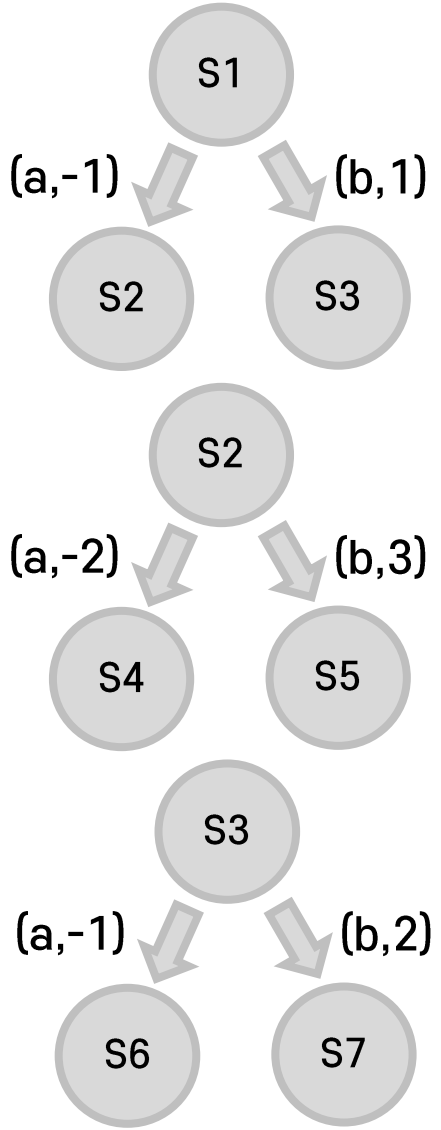
- 오른쪽의 간단한 예제를 고려해봅시다. Discount factor는 1이라 가정합니다.
- 초기 Q(s, a)는 임의의 값으로, N(s, a)는 0으로 초기화합니다. (여기서는 Q(s, a)를 0으로 초기화하였습니다.)
- S4~7은 terminal state라고 가정합니다.
- action과 reward는 (action, reward) 꼴로 표시하였습니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
N(s, a)	0	0	0	0	0	0
Q(s, a)	0	0	0	0	0	0

Monte-Carlo

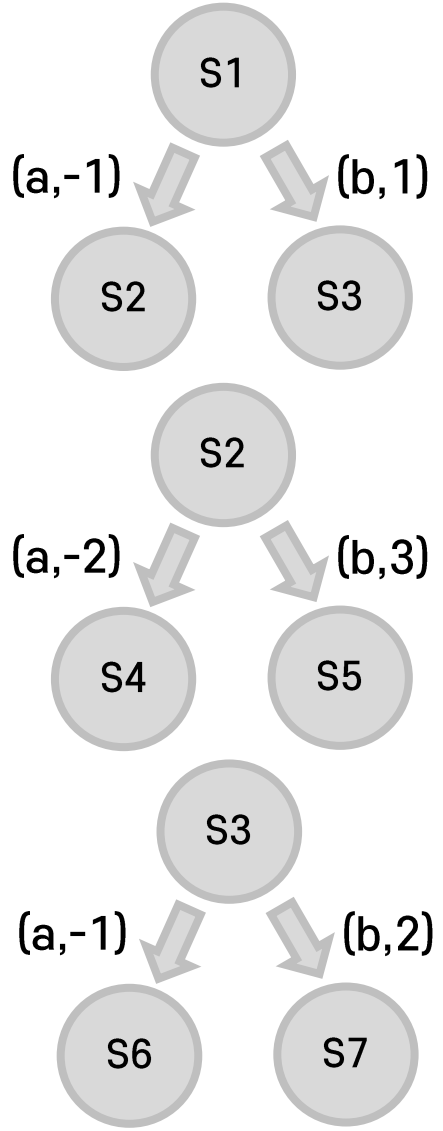
- 첫 번째 게임으로 (S1, a, S2, a, S4)라는 게임이 진행되었다고 가정합니다.
- S2에서는 -2의 보상을 받았고, S1에서는 총 -1 -2 = -3의 보상을 받았습니다.
- 따라서 아래와 같이 N, Q가 update됩니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
N(s, a)	1	0	1	0	0	0
Q(s, a)	-3	0	-2	0	0	0

Monte-Carlo

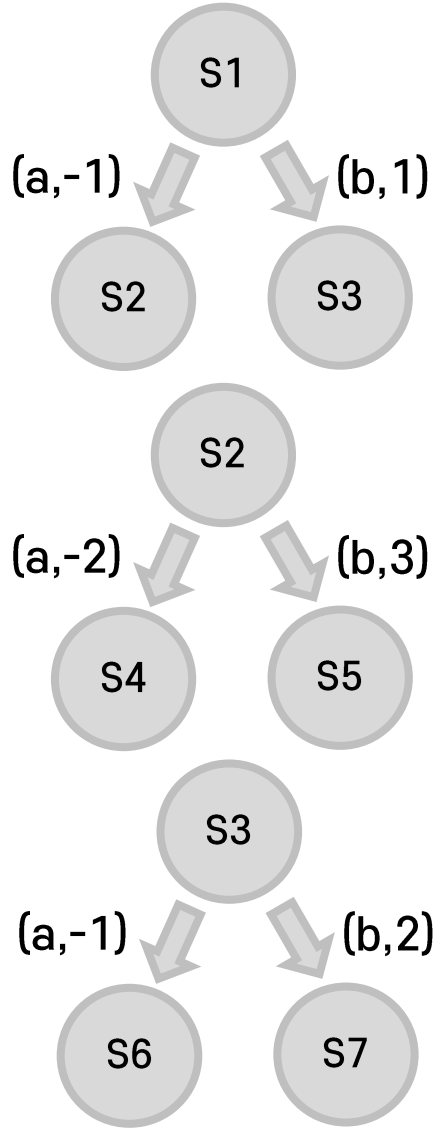
- 두 번째 게임으로 (S1, a, S2, b, S5)라는 게임이 진행되었다고 가정합니다.
- S2에서는 3의 보상을 받았고, S1에서는 총 -1 + 3 = 2의 보상을 받았습니다.
- Q(S1, a)는 -3과 2의 평균인 -0.5로 update 되었음을 알 수 있습니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
N(s, a)	2	0	1	1	0	0
Q(s, a)	-0.5	0	-2	3	0	0

Monte-Carlo

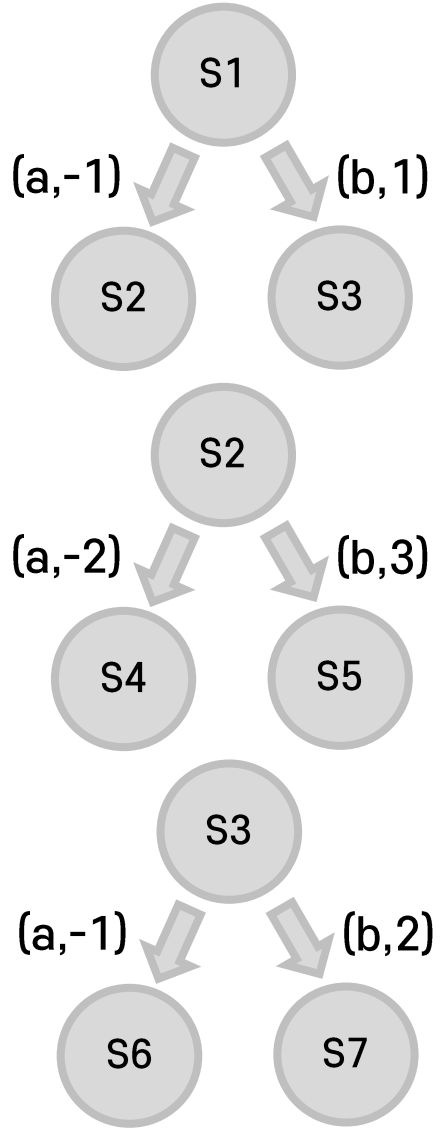
- 세 번째 게임으로 (S1, b, S3, a, S5)라는 게임이 진행되었다고 가정합니다.
- S3에서는 2의 보상을 받았고, S1에서는 총 1 + 3 = 4의 보상을 받았습니다.
- 따라서 아래와 같이 N, Q가 update됩니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
N(s, a)	2	1	1	1	1	0
Q(s, a)	-0.5	4	-2	3	2	0

Monte-Carlo

- 네 번째 게임으로 (S1, a, S2, b, S5)라는 게임이 진행되었다고 가정합니다.
- S2에서는 3의 보상을 받았고, S1에서는 총 -1 + 3 = 2의 보상을 받았습니다.
- 따라서 Q(S1, a) = -0.5 + (2 – (-0.5)) / 3 = 1/3로 update됩니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
N(s, a)	3	1	1	1	1	0
Q(s, a)	1/3	4	-2	3	2	0

Sarsa

- MC algorithm은 variance가 크다는 단점이 존재합니다. Q(s, a)는 현재의 action 뿐만이 아니라 그 이후의 모든 action에 영향을 받기 때문에 variance가 매우 커집니다. 따라서 Q(s, a)가 현재의 action에만 영향을 받고, 그 이후의 action에는 영향을 받지 않도록 하여 variance를 줄이는 기법 Sarsa가 도입됩니다. Sarsa에서는 아래 식으로 value function을 update합니다.
- Q(S, A) <- Q(S, A) + α * (R + γ * Q(S', A') – Q(S, A))
- 현재 state S에서 action A를 취한 후, 보상 R을 받고 state S'으로 이동했을 때, S'에서 취할 action A'을 도출합니다.
- 그렇다면, Q(S, A)는 현재 받을 보상 R과 미래에 받을 보상 Q(S', A')의 식으로 도출할 수 있습니다.
  그 식은 R + γ * Q(S', A') 가 됩니다.
- R + γ * Q(S', A') – Q(S, A))는 우리가 도출한 Q(S, A)값과 실제 Q(S, A)값의 차를 의미합니다. 이 값을 바로 더해주면 한 게임이 진행될 때마다 Q(S, A)가 휙휙 바뀌기 때문에 Q(S, A)가 불안정해집니다. 따라서 차에 alpha를 곱해 Q(S, A)가 R + γ * Q(S', A')로 조금 가까워지도록 Q(S, A)값을 update합니다.
- alpha값은 학습률(learning rate)이라 불리는 상수입니다. alpha 값이 클 수록 학습 속도가 빠르지만, value function이 최근의 게임에 많이 영향을 받기 때문에 불안정해집니다. 따라서 적절한 alpha값의 선택이 필요합니다.
- 한 게임이 끝날 때마다 계산한 MC와는 달리, Sarsa에서는 한 action마다 계산을 진행합니다.
- MC와 마찬가지로 무한히 게임을 진행하면 우리가 계산한 Q값은 실제 Q값에 가까워집니다.
- MC에 비해 variance가 작다는 장점이 있지만, Q(S', A')에 오차가 생길 경우, Q(S, A)에도 그 오차가 반영된다는 단점이 있습니다.

Sarsa

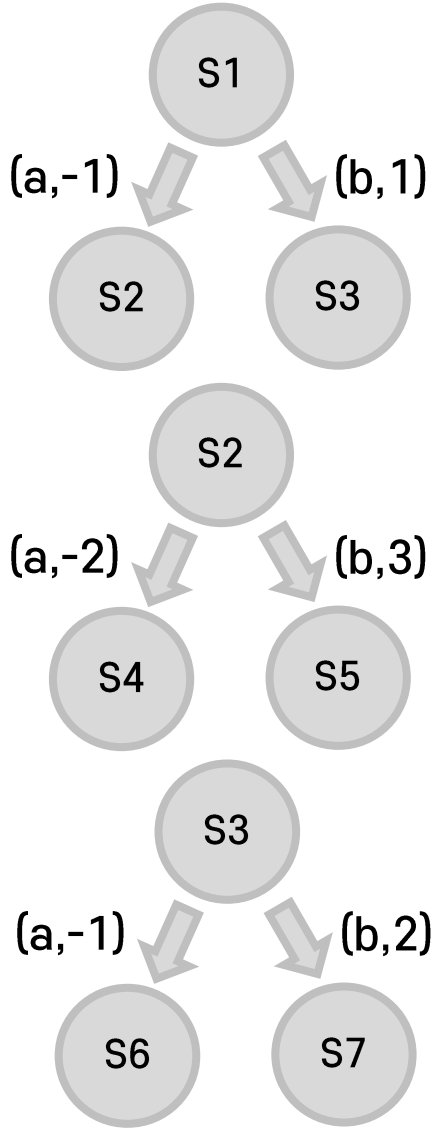
Code Explanation

```
Initialize Q(s, a) for all s∈ 𝒮 and a∈ 𝒜, arbitrarily and Q(terminal state, ) = 0
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e-greedy)
        Q(S, A) <- Q(S, A) + alpha * [R + gamma * Q(S', A') – Q(S, A)]
        S <- S', A <- A'
    until S is terminal
```

Sarsa

- 전의 예제를 Sarsa로 풀어봅시다. 마찬가지로 Discount factor는 1이라 가정합니다.
- 초기 Q(s, a)는 임의의 값으로 초기화합니다. (여기서는 Q(s, a)를 0으로 초기화하였습니다.)
- S4~7은 terminal state라고 가정합니다.
- action과 reward는 (action, reward) 꼴로 표시하였습니다.
- 학습률 alpha = 0.1이라 가정합니다.

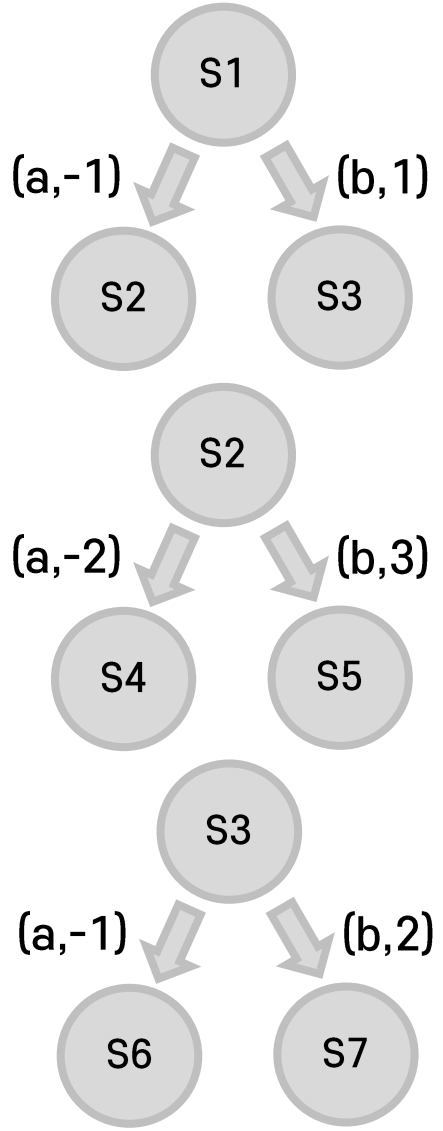
State	S1		S2		S3	
Action	a	b	a	b	a	b
Q(s, a)	0	0	0	0	0	0



Sarsa

- 초기 state S1, Action a로 초기화합니다.
- Action a를 취하고, reward R = -1과 S' = S2를 관찰합니다.
- S2에서는 b를 play한다고 가정합니다.
- 그렇다면, Q(S1, a) = Q(S1, a) + 0.1 * [R + Q(S2, b) – Q(S1, a)]
  = 0 + 0.1 * [-1 + 0 - 0] = -0.1로 갱신됩니다.

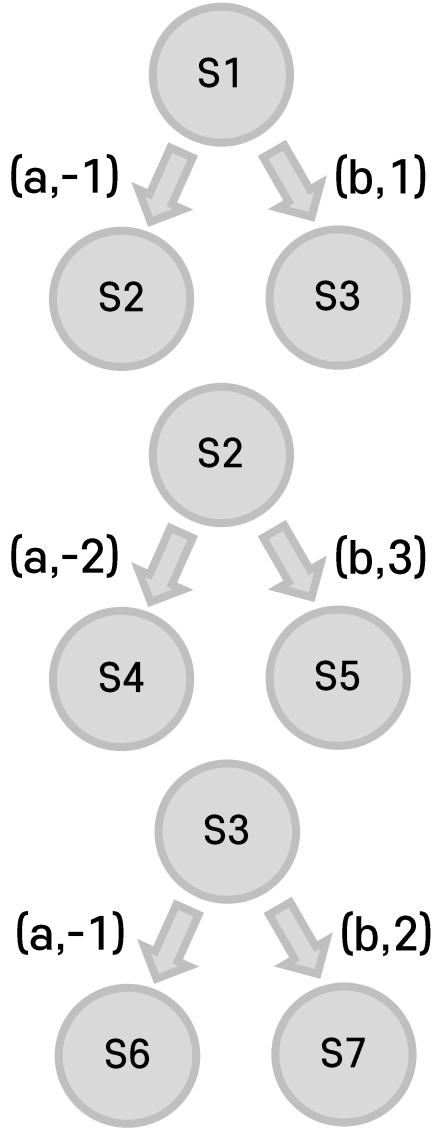
State	S1		S2		S3	
Action	a	b	a	b	a	b
Q(s, a)	-0.1	0	0	0	0	0



Sarsa

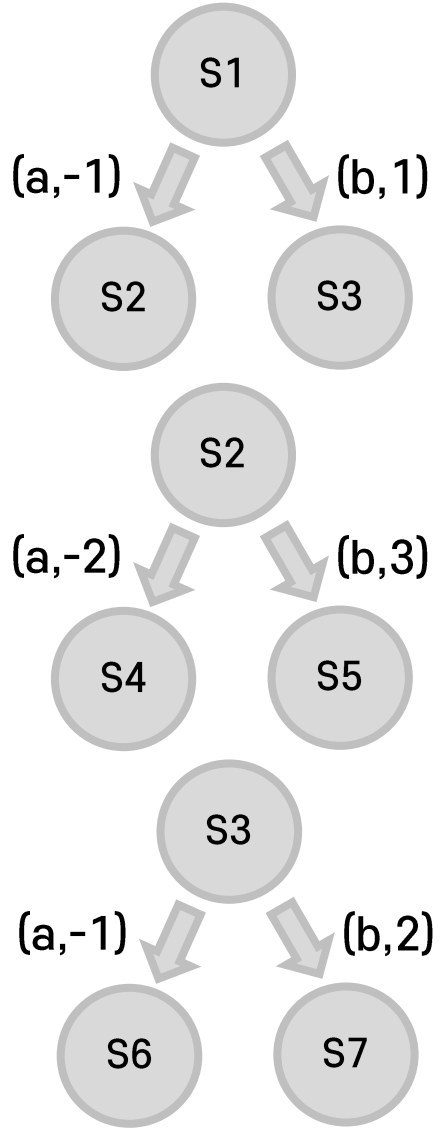
- state S2에서 action b를 취하고 R = 3, S' = S5를 관찰합니다.
- S5는 terminal state이므로 action을 취하지 않습니다.
- 그렇다면, Q(S2, b) = Q(S2, b) + 0.1 * [R + Q(S5,) – Q(S2, b)]
  = 0 + 0.1 * [3 + 0 - 0] = 0.3로 갱신됩니다.
- 이렇게 하나의 게임이 종료됩니다.

State	S1		S2		S3	
Action	a	b	a	b	a	b
Q(s, a)	-0.1	0	0	0.3	0	0



Sarsa

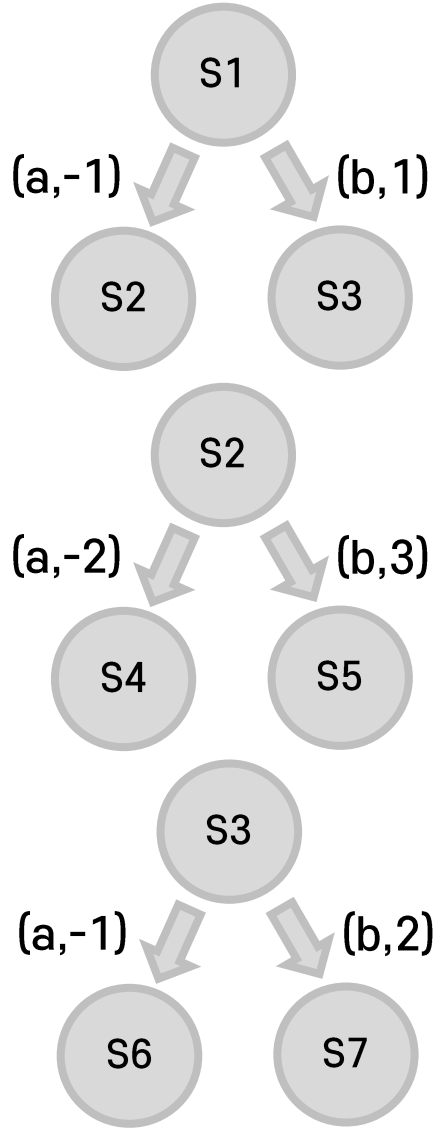
- 두 번째 게임도 첫 번째 게임과 똑같이 행동했다고 가정해봅시다.
- 먼저 S1에서 a를 취하고, S2로 이동해 b를 취할 예정이라고 합시다.
- 그렇다면 Q(S1, a) = Q(S1, a) + 0.1 * [R + Q(S2, b) – Q(S1, a)]
  = -0.1 + 0.1 * [-1 + 0.3 – (-0.1)] = -0.16 으로 update됩니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
Q(s, a)	-0.16	0	0	0.3	0	0

Sarsa

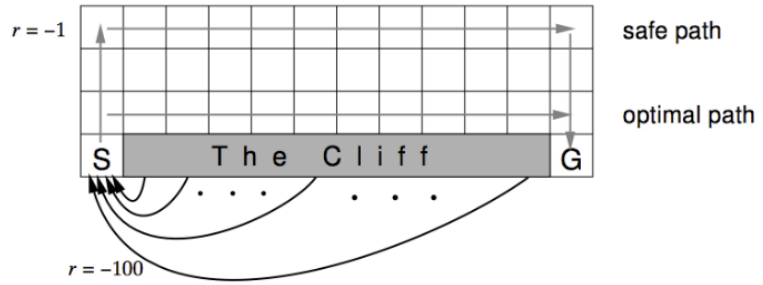
- S2에서 b를 취하면 S5로 이동하게 됩니다.
- 마찬가지로, Q(S2, b) = Q(S2, b) + 0.1 * [R + Q(S5,) – Q(S2, b)]
  = 0.3 + 0.1 * [3 + 0 – 0.3] = 0.57로 update됩니다.



State	S1		S2		S3	
Action	a	b	a	b	a	b
Q(s, a)	-0.16	0	0	0.57	0	0

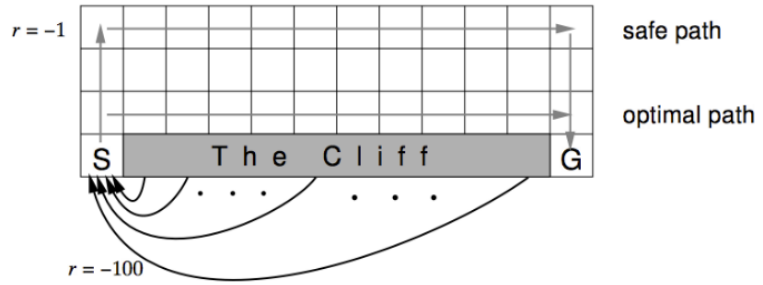
Q-learning

- Sarsa에도 단점이 존재합니다. 오른쪽과 같은 Cliff walking 게임을 고려해봅시다.
- 시작지점 S에서 시작해 도착지점 G로 가는 것이 이 게임의 목표입니다.
- 상하좌우로 1칸 이동할 때마다 -1의 보상을 받게 됩니다.
- 만약 회색으로 표시된 절벽 부분으로 이동한다면, -100의 보상을 추가로 받고 S로 이동하게 됩니다.
- Sarsa 알고리즘으로 Cliff walking 문제를 푼다면 어떻게 될까요?



Q-learning

- S 지점을 (x, y) = (0, 0)이라 하고, (1, 1)의 value function을 고려해봅시다.
- Q((1, 1), down)은 절벽으로 인해 매우 작은 값으로 update될 것입니다.
- Q((0, 1), right)이 어떤 값으로 갱신될 지 생각해봅시다. 먼저 (0, 1)에서 오른쪽으로 이동하면 (1, 1)이 되고, (1, 1)에서는 e-greedy를 사용해서 작은 확률로 아래로 간다는 action을 선택할 것입니다. 즉 Q((0, 1), right)을 R + Q((1, 1), down) 값으로 update합니다. 따라서 Q((0, 1), right)도 Q((0, 1), up)에 비해 상대적으로 작은 값이 됩니다.



- 따라서 절벽 쪽으로 향하는 action은 상대적으로 작은 Q값을 가지게 됩니다. 따라서 Sarsa 알고리즘을 사용하면 우리가 원하는 optimal path가 아닌 safe path를 발견하게 됩니다.
- 이러한 문제를 해결하기 위해 Q(S', A')이 아닌 max Q(S', A')로 update하는 기법인 Q-learning이 사용됩니다.

Q-learning

Code Explanation

```
Initialize Q(s, a) for all s∈ 𝒮 and a∈ 𝒜, arbitrarily and Q(terminal state, ) = 0
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e-greedy)
        Take action A, observe R, S'
        Q(S, A) <- Q(S, A) + alpha * [R + gamma * maxa Q(S', a)) – Q(S, A)]
        S <- S'
    until S is terminal
```

마무리

- 이번 강의에서는 model-free control을 사용하여 문제를 해결해보았습니다. value function을 v(s)가 아닌 q(s, a) 형태로 변경하여 model을 알지 못하더라도 value function을 계산할 수 있도록 하였습니다.
- MC 알고리즘에서는 단순히 지금까지 구했던 total reward의 평균값으로 value function을 정의하였습니다.
- Sarsa 알고리즘에서는 MC에서 variance가 크다는 단점을 보완하여 R + Q(S', A')으로 Q(S, A)값을 조금 움직이는 방법을 사용하였습니다. 하지만 이 방법은 Q(S', A')에 오차(bias)가 Q(S, A)에 반영되므로, variance는 작지만 bias는 크다는 특징이 있습니다.
- Q-learning에서는 Sarsa의 단점을 보완하여 target을 R + Q(S', A')에서 R + max Q(S', a)로 바꾸었습니다.
- 위의 알고리즘에서 action을 선택할 때는 e의 확률로 랜덤 행동을 취하는 e-greedy policy를 사용합니다. 이 때, e가 게임이 진행될수록 점점 감소되는 값으로 선정하면 위의 세 알고리즘에서 계산하는 value function 모두 optimal value function으로 수렴합니다. 단 수렴하는 속도에는 차이가 있습니다.

연습문제

- cliff walking 예제를 Sarsa와 Q-learning을 이용해서 해결해봅시다.
- 3강과 마찬가지로 아래의 링크에서 cliff.py를 받아서 저장합니다.
- cliff.py의 Cliff class에는 cliff walking 게임이 저장되어 있습니다.
- env = Cliff(x, y)의 형태로 cliff walking 게임을 가져올 수 있습니다. x는 가로 길이, y는 세로 길이로, x, y는 2 이상이어야 합니다.
- Cliff class의 reset 함수는 게임을 초기 상태로 초기화하고, 초기 state를 반환합니다. 게임 시작 전에 반드시 한 번 사용해야 합니다.
- Cliff class의 step 함수는 매개변수로 action을 받아 action을 수행합니다. 그 후, next state, reward, 종료 여부 를 반환합니다. action은 0이상 4미만의 정수입니다.
- 0 : 위쪽, 1 : 오른쪽, 2 : 아래쪽, 3 : 왼쪽 에 대응됩니다.
- Cliff class의 render함수는 현재 상태를 출력합니다.

연습문제

- Cliff class의 세 함수는 오른쪽처럼 사용할 수 있습니다.
- render 함수를 사용하면 아래처럼 출력이 됩니다. S는 시작점, E는 도착점, X는 현재 위치, @는 cliff를 의미합니다.

```
env = Cliff(size_x, size_y)
```

```
# initialize state
state = env.reset()
```

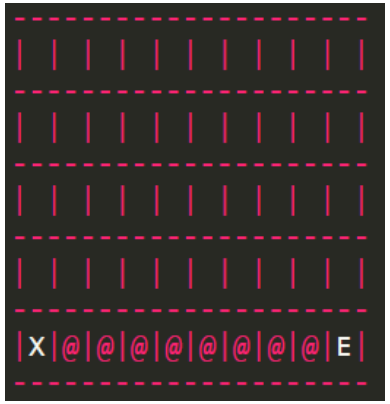
```
# Take action A, observe R, S'
next_state, reward, done = env.step(action)
```

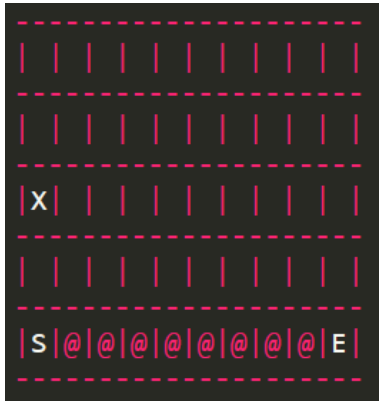
```
env.render()
```

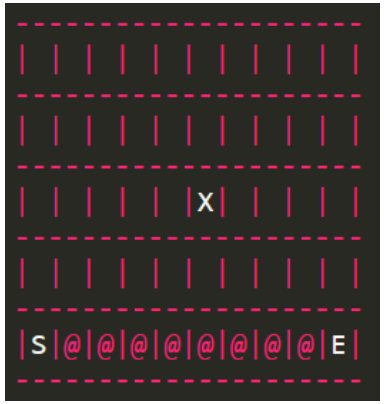
```
----------------------
| | | | | | | | | | |
----------------------
| | | | | | | | | | |
----------------------
| | | | | | | | | | |
----------------------
| | | | |X| | | | | |
----------------------
|S|@|@|@|@|@|@|@|@|E|
----------------------
```

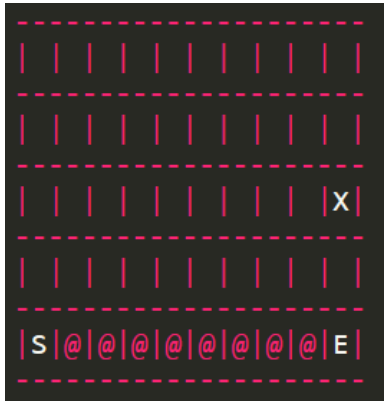
연습문제

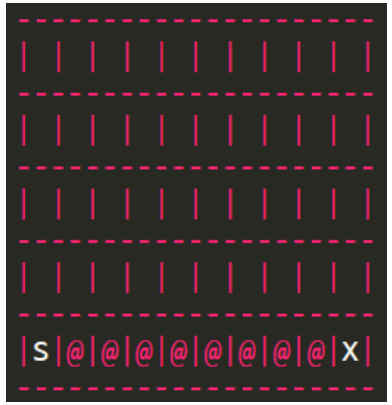
- Sarsa 알고리즘으로 학습을 할 경우(epsilon = 0.05 로 고정) 아래처럼 안전한 경로로 이동하는 것을 볼 수 있습니다.











연습문제

- Q-learning으로 학습을 할 경우(epsilon = 0.05 로 고정) 아래처럼 최적의 경로로 이동하는 것을 볼 수 있습니다.