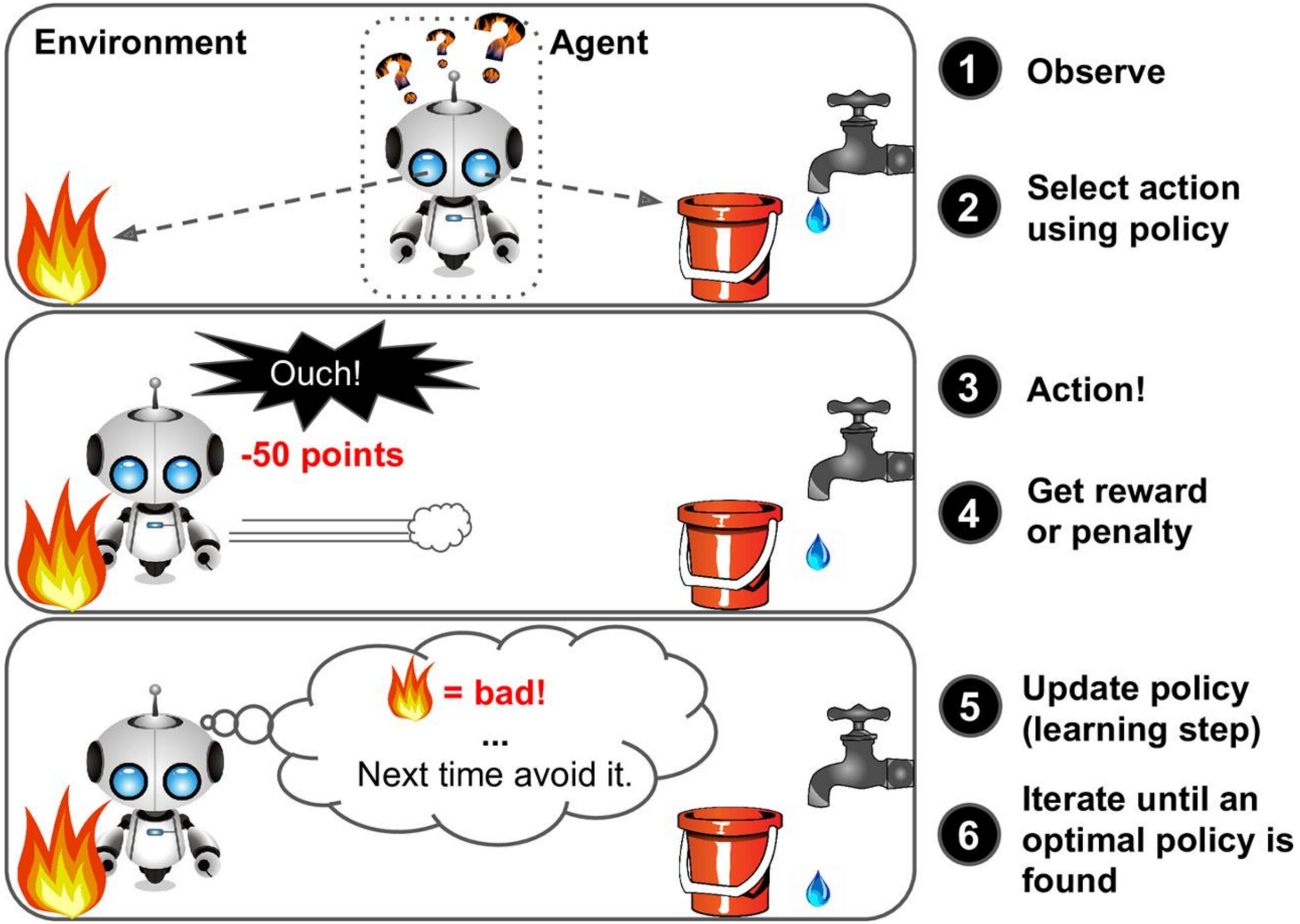
강화학습

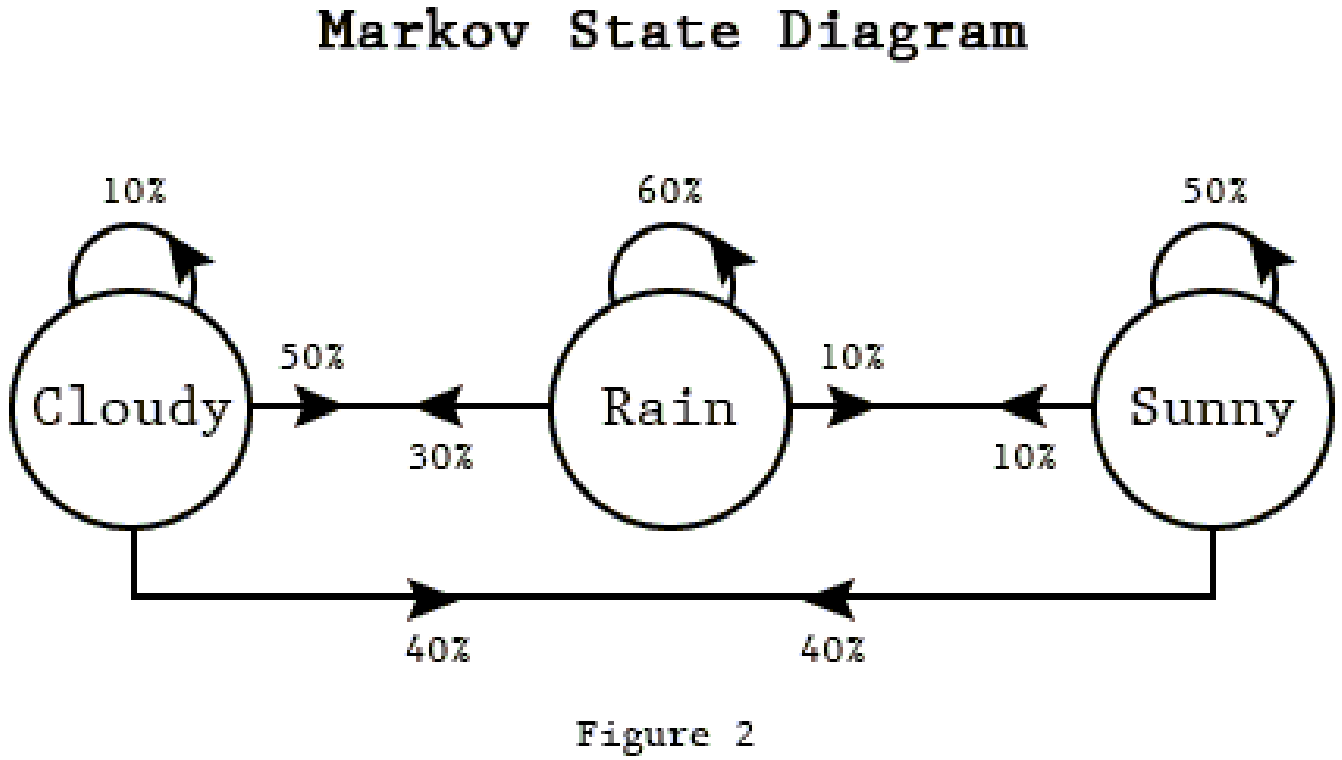
강화학습



Environment
Agent
Ouch!
-50 points
= bad!
...
Next time avoid it.
1 Observe
2 Select action using policy
3 Action!
4 Get reward or penalty
5 Update policy (learning step)
6 Iterate until an optimal policy is found

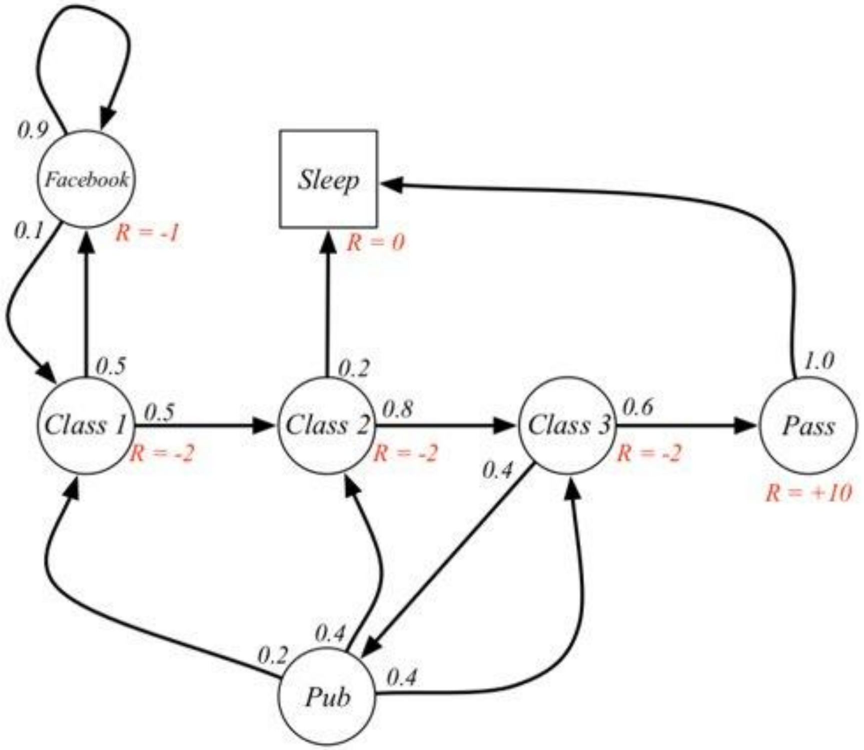
MP(Markov Process)

- 시간이 지남에 따라 s에서s'으로 전환



MDP(Markov Decision Process)

- 상태의 이동 제약 조건에 액션이 추가 됨(MP+action)
- 상태와 행동에 따라 변화 확률과 보상이 결정
- Action을 취할 때 reward가 주어짐



정책(policy)

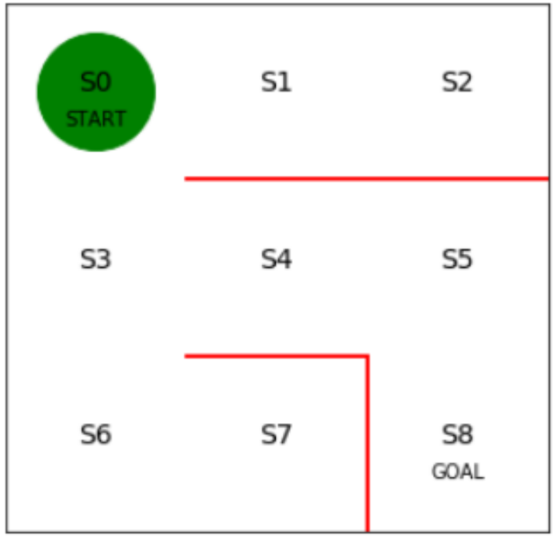
- agent가 어떻게 행동할지 결정하는 규칙
- 상태 s에서 행동 a를 선택할 확률은 θ에 의해 결정하는 정책 파이에 따름

$$정책 = \pi_\theta(a\,|\,s)$$

- 정책은 (state, action) 테이블, 함수, 신경망으로 이해할 수 있음

초기 정책 파라미터 θ

- 가능한 방향에 1, 벽 때문에 이동할 수 없으면 nan




```
#                          상,   우,  하,   좌
theta_0 = np.array([[np.nan, 1, 1, np.nan],  # s0
                    [np.nan, 1, np.nan, 1],  # s1
                    [np.nan, np.nan, np.nan, 1],  # s2
                    [1, 1, 1, np.nan],  # s3
                    [np.nan, 1, np.nan, 1],  # s4
                    [np.nan, np.nan, 1, 1],  # s5
                    [1, 1, np.nan, np.nan],  # s6
                    [np.nan, np.nan, np.nan, 1],  # s7、※s8은 목표지점
                    ])
```

θ로부터 정책 변환

```python
def softmax_convert_into_pi_from_theta(theta):
    '''비율 계산에 소프트맥스 함수 사용'''

    beta = 1.0
    [m, n] = theta.shape  # theta의 행렬 크기를 구함
    pi = np.zeros((m, n))

    exp_theta = np.exp(beta * theta)  # theta를 exp(theta)로 변환

    for i in range(0, m):
        # pi[i, :] = theta[i, :] / np.nansum(theta[i, :])
        # 단순 비율을 계산하는 코드

        pi[i, :] = exp_theta[i, :] / np.nansum(exp_theta[i, :])
        # softmax로 계산하는 코드

    pi = np.nan_to_num(pi)  # nan을 0으로 변환

    return pi
```

```python
softmax_convert_into_pi_from_theta(theta_0)
```

```
array([[0.        , 0.5       , 0.5       , 0.        ],
       [0.        , 0.5       , 0.        , 0.5       ],
       [0.        , 0.        , 0.        , 1.        ],
       [0.33333333, 0.33333333, 0.33333333, 0.        ],
       [0.        , 0.5       , 0.        , 0.5       ],
       [0.        , 0.        , 0.5       , 0.5       ],
       [0.5       , 0.5       , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 1.        ]])
```

정책에 따른 행동 선택

- 현재 상태 s에서 정책에 따라 action 선택해서 s_next로 이동

```python
def get_action_and_next_s(pi, s):
    direction = ["up", "right", "down", "left"]
    # pi[s,:]의 확률을 따라, direction값이 선택된다
    next_direction = np.random.choice(direction, p=pi[s, :])

    if next_direction == "up":
        action = 0
        s_next = s - 3  # 위로 이동하면 상태값이 3 줄어든다
    elif next_direction == "right":
        action = 1
        s_next = s + 1  # 오른쪽으로 이동하면 상태값이 1 늘어난다
    elif next_direction == "down":
        action = 2
        s_next = s + 3  # 아래로 이동하면 상태값이 3 늘어난다
    elif next_direction == "left":
        action = 3
        s_next = s - 1  # 왼쪽으로 이동하면 상태값이 1 줄어든다

    return [action, s_next]
```

정책을 사용해 미로 탈출

```
def goal_maze_ret_s_a(pi):
    s = 0  # 시작 지점
    #s_a_history = [[0, np.nan]]  # 에이전트의 행동 및 상태의 히스토리를 기록하는 리스트
    s_a_history = []

    while (1):  # 목표 지점에 이를 때까지 반복
        [action, next_s] = get_action_and_next_s(pi, s)

        s_a_history.append([s, action, direction[action], next_s])

        if next_s == 8:  # 목표 지점에 이르면 종료
            s_a_history.append([8, np.nan])
            break
        else:
            s = next_s

    return s_a_history
```

정책을 사용해 미로 탈출

```
# 정책 수정 없이 현재 목적지 까지 가기
# 같은 정책이라도 랜덤하게 확률적으로 이동하기 때문에 목표까지 가는 단계는 항상 다르다.
pi_0 = softmax_convert_into_pi_from_theta(theta_0)
s_a_history = goal_maze_ret_s_a(pi_0)
print(s_a_history)
print("목표 지점에 이르기까지 걸린 단계 수는 " + str(len(s_a_history) - 1) + "단계입니다")
```

```
[[0, 'down', 3], [3, 'right', 4], [4, 'left', 3], [3, 'right', 4], [4, 'right', 5], [5, 'down', 8]]
목표 지점에 이르기까지 걸린 단계 수는 5단계입니다
```

경사법을 이용한 정책 변경

- 행동을 많이 하면 보상을 강화시킴
- 기회가 작은데도 선택됐다면 작은 값을 빼고, 기회가 큰 경우 큰 값을 빼서, 확률이 작은데도 선택됐다면 보상을 강화함

$$\theta_{s_i,a_j} = \theta_{s_i,a_j} + \eta \cdot \Delta\theta_{s,a_j}$$

$$\Delta\theta_{s,a_j} = \{N(s_i,a_j) + P(s_i,a_j)N(s_i,a)\}/T$$

```
delta_theta[i, j] = (N_ij - pi[i, j] * N_i) / T
```

경사법을 이용한 정책 변경

- Nij = 300, Ni = 500, pij = 0.3
  - 경사 = 300-0.3*500 = 150 , 정책변경 거의 없음

- Nij = 100, Ni = 500, pij = 0.3
  - 경사 = 200-0.3*500 = 50 , 정책변경 거의 없음

- Nij = 100, Ni = 300, pij = 0.1
  - 경사 = 100 – 0.1*300= 70, 정책변경 큼

- Nij = 100, Ni = 99, pij = 0.99
  - 경사 = 100-0.99*99 = 1.98 , 정책변경 거의 없음

- 수렴하게 되면 정책 변경은 일어나지 않음

경사법을 이용한 정책 수정

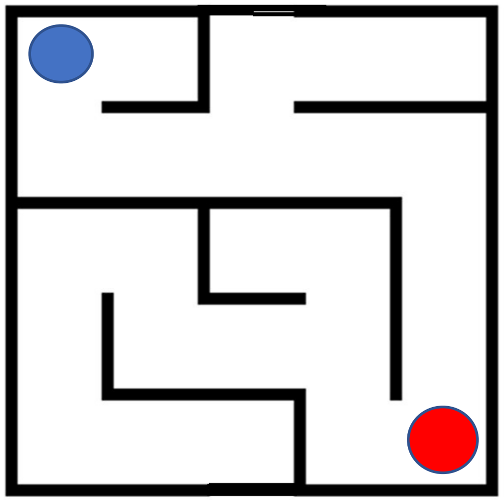
```
def update_theta(theta, pi, s_a_history):
    eta = 0.1 # 학습률
    T = len(s_a_history) - 1  # 목표 지점에 이르기까지 걸린 단계 수

    [m, n] = theta.shape  # theta의 행렬 크기를 구함
    delta_theta = theta.copy()  # Δtheta를 구할 준비, 포인터 참조이므로 delta_theta = theta로는 안됨

    # delta_theta를 요소 단위로 계산
    for i in range(0, m):
        for j in range(0, n):
            if not(np.isnan(theta[i, j])):  # theta가 nan이 아닌 경우

                SA_i = [SA for SA in s_a_history if SA[0] == i]
                # 히스토리에서 상태 i인 것만 모아오는 리스트 컴프리헨션

                SA_ij = [SA for SA in s_a_history if SA[0:2] == [i, j]]
                # 상태 i에서 행동 j를 취한 경우만 모음

                N_i = len(SA_i)  # 상태 i에서 모든 행동을 취한 횟수
                N_ij = len(SA_ij)  # 상태 i에서 행동 j를 취한 횟수

                delta_theta[i, j] = (N_ij - pi[i, j] * N_i) / T
                #delta_theta[i, j] = N_ij / T

    new_theta = theta + eta * delta_theta
```

경사법을 이용한 정책 수정

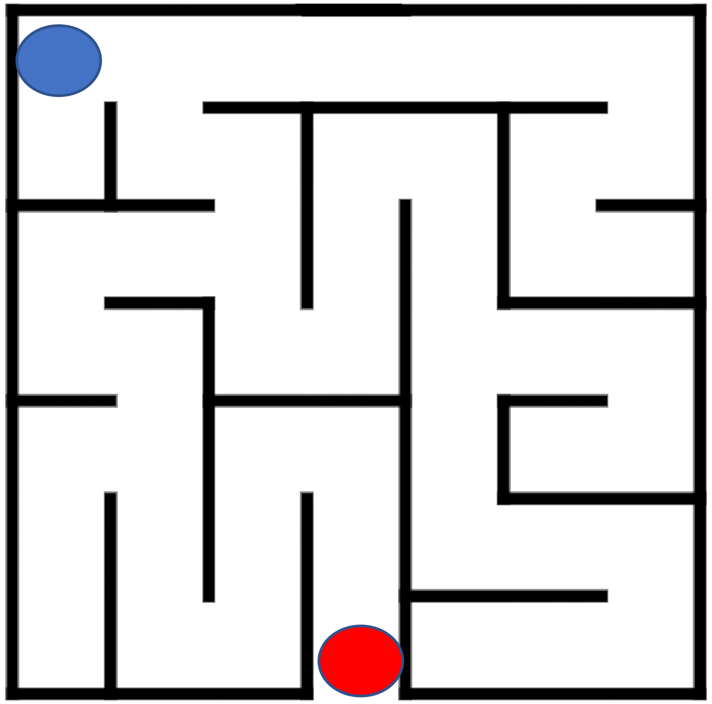
```
theta = theta_0.copy()
pi = softmax_convert_into_pi_from_theta(theta_0)
print(pi)

for i in range(4000) :
  s_a_history = goal_maze_ret_s_a(pi)
  theta = update_theta(theta, pi, s_a_history)
  pi = softmax_convert_into_pi_from_theta(theta)
print(pi)
```

```
[[0.    0.5   0.5   0.   ]
 [0.    0.5   0.    0.5  ]
 [0.    0.    0.    1.   ]
 [0.333 0.333 0.333 0.   ]
 [0.    0.5   0.    0.5  ]
 [0.    0.    0.5   0.5  ]
 [0.5   0.5   0.    0.   ]
 [0.    0.    0.    1.   ]]
[[0.    0.016 0.984 0.   ]
 [0.    0.273 0.    0.727]
 [0.    0.    0.    1.   ]
 [0.014 0.976 0.011 0.   ]
 [0.    0.984 0.    0.016]
 [0.    0.    0.987 0.013]
 [0.636 0.364 0.    0.   ]
 [0.    0.    0.    1.   ]]
```

다양한 크기의 미로생성



5×5



- 7×7 미로

Q-Learning

- 가치(미래에 받을 보상의 합)를 최대화
- 이전 값과 새로운 정보의 가중치를 더해서 가치를 반복적으로 수정
- ε-greedy 알고리즘 적용(탐색 및 활용)

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

가치 함수

- 즉각적인 보상 보다는 앞으로 받을 보상이 중요
- 미래에 대한 보상과 현재의 보상이 동일한 가중치를 가지고 있지 않음
- 감가율(discount factor)적용

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t} r_T$$

$$R_t = r_t + \gamma R_{t+1}$$

- 즉시 보상 보다는 누적 보상(Return)을 사용해 가치를 계산

- 가치 함수 : 지금 상태에서 미래에 받을 것으로 기대되는 보상의 합(누적 보상의 기대값)

최적 정책

- 상태 s일 때 보상의 합(가치함수)이 최대가 되도록 행동을 선택

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ \mathbb{E}[R_t \mid s_t = s, a_t = a]$$

- Q-Learning에서는 이러한 정책을 대신할 함수 Q를 학습

$$Q^*(s,\ \mathrm{a})$$

가치 계산

- 지금 s 상태에서 a 행동을 했을 때 미래에 받을 것으로 기대되는 보상의 합

$$Q^*(s,a) = \mathbb{E}\,[R_t \mid s_t = s,\ a_t = a]$$

- Q 함수를 찾는 벨만 방정식을 이용해 재귀적으로 Q 함수를 정의하고 순차적으로 업데이트함
- 현재 보상(s에서 행동 a)과 다음 상태에서의 Q함수의 최대값을 더함

$$Q^*(s,a) = r_t + \gamma \max_{a'} Q^*(s',a')$$

$$q(s,a) \leftarrow r + \gamma q_\pi(s',a')$$

가치 계산

- 실제가치는 현재가지 + 미래가치로 근사화

$$Q(s,a) \simeq R + Q(s',a')$$

- 가치에 감가율 적용

$$Q(s,a) \simeq R + \gamma Q(s',a')$$

- Q가 상태 테이블 이면 미래 가치는 s'의 Q최대값

$$Q(s,a) \simeq R + \gamma max Q(s',a')$$

Q-Learning 업데이트

$$Q^{\pi}(s,a) = R^{a}_{ss'} + \gamma \max_{a'} Q^{\pi}(s',a')$$

$$\begin{aligned} Q^{\pi}(s,a) &\leftarrow (1-\alpha)Q^{\pi}(s,a) + \alpha(R^{a}_{ss'} + \gamma \max_{a'} Q^{\pi}(s',a')) \\ &= Q^{\pi}(s,a) + \alpha\left(R^{a}_{ss'} + \gamma \max_{a'} Q^{\pi}(s',a') - Q^{\pi}(s,a)\right) \end{aligned}$$

$$q(s,a) = q(s,a) + \alpha(r + \gamma \max_{a'} q(s',a') - q(s,a))$$

가치 함수와 벨만 방정식

가치함수

$$
\begin{aligned}
v(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \dots\right) \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]
\end{aligned}
$$

벨만 방정식

$$q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\right]$$

벨만 최적 방정식

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a')$$

Q 함수

- Q함수는 랜덤으로 시작

```
[a, b] = theta_0.shape
Q = np.random.rand(a, b) * theta_0 * 0.1
print(Q)
```

```
[[       nan 0.08844375 0.05066354        nan]
 [       nan 0.07065955        nan 0.01384479]
 [       nan        nan        nan 0.09008373]
 [0.05485083 0.09563426 0.0212774         nan]
 [       nan 0.08661172        nan 0.07941565]
 [       nan        nan 0.03599532 0.01905227]
 [0.02906294 0.06551004        nan        nan]
 [       nan        nan        nan 0.00265867]
 [       nan        nan        nan        nan]]
```

ε-Greedy 정책

- 지금 상태에서 Q 함수의 최대가 되는 행동을 선택
- 탐험률(exploration : 현재의 정책을 무시하고 새로운 가능성을 추구
- 활용률(exploitation) : 현재의 정책을 그대로 따름
- ε-greedy 정책 : 일정 확률로 랜덤하게 선택

$$\begin{cases} argmax_a \, q_\pi(s,a) & \text{with probability 1-}\epsilon \\ \text{any action (a)} & \text{with probability } \epsilon \end{cases}$$

ε-Greedy 정책

```
def get_action(s, Q, epsilon, pi_0):
    direction = ["up", "right", "down", "left"]

    if np.random.rand() < epsilon:
        next_direction = np.random.choice(direction, p=pi_0[s, :])
    else:
        next_direction = direction[np.nanargmax(Q[s, :])]

    if next_direction == "up":
        action = 0
    elif next_direction == "right":
        action = 1
    elif next_direction == "down":
        action = 2
    elif next_direction == "left":
        action = 3

    return action
```

상태 이동

```python
def get_s_next(s, a):
    direction = ["up", "right", "down", "left"]
    next_direction = direction[a]

    if next_direction == "up":
        s_next = s - 3
    elif next_direction == "right":
        s_next = s + 1
    elif next_direction == "down":
        s_next = s + 3
    elif next_direction == "left":
        s_next = s - 1

    return s_next
```

Q 학습

```
def Q_learning(s, a, r, s_next, Q, eta, gamma):

    if s_next == 8:
        Q[s, a] = Q[s, a] + eta * (r - Q[s, a])

    else:
        Q[s, a] = Q[s, a] + eta * (r + gamma * np.nanmax(Q[s_next,: ]) - Q[s, a])

    return Q
```

미로 탈출

```python
def goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi):
    direction = ["up", "right", "down", "left"]
    s = 0
    a = a_next = get_action(s, Q, epsilon, pi)

    s_a_history = []

    while (1):
        a = a_next
        s_next = get_s_next(s, a)
        s_a_history.append([s, a,  direction[a], s_next])

        if s_next == 8:
            r = 1
            a_next = np.nan
        else:
            r = 0
            a_next = get_action(s_next, Q, epsilon, pi)

        Q = Q_learning(s, a, r, s_next, Q, eta, gamma)

        if s_next == 8:   break
        else:  s = s_next

    return [s_a_history, Q]
```

미로 탈출

```
pi_0 = simple_convert_into_pi_from_theta(theta_0)

eta = 0.1  # 학습률
gamma = 0.9  # 시간할인율
epsilon = 0.5  # ε-greedy 알고리즘 epsilon 초깃값

for episode in range(100) :
    print("에피소드: " + str(episode+1))
    epsilon = epsilon / 2

    [s_a_history, Q] = goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi_0)

    print("목표 지점에 이르기까지 걸린 단계 수는 " + str(len(s_a_history) - 1) + "단계입니다")

print(s_a_history)
```

미로 탈출

```
에피소드: 1
목표 지점에 이르기까지 걸린 단계 수는 241단계입니다
에피소드: 2
목표 지점에 이르기까지 걸린 단계 수는 17단계입니다
에피소드: 3
목표 지점에 이르기까지 걸린 단계 수는 271단계입니다
에피소드: 4
목표 지점에 이르기까지 걸린 단계 수는 9단계입니다
에피소드: 5
목표 지점에 이르기까지 걸린 단계 수는 7단계입니다
에피소드: 6
목표 지점에 이르기까지 걸린 단계 수는 3단계입니다
에피소드: 7
목표 지점에 이르기까지 걸린 단계 수는 3단계입니다
```

정책 경사법 vs. Q러닝

- 정책 경사법

```
while is_continue:
    s_a_history = goal_maze_ret_s_a(pi)
    new_theta = update_theta(theta, pi, s_a_history)  # 파라미터 Θ를 수정
```

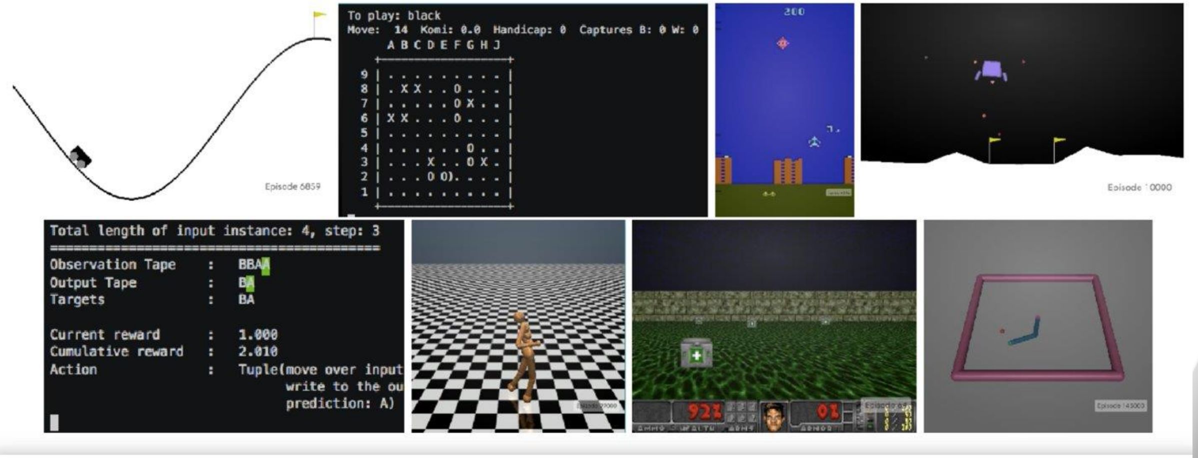
- Q 러닝

```
while is_continue:
    epsilon = epsilon / 2
    [s_a_history, Q] = goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi_0)
```

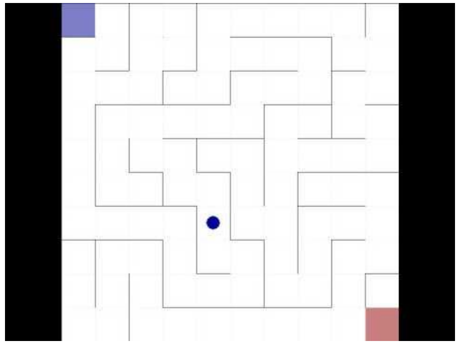
OpenAI Gym

- 인공지능을 개발함으로써 전적으로 인류에게 이익을 주는 것을 목표로 하는 인공지능 연구소(2015, 일론 머스크)
- 특허와 연구를 대중에 공개
- 간단한 게임들을 통해서 강화학습을 테스트 할 수 있는 환경

OpenAI Gym

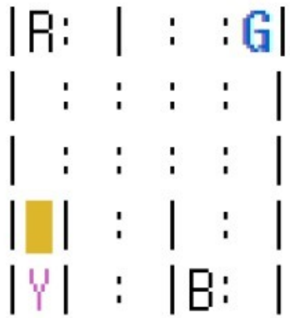


OpenAI Gym

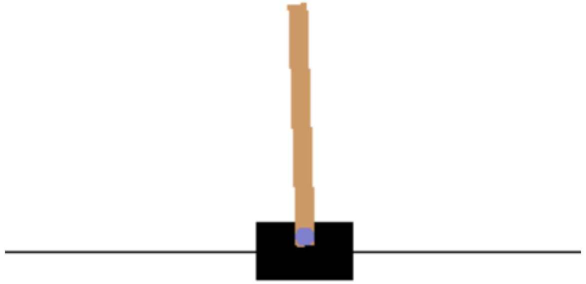


maze

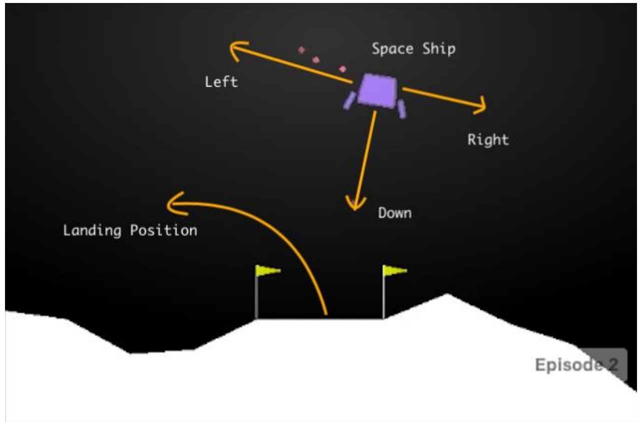
Frozen lake



Taxi



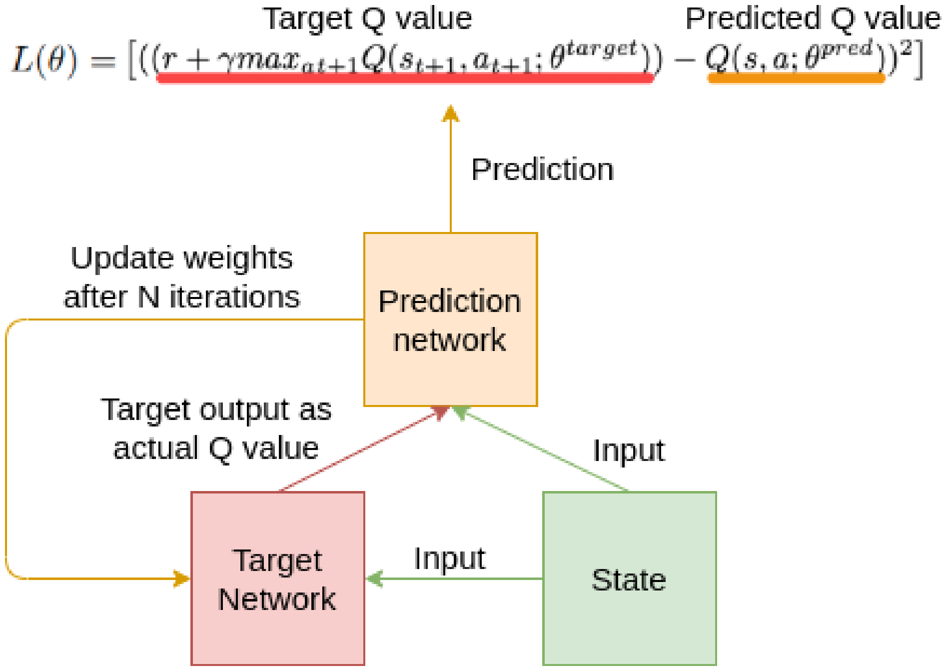
CartPole



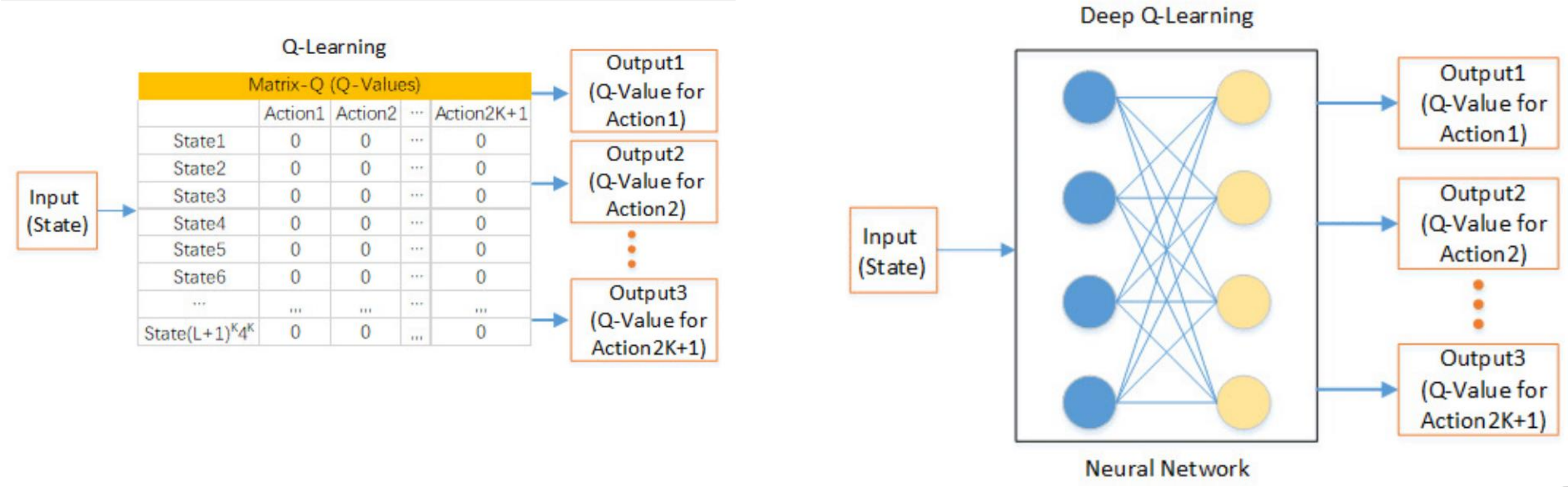
Lunar Lander

DQN(Deep Q-Network)

- Q 함수를 테이블이 아닌 신경망을 사용해 학습
- Replay memory
  - 과거 상태 및 행동을 기억
  - 랜덤 샘플링
- Target q-network
  - 별도의 target 신경망 구성
  - 주기적으로 현재 신경망으로 업데이트



Network 구조



https://www.researchgate.net/figure/Q-table-vs-deep-neural-network_fig4_348372871

DQN의 비용함수

$$Cost = \left[Q(s, a; \theta) - \left(r(s, a) + \gamma \max_{a} Q(s', a; \theta) \right) \right]^2$$

예측값(s,a) 현재보상 미래보상(s',a')