

## Q-learning의 핵심

Q에는 1) 내가 있는 상태 state, 2) 내가 하는 행동 action input과 출력 3)quality(reward)가 있다.

frozen lake게임에서 process

$Q(s1, \text{left})=0$ ,  $Q(s1, \text{RIGHT})=1.5$   $Q(s1, \text{UP})=0$ ,  $Q(s1, \text{Down})=0.3$  이 제공된다면 max value를 선택한다. max의 argument RIGHT를 선택한다

$$\text{Max } Q = \max_{a'} Q(s, a') \quad Q\text{가 가질 수 있는 최대값 } a'$$

$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$  Q가 최대값이 되게끔 하는 argument a -> optimal  $\Pi$

## Q러닝의 가정

S'에서 Q가 있어야 된다.

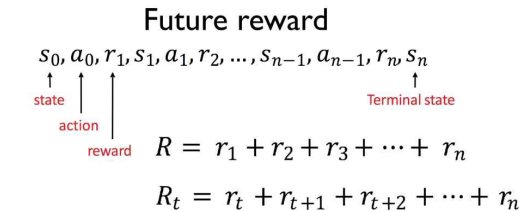
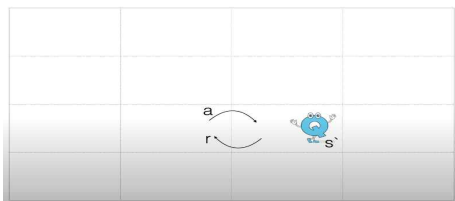
s에서 a를 하고 s'으로 갈거다. a를 하면 r을 받는다

$Q(s,a)$ 이라면  $Q(s',a')$ 이 존재할거다

$Q = r + \max Q(s', a')$  이게 되고 이 Q를 학습하는 거다.

현재  $Q(s,a)=?$

여기서  $Q = r + \max_{a'} Q(s', a')$



Reward의 총합은  $R = r_1 + r_2 + r_3 + \dots + r_n$

$$R_{t+1} = r_{(t+1)} + r_{(t+2)} + r_{(t+3)} + \dots r_n$$
$$R_{-}(t) = r_{-t} + R_{-t+1} \text{ 가 된다.}$$

$R^*_t = r_t + \max R(t+1)$  가 되고, 결국

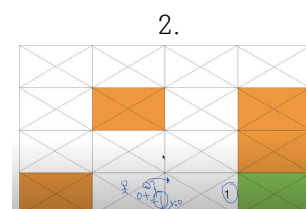
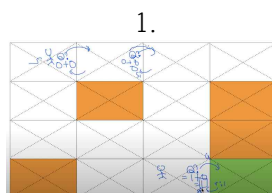
$$Q(s,a) = r + \max_{a'} Q(s',a') \text{ 가 된다.}$$

## frozen lake 문제에서 순서

[illegible]

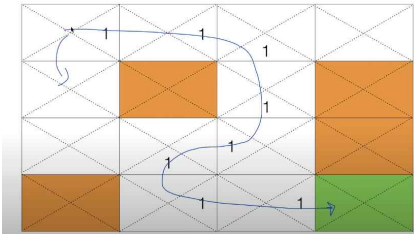
initial Q-table

0으로 초기화



1 -> 2 -> ... n번 진행하면서 각 상태의 Q를 update함

## Exploit VS Exploration



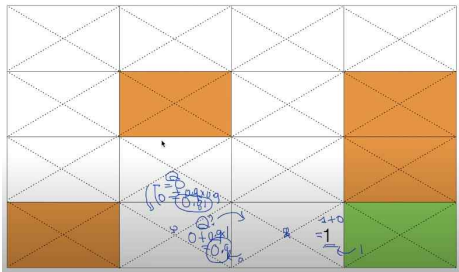
exploration을 통해 가본 길 말고 새로운 길을 탐색한다

"Select an action a and execute it"

a를 선택할 때 exploit이나 exploration을 선택한다!

Discounted future reward

$$R_t = r_t + \Gamma r_{t+1} + \Gamma^2 r_{t+2} + \Gamma^3 r_{t+3} + \Gamma^4 r_{t+4} + \dots$$



discount factor는 간단하게 미래의 보상을 현재의 값으로 받아들이기 위한 장치이다.

frozen lake 문제에서 e-greedy를 사용하여 action 선택하는 Q-learning

[코드 실행 결과]

```

done = False
# decaying E-greedy
e = 1. / ((i // 100) + 1)
# Q-table learning algorithm
while not done:
    # e-greedy 선택
    action = np.argmax(Q[state])
    # S'의 상태와 보상
    new_state, reward, done, _ = env.step(action)
    for i in range(num_episodes):
        while not done:
            # Q-table update
            Q[state] = (1 - e) * Q[state] + e * (reward + gamma * Q[new_state])
            state = new_state
            done = True
    done = False
    e = 1. / ((i // 100) + 1)
    # Q-table learning algorithm
    while not done:
        # e-greedy 선택
        action = np.argmax(Q[state])
        # S'의 상태와 보상
        new_state, reward, done, _ = env.step(action)
        # Q-table update
        Q[state] = (1 - e) * Q[state] + e * (reward + gamma * Q[new_state])
        state = new_state
        done = True
    done = False
    e = 1. / ((i // 100) + 1)

```

Final Q-Table Values

```

[[8.52311336e-03 1.66749997e-02 2.70218149e-01 2.80210662e-02]
 [2.47045239e-03 2.31112330e-03 5.57238716e-04 2.49400807e-01]
 [0.00000000e+00 9.28030602e-03 0.00000000e+00 1.77576647e-01]
 [0.00000000e+00 4.20792160e-03 4.05497794e-03 1.77307325e-01]
 [5.00140004e-01 2.56719039e-04 3.66819825e-04 1.77198216e-05]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.10559926e-04 5.23709623e-05 2.00988800e-01 1.46048185e-04]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.46493817e-02 2.08730809e-03 2.27937466e-03 7.96254474e-01]
 [2.14727180e-03 3.89153911e-01 4.46251794e-04 3.20943469e-04]
 [8.69373763e-01 4.18870265e-04 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.69136871e-03 0.00000000e+00 8.56428331e-01 0.00000000e+00]
 [0.00000000e+00 9.92171820e-01 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]

```

Process finished with exit code 0

Success rate : 0.553

deterministic일땐  $r + \Gamma \max_{a'} Q(s', a')$ 이  $Q$ 이고

확률적(미끄러운 lake) 일땐  $(1 - \Gamma)Q(s, a) + \Gamma(r + \Gamma \max_{a'} Q(s', a'))$

100\*100에서 Q-table을 사용할 수 있을까?

$100 * 100 * 4(\text{action}) = 40000$ 개

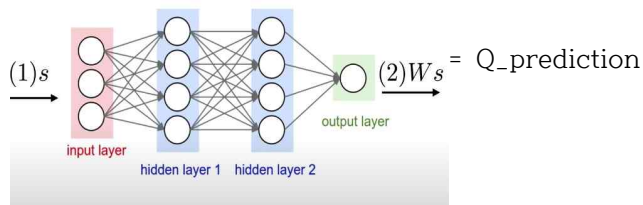
실제 80\*80pixel + 2color에서는?

$2^{(80*80)}$ 이다

Q-table로는 현실 세계의 문제를 못 푼다

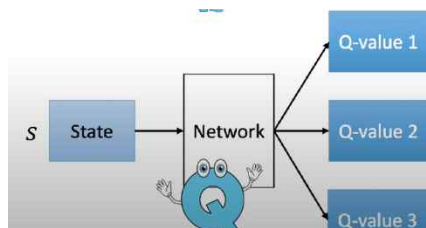
Q-network

상태 하나에 모든 가능한 액션값의 Q-value을 찾는다



$$\text{cost}(W) = (Ws - y)^2$$

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



$$\text{cost}(W) = (Q_{\text{pred}} - Q^*)^2$$

cost를 minimize하도록 학습하는 것이 목표이다

$\hat{Q} = Q \text{ hat} = \text{예측값}$

$\hat{Q}(s, a | \theta)$   $\theta = \text{weight}$  즉 weight에 따라서 s, a가 달라진다.

$$\rightarrow W_s = Q_{\text{prediction}} = \hat{Q}(s, a | \theta) \sim Q^*(s, a)$$

수학적 기호 표기

$$\min_{\theta} \sum_{t=0}^T [\underbrace{\hat{Q}(s_t, a_t | \theta)}_{Q_{\text{pred}}} - \underbrace{(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a' | \theta))}_{\text{target}}]^2$$

$\text{cost}(W) = (Q_{\text{pred}} - Q^*)^2$  이다.

이제 cost를 최소화 하는  $\theta$ 를 구하는 것이 문제이다.

deep Q learning Algorithm

Q 네트워크의  $w$ 를 초기화 한다

1번째 상태를 가져온다. 그리고 전처리 한다 ( $\phi$ ) [단순히 S라고 생각하면 된다]

for

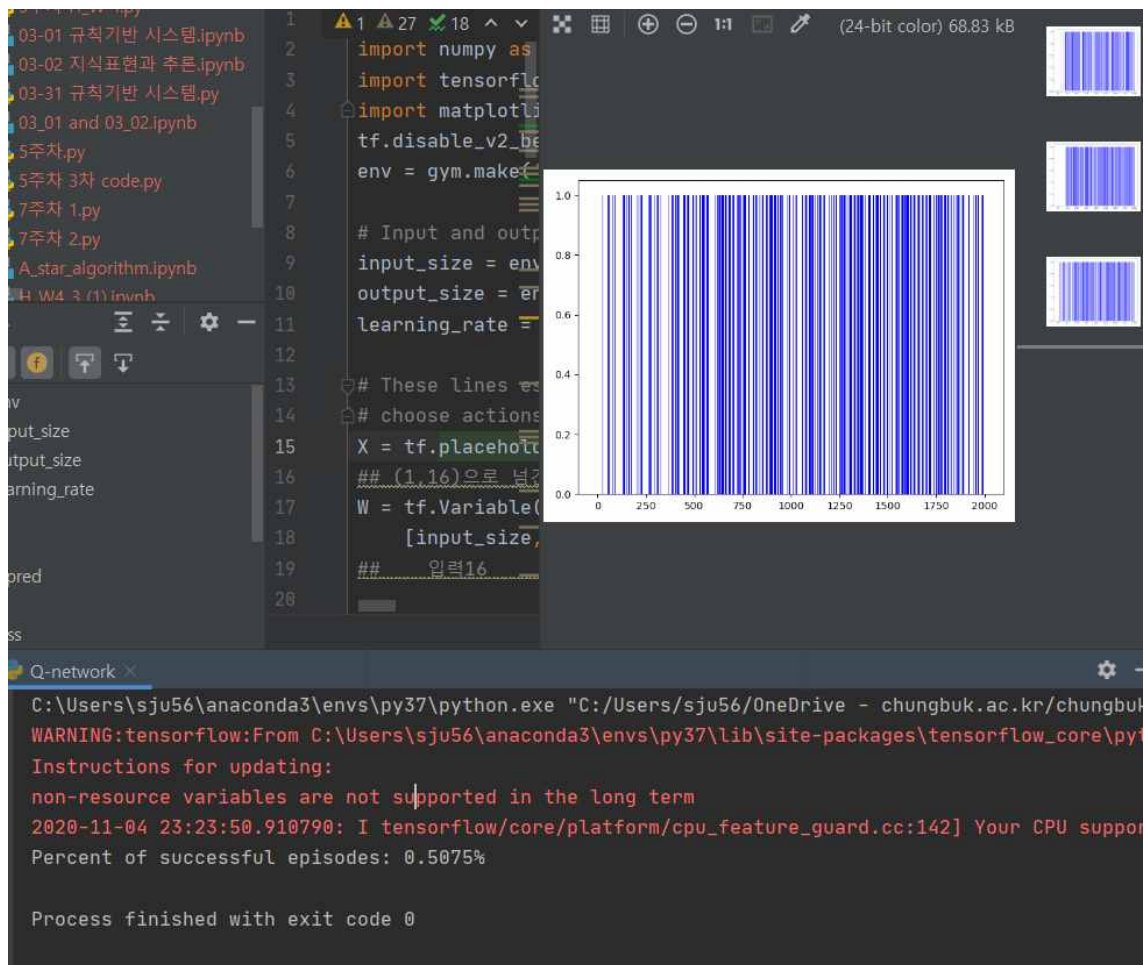
e그리디로 랜덤하게 action을 취하거나 현재 Q에서 max값을 고른다

그리고 a을 한다. 그리고  $S_{t+1}$  와  $r_t$ 를 받는다

$$y_j = \begin{cases} r_j & \text{마지막 일 때} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{마지막이 아닐 때} \end{cases}$$

그리고  $y_i$ 를 구하고 cost값을 최소화 하는  $\theta$ 를 찾으려 한다  $(y_j - Q(\phi_j, a_j; \theta))^2$

11월 2주차 코드 완성 후 결과



Q-table보다 성능이 안 좋다. 0.5%정도 나온다.

이유 : Q-network에서는 층이 하나였다. ->deep learning은 층이 여러 개로 해야 잘 나온

다. 신호등 인식과 차선 인식 yolo에서는 x, y, 층이 1000개 었다.

2.sample들의 correlations 연관성이 많다. ->minibatch와 같은 상태들을 랜덤하게 선택해서 최적해를 찾아야 한다???

3. Q\_pred와 Q\_target의 parameter( $\theta$ )값을 다르게 하고 1000번, 2000번에 한번씩 target의 parameter를 prediction의 parameter로 update시킨다

->1,2,3, 해결법으로 Q-network를 upgrade시킨 것이 deep mind의 DQN이다.