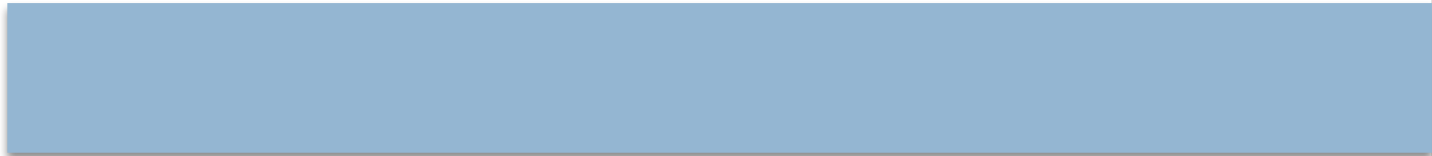


# 13장 강화 학습



# 학습 목표

- 전통 Q-학습을 이해한다.
- 심층 Q-학습 신경망을 이해한다.
- 심층 Q-학습 신경망의 문제점, 타겟 신경망을 알아본다.
- 심층 Q-학습 신경망의 구현한다.



# 강화학습이란?

- 지금까지 우리가 살펴본 딥러닝에는 항상 훈련 데이터와 정답 레이블이 있었다.
- 만약 딥러닝을 탑재한 에이전트가, 환경에서 스스로 행동하여서 학습할 수 있다면 어떨까?



그림 13-1 일반적인 딥러닝과 강화 학습의 차이점



# 알파스타

- 스타크래프트는 불완전한 정보를 가지고 있고, 실시간으로 경기가 진행되며, 장기 계획이 필요한 어려운 게임이다.
- 하지만 스타크래프트에서도 강화 학습 인공지능이 인간을 5-0으로 물리친 바 있다.

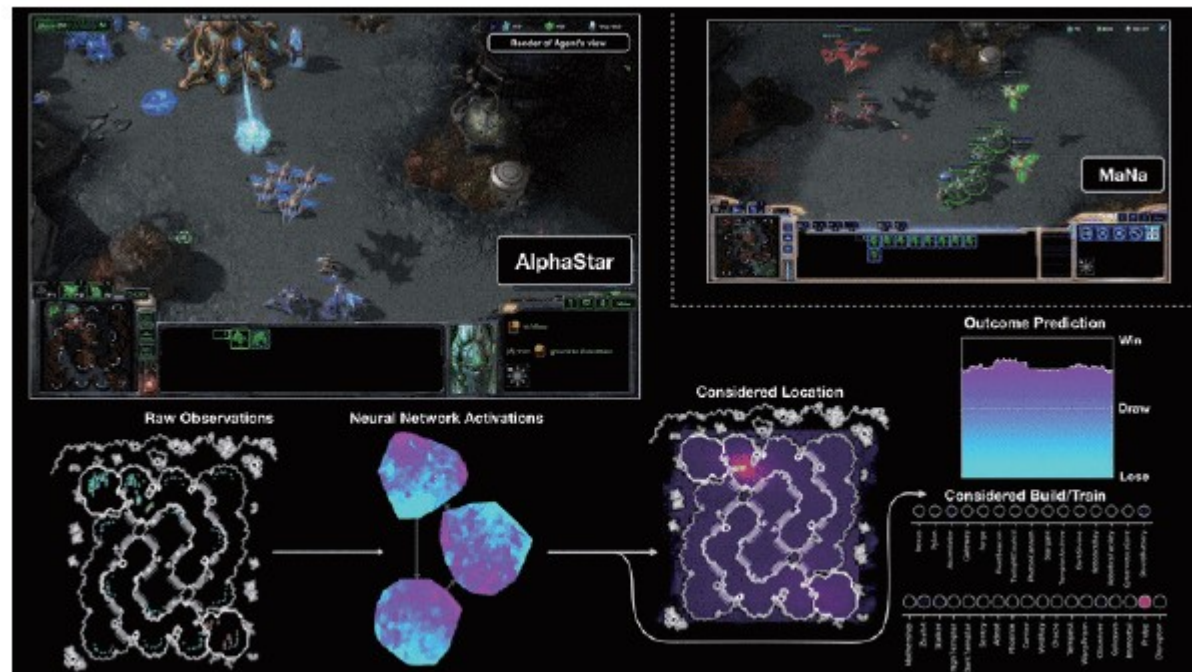
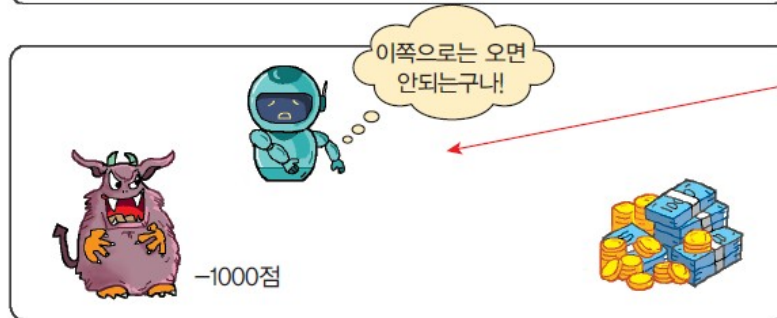
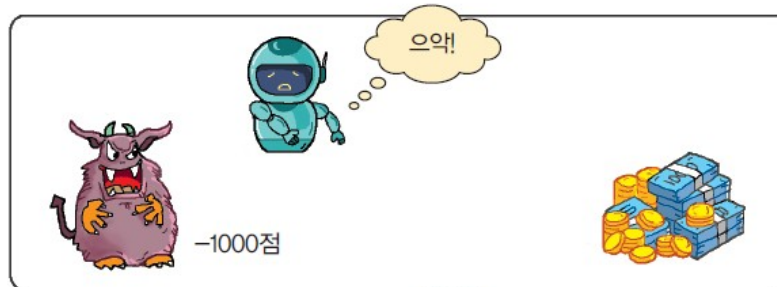
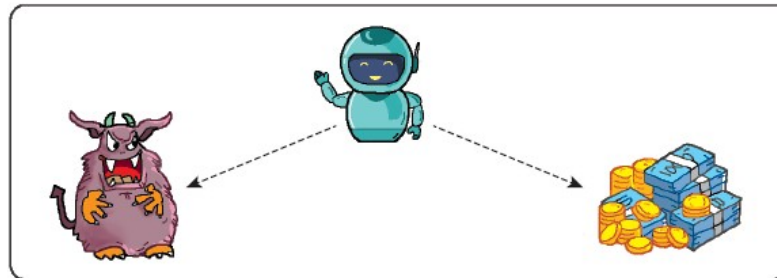


그림 13-2 스타크래프트 게임을 수행하는 알파스타



# 강화학습의 기본 원리



이것이 바로 강화 학습이다.

# 강화 학습과 다른 학습 방법의 비교

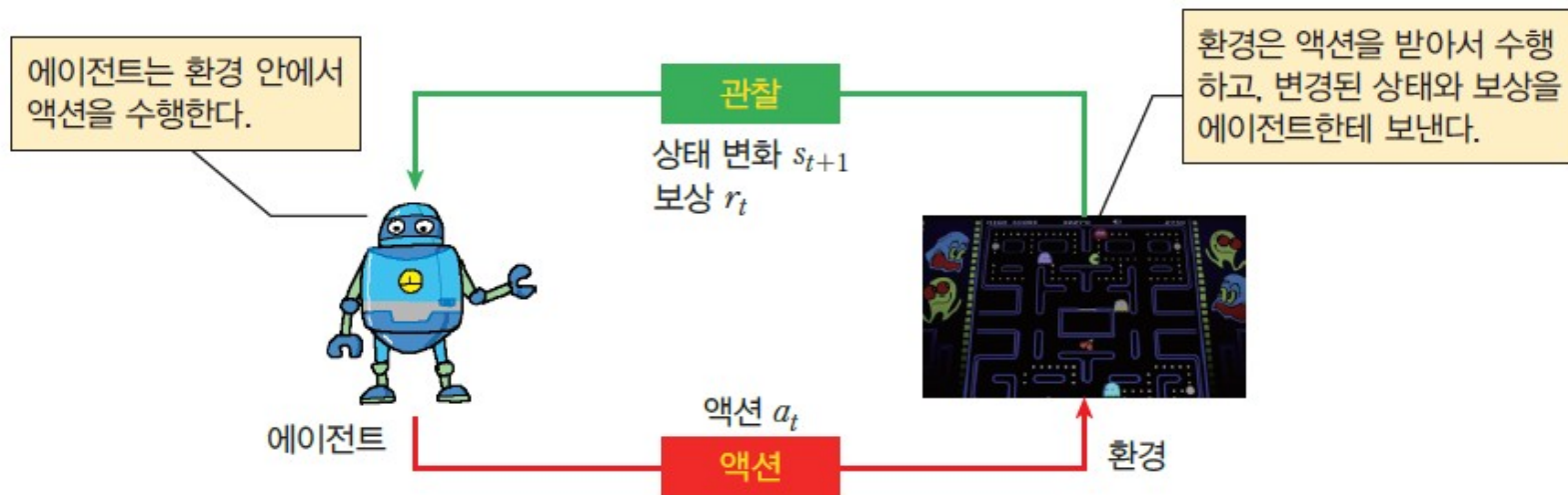


	지도 학습	비지도 학습	강화 학습
데이터	$(x, y)$ $x$ 는 데이터이고 $y$ 는 레이블이다.	$(x)$ $x$ 는 데이터이고 레이블은 없다.	(상태, 액션)의 짝
목적	$x \rightarrow y$ 로 매핑하는 함수를 학습하는 것이다.	데이터 안에 내재한 구조를 학습한다.	많은 시간 단계에서 미래 보상을 최대화한다.
예	이미지에서 과일과 강아지를 인식한다. 	같은 과일끼리 구분한다. 	과일을 먹으면 장기적으로 건강에 좋다는 것을 깨우친다. 



# 강화 학습 프레임워크

- 에이전트(agent): 강화 학습의 중심이 되는 객체
- 환경(Environment): 에이전트가 작동하는 물리적 세계
- 상태(state): 에이전트의 현재 상황, 미로에서의 에이전트의 위치가 상태일 수 있다.
- 보상(reward) : 환경으로부터의 피드백,
- 액션(action) : 에이전트의 행동

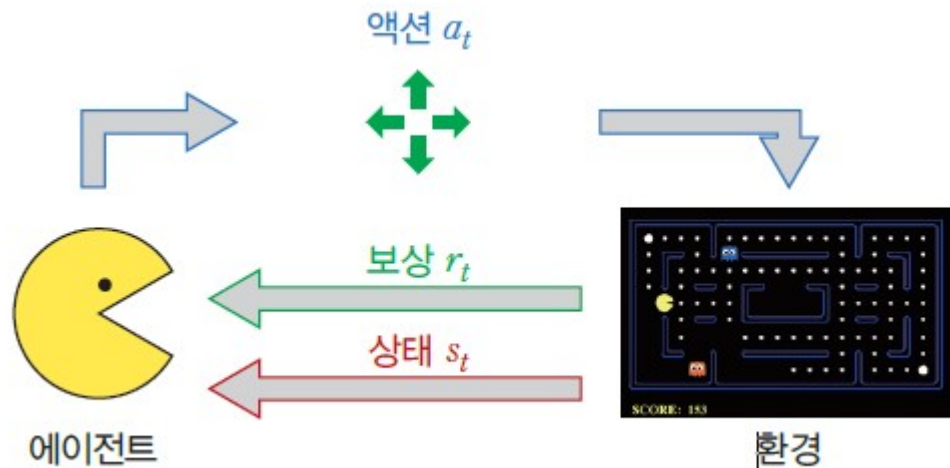






# 게임에서의 강화학습

- 강화 학습에서 에이전트는 환경 안에서 자신의 보상을 극대화하려고 한다. 보상은 성공 또는 실패에 대한 피드백이다. 에이전트가 행동할 때마다 보상을 받을 필요는 없지만, 보상이 지연될 수 있다. 즉 마지막에 하나의 보상만을 받는 경우도 많다.

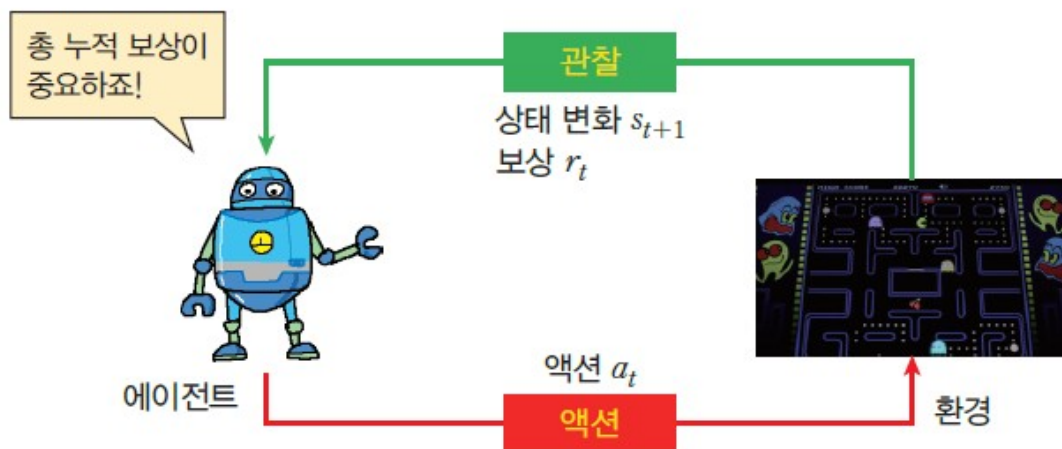




# 보상

- 보상은 에이전트 액션이 성공했는지 실패했는지를 알려주는 중요한 피드백이다.
- 보상  $r_t$  는 시간  $t$ 에서 에이전트가 받는 보상이다.
- 에이전트가 받는 전체 보상을  $R_t$  라고 하면  $R_t$  는 다음과 같은 수식으로 나타낼 수 있다.

$$R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} + r_{t+2} + \dots$$



# 할인된 보상

- 에이전트가 미래에 받을 보상은 약간 할인해서 계산해야 한다.





# 총 보상

- 강화 학습에서도 “할인된 보상”이라는 개념을 사용한다. 미래의 보상에는 할인 계수 람다를 곱하여 총 보상을 계산한다. 할인 계수  $\lambda$ 는 0에서 1 사이의 값이다.

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$



# Q 함수

- Q 함수는 상태  $s$ 에 있는 에이전트가 어떤 액션  $a$ 를 실행하여서 얻을 수 있는 미래 총보상값의 기대값(확률적인 환경을 가정했을 경우)이다.

$$Q(s_t, a_t) = E[R_t | s_t, a_t]$$

상태	액션
----	----

- 확률적인 환경이 아니다면 Q 함수는 상태  $s$ 에 있는 에이전트가 어떤 액션  $a$ 를 실행하여서 얻을 수 있는 총 보상값이다.

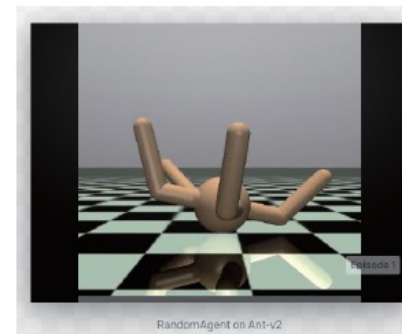


- 현재 상태  $s$ 에서 가장 좋은 액션을 추론하기 위해서는 어떤 정책  $r(s)$ 을 필요로 한다.
- 가장 상식적인 정책은 미래 보상을 최대화할 수 있는 액션을 선택하는 것이다.

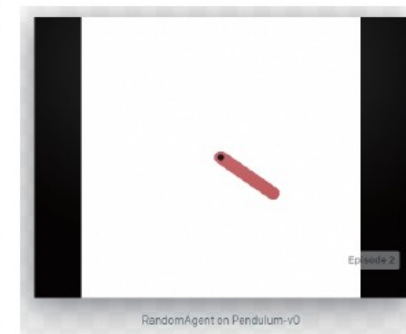
$$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

- 현재 상태에서 가능한 모든 액션 중에서 가장  $Q$  값이 높은 액션을 선택하면 된다.

- OpenAI 재단은 인공지능을 위한 여러 가지 프로젝트를 진행하는 비영리 재단이다.
- 특히 강화 학습을 위한 Gym 라이브러리(<https://gymnasium.farama.org/#>)가 유명하다



(a) 랜덤 보행 게임



(b) CartPole 게임



(c) 스페이스 인베이더 게임



(d) 루나 랜더 게임



# CartPole 게임

```
import gym

env = gym.make("CartPole-v1", render_mode="human")          # (1)
observation = env.reset()                                    # (2)

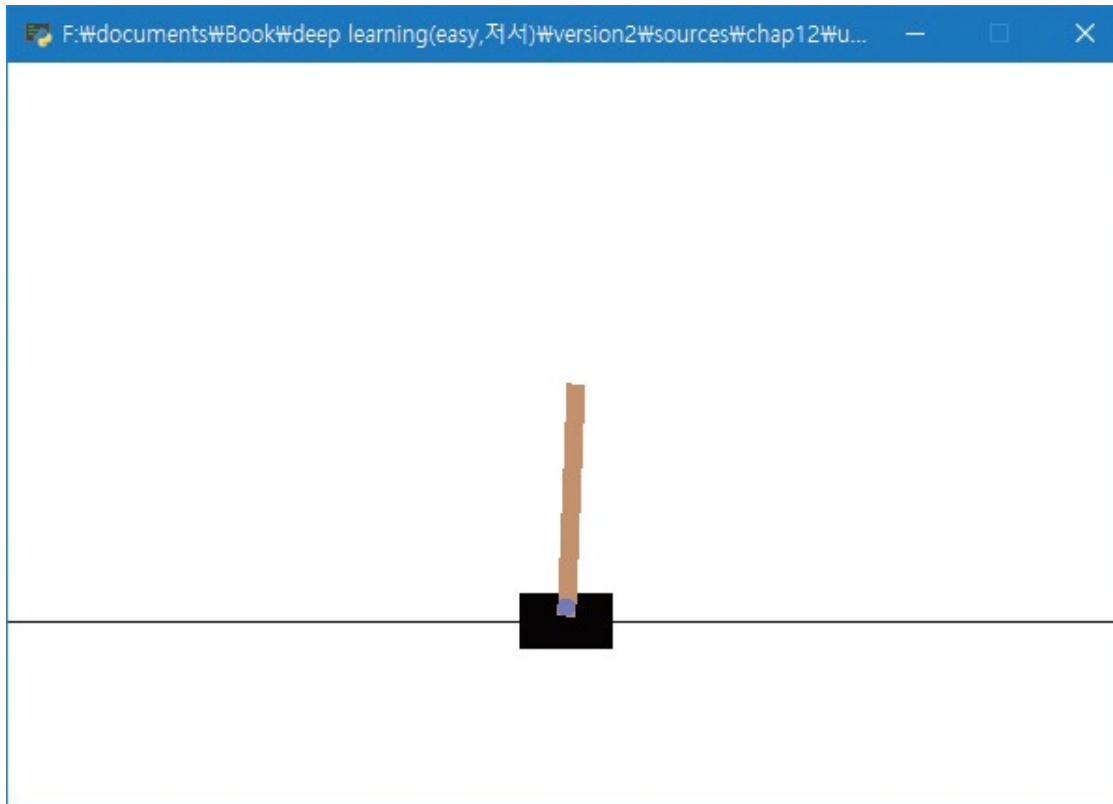
for _ in range(1000):                                       # (3)
    env.render()                                             # (4)
    action = env.action_space.sample()                      # (5)
    observation, reward, done, _, _ = env.step(action)      # (6)

    if done:
        observation = env.reset()                            # (7)
env.close()
```



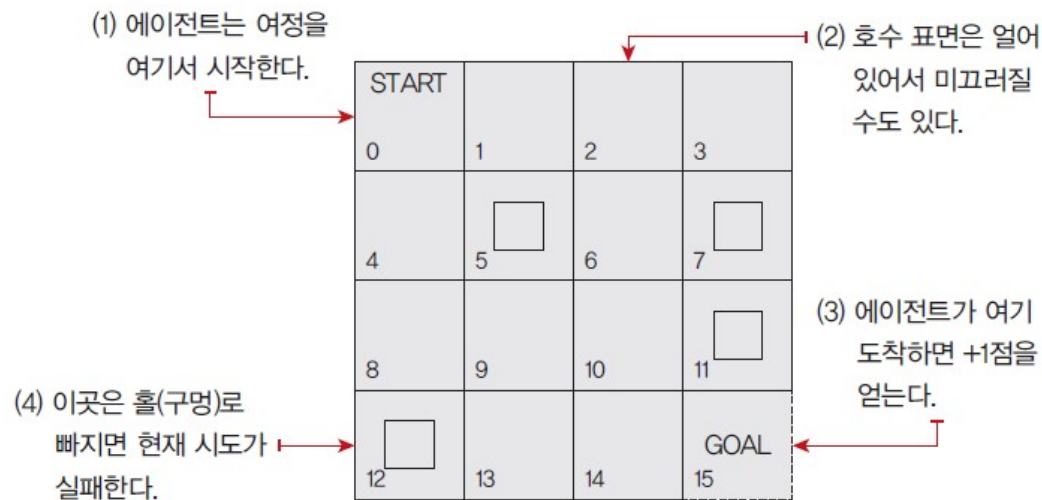
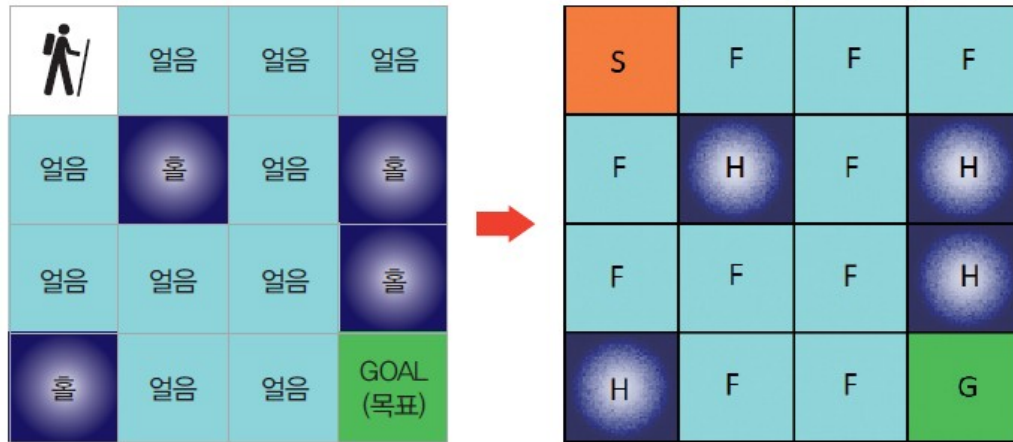


# CartPole 게임





# FrozenLake 게임





# FrozenLake 게임

- 얼음 호수 위를 에이전트가 걸어간다고 생각하자. 얼음 호수에는 홀도 있고 목표도 있다.
- 홀에 빠지면 게임은 종료된다.
- 홀에 빠지지 않고 목표에 도착하면 게임에서 1점을 얻는다.
- 얼음 위에서 미끄러져서 의도하지 않은 위치로 갈 수도 있지만 일단 이 가정은 제외하자.

	얼음	얼음	얼음
얼음	홀	얼음	홀
얼음	얼음	얼음	홀
홀	얼음	얼음	GOAL (목표)



# FrozenLake 게임

```
import gym

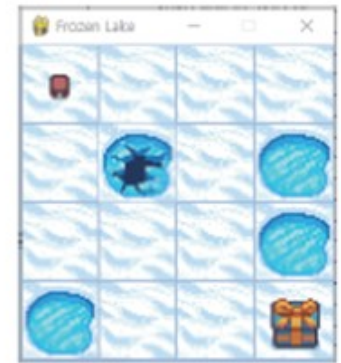
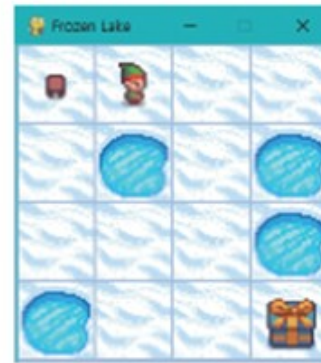
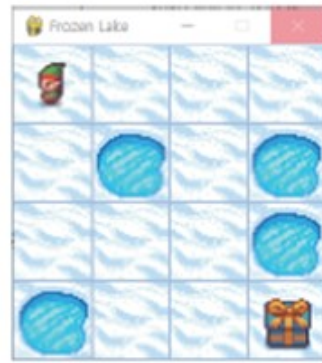
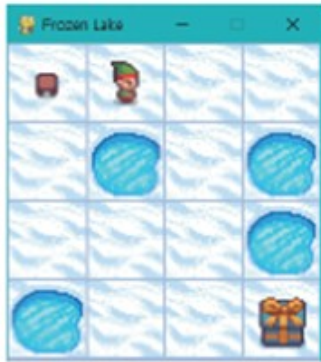
env = gym.make('FrozenLake-v1', render_mode='human', is_slippery=False)
observation = env.reset()

for _ in range(100):
    env.render()
    action = env.action_space.sample()    # (1)
    observation, reward, done, _, _ = env.step(action) # (2)

    if done:
        observation = env.reset()
env.close()
```



# FrozenLake 게임





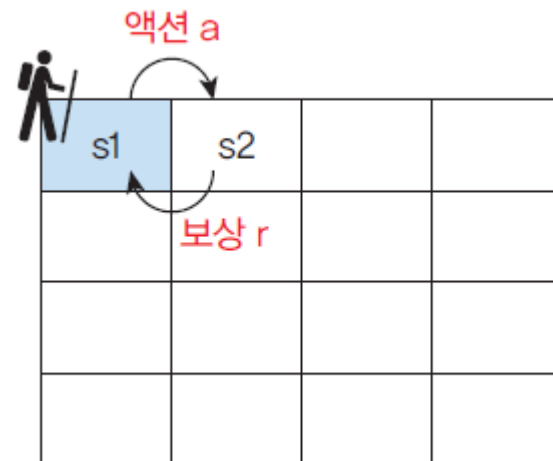
# 전통 Q-학습

- 전통적인 강화 학습 알고리즘 중의 하나인 Q-학습(Q-learning)을 먼저 살펴보자.
- 앞 절에서 설명한 얼음 호수(frozen lake) 문제를 가지고 Q-학습을 설명한다.
- 얼음 호수 위를 에이전트가 걸어간다고 생각하자. 얼음 호수에는 홀도 있고 목표도 있다.
- 홀에 빠지면 게임은 종료된다.
- 홀에 빠지지 않고 목표에 도착하면 게임에서 1점을 얻는다.



# FrozenLake 게임

- 이 게임은 아주 간단해 보이지만, 아무것도 모르는 에이전트 입장에서는 결코 만만한 문제가 아니다. 우리는 전체 게임 보드를 볼 수 있지만, 에이전트는 현재 있는 장소밖에는 알지 못한다.
- 에이전트가 상태  $s1$ 에서 오른쪽으로 이동하여서(이것이 액션이다) 상태  $s2$ 로 갔다면 어떤 보상  $r$ 을 받게 된다. 보상은 대부분 0이고 에이전트가 목표 상태로 갔을 때만 1이 된다.
- 처음에는 보상이 거의 0이기 때문에 에이전트는 처음에는 판단하기가 어렵다. 에이전트가 목표에 도달한 경우에만 보상으로 1을 받는다.







# FrozenLake 게임

- 전통적인 방법은 “동적 프로그래밍(dynamic programming)”이라고 불리는 방법으로, 기본적으로 복잡한 문제를 “약간씩 겹치는 서브 문제”들로 분해하고 이들 서브 문제들의 결과를 테이블에 저장하는 방법이다.
- 에이전트가 어떤 상태에서 특정한 액션을 하고 보상을 받을 때마다 테이블에 기록한다. 에이전트가 시행착오를 거듭할수록 테이블은 점점 정확해진다.



# 동적 프로그래밍의 예

- 피보나치 수열 계산

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

순환호출 방법

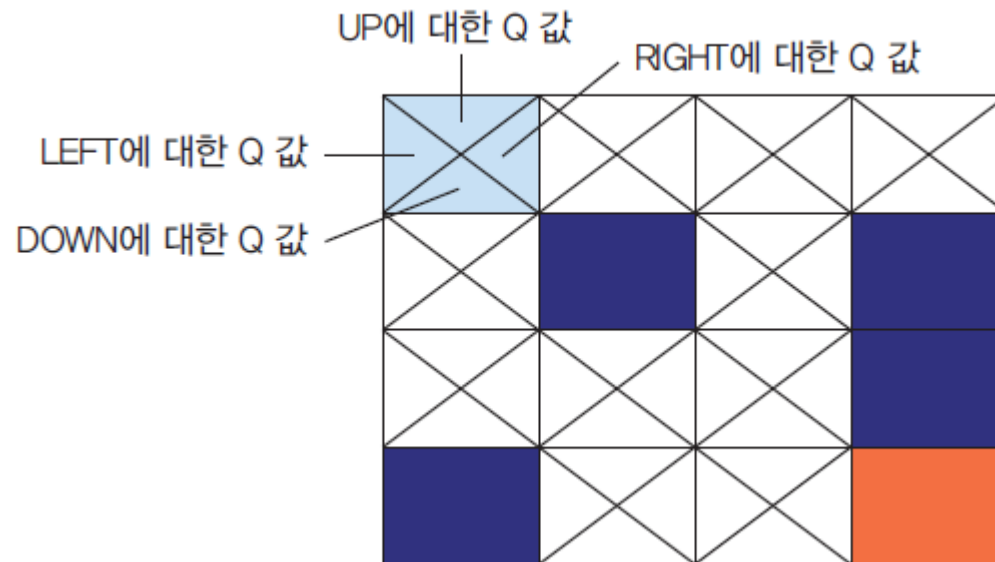
```
int fib(int n)
{
    int f[n+2]; int i;

    f[0] = 0; f[1] = 1;
    for (i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

동적 프로그래밍: 테이블을  
이용하고 약간 겹치는 문제

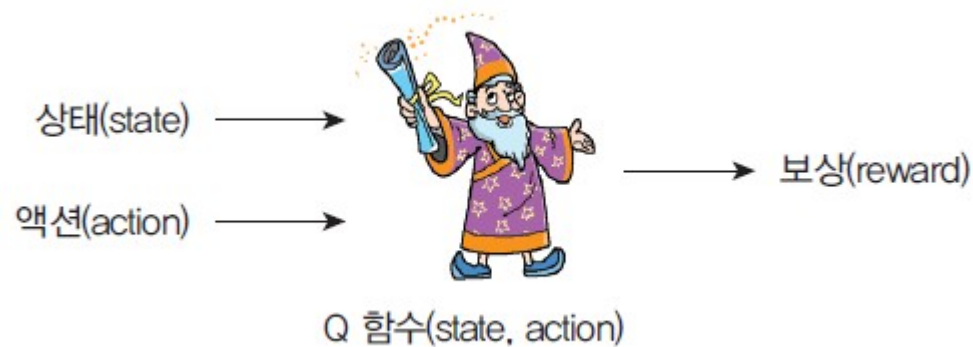


# Q값을 저장하는 배열을 생성한다.

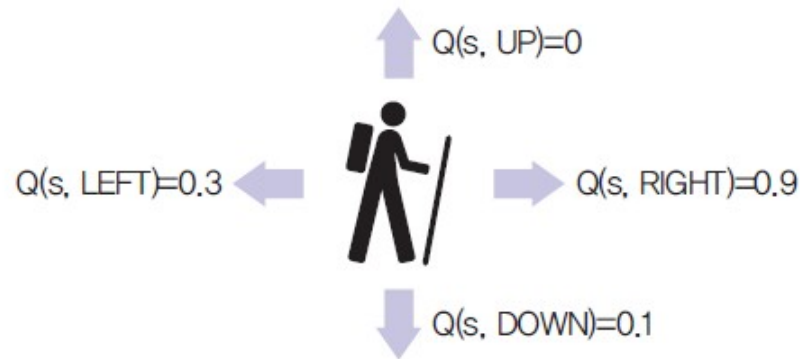


# Q 함수

- 어떤 상태에서 특정한 행동을 하여서 받은 총 보상값을 Q 함수라고 한다.
- Q 함수는 에이전트의 현재 상태와 에이전트가 실행하는 액션을 받아서 총 보상값을 반환하는 함수이다.



- 예를 들어서 특정한 상태  $s$ 에서 다음과 같이  $Q$  값이 계산되었다고 하자.



- 가장 상식적인 정책은  $Q$  값 중에서 최대값을 찾고 최대값과 관련된 액션을 실행하는 것이다.

$$\pi(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$$



# Q 값 순환 관계식

- 총 보상은 다음과 같이 순환적으로 계산할 수 있다.

$$R_t = r_t + r_{t+1} + \dots + r_n$$



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

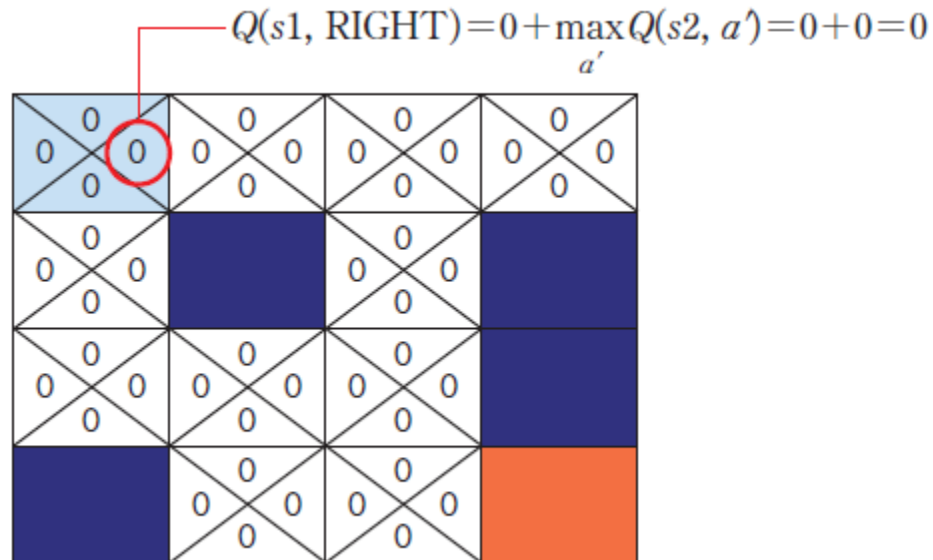
- 이것과 유사하게 상태  $s$ 에서의 Q 값은 다음과 같이 순환적으로 계산할 수 있다. 즉 상태  $s$ 에서 액션  $a$ 를 실행하였을 때 받는 보상  $r$ 에, 다음 상태에서의 Q 값 중에서 최대값을 더하게 된다.

$$Q(s, a) = r + \max_{a'} Q(s', a')$$

전통적인 Q-학습에서는  
결국 이 순환 관계식을  
사용하여  
테이블 내의 Q 값들이 계속

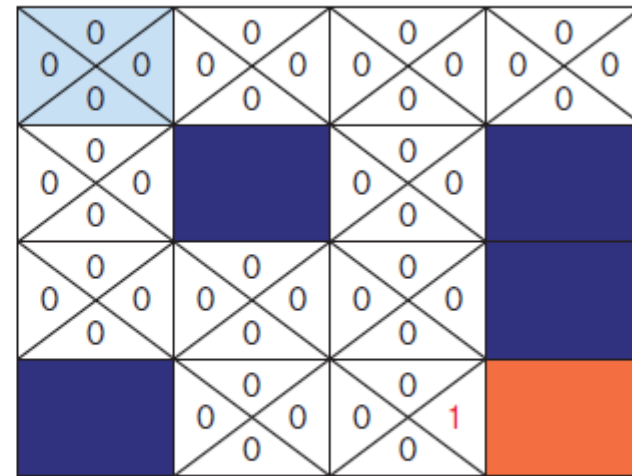
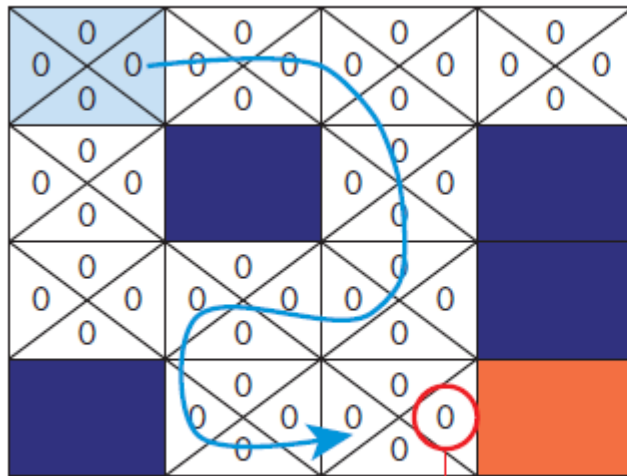
# 얼음 호수 문제에서 실제로 Q 값을 계산해보자.

- 시작할 때는 모든 Q 값이 전부 0이다. 에이전트가 시작 상태 s1에서 오른쪽에 있는 상태 s2로 갔을 때의 Q 값을 계산해보자.





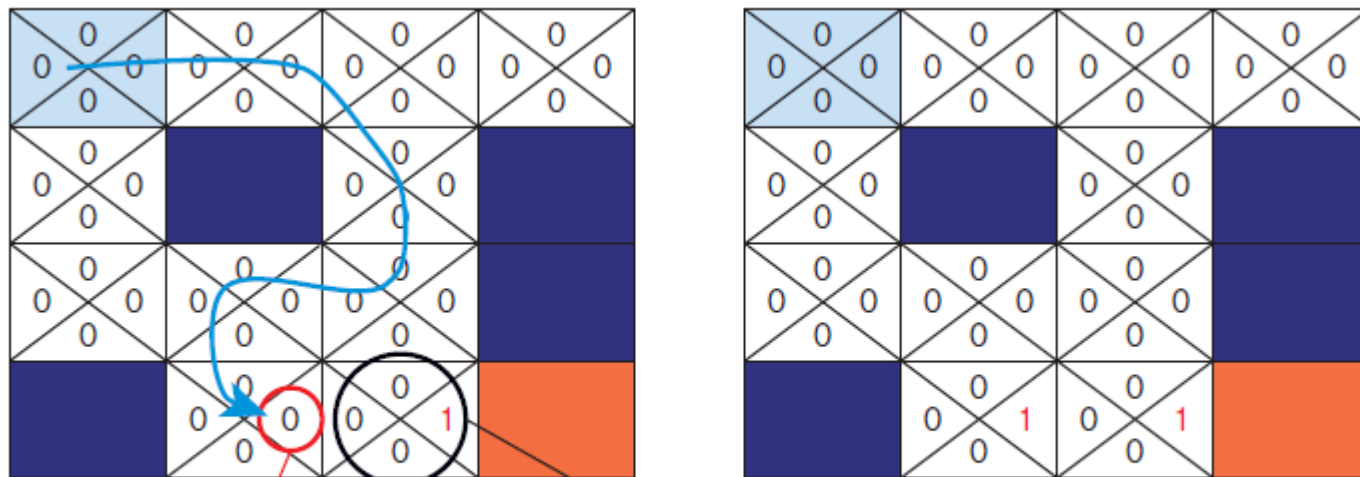
 계속 Q 값은 0이 되지만 반전이  
있다.



$$Q(s_{15}, \text{RIGHT}) = 1 + \max_{a'} Q(s_{16}, a') = 1 + 0 = 1$$



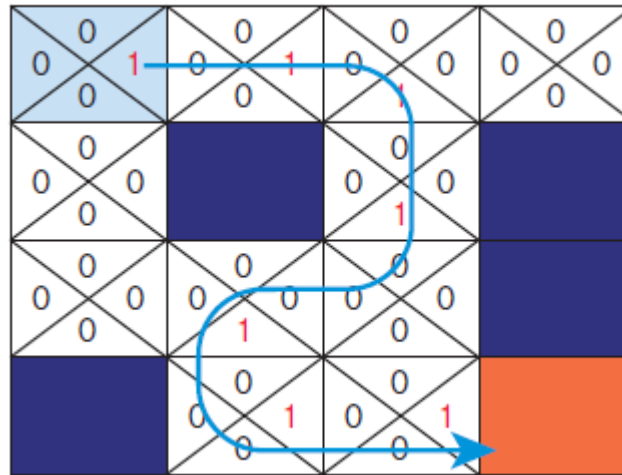
# 상태 s14에서의 Q 값 계산



$$Q(s_{14}, \text{RIGHT}) = 0 + \max_{a'} Q(s_{15}, a') = 0 + 1 = 1$$

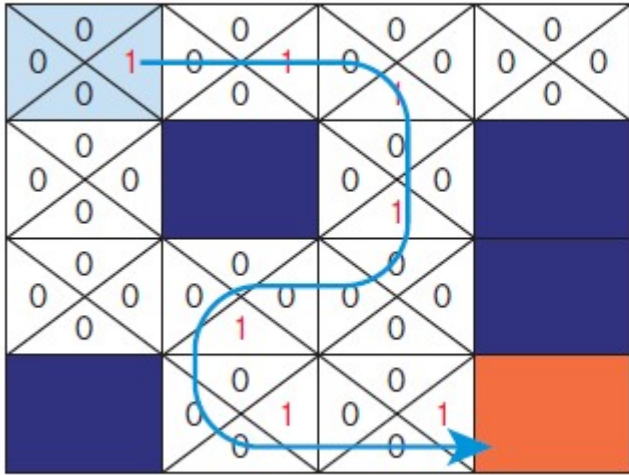


- 이런 식으로 계속 Q 값이 업데이트된다. 따라서 에피소드를 많이 진행하면 다음과 같이 Q 값이 설정될 수 있다.





- 지금까지 우리가 살펴본 Q-학습은 에이전트가 항상 동일한 경로만을 탐색하는 문제가 있다.
- 이 경로는 물론 최적 경로는 아니다. 하지만 우리의 정책대로 한다면 이렇게 움직일 수밖에 없다.



# 어떻게 하면 새로운 경로도 찾을 수 있을까?

- 처음에는 Q 값이 작은 액션이라고 하더라도 시도해볼 필요가 있다. 이것을 탐사라고 한다.
- 강화 학습에서도 처음에는 모험을 할 필요가 있다. 이것은  $\epsilon$ -greedy 알고리즘으로 가능하다.



처음에는 여기 저기  
모험을 해볼 필요가  
있다.



# $\epsilon$ -greedy 알고리즘

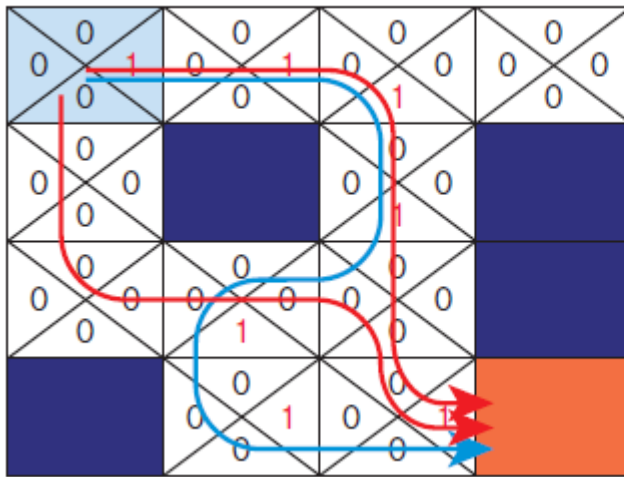
- $\epsilon$ -greedy 알고리즘에서는 epsilon 의 확률로 새로운 액션을 선택한다. (1- epsilon ) 확률로 기존의 Q 값을 선택한다.
- 여기서 epsilon 은 처음에는 크게, 반복이 진행되면 점점 작게 하는 것이 관행이다.

```
for i in range(10000):  
    epsilon = 0.1/(i+1)  
    if random.random() < epsilon:  
        action = random  
    else:  
        action = argmax(Q(s, a))
```

교과서 오타!



# $\epsilon$ -greedy 알고리즘



가끔씩 모험을 하면  
빨간색 경로를 발견할  
수도 있습니다.





# 할인(discount)된 보상

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

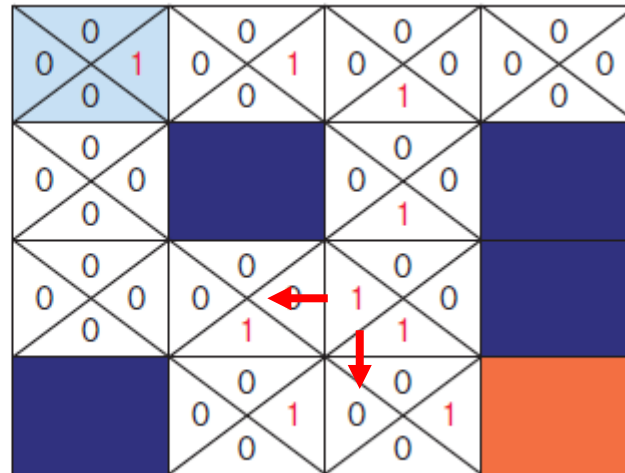
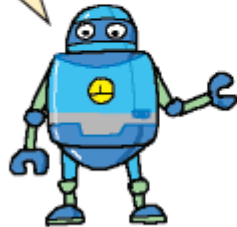
$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$



# 할인된 보상이 필요한 이유

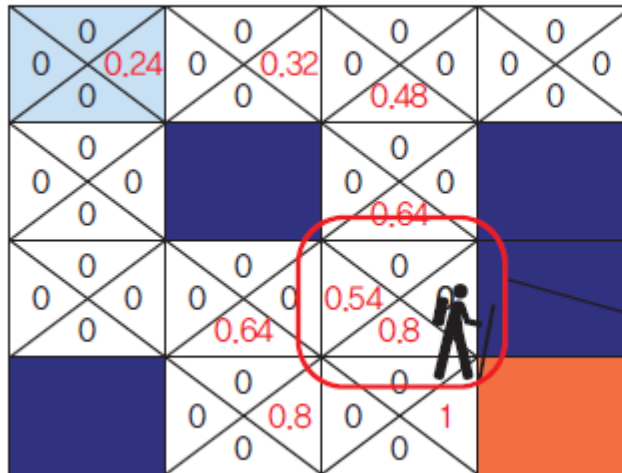
- 할인된 보상이 필요한 이유는 에이전트가 찾은 경로가 여러 개 있는 경우, 어떤 경로가 더 최단 경로인지를 판단해야 하기 때문이다

미래 보상이 할인되지  
않아서 어떤 경로가 더  
좋은지 알 수가 없군!





# 보상이 할인되었다면



에이전트가 상태 s11에 있다면  
DOWN 액션을 취할 수 있다.



- 
- |   |   |   |
|---|---|---|
| F | F | F |
| F | H | F |
| F | F | H |



# 최종적인 Q 값 업데이트 방정식

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a')]$$

기존의 Q 값

새로운 Q 값



$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

# 얼음 호수 게임에서 Q-학습의 구현

```
import numpy as np
import gym
import random
import time
import os

# FrozenLake 환경 생성
env = gym.make('FrozenLake-v1', is_slippery=False)

# 하이퍼 파라미터를 설정한다.
num_episodes = 10000
max_steps_per_episode = 100
learning_rate = 0.1
discount_rate = 0.99
exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.001
```

# 얼음 호수 게임에서 Q-학습의 구현

```
action_space_size = env.action_space.n
state_space_size = env.observation_space.n
q_table = np.zeros((state_space_size, action_space_size))
```

# 학습 과정

```
rewards_all_episodes = []
for episode in range(num_episodes):
    state, _ = env.reset()
    done = False
    rewards_current_episode = 0
```

```
    for step in range(max_steps_per_episode):
```

```
        # 탐험-활용 트레이드오프
```

```
        exploration_rate_threshold = random.uniform(0, 1)
```

```
        if exploration_rate_threshold > exploration_rate:
```

```
            action = np.argmax(q_table[state,:])
```

```
        else:
```

```
            action = env.action_space.sample()
```

# 얼음 호수 게임에서 Q-학습의

# 행동 수행

```
new_state, reward, done, _, _ = env.step(action)
if isinstance(new_state, tuple): # new_state가 정수인지 확인
    new_state = new_state[0]
```

# Q-table 업데이트

```
q_table[state, action] = q_table[state, action] * (1 - learning_rate) + \
    learning_rate * (reward + discount_rate *
np.max(q_table[new_state, :]))
```

state = new\_state

rewards\_current\_episode += reward

```
if done == True:
    break
```

# 탐험율 감소

```
exploration_rate = min_exploration_rate + \
    (max_exploration_rate - min_exploration_rate) * np.exp(-
    exploration_decay_rate*episode)
```

```
rewards_all_episodes.append(rewards_current_episode)
```







# 얼음 호수 게임에서 Q-학습의 구현

```
q_table[state, action] = q_table[state, action] * (1 - learning_rate) + \
    learning_rate * (reward + discount_rate * np.max(q_table[new_state, :]))

# 학습 완료 후 결과 출력
rewards_per_thousand_episodes = np.split(np.array(rewards_all_episodes),
num_episodes/1000)
count = 1000
print("*****천 에피소드당 평균 보상*****\n")
for r in rewards_per_thousand_episodes:
    print(count, ": ", str(sum(r/1000)))
    count += 1000
print("\n\n*****Q-table*****\n")
print(q_table)
```



# Q 테이블의 실제 모습

상태 \ 액션				
상태 s0	0	0	0	0
상태 s1	0	0	0	0
상태 s2	0	0	0	0
상태 s3	0	0	0	0
...	...			



# 실행결과

\*\*\*\*\*Average reward per thousand episodes\*\*\*\*\*

1000 :	0.231000000000000018
2000 :	0.736000000000000005
3000 :	0.915000000000000007
4000 :	0.954000000000000007
5000 :	0.985000000000000008
6000 :	0.983000000000000008
7000 :	0.984000000000000008
8000 :	0.986000000000000008
9000 :	0.990000000000000008
10000 :	0.993000000000000008



# Q 테이블을 출력해보자.

\*\*\*\*\*Q-table\*\*\*\*\*

```
[[0.94148015 0.95099005 0.93206534 0.94148015]
 [0.94148015 0.      0.42181731 0.86360308]
 [0.23652444 0.75587638 0.0182078 0.0623072 ]
 [0.13117909 0.      0.00567002 0.      ]
 [0.95099005 0.96059601 0.      0.94148015]
 [0.      0.      0.      0.      ]
 [0.      0.98008937 0.      0.1000408 ]
 [0.      0.      0.      0.      ]
 [0.96059601 0.      0.970299 0.95099005]
 [0.96059601 0.98009999 0.9801 0.      ]
 [0.97029887 0.99      0.      0.97018323]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.94240115 0.99      0.9336506 ]
 [0.98009987 0.98999953 1.      0.98009809]
 [0.      0.      0.      0.      ]]
```



```
env = gym.make('FrozenLake-v1', is_slippery=False, render_mode="human")

# 에이전트 테스트
for episode in range(3):
    state, _ = env.reset()
    done = False
    print("*****에피소드 ", episode+1, "*****")
    time.sleep(1)

    for step in range(max_steps_per_episode):
        env.render()
        time.sleep(0.3)

        action = np.argmax(q_table[state,:])
        new_state, reward, done, _, _ = env.step(action)
        env.render()

    if done:
        env.render()
        if reward == 1:
            print("****목표에 도달했습니다!****")
            time.sleep(2)
        else:
            print("****구멍에 빠졌습니다!****")
            time.sleep(2)
            os.system('clear')
        break
    state = new_state
env.close()
```

# 실행결과

\*\*\*\*\*에피소드 1 \*\*\*\*\*

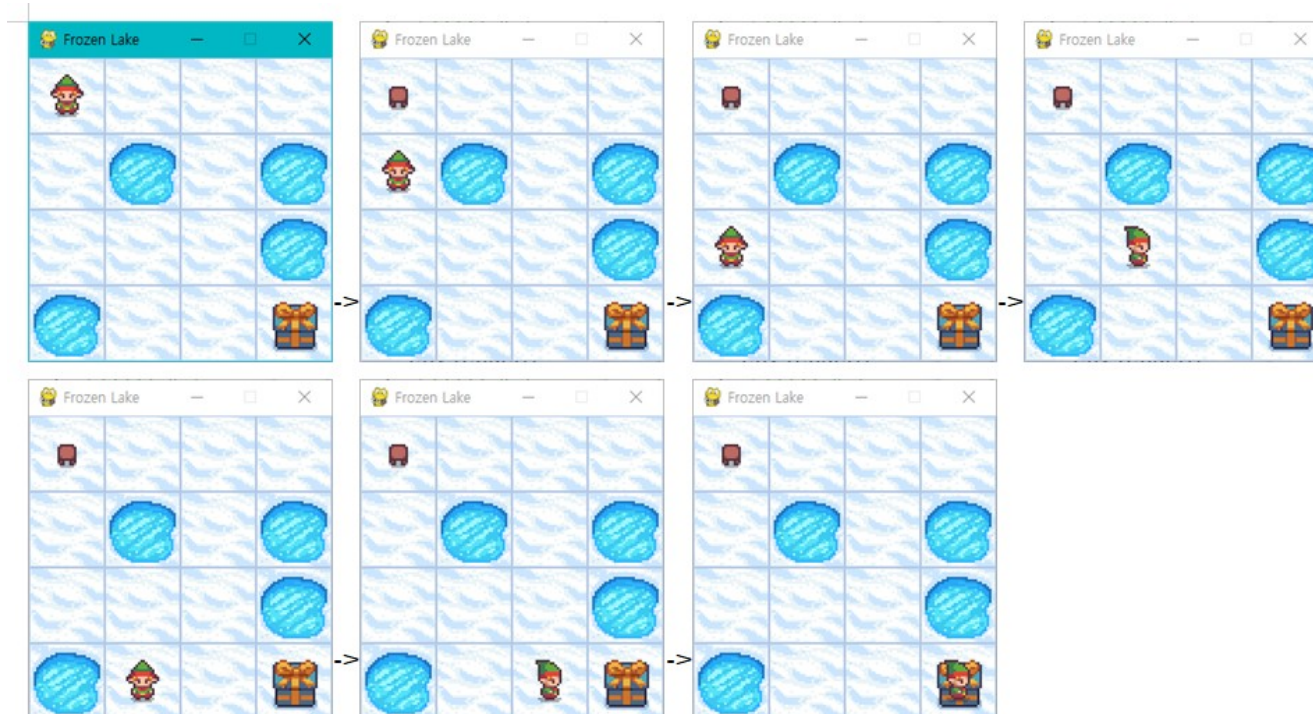
\*\*\*목표에 도달했습니다!\*\*\*

\*\*\*\*\*에피소드 2 \*\*\*\*\*

\*\*\*목표에 도달했습니다!\*\*\*

\*\*\*\*\*에피소드 3 \*\*\*\*\*

\*\*\*목표에 도달했습니다!\*\*\*





# Deep Q-학습

가치 학습(value learning)	정책 학습(policy learning)
$Q(s, a)$ 를 계산한다.	$\pi(s)$ 를 찾는다.
$a = \underset{a}{\operatorname{argmax}} Q(s, a)$	$\pi(s)$ 에서 액션 $a$ 를 샘플링한다.

첫 번째 방법은 신경망이 Q 함수를 학습한다. 우리는 Q 함수로부터 액션을 결정할 수 있다.

두 번째 경우는 신경망이 직접적으로 정책을 학습한다. 정책에서 바로 액션을 결정한다.  
두 번째 방법에서는 중간 단계의 Q 함수가 없다.



# 왜 신경망을 사용하는가?

- 전통적인 Q-학습은 에이전트를 위한 치트 시트를 만드는 간단하지만 강력한 알고리즘이다.
- 하지만 이 치트 시트가 너무 길면 어떻게 될까?
- 10,000개의 상태와 상태당 1,000개의 액션이 있는 환경을 상상해보자. 천만 개의 셀을 가지는 Q-테이블이 필요하다. 해당 테이블을 저장하고 업데이트하는 데 필요한 메모리 양은 상태 수가 증가함에 따라 감당할 수 없을 만큼 증가한다.



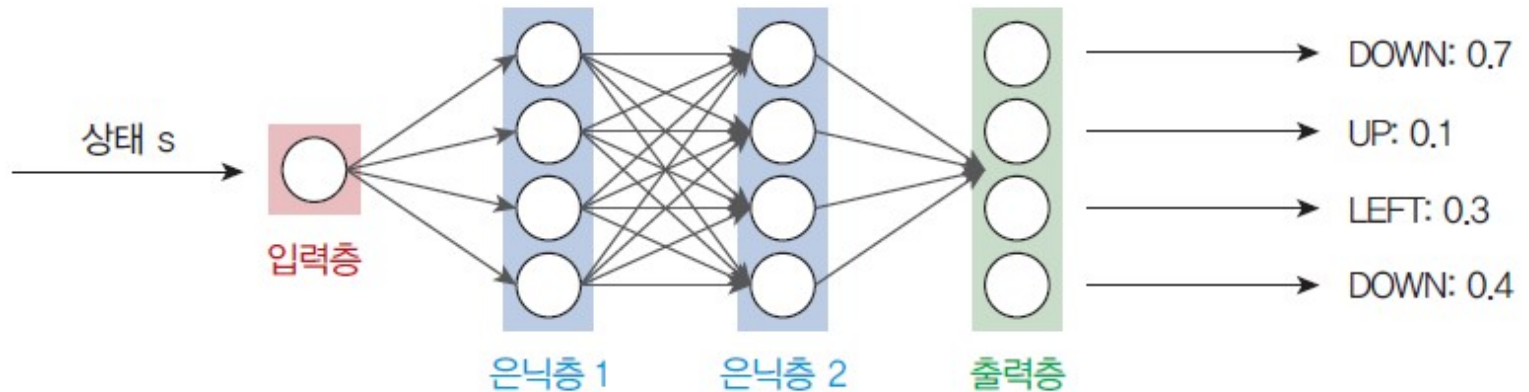
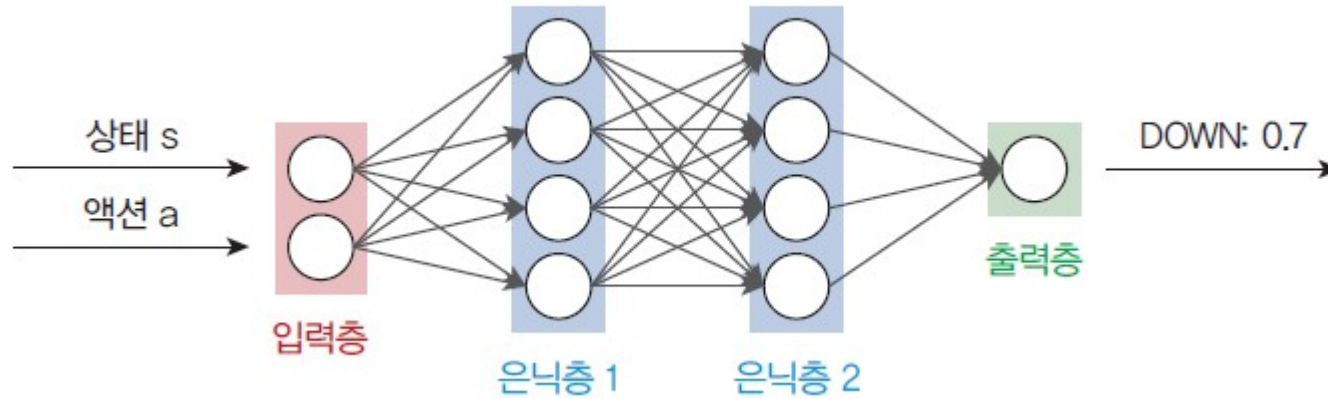




- 예를 들어서  $100 \times 100$  크기의 화면을 가지고 있는 비디오 게임의 경우, 한 픽셀이 8바이트라고 하면 상태의 수는 얼마나 될까?
- 하나의 픽셀이 가질 수 있는 상태의 값은 256개이고 이러한 픽셀이  $100 \times 100$ 개나 있으므로 무려  $256^{100 \times 100}$ 이나 된다.
- 이렇게 탐색 공간이 무척 큰 경우가 바로 심층 신경망이 가장 필요한 경우이다.

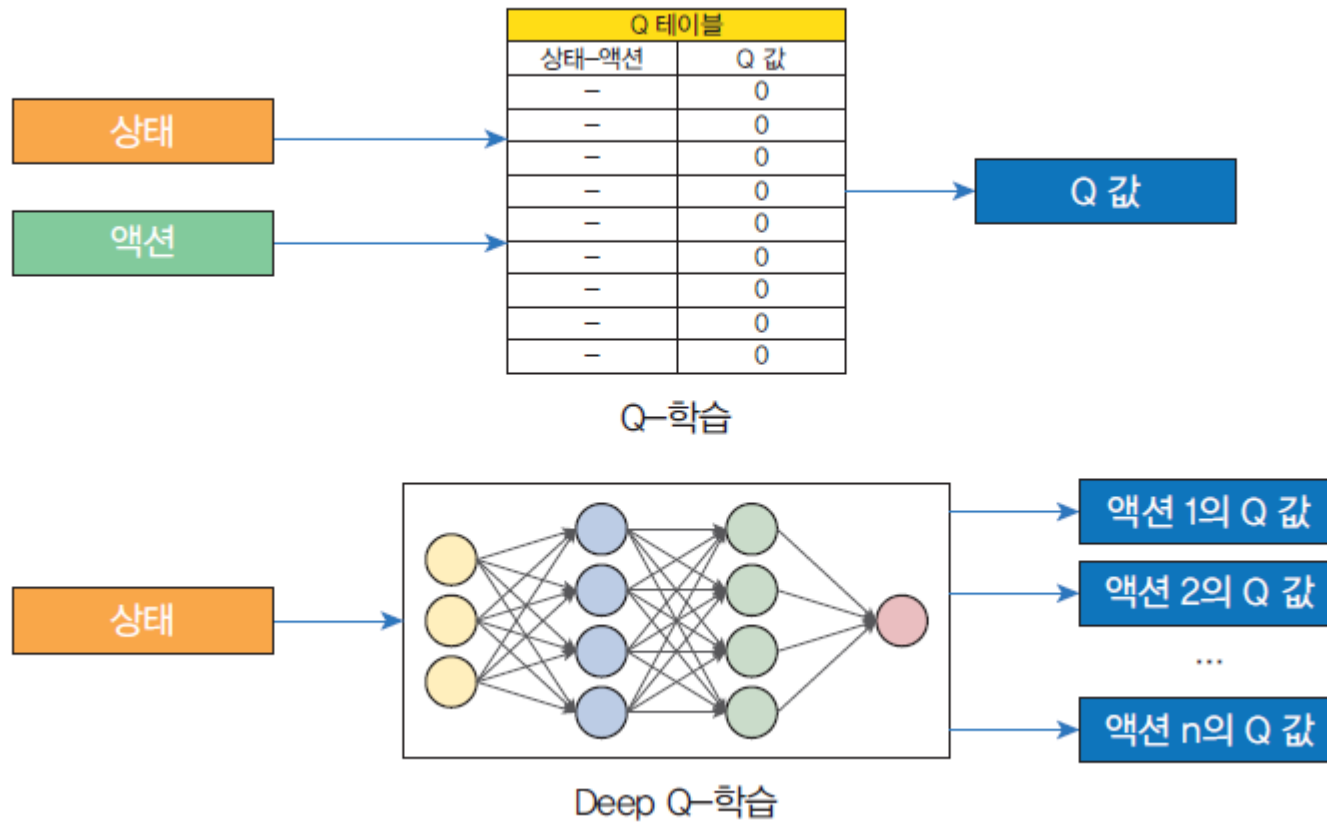


# DQN(Deep Q Network)





# Q-학습 vs Deep Q-학습





# 학습 방법

- 선형 회귀 신경망을 통하여 생성된 출력값을 예측값을  $Q(s, a)$ 라고 하자.
- 정답은 무엇일까? 특정한 액션  $a$ 를 실행한 후라면  $Q$  값은 정의에 의하여 다음과 같이 변경되어야 한다. 이것이 정답이 된다.

$$(r + \gamma \max_{a'} \hat{Q}(s', a'))$$

- 위의 값을 신경망이 생성한  $Q$  값과 비교하면서 차이를 줄이는 방향으로 가중치를 변경하면 된다.

$$E(\theta) = \sum_{t=0}^T \left[ \underbrace{\hat{Q}(s_t, a_t | \theta)}_{\text{예측값(predicted)}} - \underbrace{\left( r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}', a') \right)}_{\text{목표값(target)}} \right]^2$$



# 알고리즘

$Q(s, a)$  값을 난수로 초기화한다.

초기 상태  $s$ 를 얻는다.

**for**  $t=1, T$  **do**

**if** 난수  $< \epsilon$       액션  $a_t$ 를 랜덤하게 선택한다.

**else**       $a_t = \underset{a}{\operatorname{argmax}} Q^*(s_t, a | \theta)$

    액션  $a_t$ 를 실행하고 상태가 변경되고 보상  $r_t$ 를 받는다.

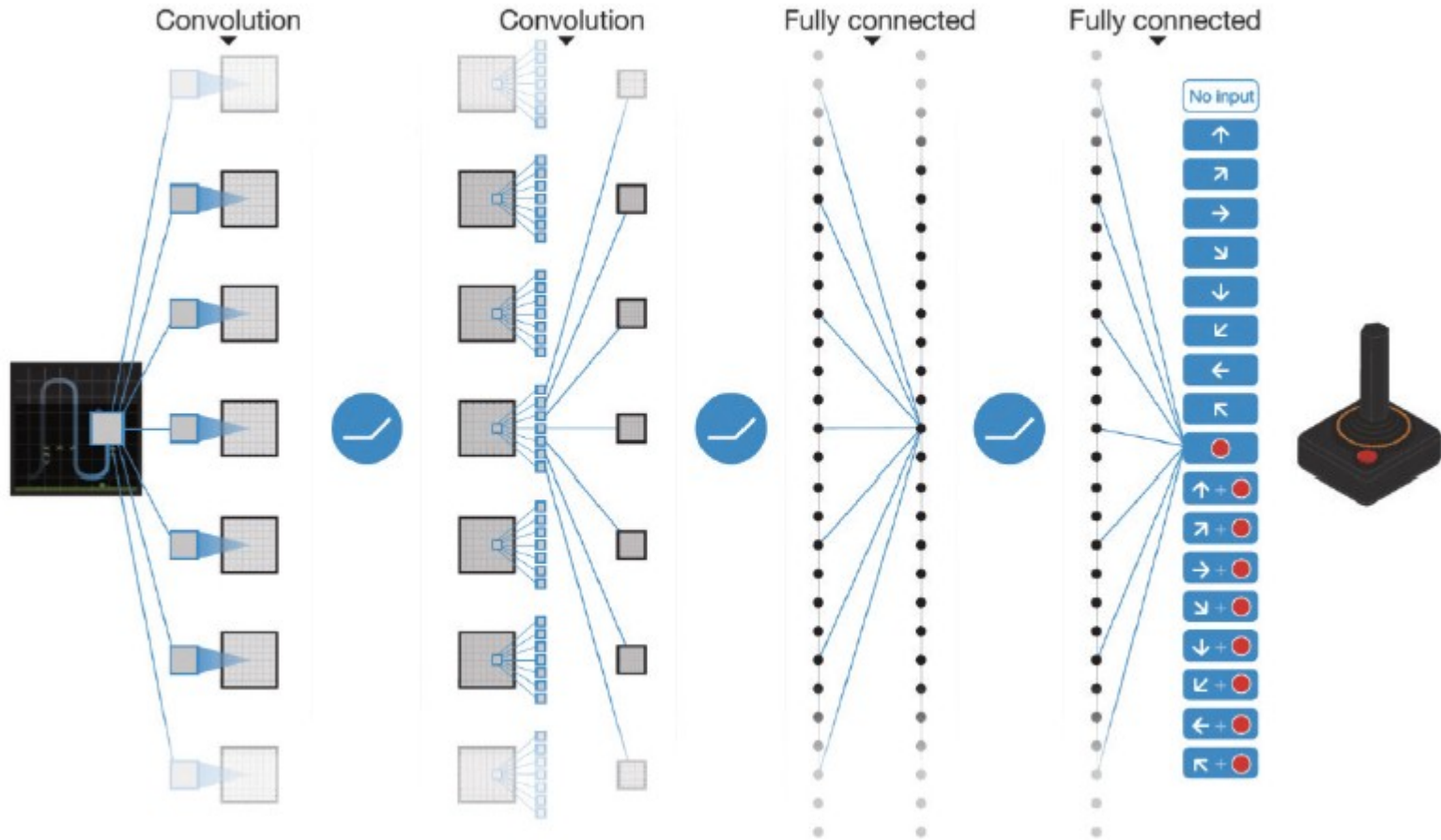
$$y_t = r_t + \gamma \underset{a'}{\operatorname{argmax}} Q(s_{t+1}, a' | \theta)$$

$(y_t - Q(s_t, a_t | \theta))^2$ 을 줄이기 위하여 경사 하강법을 사용한다.

하나의 액션이 수행되고 보상과 다음 상태가 나왔기 때문에 보다 정확한 Q값을 얻을 수 있다. 이것이 타겟(정확한 Q값)이 된다.  
타겟과 현재 Q값의 차이를 이용하여 학습시킨다.

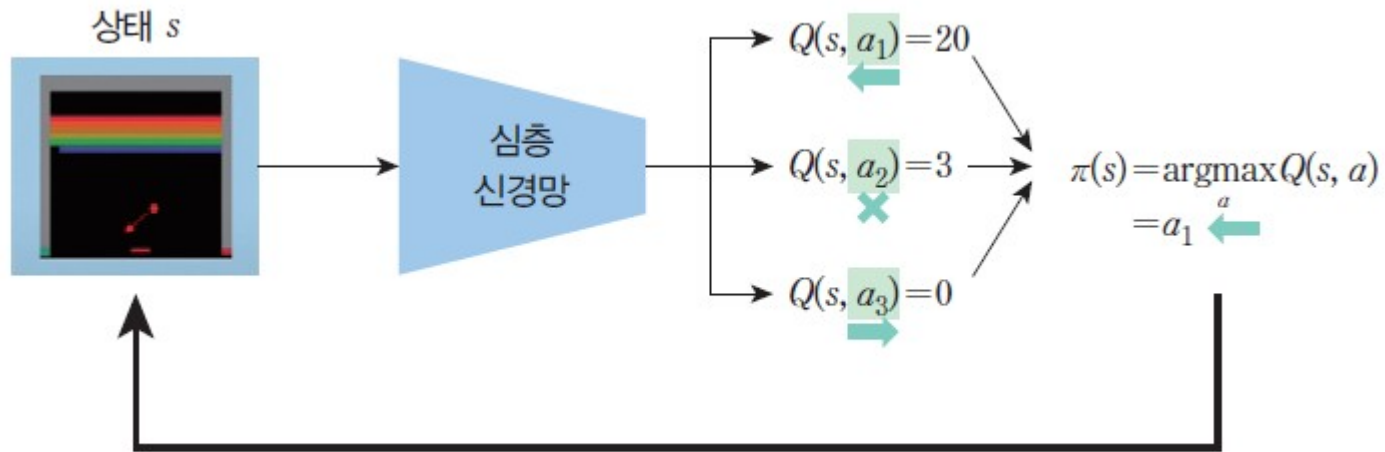


# 실제 적용 예: 벽돌 깨기 게임





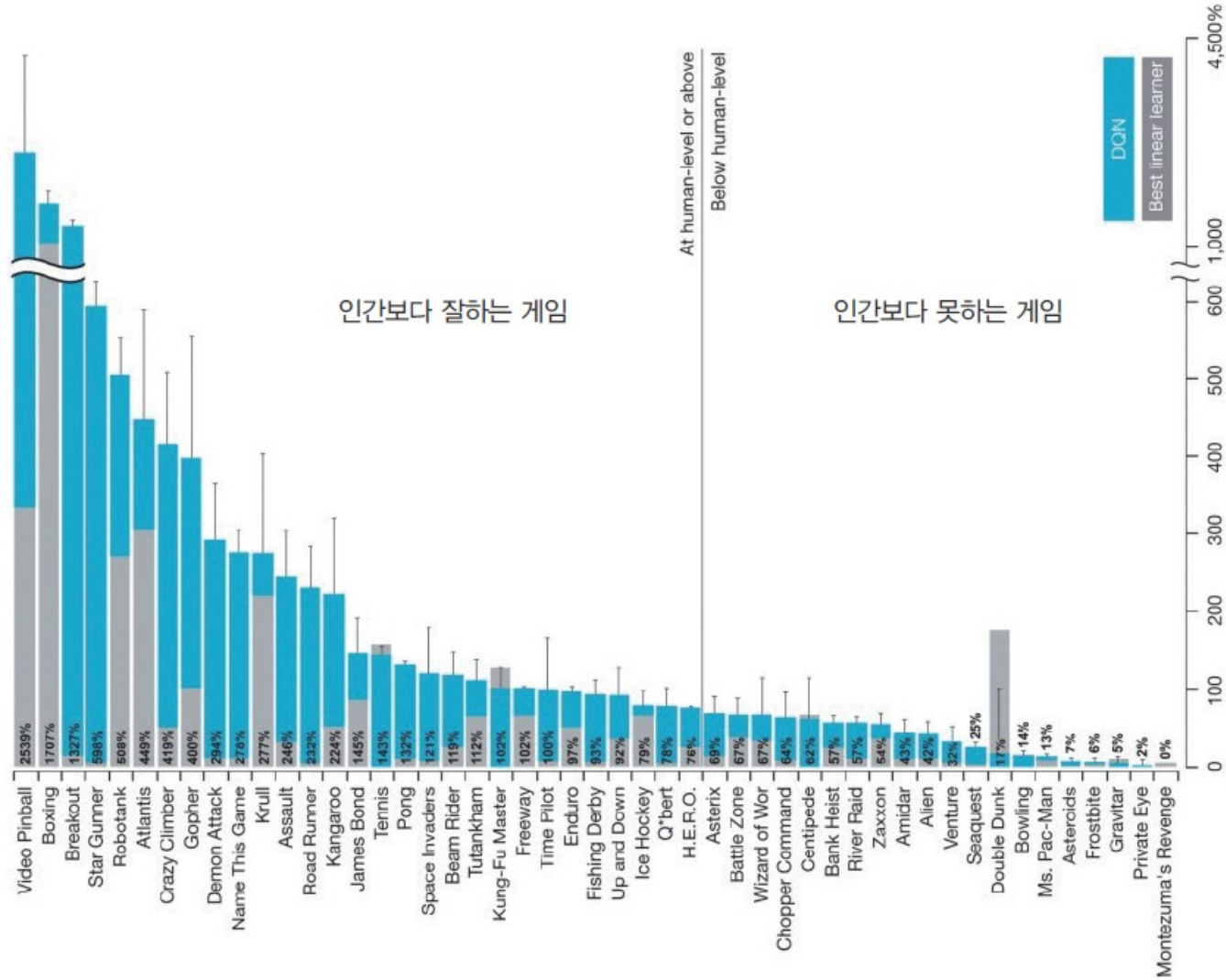
# 신경망의 학습



액션을 환경으로 보내고 다음 상태를 받는다.



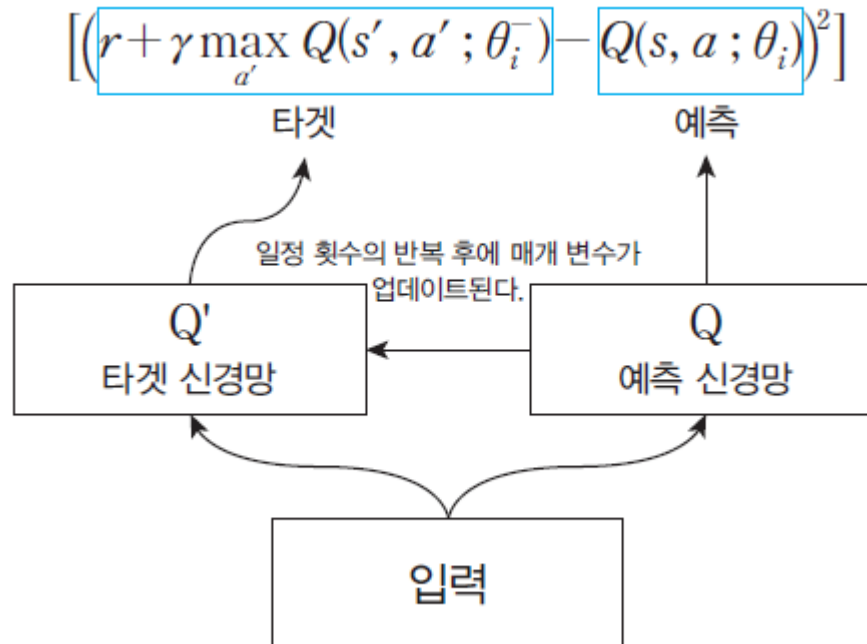
# 강화 학습을 이용한 게임의 성능





# 문제점

- Deep Q-학습에서는 약간의 문제가 있다. 우리는 목표 Q 값을 사실 정확히 알지 못한다. 그저 현재의 Q 값을 이용하여 추정할 뿐이다.
- 따라서 위의 알고리즘에서 볼 수 있듯이 반복할 때마다 목표가 변경된다.
- 이 문제를 해결하기 위하여 2개의 신경망을 사용하기도 한다.

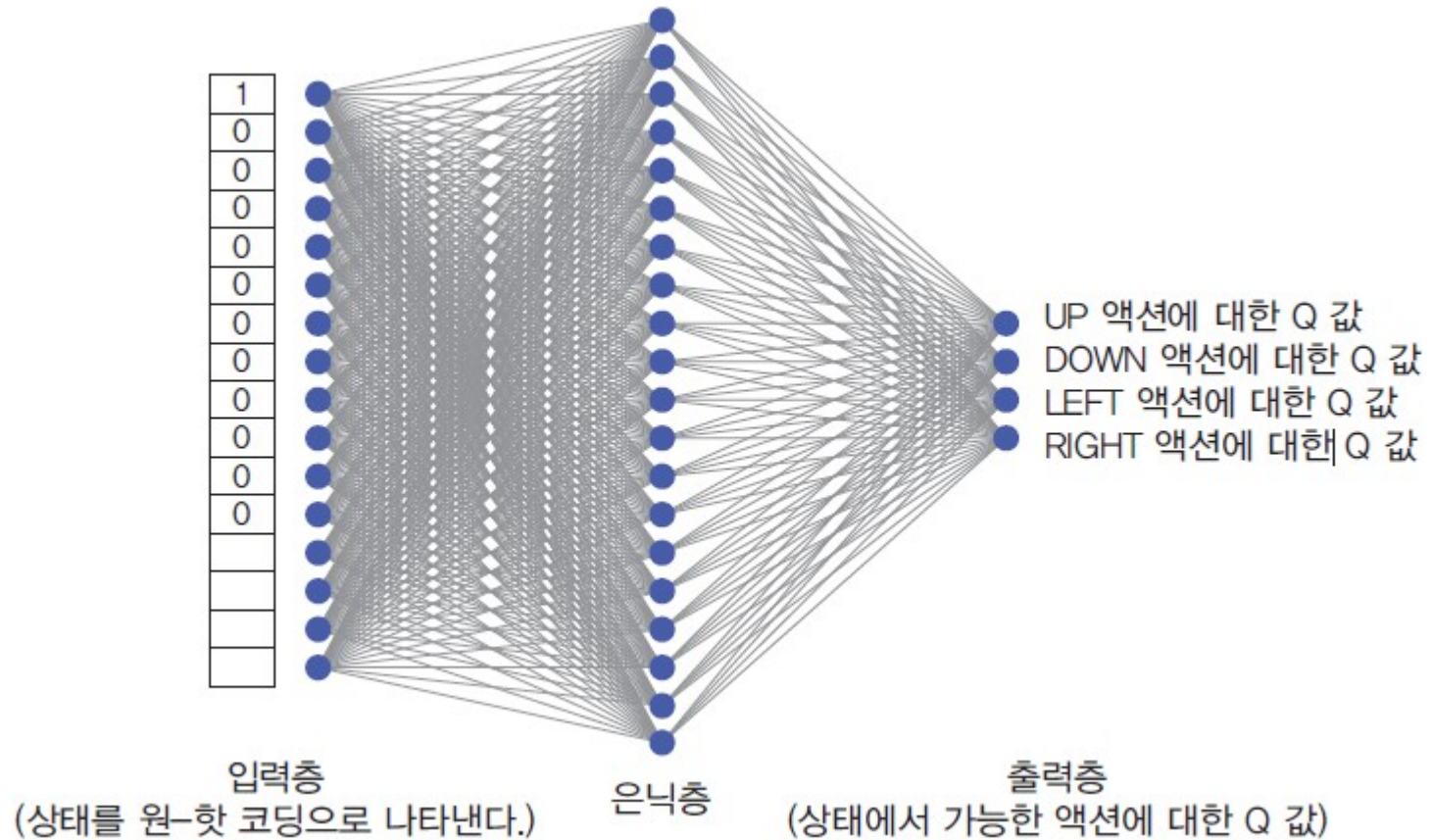


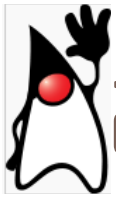


# 심층 Q-학습의 단점

- 액션 공간이 비연속적이고 작을 때는 가능, 하지만 연속적인 액션 공간은 처리가 불가능하다.
- 정책은 Q 함수로부터 결정적(deterministic)으로 계산된다. 따라서 확률적(stochastic)인 정책을 학습할 수 없다.

# 예제: 얼음 호수 게임에서 심층 Q-학습의 구현





# 리플레이 버퍼(replay buffer)

- 리플레이 버퍼는 학습 효율성과 안정성을 향상시키기 위해 RL(강화 학습) 알고리즘에서 일반적으로 사용되는 기술입니다. 환경에서 에이전트의 과거 경험 또는 "전환"의 기록을 저장하는 메모리 구조입니다.
- 각 전환은 에이전트의 상태, 에이전트가 수행한 작업, 환경에서 받은 보상 및 결과로 나타나는 다음 상태로 구성됩니다. 이러한 전환은 에이전트가 교육 중에 환경과 상호 작용할 때 수집됩니다.
- 학습하는 동안 RL 알고리즘은 리플레이 버퍼에서 무작위로 전환을 샘플링하고 이를 사용하여 에이전트의 정책 및 가치 함수를 업데이트합니다. 리플레이 버퍼에서 샘플링함으로써 에이전트는 다양한 경험 세트에서 학습하므로 지역 최적값에 갇히는 것을 방지하고 보다 효율적으로 학습할 수 있습니다.
- 리플레이 버퍼는 DQN(Deep Q-Networks)과 같은 심층 RL 알고리즘에서 일반적으로 에이전트가 경험에서 학습하는 온라인 학습의 한계를 극복하기 위해 사용됩니다. 리플레이 버퍼를 사용하면 에이전트가 과거 경험 배치에서 학습할 수 있으므로 학습 프로세스의 분산을 줄이고 학습된 정책의 안정성을 개선하는 데 도움이 됩니다.



# 예제: 얼음 호수 게임에서 심층 Q-학습의 구현

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import models, layers, optimizers
import random
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # 모든 로그 메시지를 제거
tf.get_logger().setLevel('ERROR')

env = gym.make('FrozenLake-v1', is_slippery=False)
```



# 예제: 얼음 호수 게임에서 심층 Q-학습의 구현

```
num_episodes = 300          # 총 에피소드 수 (학습 반복 횟수)
learning_rate = 0.001        # 학습률 (신경망 가중치 업데이트의 크기)
discount_factor = 0.95       # 할인율 (미래 보상의 현재 가치)
epsilon = 1.0                # 초기 탐험률 (무작위 행동 선택 비율)
epsilon_decay = 0.995        # 탐험률 감소 비율 (매 에피소드마다 탐험률 감소)
min_epsilon = 0.01           # 최소 탐험률 (탐험률의 하한선)
batch_size = 64              # 미니배치 크기 (리플레이 메모리에서 샘플링되는 경험의 수)
memory_size = 2000           # 리플레이 메모리 크기 (저장되는 최대 경험 수)

# 리플레이 메모리
memory = []
```



# 예제: 얼음 호수 게임에서 심층 Q- 학습의 구현

```
def build_model(input_shape, output_shape):  
    model = models.Sequential()  
    model.add(layers.Input(shape=(input_shape,)))  
    model.add(layers.Dense(24, activation='relu'))  
    model.add(layers.Dense(24, activation='relu'))  
    model.add(layers.Dense(output_shape, activation='linear'))  
    model.compile(optimizer=optimizers.Adam(learning_rate=learning_rate), loss='mse')  
    return model
```

# 예제: 얼음 호수 게임에서 심층 Q- 학습의 구현

```
input_shape = env.observation_space.n # 입력 형태 (환경의 상태 공간 크기)
output_shape = env.action_space.n    # 출력 형태 (환경의 행동 공간 크기)

model = build_model(input_shape, output_shape) # 신경망 모델 생성

# 상태를 원-핫 인코딩하는 함수
def one_hot_state(state):
    one_hot = np.zeros(input_shape)
    one_hot[state] = 1
    return one_hot
```



# 예제: 얼음 호수 게임에서 심층 Q-학습의 구현

```
for episode in range(num_episodes): # 총 에피소드 수 만큼 반복
    state, _ = env.reset()          # 환경을 초기 상태로 리셋
    state = one_hot_state(state)    # 상태를 원-핫 인코딩
    done = False                    # 에피소드 종료 여부 초기화
    total_reward = 0                # 총 보상 초기화

    print("에피소드", episode)      # 현재 에피소드 번호 출력
    while not done:                 # 에피소드가 끝날 때까지 반복
        if np.random.rand() < epsilon: # 무작위 행동 선택 (탐험)
            action = env.action_space.sample()
        else:                        # Q-네트워크를 통한 행동 선택 (활용)
            q_values = model.predict(state.reshape(1, -1))
            action = np.argmax(q_values[0])

        next_state, reward, done, _, _ = env.step(action) # 행동 수행 후 다음 상태, 보상,
        종료 여부 가져옴
        next_state = one_hot_state(next_state) # 다음 상태를 원-핫 인코딩
        total_reward += reward                # 총 보상에 현재 보상 추가
```



# 예제: 얼음 호수 게임에서 심층 Q-

```
if done:                                # 에피소드가 끝났다면
    reward = -1 if reward == 0 else reward # 실패시 보상 조정
```

```
memory.append((state, action, reward, next_state, done)) # 경험을 리플레이
메모리에 저장
```

```
if len(memory) > memory_size: # 메모리가 가득 찼다면
    memory.pop(0)              # 가장 오래된 경험 제거
```

```
if len(memory) >= batch_size: # ①메모리가 충분히 채워졌다면
    minibatch = random.sample(memory, batch_size) # 무작위로 미니배치 샘플링
    states, actions, rewards, next_states, dones = zip(*minibatch) # ②
    states = np.vstack(states) # ③
    next_states = np.vstack(next_states)
    targets = model.predict(states) # ④
    next_q_values = model.predict(next_states)
    for i in range(batch_size): # ⑤각 미니배치에 대해 타겟 계산
        target = rewards[i]
        if not dones[i]:
            target += discount_factor * np.max(next_q_values[i])
        targets[i][actions[i]] = target
    model.fit(states, targets, epochs=1, verbose=0) # ⑥ 모델 학습
```

```
state = next_state # ⑦ 상태 업데이트
```



# 예제: 얼음 호수 게임에서 심층 Q- 학습의 구현

```
epsilon = max(min_epsilon, epsilon * epsilon_decay) #⑧ 탐험률 감소
```

```
if episode % 100 == 0:          # 매 100 에피소드마다 출력  
    print(f"Episode: {episode}, Total reward: {total_reward}, Epsilon: {epsilon}")
```



# 실행 결과

에피소드 1

에피소드 2

...

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 19ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 16ms/step

[1m1/1[0m [32m—————[0m[37m[0m [1m0s[0m 23ms/step

성공 횟수: 100/100

# 실행 결과





# Summary

- 강화 학습(Reinforcement Learning)에서는 에이전트가 어떤 행동을 취할 때마다 외부에서 처벌이나 보상이 주어진다. 컴퓨터는 이 보상을 최대화하는 방향으로 학습을 진행시킨다.
- 상태(state) 는 에이전트의 현재 상황이다. 보상(reward) 은 환경으로부터의 피드백이다. 액션(action) 는 에이전트의 행동이다.
- 강화 학습에서 보상은 “할인된 보상”이다. 미래의 보상에는 할인 계수  $\lambda$ 를 곱하여 총 보상을 계산한다.
- Q 함수는 상태  $s$ 에 있는 에이전트가 어떤 액션  $a$ 를 실행하여서 얻을 수 있는 미래 보상값의 기대값이다.
- 탐사는 가끔 모험을 하는 것이고 활용은 기존의 Q 값을 사용하는 것이다.
- 심층 Q-학습은 Q 값을 심층 신경망을 이용하여 학습한다. 우리는 신경망을 사용하여 이러한 Q 값을 근사할 수 있다. 이러한 신경망을 DQN(Deep Q Network)이라고 한다.



# Q & A

