

13장 강화 학습

강화 학습이란?

- 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

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

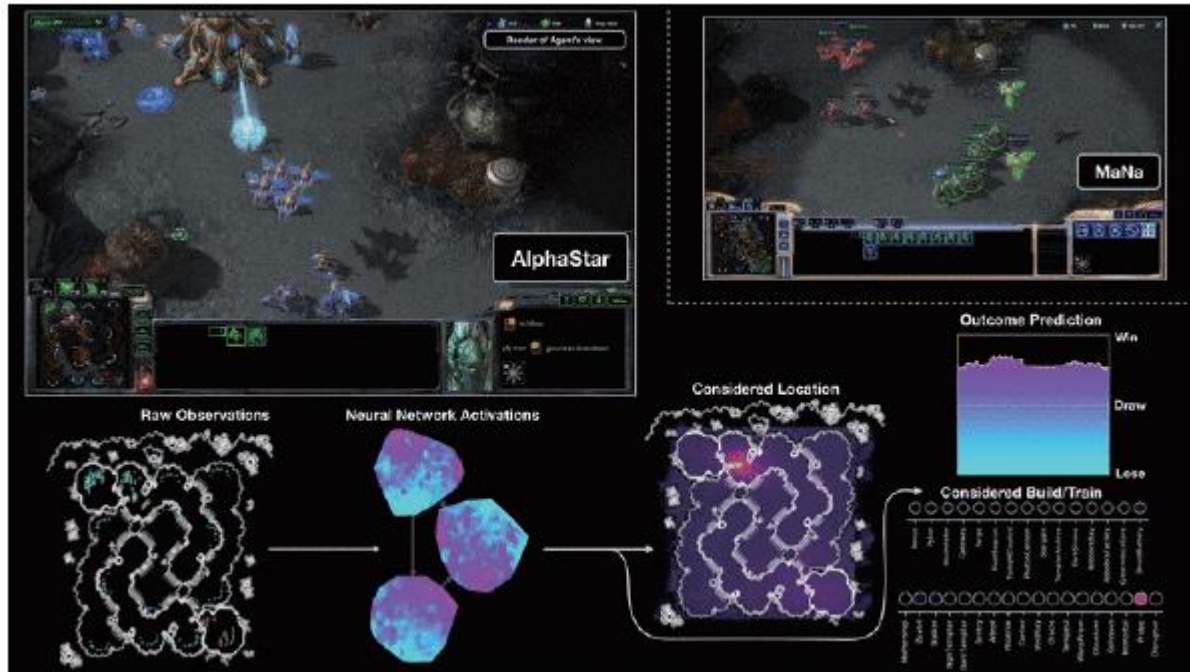
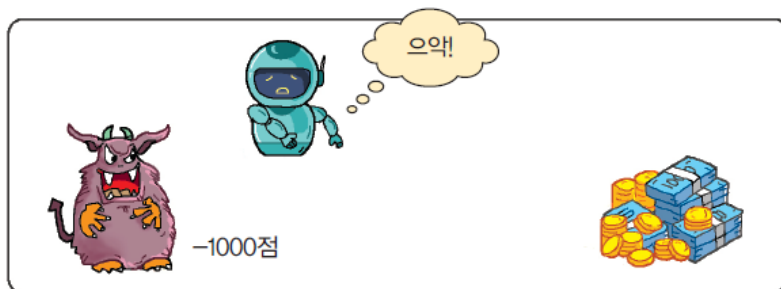
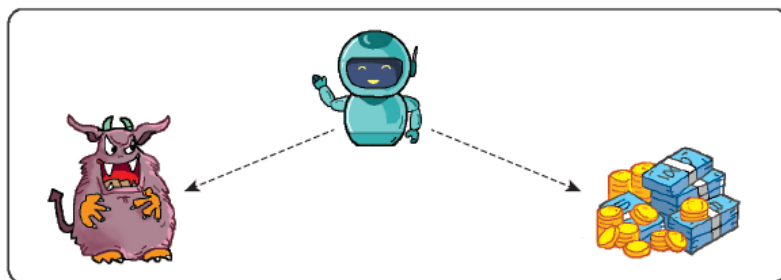


그림 13-2 스타크래프트 게임을 수행하는 알파스타



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

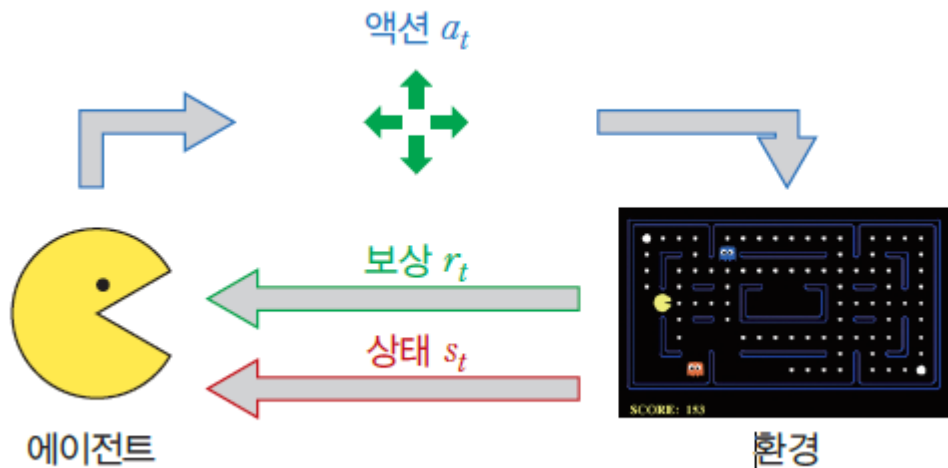
- 에이전트(agent): 강화 학습의 중심이 되는 객체
- 환경(Environment): 에이전트가 작동하는 물리적 세계
- 상태(state): 에이전트의 현재 상황, 미로에서의 에이전트의 위치가 상태일 수 있다.
- 보상(reward) : 환경으로부터의 피드백,
- 액션(action) : 에이전트의 행동

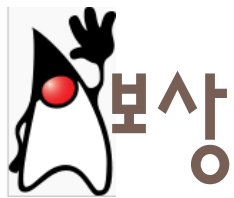




게임에서의 강화학습

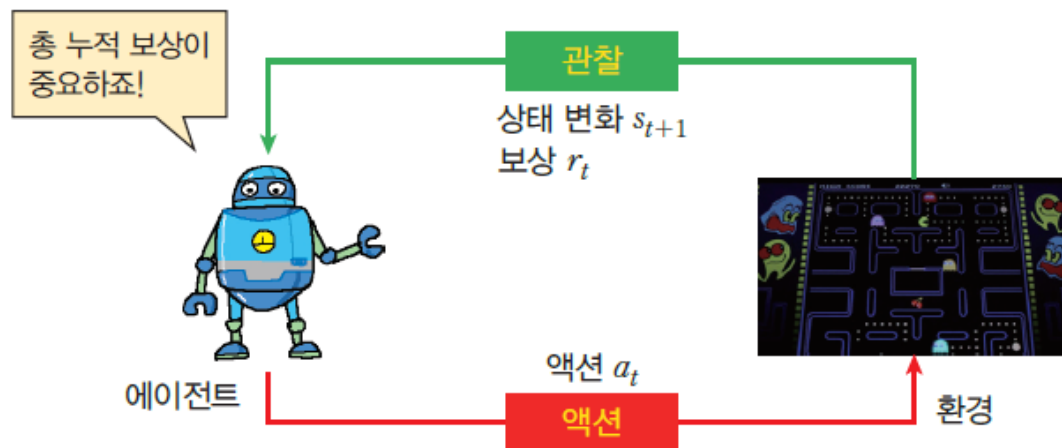
- 강화 학습에서 에이전트는 환경 안에서 자신의 보상을 극대화하려고 한다. 보상은 성공 또는 실패에 대한 피드백이다. 에이전트가 행동할 때마다 보상을 받을 필요는 없지만, 보상이 지연될 수 있다. 즉 마지막에 하나의 보상만을 받는 경우도 많다.





- 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.





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

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



- 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 아니라면 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 $\pi(s)$ 을 필요로 한다.
- 가장 상식적인 정책은 미래 보상을 최대화할 수 있는 액션을 선택하는 것이다.

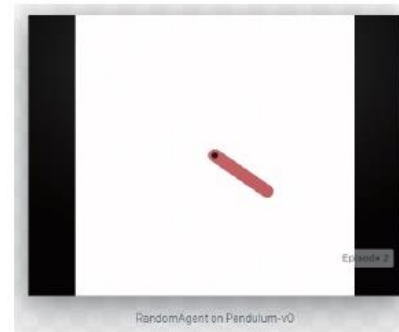
$$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

- 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://gym.openai.com/>)가 유명하다



(a) 랜덤 보행 게임



(b) CartPole 게임



(c) 스페이스 인베이더 게임



(d) 루나 랜더 게임

A stylized logo for the CartPole game, featuring a black cart with a red wheel and a black pole with a red ball at the top, all on a white background.

CartPole 게임

```
import gym

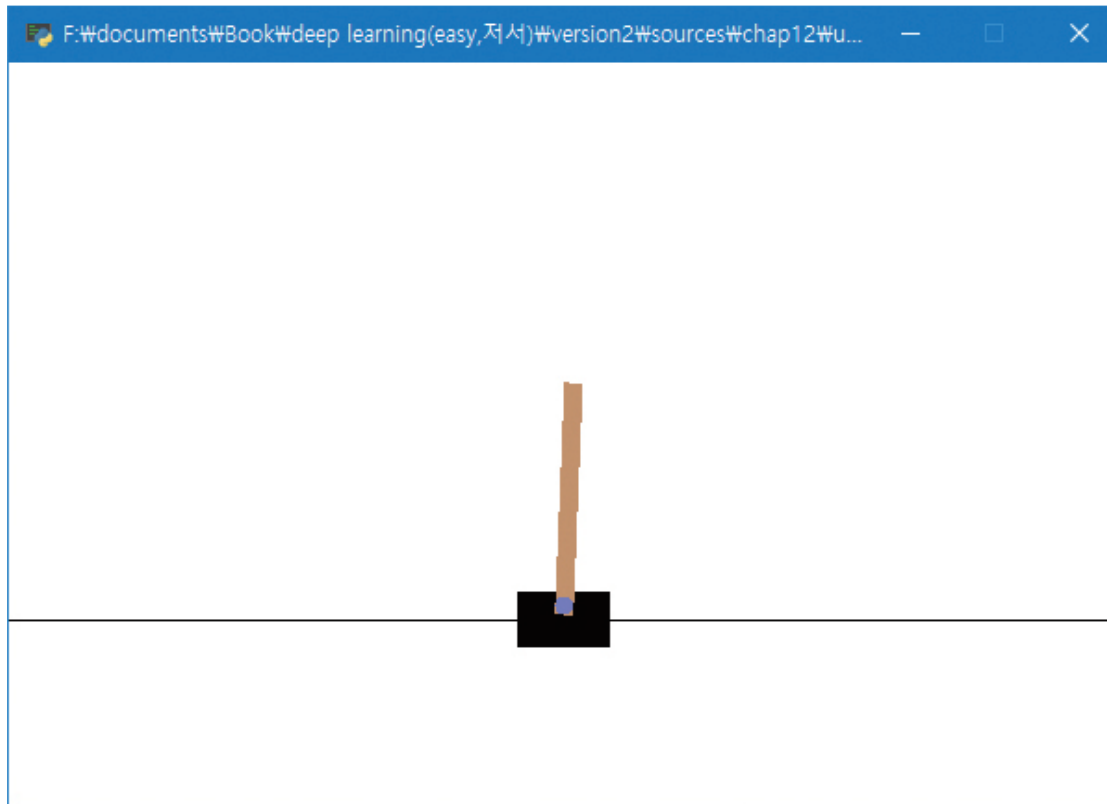
env = gym.make("CartPole-v1")          # (1)
observation = env.reset()              # (2)

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

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

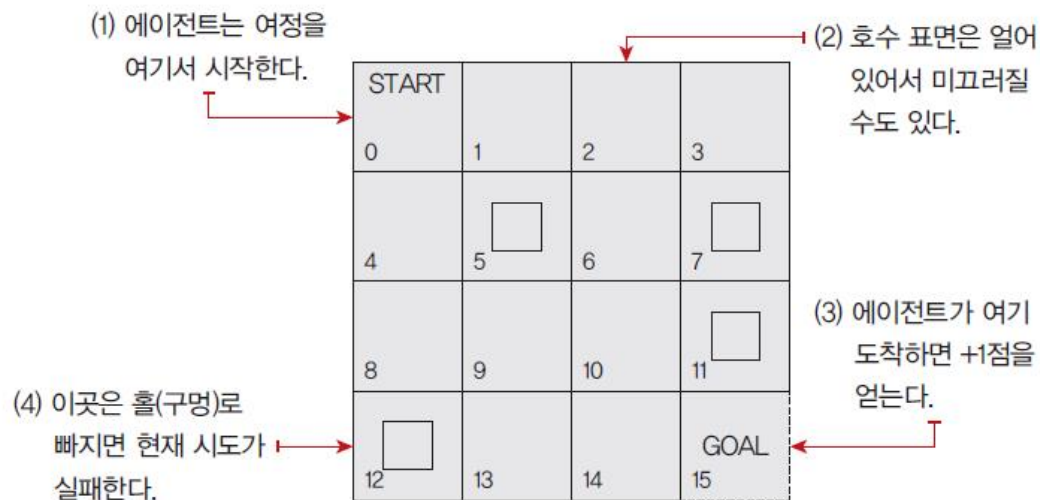
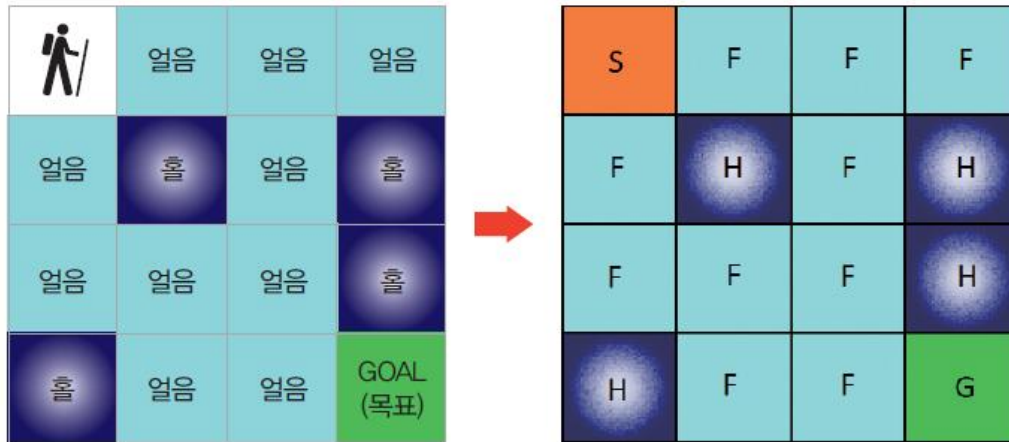


CartPole 게임





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-v0", is_slippery=False)
observation = env.reset()

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

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



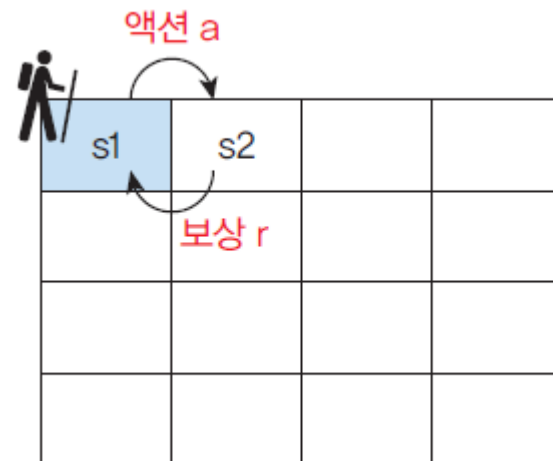
FrozenLake 게임



- 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

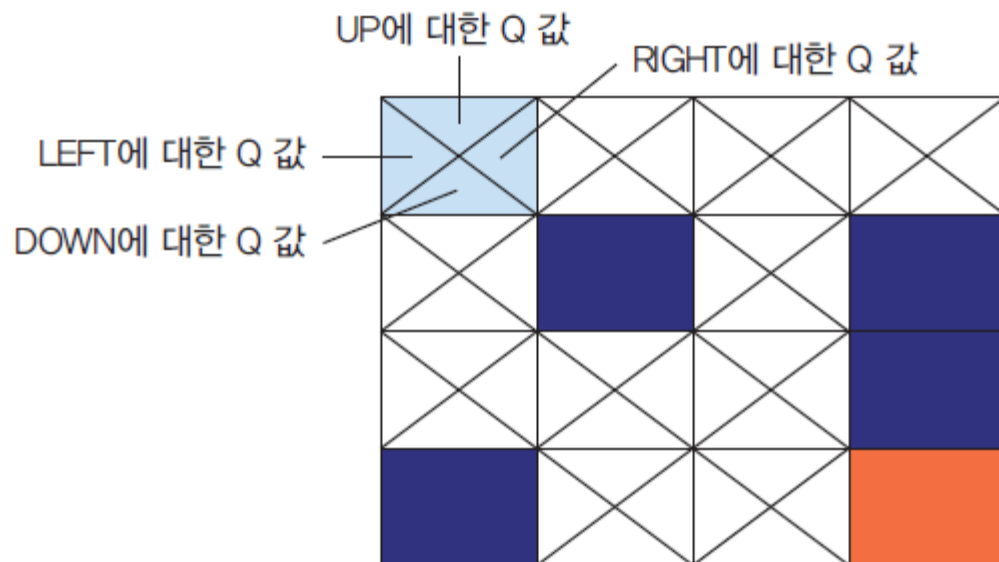
```
int fib(int n)
{
    int f[n+2]; int i;

    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 계산하는 문제



Q 값을 저장하는 배열을 생성한다.



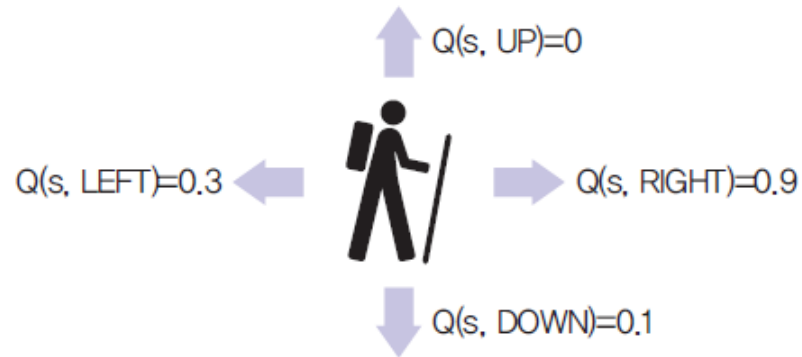


- 어떤 상태에서 특정한 행동을 하여서 받은 총 보상값을 Q 함수라고 한다.
- Q 함수는 에이전트의 현재 상태와 에이전트가 실행하는 액션을 받아서 총 보상값을 반환하는 함수이다.



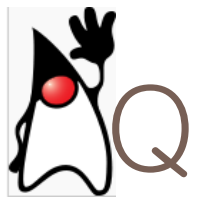


- 예를 들어서 특정한 상태 s 에서 다음과 같이 Q 값이 계산되었다고 하자.



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

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



가치 순환 관계식

- 총 보상은 다음과 같이 순환적으로 계산할 수 있다.

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



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

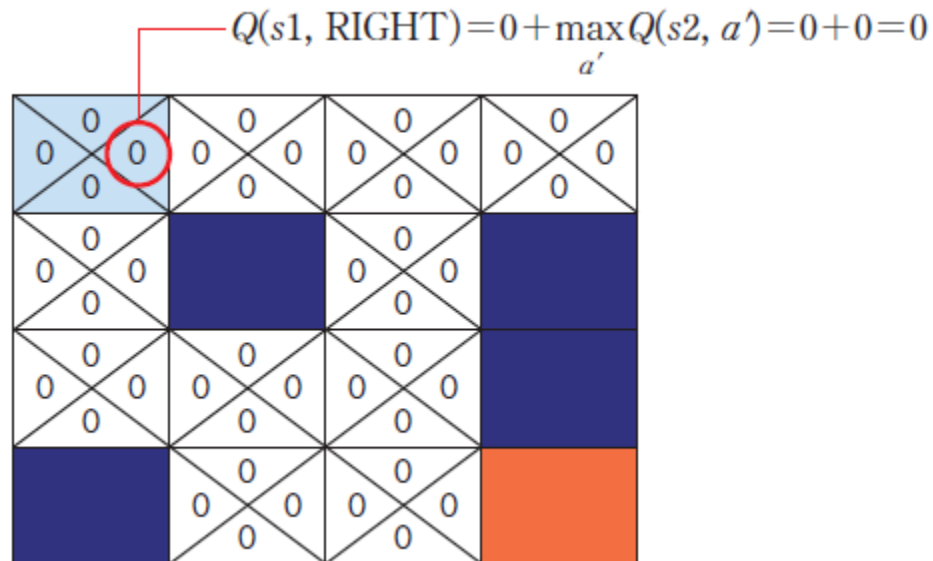
- 이것과 유사하게 상태 s 에서의 Q 값은 다음과 같이 순환적으로 계산할 수 있다. 즉 상태 s 에서 액션 a 를 실행하였을 때 받는 보상 r 에, 다음 상태에서의 Q 값 중에서 최대값을 더하게 된다.

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

가장 중요한 수식이다. 전통적인 Q-학습에서는 결국 이 순환 관계식을 사용하여 테이블 내의 Q 값들이 계속 업데이트된다

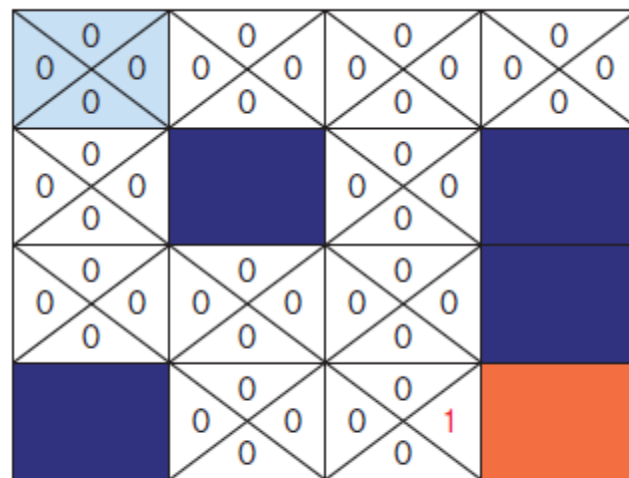
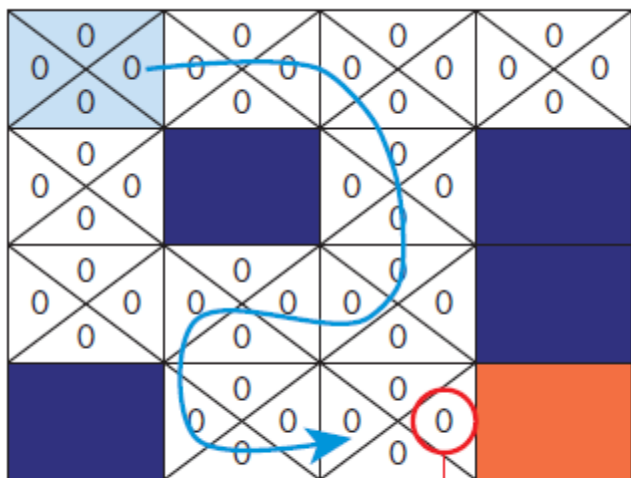
어음 함수 문제에서 실제로 Q 값을 계산해보자.

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





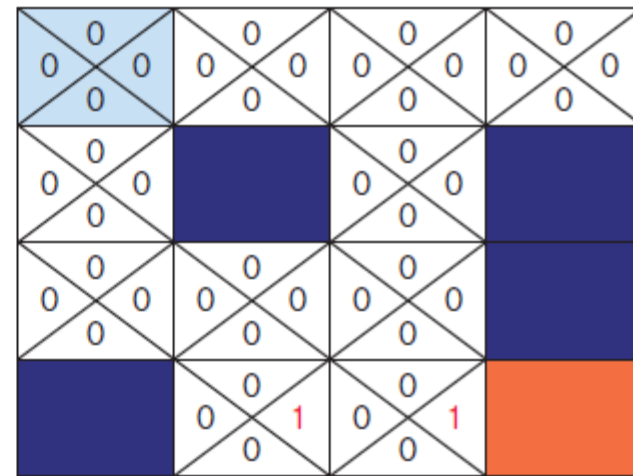
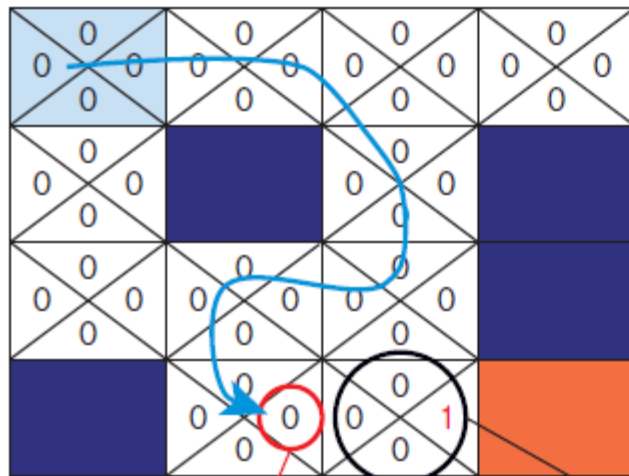
계속 Q 값은 0이 되지만 바뀔 것이다.



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



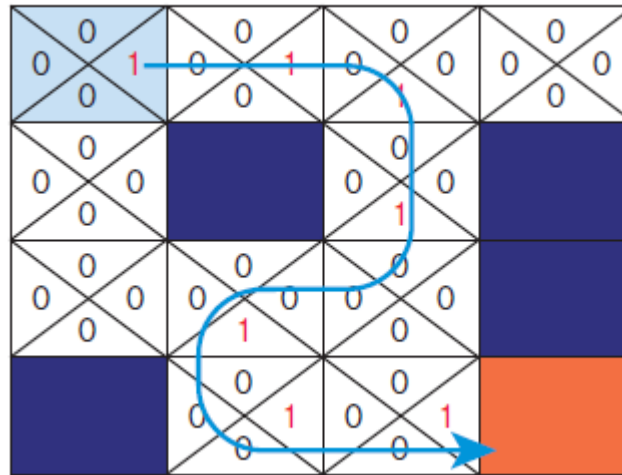
상태 S14에서의 Q 값 계산



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

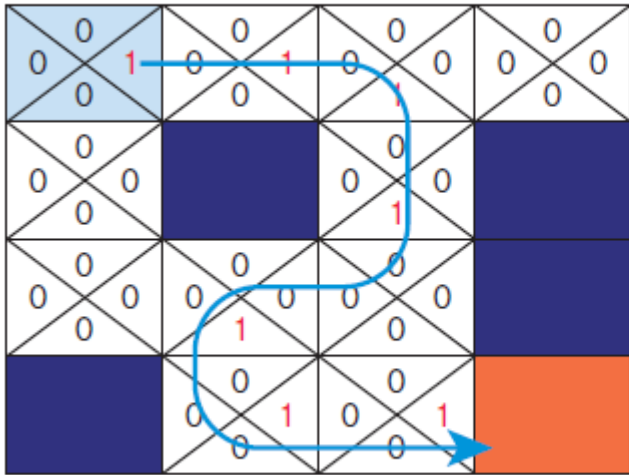


- 이런 식으로 계속 Q 값이 업데이트된다. 따라서 에피소드를 많이 진행하면 다음과 같이 Q 값이 설정될 수 있다.





- 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 e-greedy 알고리즘으로 가능하다.



처음에는 여기 저기 모험을 해볼
필요가 있다.

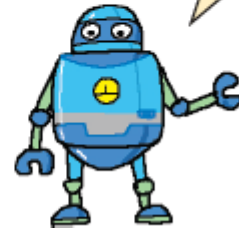
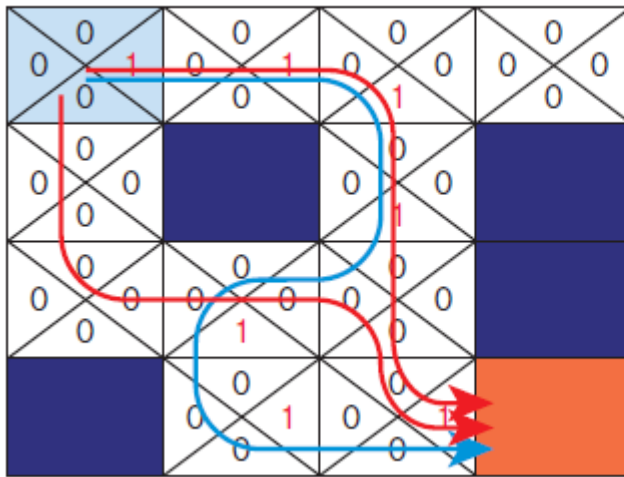


e-greedy 알고리즘

- e-greedy 알고리즘에서는 **epsilon** 의 확률로 새로운 액션을 선택한다. $(1 - \text{epsilon})$ 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))
```

고아서 오타!



가끔씩 모험을 하면
빨간색 경로를 발견할
수도 있습니다.



할인(discount)된 보상

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

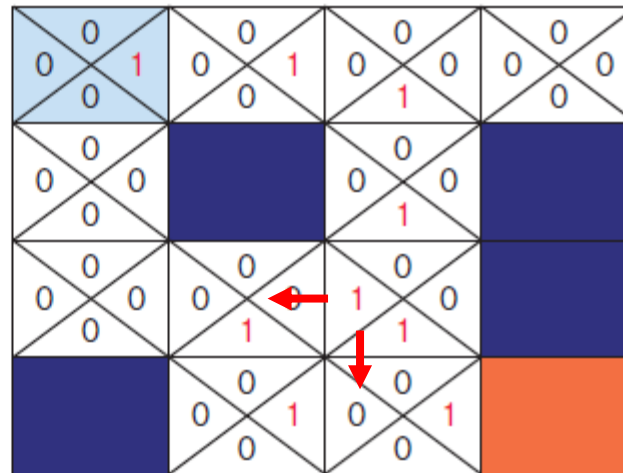
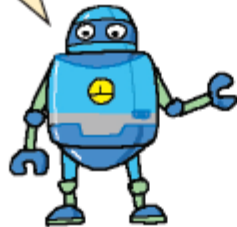
$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$



할인된 보상이 필요한 이유

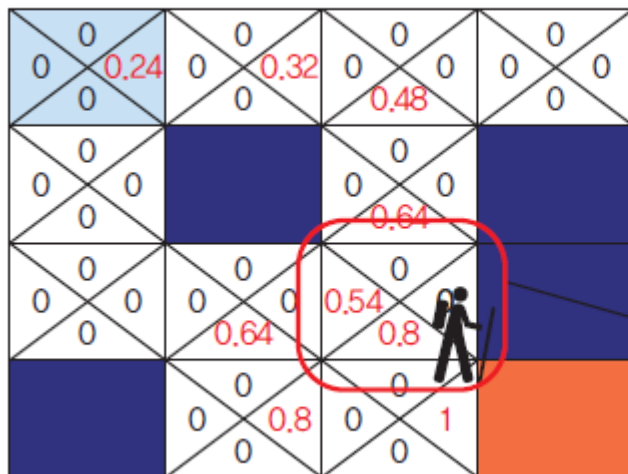
- 할인된 보상이 필요한 이유는 에이전트가 찾은 경로가 여러 개 있는 경우, 어떤 경로가 더 최단 경로인지를 판단해야 하기 때문이다

미래 보상이 할인되지
않아서 어떤 경로가 더
좋은지 알 수가 없군!





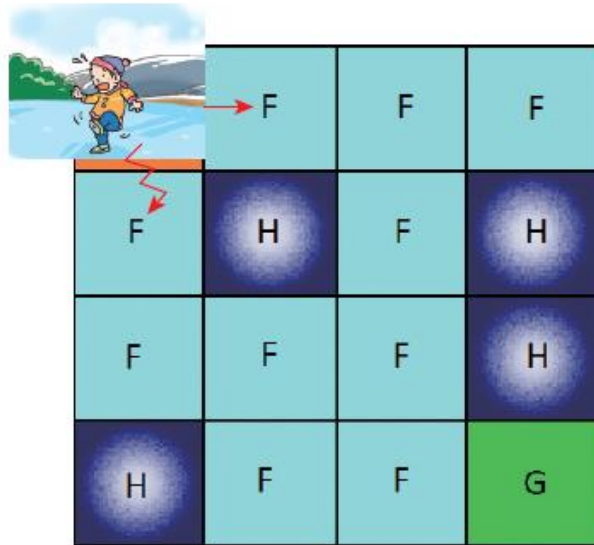
보상이 할인되어 다면

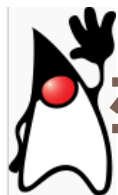


에이전트가 상태 s11에 있다면
DOWN 액션을 취할 수 있다.

확률적인 환경

- 앞에서 살펴본 Q-학습은 환경이 결정된 환경에서는 잘 작동한다. 하지만 확률적인 환경에서는 전혀 학습이 되지 않는다.
- 확률적인 환경이란 액션을 실행하였을 때 에이전트가 의도한 대로 가지 않을 수도 있는 환경이다.





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



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



어둠 호수 게임에서 Q-학습의 구현

```
import gym
import numpy as np

env = gym.make('FrozenLake-v0', is_slippery=False)

discount_factor = 0.95
epsilon = 0.5
epsilon_decay_factor = 0.999
learning_rate = 0.8
num_episodes = 30000
```



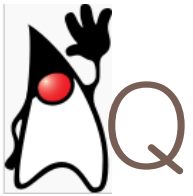
어둠 호수 게임에서 Q-학습의 구현

```
q_table = np.zeros([env.observation_space.n, env.action_space.n])





for i in range(num_episodes):
    state = env.reset()
    epsilon *= epsilon_decay_factor      # 입실론: 탐사와 활용 비율 결정
    done = False

    while not done:
        if np.random.random() < epsilon:      # 난수가 입실론보다 작으면 탐사
            action = env.action_space.sample() # 랜덤 액션
        else:                                  # 난수가 입실론보다 작으면 활용
            action = np.argmax(q_table[state, :]) # Q 테이블에서 가장 큰 값

        new_state, reward, done, _ = env.step(action)
        # 새로 얻은 정보로 Q-테이블 갱신
        q_table[state, action] += learning_rate * (reward + discount_factor *
np.max(q_table[new_state, :]) - q_table[state, action])
        state = new_state
    if i==(num_episodes-1):
        env.render()
```



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
...	...			



```
FHFFH
FFFH
HFFG
(Right)
SFFF
FHFFH
FFFH
HFFG
(Down)
SFFF
FHFFH
FFFH
HFFG
(Right)
SFFF
FHFFH
FFFH
HFFG
(Right)
SFFF
FHFFH
FFFH
HFFG
```

서고저이 하삼의 경우



테이블을 출력해보자.

```
>>> print(q_table)
```

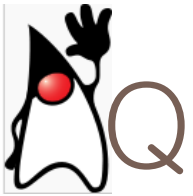
```
[[0.73509189 0.77378094 0.6983373  0.73509189]
 [0.73509189 0.          0.65964159 0.6927506 ]
 [0.69811383 0.          0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.77378094 0.81450625 0.          0.73509189]
 [0.          0.          0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.81450625 0.          0.857375   0.77378094]
 [0.81450625 0.9025     0.9025     0.          ]
 [0.857375   0.95       0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.          0.          0.          ]
 [0.          0.9025     0.95       0.857375   ]
 [0.9025     0.95       1.          0.9025     ]
 [0.          0.          0.          0.          ]]
```

성공적인 학습의 경우



```
(Left)
SFFF
FHFH
FFFH
HFFG
(Left)
SFFF
FHFH
FFFH
HFFG
(Left)
SFFF
FHFH
FFFH
HFFG
(Left)
SFFF
FHFH
FFFH
HFFG
(Left)
SFFF
FHFH
FFFH
HFFG
```

실패한 학습의 경우: 난수가
사용되므로 아래 쪽까지 탐사
가 안되면 Q 테이블이 올바르
게 만들어지지 않는다.



테이블을 출력해보자.

```
>>> print(q_table)
```

```
[ [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.]  
  [0. 0. 0. 0.] ]
```

실패한 학습의 경우

epsilon 설정

- 처음에 탐사를 많이 하도록 입실론을 0.9 정도 설정하면 더 좋은 결과

```
discount_factor = 0.95
```

```
epsilon = 0.9
```

```
epsilon_decay_factor = 0.999
```

```
learning_rate = 0.8
```

```
num_episodes = 3000
```




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 추가 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.

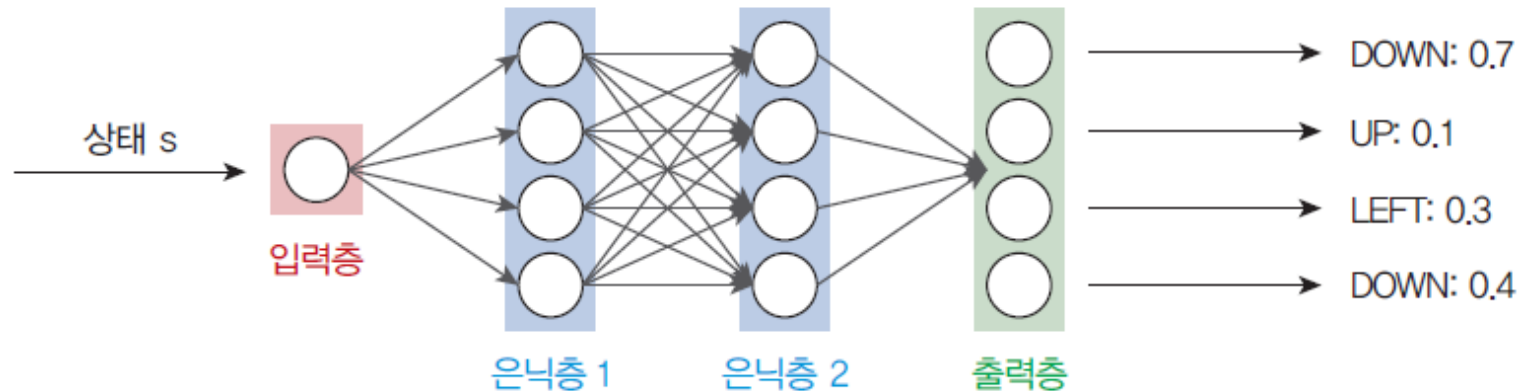
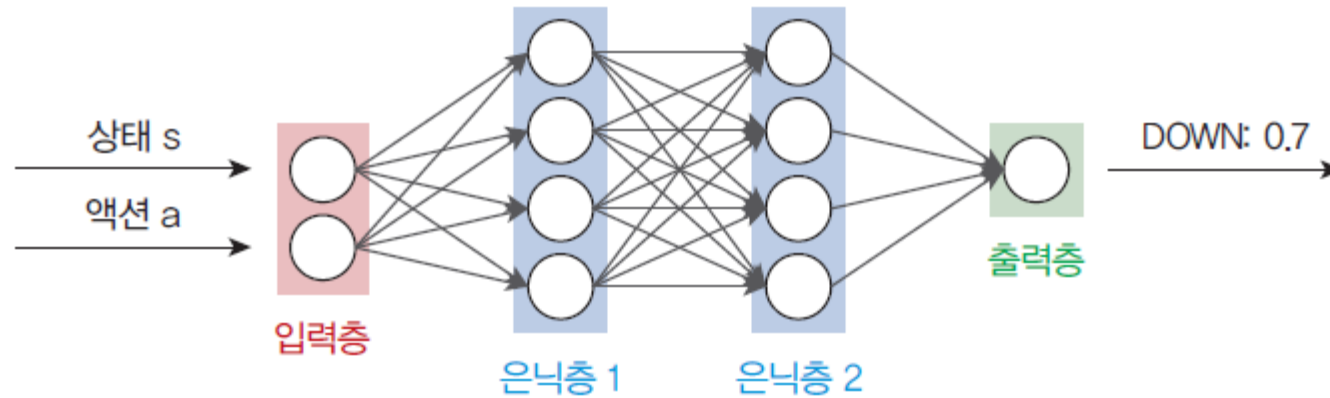




- 예를 들어서 100×100 크기의 화면을 가지고 있는 비디오 게임의 경우, 한 픽셀이 8바이트라고 하면 상태의 수는 얼마나 될까?
- 하나의 픽셀이 가질 수 있는 상태의 값은 256개이고 이러한 픽셀이 100×100 개나 있으므로 무려 $256^{100 \times 100}$ 이나 된다.
- 이렇게 탐색 공간이 무척 큰 경우가 바로 심층 신경망이 가장 필요한 경우이다.

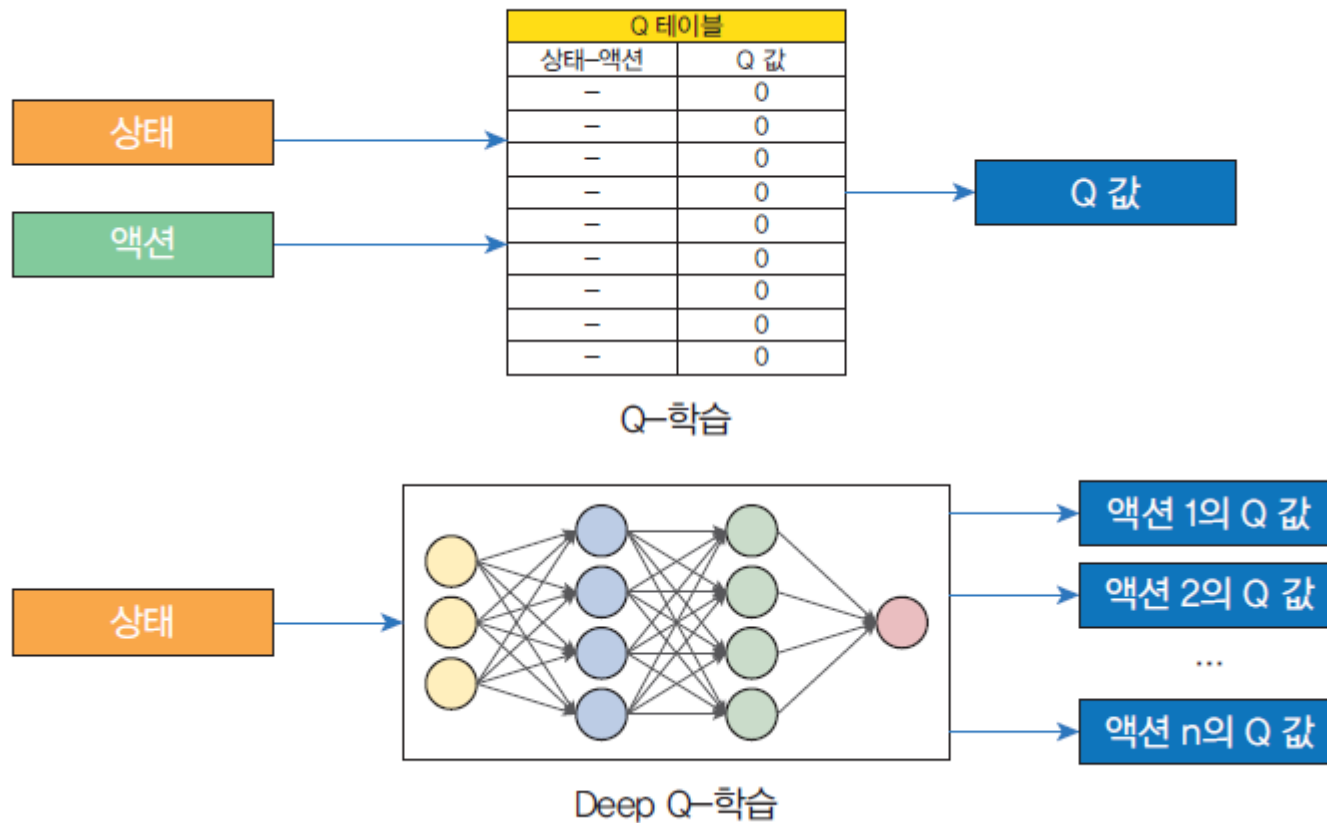


DQN(Deep Q Network)





Q-학습 vs Deep Q-학습



- 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.

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



$Q(s, a)$ 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.

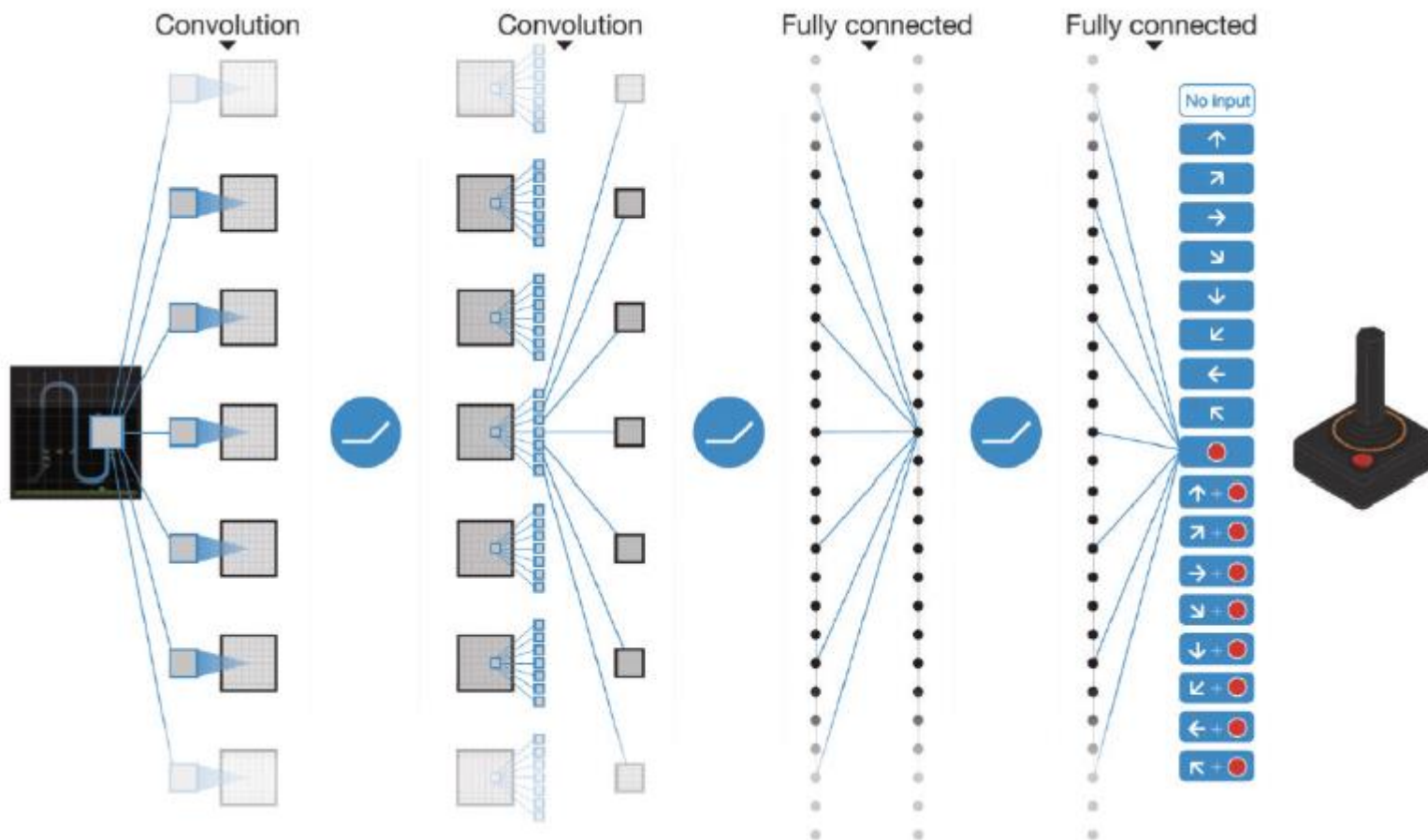
$$y_t = r_t + \gamma \underset{a'}{\operatorname{argmax}} Q(s_{t+1}, a' | \theta)$$

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

하나의 액션이 수행되고 보상과 다음
기 때문에 보다 정확한 Q값을 얻을 수
이 타겟(정확한 Q값)이 된다.
타겟과 현재 Q값의 차이를 이용하여

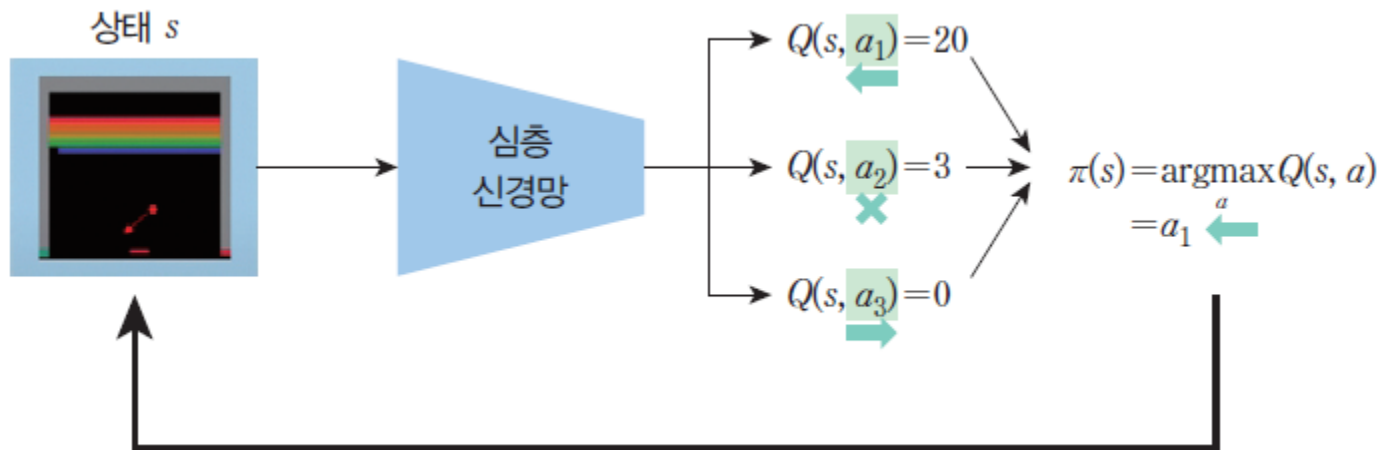


실제 적용 예: 비디오 게임





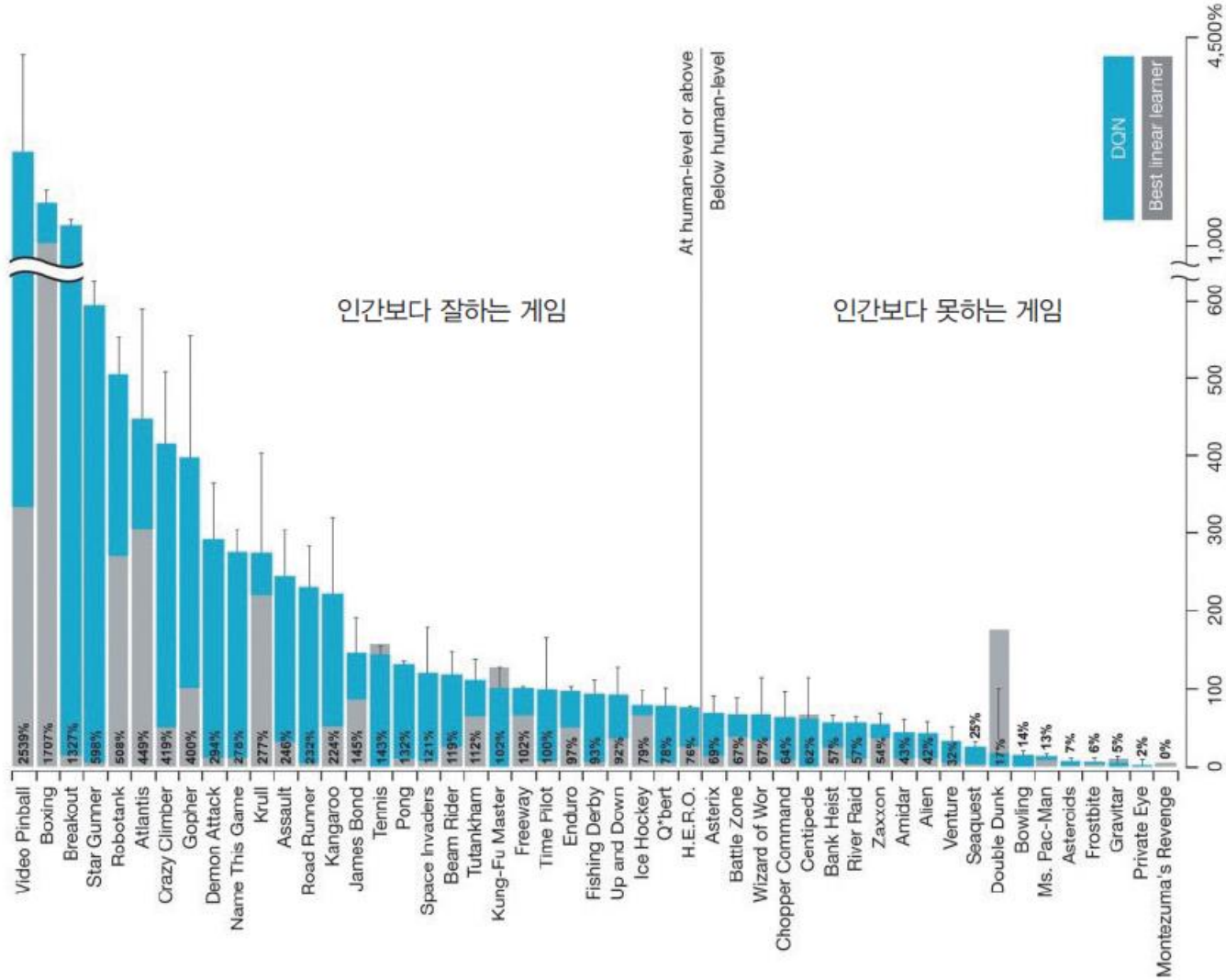
신경망의 학습



액션을 환경으로 보내고 다음 상태를 받는다.

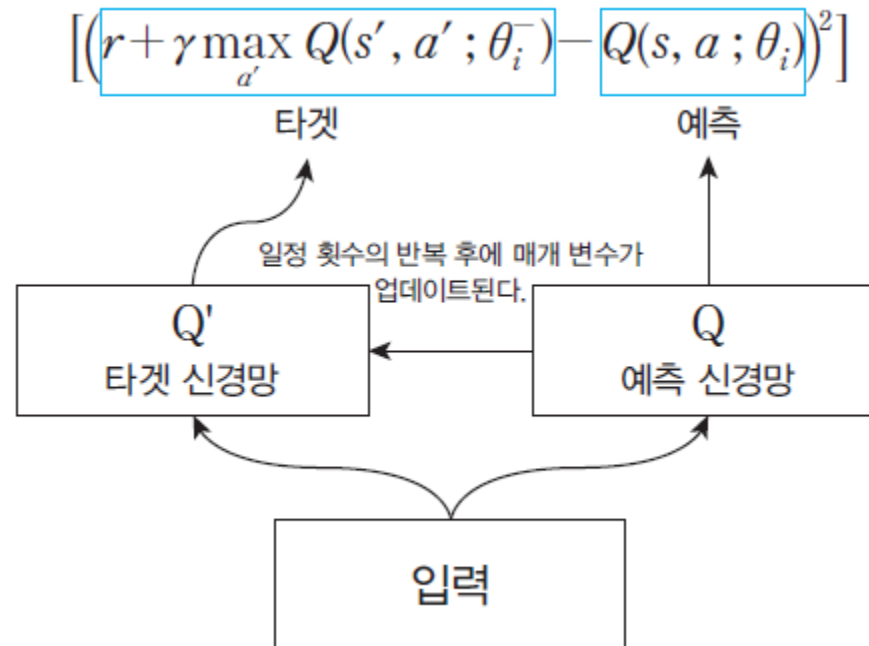


강화 학습을 이용한 게임의 성능





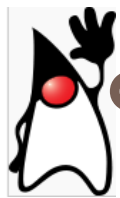
- Deep Q-학습에서는 약간의 문제가 있다. 우리는 목표 **Q** 값을 사실 정확히 알지 못한다. 그저 현재의 **Q** 값을 이용하여 추정할 뿐이다.
- 따라서 위의 알고리즘에서 볼 수 있듯이 반복할 때마다 목표가 변경된다.
- 이 문제를 해결하기 위하여 **2**개의 신경망을 사용하기도 한다.



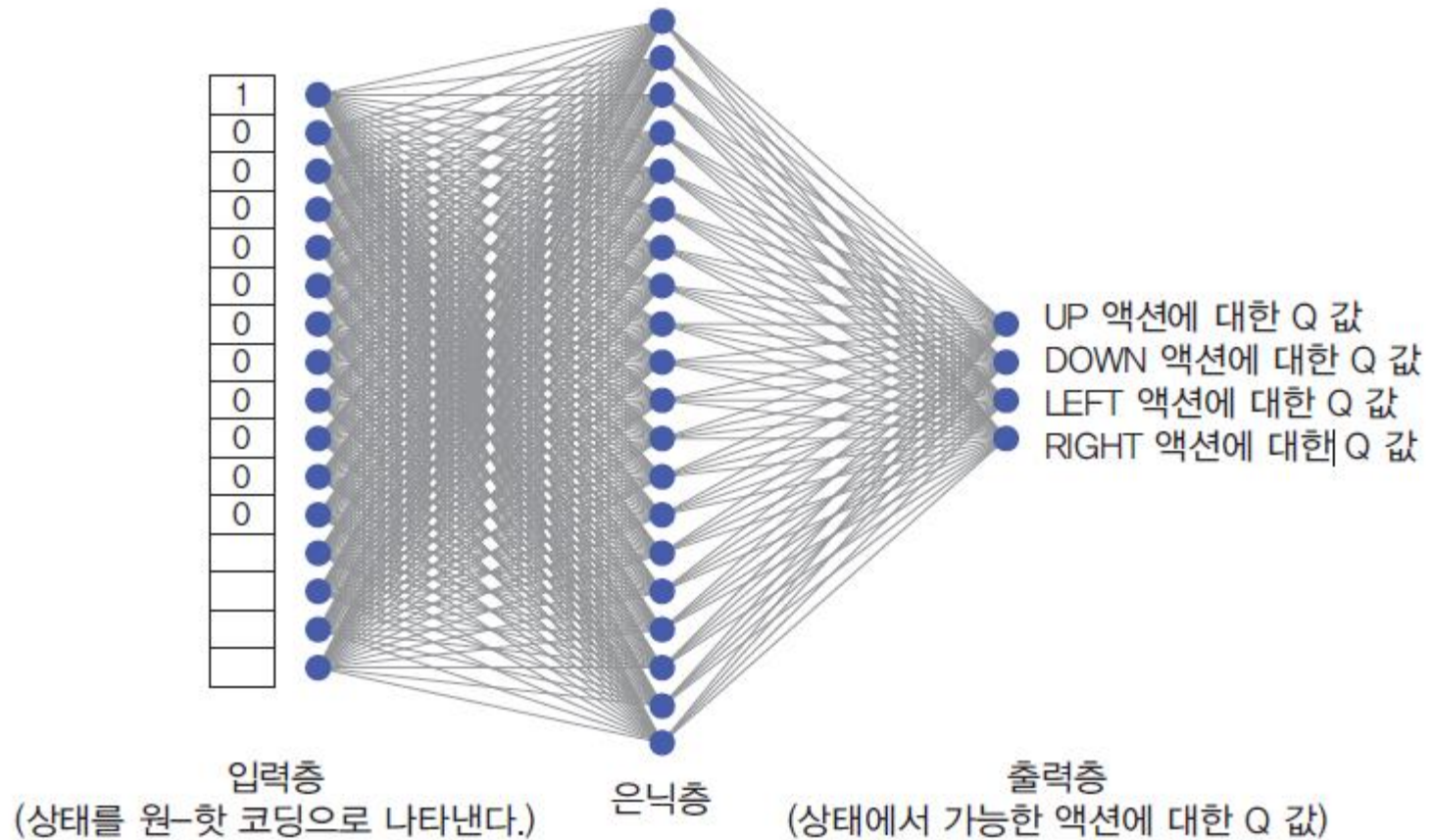


심층 Q-학습의 단점

- 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





예제: 얼음 호수 게임에서 심층 Q-학습의 구현

```
import gym
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer
from tensorflow.keras.layers import Dense

env = gym.make('FrozenLake-v0', is_slippery=False)

discount_factor = 0.95
epsilon = 0.5
epsilon_decay_factor = 0.999
num_episodes=500
```



예제: 어둠 호수 게임에서 심층 Q-학습의 구현

```
model = Sequential()  
model.add(InputLayer(batch_input_shape=(1, env.observation_space.n)))  
model.add(Dense(20, activation='relu'))  
model.add(Dense(env.action_space.n, activation='linear'))  
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

```
def one_hot(state):  
    state_m=np.zeros((1, env.observation_space.n))  
    state_m[0][state]=1  
    return state_m
```



예제: 어둠 호수 게임에서 심층 Q-학습의 구현

```
for i in range(num_episodes):           # 에피소드만큼 반복
    state = env.reset()                  # 환경 초기화
    epsilon *= epsilon_decay_factor      # 입실론을 점점 작게 만든다.
    done = False                         # 게임 종료 여부

    while not done:                     # 게임이 종료되지 않았으면
        if np.random.random() < epsilon: # 입실론보다 난수가 작으면
            action = env.action_space.sample() # 액션을 랜덤하게 선택
        else:
            action = np.argmax(model.predict(one_hot(state))) # 가장 큰 Q 값 액션

        # 게임을 한 단계 진행한다.
        new_state, reward, done, _ = env.step(action) # 게임 단계 진행
```




예제: 얼음 호수 게임에서 심층 Q-학습의 구현

① 목표값을 계산한다.

```
target = reward + discount_factor * np.max(model.predict(one_hot(new_state)))
```

② 현재 상태를 계산한다.

```
target_vector = model.predict(one_hot(state))[0]
```

```
target_vector[action] = target
```

③ 학습을 수행한다.

```
model.fit( one_hot(state), target_vector.reshape(-1, env.action_space.n),  
epochs=1, verbose=0)
```

상태를 다음 상태로 바꾼다.

```
state = new_state
```

```
print(i)
```

④ 마지막 상태만 화면에 표시한다.

```
if i==(num_episodes-1):
```

```
    env.render()
```



- 간단한 게임이지만 학습 시간이 아주 많이 걸린다.

```
(Right)
SFFF
FHFH
FFFH
HFFG
499
(Down)
SFFF
FHFH
FFFH
HFFG
```