과제 12/04

2315028 김성현

▼ 강화학습

: 지도학습과 달리 데이터가아닌 에이전트가 환경과 상호작용을 통해 누적된 보상이 최대가 되도록 특정 상태에서 취할 수 있는 적합한 행동을 선택하는 정책을 찾아가는 방법

▼ 마르코프 결정과정(MDP, markov decisioin process)

환경과 에이전트의 상호작용을 순차적으로 정의하는 수학적인 틀

MDP : 완전히 관측가능한 확률적환경에서(<>결정론적) 마르코프성질(미래상태는 과거상태가아닌 현재상태에만 의존함)을 만족하는 상태전이모형과 누적보상(보상이 쌓임)을 사용하는 순차적의사결정문제에대한 확률모델

- 마르코프 성질 : 과거의 복잡한 이력을 고려하지않아도 현재상태와 행동만으로 미래예측 가능 → 현실문제를 다루는데 매우 유용

  과거를 버리는 것이 아닌 현재는 과거의 총합이기에.. 또 고려하지않아도 된다는 의미

- 마르코프 구성요소

  상태(S) : 특정시점 별 환경의 상태집합

  행동(A) : 에이전트가 상태에서 선택할 수 있는 행동

  전이확률(P) : 현재상태와 행동에따라 다음상태로 전이될 확률

  보상(R) : 행동결과에 따라 에이전트가 받는 보상

  MDP = (S, A, P, R)

- 벨먼 방정식
  - MDP의 목표 : 에이전트가 각 상태마다 최적의 행동을 선택하도록만드는 규칙인 최적의 정책을 찾아 최대의 누적보상을 얻는 것! → 벨먼 방정식이 최적 정책을 찾는데 핵심적인 역할을함

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

V : 가치
s : 상태
a : 행동

현재 상태의 즉각적 보상

미래 상태의 기대 가치의 합

특정 상태의 가치와 행동, 보상, 다음 상태의 가치를 연결

- 벨먼 방정식 : 현재상태의 가치가 현재행동에서 얻는 즉각적인 보상과 미래상태의 기대가치의 합으로 정의됨. 재귀적인 특성을 지님(현재는 과거가되고, 미래는 현재가 되기에..)

  V(s) : 상태 s에서 시작해 최적정책을 따를때 얻을 수 있는 기대되는 최대보상의 기대치

  보상 - 즉각적 피드백으로 미래에대한 고려는 없음

  가치 - 장기적인 보상에대한 기대가치. 미래의 보상까지 고려해 계산

▼ Q-학습 (Q-learning)

: 에이전트가 환경과 상호작용하면서 최적의 행동을 학습하는 모델없는 강화학습 알고리즘. 주어진 환경에서 누적보상을 최대화할 수 있는 행동을 학습하며 Q-값을 갱신하며 학습을 진행함

MDP상황에서 보통 전이확률, 보상함수를 알아야하는데 Q-학습에서는 몰라도 됨

- Q-테이블 (보상테이블)

  행 : 가능한 상태 / 열 : 가능한 행동 / 각 셀 : 특정상태에서의 행동에대한 보상을 계산한 Q값이 있음

상태(state)	LEFT (←)	DOWN(↓)	RIGHT(→)	UP(↑)
0	Q(0, ←)	Q(0, ↓)	Q(0, →)	Q(0, ↑)
1	Q(1, ←)	Q(1, ↓)	Q(1, →)	Q(1, ↑)
2	Q(2, ←)	Q(2, ↓)	Q(2, →)	Q(2, ↑)
...				
14	Q(14, ←)	Q(14, ↓)	Q(14, →)	Q(14, ↑)
15	Q(15, ←)	Q(15, ↓)	Q(15, →)	Q(15, ↑)

  각 셀에 특정상태의 행동에대한 보상을 계산하는 Q-값이 들어있음

  Q-테이블을 통해 에이전트에게 가장 유리한 행동이 무엇인지 알려줌

- 보상

  Q값 - 보상에서 출발하기에 에이전트가 목표상태에 도달해 **보상**이 발생해야 테이블을 채워나갈 수 있음. 그전에는 모든값이 0으로 초기화

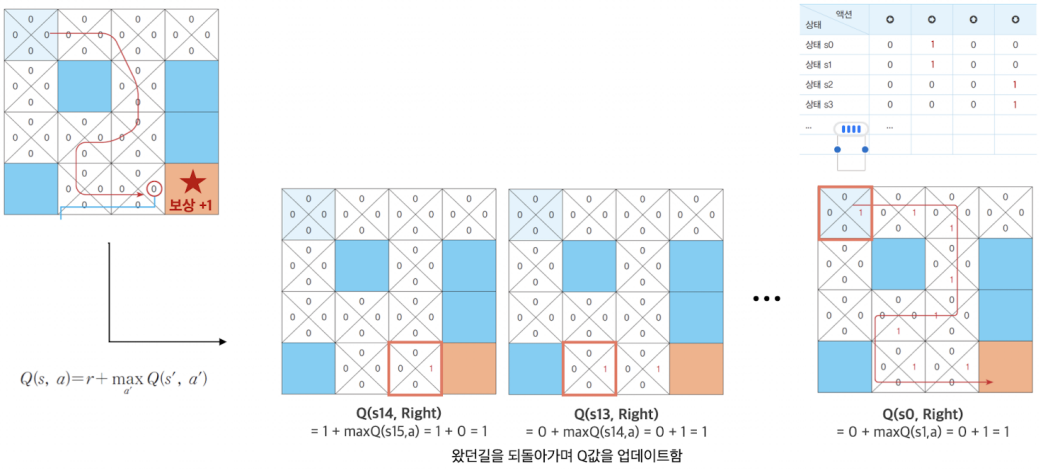
Q함수 - 에이전트의 현재상태와 에이전트가 실행하는 행동을 입력받아 총 보상값을 반환함

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

현재의 보상에다가 다음 상태에서의 최댓값을 합침

Q값 : 어떤 상태에서 어떤행동을 한 경우, 받을 수 있는 모든 보상을 더한 값. 다음상태의 Q값과 보상의 합으로 구성됨

- 업데이트



Q(s14, right) = 1 + max(Q(s15, a0))

▼ 실습환경

- 시뮬레이터 : 환경, 에이전트의 행동, 각 행동 후에 받는 보상을 모델링해주어야 함. 시뮬레이터를 사용해 시뮬레이션 된 환경에서 행동을하고, 그 결과를 측정해 연습을 통해 학습을 함

  ** 환경 초기화, 환경의 현재상태 가져오기, 환경에 행동적용, 행동에대한 보상을 계산, 목표달성 여부 확인

  ▼ 실습코드

  1번 방법) 무작위행동

```python
import gym

# 게임 환경 생성      게임명
env = gym.make("FrozenLake-v1", render_mode='human', is_slippery=False)
print("가능한 상태: ", env.observation_space.n)
print("가능한 행동: ", env.action_space.n)
env.reset()

n=30
env.render()
for i in range(n):
  # 에이전트가 랜덤한 행동 하나 선택
  action = env.action_space.sample()
  # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
  state, reward, done, truncated, info = env.step(action)
  # (행동, 상태, 보상) -> 다음(행동, 상태, 보상) -> ...
  print(f"({action}, {state}, {reward})", end="->")
  # 게임을 화면에 렌더
  env.render()
  # 게임이 끝나면(=done) 종료
  if done:
    break

env.close()
```

2번방법) 구현 (ft,. Q테이블)

```python
import gym
import numpy as np

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

# Q-테이블 초기화
states = env.observation_space.n
actions = env.action_space.n
q_table = np.zeros((states, actions))

n = 500
for i in range(n):
    env.reset()
    state = 0
    done = False

    while not done:
        # 최대 Q값이 0보다 큰지 작은지에 따라,
        if np.argmax(q_table[state]) > 0:
            # Q-테이블에서 가장 큰 값 가지는 행동을 선택
            action = np.argmax(q_table[state])
        else:
            # 무작위로 행동 선택
            action = env.action_space.sample()
        # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
        new_state, reward, done, truncated, info = env.step(action)
        # 새로 얻은 정보로 Q-테이블 갱신
        q_table[state, action] = reward + np.max(q_table[new_state])
        # 다음 상태로 상태 전이
        state = new_state

    print(f"{i}번째 에피소드 후 Q-table")
    print(q_table)
```

0으로 초기화. (행, 열)

Q값에서 가장 큰 값을 가지는 것을 선택 / 모두 0인 처음인 상태에는 무작위로 돌아다니도록

새로운 행동을 했을때의 정보들을 저장하고, Q테이블의 내용을 업데이트함(공식 그대로, 보상 + 새로운 상태에서의 가장 큰값)

→ 과연 학습일까? 하나의 행동밖에 못함

▼ e-greedy 알고리즘 : 탐험과 활용

강화학습 - 탐험 / 활용

** Q-테이블의 정보를 활용하지만 때때로 무작위로 탐험하는 방식을 추가해 문제를 해결

e의 확률로 행동(탐험)을 선택하고, (1-e)의 확률로 기존의 Q-값이 가장 큰 행동(활용)을 선택함

단, 학습단계 초기에는 e를 크게 설정해 다양한 행동을 시도하고, 학습이 진행될수록 e를 점차 줄여나가 학습된 최적행동에 집중하도록 함

▼ 할인계수 : 시간 차

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

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

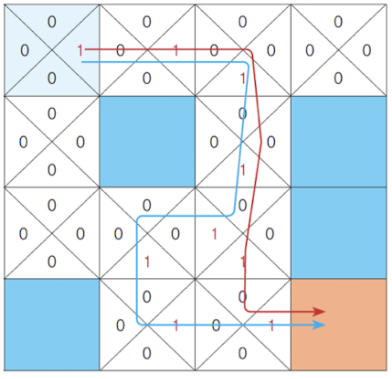
r(감마) : 할인계수(0~1 사이의 값을 지님)

r 크면 → 미래 보상을 중요하게 여김

r 작으면 → 현재 보상을 더 중요하게 여김

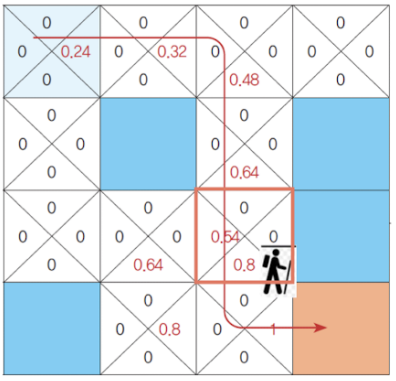
할인계수는 각 단계가 지날때마다 지수적으로 감소

할인 요인 미적용



- 미래 보상이 할인되지 않으면 각 상태 전이 시마다 같은 보상이 제공되어 어떤 경로가 더 좋은지 알 수가 없음

할인 요인 적용



- 할인율이 적용되어 어떤 경로(행동 선택)가 더 좋은지 명확하게 구분할 수 있음

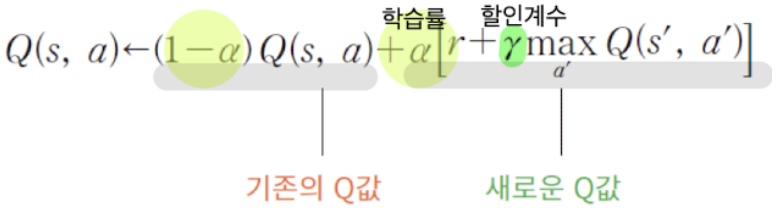
1과 0으로만 나타내면 무엇이 더 중요한지 차이를 나타내기에 부족함. 할인계수를 이용해 중요도를 더욱 세밀히 나타냄

미래 보상의 현재가치를 결정하는 매개변수로, 보상의 시간적 중요도를 조절함

시간적 중요도 매기는 방법 : 먼 미래의 보상은 예상하기 어렵기에 할인계수를 적용해 중요도를 낮추고, 현재와 가까운 즉각적인 보상에는 더 높은 가치를 부여함

▼ 학습률 : 확률적

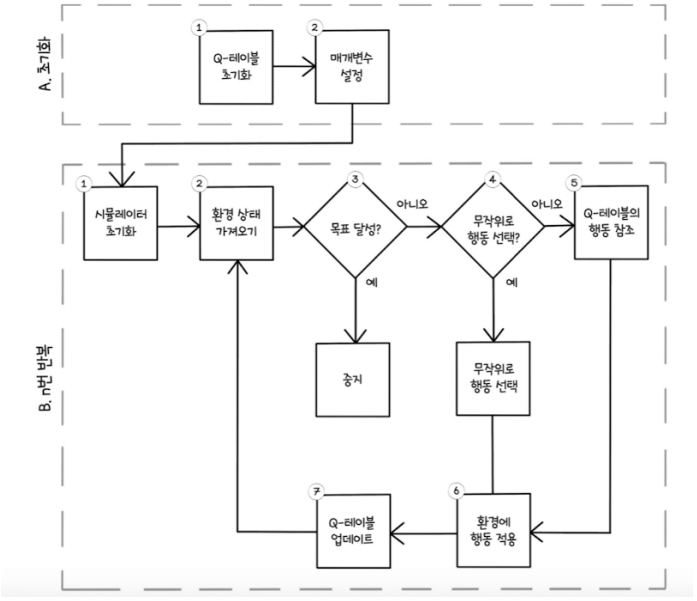
배경 : 결정적환경에서는 잘 작동하지만 확률적환경에서는 Q학습은 잘되지않음.



할인계수를 적용하고, 기존의 Q값도 고려해서 사용함. 또한 학습률 a를 도입해 대체하는 정도를 결정함.

▼ 최종 : Q-러닝

- 알고리즘 라이프사이클



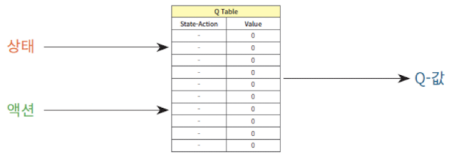
- 코드

```python
import gym
import numpy as np

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

# Q-테이블 초기화
states = env.observation_space.n
actions = env.action_space.n
q_table = np.zeros((states, actions))

# 초매개변수 설정
discount_factor = 0.9
epsilon = 0.9
epsilon_decay_factor = 0.999 입실론을 줄여나가도록
learning_rate = 0.8
num_episodes = 1000

```

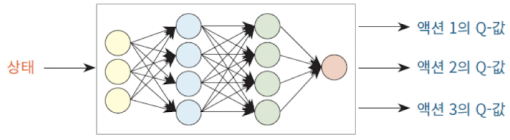
입실론, 현재 + 미래

```python
for i in range(num_episodes):
    env.reset()
    state = 0
    epsilon *= epsilon_decay_factor # 감쇠 적용
    done = False

    while not done:
        # 입실론 크기에 따라 탐사(exploration) 혹은 활용(exploitation)
        if np.random.random() < epsilon: 탐험
            # 무작위로 행동 선택
            action = env.action_space.sample()
        else:
            # Q-테이블에서 가장 큰 값 가지는 행동을 선택
            action = np.argmax(q_table[state]) 활용
        # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
        new_state, reward, done, truncated, info = env.step(action)
        # 새로 얻은 정보로 Q-테이블 갱신 (최종 공식 적용)
        q_table[state, action] += learning_rate * (reward + discount_factor *
                                                   np.max(q_table[new_state])
                                                   - q_table[state, action])

        # 다음 상태로 상태 전이
        state = new_state

    print(f"{i}번째 에피소드 후 Q-table")
    print(q_table)
```

▼ Deep Q-Network



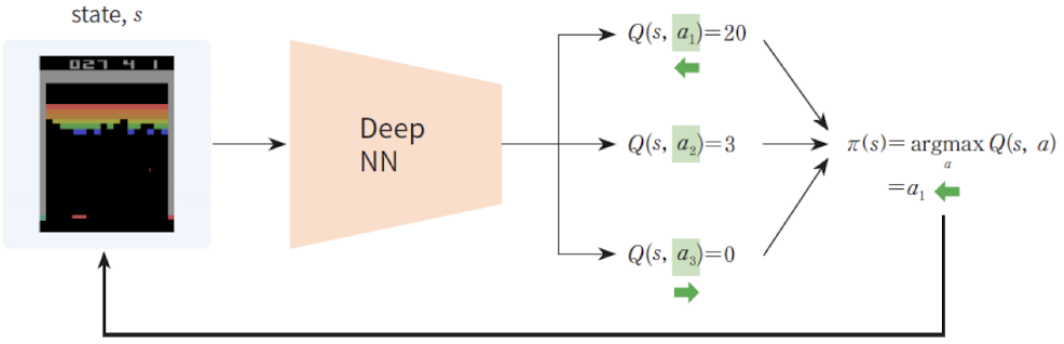
Q-학습



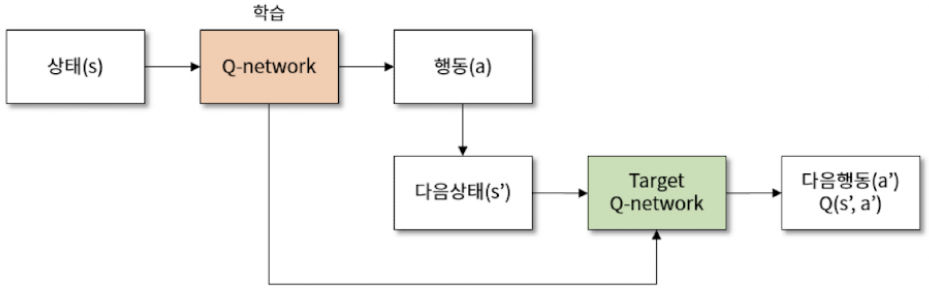
Deep Q-학습

Q-학습은 모든 상태-행동쌍을 저장해야하기에 테이블 크기가 너무 커져 비효율적임

→ 심층신경망을 이용해 Q값을 근사하는 방식으로 Q학습을 함

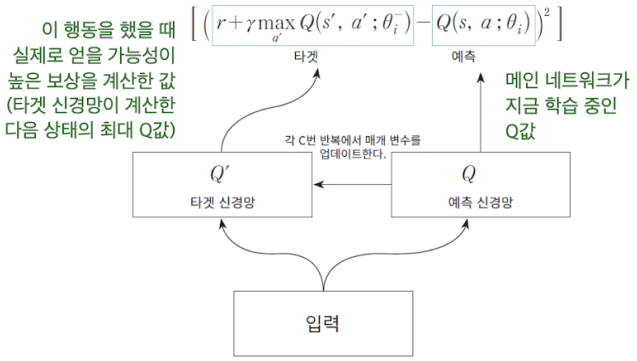


상태s가 입력으로 주어질때 가능한 모든 행동의 Q값을 반환하며, 그 중 가장 큰 값을 가지는 행동을 선택해 학습 시 Q값이 최대 누적보상을 예측하도록 학습



강화학습은 보상으로 학습해야함. Q값은 환경의 피드백에 따라 계속 변화하므로 학습이 불안정할 수 있음

→ DQN은 입력에대한 예측을 수행하는 예측신경망과 별개로 안정적인 목표값을 제공하는 타겟 신경망을 도입함



- 예측신경망 - 학습과정에서 가중치가 계속 업데이트되어 변함
- 타겟신경망 - 일정주기마다 메인네트워크의 가중치를 동기화해 고정된 가중치를 사용해서 안정적인 목표값을 제공

→ 목표값과 예측값을 비교해 손실을 계산해 손실을 최소화하는 방향으로 학습을 진행

▼ 심층신경망

▼ mnist 데이터

1. 데이터로드 및 정규화

```python
# 1. 데이터 로드 및 정규화
mnist = datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 데이터 정규화: 각 픽셀의 값은 기본적으로 0에서 255 사이의 정수이지만, 안정적인 학습을 위해 0과 1사이의 실수로 정규화
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# 데이터 형태 확인
print(f"Training data shape: {x_train.shape}, Training labels shape: {y_train.shape}")
```

✓ 0.1s Python

```
Training data shape: (60000, 28, 28), Training labels shape: (60000,)
```

2. 모델 정의

```python
# 2. 모델 정의
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # 입력: 28x28 이미지를 1D 벡터로 변환
    layers.Dense(512, activation='relu'),  # 은닉층: 512개의 뉴런
    layers.Dropout(0.2),                   # 드롭아웃: 과적합 방지. 512개중 20%를 꺼줌
    layers.Dense(10, activation='softmax') # 출력층: 10개 클래스의 확률
])

# 모델 요약 출력
model.summary()
```

✓ 0.0s

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 512)               401920

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 407050 (1.55 MB)
Trainable params: 407050 (1.55 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

3. 모델 컴파일

```python
# 3. 모델 컴파일
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

✓ 0.0s

4. 모델 학습

```python
# 4. 모델 학습
history = model.fit(x_train, y_train, epochs=10, batch_size=64)
```

✓ 14.3s

```
Epoch 1/10
938/938 [==============================] - 2s 2ms/step - loss: 0.0229 - accuracy: 0.9930
Epoch 2/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0205 - accuracy: 0.9931
Epoch 3/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0194 - accuracy: 0.9939
Epoch 4/10
938/938 [==============================] - 1s 2ms/step - loss: 0.0185 - accuracy: 0.9942
Epoch 5/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0166 - accuracy: 0.9948
Epoch 6/10
938/938 [==============================] - 1s 2ms/step - loss: 0.0132 - accuracy: 0.9959
Epoch 7/10
938/938 [==============================] - 1s 2ms/step - loss: 0.0139 - accuracy: 0.9952
Epoch 8/10
938/938 [==============================] - 1s 2ms/step - loss: 0.0120 - accuracy: 0.9959
Epoch 9/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0137 - accuracy: 0.9955
Epoch 10/10
938/938 [==============================] - 2s 2ms/step - loss: 0.0123 - accuracy: 0.9957
```

5. 모델 평가

```python
# 5. 모델 평가
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}, Test Loss: {test_loss:.4f}")
```

✓ 0.2s

```
313/313 [==============================] - 0s 545us/step - loss: 0.0765 - accuracy: 0.9822
Test Accuracy: 0.9822, Test Loss: 0.0765
```

▼ Q

- 무작위 행동

```python
import gym

# 게임 환경 생성
env = gym.make("FrozenLake-v1", render_mode='human', is_slippery=False)
print("가능한 상태: ", env.observation_space.n)
print("가능한 행동: ", env.action_space.n)
env.reset()

n=30
env.render()
for i in range(n):
  # 무작위로 행동 하나 선택
  action = env.action_space.sample()
  # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
  state, reward, done, truncated, info = env.step(action)
  # (행동, 상태, 보상) -> 다음(행동, 상태, 보상) -> ...
  print(f"({action}, {state}, {reward})", end="->")
  # 게임을 화면에 렌더
  env.render()
  # 게임이 끝나면(=done) 종료
  if done:
    break

env.close()
```

✓ 13.1s Python

```
가능한 상태:  16
가능한 행동:  4
2024-12-04 21:22:32.439 python[18860:418708] +[IMKClient subclass]: chose IMKClient_Modern
2024-12-04 21:22:32.439 python[18860:418708] +[IMKInputSession subclass]: chose IMKInputSession_M
(2, 1, 0.0)->(3, 1, 0.0)->(2, 2, 0.0)->(3, 2, 0.0)->(1, 6, 0.0)->(3, 2, 0.0)->(2, 3, 0.0)->(1, 7,
```

- Q-학습

```python
import gym
import numpy as np

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

# Q-테이블 초기화
states = env.observation_space.n
actions = env.action_space.n
q_table = np.zeros((states, actions))

n = 500
for i in range(n):
    env.reset()
    state = 0
    done = False

    while not done:
        # 최대 Q값이 0보다 큰지 작은지에 따라,
        if np.argmax(q_table[state]) > 0:
            # Q-테이블에서 가장 큰 값 가지는 행동을 선택
            action = np.argmax(q_table[state])
        else:
            # 무작위로 행동 선택
            action = env.action_space.sample()
        # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
        new_state, reward, done, truncated, info = env.step(action)
        # 새로 얻은 정보로 Q-테이블 갱신
        q_table[state, action] =  reward + np.max(q_table[new_state])
        # 다음 상태로 상태 전이
        state = new_state

    print(f"{i}번째 에피소드 후 Q-table")
    print(q_table)
```

```
0번째 에피소드 후 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.]
 [0. 0. 0. 0.]]
1번째 에피소드 후 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. 1. 0.]
 [0. 0. 0. 0.]]
```

- Q-러닝

```python
import gym
import numpy as np

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

# Q-테이블 초기화
states = env.observation_space.n
actions = env.action_space.n
q_table = np.zeros((states, actions))

# 초매개변수 설정
discount_factor = 0.9
epsilon = 0.9
epsilon_decay_factor = 0.999
learning_rate = 0.8
num_episodes = 1000

for i in range(num_episodes):
    env.reset()
    state = 0
    epsilon *= epsilon_decay_factor # 감쇠 적용
    done = False

    while not done:
        # 입실론 크기에 따라 탐사(exploration) 혹은 활용(exploitation)
        if np.random.random() < epsilon:
            # 무작위로 행동 선택
            action = env.action_space.sample()
        else:
            # Q-테이블에서 가장 큰 값 가지는 행동을 선택
            action = np.argmax(q_table[state])
        # 지정 행동 실행 후, (새로운 상태, 보상, 종료 여부 등 정보)를 반환
        new_state, reward, done, truncated, info = env.step(action)
        # 새로 얻은 정보로 Q-테이블 갱신 (최종 공식 적용)
        q_table[state, action] += learning_rate * (reward + discount_factor * np.max(q_table[new_state]) - q_table[state, action])
        # 다음 상태로 상태 전이
        state = new_state

    print(f"{i}번째 에피소드 후 Q-table")
    print(q_table)
```

```
0번째 에피소드 후 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.]]
1번째 에피소드 후 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.          0.81        0.9         0.729     ]
 [0.81        0.9         1.          0.81      ]
 [0.          0.          0.          0.        ]]
```