강화학습

https://youtu.be/0LJmCZtd5jo

Reinforcement Learning

Markov Decision Process

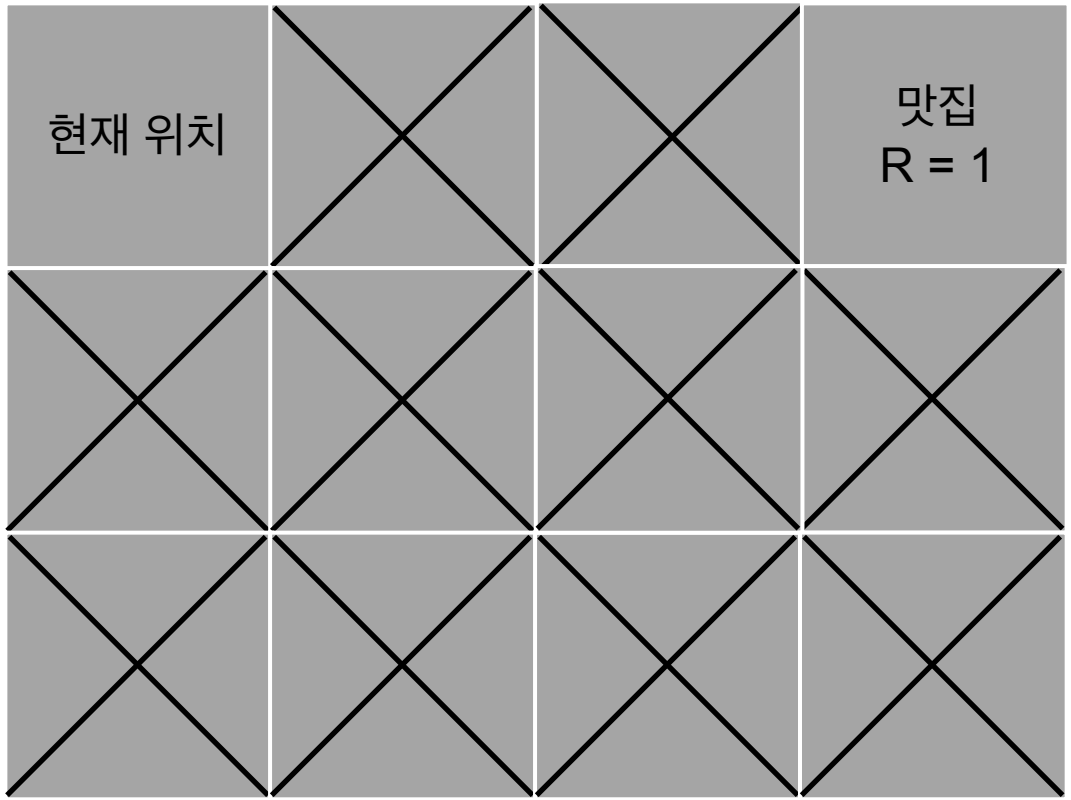
Optimal Policy

Bellman Equation

DQN

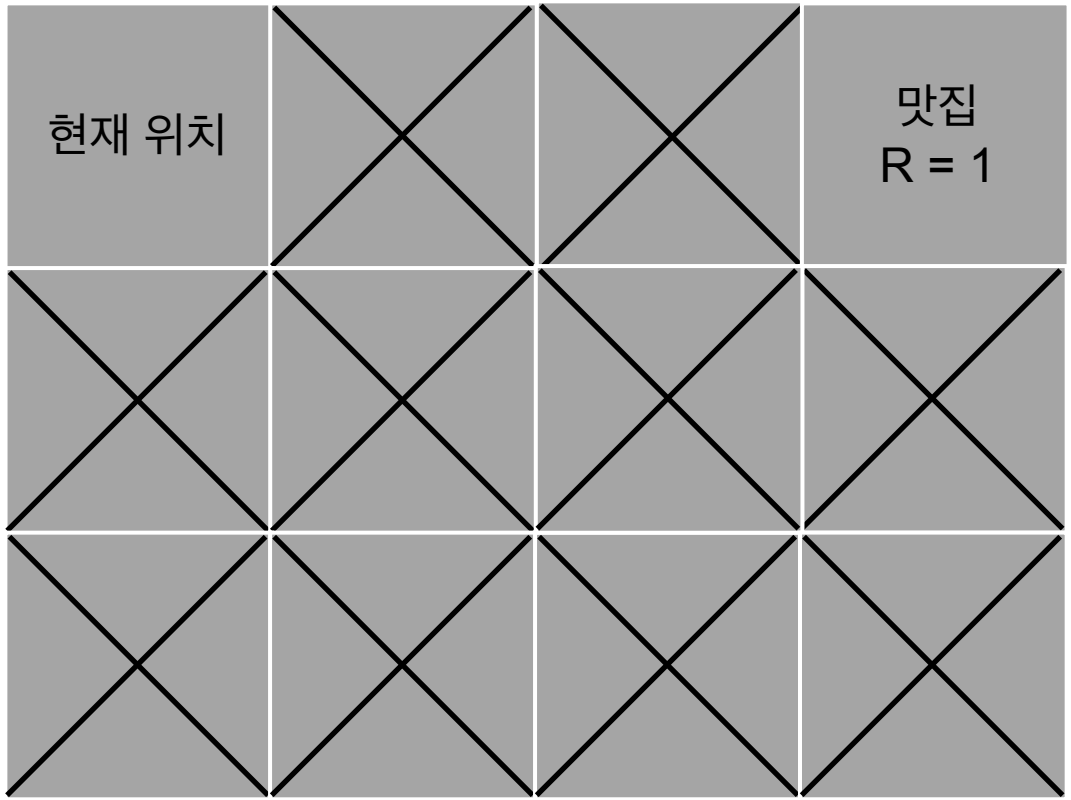
Reinforcement Learning

- 강화학습이란
  - 현재 상태(S, State)에서 행동(A, Action)에 대한 보상(R, Reward)를 최대로 만드는 것
  - 칸 안에는 Q-value
  - episode에 따라 reward가 갱신됨
  - 다음 state의 가장 큰 값으로 갱신



Reinforcement Learning

- Greedy Algorithm (Exploitation)
  - 탐욕적인 알고리즘

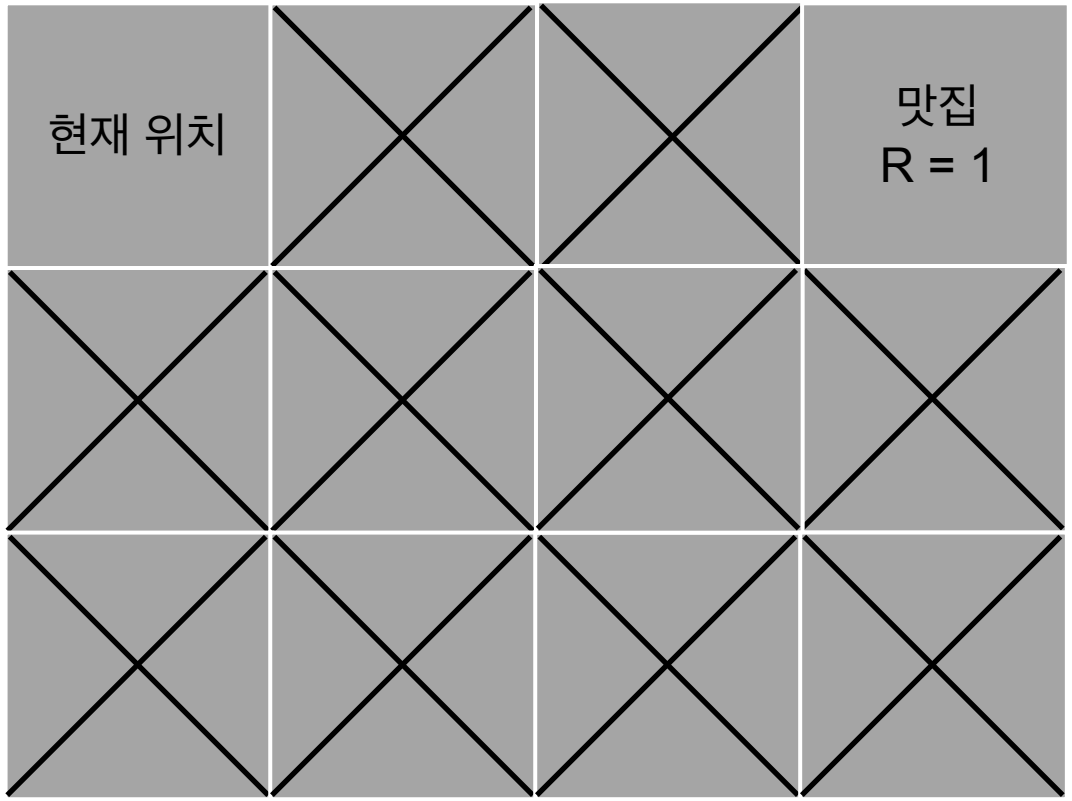


Reinforcement Learning

- ε-Greedy algorithm (Exploration)
  - 새로운 경로, 맛집을 찾고 싶을 때

- Decaying ε-Greedy Algorithm
  - ε을 점점 줄여나가면서 학습

- Discount Factor (γ)
  - γ: 0~1 사이의 값

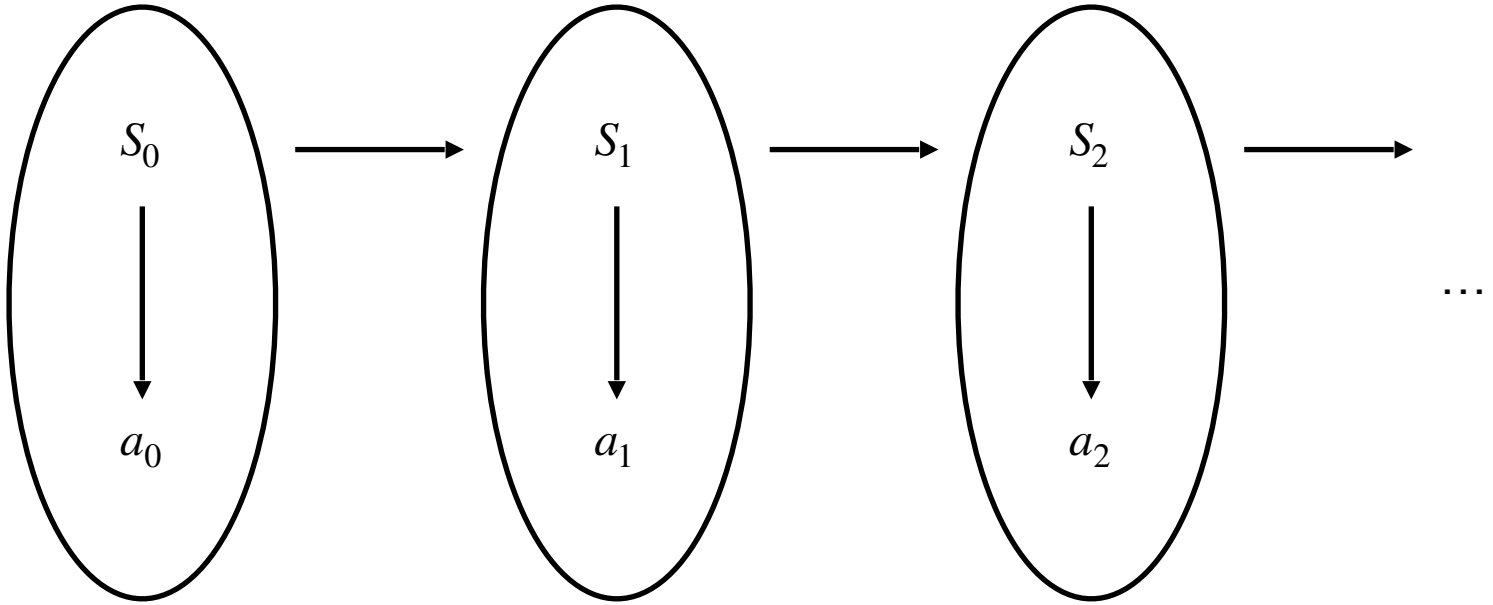


Reinforcement Learning

- Q - update (soft copy)
  - $Q(S_t, a_t) \leftarrow (1-\alpha)Q(S_t, a_t) + \alpha(R_{t+1} + \gamma maxQ(S_{t+1}, a_{t+1}))$
  - α: 0~1 사이의 값 (기존 값과 새로운 값의 balancing)

Markov Decision Process

- MDP(Markov Decision Process)란
  - Decision, 즉 action들을 취하는 과정을 나타낸 것
  - $p(a_0 \mid S_0)$



Markov Decision Process

1. $p(a_1 \mid S_0, a_0, S_1) \rightarrow p(a_1 \mid S_1)$

: policy

2. $p(S_2 \mid S_0, a_0, S_1, a_1) \rightarrow p(S_2 \mid S_1, a_1)$

: transition probability

Optimal Policy

- 강화학습의 목표 = 보상(Reward)를 최대화하는 것.
- Expected Reward
- Goal = Maximize Expected Reward
- Reward를 maximize한다는 것은 현재 상태에서 reward을 maximize하는 action을 찾는 것, 즉 optimal policy를 찾는 것이 목표다

- $G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$

Optimal Policy

- State Value function: 지금 state로부터 시작해서 기대되는 Return, 현재 State에 대한 평가
- $V(S_t) \overset{\text{def}}{=} \int_{a_t}^{a_\infty} G_t p(a_t, S_{t+1}, a_{t+1}, \ldots \mid S_t) da_t : a_\infty$

- Action Value function: 지금 state에서의 지금 행동으로부터 기대되는 Return, 현재 action에 대한 평가 (Q - value)
- $Q(S_t, a_t) \overset{\text{def}}{=} \int_{S_{t+1}}^{a_\infty} G_t p(S_{t+1}, a_{t+1}, S_{t+2}, \ldots \mid S_t, a_t) da_{t+1} : a_\infty$

- State Value Function을 maximize 하는 것

$$E[f(x)] = \int f(x)p(x)dx$$

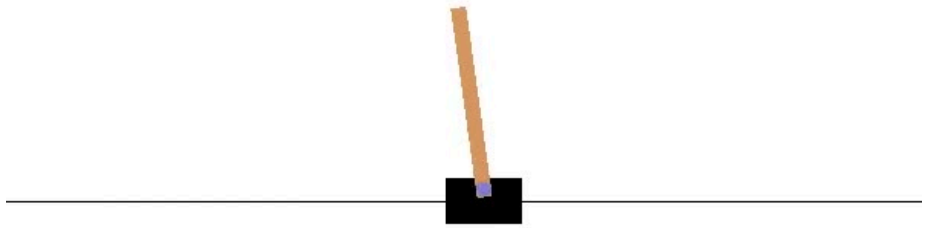
Bellman Equation

- $V(S_t) \stackrel{\text{def}}{=} \int_{a_t}^{a_\infty} G_t p(a_t, S_{t+1}, a_{t+1}, \ldots | S_t) da_t : a_\infty$
- $= \int_{a_t} \int_{S_{t+1}}^{a_\infty} G_t p(S_{t+1}, a_{t+1}, \ldots | S_t, a_t) dS_{t+1} : a_\infty p(a_t | S_t) da_t$
- $= \int_{a_t} Q(S_t, a_t) p(a_t | S_t) da_t$

- $p(x, y) = p(x | y) p(y)$
- $p(x, y | z) = p(x | y, z) p(y | z)$

DQN

- Cart-Pole



DQN

```python
import gym # 카트폴같은 여러 게임 환경 제공 패키지
# pip install gym 으로 설치 가능#
import random # 에이전트가 무작위로 행동할 확률을 구하기 위함.
import math # 에이전트가 무작위로 행동할 확률을 구하기 위함.
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from collections import deque
#Deque : 먼저 들어온 데이터가 먼저 나가는 FIFO 자료구조의 일종으로 double-ended queue의 약자로, 일반적인 큐오
import matplotlib.pyplot as plt
# 필수 모듈 임포트하기
```

DQN

```
# 하이퍼파라미터 정의
EPISODES = 50     # 애피소드(총 플레이할 게임 수) 반복횟수
EPS_START = 0.9   # 학습 시작시 에이전트가 무작위로 행동할 확률
# ex) 0.5면 50% 절반의 확률로 무작위 행동, 나머지 절반은 학습된 방향으로 행동
# random하게 EPisolon을 두는 이유는 Agent가 가능한 모든 행동을 경험하기 위함.
EPS_END = 0.05    # 학습 막바지에 에이전트가 무작위로 행동할 확률
#EPS_START에서 END까지 점진적으로 감소시켜줌.
# --> 초반에는 경험을 많이 쌓게 하고, 점차 학습하면서 똑똑해지니깐 학습한대로 진행하게끔
EPS_DECAY = 200   # 학습 진행시 에이전트가 무작위로 행동할 확률을 감소시키는 값
GAMMA = 0.8       # 할인계수 : 에이전트가 현재 reward를 미래 reward보다 얼마나 더 가치있게 여기는지에 대한 값
# 일종의 할인율
LR = 0.001        # 학습률
BATCH_SIZE = 64   # 배치 크기
```

DQN

```
class DQNAgent:
    def __init__(self):
        self.model = nn.Sequential(
            nn.Linear(4, 256),#신경망은 카트 위치, 카트 속도, 막대기 각도, 막대기 속도까지 4가지 정보를 입
            nn.ReLU(),
            nn.Linear(256, 2)#왼쪽으로 갈 때의 가치와 오른쪽으로 갈 때의 가치
        )
        self.optimizer = optim.Adam(self.model.parameters(), LR)
        self.steps_done = 0 #self.steps_done은 학습을 반복할 때마다 증가
        self.memory = deque(maxlen=10000)#memory에 deque사용.(큐가 가득 차면 제일 오래된것부터 삭제)

    def memorize(self, state, action, reward, next_state):#에피소드 저장 함수
        # self.memory = [(상태, 행동, 보상, 다음 상태)...]
        self.memory.append((state,
                            action,
                            torch.FloatTensor([reward]),
                            torch.FloatTensor([next_state])))

    def act(self, state):#행동 선택(담당)함수
        eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * self.steps_done /
        self.steps_done += 1
        #무작위 숫자와 엡실론을 비교

        # 엡실론 그리디(Epsilon-Greedy)
        # : 초기엔 엡실론을 높게=최대한 경험을 많이 하도록 / 엡실론을 낮게 낮춰가며 = 신경망이 선택하는 비율 상승
        if random.random() > eps_threshold:# 무작위 값 > 엡실론값 : 학습된 신경망이 옳다고 생각하는 쪽
            return self.model(state).data.max(1)[1].view(1, 1)
        else:# 무작위 값 < 엡실론값 : 무작위로 행동
            return torch.LongTensor([[random.randrange(2)]])
```

```
    def learn(self):#메모리에 쌓아둔 경험들을 재학습(replay)하며, 학습하는 함수
        if len(self.memory) < BATCH_SIZE:# 메모리에 저장된 에피소드가 batch 크기보다 작으면 그냥 학습을
            return
        #경험이 충분히 쌓일 때부터 학습 진행
        batch = random.sample(self.memory, BATCH_SIZE) # 메모리에서 무작위로 Batch 크기만큼 가져와
        states, actions, rewards, next_states = zip(*batch) #기존의 batch를 요소별 리스트로 분리

        #리스트를 Tensor형태로
        states = torch.cat(states)
        actions = torch.cat(actions)
        rewards = torch.cat(rewards)
        next_states = torch.cat(next_states)

        # 모델의 입력으로 states를 제공, 현 상태에서 했던 행동의 가치(Q값)을 current_q로 모음
        current_q = self.model(states).gather(1, actions)

        max_next_q = self.model(next_states).detach().max(1)[0]#에이전트가 보는 행동의 미래 가치(
        expected_q = rewards + (GAMMA * max_next_q)# rewards(보상)+미래가치

        # 행동은 expected_q를 따라가게끔, MSE_loss로 오차 계산, 역전파, 신경망 학습
        loss = F.mse_loss(current_q.squeeze(), expected_q)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

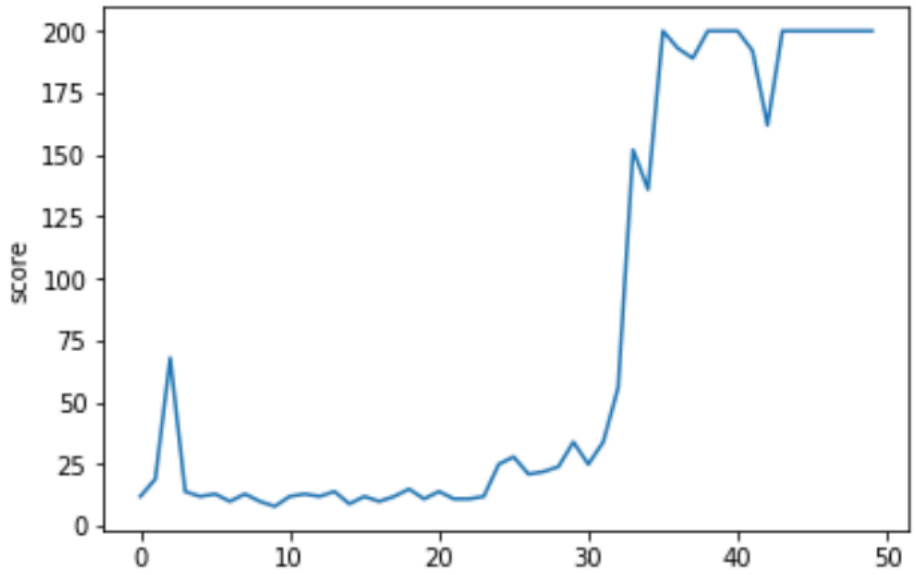
DQN

```python
for e in range(1, EPISODES+1):#50번의 플레이(EPISODE수 만큼)
    state = env.reset() # 매 시작마다 환경 초기화
    steps = 0
    while True: # 게임이 끝날때까지 무한루프
        env.render()
        state = torch.FloatTensor([state]) # 현 상태를 Tensor화

        # 에이전트의 act 함수의 입력으로 state 제공
        # Epsilon Greedy에 따라 행동 선택
        action = agent.act(state)

        # action : tensor, item 함수로 에이전트가 수행한 행동의 번호 추출
        # step함수의 입력에 제공 ==> 다음 상태, reward, 종료 여부(done, Boolean Value) 출력
        next_state, reward, done, _ = env.step(action.item())

        # 게임이 끝났을 경우 마이너스 보상주기
        if done:
            reward = -1

        agent.memorize(state, action, reward, next_state) # 경험(에피소드) 기억
        agent.learn()

        state = next_state
        steps += 1

        if done:
            print("에피소드:{0} 점수: {1}".format(e, steps))
            score_history.append(steps) #score history에 점수 저장
            break
```

DQN

```python
# 점수 시각화
plt.plot(score_history)
plt.ylabel('score')
plt.show()
```

```
에피소드:35 점수: 136
에피소드:36 점수: 200
에피소드:37 점수: 193
에피소드:38 점수: 189
에피소드:39 점수: 200
에피소드:40 점수: 200
에피소드:41 점수: 200
에피소드:42 점수: 192
에피소드:43 점수: 162
에피소드:44 점수: 200
에피소드:45 점수: 200
에피소드:46 점수: 200
에피소드:47 점수: 200
에피소드:48 점수: 200
에피소드:49 점수: 200
에피소드:50 점수: 200
```



Q & A