

# Reinforcement Learning Basic

## Week5. Deep Q-Network

박준영

Hanyang University  
Department of Computer Science

## 지난 시간

- 인공 신경망을 통한 근사적 해법
- 탐험(exploration)과 활용(exploitation)의 딜레마
- Deep Q-Network

1 Policy Gradient

2 REINFORCE

## Review: Value-based RL

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi} [G_t | S_t = s] \\q_{\pi}(s, a) &= \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a]\end{aligned}$$

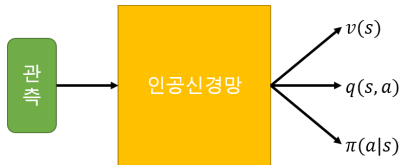
가치 함수를 안다면 정책은 다음과 같이 탐욕적으로 만들면 된다.

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

## Review: Policy

$$\pi(a|s) = P[A_t = a | S_t = s]$$

정책: 상태가 입력으로 들어오면 행동을 반환하는 함수  
→ 정책을 바로 만드는 방법은 없을까?



## Why the policy?

- RL의 궁극적인 목표는 정책을 구하는 것.
- Value가 중요하지 않은 문제도 존재한다.
- Action space가 연속적일 때도 사용할 수 있다.
- 확률적인 정책을 만들 수 있다.

## 정책의 표현

$$\pi(a|s) = P[A_t = a | S_t = s]$$

- 정책은 조건부 확률로 표현할 수 있다.  
→ Softmax 함수를 이용해 인공 신경망을 만든다.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

## Review: DQN

DQN에서  $Q(s, a)$ 에 대한 손실 함수는 다음과 같다.

$$\mathcal{L} = (R + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2$$

- 목표:  $R + \gamma \max_{a'} Q(s', a'; \theta^-)$ .
- 인공 신경망으로 만든 함수  $Q(s, a)$ 가 목표와 **오차가 작아지게** 하는 것이 목표.
- 이때, 경사 하강법으로 학습을 시켰다.



# Policy Gradient

정책을 인공 신경망으로 근사했기 때문에 정책을 다음과 같이 표현할 수 있다.

$$\pi_{\theta}(a|s)$$

- $\theta$ 는 정책 신경망의 가중치이다.
- 목표 함수는  $J(\theta)$ 로 표현할 수 있다.

# Policy Gradient

강화학습의 목표는 **누적 보상을 최대**로 하는 **최적 정책**을 찾는 것이다.

따라서 정책 기반 강화학습의 목표를 수식으로 표현하면 다음과 같다.

$$\text{Maximize } J(\theta)$$

그런데, 인공 신경망의 학습 목표는 오차를 **최소화** 하는 것이다.

따라서 목표를 다음과 같이 수정한다.

$$\text{Minimize } [-J(\theta)]$$

# Policy Gradient

경사 하강법에서는 목표 함수를 미분하여 정책 신경망을 업데이트 한다.  
목표 함수는  $J(\theta) = v_\pi(s_0)$ 이므로 목표 함수의 미분은 다음과 같다.

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta v_\pi(s_0) \\ &= \sum_s d_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s, a) \\ &= \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} q_\pi(s, a) \\ &= \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) q_\pi(s, a) \\ &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) q_\pi(s, a)]\end{aligned}$$

# Policy Gradient

최종적으로 Policy Gradient의 정책 신경망 갱신식은

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta) \approx \theta + \alpha [\nabla_{\theta} \log \pi_{\theta}(a|s) q_{\pi}(s, a)]$$

하지만, 가치 함수가 없기 때문에 위 수식을 사용할 수 없다.



# REINFORCE

큐함수는 반환값의 기댓값이다.

→ 큐함수를 반환값으로 대체할 수 있다.

REINFORCE 알고리즘의 정책 신경망 갱신식은 다음과 같다.

$$\theta \leftarrow \theta + \alpha [\nabla_{\theta} \log \pi_{\theta}(a|s) G_t]$$

## Cross Entropy 관점에서의 REINFORCE

지도학습의 분류 문제에서 사용되는 손실함수인 Cross Entropy

$$\mathcal{L} = - \sum_i [y_i \log p_i]$$

- $y$ 의 분포와  $p$ 의 분포가 얼마나 가까운지를 측정하는 함수.
- $y$ : 정답 분포
- $p$ : 인공 신경망의 예측한 분포

## Cross Entropy 관점에서의 REINFORCE

REINFORCE 알고리즘의 손실함수를 살펴보면

$$\log \pi_{\theta}(a|s)$$

즉, 실제로 한 행동을 정답으로 하는 것을 볼 수 있다.

하지만 좋지 않은 행동을 더 강화하는 효과를 야기할 수 있다.


→ 반환값  $G_t$ 를 곱해 이러한 문제를 해결한다.

# REINFORCE 알고리즘

- 1 인공 신경망을 랜덤하게 초기화한다.
- 2 에피소드를 수행하고, 끝나면 각 timestep  $t$ 마다  $G_t$ 를 계산한다.
- 3 오차 함수  $\mathcal{L} = -\sum_t G_t \log \pi_{\theta}(a_t|s_t)$ 를 계산한다.
- 4 SGD를 이용해 신경망을 갱신한다.
- 5 수렴할 때까지 2번으로 돌아간다.



# 구현하기



```
import torch
from torch import nn, optim
from torch.distributions import Categorical

import numpy as np
import gym
```

# 구현하기

```
class PolicyNet(nn.Module):  
    def __init__(self):  
        super(PolicyNet, self).__init__()  
  
        self.net = nn.Sequential(  
            nn.Linear(4, 64),  
            nn.ReLU(inplace=True),  
            nn.Linear(64, 2),  
            nn.Softmax(dim=0)  
        )  
  
    def forward(self, x):  
        return self.net(x)
```


# 구현하기



```
class Agent:
    def __init__(self, device):
        self.device = device
        self.net = PolicyNet().to(device)
        self.opt = optim.Adam(self.net.parameters(), lr=LEARNING_RATE)

        self.log_probs = []
        self.rewards = []
```

# 구현하기



```
def get_action(self, state):  
    policy = self.net(state.to(self.device))  
  
    m = Categorical(policy)  
    action = m.sample()  
  
    self.log_probs.append(m.log_prob(action))  
    return action.item()
```

# 구현하기

```
def train(self):
    R = 0; losses = []; returns = []

    for r in self.rewards[::-1]:
        R = r + DISCOUNT_FACTOR * R
        returns.insert(0, R)

    returns = torch.tensor(returns).to(device)
    returns = (returns - returns.mean()) / (returns.std() + 1e-10)

    for log_prob, R in zip(self.log_probs, returns):
        losses.append(-log_prob * R)


    self.opt.zero_grad()

    loss = sum(losses)
    loss.backward()

    self.opt.step()

    self.log_probs = []; self.rewards = []
```

# 구현하기



```
while True:
    obs = env.reset()
    obs = torch.FloatTensor(obs)

    total_reward = 0

    while True:
        action = agent.get_action(obs)
        next_obs, reward, done, _ = env.step(action)

        agent.rewards.append(reward)
        total_reward += reward

        if done:
            break

    obs = torch.FloatTensor(next_obs)

agent.train()
```