Policy Gradient Method

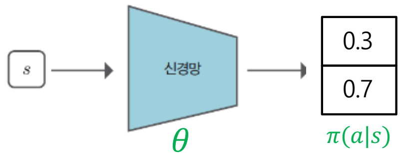
Prof. Tae-Hyoung Park
Dept. of Intelligent Systems & Robotics, CBNU

Value-Based vs. Policy Gradient

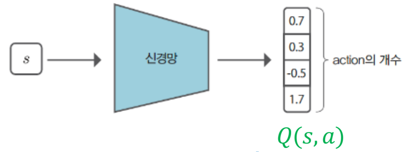
- 강화학습의 목적
  - 최적의 policy 를 구하는 것 : action 선택
- Value-Based Method (가치 기반 방법)
  - Value function (state value, action value) 을 학습/평가 후, 이를 통하여 policy 를 개선하는 방법
  - SARSA, Q-Learning, DQN, Rainbow 등
- Policy Gradient Method (정책 경사 방법)
  - Policy function 을 직접 파라메터화 하여 학습/개선
  - Policy Gradient, REINFORCE, PPO, A3C 등

Policy Gradient

- Policy Function (정책 함수)
  - $\pi(a|s)$: policy function, state s 에서 action a 를 선택할 확률
  - $\pi_\theta(a|s)$: 신경망으로 구현한 policy function (policy network)
    - θ = 신경망의 weight 벡터



(policy network)



(Q network)

- Objective Function
  - $\tau = (S_0, A_0, R_0, S_1, A_2, R_3, \cdots, S_T, A_T, R_T, S_{T+1})$: trajectory
  - $G(\tau) = R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots + \gamma^T R_T$: return (수익)
  - $J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G(\tau)]$: objective function
    - $J(\theta)$: policy network θ 에 대한 기대 수익 → 최대화
    - $\tau \sim \pi_\theta$: 시계열 trajectory τ 가 policy 신경망 π_θ 로 부터 생성됨

Policy Gradient

- Optimization
  - $J(\theta)$ 를 최대화 시키는 policy network θ 를 구하는 문제
  - Gradient ascent method
    - Gradient

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)]$$

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)]$$

$$\boxed{= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G(\tau)\nabla_\theta \log \pi_\theta(A_t|S_t)\right]}$$

    - Update

$$\boxed{\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)}$$

기울기의 (+) 방향으로 이동 → local maxima 도달

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)] \\
&= \nabla_\theta \sum_\tau \Pr(\tau|\theta)G(\tau) && \text{(기댓값 확장)} \\
&= \sum_\tau \nabla_\theta(\Pr(\tau|\theta)G(\tau)) && (\nabla_\theta\text{를 } \textstyle\sum \text{ 안으로 이동}) \\
&= \sum_\tau \{G(\tau)\nabla_\theta \Pr(\tau|\theta) + \Pr(\tau|\theta)\nabla_\theta G(\tau)\} && \text{(곱의 미분)} \\
&= \sum_\tau G(\tau)\nabla_\theta \Pr(\tau|\theta) && (\nabla_\theta G(\tau)\text{는 항상 } 0) \\
&= \sum_\tau G(\tau)\Pr(\tau|\theta)\frac{\nabla_\theta \Pr(\tau|\theta)}{\Pr(\tau|\theta)} && \left(\text{곱하기 } \frac{\Pr(\tau|\theta)}{\Pr(\tau|\theta)}\right) \\
&= \sum_\tau G(\tau)\Pr(\tau|\theta)\nabla_\theta \log \Pr(\tau|\theta) && \text{(로그-기울기 트릭)}
\end{aligned}$$

$$\begin{aligned}
\Pr(\tau|\theta) &= p(S_0)\pi_\theta(A_0|S_0)p(S_1|S_0,A_0)\cdots\pi_\theta(A_T|S_T)p(S_{T+1}|S_T,A_T) \\
&= p(S_0)\prod_{t=0}^{T}\pi_\theta(A_t|S_t)p(S_{t+1}|S_t,A_t)
\end{aligned}$$

$$\log \Pr(\tau|\theta) = \log p(S_0) + \sum_{t=0}^{T}\log p(S_{t+1}|S_t,A_t) + \sum_{t=0}^{T}\log \pi_\theta(A_t|S_t)$$

$$\begin{aligned}
\nabla_\theta \log \Pr(\tau|\theta) &= \nabla_\theta\left\{\log p(S_0) + \sum_{t=0}^{T}\log p(S_{t+1}|S_t,A_t) + \sum_{t=0}^{T}\log \pi_\theta(A_t|S_t)\right\} \\
&= \nabla_\theta \sum_{t=0}^{T}\log \pi_\theta(A_t|S_t)
\end{aligned}$$

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)\nabla_\theta \log \Pr(\tau|\theta)] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G(\tau)\nabla_\theta \log \pi_\theta(A_t|S_t)\right]
\end{aligned}$$

Policy Gradient

- Algorithm
  - $\nabla_\theta J(\theta)$ 를 구하는 알고리즘

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G(\tau)\nabla_\theta \log \pi_\theta(A_t|S_t)\right]$$

  - Monte Carlo method 적용
    - sampling 을 여러 번하여 평균을 구함
    - Agent 를 policy π_θ 에 따라 행동하게 하여 n 개의 trajectory τ 를 얻음

$$\text{샘플링}: \tau^{(i)} \sim \pi_\theta \quad (i = 1, 2, \cdots, n)$$

$$x^{(i)} = \sum_{t=0}^{T} G(\tau^{(i)}) \nabla_\theta \log \pi_\theta (A_t^{(i)} \mid S_t^{(i)})$$

$$\nabla_\theta J(\theta) \simeq \frac{x^{(1)} + x^{(2)} + \cdots + x^{(n)}}{n}$$

$\tau^{(i)}$: i 번째 episode 의 trajectory

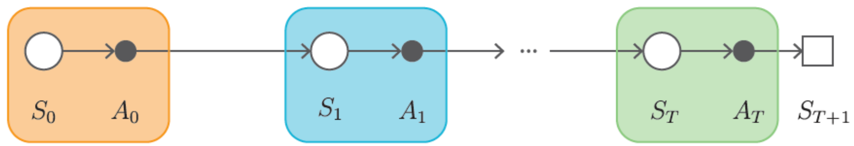
$A_t^{(i)}$, $S_t^{(i)}$: i 번째 episode 의 시간 t 에서의 action, state

  - 1 sample case (n=1)

$$\text{샘플링}: \tau \sim \pi_\theta$$

$$\nabla_\theta J(\theta) \simeq \sum_{t=0}^{T} G(\tau) \nabla_\theta \log \pi_\theta (A_t \mid S_t)$$

$$\nabla_\theta \log \pi_\theta (A_t \mid S_t) = \frac{\nabla_\theta \pi_\theta (A_t \mid S_t)}{\pi_\theta (A_t \mid S_t)}$$



$$G(\tau)\nabla_\theta \log \pi_\theta(A_0|S_0) + G(\tau)\nabla_\theta \log \pi_\theta(A_1|S_1) + \cdots + G(\tau)\nabla_\theta \log \pi_\theta(A_T|S_T)$$

$\pi_\theta(A_t|S_t)$: state S_t 에서 action A_t 를 선택할 확률

$\nabla_\theta \pi_\theta(A_t|S_t)$: state S_t 에서 action A_t 를 선택할 확률의 변화량(기울기)

Policy Gradient

- Implementation
  - Policy Network (π_θ) : 2 layer NN, classification model

```python
class Policy(Model):
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)          # 첫 번째 계층
        self.l2 = L.Linear(action_size)  # 두 번째 계층

    def forward(self, x):
        x = F.relu(self.l1(x))     # 첫 번째 계층에서는 ReLU 함수 사용
        x = F.softmax(self.l2(x))  # 두 번째 계층에서는 소프트맥스 함수 사용
        return x
```

Cart pole
입력: state 4 x batch size 32 = 128
출력: action_size = 2
action probability (softmax())

$$y_i = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}$$

Policy Gradient

- Implementation
  - Agent class

```python
class Agent:
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0002
        self.action_size = 2

        self.memory = []
        self.pi = Policy(self.action_size)
        self.optimizer = optimizers.Adam(self.lr)
        self.optimizer.setup(self.pi)

    def get_action(self, state):
        state = state[np.newaxis, :]  # 배치 처리용 축 추가
        probs = self.pi(state)        # 순전파 수행
        probs = probs[0]
        action = np.random.choice(len(probs), p=probs.data)  # 행동 선택
        return action, probs[action]  # 선택된 행동과 확률 반환

    def add(self, reward, prob):
        data = (reward, prob)
        self.memory.append(data)

    def update(self):
        self.pi.cleargrads()

        G, loss = 0, 0
        for reward, prob in reversed(self.memory):  # 수익 G 계산
            G = reward + self.gamma * G

        for reward, prob in self.memory:  # 손실 함수 계산
            loss += -F.log(prob) * G

        loss.backward()
        self.optimizer.update()
        self.memory = []  # 메모리 초기화
```

Policy network 초기화

Policy network 출력 $(\pi_\theta(A_t|S_t)) \rightarrow$ action 선택

$G(\tau)$ 계산

$$loss = -\nabla_\theta J(\theta) = -\sum_{t=0}^{T} G(\tau)\log \pi_\theta(A_t|S_t)$$

Policy network update

$$\theta^* = \arg\max_\theta J(\theta) = \arg\min_\theta -J(\theta)$$

실습

실습 #1 Simple_pg2.py

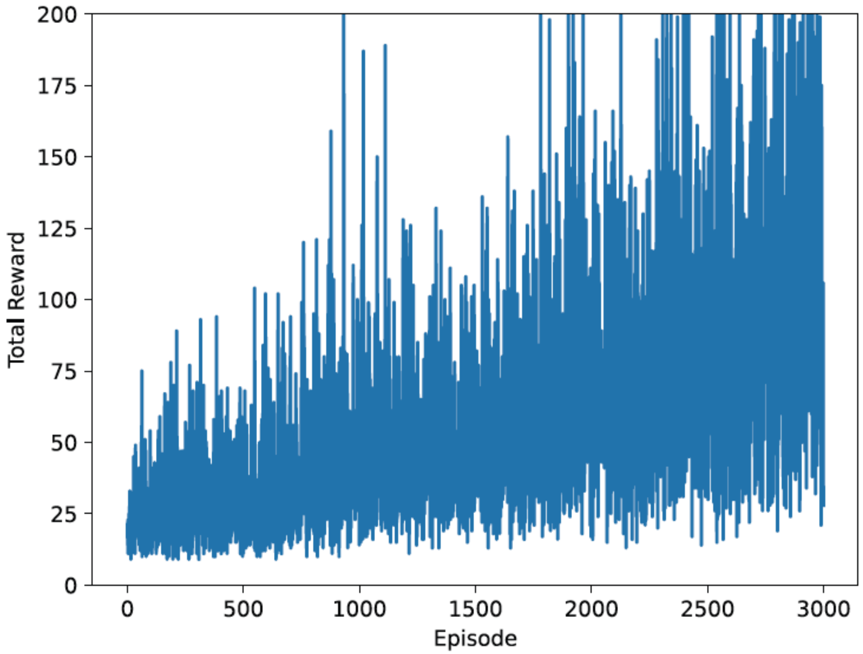
```python
import numpy as np
import gym
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L

class Policy(Model):
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)          # 첫 번째 계층
        self.l2 = L.Linear(action_size)  # 두 번째 계층

    def forward(self, x):
        x = F.relu(self.l1(x))     # 첫 번째 계층에서는 ReLU 함수 사용
        x = F.softmax(self.l2(x))  # 두 번째 계층에서는 소프트맥스 함수 사용
        return x

class Agent:
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0002
        self.action_size = 2

        self.memory = []
        self.pi = Policy(self.action_size)
        self.optimizer = optimizers.Adam(self.lr)
        self.optimizer.setup(self.pi)

    def get_action(self, state):
        state = state[np.newaxis, :]  # 배치 처리용 축 추가
        probs = self.pi(state)        # 순전파 수행
        probs = probs[0]
        action = np.random.choice(len(probs), p=probs.data)  # 행동 선택
        return action, probs[action]  # 선택된 행동과 확률 반환

    def add(self, reward, prob):
        data = (reward, prob)
        self.memory.append(data)
```

```python
    def update(self):
        self.pi.cleargrads()

        G, loss = 0, 0
        for reward, prob in reversed(self.memory):  # 수익 G 계산
            G = reward + self.gamma * G

        for reward, prob in self.memory:  # 손실 함수 계산
            loss += -F.log(prob) * G

        loss.backward()
        self.optimizer.update()
        self.memory = []  # 메모리 초기화

episodes = 3000
env = gym.make('CartPole-v0', render_mode='rgb_array')
agent = Agent()
reward_history = []

for episode in range(episodes):
    state = env.reset()[0]
    done = False
    total_reward = 0

    while not done:
        action, prob = agent.get_action(state)  # 행동 선택
        next_state, reward, terminated, truncated, info = env.step(action)  # 행동 수행
        done = terminated | truncated

        agent.add(reward, prob)  # 보상과 행동의 확률을 에이전트에 추가
        state = next_state       # 상태 전이
        total_reward += reward   # 보상 총합 계산

    agent.update()  # 정책 갱신

    reward_history.append(total_reward)
    if episode % 100 == 0:
        print("episode :{}, total reward : {:.1f}".format(episode, total_reward))

# 에피소드별 보상 합계 추이
from common.utils import plot_total_reward
plot_total_reward(reward_history)
```

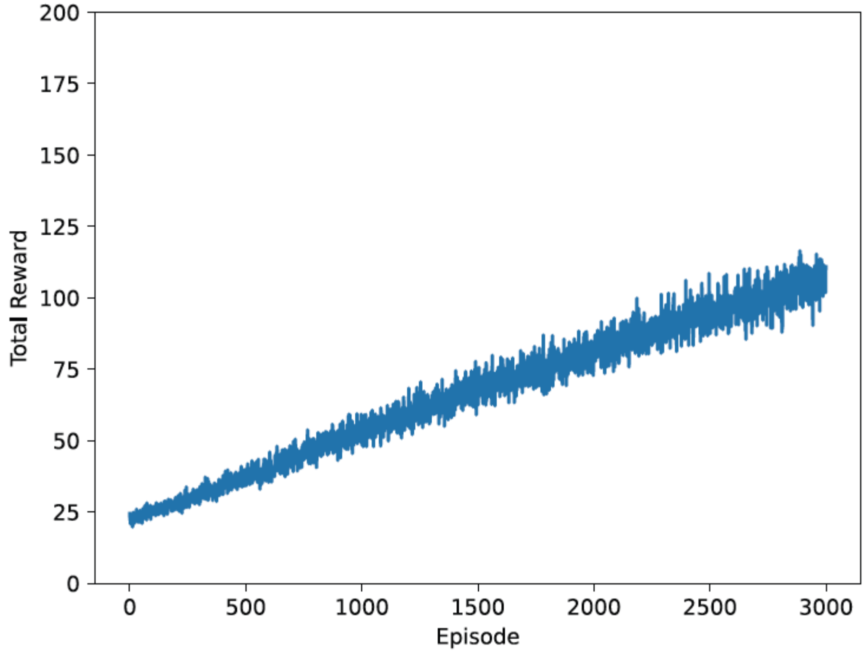
실습

```python
# 학습이 끝난 에이전트에 탐욕 행동을 선택하도록 하여 플레이
env2 = gym.make('CartPole-v0', render_mode='human')

state = env2.reset()[0]
done = False
total_reward = 0

while not done:
    action, prob = agent.get_action(state)
    next_state, reward, terminated, truncated, info = env2.step(action)
    done = terminated | truncated
    agent.add(reward, prob)
    state = next_state
    total_reward += reward
    env2.render()
print('Total Reward:', total_reward)
```

실습



Episode – total reward



Episode – total reward
(100 회 평균)

REINFORCE

- Basics
  - Ronald J. Williams, 1992
  - **Re**ward **I**ncrement = **N**onnegative **F**actor x **O**ffset **R**einforcement x **C**haracter **E**ligibility
  - Modified policy gradient method
- Policy Gradient 의 문제점

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} G(\tau) \nabla_\theta \log \pi_\theta(A_t|S_t)\right] \qquad ①$$

$G(\tau) = R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots + \gamma^T R_T$: 전체 기간 ($t = 0 \sim T$) 의 수익

  - 시간 t 에서의 action A_t 에 항상 일정한 가중치 $G(\tau)$ 를 적용: noise
  - action A_t '이후' 발생한 수익을 가중치로 부여하는 것이 합리적

REINFORCE

- Algorithm

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} G_t \, \nabla_\theta \log \pi_\theta(A_t|S_t)\right] \qquad ②$$

$G_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t} R_T$: 시간 t 이후에 발생한 수익

– ①, ② 모두 샘플 수를 늘리면 정확한 $\nabla_\theta J(\theta)$ 에 수렴

– ① 이 ② 보다 분산이 큼.

(① 의 가중치에는 관련 없는 데이터 noise 포함)

REINFORCE

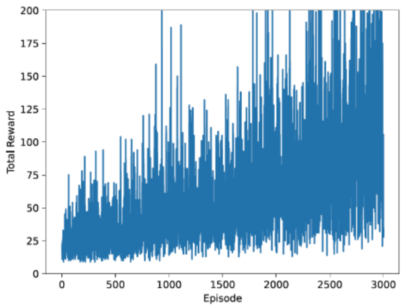
- Implementation

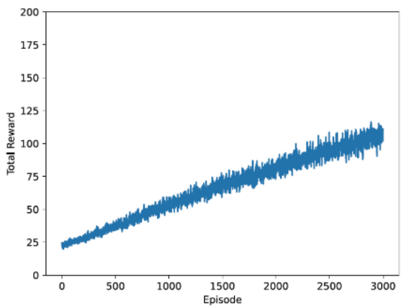
Simple_pg

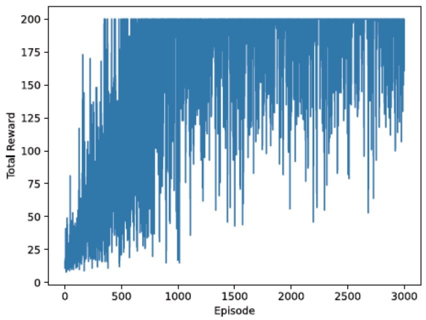
```
def update(self):
    self.pi.cleargrads()

    G, loss = 0, 0
    for reward, prob in reversed(self.memory):  # 수익 G 계산
        G = reward + self.gamma * G

    for reward, prob in self.memory:  # 손실 함수 계산
        loss += -F.log(prob) * G

    loss.backward()
    self.optimizer.update()
    self.memory = []  # 메모리 초기화
```

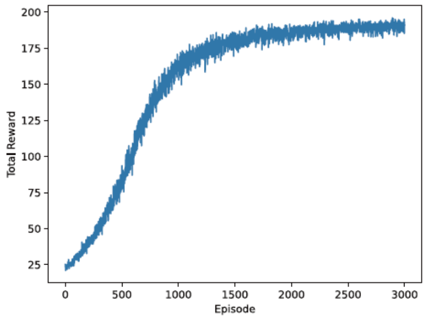
REINFORCE

```
def update(self):
    self.pi.cleargrads()

    G, loss = 0, 0
    for reward, prob in reversed(self.memory):
        G = reward + self.gamma * G  # 수익 G 계산
        loss += -F.log(prob) * G     # 손실 함수 계산

    loss.backward()
    self.optimizer.update()
    self.memory = []
```









✓ 안정적이고 빠른 학습

실습

실습 #2 Reinforce2.py

```python
import numpy as np
import gym
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L


class Policy(Model):
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.softmax(self.l2(x))
        return x


class Agent:
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0002
        self.action_size = 2

        self.memory = []
        self.pi = Policy(self.action_size)
        self.optimizer = optimizers.Adam(self.lr)
        self.optimizer.setup(self.pi)

    def get_action(self, state):
        state = state[np.newaxis, :]
        probs = self.pi(state)
        probs = probs[0]
        action = np.random.choice(len(probs), p=probs.data)
        return action, probs[action]

    def add(self, reward, prob):
        data = (reward, prob)
        self.memory.append(data)
```

```python
    def update(self):
        self.pi.cleargrads()

        G, loss = 0, 0
        for reward, prob in reversed(self.memory):
            G = reward + self.gamma * G  # 수익 G 계산
            loss += -F.log(prob) * G     # 손실 함수 계산

        loss.backward()
        self.optimizer.update()
        self.memory = []


episodes = 3000
env = gym.make('CartPole-v0', render_mode='rgb_array')
agent = Agent()
reward_history = []

for episode in range(episodes):
    state = env.reset()[0]
    done = False
    sum_reward = 0

    while not done:
        action, prob = agent.get_action(state)
        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated | truncated

        agent.add(reward, prob)
        state = next_state
        sum_reward += reward

    agent.update()

    reward_history.append(sum_reward)
    if episode % 100 == 0:
        print("episode :{}, total reward : {:.1f}".format(episode, sum_reward))

# 그래프
from common.utils import plot_total_reward
plot_total_reward(reward_history)
```

실습

```python
# 학습이 끝난 에이전트에 탐욕 행동을 선택하도록 하여 플레이
env2 = gym.make('CartPole-v0', render_mode='human')

state = env2.reset()[0]
done = False
total_reward = 0

while not done:
    action, prob = agent.get_action(state)
    next_state, reward, terminated, truncated, info = env2.step(action)
    done = terminated | truncated
    agent.add(reward, prob)
    state = next_state
    total_reward += reward
    env2.render()
print('Total Reward:', total_reward)
```

Baseline

- Idea

① 3명의 시험성적 <실제값>

A	90
B	40
C	50

분산=466.667

(3명의 과거 시험성적)

	첫 번째 시험	두 번째 시험	...	열 번째 시험
A	92	80	...	74
B	32	51	...	56
C	45	53	...	49

<평균값/예측값)>

A	82
B	46
C	49

② 3명의 시험성적 <실제값> - <예측값>

A	90
B	40
C	50

−

A	82
B	46
C	49

=

A	8
B	-6
C	1

분산=32.667

- 데이터의 분산을 줄이기 위하여 실제값과 예측값의 차이를 활용
- 예측값의 정확도가 높을수록 분산이 작아짐

Baseline

- Policy Gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right] \qquad \text{(REINFORCE)}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} (G_t - \underline{b(S_t)}) \nabla_\theta \log \pi_\theta(A_t|S_t)\right] \qquad b(S_t) : \text{baseline}$$

$V(S)$

(proof)

$$\sum_x P_\theta(x) = 1$$

$$\downarrow$$

$$\nabla_\theta \sum_x P_\theta(x) = \nabla_\theta 1 = 0$$

$$\downarrow$$

$$\begin{aligned} &\nabla_\theta \sum_x P_\theta(x) \\ &= \sum_x \nabla_\theta P_\theta(x) \\ &= \sum_x P_\theta(x) \nabla_\theta \log P_\theta(x) \\ &= \mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0 \end{aligned}$$

$$\mathbb{E}_{A_t\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(A_t \mid S_t)] = 0$$

$$\downarrow$$

$$\mathbb{E}_{A_t\sim\pi_\theta}[b(S_t) \nabla_\theta \log \pi_\theta(A_t \mid S_t)] = 0$$

$b(S_t)$는 A_t 와 독립적

$$\downarrow$$

$$\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} b(S_t) \nabla_\theta \log \pi_\theta(A_t \mid S_t)\right] = 0$$

$t = 0 \sim T$ 에 모두성립

Baseline

- Baseline Idea
  - $b(S_t)$: state S_t 에서 지금까지 얻은 보상의 평균 (ex. 시험성적 평균치)

    => state value function $V_{\pi_\theta}(S_t)$
  - 분산을 줄이고 학습 효율 증가

Actor-Critic

- Policy Gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}(G_t - b(S_t))\, \nabla_\theta \log \pi_\theta(A_t|S_t)\right]$$

REINFORCE with baseline

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}(G_t - V_w(S_t))\, \nabla_\theta \log \pi_\theta(A_t|S_t)\right]$$

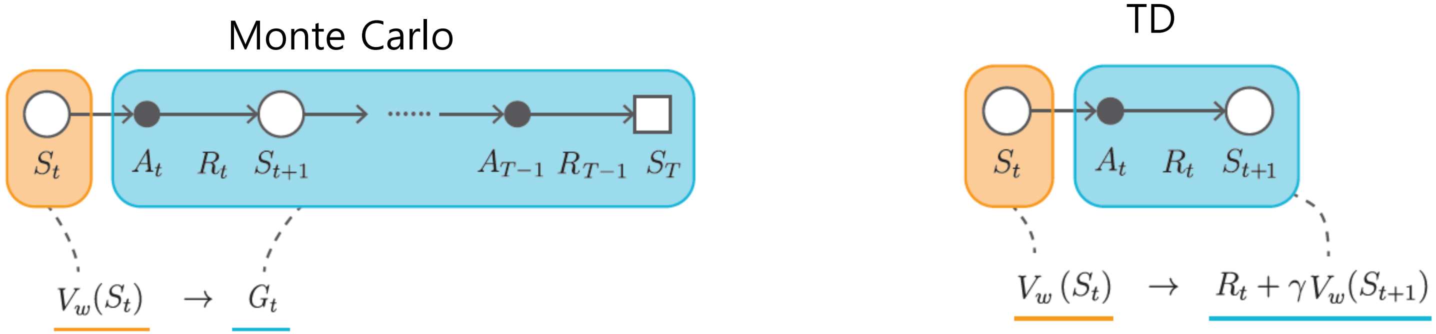
Actor-Critic (Monte Carlo)

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}(R_t + \gamma V_w(S_{t+1}) - V_w(S_t))\, \nabla_\theta \log \pi_\theta(A_t|S_t)\right]$$

Actor-Critic (TD)

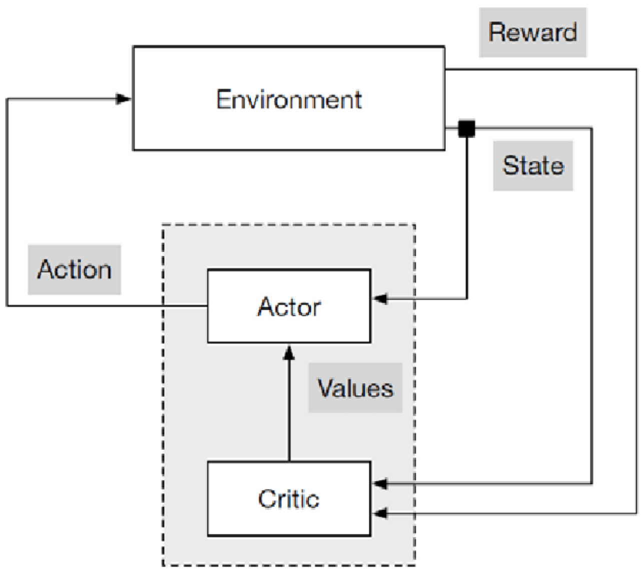
θ : policy network (actor) 의 weight 벡터
w : value network (critic) 의 weight 벡터

- $V_w(S_t)$: : value function 을 모델링한 신경망, value network
- G_t : TD 법으로 Return 값 계산



Actor-Critic

- Policy-Based & Value-Based (Hybrid method)
  - Actor (행위자)
    - Agent 의 action 을 결정하는 policy 를 학습
    - Policy-based = actor only
  - Critic (비평자)
    - 주어진 state 에서의 value function 을 학습
    - Value-based = critic only



Actor: policy network

Critic: value network

Actor-Critic

- Implementation
  - PolicyNet (Actor), ValueNet (Critic)

```
class PolicyNet(Model):  # 정책 신경망
    def __init__(self, action_size=2):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = self.l2(x)
        x = F.softmax(x)  # 확률 출력
        return x


class ValueNet(Model):  # 가치 함수 신경망
    def __init__(self):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(1)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = self.l2(x)
        return x
```

Actor-Critic

- Implementation
  - Agent

```python
class Agent:
    def __init__(self):
        self.gamma = 0.98
        self.lr_pi = 0.0002
        self.lr_v = 0.0005
        self.action_size = 2

        self.pi = PolicyNet()
        self.v = ValueNet()
        self.optimizer_pi = optimizers.Adam(self.lr_pi).setup(self.pi)
        self.optimizer_v = optimizers.Adam(self.lr_v).setup(self.v)

    def get_action(self, state):
        state = state[np.newaxis, :]  # 배치 처리용 축 추가
        probs = self.pi(state)
        probs = probs[0]
        action = np.random.choice(len(probs), p=probs.data)
        return action, probs[action]  # 선택된 행동과 해당 행동의 확률 반환
```

Actor-Critic

- Implementation
  - Agent

```python
def update(self, state, action_prob, reward, next_state, done):
    # 배치 처리용 축 추가
    state = state[np.newaxis, :]
    next_state = next_state[np.newaxis, :]

    # 가치 함수(self.v)의 손실 계산
    target = reward + self.gamma * self.v(next_state) * (1 - done)  # TD 목표
    target.unchain()
    v = self.v(state)  # 현재 상태의 가치 함수
    loss_v = F.mean_squared_error(v, target)  # 두 값의 평균 제곱 오차

    # 정책(self.pi)의 손실 계산
    delta = target - v
    delta.unchain()
    loss_pi = -F.log(action_prob) * delta

    # 신경망 학습
    self.v.cleargrads()
    self.pi.cleargrads()
    loss_v.backward()
    loss_pi.backward()
    self.optimizer_v.update()
    self.optimizer_pi.update()
```

$R_t + \gamma V_w(S_{t+1})$

$V_w(S_t)$

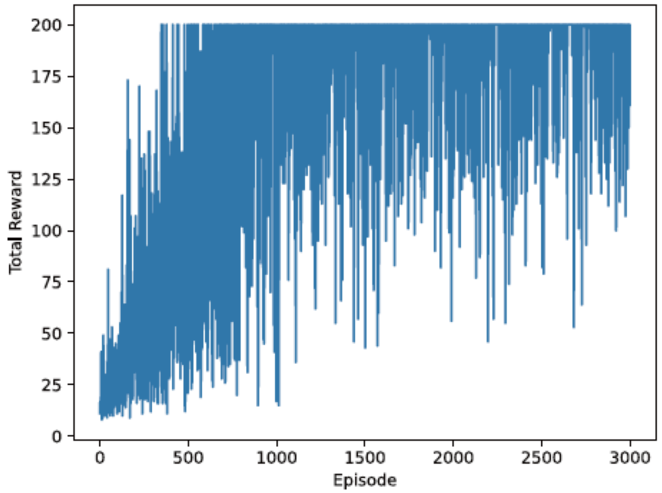
loss_v = $\|R_t + \gamma V_w(S_{t+1}) - V_w(S_t)\|$

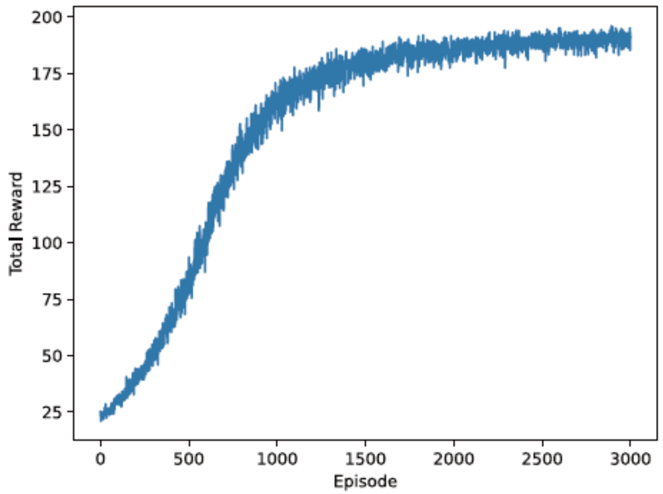
loss_pi =
$-(R_t + \gamma V_w(S_{t+1}) - V_w(S_t))\nabla_\theta \log \pi_\theta(A_t|S_t)$

Actor-Critic

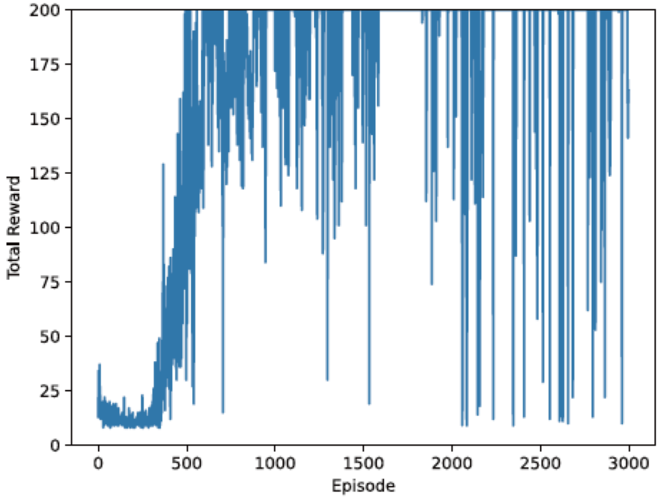
- Implementation

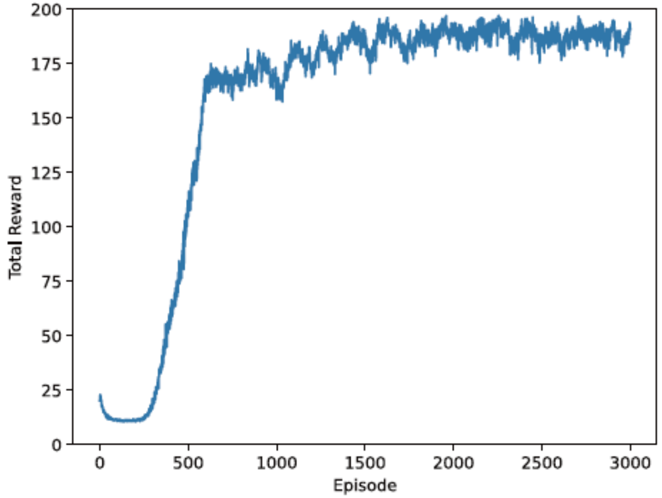
(REINFORCE)





(Actor-Critic)





실습

실습 #3 Actor_critic2.py

```python
import numpy as np
import gym
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L

class PolicyNet(Model):  # 정책 신경망
    def __init__(self, action_size=2):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = self.l2(x)
        x = F.softmax(x)  # 확률 출력
        return x

class ValueNet(Model):  # 가치 함수 신경망
    def __init__(self):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(1)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = self.l2(x)
        return x

class Agent:
    def __init__(self):
        self.gamma = 0.98
        self.lr_pi = 0.0002
        self.lr_v = 0.0005
        self.action_size = 2

        self.pi = PolicyNet()
        self.v = ValueNet()
        self.optimizer_pi = optimizers.Adam(self.lr_pi).setup(self.pi)
        self.optimizer_v = optimizers.Adam(self.lr_v).setup(self.v)
```

```python
    def get_action(self, state):
        state = state[np.newaxis, :]  # 배치 처리용 축 추가
        probs = self.pi(state)
        probs = probs[0]
        action = np.random.choice(len(probs), p=probs.data)
        return action, probs[action]  # 선택된 행동과 해당 행동의 확률 반환

    def update(self, state, action_prob, reward, next_state, done):
        # 배치 처리용 축 추가
        state = state[np.newaxis, :]
        next_state = next_state[np.newaxis, :]

        # 가치 함수(self.v)의 손실 계산
        target = reward + self.gamma * self.v(next_state) * (1 - done)  # TD 목표
        target.unchain()
        v = self.v(state)  # 현재 상태의 가치 함수
        loss_v = F.mean_squared_error(v, target)  # 두 값의 평균 제곱 오차

        # 정책(self.pi)의 손실 계산
        delta = target - v
        delta.unchain()
        loss_pi = -F.log(action_prob) * delta

        # 신경망 학습
        self.v.cleargrads()
        self.pi.cleargrads()
        loss_v.backward()
        loss_pi.backward()
        self.optimizer_v.update()
        self.optimizer_pi.update()
```

실습

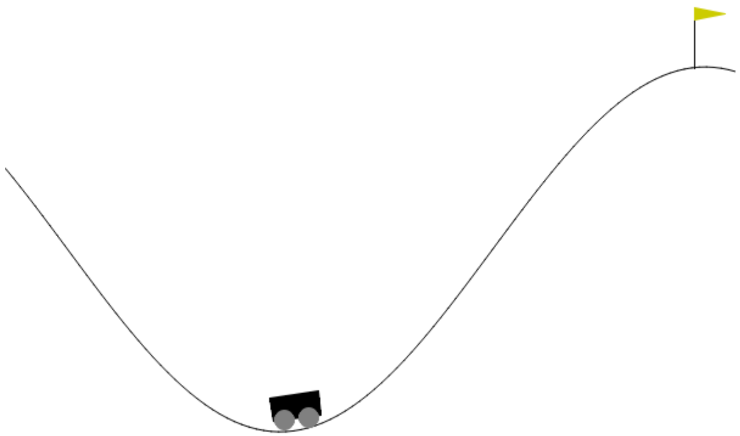
```python
for episode in range(episodes):
    state = env.reset()[0]
    done = False
    total_reward = 0

    while not done:
        action, prob = agent.get_action(state)
        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated | truncated

        agent.update(state, prob, reward, next_state, done)

        state = next_state
        total_reward += reward

    reward_history.append(total_reward)
    if episode % 100 == 0:
        print("episode :{}, total reward : {:.1f}".format(episode, total_reward))


# 그래프
from common.utils import plot_total_reward
plot_total_reward(reward_history)

# 학습이 끝난 에이전트에 탐욕 행동을 선택하도록 하여 플레이
env2 = gym.make('CartPole-v0', render_mode='human')

state = env2.reset()[0]
done = False
total_reward = 0

while not done:
    action, prob = agent.get_action(state)
    next_state, reward, terminated, truncated, info = env2.step(action)
    done = terminated | truncated

    agent.update(state, prob, reward, next_state, done)

    state = next_state
    total_reward += reward
    env2.render()
print('Total Reward:', total_reward)
```

Quiz

(Q1) Actor-Critic 을 Mountain Car 문제에 적용하되, Hyper-parameter 를 변경하여 최대의 total reward 를 갖는 policy 를 결정하라.

(제출물: PPT)

1) 프로그램 소스
2) 최적 hyperparameter
3) Episode 별 total reward graph
4) 최대 total reward 값 및 해당 policy 적용 시의 동영상



요약

- Policy Gradient Method (정책 경사법)

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \Phi_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right]$$

  - $\Phi_t = G(\tau)$: Simple policy gradient
  - $\Phi_t = G_t$: REINFORCE
  - $\Phi_t = G_t - b(S_t)$: REINFORCE with baseline
  - $\Phi_t = R_t + \gamma V_w(S_{t+1}) - V_w(S_t)$: Actor-Critic

(참고) Φ_t 에 state value function 대신 action value function 사용 가능

$$\Phi_t = Q(S_t, A_t)$$

$$\Phi_t = Q(S_t, A_t) - V(S_t) = A(S_t, A_t)$$

요약

- Value-Based vs Policy-Based

Feature	Value-Based	Policy-Based
Learns	State/action values (e.g., Q-values)	Policy directly (직접 policy 결정.효율적임)
Action selection	Choose action with highest value (ε-greedy)	Sample from learned policy (softmax)
Strength	Good for discrete actions (cart pole)	Good for continuous actions (pendulum)
Weakness	Can be unstable in large/continuous spaces	High variance in learning