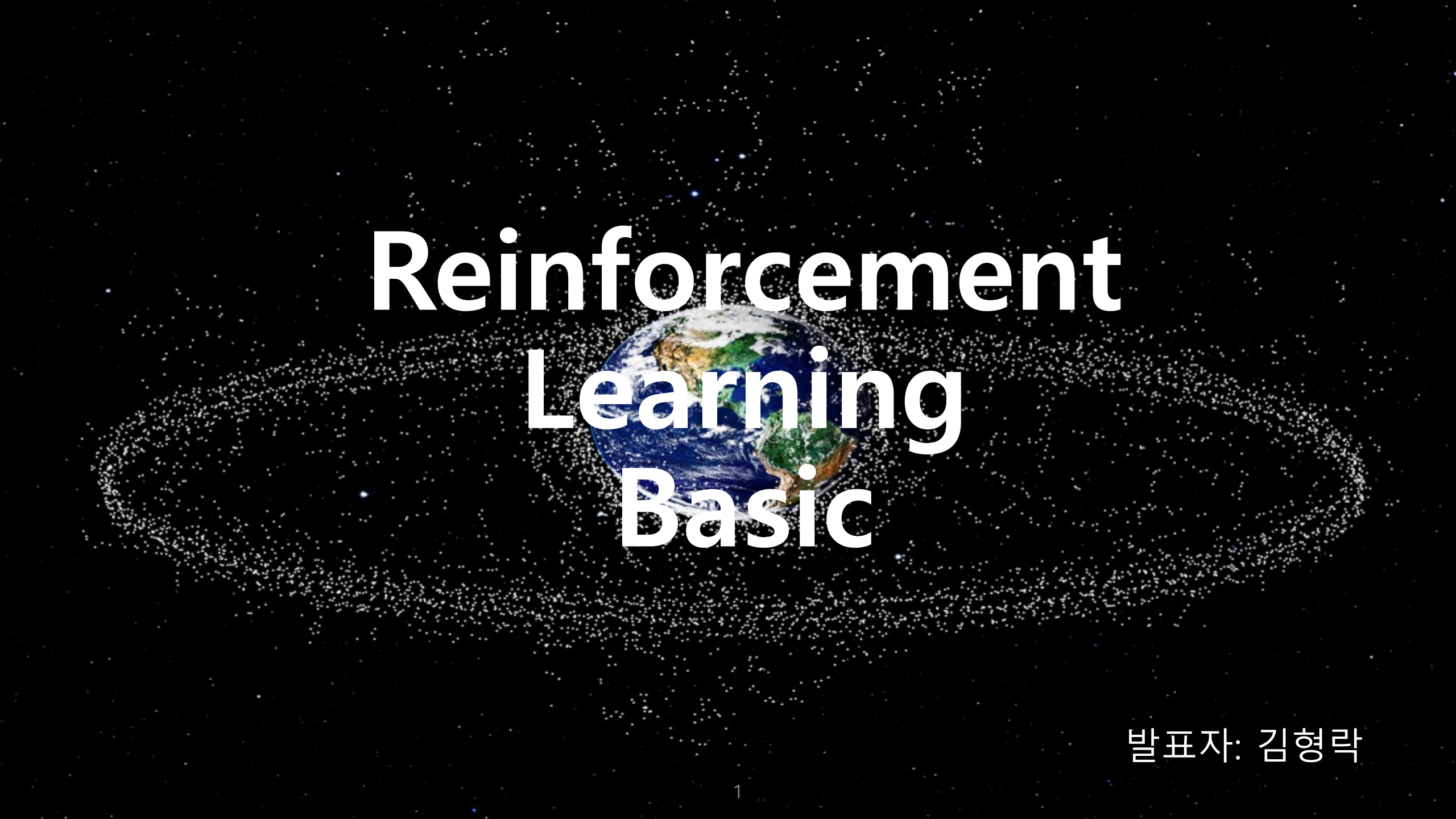


Reinforcement Learning Basic

발표자: 김형락

Content

1. Reinforcement Learning
    1. MP & MDP
    2. State Value Function& Action Value Function
    3. Dynamic Programming(Model Based)
    4. Monte-Carlo Method(Model Free)
    5. Temporal Difference
    6. DQN
2. Open AI Gym
3. Practice(Open AI Gym “CartPole-v1”)

1. Reinforcement Learning

- MP(Markov Property)

Definition

A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t]$$

- 미래의 상태는 오직 현재에 상태에만 영향을 받는 것
- Ex)
  - 동전던지기?
- 모델을 단순하게 정의

1. Reinforcement Learning

- MDP(Markov Decision Processes)

Markov Decision Processes

Instructors: Dan Klein and Pieter Abbeel

University of California, Berkeley

A Markov decision process (MDP) is a Markov reward process with decisions. It is an *environment* in which all states are Markov.

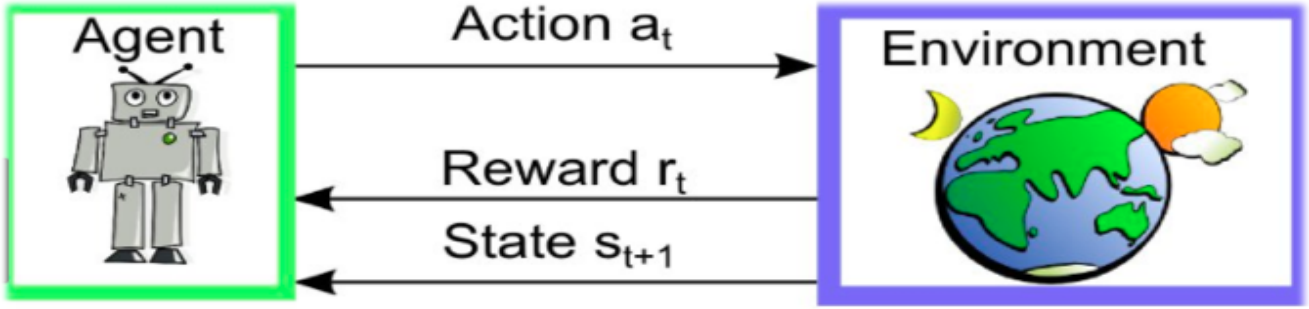
Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}^{a}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^{a}_{s} = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$
- γ is a discount factor $\gamma \in [0, 1]$.

1. Reinforcement Learning

- State Value Function & Action Value Function



Reinforcement Learning Setup

Definition

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

Definition

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a]$$

1. Reinforcement Learning

- State Value Function

The value function can be decomposed into two parts:

- immediate reward R_{t+1}
- discounted value of successor state $\gamma v(S_{t+1})$

$$
\begin{aligned}
v(s) &= \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]
\end{aligned}
$$

1. Reinforcement Learning

- State Value Function & Action Value Function

The state-value function can again be decomposed into immediate reward plus discounted value of successor state,

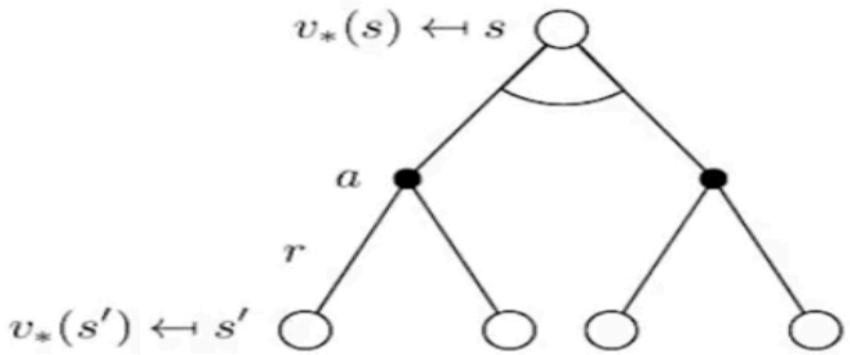
$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right]$$

The action-value function can similarly be decomposed,

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right]$$

1. Reinforcement Learning

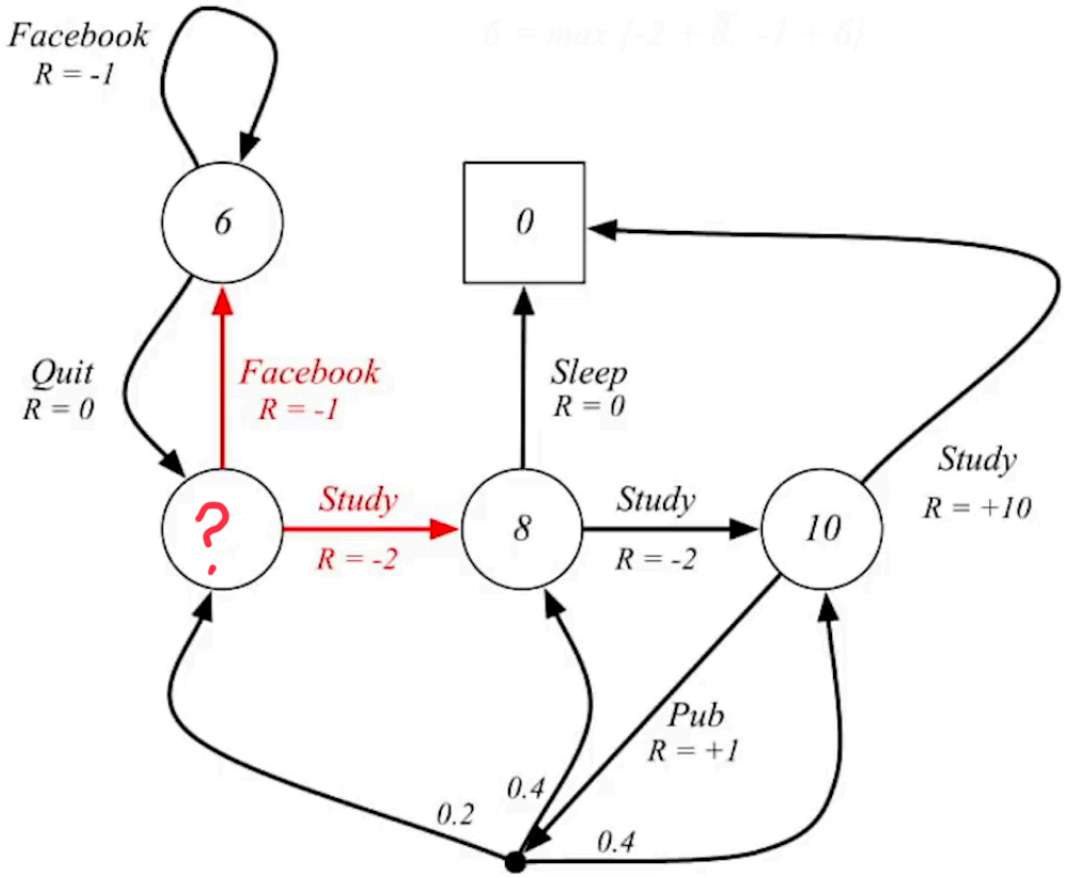
- Bellman Optimal Equation



$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

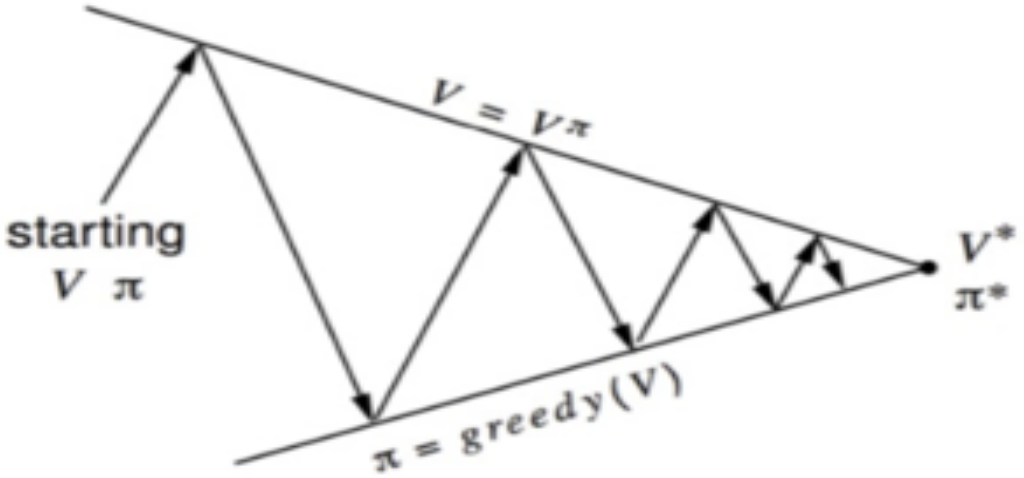
1. Reinforcement Learning

- Student MDP



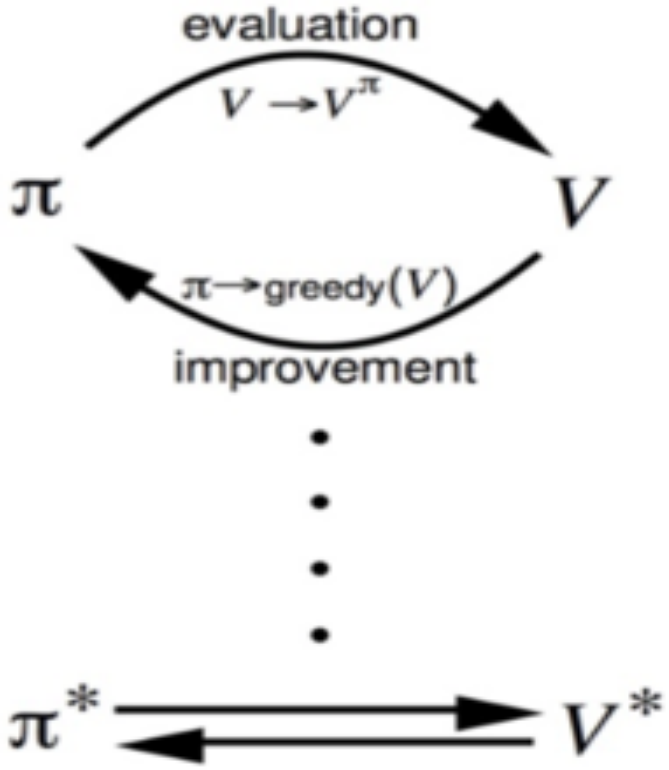
1. Reinforcement Learning

- Dynamic Programming



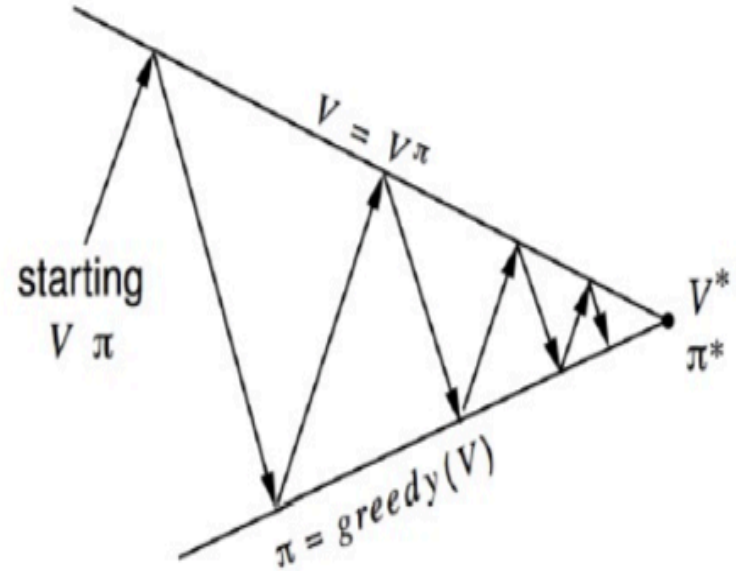
Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement



1. Reinforcement Learning

- Monte-Carlo Method(1/2)



Policy evaluation Monte-Carlo policy evaluation, $V = v_\pi$?

Policy improvement Greedy policy improvement?

1. Reinforcement Learning

- Monte-Carlo Method(2/2)

- Greedy policy improvement over $V(s)$ requires model of MDP

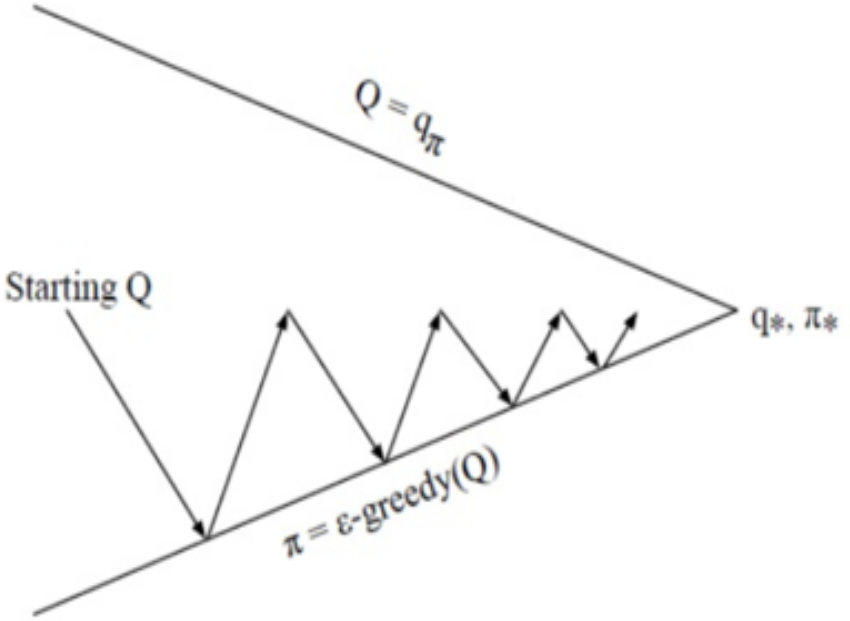
$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

- Greedy policy improvement over $Q(s, a)$ is model-free

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a)$$

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a) \\ \epsilon/m & \text{otherwise} \end{cases}$$



Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

1. Reinforcement Learning

- Temporal Difference(1/2)
  - Goal: learn v_π online from experience under policy π
  - Incremental every-visit Monte-Carlo
    - Update value $V(S_t)$ toward *actual* return G_t

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$

  - Simplest temporal-difference learning algorithm: TD(0)
    - Update value $V(S_t)$ toward *estimated* return $R_{t+1} + \gamma V(S_{t+1})$

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

    - $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
    - $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*

1. Reinforcement Learning

- Temporal Difference(2/2)

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

1. Reinforcement Learning

- Summary

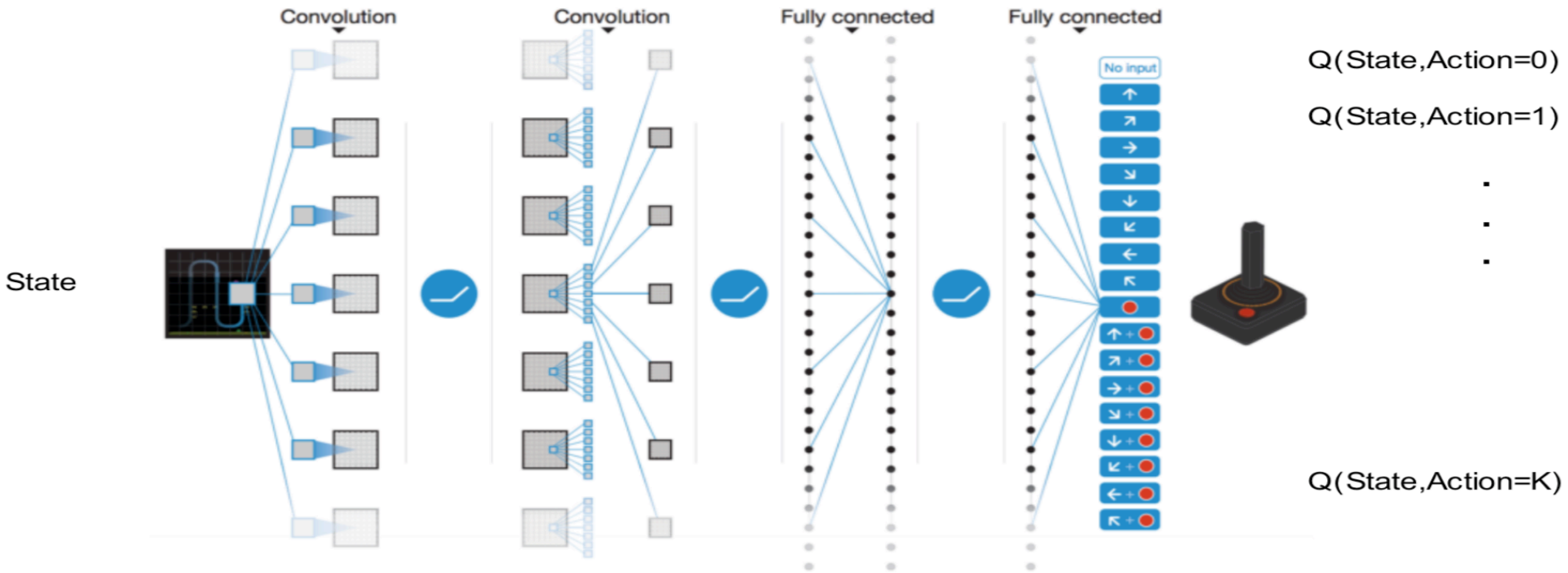
	DP	MC	TD
Advantage	Available Mathematical calculation	• Model Free • Conceptually simple	• Model Free • Step by step • N-Step
Disadvenatage	Accurate and Complete mode of ENV	Episodic High Variance	High Bias

1. Reinforcement Learning

- DQN

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

$$L = \frac{1}{2}[r + \gamma \max_a Q(s', a) - Q(s, a)]^2$$



1. Reinforcement Learning

- DQN
  - **Target Network Trick**
    - 학습 초기 $Q(s',a')$이 부정확하고 변화가 심함 ➔ 학습 성능 저하
    - DQN과 동일한 구조를 가지고 있으며 학습 도중 weight값이 변하지 않는 별도의 네트워크(Target Network)에서 $Q(s',a')$를 계산 - Target Network의 weight값들은 주기적으로 DQN의 것을 복사
  - **Replay Memory Trick**
    - Changing Data Distribution: Agent의 행동에 따라 들어오는 데이터의 분포가 변화함 (e.g. 어떤 mini batch를 학습한 후 무조건 왼쪽으로 가도록 policy가 변화 ➔ 이후 왼쪽으로 가지 않는 경우의 데이터를 얻을 수 없게 되어 학습이 불가능)
    - (State, Action, Reward, Next State) 데이터를 Buffer에 저장해 놓고 그 안에서 Random Sampling하여 mini batch를 구성하는 방법으로 해결
  - **Reward Clipping Trick**
    - 도메인에 따라 Reward의 크기가 다르기 때문에 Q value의 크기도 다름
    - Q value의 크기 variance가 매우 큰 경우 신경망 학습이 어려움
    - Reward의 크기를 [-1, +1] 사이로 제한하여 안정적 학습

1. Reinforcement Learning

- DQN

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

<Fig 1. Algorithm>

2. Open AI Gym

- Gym is a toolkit for developing and comparing reinforcement learning algorithm s. It supports teaching agents everything from walking to playing games like Pong or Pinball

SpaceInvaders-v0

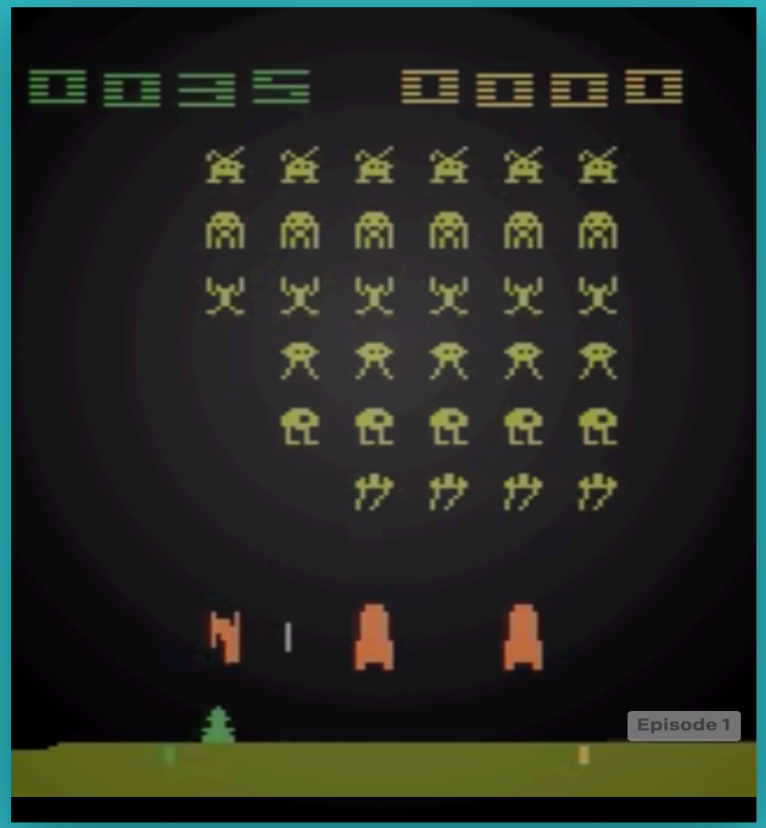
Maximize your score in the Atari 2600 game SpaceInvaders. In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3) Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from $\{2, 3, 4\}$.

The game is simulated through the Arcade Learning Environment [ALE], which uses the Stella [Stella] Atari emulator.

[ALE] *MG Bellemare, Y Naddaf, J Veness, and M Bowling. "The arcade learning environment: An evaluation platform for general agents." Journal of Artificial Intelligence Research (2012). https://github.com/mgbellemare/Arcade-Learning-Environment*

[Stella] *Stella: A Multi-Platform Atari 2600 VCS emulator https://stella-emu.github.io/*

VIEW SOURCE ON GITHUB



2. Open AI Gym

- Gym Customize

```
gym-foo/
  README.md
  setup.py
  gym_foo/
    __init__.py
    envs/
      __init__.py
      foo_env.py
```

gym-foo/setup.py

```
from setuptools import setup

setup(name='gym_foo',
      version='0.0.1',
      install_requires=['gym']
)
```

2. Open AI Gym

- Gym Customize

gym-foo/gym_foo/init.py

```
from gym.envs.registration import register

register(
    id='foo-v0',
    entry_point='gym_foo.envs:FooEnv',
)
```

gym-foo/gym_foo/envs/init.py

```
from gym_foo.envs.foo_env import FooEnv
```

2. Open AI Gym

- Gym Customize

gym-foo/gym_foo/envs/foo_env.py

```
import gym
from gym import error, spaces, utils
from gym.utils import seeding

class FooEnv(gym.Env):
  metadata = {'render.modes': ['human']}

  def __init__(self):
    pass
  def step(self, action):
    pass
  def reset(self):
    print("reset Function"
          "")
    pass
  def render(self, mode='human', close=False):
    pass
```

PEP 8: expected 1 blank line, found 0

>> cd gym-foo
>> pip install -e .
>> vi envs.py

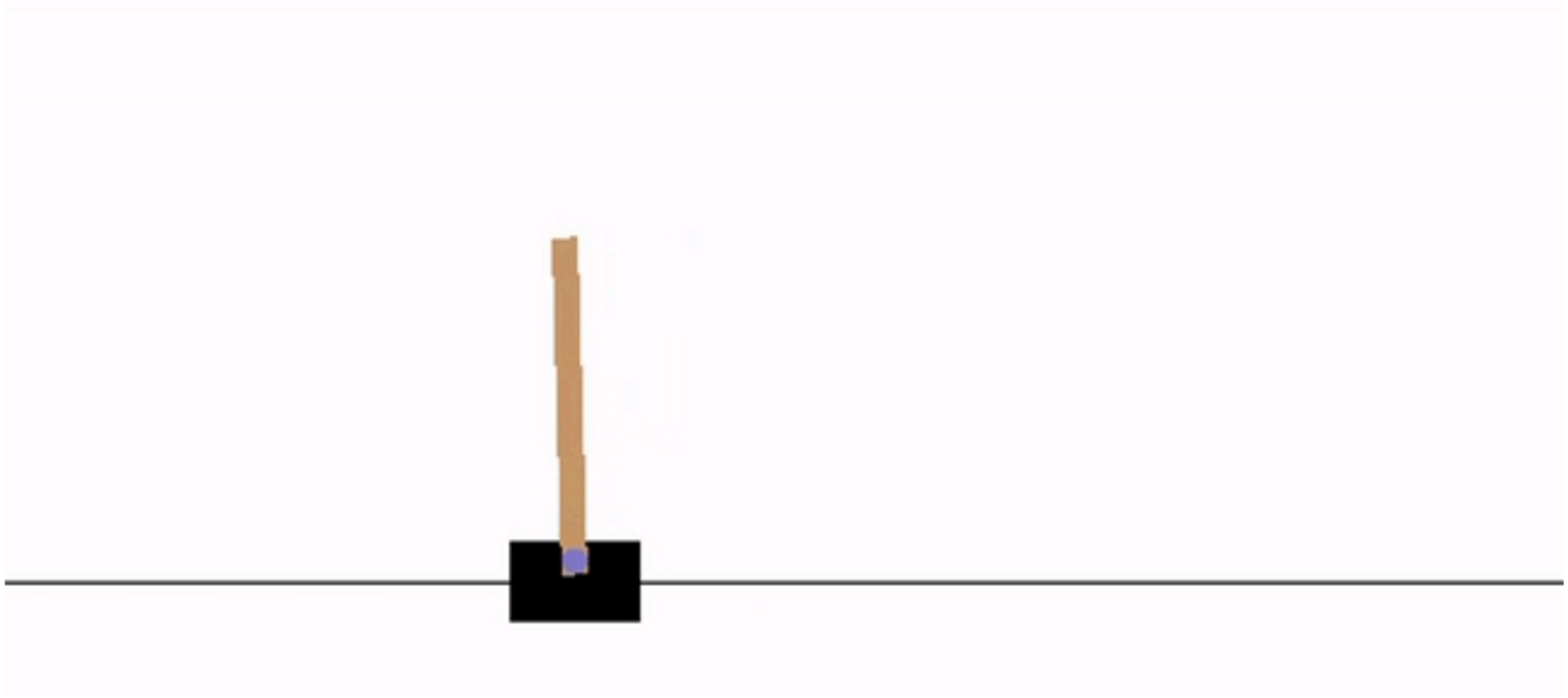
```
import gym
import gym_foo
env = gym.make('foo-v0')
env.reset()
```

```
Run:  envs
/Users/kimhyungrak/anaconda3/envs/ml_env/bin/python3.6 /Users/kimhyungrak/Desktop/dqn/gym-foo/envs.py
reset Function

Process finished with exit code 0
```

3. Practice(CartPole-v1)

- CartPole Game Problem



3. Practice(CartPole-v1)

- CartPole Game Problem

```
Observation:
    Type: Box(4)
    Num     Observation               Min         Max
    0       Cart Position             -4.8          4.8
    1       Cart Velocity             -Inf          Inf
    2       Pole Angle                 -24 deg       24 deg
    3       Pole Velocity At Tip      -Inf          Inf

Actions:
    Type: Discrete(2)
    Num     Action
    0       Push cart to the left
    1       Push cart to the right

    Note: The amount the velocity that is reduced or increased is not fixed; it depends on the angle the pole is pointing.

Reward:
    Reward is 1 for every step taken, including the termination step
```

3. Practice(CartPole-v1)

- CartPole Game Problem RL domain

3. Practice(CartPole-v1)

- Anaconda Install

1. https://wikidocs.net/2825
2. https://wikidocs.net/2826

3. Practice(CartPole-v1)

- Terminal command
  - conda create --name test_rl python=3.6
  - Window Case
    - activate test_rl
  - Linux & mac
    - source activate test_rl
  - pip3 install tensorflow
  - pip3 install numpy
  - Pip3 install jupyterlab
  - pip3 install gym
  - Pip3 install keras
  - Jupyter lab ←실행

3. Practice(CartPole-v1)

초기 라이브러리 및 함수 설정

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

ENV_NAME = "CartPole-v1" # Game 이름

GAMMA = 0.95 # Discount Factor
LEARNING_RATE = 0.001

MEMORY_SIZE = 1000000 #Replay memory최대 저장 개수
BATCH_SIZE = 20 #memory에 history data가 20개까지 있을 때

EXPLORATION_MAX = 1.0 #최대 exploration 값
EXPLORATION_MIN = 0.01 #최소 exploration 값
EXPLORATION_DECAY = 0.995 #exploration 값이 작아지게 하기 위해서
```

3. Practice(CartPole-v1)

Replay memory 저의

```
    def remember(self, state, action, reward, next_state, done):
        # agent의 state, action, reward, next_state, 게임이 끝났는지 등을 메모리 버퍼에 저장
        self.memory.append((state, action, reward, next_state, done))
```

3. Practice(CartPole-v1)

Action 함수 정의

```
    def act(self, state):
        #exploration_rate가 초반에는 1.0 이지만, 학습을 거급하면서 점점 작아진다.
        if np.random.rand() < self.exploration_rate:
            # Random으로 action을 선택
            return random.randrange(self.action_space)
        # agent가 직접 action을 선택
        q_values = self.model.predict(state)
        # 오른쪽, 왼쪽으로 예측한 값 중에 확률값이 높은것을 선택해서 return 한다
        return np.argmax(q_values[0])
```

3. Practice(CartPole-v1)

학습 Process 정의

```
def experience_replay(self):
    if len(self.memory) < BATCH_SIZE:
        return
    batch = random.sample(self.memory, BATCH_SIZE)# 버퍼에서 batch size만큼 rand
    for state, action, reward, state_next, terminal in batch:
        q_update = reward #현제  state와 action을 했을 때 reward
        if not terminal:# 이 Action을 했을 때 게임이 끝나지 않았다면
            q_update = (reward + GAMMA * np.amax(self.model.predict(state_next)[0])) # q_update -> reward를 계산
        q_values = self.model.predict(state)
        q_values[0][action] = q_update # 기존 action의 q_value값 다시 계산 : 게임이 끝났다면 기존 reward를 끝나지 않았다면 예측한 q_update 값
        self.model.fit(state, q_values, verbose=0) #model 학습
    self.exploration_rate *= EXPLORATION_DECAY
    self.exploration_rate = max(EXPLORATION_MIN, self.exploration_rate)#exploration_rate 줄여나감
```

3. Practice(CartPole-v1)

- 실습문제
  - DQN을 이용한 강화학습이 되도록 코드작성

Thank you