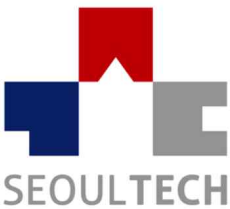
지능시스템소프트웨어 프로젝트 [설명문서]



학 과 : 컴퓨터공학과
학 번 : 16101399
이 름 : 장 우 영

목 차

1. 개요 및 목적

1.1 프로젝트 개요

여태까지 배운 다양한 알고리즘을 Open AI gym의 cart pole을 통해서 실험해보았습니다. DQN(Deep-Q-Network), DDQN(Double Deep-Q-Network), Duel DQN(Duel Deep-Q-Network) 총 3가지 알고리즘을 동일한 환경(Cart pole)에 적용해보고 성능을 측정했습니다.

	학습종료까지 걸린 에피소드
DQN	167
DDQN	191
Dueling DQN	114

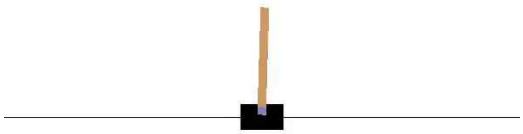
DQN에 비해, 별도 파라미터를 가지는 Target network를 구성하여 Q-value를 평가하고 학습하는 DDQN은 조금 더 학습하는 데에 시간이 필요했습니다. 반면, Value function stream과 Advantage function stream으로 각각 네트워크를 특화하여 학습하는 Dueling DQN은 114만에 학습을 완료했으며 DQN, DDQN보다 나은 성능을 확인할 수 있었습니다.

1.2 프로젝트 목적

1. OpenAI gym 강화학습 플랫폼을 사용해보고, 그 중 대표적인 환경인 'CartPole'환경을 분석해보고 적용해봅니다.
2. DQN, DDQN, Dueling DQN구조 및 방법론에 대해 이론적으로 이해해보고 강화학습 환경에 적용해봅니다.
3. DQN, DDQN, Dueling DQN구조 및 방법론에 대한 장단점을 살펴보고, 성능을 비교해봅니다.

2. 환경설정

2.1 Cart pole



Cart pole 예제

환경설명)

마찰이 없는 트랙 위, 검은색 카트(cart)에는 주황핵 폴(pole)가 하나 놓여져 있습니다. 폴과 카트의 연결부는 조작할 수 없으며, 오로지 카트(cart)만이 수평선을 따라 마찰없이 자유롭게 왔다갔다가 할 수 있습니다. 폴은 카트에 핀으로 연결돼 있는데, 이 핀을 축으로 자유롭게 회전할 수 있습니다.
에이전트는 이 카트에 왼쪽 혹은 오른쪽으로 일정한 힘을 가할 수 있습니다.

목표) 막대기가 넘어지지 않도록 유지하며, 최대한 오래 유지하기

종료조건)

1) 막대기가 수직으로부터 12도 이상 기울어짐
2) 카트가 중심으로부터 2.4 이상 벗어남
3) 에피소드의 시간 스텝이 200보다 커짐

승리조건)

1) 100번의 연속적인 시도에서 평균 195점 이상을 획득

Observation)

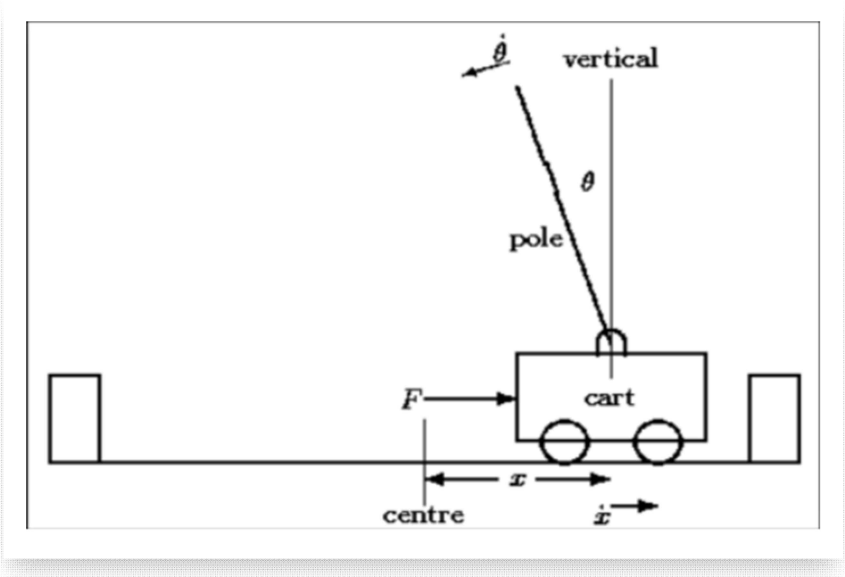
$(X, \dot{X}, \theta, \dot{\theta})$ **: 에이전트가 게임에서 이용할 수 있는 정보는 4가지입니다.**

X **: 카트의 수평선 상의 위치**

$\dot{X}$ **: 카트의 속도**

θ **: 폴이 수직선으로부터 기울어진 각도**

$\dot{\theta}$ **: 폴의 각속도**



Cart-Pole system

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418rad(-24deg)	0.418rad(24deg)
3	Pole Angular Velocity	-Inf	Inf

Cart-Pole Observation 속성

Num	Action
0	Push cart to the left
1	Push cart to the right

Cart pole Action 속성

단, 속도가 감소하거나 증가하는 양은 pole의 θ에 따라 달라지므로 고정되지 않습니다.
(∵ 기둥의 무게중심이 카트를 그 아래로 이동하는 데 필요한 에너지양에 영향을 주기 때문)

2.2 Cart pole 환경속성

- **Deterministic environment:** 현재 상태에서 어떤 행동을 했을 때, 다음 상태가 확정됩니다.
- **Fully Observable environment:** 강화학습을 진행하는 모든 시간동안, agent가 시스템의 상태를 관찰할 수 있습니다.
- **Continuous environment:** 실수 단위의 속도를 조정할 수 있어, 가능한 action의 수가 무한합니다.
- **Episodic environment:** 현재의 action이 차후 episode의 action에 영향을 미치지 않으며, 독립적입니다.
- **Single agent environment:** cartpole은 한 개로, 단일 agent로 구성된 환경입니다.

2.3 Cart pole 소스코드

1) import

```
import math
import gym
from gym import spaces, logger
from gym.utils import seeding
import numpy as np
```

2) Class CartPoleEnv

```
class CartPoleEnv(gym.Env):
    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': 50
    }

    def __init__(self):
    def seed(self, seed=None):
    def step(self, action):
    def reset(self):
    def render(self, mode='human'):
    def close(self):
```

gym.Env를 상속받아 사용하고 있습니다.
__init__(self): 생성자
seed(self, seed=None): 난수생성 seed 반환
step(self, action): 한 에피소드를 이루는 step를 구성, action에 따라 1step 진행
reset(self): 환경의 state 리셋
render(self, mode='human'): Cart pole 환경을 렌더링하여 시뮬레이션
close(self): viewer객체 반환(렌더링 종료)

CartPoleEnv클래스는 총 6가지의 메서드를 갖고 있으며, 아래에서 상세히 설명하겠습니다.

3) 생성자(init)

```
def __init__(self):
        self.gravity = 9.8
        self.masscart = 1.0
        self.masspole = 0.1
        self.total_mass = (self.masspole + self.masscart)
        self.length = 0.5  # actually half the pole's length
        self.polemass_length = (self.masspole * self.length)
        self.force_mag = 10.0
        self.tau = 0.02  # seconds between state updates
        self.kinematics_integrator = 'euler'
```

강화학습 환경에 적용되는 물리상수를 정의하고 있습니다.

중력 가속도 9.8, 카트의 무게: 1, 폴의 무게: 0.1, 카트폴의 무게 1.1, 폴의 길이: 0.5
Force of magnitude: 10, tau(토크): 0.02, 역학적분은 euler방식을 사용합니다.

```
        # Angle at which to fail the episode
        self.theta_threshold_radians = 12 * 2 * math.pi / 360
        self.x_threshold = 2.4
```

해당 threshold를 넘어서면 pole이 떨어진 것으로 간주합니다.
θ의 threshold는 12 * 2 * π/360 = 0.06π rad = 약 12도
X의 threshold는 2.4로 x의 최대위치를 넘어서면 떨어졌다고 간주합니다. (화면 밖으로 나감)

```
        # Angle limit set to 2 * theta_threshold_radians so failing observation
        # is still within bounds.
        high = np.array([self.x_threshold * 2,
                         np.finfo(np.float32).max,
                         self.theta_threshold_radians * 2,
                         np.finfo(np.float32).max],
                        dtype=np.float32)
```

높이를 구하는 행렬을 만듭니다. θ의 threshold의 2배가 되면 해당 observation은 떨어진다고 가정합니다.

```
        self.action_space = spaces.Discrete(2)
        self.observation_space = spaces.Box(-high, high, dtype=np.float32)

        self.seed()
        self.viewer = None
        self.state = None

        self.steps_beyond_done = None
```

총 2가지로 나뉘어진 action space를 정의합니다. action은 cart가 이동하는 방향으로 왼쪽과 오른쪽입니다. observation space는 pole의 높이를 기록합니다.

random seed를 설정하고, cart pole를 렌더링해주는 viewer와 해당 state 및 그 다음 step은 None으로 default 설정 해둡니다.

4) seed

```
    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]
```

seed메서드는 난수생성함수의 seed를 반환합니다.

5) step

```
    def step(self, action):
        err_msg = "%r (%s) invalid" % (action, type(action))
        assert self.action_space.contains(action), err_msg

        x, x_dot, theta, theta_dot = self.state
        force = self.force_mag if action == 1 else -self.force_mag
        costheta = math.cos(theta)
        sintheta = math.sin(theta)
```

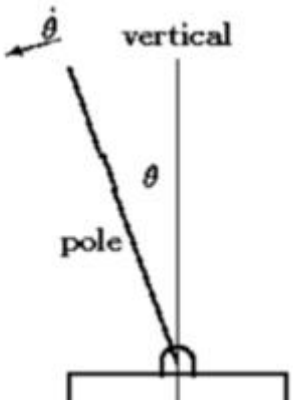
step함수에서는 파라미터로 action을 가지며, 인자로 받아온 action을 실행했을 때의 state와 reward를 반환합니다.

우선 assert함수를 통해서 인자로 들어온 action이 정당한지를 확인합니다.
cart pole환경에서는 action space가 0(왼쪽이동)과 1(오른쪽이동)로 이루어져있습니다.

먼저 $(X, \dot{X}, \theta, \dot{\theta})$ agent의 상태를 입력받고 action에 따라 Force of magnitude의 방향을 결정합니다. (오른쪽을 +로 벡터설정)
theta값에 따라, 미리 $\cos\theta$와 $\sin\theta$값도 구해둡니다.

```
        temp = (force + self.polemass_length * theta_dot ** 2 * sintheta) / self.total_mass
        thetaacc = (self.gravity * sintheta - costheta * temp) / (self.length * (4.0 / 3.0 - self.masspole * costheta ** 2 / self.total_mass))
        xacc = temp - self.polemass_length * thetaacc * costheta / self.total_mass
```

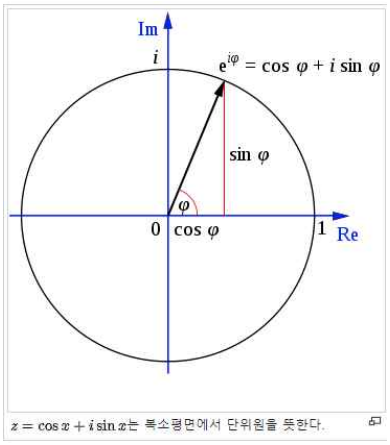
$\cos\theta$, $\sin\theta$를 통해서 pole각속도를 직각분해합니다.



$\dot{\theta}\cos\theta$는 수직방향(y축), $\dot{\theta}\sin\theta$는 수평방향(x축)으로 분해되며, 이를 통해 thetaacc(theta acceleration, 각속도)와 xacc(x acceleration, 수평방향의 cart의 가속도)를 구합니다. (∵ F = ma 운동방정식에 따라, a = F/m)

```
if self.kinematics_integrator == 'euler':
            x = x + self.tau * x_dot
            x_dot = x_dot + self.tau * xacc
            theta = theta + self.tau * theta_dot
            theta_dot = theta_dot + self.tau * thetaacc
        else:  # semi-implicit euler
            x_dot = x_dot + self.tau * xacc
            x = x + self.tau * x_dot
            theta_dot = theta_dot + self.tau * thetaacc
            theta = theta + self.tau * theta_dot

        self.state = (x, x_dot, theta, theta_dot)
```

kinematics integration방식은 euler(오일러)공식을 따릅니다.



$x = \cos x + i\sin x$는 복소평면에서 단위원을 뜻한다.

Euler`s Formula(오일러 공식)

agent의 상태를 나타내는 $(X, \dot{X}, \theta, \dot{\theta})$를 계산해냅니다.

```
        done = bool(
            x < -self.x_threshold
            or x > self.x_threshold
            or theta < -self.theta_threshold_radians
            or theta > self.theta_threshold_radians
        )
```

구한 x(카트의 위치)와 theta(pole의 각도)가 pole이 떨어진다고 간주하는 범위인지를 확인합니다. // 3) __init__ (생성자)로부터 각 threshold정의함 (에피소드 종료조건과 동일)

```
        if not done:
            reward = 1.0
        elif self.steps_beyond_done is None:
            # Pole just fell!
            self.steps_beyond_done = 0
            reward = 1.0
        else:
            if self.steps_beyond_done == 0:
                logger.warn(
                    "You are calling 'step()' even though this "
                    "environment has already returned done = True. You "
                    "should always call 'reset()' once you receive 'done = "
                    "True' -- any further steps are undefined behavior."
                )
            self.steps_beyond_done += 1
            reward = 0.0
```

만약 pole이 떨어지지 않는다면(에피소드가 끝나지 않는다면) reward 1를 부여합니다. (pole이 떨어지지 않았으므로) 만약 그 다음 단계에서 에피소드가 끝난다고(pole이 떨어진다고) 해도 reward 1은 부여하며

```
        return np.array(self.state), reward, done, {}
```

인자로 받아온 action을 했을 때, 변화한 state와 얻은 reward, 에피소드 종료 여부를 반환합니다.

6) reset

```
    def reset(self):
        self.state = self.np_random.uniform(low=-0.05, high=0.05, size=(4,))
        self.steps_beyond_done = None
        return np.array(self.state)
```

reset은 에피소드가 끝나고 state를 재설정하는 메서드입니다.
random함수는 uniform분포를 사용하여 난수설정합니다.

7) render

```
    def render(self, mode='human'):
        screen_width = 600
        screen_height = 400

        world_width = self.x_threshold * 2
        scale = screen_width/world_width
        carty = 100  # TOP OF CART
        polewidth = 10.0
        polelen = scale * (2 * self.length)
        cartwidth = 50.0
        cartheight = 30.0
```

render함수는 cart pole 에피소드를 렌더링하는 함수입니다.
cart pole 애니메이션을 위한 screen은 600 X 400으로 지정하고
가로의 길이는 카트 위치의 threshold의 2배만큼 정의합니다.
(카트의 중앙에서 양 옆으로 threshold만큼 거리를 둡니다.)

애니메이션에 등장하는 cart와 pole의 길이와 높이를 설정합니다.

```
        if self.viewer is None:
            from gym.envs.classic_control import rendering
            self.viewer = rendering.Viewer(screen_width, screen_height)
            l, r, t, b = -cartwidth / 2, cartwidth / 2, cartheight / 2, -cartheight / 2
            axleoffset = cartheight / 4.0
            cart = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            self.carttrans = rendering.Transform()
            cart.add_attr(self.carttrans)
            self.viewer.add_geom(cart)
            l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -polewidth / 2
            pole = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            pole.set_color(.8, .6, .4)
            self.poletrans = rendering.Transform(translation=(0, axleoffset))
            pole.add_attr(self.poletrans)
            pole.add_attr(self.carttrans)
            self.viewer.add_geom(pole)
            self.axle = rendering.make_circle(polewidth/2)
            self.axle.add_attr(self.poletrans)
            self.axle.add_attr(self.carttrans)
            self.axle.set_color(.5, .5, .8)
            self.viewer.add_geom(self.axle)
            self.track = rendering.Line((0, carty), (screen_width, carty))
            self.track.set_color(0, 0, 0)
            self.viewer.add_geom(self.track)

            self._pole_geom = pole

        if self.state is None:
            return None

        # Edit the pole polygon vertex
        pole = self._pole_geom
        l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -polewidth / 2
        pole.v = [(l, b), (l, t), (r, t), (r, b)]

        x = self.state
        cartx = x[0] * scale + screen_width / 2.0  # MIDDLE OF CART
        self.carttrans.set_translation(cartx, carty)
        self.poletrans.set_rotation(-x[2])

        return self.viewer.render(return_rgb_array=mode == 'rgb_array')
```

부모클래스인 gym.Env로부터 필요한 렌더링 함수를 받아옵니다.
앞서 정의한 screen 및 cart, pole의 설정을 토대로 렌더링을 마칩니다.

8) close

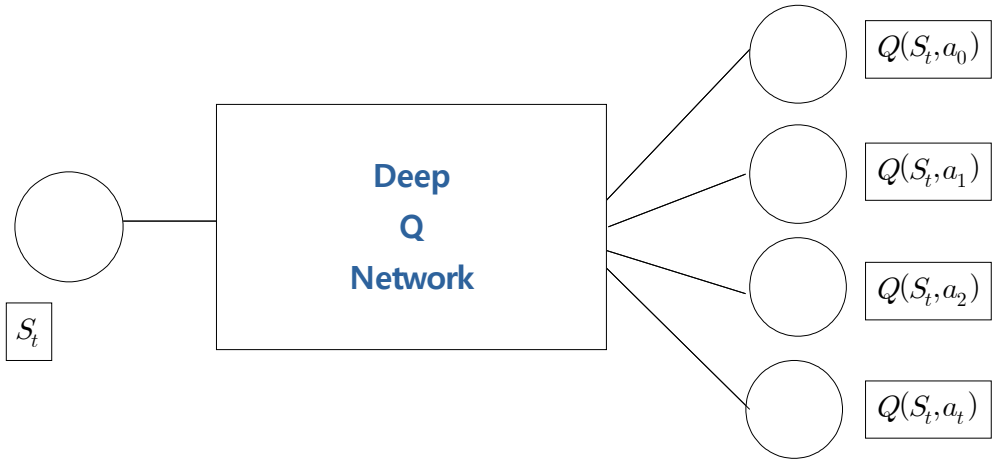
```
def close(self):
    if self.viewer:
        self.viewer.close()
        self.viewer = None
```

렌더링 객체가 열려있다면, close 메서드를 통해서 닫아줍니다. (리소스 반환합니다.)

3. 적용한 강화학습 알고리즘

3.1 DQN

3.1.1 이론적 설명



DQN방식은 Q function을 네트워크의 파라미터 θ로 근사하여 구하는 접근방식입니다. $Q(s,a;\theta) \approx Q^*(s,a)$ 근사하여 각 상태에서 가능한 모든 action의 Q-value를 계산합니다. Q-value를 계산하기위해서 Deep Q network를 구축하며, input layer는 구하고자하는 state가 입력되고 output layer는 해당 state에서 가능한 모든 action의 Q-value를 출력합니다. (input layer와 output layer는 FC layer로 구성합니다.)

정확한 Q value를 구하기 위해서, Deep Q Network도 파리미터를 학습하며 loss함수는 다음과 같이 정의됩니다.

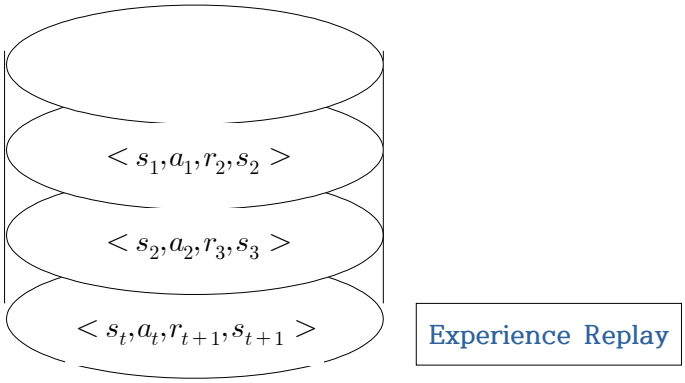
$$Loss = (r + \gamma \max_{a'} Q(s',a';\theta') - Q(s,a;\theta))^2$$

Loss값은 Target value와 Predicted value의 차의 제곱형태로 나타납니다.

- Target value: $r + \gamma \max_{a'} Q(s',a';\theta')$
- predicted value: $Q(s,a;\theta)$

Target value를 구하는 네트워크와 predicted value를 구하는 네트워크가 동일하면, 둘 사이의 발산할 위험이 크기 때문에 별도 네트워크를 사용합니다. Target value의 Q value는 별도의 Target network를 사용하며, 특정 주기마다 실제 Q network의 가중치를 복사하여 업데이트합니다.

한편 Q network는 경사하강법(gradient descent)를 사용하여 최적 해를 찾기 위해 가중치를 업데이트합니다.



DQN네트워크를 학습시키는 데이터는 Experience Replay에 저장된 데이터를 사용합니다. Cart-pole의 Transition을 그대로 네트워크에 학습하면, 시간축에 따른 correlation이 상당히 강하기 때문에 학습이 제대로 진행되지 않습니다. (특정 순간, 일련의 그러한 상황에서만 제대로 동작해서 일반화가 되지 않음)

따라서, 시간축에 따른 correlation을 줄이고자 Experience Replay에 transition정보를 저장하고 랜덤으로 샘플링해서 학습에 사용합니다. Experience Replay는 Queue 자료구조와 유사하며, 버퍼가 가득차면 오래된 에피소드 정보를 지우고 최근의 에피소드 정보로 최신화합니다.

<전반적인 DQN 알고리즘 과정>

1. Q-value를 구하고자하는 state를 DQN에 입력값으로 넣습니다.
(Cart-pole환경에서는 아타리게임환경과 같이 game screen을 state로 하지 않기 때문에 CNN과 같은 전처리 작업은 필요하지 않습니다.)
2. ϵ-greedy정책에 따라 action을 선택합니다. $a = \arg max(Q(s,a;\theta))$
3. action a를 선택한 뒤, 상태 s에서 수행하고 새로운 상태 s'로 transition된 reward를 받습니다.
4. 해당 transition을 replay buffer에 저장합니다. $< s,a,r,s' >$
5. replay buffer에서 임의의 transition 배치를 샘플링해서 loss를 계산합니다.
$Loss = (r + \gamma \max_{a'} Q(s',a';\theta') - Q(s,a;\theta))^2$
6. loss를 최소화하기 위해서 actual Q network의 파라미터에 대해서 경사하강법을 시행합니다.
7. 특정 주기마다, actual Q network의 파라미터를 target network의 파라미터에 복사합니다.
8. 이를 M개의 에피소드에 대해서 반복합니다.

3.1.2 구현설명

```
# import
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import gym
```

구현에 필요한 라이브러리를 import합니다.

```
#name tuple을 사용하여 Transition 정의
from collections import namedtuple

Transition = namedtuple(
    'Transition', ('state', 'action', 'next_state', 'reward'))
```

Transition를 저장하고자, namedtuple구조를 이용합니다.
총 4가지를 저장하며, 해당 state, action과 다음 state, reward를 저장합니다.

이는 나중에 replay Memory를 구성할 때에도 사용합니다.

```
# 상수 정의
ENV = 'CartPole-v0'  # task
GAMMA = 0.99  # discount factor
MAX_STEPS = 200  # 1에피소드 당 최대 단계 수
NUM_EPISODES = 500  # 최대 에피소드 수
```

CartPole-v0환경을 이용하며, discount factor는 0.99를 사용했습니다.
1에 가까운 높은 숫자로, 이는 미래의 reward에 비중을 두고 학습했습니다.
(∵ 오랫동안 pole이 떨어지지 않고 유지하는 게, 해당 환경의 목표)

한 에피소드당 최대 단계 수는 200으로, 200 에피소드가 넘어가면 종료됩니다.
(∵ Cart-pole 환경의 종료조건)

```
class ReplayMemory:

    def __init__(self, CAPACITY):
        self.capacity = CAPACITY  # replay Memory의 최대 저장 건수
        self.memory = []  # 실제 transition을 저장할 변수
        self.index = 0  # 저장 위치를 가리킬 index 변수

    def push(self, state, action, state_next, reward):

        if len(self.memory) < self.capacity:
            self.memory.append(None)  # 메모리가 가득차지 않은 경우

        # Transition이라는 namedtuple을 사용하여 키-값 쌍의 형태로 값을 저장
        self.memory[self.index] = Transition(state, action, state_next, reward)

        self.index = (self.index + 1) % self.capacity  # 다음 저장할 위치를 한 자리 뒤
로 수정

    def sample(self, batch_size):
        '''batch_size 갯수 만큼 무작위로 저장된 transition을 추출'''
        return random.sample(self.memory, batch_size)

    def __len__(self):
        '''len 함수로 현재 저장된 transition 갯수를 반환'''
        return len(self.memory)
```

'Transition'의 4가지 ('state', 'action', 'next_state', 'reward')를 저장하는 ReplayMemory구조를 정의합니다. 생성자를 통해서, replay memory로 저장할 최대 건수를 지정하고, 저장 및 불러올 때 사용할 memory와 index변수를 생성합니다.

ReplayMemory는 Queue구조를 띄고 있으며, 최근 transition을 저장하기 위한 push함수를 정의합니다. 메모리가 가득차지 않았을 때에는 append를 이용해서 뒤에 추가해줍니다.
Transition은 앞서 언급했던 namedtuple구조를 이용해서 키-값의 쌍 형태로 저장합니다.

sample메서드는 ReplayMemory에서 transition을 랜덤샘플링할 때, 사용합니다.
len메서드는 일반적인 자료구조에서 사용하는 것과 같이, 자료구조가 가지고 있는 데이터의 량, 즉 저장된 Transition의 개수를 반환합니다.

```
import random
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F

BATCH_SIZE = 32
CAPACITY = 10000
```

replay memory구조설계를 마치고, 그 다음은 DQN의 네트워크를 설계합니다. 네트워크 설계에는 pytorch를 이용했습니다.

```
class Brain:
    def __init__(self, num_states, num_actions):
        self.num_actions = num_actions  # 행동의 가짓수(왼쪽, 오른쪽)를 구함

        # transition을 기억하기 위한 메모리 객체 생성
        self.memory = ReplayMemory(CAPACITY)

        # 신경망 구성
        self.model = nn.Sequential()
        self.model.add_module('fc1', nn.Linear(num_states, 32))
        self.model.add_module('relu1', nn.ReLU())
        self.model.add_module('fc2', nn.Linear(32, 32))
        self.model.add_module('relu2', nn.ReLU())
        self.model.add_module('fc3', nn.Linear(32, num_actions))

        print(self.model)  # 신경망 구조 출력

        # 최적화 기법 선택
        self.optimizer = optim.Adam(self.model.parameters(), lr=0.0001)
```

신경망을 구성합니다. 생성자는 state와 action의 수를 input으로 받아서 초기 네트워크 구조를 구성합니다.

input layer의 입력값은 replayMemory에서 랜덤샘플링한 transition을 받아오며, 이를 기억하기 위한 메모리 객체 self.memory를 생성합니다.

신경망은 입력layer(FC) -> ReLU -> FC -> ReLU -> 출력layer(FC)로 구성했습니다.

```
    def replay(self):
        if len(self.memory) < BATCH_SIZE:
            return
        transitions = self.memory.sample(BATCH_SIZE)
        batch = Transition(*zip(*transitions))

        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)
        non_final_next_states = torch.cat([s for s in batch.next_state
                                           if s is not None])

        self.model.eval()
        state_action_values = self.model(state_batch).gather(1, action_batch)
        non_final_mask = torch.ByteTensor(tuple(map(lambda s: s is not None,
                                                    batch.next_state)))
        next_state_values = torch.zeros(BATCH_SIZE)
        next_state_values[non_final_mask] = self.model(
            non_final_next_states).max(1)[0].detach()
        expected_state_action_values = reward_batch + GAMMA * next_state_values
        self.model.train()
        loss = F.smooth_l1_loss(state_action_values,
                                expected_state_action_values.unsqueeze(1))
        self.optimizer.zero_grad()  # 경사를 초기화
        loss.backward()  # 역전파 계산
        self.optimizer.step()  # 결합 가중치 수정
```

Experience Replay로 신경망 학습을 하기위한 메서드를 정의합니다.
먼저, 저장된 transition의 수를 확인하고, 미니배치크기보다 작으면 pass하며 클 경우에 학습이 진행됩니다. 미니 배치는 메모리 객체로부터 추출하며, 미니 배치크기만큼 만들어줍니다.
(state*BATCH_SIZE, action*BATCH_SIZE, state_next*BATCH_SIZE, reward*BATCH_SIZE)

이후, state_action value를 추론하고 어떤 action이 좋을지 평가합니다. (왼쪽으로 이동, 오른쪽으로 이동) 여기서 정답신호로 사용할 Q(s_t, a_t)는 Q-learning으로 계산합니다.

expected_state_action_values = $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)$

해당 방식으로 loss를 구하고, 역전파 계산 수정하며 학습합니다.

```
    def decide_action(self, state, episode):
        # ε-greedy 알고리즘
        epsilon = 0.5 * (1 / (episode + 1))

        if epsilon <= np.random.uniform(0, 1):
            self.model.eval()
            with torch.no_grad():
                action = self.model(state).max(1)[1].view(1, 1)
        else:
            action = torch.LongTensor(
                [[random.randrange(self.num_actions)]])
        return action
```

현재 state에 따라, action을 결정하는 메서드를 작성합니다.
ε-greedy 방식으로 exploration하며, 서서히 ε를 낮춰서 exploration을 줄이고 exploitation (마치 learning rate decay와 같이, 처음에는 ε를 높게 설정했다가 서서히 줄여갑니다.)

```
class Agent:
    def __init__(self, num_states, num_actions):
        self.brain = Brain(num_states, num_actions)

    def update_q_function(self):
        self.brain.replay()

    def get_action(self, state, episode):
        action = self.brain.decide_action(state, episode)
        return action

    def memorize(self, state, action, state_next, reward):
        self.brain.memory.push(state, action, state_next, reward)
```

agent, 즉 CartPole 클래스를 정의합니다. 앞서 정의한 신경망을 통해서 학습하며, replay를 이용해서 학습데이터를 추출하고 q_function을 업데이트합니다.
ε-greedy방식에 따라 행동을 결정하며, 행동을 통해 알게 된 다음 state와 reward는 transition구조로 저장합니다.

```
class Environment:

    def __init__(self):
        self.env = gym.make(ENV)
        num_states = self.env.observation_space.shape[0]
        num_actions = self.env.action_space.n
        self.agent = Agent(num_states, num_actions)
```

다음은 CartPole을 실행하는 환경을 class로 정의합니다.
환경은 앞서 2.1절에서 설명한 'CartPole v0'을 사용하며, 해당 환경으로부터 observation space(4)와 action space(2)를 받아옵니다.

```
    def run(self):
        episode_10_list = np.zeros(10)
        complete_episodes = 0  # 현재까지 195단계를 버틴 에피소드 수
        episode_final = False  # 마지막 에피소드 여부
        frames = []

        for episode in range(NUM_EPISODES):  # 최대 에피소드 수만큼 반복
            observation = self.env.reset()  # 환경 초기화

            state = observation
            state = torch.from_numpy(state).type(
                torch.FloatTensor)
            state = torch.unsqueeze(state, 0)
```

환경을 실행하는 메서드를 정의합니다.
승리조건은 평균 100번의 에피소드에서 195점 이상 획득이므로, 현재까지 195단계를 버틴 에피소드를 카운팅하는 변수를 설정합니다.

첫 번째 for문은 episode 수만큼 반복하며, (each episode)
매 에피소드마다 환경을 초기화하고 state를 재설정합니다.

```
            for step in range(MAX_STEPS):
                if episode_final is True:
                    frames.append(self.env.render(mode='rgb_array'))

                action = self.agent.get_action(state, episode)  # 다음 행동을 결정
                observation_next, _, done, _ = self.env.step(
                    action.item())  # reward와 info는 사용하지 않으므로 _로 처리
                if done:
                    state_next = None

                    episode_10_list = np.hstack(
                        (episode_10_list[1:], step + 1))

                    if step < 195:
                        reward = torch.FloatTensor(
                            [-1.0])
                        complete_episodes = 0
                    else:
                        reward = torch.FloatTensor([1.0])
                        complete_episodes = complete_episodes + 1
                else:
                    reward = torch.FloatTensor([0.0])
                    state_next = observation_next
                    state_next = torch.from_numpy(state_next).type(
                        torch.FloatTensor)
                    state_next = torch.unsqueeze(state_next, 0)

                self.agent.memorize(state, action, state_next, reward)
                self.agent.update_q_function()
                state = state_next
                if done:
                    print('%d Episode: Finished after %d steps : 최근 10 에피소드의 평균 단계
수 = %.1lf' % (
                        episode, step + 1, episode_10_list.mean()))
                    break
```

에피소드의 한 step마다 다음의 작업들을 반복하며 하나의 에피소드를 완성합니다. 앞서 agent class에서 정의한 get_action() 메서드를 통해서 Q_function값에 따라 행동을 선택하고 다음 state를 done 플래그 값을 결정합니다.

episode의 종료조건은 200단계가 넘어가거나 pole의 각도가 12도 이상 휘어짐입니다. if문을 통해서 이를 판정합니다. pole이 서 있는 상태로 무사히 200단계를 넘겼다면 성공한 에피소드로 종료합니다.(reward +1) 반면 봉이 도중에 쓰러졌다면 –1 reward를 부여하며 연속으로 성공한 에피소드 기록을 초기화합니다.

```
if episode_final is True:
                # 애니메이션 생성 및 저장
                display_frame(frames)
                break

            if complete_episodes >= 10:
                print('----- 10 episode success -----')
                episode_final = True
```

연속으로 10번 이상 성공한 에피소드를 check하고 애니메이션으로 생성해서 저장합니다.

3.2 DDQN

3.2.1 이론적 설명

앞서 3.1의 DQN방식은 Q-learning에서 max연산자를 사용해서 Q-value를 과대평가하는 경향이 있었습니다. 따라서 Q-value를 추정하는 과정에서 조금의 오차(noise)가 있으면 단순히 max연산자로 조금이라도 큰 것을 고르는 한계점이 있었습니다. (실제로는 다른 action이 최적) 이러한 문제를 해결하고자 DDQN에서는 각각의 독립적인 Q-function으로 추정합니다.

독립적인 두 Q-Function 중 하나는 action을 선택하고, 다른 하나는 action을 평가하여 좀 더 효과적으로 학습합니다.

$$a_m = \arg max_a Q_m(s_{t+1}, a)$$
$$Q_m(s_t, a_t) = Q_m(s_t, a_t) + \eta(r_{t+1} + \gamma Q_t(s_{t+1}, a_m) - Q_m(s_t, a_t))$$

즉, 다음 상태s_{t+1}에서 Q-value가 최대가 되는 행동 a_m은 Actual Q-network에서 구하며, 그 때의 Q값은 Target Q-network에서 구합니다. DDQN은 Target network의 파라미터를 독립적으로 업데이트하고 action을 평가하므로 DQN의 한계점을 보완할 수 있었습니다.

3.2.2 구현설명

```
# 신경망 구성
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self, n_in, n_mid, n_out):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(n_in, n_mid)
        self.fc2 = nn.Linear(n_mid, n_mid)
        self.fc3 = nn.Linear(n_mid, n_out)

    def forward(self, x):
        h1 = F.relu(self.fc1(x))
        h2 = F.relu(self.fc2(h1))
        output = self.fc3(h2)
        return output
```

3.1.2 DQN 구현과 동일하게 필요한 라이브러리를 import하고 상수를 정의합니다. 다만 Q-network를 업데이트 과정이 다릅니다. 해당 과정을 자세히 살펴보겠습니다.

DQN과 달리 DDQN은 Actual Q network와 Target Q network로 2개를 사용했습니다. 따라서 추가된 Target Q network를 구성합니다.

```
import random
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F

BATCH_SIZE = 32
CAPACITY = 10000

class Brain:
    def __init__(self, num_states, num_actions):
        self.num_actions = num_actions

        self.memory = ReplayMemory(CAPACITY)

        # 신경망 구성
        n_in, n_mid, n_out = num_states, 32, num_actions
        self.main_q_network = Net(n_in, n_mid, n_out)
        self.target_q_network = Net(n_in, n_mid, n_out)
        print(self.main_q_network)

        self.optimizer = optim.Adam(
            self.main_q_network.parameters(), lr=0.0001)
```

transition을 기억하기 위한 메모리 객체(self.memory)를 생성하고, network를 구성합니다. 앞서 정의했던 Net class를 통해서 Actual Q network와 Target Q network를 설계합니다. 최적화는 Adam optimizer를 사용합니다.

이후에는 3.1.2의 DQN과 마찬가지로 네트워크 학습을 위한 Transition 데이터를 Experience Replay에 저장하고 이를 랜덤샘플링하는 과정을 그대로 진행합니다. 마찬가지로 샘플링한 transition 데이터는 mini batch로 학습하며 batch크기보다 작은 샘플링은 pass합니다.

```
def get_expected_state_action_values(self):
        self.main_q_network.eval()
        self.target_q_network.eval()
        self.state_action_values = self.main_q_network(
            self.state_batch).gather(1, self.action_batch)

        non_final_mask = torch.ByteTensor(tuple(map(lambda s: s is not None,
                                                    self.batch.next_state)))
        next_state_values = torch.zeros(BATCH_SIZE)

        a_m = torch.zeros(BATCH_SIZE).type(torch.LongTensor)
        a_m[non_final_mask] = self.main_q_network(
            self.non_final_next_states).detach().max(1)[1]
        a_m_non_final_next_states = a_m[non_final_mask].view(-1, 1)

        next_state_values[non_final_mask] = self.target_q_network(
            self.non_final_next_states).gather(1,
a_m_non_final_next_states).detach().squeeze()

        expected_state_action_values = self.reward_batch + GAMMA *
next_state_values

        return expected_state_action_values
```

정답값으로 사용할 Q_value를 계산합니다. actual Q network는 경사하강법에 따라 학습한 것을 토대로, 최적의 action을 선택하고 target Q network는 actual Q network와 독립적인 네트워크로 선택한 action을 평가합니다.

$$a_m = \arg max_a Q_m(s_{t+1}, a)$$
$$Q_m(s_t, a_t) = Q_m(s_t, a_t) + \eta(r_{t+1} + \gamma Q_t(s_{t+1}, a_m) - Q_m(s_t, a_t))$$

```
    def update_main_q_network(self):

        self.main_q_network.train()

        loss = F.smooth_l1_loss(self.state_action_values,
                                self.expected_state_action_values.unsqueeze(1))

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target_q_network(self):
        self.target_q_network.load_state_dict(self.main_q_network.state_dict())
```

loss값을 계산하고, 경사하강법에 따라 actual Q network를 학습합니다.
가중치를 계산하고, 오차역전파를 통해서 가중치를 업데이트합니다.

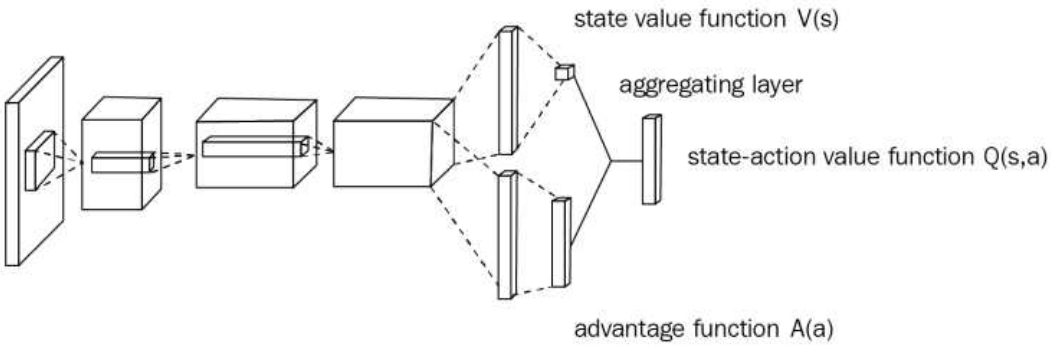
```
class Agent:
    def __init__(self, num_states, num_actions):
        self.brain = Brain(num_states, num_actions)

    def update_q_function(self):
        self.brain.replay()

    def get_action(self, state, episode):
        action = self.brain.decide_action(state, episode)
        return action

    def memorize(self, state, action, state_next, reward):
        self.brain.memory.push(state, action, state_next, reward)

    def update_target_q_function(self):
        self.brain.update_target_q_network()
```

DQN에 비교하여 update_target_q_function 새로 구현되었으며, Target Q network도 Actual Q network와 같이 독립적인 네트워크로 업데이트됩니다.

3.3 Dueling DQN

3.3.1 이론적 설명

❑ Dueling DQN



앞서 3.2에서 살펴본 Double DQN은 네트워크를 독립적으로 2개를 구성해서 Actual Q network에서 최적의 action을 찾고, Target Q network에서 평가했었습니다. 반면 Dueling DQN은 Q-function을 state value function V(s)와 advantage function A(a)로 나누어서 구한다음, 합쳐서 Q-value를 구하는 방식입니다.

$$Q(s,a) = V(s) + A(a)$$

여기서 advantage function A(a)는 해당 state에서 가능한 action 중에서 해당 action이 얼마나 좋은지를 평가하며, Q(s,a)는 V(s)와의 합으로 구할 수 있습니다.
(value function V(s)는 해당 state가 얼마나 좋은지 평가. 즉, 가능한 action들의 reward평균으로 볼 수 있음)

Dueling DQN구조에서는 state value를 구하는 value function stream과 advantage value를 구하는 advantage function stream이 이중으로(Duel) 존재하며, 각각 목적에 맞게 특화되어 학습할 수 있다는 장점이 있습니다. 분업의 원리와 마찬가지로, 각각 V(s)와 A(a)를 구하여 합쳐 더 정확하고 효율적으로 Q(s,a)를 추정할 수 있습니다.

3.3.2 구현설명

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self, n_in, n_mid, n_out):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(n_in, n_mid)
        self.fc2 = nn.Linear(n_mid, n_mid)

        # Dueling Network
        self.fc3_adv = nn.Linear(n_mid, n_out)
        self.fc3_v = nn.Linear(n_mid, 1)

    def forward(self, x):
        h1 = F.relu(self.fc1(x))
        h2 = F.relu(self.fc2(h1))

        adv = self.fc3_adv(h2)
        val = self.fc3_v(h2).expand(-1, adv.size(1))

        output = val + adv - adv.mean(1, keepdim=True).expand(-1, adv.size(1))
        return output
```

3.2.2에서 구현했던 Double DQN과 마찬가지로 필요한 라이브러리를 import하고 상수설정하고, Experience replay구조를 설계하고, 미니배치를 이용하여 랜덤샘플링 하여 학습합니다.

다만, 다른 점은 네트워크 구조로 위와 같이 Duel로 구성합니다. value function을 계산하는 stream과 advantage function을 계산하는 stream으로, 각각 신경망을 구성합니다.
input layer인 self.fc1과 은닉층인 self.fc2을 거쳐 학습하고, fc3에서 각각 stream으로 나뉘어져서 학습하고 계산합니다.

$$Q(s,a) = V(s) + A(a)$$

그리고 각각 구한 값을 합쳐서 Q-value를 계산하고 output layer로 산출합니다. 여

기서 V(s)는 입력으로 받은 1개의 state값만 계산하지만, A(a)는 입력 state에서 가능한 모든 action 값을 계산하기 때문에 크기가 다릅니다. 따라서 val 계산 시 expand로 가능한 action 수만큼 확장하여 차원을 같게 해준 뒤 계산하는 작업이 추가되었습니다.

4. 최종결과

4.1 학습결과

4.1.1 DQN

6. Cartpole DQN 학습

```
In [14]: cartpole_env = Environment()
         cartpole_env.run()
```

```
Sequential(
  (fc1): Linear(in_features=4, out_features=32, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=32, out_features=32, bias=True)
  (relu2): ReLU()
  (fc3): Linear(in_features=32, out_features=2, bias=True)
)
0 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 0.9
1 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 1.9
2 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 2.8
3 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 3.8
4 Episode: Finished after 14 steps : 최근 10 에피소드의 평균 단계 수 = 5.2
5 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 6.1
6 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 7.0
7 Episode: Finished after 20 steps : 최근 10 에피소드의 평균 단계 수 = 9.0
8 Episode: Finished after 26 steps : 최근 10 에피소드의 평균 단계 수 = 11.6
9 Episode: Finished after 16 steps : 최근 10 에피소드의 평균 단계 수 = 13.2
```

3.1.2에서 구현한 DQN네트워크를 토대로 학습을 진행합니다. 에피소드를 반복하면서 학습을 진행할수록 성공한 단계 수가 증가하는 것을 확인할 수 있습니다.

```
100 Episode: Finished after 101 steps : 최근 10 에피소드의 평균 단계 수 = 113.8
101 Episode: Finished after 80 steps : 최근 10 에피소드의 평균 단계 수 = 110.5
```

100번째 에피소드가 지나자, 평균 113 step까지 pole이 떨어지지 않고 에피소드가 유지된 것을 확인할 수 있었습니다.

```
167 Episode: Finished after 200 steps : 최근 10 에피소드의 평균 단계 수 = 200.0
```

최종학습은 167에피소드에서 마쳤습니다. 최근 10개의 에피소드 모두 200step을 넘어서 종료되었으며 학습이 잘 진행된 것을 확인할 수 있었습니다.

4.1.2 DDQN

6. Cartpole DDQN 학습

```
cartpole_env = Environment()
cartpole_env.run()
```

```
Net(
  (fc1): Linear(in_features=4, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=32, bias=True)
  (fc3): Linear(in_features=32, out_features=2, bias=True)
)
0 Episode: Finished after 40 steps : 최근 10 에피소드의 평균 단계 수 = 4.0
1 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 4.9
2 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 5.8
3 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 6.8
4 Episode: Finished after 12 steps : 최근 10 에피소드의 평균 단계 수 = 8.0
5 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 8.9
6 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 9.8
7 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 10.7
```

3.2.2에서 구현한 DDQN네트워크를 통해 학습을 진행했습니다. DQN에 비해서 조금 느린 폭으로 학습되긴 하지만, 점차 단계 수가 증가하고 있습니다.

```
191 Episode: Finished after 200 steps : 최근 10 에피소드의 평균 단계 수 = 200.0
```

최종 학습은 191 에피소드에서 마쳤습니다. DQN이 167 에피소드에서 학습을 마친 것에 비해 느리긴 하지만 학습이 잘 진행된 것을 확인할 수 있었습니다.

4.1.3 Dueling DQN

6. Cartpole Dueling DQN 학습 ¶

```
cartpole_env = Environment()
cartpole_env.run()
```

```
Net(
  (fc1): Linear(in_features=4, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=32, bias=True)
  (fc3_adv): Linear(in_features=32, out_features=2, bias=True)
  (fc3_v): Linear(in_features=32, out_features=1, bias=True)
)
0 Episode: Finished after 15 steps : 최근 10 에피소드의 평균 단계 수 = 1.5
1 Episode: Finished after 13 steps : 최근 10 에피소드의 평균 단계 수 = 2.8
2 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 3.8
3 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 4.8
4 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 5.8
5 Episode: Finished after 10 steps : 최근 10 에피소드의 평균 단계 수 = 6.8
6 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 7.7
7 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 8.6
8 Episode: Finished after 9 steps : 최근 10 에피소드의 평균 단계 수 = 9.5
```

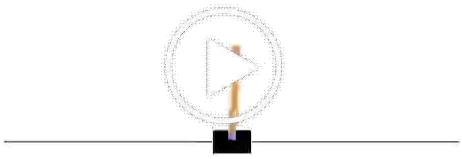
3.3.2에서 구현한 Dueling DQN신경망을 이용해서 학습을 진행합니다. 학습 초반에는 DQN이나 DDQN보다 학습이 조금 느린 것을 확인할 수 있습니다.

```
114 Episode: Finished after 188 steps : 최근 10 에피소드의 평균 단계 수 = 198.8
```

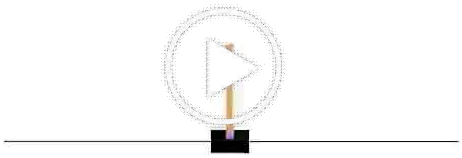
하지만, 최종학습은 114에피소드에서 마쳤습니다. DQN(167에피소드), DDQN(191에피소드)에 비교해서 상당히 빠른 속도로 학습이 되었으며, 성능이 좋다는 것을 확인할 수 있었습니다.

4.2 실행결과

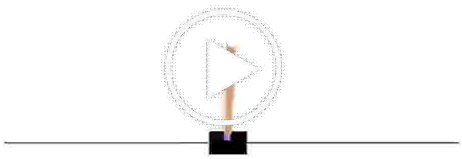
4.2.1 DQN



4.2.2 DDQN



4.2.3 Dueling DQN



DQN, DDQN, Dueling DQN 모두, 학습이 잘 진행된 것을 확인할 수 있었습니다.

5. 참고문헌

논문)

- AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983
- Savinay Nagendra, Nikhil Podila, Rashmi Ugarakhod, Koshy George, "Comparison of Reinforcement Learning Algorithms applied to the Cart-Pole Problem", IEEE, 2017

서적)

- Sudharsan Ravichandiran(2019), 『강화학습 입문: 파이썬 예제와 함께하는 openAI Gym과 TensorFlow 실습 가이드』(김승현 외 3명 역), 홍릉과학출판사
- 이웅원, 양혁렬, 김건우, 이영무, 이의령(2017), 『파이썬과 케라스로 배우는 강화학습』, 위키북스
- 오가와 유타로(2018), 『PyTorch를 활용한 강화학습/심층강화학습 실전 입문』(심효섭 역), 위키북스

웹페이지)

- OpenAI 위키 CartPole v0 https://github.com/openai/gym/wiki/CartPole-v0