3. 강화학습 Python 코드 작성

1) Conda 가상환경 생성과 라이브러리 다운로드

가상환경을 만드는 이유는 여러가지 이지만, 이 프로젝트에서는 파이썬 패키지의 버전을 관리하기 위해서이다.

> - 예를 들어, 프로젝트 A에서는 python2의 코드를 사용하는데, B에서는 python3을 사용하면 코드가 서로 호환이 안 되므로, 각각의 가상환경을 만들어서 다른 버전의 python을 다운로드하면 된다.

우선 아나콘다를 다음 사이트에서 다운로드한다.

https://www.anaconda.com/

> - 만약 conda 명령어가 작동하지 않는다면 환경변수를 설정을 해야한다.

conda 가상환경 생성을 위해 다음과 같은 명령어를 실행한다.

```
# env_name에 쓰고싶은 가상환경 이름을 적는다.
# conda create로 가상환경을 생성한다.
conda create -n env_name python="3.9.7"

# conda activate로 가상환경을 활성화한다.
conda activate env_name

# pip install로 파이썬 패키지를 다운로드한다.
pip install numpy matplotlib matplotlib-inline torch torchvision gym

# 만약 운영체제가 윈도우가 아니라면
pip install pygame
```

> 저자가 사용한 버전은 다음과 같다.
>
> - python - 3.9.7
> - numpy - 1.21.2
> - matplotlib - 3.5.1
> - matplotlib-inline - 0.1.2
> - torch - 1.10.2 (파이토치)
> - torchvision - 0.11.3

- gym - 0.21.0
- pygame - 2.1.2 (윈도우가 아닌 운영체제는 이 모듈이 필요하다.)

2) 모듈 불러오기

```
# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T
import gym
from typing import Dict, NamedTuple, List
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import os
```

3) Experience 클래스 정의하기

Deep Q-learning 알고리즘의 *Transition* 을 만들기 위해서는 다음과 같은 정보가 필요하다.

논문의 용어를 가져오면 다음과 같다.

1. obs: s_t
2. action: a_t
3. reward: r_t
4. done: $t = T$이면 True이고, $t \neq T$이면 False
5. next_obs: s_{t+1}

```
class Experience(NamedTuple):
    obs: np.ndarray
    action: np.ndarray
    reward: float
    done: bool
    next_obs: np.ndarray

# 하나의 에피소드에서 모든 Experience를 저장한다.
Trajectory = List[Experience]

# Trajectory에서 일정한 크기의 Experience를 임의로 뽑아낸다.
Buffer = List[Experience]
```

4) 모델 클래스 생성하기

이는 뉴럴 네트워크를 사용하기 위한 코드이다.

```
# PyTorch를 사용해서 다음과 같은 뉴럴 네트워크 구조를 만든다.
class model(nn.Module):
    def __init__(self, input_size, output_size):
        super(model, self).__init__()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, output_size)
        )

    def forward(self, x):
        x = self.linear_relu_stack(x)
        return x
```

5) 학습자 클래스 정의하기

저번 챕터의 *Q-learning* 알고리즘에서 설명한 $Q(S, A)$를 업데이트 방법에 주목하자.

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_{a} Q(S', a) - Q(S, A) \right]$$

이 수식을 학습자 클래스의 **update_q_net**에서 코드로 볼 수 있다.

```
class Trainer:

    # generate_trajectory는 학습 전에 쓰이는 데이터를 수집한다.
    @staticmethod
    def generate_trajectory(
        q_net: model, buffer_size: int, epsilon: float
    ):
        # 이는 Replay Memory를 사용하기 위한 버퍼이다.
        buffer: Buffer = []

        # 환경으로부터 초기 상태 값을 가져온다.
        observation = env.reset()

        # 에이전트의 Transition을 저장한다.
        trajectory_from_agent: [Trajectory] = []

        # 환경의 전 상태값을 저장한다.
        last_obs_from_agent = np.ndarray

        # Return을 계산하기 위해 축적된 보상을 저장한다.
        cumulative_reward_from_agent = 0

        # Return들을 저장한다.
        cumulative_rewards: List[float] = []


        # 설정의 버퍼의 크기 만큼만 버퍼에 Transition을 저장한다.
        while len(buffer) < buffer_size:
            #=============================================
            # 엡실론 greedy 방법으로 행동을 선택한다.
            #=============================================
            if not len(buffer) == 0: # not first step:
                action_values = (
                    q_net(torch.from_numpy(observation)).detach().numpy()
                )

                if random.uniform(0, 1) <= epsilon:
                    action_ind = env.action_space.sample()
                else:
                    action_ind = np.argmax(action_values)
            else:
                action_ind = env.action_space.sample()

            action = [action_ind]
            #=============================================
```

```
            # 에이전트가 한번의 step을 진행한다.
            env.render()
            next_observation, reward, done, _ = env.step(action_ind)


            # 만약 Terminal 상태에 도달하지 않았다면 다음 코드를 실행한다.
            if not done:
                exp = Experience(
                    obs = observation,
                    reward = reward,
                    done = False,
                    action = action,
                    next_obs = next_observation
                )
                trajectory_from_agent.append(exp)
                cumulative_reward_from_agent += reward
                observation = next_observation

            # 만약 Terminal 상태에 도달했다면 다음 코드를 실행한다.
            else:
                last_experience = Experience(
                    obs = observation,
                    reward = reward,
                    done = done,
                    action = action,
                    next_obs = next_observation
                )

                cumulative_reward = cumulative_reward_from_agent + reward
                cumulative_rewards.append(cumulative_reward)
                cumulative_reward_from_agent = 0

                buffer.extend(trajectory_from_agent)
                buffer.append(last_experience)

                # 하나의 Trajectory가 수집되었으므로 환경을 초기화하고 다시 반복한다.
                env.reset()

        # 얻어진 데이터와 보상 축적값의 평균값을 리턴한다.
        return buffer, np.mean(cumulative_rewards)

    # update_q_net은 뉴럴 네트워크에 대하여 Gradient Descent를 진행한다.
    @staticmethod
    def update_q_net(
        q_net: model,
        optimizer: torch.optim,
```

```
    buffer: Buffer,
    action_size = int
):
    BATCH_SIZE = 1000
    NUM_EPOCH = 3
    GAMMA = 0.9
    batch_size = min(len(buffer), BATCH_SIZE)
    random.shuffle(buffer)

    # 버퍼에서 배치의 크기만큼 데이터를 추출한다.
    batches = [
        buffer[batch_size * start : batch_size * (start + 1)]
        for start in range(int(len(buffer) / batch_size))
    ]

    for _ in range(NUM_EPOCH):
        for batch in batches:
            obs = torch.from_numpy(np.stack([ex.obs for ex in batch]))
            reward = torch.from_numpy(
                np.array([ex.reward for ex in batch], dtype = np.float32).reshape(-1, 1)
            )
            done = torch.from_numpy(
                np.array([ex.done for ex in batch], dtype = np.float32).reshape(-1, 1)
            )
            action = torch.from_numpy(np.stack([ex.action for ex in batch])).to(torch.int64)
            next_obs = torch.from_numpy(np.stack([ex.next_obs for ex in batch]))

            # 전 챕터에서 설명한 Deep Q-learning의 타겟이다.
            target = (
                reward
                + (1.0 - done)
                * GAMMA
                * torch.max(q_net(next_obs).detach(), dim=1, keepdim=True).values
            )

            #============================================
            # 뉴럴 네트워크의 에러를 구한다.
            #============================================
            mask = torch.zeros((len(batch), action_size))
            mask.scatter_(1, action, 1)

            prediction = torch.sum(q_net(obs) * mask, dim = 1, keepdim = True)
            criterion = nn.MSELoss()
            loss = criterion(prediction, target)
            #============================================
```

```
#============================================
# Gradient Descent
#============================================
optimizer.zero_grad()
loss.backward()
optimizer.step()
#============================================
```

6) 에이전트 학습시키기

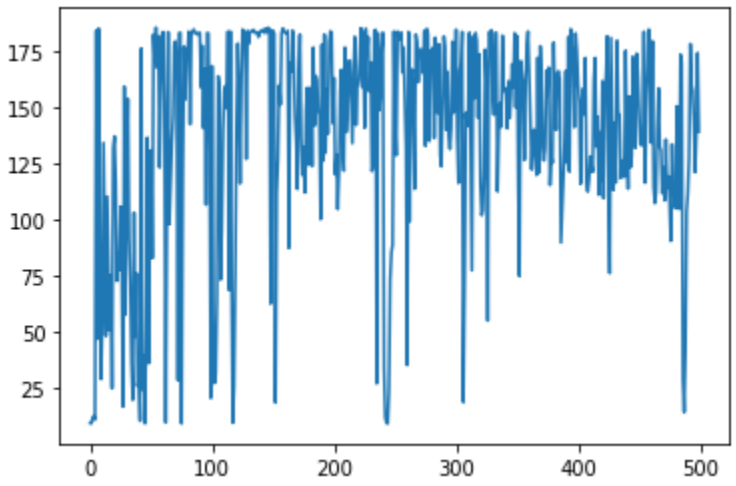
```
#============================================
# 만약 환경이 실행중이라면 종료
#============================================
try:
    env.close()
except:
    pass
#============================================

#============================================
# 학습에 필요한 값들을 초기화한다.
#============================================
env.reset()

num_actions = env.action_space.n
num_obs = env.observation_space.shape[0]

# 아래 3개의 변수는 엡실론 스케쥴링을 위해 필요하다.
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
#============================================

try:
    qnet = model(num_obs, num_actions)

    experiences: Buffer = []
    optim = torch.optim.Adam(qnet.parameters(), lr = 0.001)

    cumulative_rewards: List[float] = []

    NUM_TRAINING_STEPS = int(os.getenv('QLEARNING_NUM_TRAINING_STEPS', 500))
    NUM_NEW_EXP = int(os.getenv("QLEARNING_NUM_NEW_EXP", 10000))
    BUFFER_SIZE = int(os.getenv("QLEARNING_BUFFER_SIZE", 1000))

    for n in range(NUM_TRAINING_STEPS):

        # 엡실론을 설정한다.
        eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * n /EPS_DECAY)

        # 학습에 사용될 데이터를 가져온다.
        new_exp, _ = Trainer.generate_trajectory(qnet, NUM_NEW_EXP, epsilon = eps_threshold)

        # 전 챕터에서 언급한 데이터의 독립성을 보장하기 위해 필요하다.
        random.shuffle(experiences)
```

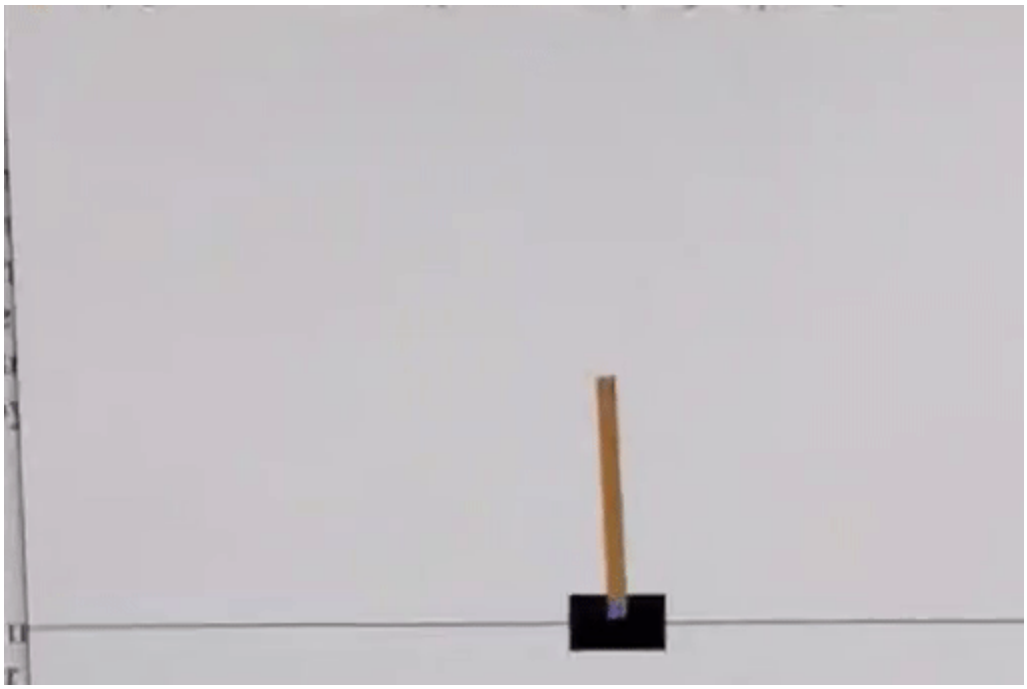
```
        # 버퍼 사이즈 만큼만 데이터를 수집한다.
        if len(experiences) > BUFFER_SIZE:
            experiences = experiences[:BUFFER_SIZE]
        experiences.extend(new_exp)

        # 학습을 진행하고 보상값을 저장 및 출력한다.
        Trainer.update_q_net(qnet, optim, experiences, num_actions)
        _, rewards = Trainer.generate_trajectory(qnet, 10000, epsilon = 0)
        cumulative_rewards.append(rewards)
        print(f"Training step: {n+1}\treward: {rewards}")
except KeyboardInterrupt:
    print("\nTraining interrupted")
finally:
    env.close()

# 학습이 끝나면 보상의 변화를 그래프로 표현하고 모델을 저장한다.
try:
    plt.plot(range(NUM_TRAINING_STEPS), cumulative_rewards)

    # SAVE MODEL
    torch.save(qnet, './saved_models/model.pt')

except ValueError:
    print("\nPlot failed on interrupted training.")
```

7) 학습 결과

위의 코드로 학습을 진행한 결과는 다음과 같다.

```
Training step: 1        reward: 9.72340425531915
Training step: 2        reward: 9.541666666666666
Training step: 3        reward: 11.651162790697674
Training step: 4        reward: 12.829268292682928
Training step: 5        reward: 11.227272727272727
Training step: 6        reward: 183.9090909090909
Training step: 7        reward: 47.095238095238095
Training step: 8        reward: 184.8181818181818
Training step: 9        reward: 82.1875
Training step: 10        reward: 29.185185185185187
.
.
.
Training step: 490        reward: 103.78571428571429
Training step: 491        reward: 111.78571428571429
Training step: 492        reward: 127.15384615384616
Training step: 493        reward: 178.0
Training step: 494        reward: 159.83333333333334
Training step: 495        reward: 157.33333333333334
Training step: 496        reward: 143.25
Training step: 497        reward: 121.07692307692308
Training step: 498        reward: 138.53846153846155
Training step: 499        reward: 174.0
Training step: 500        reward: 139.07692307692307
```





8) Summary

지금까지 *Deep Q-learning*을 사용하여 에이전트를 학습시키는 코드를 작성했다.
수학적인 개념을 코드에 적용시키기는 쉽지 않기 때문에 이번 챕터는 의미가 있다.

다음 챕터에서는 이러한 *Deep Q-learning*을 복습하고, 더 나아간 방법을 제시하기 위해 **Unity**를 사용하여 만들어진 환경과 파이썬 코드를 연동하여 에이전트 학습을 진행하고자 한다.