Reinforcement Learning Homework Assignment

Mountain Car

컴퓨터공학과
120200366 이종석

1. Algorithm - Double Deep Q Network (Double DQN)

- Overestimation 문제를 방지 하기 위해서 DQN과 Target Network 총 2개의 네트워크를 사용한다.
- DQN은 action selection을 위해 사용되는 네트워크이고, Target Network는 Q value 계산을 위해 사용되는 네트워크 이다.

$$y = r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_{\theta}(s', a'))$$

<action select DQN>

$$y = r + \gamma Q_{\theta'}(s', a')$$

<Q value computation Target Network>

1. Algorithm - Double Deep Q Network (Double DQN)

- DQN은 action selection을 위해 사용되는 네트워크이고, Target Network는 Q value 계산을 위해 사용되는 네트워크 이다.
- Action Selection을 위해 Q Network는 계속 업데이트를 하고 Q value 계산을 위해 Target Network는 계속 train하지 않고 일정 간격마다 Q Network로 갱신한다.
- Experience Replay Buffer을 통해 DQN의 학습과정에서 매 step마다 주어지는 state, action, reward, next_state, done등을 경험치로 축적하여 축적된 경험치를 이용하여 Network를 학습할 때 이용한다.

2. Code

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

BUF_SIZE = 10000
BATCH_SIZE = 128
UPDATE_TARGET = 500  # 500씩 만큼 target network 업데이트
GAMMA = 0.99

seed = 17
np.random.seed(seed)
random.seed(seed)
env = gym.make('MountainCar-v0')

env.seed(seed)
torch.manual_seed(seed)
```

- DDQN에 사용되는 buffer size, batch size, update target, gamma 의 변수 초기화 및 mountain car 환경 생성 및 epsilon 사용을 위한 seed 초기화 등이 선언 되어있다.

2. Code

- 학습에 이용되는 정보를 저장할 Replay Buffer를 클래스 객체로 나타낸 코드이다. 매개변수 cap을 통해 buffer의 용량을 정하고 mem 배열 변수에 학습에 이용된 (state, action, reward, next_state, done) 정보를 저장한다. 저장하는 함수는 add를 통해 구현하였다. buffer에서 현재 사용된 element의 아이템을 추출할 수 있도록 배열 mem에서 현재 위치를 저장해놓았다. Sample을 통해 buffer에서 batch size만큼의 (state, action, reward, next_state, done) 정보를 샘플링한다.

```python
class Buffer:
    def __init__(self, cap):
        # cap : buffer size = 10000 (capacity)
        self.cap = cap
        self.mem = []
        self.pos = -1  # 마지막으로 기록된 mem 요소의 위치

    def add(self, element):
        if len(self.mem) < self.cap:
            self.mem.append(None)
        new_pos = (self.pos + 1) % self.cap
        self.mem[new_pos] = element
        self.pos = new_pos

    def sample(self, batch_size):
        return random.sample(self.mem, batch_size)

    def __len__(self):
        return len(self.mem)

    def __getitem__(self, item):
        return self.mem[(self.pos + 1 + item) % self.cap]
```

2. Code

```python
class Model(nn.Module):
    # input dim(state dim) = 2, output dim(action dim) : 3
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 3)

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

- Q Network의 Neural Network를 구현하였다.
- 모두 linear 함수로 mountain car problem의 state는 총 2개, output dim은 총 3개이므로 input dim은 2, output dim은 3으로 하였고 히든레이어의 수는 모두 64로 구현하였다.

2. Code

```
class DQN:
    def __init__(self):
        self.network = Model().to(device)
        self.target_network = copy.deepcopy(self.network).to(device)
        self.optimizer = optim.Adam(self.network.parameters(), lr=5e-4)
```

- DQN의 클래스 구현 코드이다. DQN의 Q network는 앞 페이지의 Model 객체를 이용하여 선언을 하였고, DQN 네트워크 생성 시 target network는 Q network와 동일한 파라미터를 가진 네트워크로 생성하였다. Optimizer는 Adam을, learning rate는 5e-4로 설정하였다.

2. Code

```
def train(self, batch):
    states, actions, rewards, next_states, dones = zip(*batch)
    states = torch.from_numpy(np.array(states)).float().to(device)
    actions = torch.from_numpy(np.array(actions)).to(device).unsqueeze(1)
    rewards = torch.from_numpy(np.array(rewards)).float().to(device).unsqueeze(1)
    next_states = torch.from_numpy(np.array(next_states)).float().to(device)
    dones = torch.from_numpy(np.array(dones)).to(device).unsqueeze(1)
```

- DQN의 train 함수의 초기 부분이다. 학습에 이용될 batch size 만큼의 (state, action, reward, next_state, done) 정보를 buffer를 통해 매개변수로 전달 받고 해당 정보들을 학습에 이용될 수 있도록 학습에 이용될 devic에 맞춰 데이터를 변환하고 는 과정을 담고 있다.

2. Code

```
with torch.no_grad():  # Double DQN
    argmax = self.network(next_states).detach().max(1)[1].unsqueeze(1)
    target = rewards + (GAMMA * self.target_network(next_states).detach().gather(1, argmax)) * (~dones)

Q_current = self.network(states).gather(1, actions.type(torch.int64))
self.optimizer.zero_grad()
# loss = F.mse_loss(target, Q_current)
loss = F.mse_loss(target, Q_current)

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

- DQN의 train 함수의 이어지는 부분이다(2). Double DQN의 학습을 위해 action select를 위한 Q network(코드 상 self.network)를 학습하는 코드이다. Target network의 weight로 평가를 하기 위해 Q network와 target network와의 loss를 측정하여 actio의 선택과 평가를 분리한 코드이다.

2. Code

```
def act(self, state):
    state = torch.tensor(state).to(device).float()
    with torch.no_grad():
        Q_values = self.network(state.unsqueeze(0))
    return np.argmax(Q_values.cpu().data.numpy())


def update_target(self):
    self.target_network = copy.deepcopy(self.network)
```

- DQN의 train 함수의 이어지는 부분이다(3). Q network의 weight에 따라 action을 정하는 함수 act와 N번째 에피소드마다 Q network의 파라미터를 복사하는 방식으로 target network를 업데이트 하는 함수 update_target으로 구성되어 있다.

2. Code

```
def transform_state(state):
    state = (np.array(state) + np.array((1.2, 0.0))) / np.array((1.8, 0.07))
    result = []
    result.extend(state)
    return np.array(state)


def eps_greedy(env, dqn, state, eps):
    if random.random() < eps:
        return env.action_space.sample()
    return dqn.act(state)
```

- 해당 assigment 에서 사용되는 util 함수들이다. Mountain car의 state의 범위로는 position [-1.2, 0.6], velocity [-0.07, 0.07]으로 정해져 있지만 normalization를 위해 transform_state의 함수의 수식을 이용하여 position [0, 1], velocity [-0.07, 0.07]로 정규화하였다. Epsilon에 따라서 random action을 지정하거나 Q를 최대화 하는 action을 선택하는 eps_greedy 함수를 선언하였다.

2. Code

```
dqn = DQN()
buf = Buffer(BUF_SIZE)

episodes = 1500
eps = 1
eps_coeff = 0.995
dqn_updates = 0
output_period = 100

rews = deque(maxlen=output_period)  # 각 에피소드에 대한 보상을 여기에 사용한다.
rews_all = [None] * episodes  # 그래프에서 나타낼 rewards를 모두 rews_all에 저장
```

- DQN 네트워크의 생성과 buffer의 생성이다. 또한 episode 의 수를 위한 변수 episodes, epsilon 변수와 episode가 진행됨에 따라 epsilon의 영향을 줄이기 위한 eps, eps_coeff 변수, Q network의 update count를 저장하여 500번 마다 target network의 update를 위한 변수 dqn_updates, 해당 에피소드의 reward를 저장할 변수 rews, 그래프에 나타낼 reward를 나타낼 변수 rews_all의 선언이다.

2. Code

- Episode 만큼 DQN를 학습하는 과정의 코드이다. Total_reward를 통해 리워드의 총합을 더하고 eps을 통해 현재 state에 대한 액션을 정하여 next_state 및 reward, done을 저장한 수 수식에 의한 reward를 구한다. Buffer의 사이즈가 batch_size 보다 크게되면 dqn을 학습하고 dqn이 N번 학습 될 때 마다 target network를 학습한다. State에 next_state를 대입하여 이와 같은 과정을 도착하거나 mountain car의 timestep 200번이 끝날 때 까지 반복한다.

```
for i in range(1, episodes + 1):
    state = transform_state(env.reset())
    done = False

    total_reward = 0
    while not done:
        action = eps_greedy(env, dqn, state, eps)
        next_state, reward, done, _ = env.step(action)
        next_state = transform_state(next_state)
        total_reward += reward
        reward += 300 * (GAMMA * abs(next_state[1]) - abs(state[1]))
        buf.add((state, action, reward, next_state, done))
        if len(buf) >= BATCH_SIZE:
            dqn.train(buf.sample(BATCH_SIZE))
            dqn_updates += 1
        if not dqn_updates % UPDATE_TARGET:
            dqn.update_target()
        state = next_state
```

2. Code

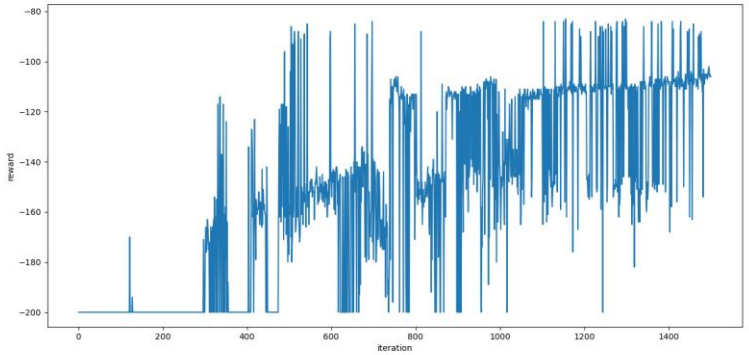
```
    eps *= eps_coeff
    rews.append(total_reward)
    rews_all[i - 1] = total_reward

    mean_r = np.mean(rews)
    max_r = np.max(rews)
    min_r = np.min(rews)
    print(f'\repisode {i}, eps = {eps}, mean = {mean_r}, min = {min_r}, max = {max_r}', end="")
    if not i % output_period:
        print(f'\repisode {i}, eps = {eps}, mean = {mean_r}, min = {min_r}, max = {max_r}')
```

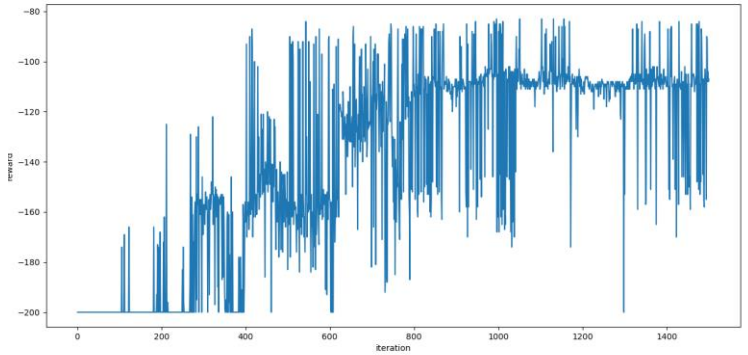
- 각 에피소드가 수행될 때 마다 epsilon의 영향을 줄이기 위해 eps에 계속 eps_coeff를 곱한다. 현재 episode에 도착을 하거나 200번의 timestep이 끝날 때 까지의 reward를 모두 rews에 저장한다. 모든 episode의 reward를 rews_all에 저장을 한다. Mean_r = np.mean(rews) 부터 맨 마지막 줄의 print(…) 까지는 해당 에피소드의 epsilon, 평균 reward, 최소 reward, 최대 reward를 출력한다.

3. Graph

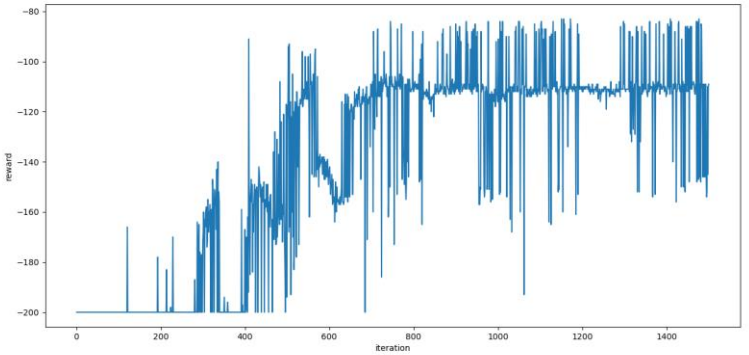
Buffer Size에 따른 그래프 비교
UPDATE_TARGET = 500, GAMMA = 0.99, EPS_COEFF = 0.995 등 모든 파라미터 동일



Buffer Size 8000



Buffer Size 10000

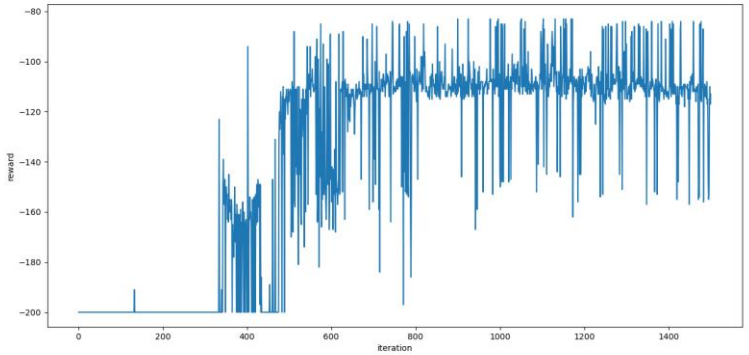


Buffer Size 12000

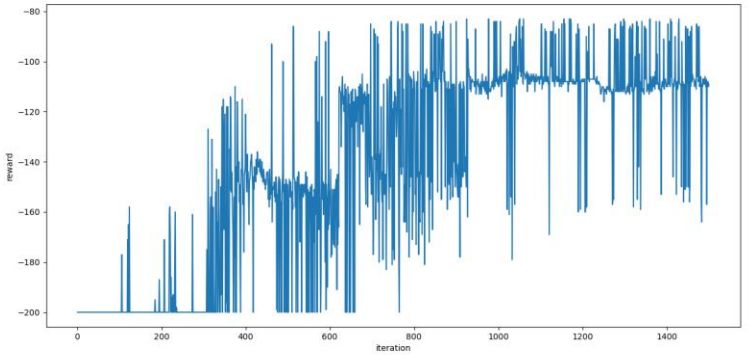
➔ Buffer의 size가 클 수록 reward의 안정화, 빠른 학습에 더 높은 성능

3. Graph

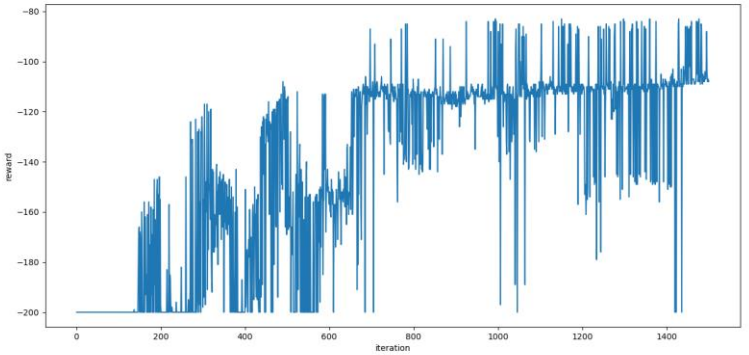
UPDATE_TARGET에 따른 그래프 비교
Buffer Size = 10000, GAMMA = 0.99, EPS_COEFF = 0.995 등 모든 파라미터 동일



UPDATE TARGET 300



UPDATE TARGET 500



UPDATE TARGET 700

➔ Reward의 안정화는 update target에 영향을 많이 받지 않지만

Mountain Car Problem의 해결 iteration은 update target이
높을 수록 더 빨라짐

4. Implementing

- 여러 파라미터의 값 설정에 따라 실험 결과가 바뀌는 것을 알 수 있었다.

- Double DQN에서 Q Network와 Target Network의 역할을 알 수 있었다.
  어떤 시점에 어떤 네트워크를 사용하고 학습할지를 꼭 반드시 명심하고
  프로젝트를 진행해야 할 것 같다.

5. Running Method & Library

- 실행 방법 :
  라이브러리 설치 완료 후, command 프로젝트 폴더 변경 후 python main.py 입력

- 라이브러리 : (설치 : pip install requirements.txt)
  torch, matplot, copy, gym

6. Reference

- DQN을 사용한 MountainCar-v0의 코드 이해 및 기본 코드

  https://m.blog.naver.com/sonmh12/222036705425

- DDQN의 심층적 이해 :

  https://ropiens.tistory.com/134

감사합니다.