DQN Implementation

A) 코드 설명

1) memory.py (ReplayMemory Class)

```
class ReplayMemory:
    def __init__(
        self,
        observation_shape: tuple = (),
        action_shape: tuple = (),
        buffer_size: int = 50000,
        num_steps: int = 1,
    ):
self.observation_shape = observation_shape
self.action_shape = action_shape
self.buffer_size = buffer_size
self.num_steps = num_steps

self.mem = deque()
```

Replay memory의 기본 setting은 변경하지 않았다.
여기서 self.mem은 reply memory를 저장하는 공간으로 deque 라이브러리를 사용했다.
deque는 최대 저장 가능한 원소 개수를 정할 수 있으며, 정해진 개수보다 많은 데이터가 담길 경우 가장 최근에 넣은 데이터가 out된다. 따라서, replay memory에 max 값보다 큰 데이터가 들어가게 된다면, 가장 최근에 넣은 메모리는 out되어 굳이 FIFO 기능을 구현하지 않아도 되기에 이를 사용했다.

```
def write(self, state, action, reward, next_state, done):
    self.mem.append([state, action, reward, next_state, done])
```

memory.py의 ReplayMemory 클라스에 있는 write 함수이다.
이 함수는 state, action, reward, next state, done flag 정보를 한 묶음으로 list화하여 replay memory에 저장하는 역할을 한다.

```python
# 버퍼에서 Uniform Random으로 Transition들을 뽑습니다.
def sample(self, num_samples: int = 1): # -> Tuple[np.ndarray]:
    batch = random.sample(self.mem, num_samples)

    states, actions, rewards, next_states, dones = [], [], [], [], []
    for state, action, reward, next_state, done in batch:
        states.append(state)
        actions.append(action)
        rewards.append(reward)
        next_states.append(next_state)
        dones.append(done)

    return [states, actions, rewards, next_states, dones]
```

memory.py의 ReplayMemory 클라스에 있는 sample 함수이다.
이 함수는 Original DQN을 훈련시킬 때 사용하는 것이다.
우선 batch size를 input으로 받는다. 그 이후, random.sample을 통해 batch size 만큼 random하게 샘플을 추출한다. 추출된 샘플들은 states, actions, rewards, next_states, dones 중 각각 해당하는 곳에 list 형식으로 넣어준다. 최종적으로 states, actions, rewards, next_states, dones을 리턴해준다.

```python
def nStep_sample(self):
    return self.mem
```

memory.py의 ReplayMemory 클라스에 있는 nStep_sample 함수이다.
이 함수는 n-step DQN을 훈련시킬 때 사용하는 것이다. n-step할 때 저장하고 있는 memory를 그냥 return 해준다.
여기서의 memory는 origianl DQN과는 다르게 n step까지는 정보를 저장하는데 사용된다.

```python
def clean(self):
    self.mem=deque()
```

memory.py의 ReplayMemory 클라스에 있는 clean 함수이다.
이 함수는 replay memory 혹은 그냥 memory를 초기화시키는데 사용된다

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

memory.py의 ReplayMemory 클라스에 있는 __len__ 함수로 객체에 len() 함수를 사용할 때 메모리의 현재 크기를 return한다.
2) model.py (SimpleMLP Class)

```python
class SimpleMLP(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()

        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, output_size)

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

이 함수는 model.py 내 SimpleMLP 클라스다. input으로는 모델의 input size와 output size를 받는다.
모델을 제작할때는 pytorch를 사용했으며, 모델은 torch.nn.Module을 상속한다.
모델은 1개 입력층, 1개 히든 레이어, 1개 출력층을 가지고 있다.
입력층은 input : 4, output : 128, 히든 레이어는 input : 128, output : 64, 출력층은 input : 64, output : 2 로 구성되어 있다.
각 층은 간단한 Linear 형식으로 제작했고, 입력층과 히든 레이어에는 각 output에 활성함수인 relu를 적용해줬다.

3) trainer.py (DQNTrainer Class)
3-1) __init__

```python
class DQNTrainer:
    def __init__(
        self,
        config
    ):
        ## Train Factor
        self.config = config
        self.env = gym.make(config.env_id)
        self.epsilon = self.config.eps_start  # My little gift for you

        ## Model Information
        self.predmodel=SimpleMLP(4, 2)
        self.fixmodel = SimpleMLP(4, 2)
        self.fixmodel.load_state_dict(self.predmodel.state_dict())
        self.savemodel = SimpleMLP(4, 2)

        ## Optimizer option learning rate = 0.0003
        self.optimizer = self.config.optim_cls(self.predmodel.parameters(), self.config.optim_kwargs['lr'])

        ## Replay Memory
        self.mem=ReplayMemory()

        ## Mode Information
        print("*** Select the Mode ***")
        print("*** Only Number Please ***")
        print("1. Original DQN")
        print("2. N-Step DQN")
        self.mode=int(input("Mode : "))
        self.n = 1
        if self.mode == 2: self.n = int(input("Step Number : "))
```

DQNTrainer 클라스는 input으로 config 객체를 받는다. 이 객체는 DQN 과정에서의 하이퍼

파라미터를 일부 담고 있다.
config를 객체에 저장하고 객체의 환경을 self.env에 연결한다. 이 환경은 gym 라이브러리에서 제공하는 것이다.
predmodel로 Q 값의 최대를 찾을 것이며, fixmodel로 predmodel과의 값 차이를 비교해 모델을 업데이트하는데 사용할 것이다. (Fixed model DQN)
savemodel로 train하는 과정에서 가장 성능이 좋다고 여겨지는 부분의 모델을 저장한다.
optimizer는 adam optimizer를 사용할 것이며, learning rate는 0.0003으로 설정했다. 여러 차례 실험 결과 learning rate을 0.0003으로 설정하는 것이 가장 안정적이라고 판단했다.
self.mem는 replay memory를 저장하는 객체를 저장한다.
__init__ 함수가 끝나기 전에, 이번 train을 실행할 때 origianl DQN을 할지 N-step DQN을 할지에 대한 mode를 설정한다. 만약 선택한 mode가 N-step DQN일 경우, 스텝 수를 설정하는 n 값을 받아온다. 반대로 original DQN일 경우, n은 자동으로 1로 설정된다.

3-2) train

```
def train(self, num_train_steps: int):

    episode_rewards = []
    ## Train According to the Mode
    if self.mode == 1: episode_rewards = self.Origin_DQN(num_train_steps)
    elif self.mode == 2: episode_rewards = self.NStep_DQN(num_train_steps)

    return episode_rewards
```

DQNTrainer 클라스의 train 함수이다.
이 함수는 mode 별로 다른 training 알고리즘을 사용한다.
mode가 1일 경우는 original DQN(mini batch based), mode가 2일 경우는 n-step DQN이다. 각 함수는 episode 별 reward를 저 저장한 list를 episode_rewards에 저장한다. train 함수는 최종적으로 episode_rewards를 return한다.

3-3) update_target

```
# Update the target network's weights with the online network's one.
def update_target(self):
    self.fixmodel.load_state_dict(self.predmodel.state_dict())
```

DQNTrianer 클라스의 update_target 함수이다.
이 함수는 fixed model의 파라미터를 predict model의 파라미터로 업데이트하는 함수이다.

3-4) update_epsilon_origin

```
# Update epsilon over training process.
def update_epsilon_Origin(self, eGreedy):
    eGreedy -= 0.0003
    eGreedy = max(self.config.eps_end, eGreedy)
    return eGreedy
```

DQNTrainer 클래스의 update_epsilon_origin 함수이다.
여러 차례 실험 결과 original DQN을 훈련시킬 때, 1500정도 안 밖의 episode만 실행되는 것을 확인했다. 따라서, 적절한 탐색이 이뤄질 수 있도록 하기 위해 episode 마다 적절하게 epsilon이 줄어드는 메커니즘으로 이 함수를 구현했다. 한편, 가끔의 탐색을 추가하기 위해 epsilon의 최소 값을 0.01로 설정해뒀다.

3-5) update_epsilon_NSTEP

```
def updata_epsilon_NSTEP(self, eGreedy):
    eGreedy -= 0.00001
    eGreedy = max(self.config.eps_end, eGreedy)
    return eGreedy
```

DQNTrainer 클래스의 update_epsilon_NSTEP 함수이다.
여러 차례 실험 결과 N-step DQN을 훈련시킬 때, original DQN과 유사한 방식으로 하면 성능이 좋지 못한 것을 확인했습니다. 따라서, 적절한 탐색이 이뤄짐과 동시에 최대 값을 근사할 수 있도록 하기 위한 메커니즘으로 이 함수를 구현했다. 한편, 가끔의 탐색을 추가하기 위해 epsilon의 최소 값을 0.01로 설정해뒀다.

3-6) update_network_Origin

```
# Update online network with samples in the replay memory.
def update_network_Origin(self):
    states, actions, rewards, next_states, dones = self.mem.sample(self.config.batch_size)

    states = torch.Tensor(np.array(states))
    actions = torch.Tensor(np.array(actions))
    actions = actions.type(torch.int64)
    rewards = torch.Tensor(rewards)
    next_states = torch.Tensor(np.array(next_states))
    dones = torch.Tensor(dones)

    predQ = self.predmodel(states)
    predictedQ = torch.gather(predQ, 1, actions.unsqueeze(-1))

    fixQ = self.fixmodel(next_states).max(1)[0]
    expectQ = rewards + (self.config.discount_rate * fixQ)

    loss = F.mse_loss(predictedQ.squeeze(), expectQ)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

DQNTrainer 클라스에 있는 update_network_Origin 함수이다.
이 함수는 original DQN의 네트워크를 업데이트 하는 용도로 사용한다.
우선, reply memory로부터 batch sample을 받아온다.
sample size는 self.config.batch_size에 의해 정해진다.
sample들은 states, actions, rewards, next_states, dones로 나눠지며 tensor 형태로 변경된다.
tensor로 변경된 sample 데이터를 기반으로 prediction model을 사용해 predict Q를 계산한다. torch.gather 함수를 사용해 계산된 Q값과 actions을 매치시킨 Q 값만을 도출한다.
이후, fixed model을 사용해 fixed된 target 값을 도출한다. 이는 max Q 값이다.
마지막으로, mean square loss로 predict Q와 fixed Q의 loss를 도출하고 optimizer을 사용해여 기울기 값을 prediction model의 weight에 적용한다.

3-7) update_network_NSTEP

```
def update_network_NSTEP(self):
    ## FIFO first in memory
    needUpdate = self.mem.nStep_sample()
    temp = needUpdate.popleft()

    ## make data
    updateState = temp[0]
    updateAction = temp[1]
    updateReward = temp[2]
    updateNextState = temp[3]
    updateDone = temp[4]

    updateState = torch.Tensor(np.array([updateState]))
    updateAction = torch.Tensor(np.array([updateAction]))
    updateAction = updateAction.type(torch.int64)
    updateNextState = torch.Tensor(np.array([updateNextState]))

    ## Prediction
    predQ = self.predmodel(updateState)
    predictedQ = torch.gather(predQ, 1, updateAction.unsqueeze(-1))

    ## N-Step learning
    nStepQ = 0
    gamma = self.config.discount_rate
    mul = 1
    for i in range(self.n):
        nStepQ += needUpdate[i][2] * mul
        mul = mul * gamma
    nStepQ += mul * self.fixmodel(torch.Tensor(np.array([needUpdate[-1][0]]))).max(1)[0]
## Back propagation
loss = F.mse_loss(predictedQ.squeeze(), torch.Tensor([nStepQ]))
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

DQNTrainer 클라스에 있는 update_network_NSTEP 함수이다.
우선, memory에 있는 값을 needUpdate에 저장한다.
그리고 가장 먼저 들어갔던 메모리의 state, action, reward, next_state, done을 tensor 형태로 변형한다.
tensor로 변경된 sample 데이터를 기반으로 prediction model을 사용해 predict Q를 계산한다. torch.gather 함수를 사용해 계산된 Q값과 actions을 매치시킨 Q 값만을 도출한다.
그 이후, 실질 n-step learning을 진행한다. 메모리 list에 있는 메모리를 순서대로 n-step 알고리즘에 따라 더 해준다. 여기서 discount 값은 config에 있는 것을 사용했다.
n-step을 통해 Q 값을 얻고 나서, mean square loss로 predict Q와 fixed Q의 loss를 도출하고 optimizer을 사용해여 기울기 값을 prediction model의 weight에 적용한다.

3-8) Origin_DQN

```
def Origin_DQN(self, num_train_steps):
    ## Maximum step setting
    maxStep = num_train_steps
    maxStep = 100000

    ## episode and score per episode
    episode = 1
    scoreList = []

    ## eGreedy option
    eGreedy = self.config.eps_start
    eGreedyMin = self.config.eps_end

    ## save the optimal model, tp -> store the 30 data, mx -> maximum average of tp
    tp = []
    mx = 0

## Train until maxStep smaller than zero
while maxStep > 0:
    ## New Episode
    state = self.env.reset(seed=random.randint(1, 1024))

    score = 0
    for step in range(501):
        getAction=torch.Tensor(np.array([state]))

        state=torch.Tensor(state)

        ## eGreedy based select action
        action = -1
        if random.random() < eGreedy: action = random.randint(0, 1)
        else: action = int(self.predmodel(getAction).max(1)[1])

        ## Next State
        next_state, reward, done, info = self.env.step(action)

        ## Episode is over, give reward -2
        if done : reward = -2

        ## Add replay memory
        self.mem.write(state.numpy(), action, reward, next_state, done)

        ## Learn DQN
        if len(self.mem) > self.config.batch_size+10: self.update_network_Origin()
```

```
    ## Add score
    score+=1

    ## Decrease the maxStep
    maxStep-=1

    ## Update the fixed model every 30 steps
    if maxStep % 30 == 0: self.update_target()

    ## Change the state
    state = next_state

    ## Finish the episode
    if done:
        print("에피소드 : {0}, 점수 : {1}, 남은 Step : {2}".format(episode+1, score, maxStep))
        scoreList.append(score)
        tp.append(score)
        break

    ## Increase the episode
    episode += 1
    eGreedy = self.update_epsilon_Origin(eGreedy)
    eGreedy = max(eGreedyMin, eGreedy)

    if episode % 50 == 0:
        ## Save the best model
        num = sum(tp) / len(tp)
        if num > mx and tp[-1] > 120:
            mx = num
            tp=[]
            self.savemodel.load_state_dict(self.predmodel.state_dict())
        else: tp = []

## Final test
self.finalTest()

return scoreList
```

DQNTrainer 클라스에 있는 Origin_DQN 함수이다.

1. 이 함수는 우선 input으로 learning step을 받고, episode를 1로 설정하고, scoreList는 에피소드 마다 얻어진 reward를 저장하는 것이다. 이후 start epsilon과 minimal epsilon을 설정해준다. tp와 mx는 성능이 좋은 모델을 저장해놓기 위한 것이다. tp는 10~20 개 에피소드의 reward를 저장하고 평균 값을 구하는데 사용되며, mx 값은 평균 값의 최대를 저장하는데 사용된다.

2. DQN learning은 maxStep이 0 이하가 될 때까지 진행된다. 매 에피소드 마다, random

seed를 사용해 랜덤한 환경을 설정해주고, score를 0으로 초기화시킨다. 그리고, 각 step 마다 score는 1씩 증가하고 maxStep은 1씩 감소한다. 매 30 step 마다, fix model의 weight은 prediction model의 weight로 업데이트한다.
3. 현재 state가 결정되고 난 이후, 랜덤 값이 epsilon 보다 작을 경우 랜덤으로 search하고, 아닌 경우는 해당 state에서 prediction model로부터 도출된 가장 큰 Q 값의 action을 선택한다.
4. 환경에 action을 적용해 next state, reward, done flag, info를 받아온다.
만약 에피소드가 종료됐다는 done이 True일 경우 reward를 -2로 설정해주고 그렇지 않은 경우 그냥 넘어간다.
5. action을 통해 얻을 결과를 state, action, reward, next state, done 순서로 replay memory에 저장해둔다.
6. 만약 replay memory의 크기가 batch size보다 클 경우, mini batch learning을 실시하여 network update를 진행한다. (update_network_Origin)
7. 에피소드의 한 step이 종료됐을 때 state를 next state로 변경해준다.
8. 만약, done flag가 True일 경우, 해당 에피소드는 종료된다. 종료됐을 때 에피소드에서 얻은 reward를 scoreList와 tp에 저장한다.
9. 한 개의 에피소드가 종료됐다면, eplison을 업데이트해줍니다.
10. 매 20 차례 에피소드마다 tp를 사용해 해당 구간의 에피소드 reward 평균을 구한다. 만약 평균이 mx보다 높은 경우 mx를 가장 큰 값으로 변경하고 savemodel에 predicition model의 weight를 이식해준다.
11. 최종적으로 모든 learning이 종료됐다면, finalTest 함수를 사용해 original DQN을 실험해본다

3-9) N-Step_DQN

```
def NStep_DQN(self, num_train_steps):
    ## Maximum step setting
    maxStep = num_train_steps
    maxStep = 100000

    ## episode and score per episode
    episode = 1
    scoreList = []

    ## eGreedy option
    eGreedy = self.config.eps_start
    eGreedyMin = self.config.eps_end

    ## save the optimal model, tp -> store the 30 data, mx -> maximum average of tp
    tp = []
    mx = 0
 ## Train until maxStep smaller than zero
 while maxStep > 0:
     ## New Episode
     state = self.env.reset(seed=random.randint(1, 1024))

     score = 0
     for step in range(501):
         getAction = torch.Tensor(np.array([state]))

         state = torch.Tensor(state)

         ## eGreedy based select action
         action = -1
         if random.random() < eGreedy:
             action = random.randint(0, 1)
         else:
             action = int(self.predmodel(getAction).max(1)[1])

         ## Next State
         next_state, reward, done, info = self.env.step(action)

         ## Episode is over, give reward -2
         if done: reward = -1

         ## Add replay memory
         self.mem.write(state.numpy(), action, reward, next_state, done)

         ## Update the DQN
         if len(self.mem) == self.n+1: self.update_network_NSTEP()
```

```
    ## Increase the score
    score += 1

    ## Decrease the maxStep
    maxStep -= 1

    ## Updata eGreedy
    eGreedy = self.updata_epsilon_NSTEP(eGreedy)

    ## Updata the fixed model   ### 원래 % 30, 변경 % 1
    if maxStep % 1 == 0: self.update_target()
    ## Change state to next state
    state = next_state

    ## Episode is over
    if done:
        print("에피소드 : {0}, 점수 : {1}, 남은 Step : {2}".format(episode + 1, score, maxStep))
        scoreList.append(score)
        tp.append(score)
        self.finishLearning()
        break

    ## Update the episode
    episode += 1

    ## Save the best model
    if episode % 10 == 0:
        num = sum(tp) / len(tp)
        if num >= mx and tp[-1] > 120:
            mx = num
            tp = []
            self.savemodel.load_state_dict(self.predmodel.state_dict())
        else:
            tp = []

## Final test
self.finalTest()

return scoreList
```

DQNTrainer 클라스에 있는 NSTEP_DQN 함수이다.

1. 이 함수는 우선 input으로 learning step을 받고, episode를 1로 설정하고, scoreList는 에피소드 마다 얻어진 reward를 저장하는 것이다. 이후 start epsilon과 minimal epsilon을 설정해준다. tp와 mx는 성능이 좋은 모델을 저장해놓기 위한 것이다. tp는 10~20 개 에피소드의 reward를 저장하고 평균 값을 구하는데 사용되며, mx 값은 평균 값의 최대를 저장하는데 사용된다.

2. DQN learning은 maxStep이 0 이하가 될 때까지 진행된다. 매 에피소드 마다, random seed를 사용해 랜덤한 환경을 설정해주고, score를 0으로 초기화시킨다. 그리고, 각 step 마다 score는 1씩 증가하고 maxStep은 1씩 감소한다. 매 30 step 마다, fix model의 weight은 prediction model의 weight로 업데이트한다.

3. 현재 state가 결정되고 난 이후, 랜덤 값이 epsilon 보다 작을 경우 랜덤으로 search하고, 아닌 경우는 해당 state에서 prediction model로부터 도출된 가장 큰 Q 값의 action을 선택한다.
4. 환경에 action을 적용해 next state, reward, done flag, info를 받아온다.
만약 에피소드가 종료됐다는 done이 True일 경우 reward를 -1로 설정해주고 그렇지 않은 경우 그냥 넘어간다.
5. action을 통해 얻을 결과를 state, action, reward, next state, done 순서로 memory에 저장해둔다. (여기서 memory는 replay memory와는 다른 개념으로, n step까지의 경험을 저장해두는 곳이다.)
6. 만약 memory의 크기가 n step보다 1 클 경우, n-step 알고리즘을 사용해 prediction model의 weight를 업데이트 한다. (updat_network_NSTEP)
7. 에피소드의 한 step이 종료됐을 때 state를 next state로 변경해준다.
8. 처음 n-step DQN을 했을 때 fixed target으로 했지만, 실행할 때마다 성능이 천차만별이어서 그냥 prediction model만 변경하는 방식으로 변형했다.
9. n-step DQN은 original DQN과는 다르게 매 step마다 epsilon에 변화를 준다.
10. 만약, done flag가 True일 경우, 해당 에피소드는 종료된다. 종료됐을 때 에피소드에서 얻은 reward를 scoreList와 tp에 저장한다.
11. 매 실행 때마다 성능이 천차만별이기에 가장 좋다고 생각되는 모델을 저장하기 위해 매 10개 에피소드의 평균 reward 값을 비교해 가장 평균 값을 가지는 구간의 model을 save model에 저장해둔다.
11. 최종적으로 모든 learning이 종료됐다면, finalTest 함수를 사용해 n-step DQN을 실험해본다

3-10) finalTest

```
def finalTest(self):
    answer = []
    for i in range(5):
        seed = random.randint(1, 1024)
        for episode in range(1, 11):
            done = False

            point = 0

            state = self.env.reset(seed=seed)
            while not done:
                getAction = torch.Tensor(np.array([state]))
                state = torch.Tensor(state)

                action = int(self.savemodel(getAction).max(1)[1])
                next_state, reward, done, info = self.env.step(action)

                point += 1
                state = next_state

            answer.append(point)

    print("{0}-Step DQN Final test average score : {1}".format(self.n, (sum(answer) / len(answer))))
```

DQNTrainer 클라스에 있는 finalTest 함수이다.
이 함수는 최종적으로 train된 model의 성능을 측정하기 위한 함수이다.
실험은 총 5개 랜덤 seed로 진행될 것이다.
각 seed 별로 10개 에피소드 실험을 진행할 것이다.
실험을 통해 얻어진 모든 reward는 answer에 저장될 것이다.
그 이후 최종적으로 평균 reward를 계산해 보여준다.

3-11) finishLearning

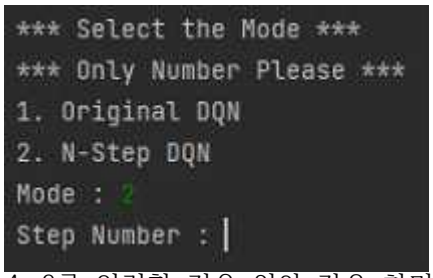
```
def finishLearning(self):
    while self.mem:
        temp = self.mem.mem.popleft()

        st = temp[0]
        ac = temp[1]
        re = temp[2]
        nest = temp[3]

        st = torch.Tensor(np.array([st]))
        ac = torch.Tensor(np.array([ac]))
        ac = ac.type(torch.int64)
        nest = torch.Tensor(np.array([nest]))

        preQ = self.predmodel(st)
        preddQ = torch.gather(preQ, 1, ac.unsqueeze(-1))

        nStepQ = 0
        gamma = 0.98
        mul = 1
        for i in range(len(self.mem.mem)):
            nStepQ += self.mem.mem[i][2] * mul
            mul = mul * gamma

        if len(self.mem) > 0:
            nStepQ += mul * self.fixmodel(torch.Tensor(np.array([self.mem.mem[-1][0]]))).max(1)[0]

        losse = F.mse_loss(preddQ.squeeze(), torch.Tensor([nStepQ]))
        self.optimizer.zero_grad()
        losse.backward()
        self.optimizer.step()
```

DQNTrainer 클라스에 있는 finishLearning 함수이다.
n-step DQN을 훈련하는 과정에서 에피소드가 종료된 경우, memory에 아직 훈련되지 못한 state들이 존재한다. 이 정보 또한 훈련과정에서 얻은 데이터이기에 훈련해줘야 할 필요가 있다. 따라서, n의 값을 조금씩 낮춤으로써 k-step DQN learning을 진행했다. 이 알고리즘은 기존 update_network_NSTEP에서의 알고리즘과 대부분 일치한다.
(여기서 1 <= k < n)

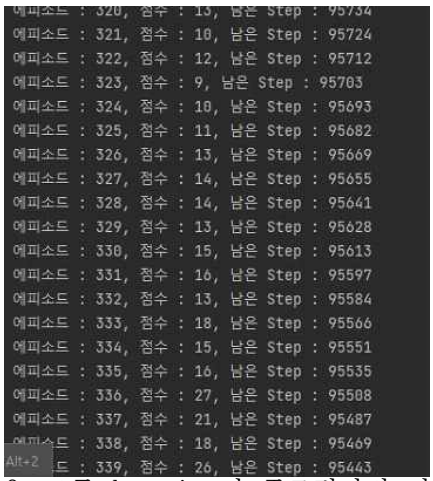
B) 실행 방법

1. train.py를 실행한다면 아래와 같은 화면을 볼 수 있다.

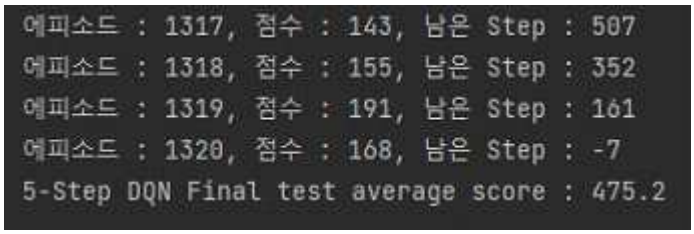
2. “Mode : ”에 원하는 mode의 번호를 입력한다.
3. 1을 입력할 경우 original DQN을 사용한 learning을 확인할 수 있다.

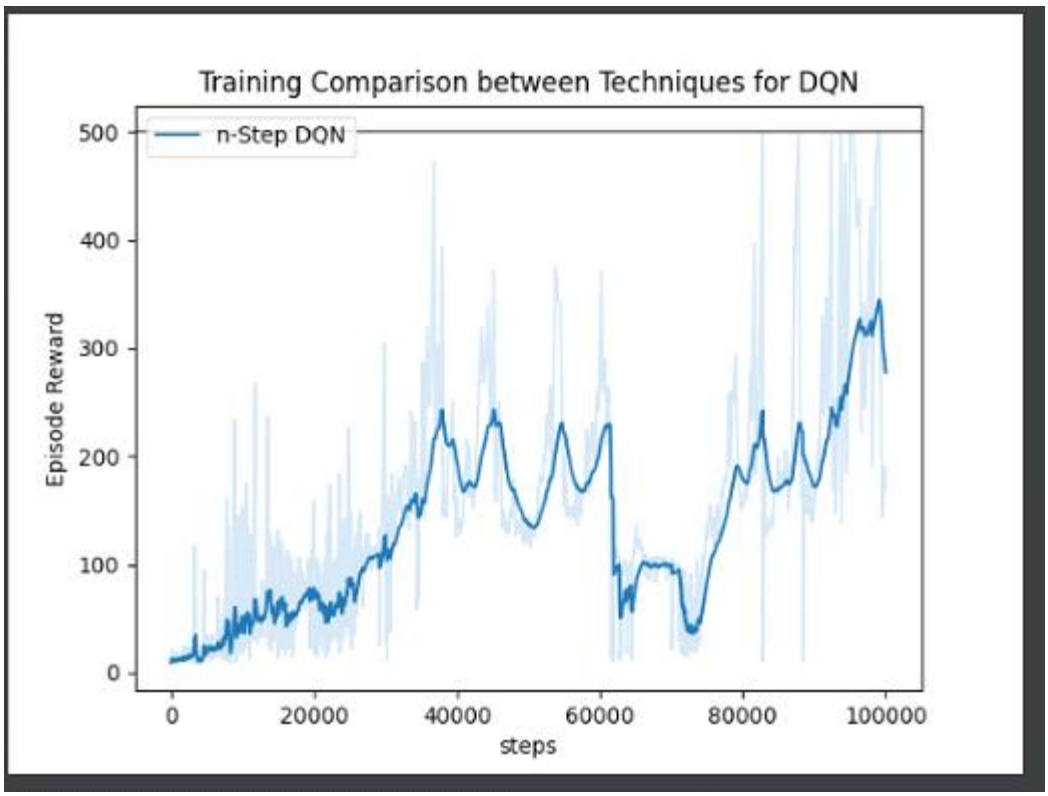


4. 2를 입력할 경우 위와 같은 화면을 확인할 수 있는데, 여기에는 n-step DQN의 n 값을 입력으로 받는 것이다. n의 값은 2~16 사이의 값을 넣어주면 된다.
5. 모드와 n 값을 모두 입력했다면 아래와 같은 화면을 확인할 수 있는데, 화면에서는 현재 에피소드 번호, 해당 에피소드에서 얻은 총 reward, 현재 남은 step 수를 출력되는 것을 확인할 수 있다.



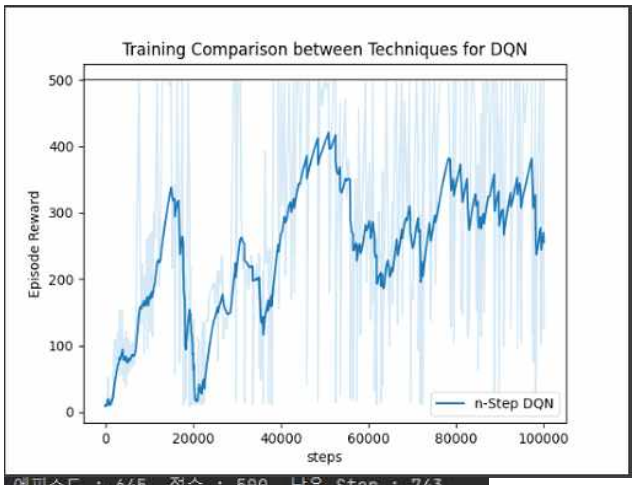
6. 모든 learning이 종료된다면 아래 그림과 같이 랜덤 seed를 통해 실시된 테스트 결과를 확인할 수 있다. (예시)




Training Comparison between Techniques for DQN
n-Step DQN
Episode Reward
steps
0
100
200
300
400
500
0
20000
40000
60000
80000
100000

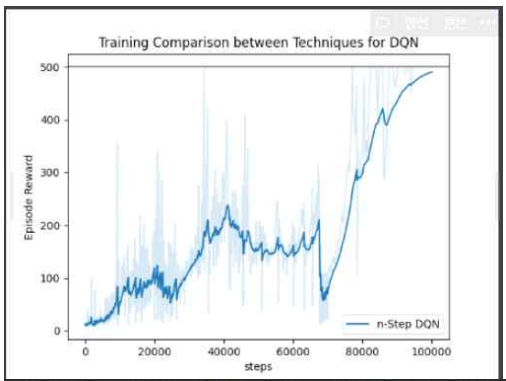
C) 실험 결과

1) Original DQN



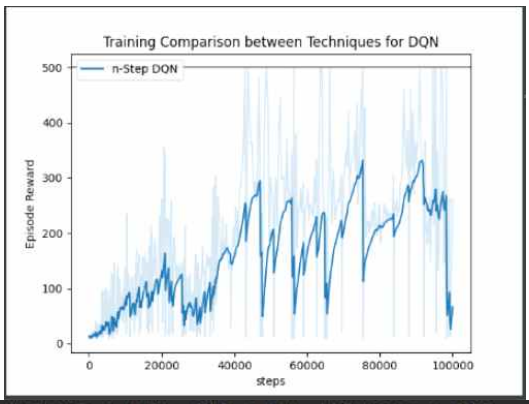

```
에피소드 : 645, 점수 : 500, 남은 Step : 743
에피소드 : 646, 점수 : 103, 남은 Step : 640
에피소드 : 647, 점수 : 108, 남은 Step : 532
에피소드 : 648, 점수 : 500, 남은 Step : 32
에피소드 : 649, 점수 : 126, 남은 Step : -94
1-Step DQN Final test average score : 279.6

Process finished with exit code 0
```

2) 2-step DQN



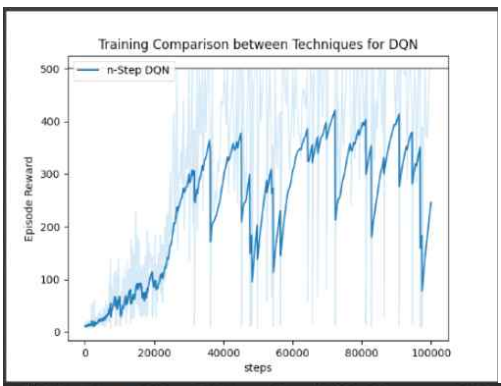

```
에피소드 : 1080, 점수 : 500, 남은 Step : 3414
에피소드 : 1081, 점수 : 500, 남은 Step : 2914
에피소드 : 1082, 점수 : 500, 남은 Step : 2414
에피소드 : 1083, 점수 : 500, 남은 Step : 1914
에피소드 : 1084, 점수 : 500, 남은 Step : 1414
에피소드 : 1085, 점수 : 500, 남은 Step : 914
에피소드 : 1086, 점수 : 500, 남은 Step : 414
에피소드 : 1087, 점수 : 500, 남은 Step : -86
2-Step DQN Final test average score : 500.0

Process finished with exit code 0
```

3) 3-step DQN



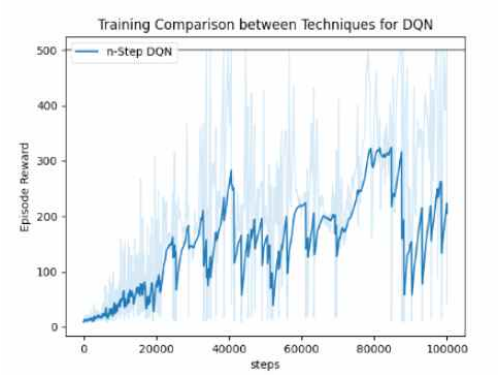

```
에피소드 : 1076, 점수 : 16, 남은 Step : 527
에피소드 : 1077, 점수 : 20, 남은 Step : 507
에피소드 : 1078, 점수 : 127, 남은 Step : 380
에피소드 : 1079, 점수 : 128, 남은 Step : 252
에피소드 : 1080, 점수 : 262, 남은 Step : -10
3-Step DQN Final test average score : 224.4

Process finished with exit code 0
```

4) 4-step DQN



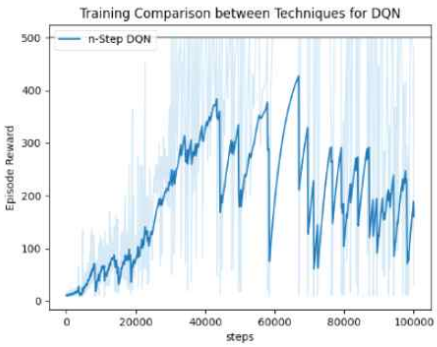

```
에피소드 : 898, 점수 : 16, 남은 Step : 2564
에피소드 : 899, 점수 : 337, 남은 Step : 2227
에피소드 : 900, 점수 : 500, 남은 Step : 1727
에피소드 : 901, 점수 : 368, 남은 Step : 1359
에피소드 : 902, 점수 : 370, 남은 Step : 989
에피소드 : 903, 점수 : 500, 남은 Step : 489
에피소드 : 904, 점수 : 500, 남은 Step : -11
4-Step DQN Final test average score : 313.2

Process finished with exit code 0
```

5) 5-step DQN



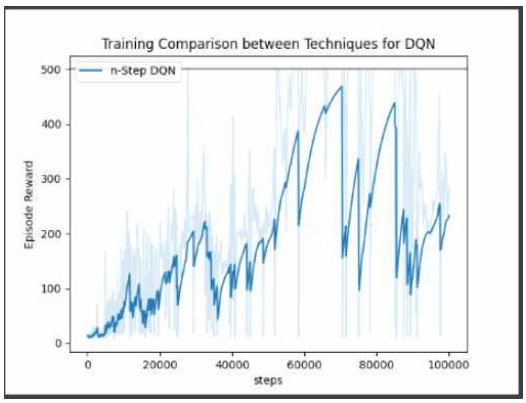

```
에피소드 : 1203, 점수 : 16, 남은 Step : 1443
에피소드 : 1204, 점수 : 14, 남은 Step : 1429
에피소드 : 1205, 점수 : 500, 남은 Step : 929
에피소드 : 1206, 점수 : 392, 남은 Step : 537
에피소드 : 1207, 점수 : 500, 남은 Step : 37
에피소드 : 1208, 점수 : 43, 남은 Step : -6
5-Step DQN Final test average score : 330.2
```

6) 6-step DQN



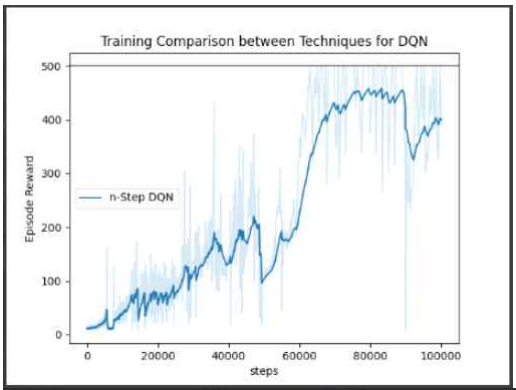

```
에피소드 : 1008, 점수 : 14, 남은 Step : 1540
에피소드 : 1009, 점수 : 500, 남은 Step : 1040
에피소드 : 1010, 점수 : 500, 남은 Step : 540
에피소드 : 1011, 점수 : 479, 남은 Step : 61
에피소드 : 1012, 점수 : 51, 남은 Step : 10
에피소드 : 1013, 점수 : 29, 남은 Step : -19
6-Step DQN Final test average score : 500.0
```

7) 7-step DQN



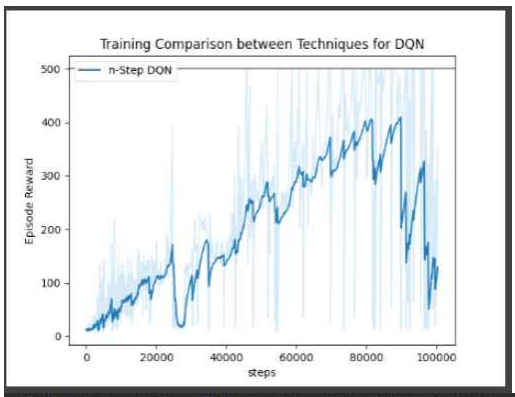

```
에피소드 : 1040, 점수 : 346, 남은 Step : 1037
에피소드 : 1041, 점수 : 320, 남은 Step : 717
에피소드 : 1042, 점수 : 236, 남은 Step : 481
에피소드 : 1043, 점수 : 247, 남은 Step : 234
에피소드 : 1044, 점수 : 286, 남은 Step : -52
7-Step DQN Final test average score : 500.0
```

8) 8-step DQN



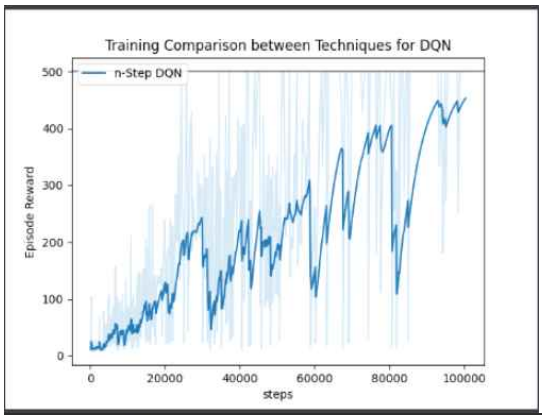

```
에피소드 : 1154, 점수 : 500, 남은 Step : 1429
에피소드 : 1155, 점수 : 351, 남은 Step : 1078
에피소드 : 1156, 점수 : 328, 남은 Step : 750
에피소드 : 1157, 점수 : 500, 남은 Step : 250
에피소드 : 1158, 점수 : 388, 남은 Step : -138
8-Step DQN Final test average score : 495.0
```

9) 9-step DQN



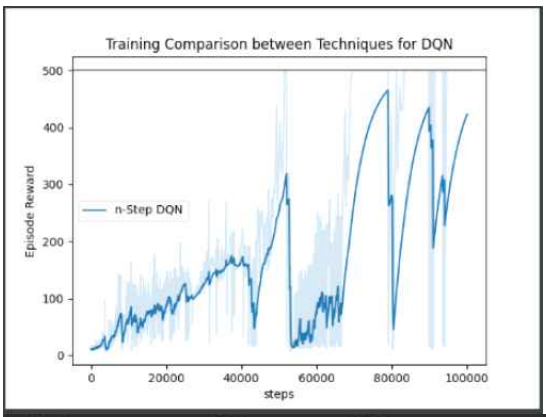

```
에피소드 : 1036, 점수 : 21, 남은 Step : 439
에피소드 : 1037, 점수 : 28, 남은 Step : 411
에피소드 : 1038, 점수 : 292, 남은 Step : 119
에피소드 : 1039, 점수 : 61, 남은 Step : 58
에피소드 : 1040, 점수 : 359, 남은 Step : -301
9-Step DQN Final test average score : 500.0
```

10) 10-step DQN



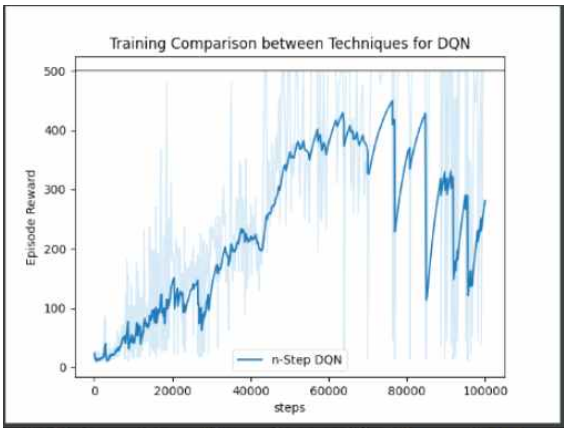

```
에피소드 : 967, 점수 : 500, 남은 Step : 1150
에피소드 : 968, 점수 : 500, 남은 Step : 650
에피소드 : 969, 점수 : 500, 남은 Step : 150
에피소드 : 970, 점수 : 500, 남은 Step : -350
10-Step DQN Final test average score : 500.0
```

11) 11-step DQN



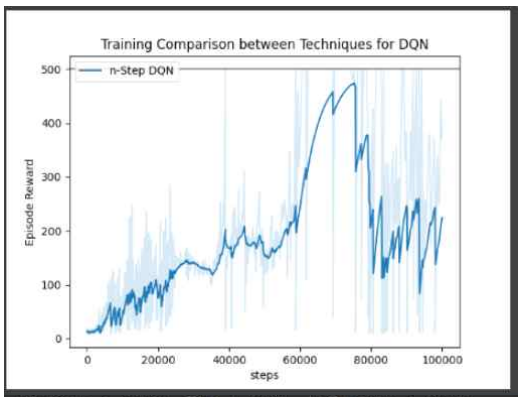

```
에피소드 : 1265, 점수 : 500, 남은 Step : 1376
에피소드 : 1266, 점수 : 500, 남은 Step : 876
에피소드 : 1267, 점수 : 500, 남은 Step : 376
에피소드 : 1268, 점수 : 500, 남은 Step : -124
11-Step DQN Final test average score : 500.0

Process finished with exit code 0
```

12) 12-step DQN



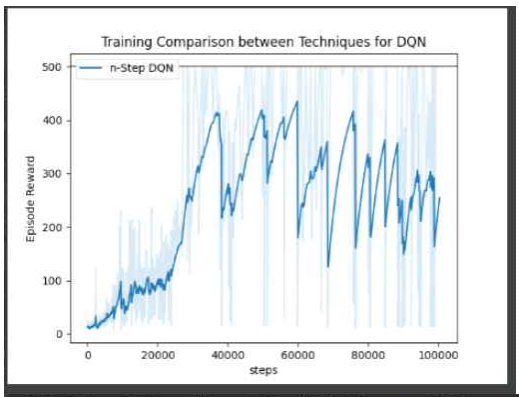

```
에피소드 : 886, 점수 : 500, 남은 Step : 1031
에피소드 : 887, 점수 : 39, 남은 Step : 992
에피소드 : 888, 점수 : 500, 남은 Step : 492
에피소드 : 889, 점수 : 500, 남은 Step : -8
12-Step DQN Final test average score : 307.8

Process finished with exit code 0
```

13) 13-step DQN



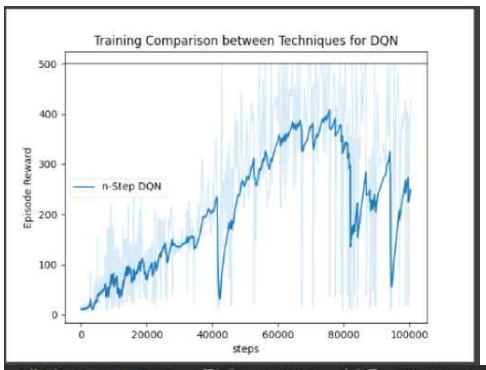

```
에피소드 : 1013, 점수 : 246, 남은 Step : 1060
에피소드 : 1014, 점수 : 252, 남은 Step : 808
에피소드 : 1015, 점수 : 444, 남은 Step : 364
에피소드 : 1016, 점수 : 370, 남은 Step : -6
13-Step DQN Final test average score : 500.0

Process finished with exit code 0
```

14) 14-step DQN



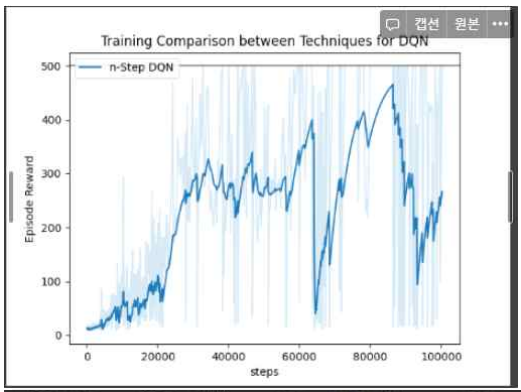

```
에피소드 : 852, 점수 : 20, 남은 Step : 1170
에피소드 : 853, 점수 : 500, 남은 Step : 670
에피소드 : 854, 점수 : 500, 남은 Step : 170
에피소드 : 855, 점수 : 500, 남은 Step : -330
14-Step DQN Final test average score : 500.0

Process finished with exit code 0
```

15) 15-step DQN




```
에피소드 : 956, 점수 : 391, 남은 Step : 455
에피소드 : 957, 점수 : 24, 남은 Step : 431
에피소드 : 958, 점수 : 20, 남은 Step : 411
에피소드 : 959, 점수 : 272, 남은 Step : 139
에피소드 : 960, 점수 : 429, 남은 Step : -290
15-Step DQN Final test average score : 207.0
```

16) 16-step DQN




```
에피소드 : 983, 점수 : 414, 남은 Step : 843
에피소드 : 984, 점수 : 500, 남은 Step : 343
에피소드 : 985, 점수 : 98, 남은 Step : 245
에피소드 : 986, 점수 : 500, 남은 Step : -255
16-Step DQN Final test average score : 500.0

Process finished with exit code 0
```