DQfD Implementation

1. 실행환경
운영체제 : window
언어 : python 3.8
라이브러리 : pytorch

state는 4가지 값으로 구성되어 있고, 환경에 가해져야 하는 action의 수는 2가지이다.
action은 0 혹은 1로 분류된다.

2. 기본 Hyperparameter 및 변수

```
class DQfDAgent(object):
    def __init__(self, env, use_per, n_episode):
        self.n_EPISODES = n_episode
        self.env = env
        self.use_per = use_per

        self.demo = get_demo_traj()
        self.demoStore = deque()
        self.democheck = defaultdict(list)
        self.mem = deque(maxlen=256)

        self.gamma = 0.95
        self.eGreedy = 0.01
        self.margin = 0.8
        self.nstep = 10

        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        self.predmodel = DQfDNetwork(4, 2).to(self.device)
        self.fixedQ = DQfDNetwork(4, 2).to(self.device)

        self.optimizer = optim.Adam(self.predmodel.parameters(), lr=0.00002, weight_decay=1e-5) ## For preventing overfitting (L2 regulazation)
```

여기에서 episode는 총 250개까지 실행이 가능하며, env는 앞서 언급한 CartPole-v1이다. demostore는 모든 demonstration data를 저장하기 위한 것이고, democheck는 episode 별로 demonstration data를 순서대로 저장하기 위한 것이다.

discount value인 gamma는 0.95로 설정했고 일정한 탐색을 유지할 수 있도록 하기 위해 eGreedy를 0.01로 설정해뒀다. margin은 large margin classification loss를 구하기 위해 정해 놓은 변수로 DQfD 논문에서는 margin을 0.8로 두었기에 본인도 0.8로 설정해두었다. 아울러, n-step loss를 적용할 때, n을 10으로 설정해뒀다. 논문에서도 n-step을 10으로 설정해뒀다. 또, 빠른 계산을 하기 위해 device 설정도 해뒀다.

논문에서 loss를 계산할 때 Double DQN 형식을 사용한다고 언급하여 predmodel과 fixedQ 두가지로 분리하여 학습을 진행하도록 했다.

여기서 가장 핵심은 optimizer이다. optimizer는 gradient discent할 때 사용하는 것이기도 하지만 논문에서 언급한 L2 regulazation loss를 빠르게 구하도록 도와주는 역할도 한다. 여기서 weight_decay는 λ_3이며 λ_3값은 논문에서 언급했던 것과 같이 10^{-5}로 설정해뒀다.

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q).$$

3. Network 설정

```
class DQfDNetwork(nn.Module):
    def __init__(self, in_size, out_size):
        super(DQfDNetwork, self).__init__()
        ## TODO
        self.relu = nn.ReLU()
        self.layer1 = nn.Linear(in_size, 50)
        #self.layer2 = nn.Linear(50, 30)
        self.outLayer = nn.Linear(50, out_size)

    def forward(self, x):
        ## TODO
        x = self.relu(self.layer1(x))
        #x = self.relu(self.layer2(x))
        return self.outLayer(x)
```

Q network을 복잡하게 구성하려 했지만 CartPole 환경은 state가 4개이고 action이 2개 밖에 존재하지 않는 환경이기에 적당히 작은 network를 구성했다.
입력층은 입력이 4, output이 50, 출력층은 입력이 50, output이 2로 설정될 것이다.

4. 기본 함수

4-1. get_action

```
def get_action(self, state):

    state = torch.Tensor([state]).to(self.device)
    output = self.predmodel(state).to(self.device)

    action = int(torch.argmax(output))
    if np.random.rand(1) < self.eGreedy:
        action = np.random.randint(0, 2)
        if action == 0: action = 1
        else: action = 0

    return action
```

이 함수는 state를 input으로 받아, Q-network으로부터 action을 추측하는데 사용된다. 여기에서 eGreedy는 0.01로 확정되어 있으며, 최종적으로 해당 state에서 가장 좋은 action 혹은 탐색하고자하는 action을 output으로 제공한다.

4-2. sampleDemo

```
def sampleDemo(self, numSize):
    batch = random.sample(self.demoStore, numSize)

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

    return states, actions, rewards, next_states, dones, info
```

이 함수는 mini batch size를 input으로 받아 ramdom으로 현재 보유하고 있는 demonstration data로부터 sample을 추출하는 역할을 한다.

4-3. Lfunction

```
def Lfunction(self, aE, a):
    return 0.0 if aE == a else 0.8
```

이 함수는 demonstration data를 사용해 large margin classification loss를 계산할 때 사용한다. input으로는 expert의 a와 현재의 a를 input으로 받는데, 만약 expert가 해당 state에서 행한 행동이 현재 하려는 행동과 같을 때 0을 return하고, 그렇지 않은 경우에는 0.8을 return 한다. 이 함수는 논문에서 언급했던 equation 중 $l(a_E, a)$에 해당한다.

$$J_E(Q) = \max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E)$$

4-4. get_demo_traj

```
def get_demo_traj():
    return np.load("./demo_traj_2.npy", allow_pickle=True)
```

이 함수는 교수님이 제공해주신 함수로 demonstration data를 numpy 형태로 받아오는 역할을 수행한다.

5. DQfD

5-1. Pre-train

```
def train(self):
    ###### 1. DO NOT MODIFY FOR TESTING ######
    test_mean_episode_reward = deque(maxlen=20)
    test_over_reward = False
    test_min_episode = np.inf
    ###### 1. DO NOT MODIFY FOR TESTING ######

    # Do pretrain
    self.fixedQ.load_state_dict(self.predmodel.state_dict())
    print("---------Pre-train Start--------")
    self.pretrain()
    print("---------Pre_train End----------")
    ## TODO
```

이 함수는 train으로 실질적인 train을 담당한다. 시작하기 앞서, test하기 위한 변수를 설정해뒀다. 그리고 fixedQ와 predmodel의 파라미터를 동일하게 만들어 놓고 pretrain 함수로 넘어간다.

```
def pretrain(self):
    ## Do pretrain for 1000 steps
    ## Check time
    start = time.time()

    ## Just store the demo data
    for epiNum, epi in enumerate(self.demo):
        for stepNum, step in enumerate(epi):
            state, action, reward, next_state, done = step
            self.democheck[epiNum].append([state, action, reward, next_state, done, (stepNum)])
            self.demoStore.append([state, action, reward, next_state, done, (epiNum, stepNum)])

    ## Pre-train 1000 times
    for loop in range(1001):
        self.update_preTrain(250)

        if loop % 100 == 0: print("{0} loop end".format(loop))

    print("-----Time : {0}".format(time.time()-start))
```

이 함수는 pretrain 함수이다. 실행하기 앞서 demonstration data를 각각 democheck와 demostore에 저장한다. democheck는 n-step learning을 위한 것이고 demoStore는 sampling을 위한 것이다.

이렇게 demonstration data를 저장하고 난 이후, 1000번의 pre-train 과정을 거치게 된다. 최대한 demonstration data로부터 좋은 value를 도출하여 최소한의 episode만으로 학습을 진행하기 위해 최대한 많은 mini batch를 실시했다. demonstration data의 크기가 400개 조금 넘었기 때문에 반이 넘는 250으로 mini batch size를 결정했다.

```python
def update_preTrain(self, size):
    for loop in range(size):
        ## Sampling
        sample = random.sample(self.demoStore, 1)

        state, action, reward, next_state, done, info = sample[0]

        state = torch.Tensor(np.array([state])).to(self.device)
        action = torch.Tensor(np.array([action])).to(self.device)
        action = action.type(torch.int64)
        next_state = torch.Tensor(np.array([next_state])).to(self.device)
## ----------
## one step
## ----------
predQ = self.predmodel(state)
predictedQ = torch.gather(predQ, 1, action.unsqueeze(-1))

expectQ = reward + self.fixedQ(next_state).max(1)[0]

lossOne = F.mse_loss(predictedQ, torch.Tensor([expectQ]))
## ----------
## n step
## ----------
epi, step = info

s, a, r, ns, d, i = zip(*self.democheck[epi][step:step + 10])
length = len(s)

nStepQ = 0.0
mul = 1.0
for i in range(length):
    nStepQ += mul * r[i]
    mul = mul * self.gamma
nStepQ += mul * self.fixedQ(torch.Tensor([ns[-1]])).max(1)[0]

lossN = F.mse_loss(predictedQ, torch.Tensor([nStepQ]))
## ------------
## expert loss
## ------------
aE = int(action)
maxValue = torch.tensor(-9999999999)

for a in range(2):
    a = torch.Tensor([[a]])
    a = a.type(torch.int64)
    value = torch.gather(predQ, 1, a) + self.Lfunction(aE, a)
    if maxValue < value[0]: maxValue = value[0]

aE = torch.Tensor([[aE]])
aE = aE.type(torch.int64)
expertQ = torch.gather(predQ, 1, aE)

lossE = F.mse_loss(maxValue, expertQ[0])
```

```python
## -------------
## apply loss function
## -------------
self.optimizer.zero_grad()
totalLoss = lossOne + lossN + lossE
totalLoss.backward()
self.optimizer.step()

if loop % 5 == 0: self.fixedQ.load_state_dict(self.predmodel.state_dict())
```

Algorithm 1 Deep Q-learning from Demonstrations.

Inputs: $\mathcal{D}^{replay}$: initialized with demonstration data set, θ: weights for initial behavior network (random), θ': weights for target network (random), τ: frequency at which to update target net, k: number of pre-training gradient updates
for steps $t \in \{1, 2, \ldots k\}$ **do**
  Sample a mini-batch of n transitions from $\mathcal{D}^{replay}$ with prioritization
  Calculate loss $J(Q)$ using target network
  Perform a gradient descent step to update θ
  if $t \bmod \tau = 0$ **then** $\theta' \leftarrow \theta$ **end if**
end for
for steps $t \in \{1, 2, \ldots\}$ **do**
  Sample action from behavior policy $a \sim \pi^{\epsilon Q_\theta}$
  Play action a and observe (s', r).
  Store (s, a, r, s') into $\mathcal{D}^{replay}$, overwriting oldest self-generated transition if over capacity
  Sample a mini-batch of n transitions from $\mathcal{D}^{replay}$ with prioritization
  Calculate loss $J(Q)$ using target network
  Perform a gradient descent step to update θ
  if $t \bmod \tau = 0$ **then** $\theta' \leftarrow \theta$ **end if**
  $s \leftarrow s'$
end for

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q).$$

optimizer

N-Step Return weight $\lambda_1 = 1.0$
Supervised loss weight $\lambda_2 = 1.0$.
L2 regularization weight $\lambda_3 = 10^{-5}$.
Expert margin $l(a_E, a) when a \neq a_E = 0.8$.

논문에서 사용한 parameter 값들

update_preTrain 함수가 시작하면 우선 for loop을 돈다. 이 for loop은 mini batch size 만큼 돈다. 매 루프 마다 아래와 같은 행동이 반복된다.

1. demonstration data 집합으로부터 한 가지의 데이터를 random으로 추출한다. (Priority 고려하지 않음)
2. demondtration data를 학습시키지 좋은 형태로 변환시킨다.
3. 1-step learning을 실시한다. 추출한 데이터 중 state, action reward, next state를 사용해 loss를 계산한다. 기존 DQN과 같은 방식이다.
4. n-step learning을 실시한다. 추출한 데이터를 사용하지만 info 변수에 저장되어 있는 episode 번호와 step 번호를 기반으로 n step 관련 정보를 도출해 loss를 계산한다. 기존 n-step DQN과 같은 방식이다.
5. large margin classification loss를 계산한다.
6. 모든 loss를 종합하여 Q network에 반영해준다.
7. 만약 일정 step이 지난 경우 fixedQ의 파라미터를 predmodel의 파라미터와 같게 업데이트 시켜준다.

1000 loop가 종료되면 모든 pre-train 작업이 끝난 것이다.

5-2. Real time train

```
def train(self):
    ###### 1. DO NOT MODIFY FOR TESTING ######
    test_mean_episode_reward = deque(maxlen=20)
    test_over_reward = False
    test_min_episode = np.inf
    ###### 1. DO NOT MODIFY FOR TESTING ######

    # Do pretrain
    self.fixedQ.load_state_dict(self.predmodel.state_dict())
    print("---------Pre-train Start--------")
    self.pretrain()
    print("---------Pre_train End----------")
    ## TODO

    ## draw graph
    number = []
    meanScore = []
    z = 0
for e in range(self.n_EPISODES):
    ########## 2. DO NOT MODIFY FOR TESTING ##########
    test_episode_reward = 0
    ########## 2. DO NOT MODIFY FOR TESTING  ##########

    ## TODO
    done = False
    state = self.env.reset()
    #self.env.render()
    mem = deque() ## For n step
    totalmem = deque(maxlen=200)

    score = 0
    count = 0
while not done:
    ## TODO
    action = self.get_action(state)

    ## TODO
    next_state, reward, done, _ = self.env.step(action)
    ########## 3. DO NOT MODIFY FOR TESTING ##########
    test_episode_reward += reward
    ########## 3. DO NOT MODIFY FOR TESTING  ##########

    ## Store memory
    mem.append([state, action, reward, next_state, done])
    totalmem.append([state, action, reward, next_state, done])
```

```python
## Update as n step
if len(mem) == 10:
    count += 1
    ## ------------
    ## one step
    ## ------------
    state = mem[0][0]
    action = mem[0][1]
    reward = mem[0][2]
    ns = mem[0][3]
    done = mem[0][4]

    state = torch.Tensor(np.array([state])).to(self.device)
    action = torch.Tensor(np.array([action])).to(self.device)
    action = action.type(torch.int64)
    ns = torch.Tensor(np.array([ns])).to(self.device)

    predQ = self.predmodel(state)
    predictedQ = torch.gather(predQ, 1, action.unsqueeze(-1))

    expectQ = reward + self.fixedQ(ns).max(1)[0]

    lossOne = F.mse_loss(predictedQ, torch.Tensor([expectQ]))

## ------------
## n step
## ------------
nStepQ = 0.0
mul = 1.0
for i in range(10):
    nStepQ += mul * mem[i][2]
    mul = mul * self.gamma
nStepQ += mul * self.fixedQ(torch.Tensor([mem[-1][3]])).max(1)[0]

lossN = F.mse_loss(predictedQ, torch.Tensor([nStepQ]))

## ------------
## expert loss
## ------------
aE = int(action)
maxValue = torch.tensor(-9999999999)

for a in range(2):
    a = torch.Tensor([[a]])
    a = a.type(torch.int64)
    value = torch.gather(predQ, 1, a) + self.Lfunction(aE, a)
    if maxValue < value[0]: maxValue = value[0]

aE = torch.Tensor([[aE]])
aE = aE.type(torch.int64)
expertQ = torch.gather(predQ, 1, aE)

    self.optimizer.zero_grad()
    totalLoss = lossOne + lossN + lossE * 0
    totalLoss.backward()
    self.optimizer.step()

    self.update_preTrain(20)

    count +=1
    if count % 5 == 0: self.fixedQ.load_state_dict(self.predmodel.state_dict())
    mem.popleft()

score += reward
```

Algorithm 1 Deep Q-learning from Demonstrations.

Inputs: $\mathcal{D}^{replay}$: initialized with demonstration data set, θ: weights for initial behavior network (random), θ': weights for target network (random), τ: frequency at which to update target net, k: number of pre-training gradient updates
for steps $t \in \{1, 2, \dots k\}$ **do**
  Sample a mini-batch of n transitions from $\mathcal{D}^{replay}$ with prioritization
  Calculate loss $J(Q)$ using target network
  Perform a gradient descent step to update θ
  if $t \bmod \tau = 0$ **then** $\theta' \leftarrow \theta$ **end if**
end for
for steps $t \in \{1, 2, \dots\}$ **do**
  Sample action from behavior policy $a \sim \pi^{\epsilon Q_\theta}$
  Play action a and observe (s', r).
  Store (s, a, r, s') into $\mathcal{D}^{replay}$, overwriting oldest self-generated transition if over capacity
  Sample a mini-batch of n transitions from $\mathcal{D}^{replay}$ with prioritization
  Calculate loss $J(Q)$ using target network
  Perform a gradient descent step to update θ
  if $t \bmod \tau = 0$ **then** $\theta' \leftarrow \theta$ **end if**
  $s \leftarrow s'$
end for

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q).$$

optimizer

N-Step Return weight $\lambda_1 = 1.0$
Supervised loss weight $\lambda_2 = 1.0$.
L2 regularization weight $\lambda_3 = 10^{-5}$.
Expert margin $l(a_E, a) when a \neq a_E = 0.8$.

versus agent data. All the losses are applied to the demonstration data in both phases, while the supervised loss is not applied to self-generated data ($\lambda_2 = 0$).

demonstration data를 또 학습시키는 코드, 이미 충분히 학습시켰기 때문에, mini batch size를 20으로 설정해둔다.

Pre-train 함수가 종료된 이후, agent는 정상적인 train 활동에 들어간다. agent는 총 250개의 에피소드를 실행할 수 있으며, 도중 목적을 달성하면 종료한다. 매 에피소드 별 루프 행동을 아래 설명과 같다.

1. 한 에피소드에 필요한 변수들을 다 초기화시킨다.
2. 에피소드를 실행한다.
3. 현재 state로부터 predmodel이 추측한 action을 사용해 환경에 계속 적용시킨다. 이 과정에서 모든 n-step learning을 위해 state-action 정보를 저장해둔다.
4. 환경에 action을 적용시키는 도중 저장해둔 state-action 정보의 개수가 n-step의 n과 같아졌을 때 실제 train을 진행한다.

4-1. old real time data를 mini batch하여 1-step Q-learning을 실시한다. (pre-train의 1-step Q-learning과 같다.)

4-2. 현재까지 쌓아둔 real time data를 사용해 n-step Q-learning을 실시한다. (pre-train의 n-step Q-learning과 같다.)

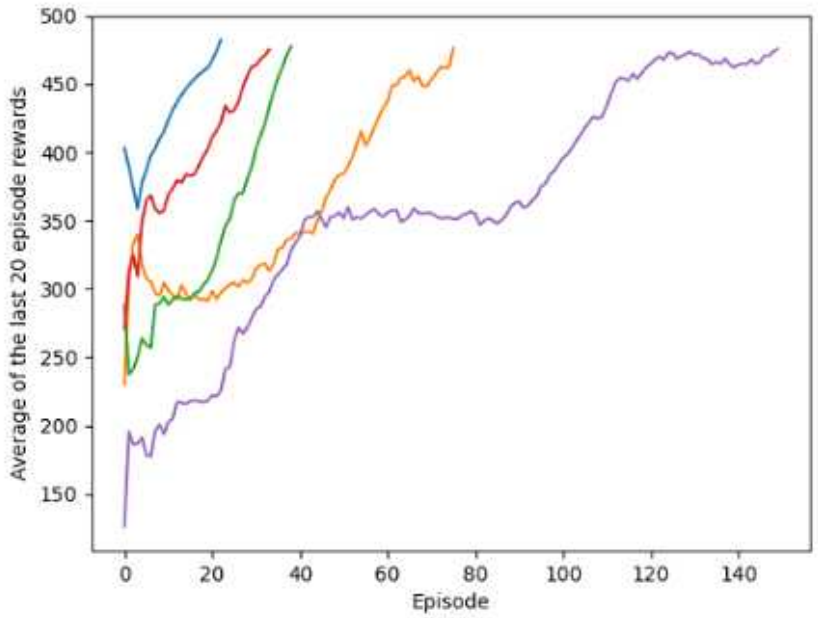
4-3. large margin classification loss를 계산한다. 그러나 논문에서 언급했듯이, self generated data에 대해서는 large margin classification loss를 적용하지 않는다. 따라서, 최종적으로 0을 곱해줬다.

5. real time data에 대한 train이 끝난 이후, demonstration data에 대한 train도 실시한다. pre-train 과정에서 demonstration data에 대한 train이 충분히 이뤄졌다고 판단하여 mini batch size를 20으로 설정해뒀다.
6. 만약 일정 step이 지난 경우 fixedQ의 파라미터를 predmodel의 파라미터와 같게 업데이트 시켜준다.
7. 일정 목적을 달성했을 경우 train 함수는 종료되며, test 관련 정보를 return한다.

6. Result

실험을 여러 차례 진행했지만, 가장 마지막에 했던 4번의 실험 결과이다.

1 Try :




```
END train function
Minimum number of episodes for 475 : 22, Average of 20 episodes reward : 482.4
```

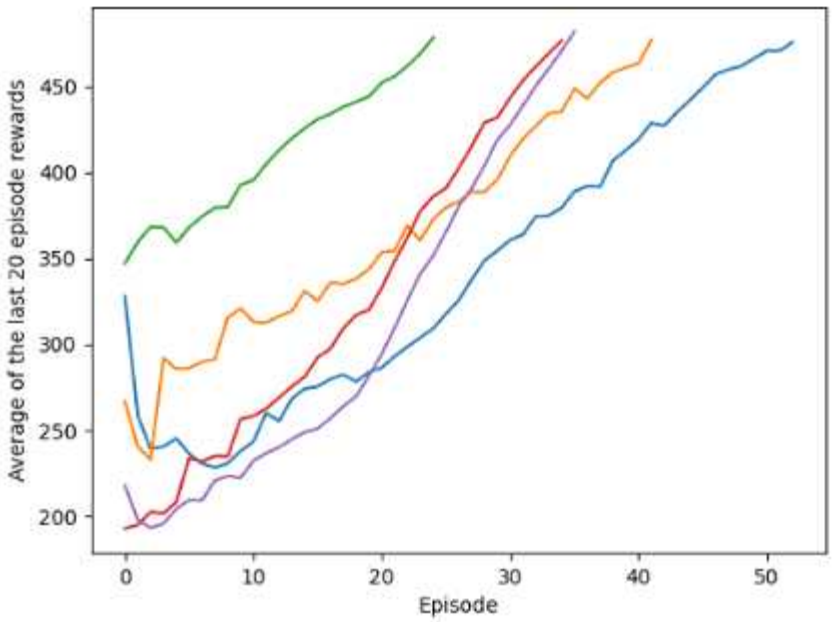
```
END train function
Minimum number of episodes for 475 : 75, Average of 20 episodes reward : 476.5
```

```
END train function
Minimum number of episodes for 475 : 38, Average of 20 episodes reward : 477.5
```

```
END train function
Minimum number of episodes for 475 : 33, Average of 20 episodes reward : 475.15
```

```
============================
END train function
Minimum number of episodes for 475 : 149, Average of 20 episodes reward : 475.9
Final end time : 3779.6840391159058
END main function
Average number of episodes for 475 : 63.4
```

2 Try :




```
END train function
Minimum number of episodes for 475 : 52, Average of 20 episodes reward : 475.65
--------Pre-train Start--------
```

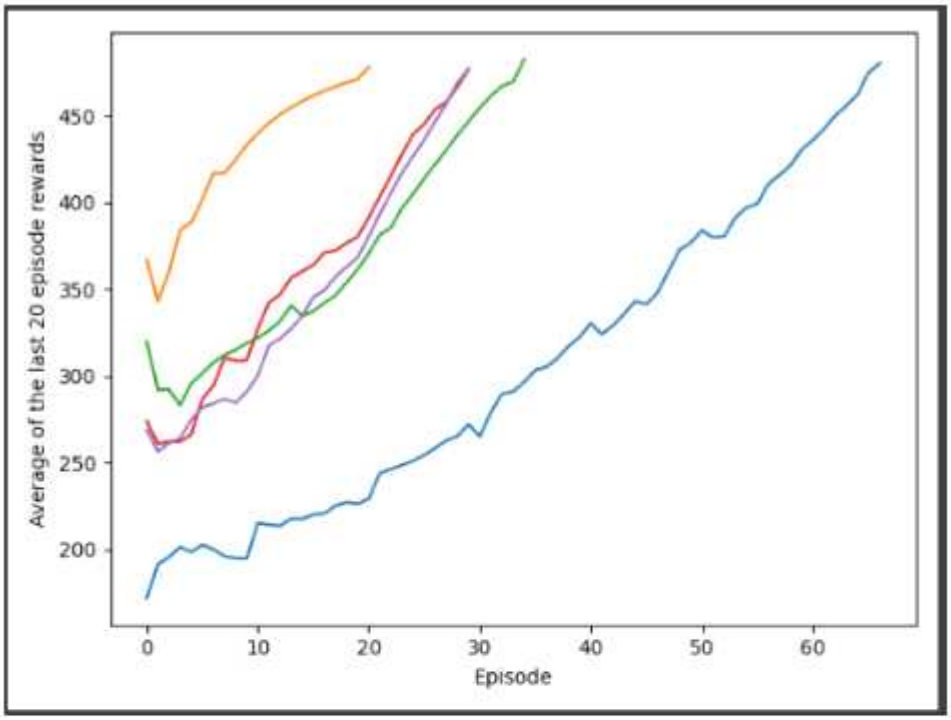
```
END train function
Minimum number of episodes for 475 : 41, Average of 20 episodes reward : 477.1
```

```
END train function
Minimum number of episodes for 475 : 24, Average of 20 episodes reward : 478.25
```

```
END train function
Minimum number of episodes for 475 : 34, Average of 20 episodes reward : 476.45
```

```
END train function
Minimum number of episodes for 475 : 35, Average of 20 episodes reward : 481.9
Final end time : 2785.773815870285
END main function
Average number of episodes for 475 : 37.2
```

3 Try :




```
END train function
Minimum number of episodes for 475 : 66, Average of 20 episodes reward : 480.3
```

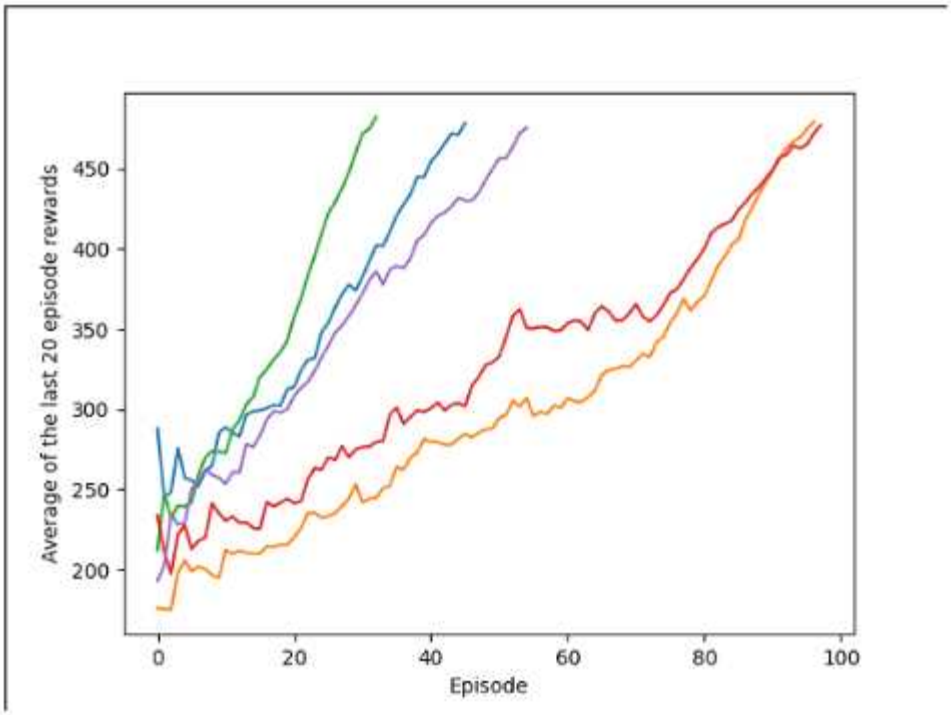
```
END train function
Minimum number of episodes for 475 : 28, Average of 20 episodes reward : 478.15
```

```
END train function
Minimum number of episodes for 475 : 34, Average of 20 episodes reward : 482.55
```

```
END train function
Minimum number of episodes for 475 : 29, Average of 20 episodes reward : 476.9
```

```
END train function
Minimum number of episodes for 475 : 29, Average of 20 episodes reward : 477.2
Final end time : 4441.148767335892
END main function
Average number of episodes for 475 : 35.6
```

4 Try :




```
END train function
Minimum number of episodes for 475 : 45, Average of 20 episodes reward : 478.5
```

```
END train function
Minimum number of episodes for 475 : 96, Average of 20 episodes reward : 479.65
```

```
END train function
Minimum number of episodes for 475 : 32, Average of 20 episodes reward : 482.35
```

7. 실행창

7.1 Pre-train 시작과 끝

```
--------Pre-train Start--------
0 loop end
100 loop end
200 loop end
300 loop end
400 loop end
500 loop end
600 loop end
700 loop end
800 loop end
900 loop end
1000 loop end
-----Time : 242.7950894832611
--------Pre_train End---------
```

7.2 Real time data 이용한 train (per episode)

```
============================
------Now episode : 44
------Now score : 500.0
============================
```

7.3 Result of one seed

```
============================
END train function
Minimum number of episodes for 475 : 45, Average of 20 episodes reward : 478.5
```