# Computer Network Report

120210211 이지환

## 1. Requirements

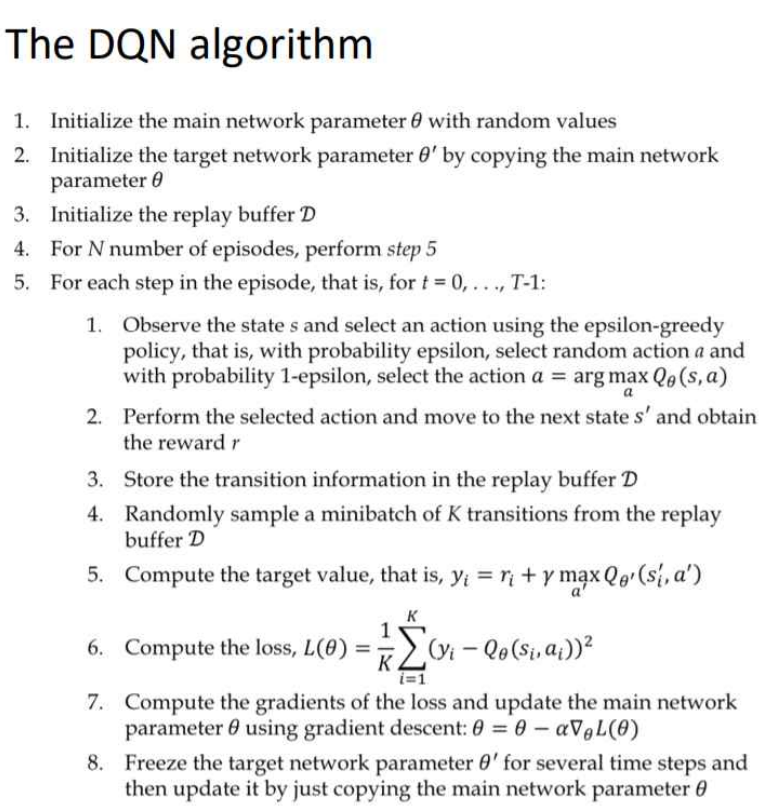
pytorch:1.8.1 CUDA:10.2, gym library
(DQN과 Dueling DQN의 경우, gpu로 학습했다.)

## 2. Algorithm 설명 및 코드 구현 설명

"Mountain Car Climbing" solving problem에 DQN,Dueling DQN, Q-Learning model-free method algorithm들을 적용해보았다.

### [1] DQN

DQN 알고리즘은 기존 TD(SARSA,Q-learning) 알고리즘에서 매번 q table을 업데이트를 해야하는 한계점을 개선하기 위해 제안된 알고리즘이다. q-table을 사용하는 대신, neural network를 통해 방문하지 않은 state에 대해서도 q value를 예측하도록 하는데에서 기존 알고리즘보다 매우 효율적인 방법이다.
DQN 알고리즘의 흐름도는 강의자료에 제시된 pseudo code를 따라 구현하였다.



The DQN algorithm

1. Initialize the main network parameter $\theta$ with random values
2. Initialize the target network parameter $\theta'$ by copying the main network parameter $\theta$
3. Initialize the replay buffer $\mathcal{D}$
4. For $N$ number of episodes, perform *step 5*
5. For each step in the episode, that is, for $t = 0, \ldots, T\text{-}1$:
    1. Observe the state $s$ and select an action using the epsilon-greedy policy, that is, with probability epsilon, select random action $a$ and with probability 1-epsilon, select the action $a = \arg\max_a Q_\theta(s,a)$
    2. Perform the selected action and move to the next state $s'$ and obtain the reward $r$
    3. Store the transition information in the replay buffer $\mathcal{D}$
    4. Randomly sample a minibatch of $K$ transitions from the replay buffer $\mathcal{D}$
    5. Compute the target value, that is, $y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a')$
    6. Compute the loss, $L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2$
    7. Compute the gradients of the loss and update the main network parameter $\theta$ using gradient descent: $\theta = \theta - \alpha\nabla_\theta L(\theta)$
    8. Freeze the target network parameter $\theta'$ for several time steps and then update it by just copying the main network parameter $\theta$

DQN 네트워크는 다음과 같이 구성하였다.

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

        self.fc1 = nn.Linear(input_size, 128) ## state value(position,velocity) 층 2개
        self.fc2 = nn.Linear(128, 128)
        self.fc3=nn.Linear(128,64)
        self.fc4 = nn.Linear(64, output_size) ## action 후보 층 3개

    def forward(self, x):

        x = x.to(device)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x= F.relu(self.fc3(x))
        x = self.fc4(x)

        return x

    def sample_action(self, obs, epsilon):

        out = self.forward(obs)
        coin = random.random()

        if coin < epsilon:
            return random.randint(0, 1) ## exploration
        else:
            return out.argmax().item() ## exploitation
```

네트워크는 총 FC(fully-connected) layer가 4개로 이루어져 있으며, 첫 번째 fc층에서의 input_size는 mountain car의 state value (position,velocity) 2개가 input으로 들어간다. 그리고 hidden-layer로 (128,128) layer와, (128,64) layer 총 두 개를 구성해 보았다.
그리고 exploration과 explotation을 적절히 구성하기위해 강의자료에 있는 (ex016_part1.py) 코드를 참고해 random value가 epsilon보다 작을경우에는 exploration을 하도록, 클 경우에는 explotation을 하도록 하는 함수를 작성하였다.

그리고 replay-buffer를 만들기위해 'ReplayBuffer'라는 별도의 클래스를 생성하였다.

```python
class ReplayBuffer():
    def __init__(self):
        self.buffer =deque([],maxlen=BUFFER_LIMIT)  # double-ended queue

    def push(self, *args): ## deque --> append
        self.buffer.append(Transition(*args))

    def sample(self, n):
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, next_state, done = transition
            s_lst.append(s);a_lst.append([a]);r_lst.append([r]);s_prime_lst.append(next_state);done_mask_lst.append([done

        def list_to_tensor_float(lst):
            return torch.tensor(lst,dtype=torch.float).to(device)

        def list_to_tensor(lst):
            return torch.tensor(lst).to(device)

        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst=list_to_tensor_float(s_lst),list_to_tensor(a_lst),list_to_tensor(

        return s_lst,a_lst,r_lst,s_prime_lst,done_mask_lst

    def size(self):
        return len(self.buffer)
```

buffer에 저장할 자료구조는 deque를 사용하였다. deque는 양방향 큐로 데이터를 넣고 빼기에 용이해 deque를 사용하여 buffer를 구성하도록 했다. push함수는 나중에 데이터를 넣을 때 사용하는 함수이다. Replay buffer에서 sample을 가져오기 위해, sample 함수를 작성하였는데, 현재 state,action,reward,next_state, done(terminal 여부) 순서대로 데이터를 sampling하도록 작성하였다. hyperparameter만큼 설정한 mini_batch만큼 가져오고, 데이터의 타입이 list이기 때문에, torch의 tensor형으로 변경하기 위한 함수도 추가로 정의하였다.

```
def train(q, q_target, memory, optimizer):

    for i in range(200):
        s, a, r, s_prime, done_mask = memory.sample(BATCH_SIZE)

        q_out = q(s)
        q_a = q_out.gather(1, a)
        max_q_prime = q_target(s_prime).max(1)[0].unsqueeze(1) ## target network
        target = r + GAMMA * max_q_prime * done_mask
        loss = F.mse_loss(q_a, target)

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

train 함수는 network가 학습을 진행하면서 파라미터를 계속해서 업데이트하여, loss를 줄이도록 코드를 작성한 부분이다. 이 과정들은 regression task이기 때문에, loss function으로 더 적합한 MSE loss를 사용하였다.

- **하이퍼 파라미터**
  learning rate=0.005
  gamma=0.98
  buffer_limit=50000
  batch size=32
  optimizer=adm optimizer

네트워크 학습에 적용한 하이퍼파라미터는 위와 같다.

총 episode의 개수는 10000개이며, 강의자료 예제 코드와 동일하게 epilson의 value를 학습이 진행될수록 줄어들도록 코드를 작성하였다. 강의자료 코드에서는 초기 확률값을 8%로 설정했으나, 7%로 설정했을때가 성능이 더 좋아 7%로 변경하였다. 즉 초반의 exploration 확률은 7% 였지만 학습이 진행될수록 exploration의 정도를 1%에 가깝게 학습하였다.
그리고 buffer memory size가 2000개가 넘으면 학습이 진행되도록 하였다.
episode의 개수가 설정한 num_traj(여기에서는 20개로 설정) 나눈 나머지가 0이 되었을때는 q(main network)의 parameter를 q_target network의 parameter에 복사가 되도록 구성하였다. 그리고 iteration과 평균 reward를 출력하여 학습이 잘 진행이 되는지도 확인해 보았다.

그래프를 출력하기 위해 matplot 라이브러리를 활용하여 그래프를 출력했다. iteration마다의 reward의 그래프를 만들어 iteration마다의 reward 변화정도를 한눈에 볼 수 있다.

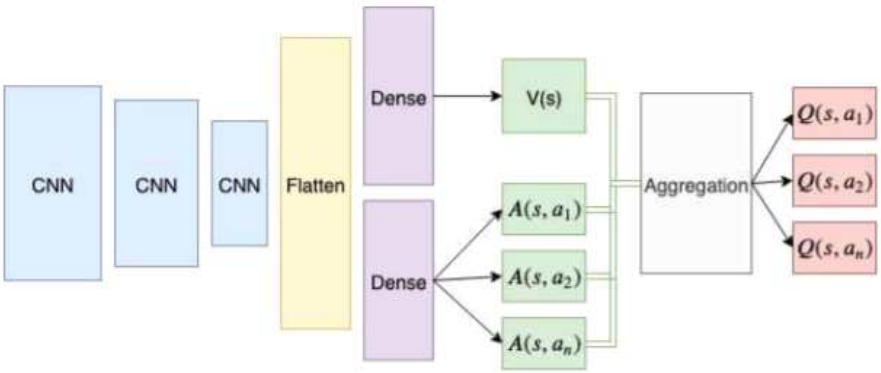
```
q = DQN(input_size, output_size).to(device)
q_target = DQN(input_size, output_size).to(device)

q_target.load_state_dict(q.state_dict()) ## 모델 불러오기
memory = ReplayBuffer()

optimizer = optim.Adam(q.parameters(), lr=LR)

for n_epi in range(10000): ## iteration 횟수와 관련
    epsilon = max(0.01, 0.07 - 0.01 * (n_epi / 200))  # Linear annealing from 7% to 1%
    s = env.reset()
    done = False

    while not done:
        a = q.sample_action(torch.from_numpy(s).float(), epsilon)
        s_prime, r, done, info = env.step(a)  ## state, reward, terminal state (1? NO, 0? Yes), probability
        done_mask = 0.0 if done else 1.0
        memory.push(s, a, r / 100.0, s_prime, done_mask)
        s = s_prime

        score += r
        if done:
            break

    if memory.size() > 2000: ### memeory size가 2000 넘으면 학습시작
        train(q, q_target, memory, optimizer)

    if n_epi % num_traj == 0 and n_epi != 0:
        iteration+=1
        q_target.load_state_dict(q.state_dict()) ## 학습된 모델 불러오기
        print("Iteration:{}, Reward : {:.1f}".format(iteration,score / num_traj))

        reward_list.append(score/num_traj)
        score = 0.0

env.close()

return reward_list
```



```
if __name__ == '__main__':

    mean_return_list=start_train()
    plt.plot(mean_return_list)
    plt.xlabel('Iteration')
    plt.ylabel('Reward')
    plt.title('DQN MountainCar-v0')
    plt.savefig('MountainCar_dqn_result.png', format='png', dpi=300)
```

[2] Dueling DQN

dueling dqn은 Q function과 value function간의 차이를 계산한다.(Advantage Function) 즉 식은 A(s,a)=Q(s,a)-v(s)이다. Dueling DQN의 가장큰 장점은 매 timestep마다의 value를 계산할 필요가 없다는 장점이 있다.
DQN과의 알고리즘과 매우 비슷하나, 코드에서 달라진 점은 모델이 2개가 생성이 된다는 점이다. (V network, A network)
코드 설명은 DQN network에서 달라진 부분만 설명하겠다.




강의 자료에서는 cnn network와 fully-connected를 예시로든 네트워크지만, 코드를 구현할 때 별도의 cnn network 없이 fully-connected layer만으로 네트워크를 구성하였다.

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

        super(DuelingQnet, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc_value = nn.Linear(128, 64)
        self.fc_adv = nn.Linear(128, 64)
        self.value = nn.Linear(64, 1)
        self.adv = nn.Linear(64, output_size)

    def forward(self, x):

        x = x.to(device)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        v = F.relu(self.fc_value(x))
        a = F.relu(self.fc_adv(x))
        v = self.value(v)
        a = self.adv(a)
        a_avg = torch.mean(a)
        q = v + a - a_avg
        return q

    def sample_action(self, obs, epsilon):
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.randint(0, 1)
        else:
            return out.argmax().item()
```

- Value Network 구조
  FC1(2,128) --> FC(128,128) --> FC(128,64) --> FC(64,1)

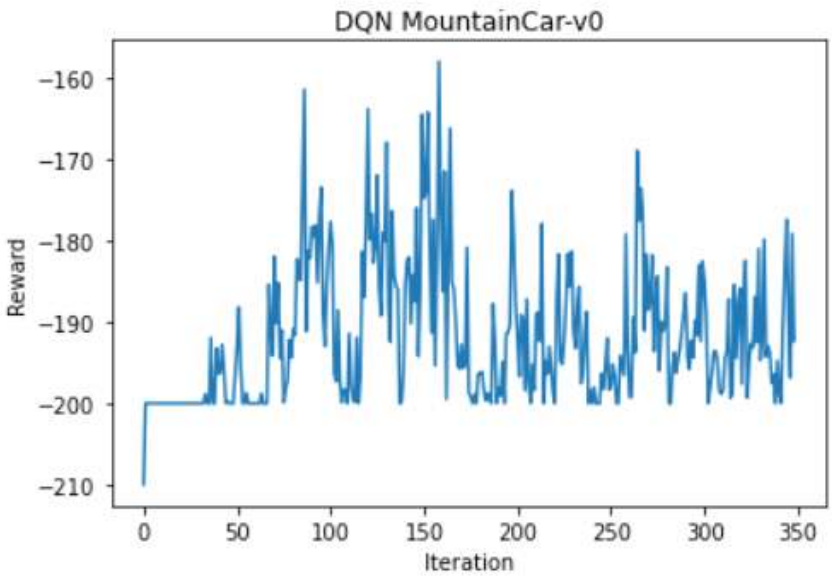
- Advantage Network 구조
  FC1(2,128) --> FC(128,128) --> FC(128,64) --> FC(64,3)

최종적으로 return되는 값은 value+advantage-advantage 평균값이다.

**성능 비교**

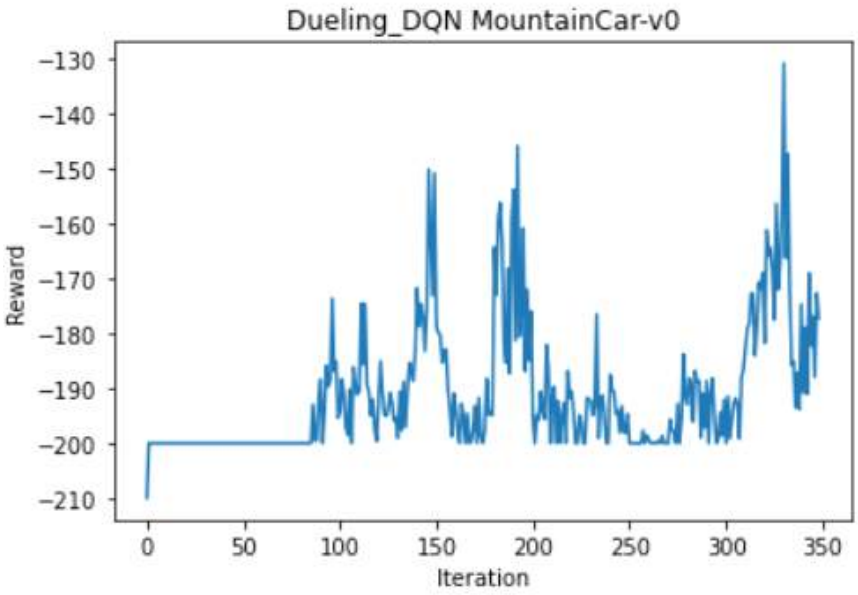
DQN과 Dueling DQN을 동일한 서버 환경(gpu)과 동일한 iteration으로 실험한 결과 다음과 같다.

- DQN




Mountaincar의 timestep은 200이기 때문에, timstep이 진행되는 동안 목표 position에 도달하지 못하면, timestep마다 reward -1이 더해져 -200이다. 즉 -200을 넘은 순간 목표한 reward에 도달한 것으로 판단할 수 있다. DQN의 경우에는 iteration이 반복될때마다 reward 값이 들쑥날쑥한데, 대체적으로 iteration마다 goal state를 도달했음을 확인하였다.

- Dueling DQN




Dueling DQN의 경우, iteration 80정도까지는 goal state에 도달하지 못했다가, iteration 80이 넘은 뒤로는 goal state에 꾸준히 도달함을 확인 할 수 있었다.

그래프를 통해 Dueling DQN은 일정학습이 반복되면 goal state에 도달하고, DQN은 Dueling DQN보다 좀더 많이 goal state에 도달함을 알 수 있다.

추가로 policy gradient network도 시도해보았으나, iteration마다 goal state에 도달하지 못하였다. (동일하게 -200 reward를 출력했다.)

- Q learning

추가적으로 q-learning에서의 학습도 진행해보았다.
q-learning의 경우에는 대표적인 off-policy 알고리즘으로, epsilon 값에 따라서 일정확률로는 greedy policy를 택하고 Q-value를 업데이트 할 때에는 greedy policy를 선택하는 방법이다. (policy가 2개)

알고리즘 자체는 강의자료 pseudo code를 작성하면, 어렵지 않게 구현할 수 있다.

## Q-Learning: The algorithm

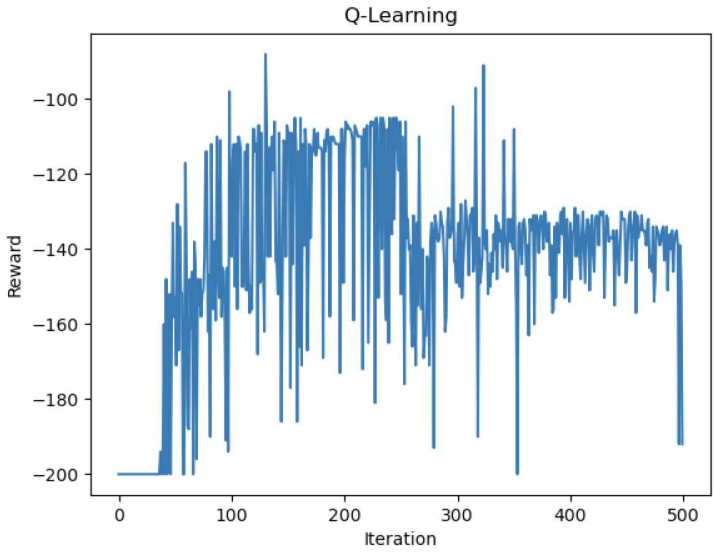
- Algorithm

1. Initialize a Q function $Q(s, a)$ with random values
2. For each episode:
    1. Initialize state $s$
    2. For each step in the episode:
        1. Extract a policy from $Q(s, a)$ and select an action $a$ to perform in state $s$
        2. Perform the action $a$, move to the next state $s'$, and observe the reward $r$
        3. Update the Q value as
        $Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s'a') - Q(s,a)\right)$
        4. Update $s = s'$ (update the next state $s'$ to the current state s)
        5. If $s$ is not a terminal state, repeat *steps 1* to *5*

```python
def run_episode(model, env, eps, gamma):
    observation = env.reset()
    done = False
    totalreward = 0
    num_traj = 0
    while not done and num_traj < 10000:
        action = model.sample_action(observation, eps)
        prev_observation = observation
        observation, reward, done, info = env.step(action)
        # update the model
        next = model.predict(observation)
        G = reward + gamma*np.max(next[0])
        model.update(prev_observation, action, G)

        totalreward += reward
        num_traj += 1

    return totalreward


def main(show_plots=True):
    env = gym.make('MountainCar-v0')
    ft = FeatureTransformer(env)
    model = Model(env, ft, "constant")
    gamma = 0.99



    N = 500 # iteration 500
    totalrewards = np.empty(N)
    for n in range(N):
        eps = 0.1*(0.97**n)

        totalreward = run_episode(model, env, eps,
        totalrewards[n] = totalreward
        if (n + 1) % 10 == 0:
            print("Iteration:", n+1, "total reward:",
```

초기 value는 0으로 설정하고, action을 선택하기위해 매 state마다의 action을 취하고, 가장 큰 state-action에서 가장큰 value값(reward)을 업데이트하도록 하였다. 위에서 DQN과 Dueling DQN과 동일하게, iteration이 반복될수록 epsilon value값에 0.97 제곱을 곱해 exploration을 줄이도록 학습하였다.

- Q-Learning 성능




q-learning은 매번 state와 마다의 action의 Q-value를 계산해야 하는 비효율성이 있지만, mountaincar에서의 성능은 neural network를 사용하는 것보다, q-learning이 좋은 것을 확인할 수 있었다. 그 이유는 Q-learning이 관찰 상태에서 취할 수 있는 모든 q값을 학습하는 방법을 사용해서 풀어내기 때문이라고 볼 수 있다.

3. Code Reference

[1] pytorch tutorial
https://tutorials.pytorch.kr/intermediate/reinforcement_q_learning.html

[2] 강의 자료 코드
ex016.py, ex016-part1.py

[3] Q learning
https://github.com/lazyprogrammer/machine_learning_examples