Summary Report [Week 15]. DQN

20190106 KimByungjun (김병준)

● Exercises

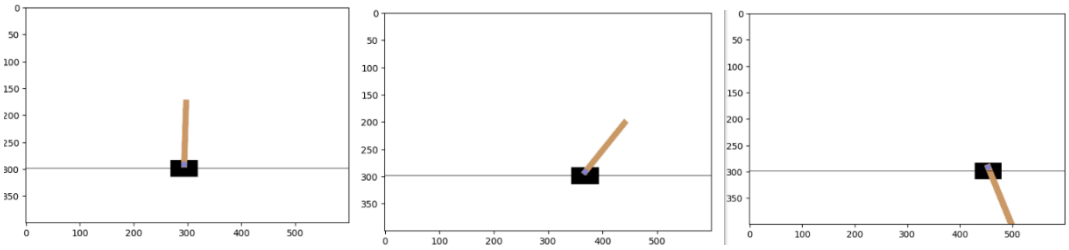
이번 Exercise 에선 Deep Q-Network(DQN)을 Cartpole game 에 적용하여, 높은 점수를 받도록 training 을 시켜보고자 한다. Cartpole 에서 고득점을 위해서는 밀대가 좌우로 움직일 때, cart 또한 축을 따라 좌우로 움직여 밀대가 최대한 오랫동안 쓰러지지 않도록 해야 한다. 이와 관련된 모든 환경부터 알아보자.

[Fig 1]와 같이 State space 는 cart 의 위치(0), cart 의 속도(1), pole 의 각(2), pole 의 각속도(3) 총 4 가지 상태로 표현되고, action space 의 경우, cart 를 왼쪽으로 밀 때 0, 오른쪽으로 밀 때 1 이 되도록 action 이 정의되었다. Reinforcement learning(RL)에서 핵심인 reward function 은, 각 time step 마다 +1 의 보상을 주고, pole 이 중심으로부터 2.4 unit 보다 많이 기울어져 있으면 해당 episode 는 terminate 된다. 즉, pole 이 쓰러지지 않고 오래 버텨야 더 많은 reward 를 해당 episode 에서 받는다.

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Num	Action
0	Push cart to the left
1	Push cart to the right

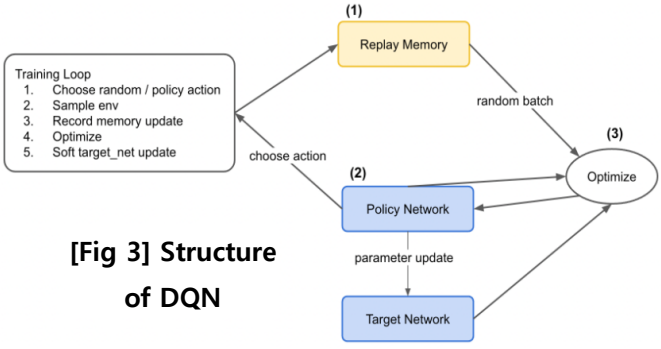
[Fig 1] Cartpole state space(left) and action state(right)



[Fig 2] Movement of cartpole system in random actions

위 CartPole environment 에서 100 회의 random action 을 부여할 때 cart 와 pole 의 움직임을 시각화 해보자. 100 회의 frame 모두를 담을 수 없어, representative 한 frame 만 첨부하자면, **[Fig 2]**와 같이 pole 은 쓰러져 가지만, randomized 한 action 에 의해 cart 는 pole 의 쓰러짐을 막지 못하여 pole 이 쓰러져버렸다.

CartPole 이 학습할 수 있게 DQN 을 구현해보자. **[Fig 3]**은 DQN 구조이다. DQN 은 policy network 와 target network 를 가지며, policy network 가 우리가 train 할 대상이며 target network 는 정답의 기준이 되는 network 이다. 중요한 점으로, target network 와 policy network 가 단일 parameter 에 의해 결정이 되는 구조라면, 해당 parameter 는 학습을 통해 매 step 마다 값이 바뀌며, 이는 학습이 unstable 하게 이루어질 수 있다. 이를 방지하기 위해, target network 에서는 policy network 가 학습되는 동안 값을 유지하고 있다가, 일정 주기마다 target network 를 update 한다.



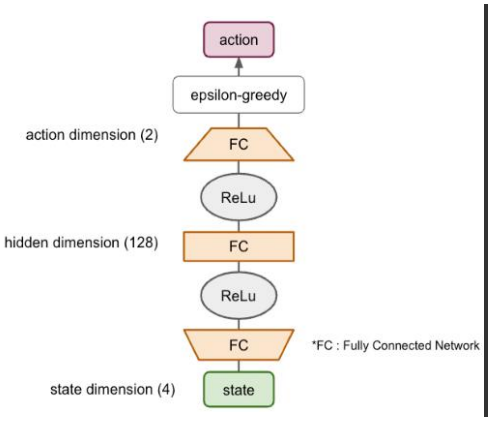
[Fig 3] Structure of DQN

학습 과정 동안, policy network 에서 sampling 된 actions 은 replay memory 라는 곳에 저장된다. Replay memory 에는 학습 중에 주어지는 state, action 등 experience 를 저장하고 나중에 network update 에 사용된다. 이 때, random sampling 에 의해 DQN 의 correlation 을 완화시키고 성능을 향상시킨다. 이는 ReplayBuffer class 를 통해 정의되는데, **[Fig 4]**와 같이 구현된다. put method 는 state, action, reward, next_state, done_make 의 정보가 담긴 transition 을 받아 buffer 에 추가하고, sample method 는 32 개의 item 을 sampling 해 tensor 형으로 전환 후 mini batch 를 생성한다.

```
class ReplayBuffer():
    def __init__(self):
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition):
        self.buffer.append(transition)

    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, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])

        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)

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

[Fig 4] ReplayBuffer class

이제 Q-network 을 정의해보자. 주어진 state 에 대해 각기 다른 action 에 대응되는 Q-value 를 반환하는 neural network 구조를 띄고 있다(**[Fig 5]**). Q-value 는 agent 가 특정 action 을 선택했을 때 받을 수 있는 누적 reward 기댓값으로 해석할 수 있으며, 가장 높은 Q-value 는 즉 최대 reward 를 기대할 수 있는 action 를 선택하는 지표가 된다. **[Fig 5]**의 구조를 가지는 QNet class 는 **[Fig 6]**처럼 구현한다. 더 견고한 network 를 만들기 위해, 작은 양수 ϵ 확률로 random 한 action 을 취하는 epsilon greedy method 를 적용한다.



[Fig 5] Structure of Q-Network

```python
class Qnet(nn.Module):
    def __init__(self):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(4, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 2)

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

    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()
```

[Fig 6] QNet class

1) Referencing the Loss function in [Fig 7], implement the loss that will be used to update the model.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r+\gamma \max_{a'} Q(s',a';\theta_i^-)}_{\text{target}} - \underbrace{Q(s,a;\theta_i)}_{\text{prediction}}\right)^2$$

(target network: θ_i^-, policy network: θ_i)

[Fig 7] Loss function

```python
[14] def update_model(q, q_target, memory, optimizer, criterion):
        for i in range(10):
            s, a, r, s_prime, done_mask = memory.sample(batch_size)

            """ ### Implement the code yourself ### """
            # get predicted q value (using policy network)
            q_val = q(s)
            q_a = q_val.gather(1, a)

            # get target q value (using target network)
            q_target_val = q_target(s_prime)
            max_q = q_target_val.max(dim=1)[0].unsqueeze(1)

            # Update the target value only if the episode ends
            target = r + gamma * max_q * done_mask


            # define loss
            loss = criterion(q_a, target)

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

[Fig 8] Code for Exercise

<Explanation>

[Fig 7]의 loss function 을 사용하기 위해, target network 에서 target Q-value 와 policy network 에서 predicted Q-value 를 계산해야 한다. 우선, ReplayBuffer class object 인 memory 의 sample method 를 통해 s, a, r, s_prime, done_mask (순서대로 state, action, reward, next_state, done_make)를 변수화한다. Predicted Q-value 는 단순히 현 state 에서 action a 를 했을 때에 대응되는 q-value 를 저장하면 되고, **[Fig 8]**를 넘어서 q(s).gather(1,a) 로 줄일 수도 있다. 이후, target network q_target 을 이용하여 next state s_prime 의 Q-values 중 최대값을 max_q 변수에 저장한다(차원을 맞춰주기 위해 unsqueeze method 를 추가한다).

$r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ 에 해당하는 변수가 target 이다. r 은 reward, hyperparamter γ 는 미래 시점에 대한 reward 를 낮춰주기 위한 discount factor 이다. 곱해진 done_mask 는 episode 가 종료될 때 target 의 update 가 진행되어야 하며, 이에 대한 여부를 의미하는 1(done), 0(not done) 로 구성된 변수이다. 완성된 target 은 q_a 와 함께 loss 를 계산하는 데 사용된다. Loss function 은 상황에 따라 L1 혹은 L2 를 선택적으로 사용하는 Smooth L1 loss function 을, optimizer 는 Adam 을 사용한다.

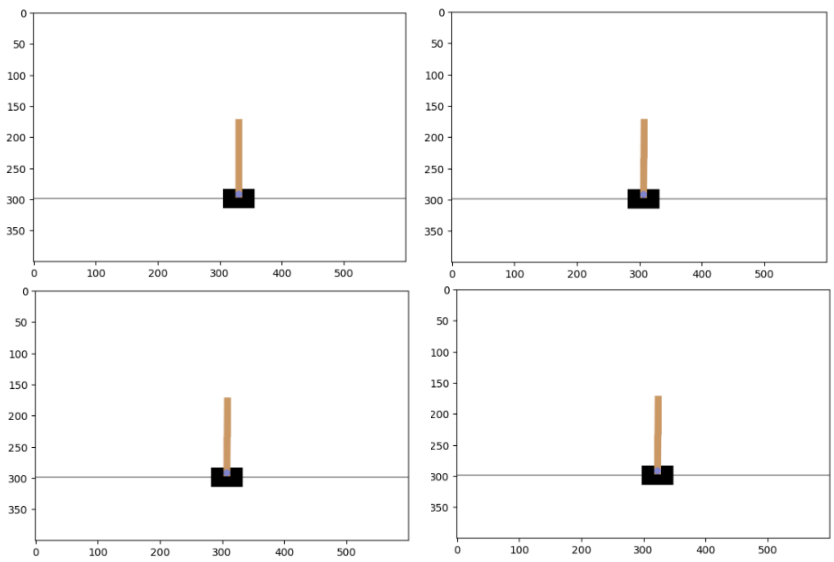
Model 을 훈련해보자. Training code 는 **[Fig 9]**와 같고, training 결과는 **[Fig 10]**과 같다. Episode 가 증가함에 따라, score 가 증가하고 있음을 알 수 있으며, 특히 n_episode 가 300 이 되면서부터 score 가 급격히 증가하였음을 알 수 있다. 즉, pole 이 완전히 기울어지지 않고 시간이 지나도 오래 버티려고 훈련이 되었다는 것을 알 수 있다. n_buffer 의 크기가 50000 으로 수렴하려 할수록 score 는 더 이상 증가하지 않고, 오히려 진동하고 있는 경향성을 보인다. 즉, 과정 중 overfitting 이 일어날 수도 있음을 알려준다. 앞서 언급한 epsilon greedy method 에 쓰이는 Epsilon 값은 훈련이 진행될수록 값을 선형적으로 줄이는 (8%에서 1%로) linear annealing 을 적용하였기에, 출력할 때 episode 가 누적됨에 따라 epsilon 값이 줄고 있음을 알 수 있다.

```python
q = Qnet()
q_target = Qnet()
q_target.load_state_dict(q.state_dict())
memory = ReplayBuffer()

print_interval = 20
score = 0.0

optimizer = optim.Adam(q.parameters(), lr=learning_rate)
criterion = F.smooth_l1_loss

for n_epi in range(1000):
    epsilon = max(0.01, 0.08 - 0.01*(n_epi/200)) #Linear annealing from 8% to 1%
    s = env.reset()
    done = False

    while not done:
        a = q.sample_action(torch.from_numpy(s).float(), epsilon)
        s_prime, r, done, truncated = env.step(a)
        done_mask = 0.0 if done else 1.0
        memory.put((s,a,r/100.0,s_prime, done_mask))
        s = s_prime

        score += r
        if done:
            break

    if memory.size() > 2000:
        update_model(q, q_target, memory, optimizer, criterion)

    if n_epi%print_interval==0 and n_epi!=0:
        q_target.load_state_dict(q.state_dict())
        print("n_episode :{}, score : {:.1f}, n_buffer : {}, eps : {:.1f}%".format(
                                                    n_epi, score/print_interval, memory.size(), epsilon*100))
        score = 0.0
env.close()
```

[Fig 9] Training Code

```
n_episode :20, score : 10.2, n_buffer : 204, eps : 7.9%
n_episode :40, score : 9.9, n_buffer : 403, eps : 7.8%
n_episode :60, score : 9.8, n_buffer : 599, eps : 7.7%
n_episode :80, score : 10.2, n_buffer : 803, eps : 7.6%
n_episode :100, score : 9.7, n_buffer : 997, eps : 7.5%
n_episode :120, score : 10.2, n_buffer : 1202, eps : 7.4%
n_episode :140, score : 10.1, n_buffer : 1404, eps : 7.3%
n_episode :160, score : 9.3, n_buffer : 1590, eps : 7.2%
n_episode :180, score : 9.8, n_buffer : 1785, eps : 7.1%
n_episode :200, score : 9.4, n_buffer : 1974, eps : 7.0%
n_episode :220, score : 15.2, n_buffer : 2277, eps : 6.9%
n_episode :240, score : 12.7, n_buffer : 2530, eps : 6.8%
n_episode :260, score : 17.2, n_buffer : 2875, eps : 6.7%
n_episode :280, score : 25.6, n_buffer : 3386, eps : 6.6%
n_episode :300, score : 94.3, n_buffer : 5273, eps : 6.5%
n_episode :320, score : 64.7, n_buffer : 6567, eps : 6.4%
n_episode :340, score : 160.8, n_buffer : 9782, eps : 6.3%
n_episode :360, score : 165.4, n_buffer : 13091, eps : 6.2%
n_episode :380, score : 195.6, n_buffer : 17003, eps : 6.1%
n_episode :400, score : 193.2, n_buffer : 20865, eps : 6.0%
n_episode :420, score : 170.6, n_buffer : 24277, eps : 5.9%
n_episode :440, score : 194.9, n_buffer : 28175, eps : 5.8%
n_episode :460, score : 193.0, n_buffer : 32035, eps : 5.7%
n_episode :480, score : 190.1, n_buffer : 35836, eps : 5.6%
n_episode :500, score : 180.0, n_buffer : 39436, eps : 5.5%
n_episode :520, score : 183.0, n_buffer : 43096, eps : 5.4%
n_episode :540, score : 192.8, n_buffer : 46952, eps : 5.3%
n_episode :560, score : 185.0, n_buffer : 50000, eps : 5.2%
n_episode :580, score : 188.2, n_buffer : 50000, eps : 5.1%
n_episode :600, score : 193.3, n_buffer : 50000, eps : 5.0%
n_episode :620, score : 192.3, n_buffer : 50000, eps : 4.9%
n_episode :640, score : 186.3, n_buffer : 50000, eps : 4.8%
n_episode :660, score : 151.3, n_buffer : 50000, eps : 4.7%
n_episode :680, score : 170.8, n_buffer : 50000, eps : 4.6%
n_episode :700, score : 172.2, n_buffer : 50000, eps : 4.5%
n_episode :720, score : 185.4, n_buffer : 50000, eps : 4.4%
n_episode :740, score : 185.2, n_buffer : 50000, eps : 4.3%
n_episode :760, score : 150.7, n_buffer : 50000, eps : 4.2%
n_episode :780, score : 171.8, n_buffer : 50000, eps : 4.1%
n_episode :800, score : 166.4, n_buffer : 50000, eps : 4.0%
n_episode :820, score : 166.1, n_buffer : 50000, eps : 3.9%
n_episode :840, score : 156.1, n_buffer : 50000, eps : 3.8%
n_episode :860, score : 162.1, n_buffer : 50000, eps : 3.7%
n_episode :880, score : 112.8, n_buffer : 50000, eps : 3.6%
n_episode :900, score : 149.3, n_buffer : 50000, eps : 3.5%
n_episode :920, score : 157.5, n_buffer : 50000, eps : 3.4%
n_episode :940, score : 145.4, n_buffer : 50000, eps : 3.3%
n_episode :960, score : 147.9, n_buffer : 50000, eps : 3.2%
n_episode :980, score : 178.8, n_buffer : 50000, eps : 3.1%
```

[Fig 10] Training Results



[Fig 11] Movement of Cartpole system after training

이제, Cartpole model 이 잘 훈련되었는 지 확인해보자. 앞서 말했듯이, 모든 움직임을 담을 수 없어 **[Fig 11]**과 같이 전체 과정 중 대표적인 4 frames 만을 첨부하였다. 대표적이라 하기에도 애매할 정도로, cartpole 이 균형을 잘 잡아 cart 자체의 (horizontal) position 값의 변동외에는 큰 변화가 없음을 시각적으로도 알 수 있다. Reward 를 더 많이 받기 위해 pole 을 쓰러뜨리지 않도록 model 이 잘 훈련되었다고 볼 수 있다. 초기 **[Fig 2]**의 움직임에 비하면 굉장히 훈련이 잘 되었다!