[Final] DQN with Demonstrations

2023710158 기계공학과 박경빈

1. 학습 환경

- **python == 3.9.16**
- **pytorch ==1.8.1**
- **numpy == 1.24.3**
- **gym == 0.22.0**
- **pygame == 2.4**

2. HyperParamter 설정

설정한 hyperparameter는 다음과 같습니다. 논문에서 제시한 대로 n step loss와 supervised weight는 1.0으로 놔두고 supervised loss margin은 0.8을 사용하였으며 L2 loss weight는 1e-5를 사용하였습니다. 또한 optimizer는 adam을, lr 은 3*1e-4, step size = 4, batch size는 64를 사용하였습니다.

Network는 총 2개의 hidden layer를 사용하였고 hidden unit은 64를 사용하여 input -> hidden -> hidden -> output로 구성이 되어있습니다. 학습 과정에서 추가적인 hyperparameter는 각 함수 설명할 때 추가로 설명하겠습니다. 코드는 다음과 같습니다.

```
self.lr = 0.0005
self.gamma = 0.99
self.epsilon = 0.01
self.eps_decay = 1e-5
self.eps_min = 0.001
self.target_update_freq = 20
self.hidden_size = 128
self.state_dim = env.observation_space.shape[0]
self.action_dim = env.action_space.n
```

```
self.n_step = 4
self.margin = 0.8
self.lambda_1 = 1.0
self.lambda_2 = 1.0
self.lambda_3 = 1e-5
```

```
self.main_net = DQfDNetwork(state_dim, action_dim, self.hidden_size)
self.target_net = DQfDNetwork(state_dim, action_dim, self.hidden_size)
self.loss_func = nn.SmoothL1Loss()
self.optimizer = optim.Adam(self.main_net.parameters(), lr=self.lr)
self.memory = ReplayMemory(self.n_step, self.gamma)
```

3. 구현 설명

- DQfD Network

우선 첫 번째는 DQfD Network입니다. 위에서 설명했듯이 64개의 Unit을 가지는 2개의 Hidden layer로 Q network를 구성하였습니다. 이전 3번째 과제와 같이 학습 환경이 간단하다고 생각하여 복잡하지 않게 Network를 구성하였습니다.

```
class DQfDNetwork(nn.Module):
    '''
    Pytorch module for Deep Q Network
    '''
    def __init__(self, state_dim, action_dim, hidden_size):
        '''
        Define your Q network architecture here
        '''
        super(DQfDNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, action_dim)

    def forward(self, state):
        '''
        Get Q values for actions given state
        '''
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        q_values = self.fc3(x)
        return q_values
```

- ReplayMemory

Replay memory에서는 전문가로부터 얻은 경험데이터인 Demonstration Data를 저장할 deque와 Demo 이후 스스로 학습하는 data를 저장할 deque를 각각 생성해줍니다. 일단 is_demo를 통해 demonstration data인지 아닌지를 구분합니다. 만약 demo라면 demonstration data의 value값만 transitions에 저장해 주고 이를 n_step_buffer에 저장합니다. 이때 n_step이기 때문에 states, actions, rewards, next_states, dones 이외에도 n_reward, n_state, n_done을 추가로 저장해줍니다. 이때 n_reward는 discount factor인 gamma(0.99)를 이용하여 다음과 같이 계산합니다.

```
n_reward = sum([self.gamma ** i * reward for i in range(self.n_step)])
```

또한 demo인지 아닌지를 확인하기 위해 is_demo 또한 반환해줍니다. 요기서 step_transition은 n_step_data를 생성할 때 임시로 데이터를 저장해놓은 deque인데 큰 reward가 학습 도중 큰 gradient를 만들게 되면 학습 저하에 영향을 주어 따로 scaling을 진행하였습니다. Reward_scaling의 hyperparmter는 0.01 ~ 0.1까지 확인해본 결과 0.01에서 가장 좋은 결과값을 보여 0.01을 이용하였습니다. 이후 demo가 아니라면 demo data를 받아오는 과정을 생략하고 위의 과정과 동일하게 buffer에 저장해줍니다. 코드는 아래와 같습니다.

<if is_demo>

```
class ReplayMemory(object):
    def __init__(self, n_step, gamma, buffer_size=50000):
        self.n_step = n_step
        self.gamma = gamma
        self.buffer_size = buffer_size
        self.demonstrations = deque()
        self.buffer = deque(maxlen=self.buffer_size)
        self.n_step_buffer = deque(maxlen=self.n_step)
        self.reward_shaping = 0.01


    def add(self, transition, is_demo = False):
        if is_demo:
            transitions = deque(zip(*transition.values()))
            for demo in transitions:
                self.n_step_buffer.append(demo)
                if len(self.n_step_buffer) == self.n_step:
                    states, actions, rewards, next_states, dones = zip(*self.n_step_buffer)
                    state = states[0]
                    action = actions[0]
                    reward = rewards[0]
                    next_state = next_states[0]
                    done = dones[0]
                    n_reward = sum([self.gamma ** i * reward for i in range(self.n_step)])
                    n_state = next_states[-1]
                    n_done = dones[-1]
                    step_transition = (state, action, self.reward_shaping * reward, next_state, done, self.reward_shaping * n_reward, n_state, n_done, is_demo)
                    self.demonstrations.append(step_transition)
                    if n_done:
                        self.n_step_buffer.clear()
            self.n_step_buffer.clear()
```

<else>

```
else:
    self.n_step_buffer.append(transition)
    if len(self.n_step_buffer) == self.n_step:
        states, actions, rewards, next_states, dones = zip(*self.n_step_buffer)
        state = states[0]
        action = actions[0]
        reward = rewards[0]
        next_state = next_states[0]
        done = dones[0]
        n_reward = sum([self.gamma ** i * reward for i in range(self.n_step)])
        n_state = next_states[-1]
        n_done = dones[-1]
        step_transition = (state, action, self.reward_shaping * reward, next_state, done, self.reward_shaping * n_reward, n_state, n_done, is_demo)
        self.buffer.append(step_transition)
        if n_done:
            self.n_step_buffer.clear()
```

def sample에는 demonstration data와 스스로 학습한 data를 모두 모아서 uniform 하게 random으로 뽑습니다. 논문에서는 PER을 사용하여 각각의 우선순위를 설정하였지만 이번 구현에서는 PER을 사용하지 않기 때문에 Unifrom Random을 사용하여 sample함수를 구현하였습니다. 코드는 아래와 같습니다.

```python
def sample(self, batch_size):
    '''
    samples random batches from buffer
    '''
    sample_buffer = list(self.demonstrations) + list(self.buffer)
    sample = random.sample(sample_buffer, batch_size)

    states, actions, rewards, next_states, dones, n_rewards, n_states, n_dones, is_demo = zip(*sample)

    states = torch.tensor(np.array(states), dtype=torch.float)
    actions = torch.tensor(np.array(actions), dtype=torch.int64).unsqueeze(1)
    rewards = torch.tensor(np.array(rewards), dtype=torch.float).unsqueeze(1)
    next_states = torch.tensor(np.array(next_states), dtype=torch.float)
    dones = torch.tensor(np.array(dones), dtype=torch.float).unsqueeze(1)
    n_rewards = torch.tensor(np.array(n_rewards), dtype=torch.float).unsqueeze(1)
    n_next_states = torch.tensor(np.array(n_states), dtype=torch.float)
    n_dones = torch.tensor(np.array(n_dones), dtype=torch.float).unsqueeze(1)
    is_demo = torch.tensor(np.array(is_demo), dtype=torch.float).unsqueeze(1)

    return states, actions, rewards, next_states, dones, n_rewards, n_next_states, n_dones, is_demo
```

- DQfD Agent

이 부분에서는 action을 선택하고 loss함수를 계산하였으며 pretrain을 진행하는 코드를 작성하였습니다. 우선 def select_action 부분은 이전 과제에서도 많이 작성했던 부분으로 시간에 따라 감쇠하는 eps_greedy에 따라 행동을 선택하며 코드는 다음과 같습니다.

```python
def get_action(self, state):
    '''
    Select actions for given states with epsilon greedy
    '''
    if np.random.uniform() < self.epsilon:
        return np.array([self.env.action_space.sample()])
    else:
        with torch.no_grad():
            q_values = self.main_net(state)
            action = q_values.argmax().numpy()
        return np.array([action])
```

다음은 loss 계산 함수 부분입니다. 이 부분은 논문과 같게 구현하였습니다. 논문에서는 one_step_loss, n_step_loss, margin_loss, L2_regularization_loss 총 4가지를 이용해서 loss를 계산하였으며 논문에서 제시한 loss계산은 다음과 같습니다.

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

$J_{DQ}(Q)$(one_step_loss)와 $J_N(Q)$(N_step_loss)의 자세한 계산 식은 논문 review에서 확인할 수 있습니다. 논문에서는 MSEloss를 이용하여 구하였지만 구현을 할 때에는 Smooth L1 Loss를 이용해서 구했습니다. $J_E(Q)$는 margin loss로서 demo에서의 action과 같은 action을 하도록 하는 large margin loss입니다. Margin을 q_value와 같은 모양의 tensor값으로 init을 한 후에 demo의 action에 해당되는 action 부분을 0으로 바꾸어 완성하였습니다. 이후 expert_mask에 margin(0.8)을

곱한 값과 q_value값을 이용하여 demo라면 supervised loss가 적용되기 때문에 1을, 아니라면 적용되면 안되기 때문에 0을 곱해주어 사용했습니다. $J_{L2}(Q)$는 L2 regularization으로 L2 norm을 이용하여 overfitting을 방지하는 방식으로 loss function에 반영하여 구현하였습니다. 이때 각 loss마다 위에서 코드로 첨부한 hyperparmeter인 lambda 값을 곱하여 사용하였습니다. 코드는 다음과 같습니다.

```
def calculate_loss(self, mini_batch):
    '''
    Implement DQfD loss function
    '''

    states, actions, rewards, next_states, dones, n_rewards, n_next_states, n_dones, is_demo_ = mini_batch
    # Q values of current states
    q_value = self.main_net(states)
    q_values = self.main_net(states).gather(1, actions)
    with torch.no_grad():
        next_q_values = self.target_net(next_states).max(dim=1, keepdim=True)[0]
        target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

    one_step_q_loss = self.loss_func(q_values, target_q_values)

    n_step_q_values = self.main_net(states).gather(1, actions)
    with torch.no_grad():
        n_step_next_q_values = self.target_net(n_next_states).max(dim=1, keepdim=True)[0]
        n_step_target_q_values = n_rewards + (1 - n_dones) * (self.gamma**self.n_step) * n_step_next_q_values

    n_step_q_loss = self.loss_func(n_step_q_values, n_step_target_q_values)

    expert_mask = torch.zeros_like(q_value)
    expert_mask.scatter_(1, actions, 0)
    margin_loss = is_demo_*((q_value + self.margin*expert_mask).max(1)[0].unsqueeze(1) - q_values)

    l2_reg_loss = torch.tensor(0.0)
    for param in self.main_net.parameters():
        l2_reg_loss += torch.norm(param)


    total_loss = one_step_q_loss + self.lambda_1 *n_step_q_loss + self.lambda_2 *margin_loss.mean() + self.lambda_3 * l2_reg_loss
```

One_step_loss

N_step_loss

Margin_loss

L2_reg_loss

def pretrain 부분에서는 get_demo-traj()를 통해 demonstration을 받고 이를 add하여 demo에 사용하였습니다. 코드는 다음과 같습니다.

```
def pretrain(self):
    '''
    DQfD pre-train with the demonstration dataset
    '''
    demonstration = get_demo_traj()
    # Add the demonstration dataset into the replay buffer
    self.memory.add(demonstration, is_demo=True)

    # Pre-train for 1000 steps
    for pretrain_step in range(1000):
        pretrain_batch = self.memory.sample(batch_size=64)
        loss = self.calculate_loss(pretrain_batch)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if pretrain_step % self.target_update_freq == 0:
            self.target_net.load_state_dict(self.main_net.state_dict())
```

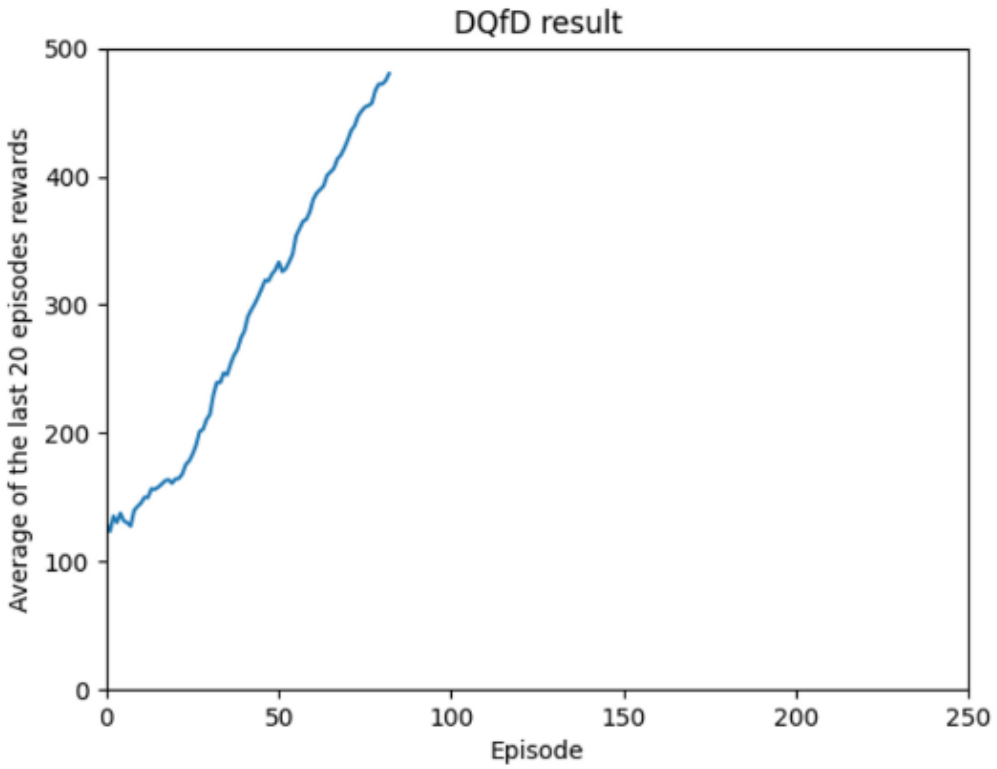
또한 max(self.eps_min, self.epsilon – self.eps_decay)부분을 통해 eps_greedy를 위한 eps 값을 계산하여 학습을 진행하였고 학습 부분은 다음과 같습니다.

```
################################################################################
        transition = (state, action, reward, next_state, done)
        self.memory.add(transition)

        batch = self.memory.sample(batch_size=64)
        loss = self.calculate_loss(batch)
        self.epsilon = max(self.eps_min, self.epsilon - self.eps_decay)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        if train_step % self.target_update_freq == 0:
            self.target_net.load_state_dict(self.main_net.state_dict())

        state = next_state
################################  DO NOT CHANGE  ################################
```

4. 학습 결과

학습 결과는 다음과 같습니다.

```
[Episode 81] Avg. score: 474.65
[Episode 82] Avg. score: 480.15
END train
```



총 82개의 에피소드에서 400점이 넘는 것을 확인할 수 있었고 475점이 넘어 early stop 된 것을 확인할 수 있습니다.