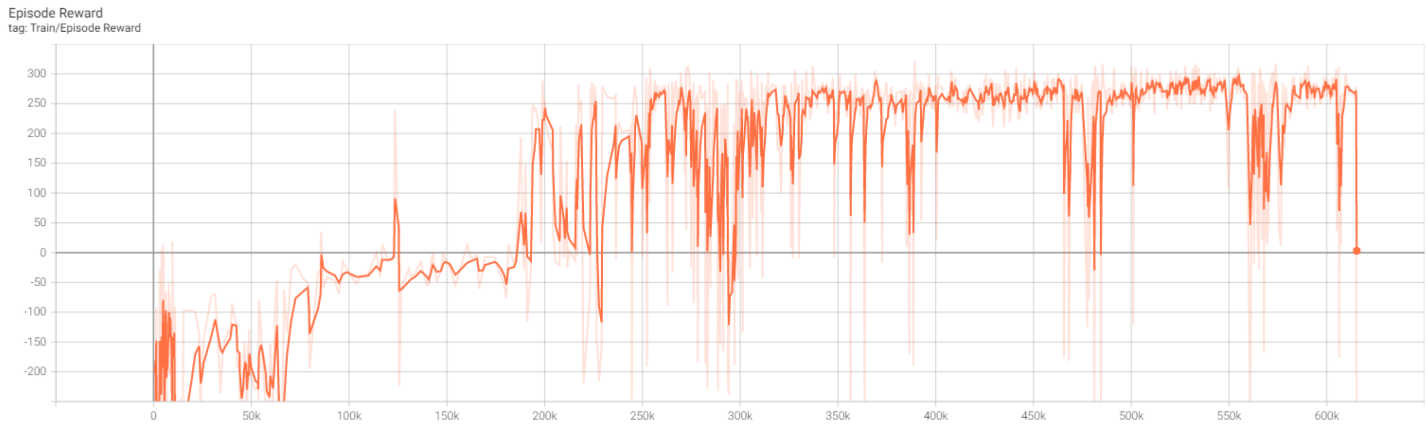
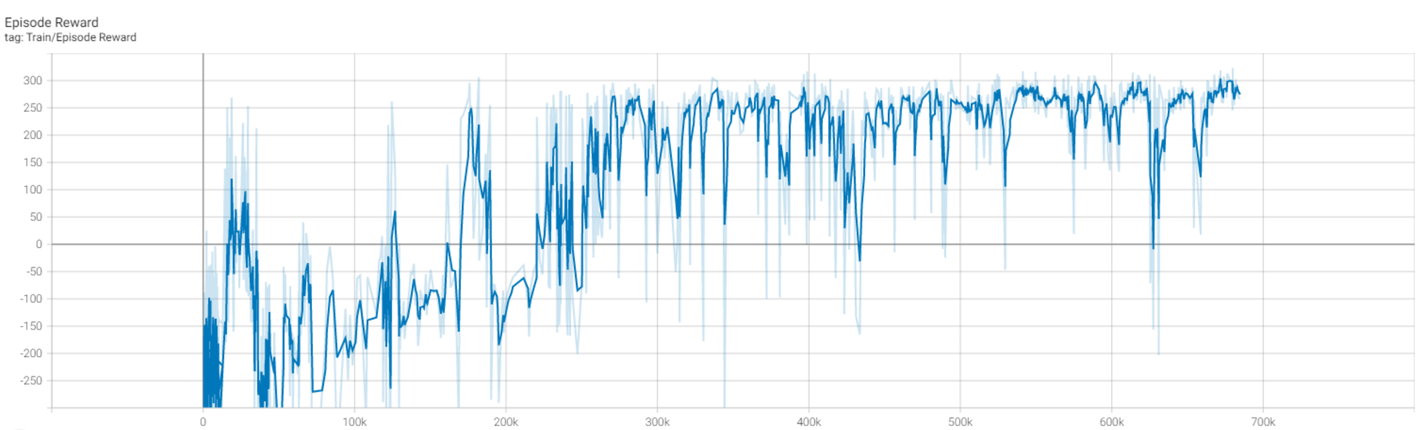
Lab6

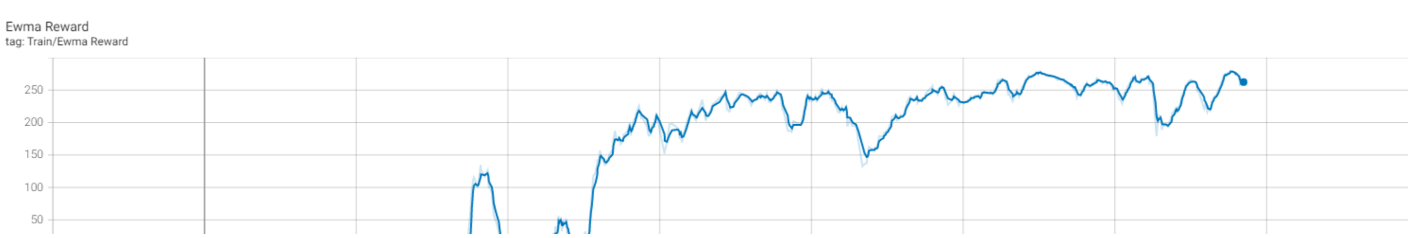
Report (80%)

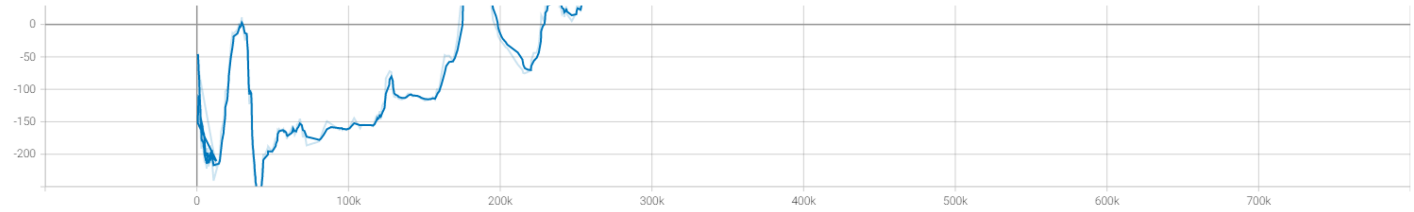
■ A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%)



■ A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2 (5%)



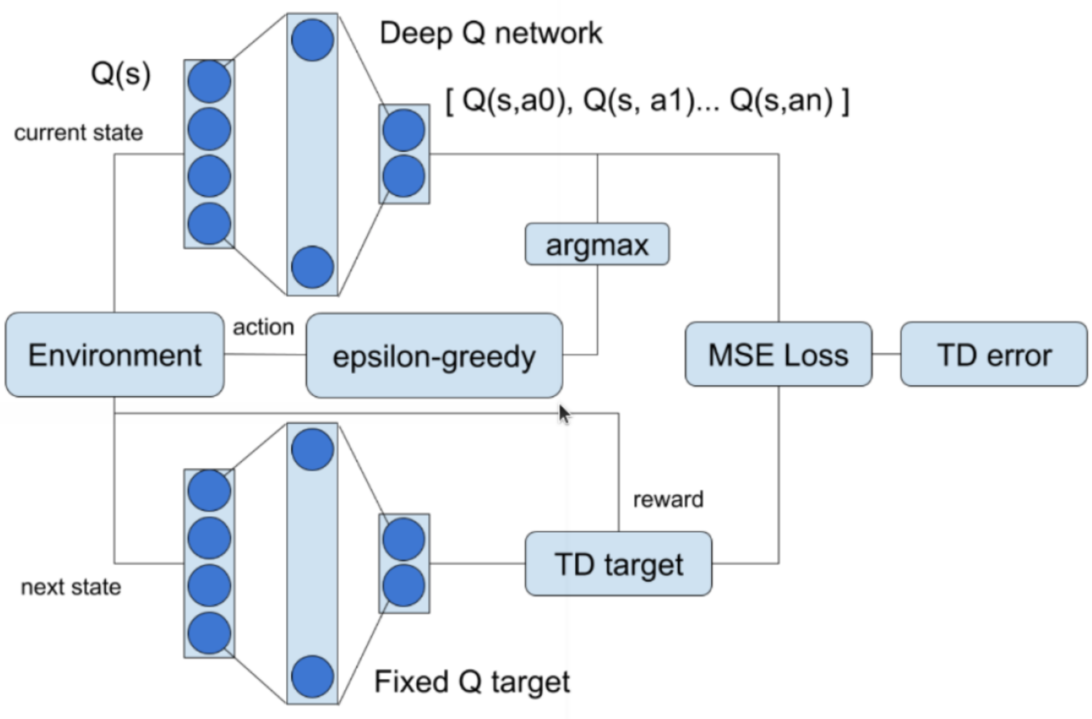




■ Describe your major implementation of both algorithms in detail. (20%)

• DQN

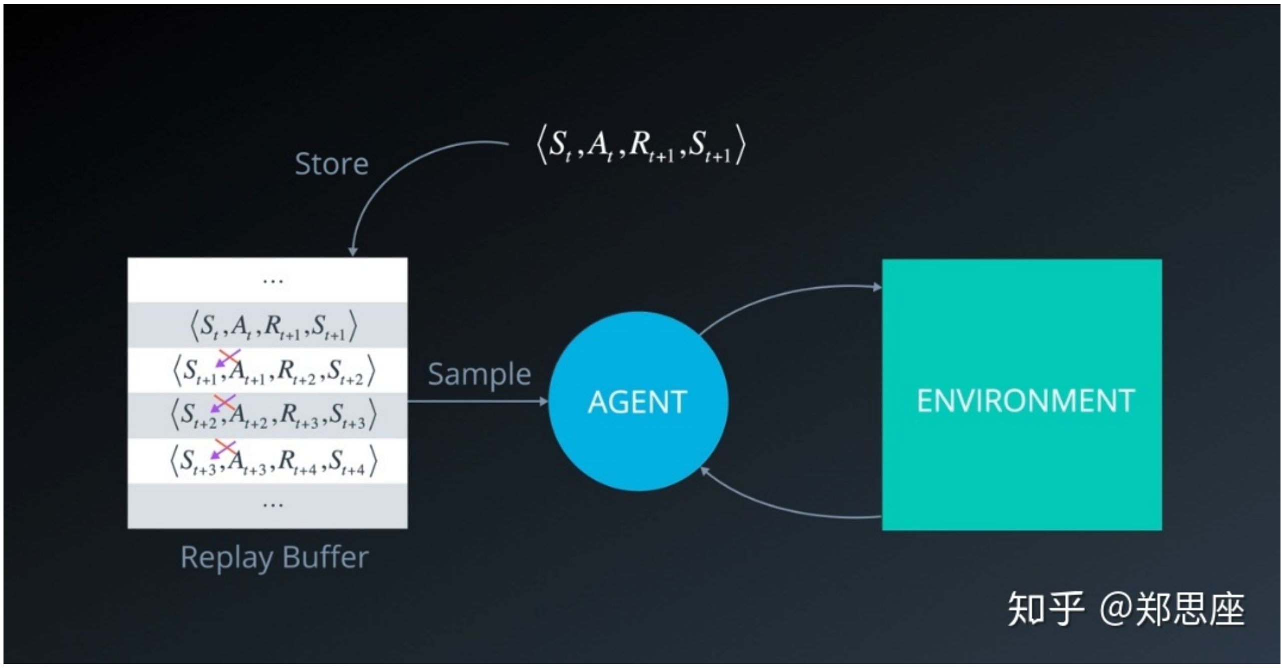
DQN(Deep Q learning)是指深度的Q learning，即把Q表換成卷積神經網路，DQN可以減少訓練所需的數據量，並能應付更大的Action數量、State數量，且能達到不錯的效果。




```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(300, 300)):
        super(Net, self).__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim, hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()
    def forward(self, x):
        ## TODO ##
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

建立一個NN來預測Q(s,a)的value

NN的輸入是8-dimension observation，輸出是4-dimension action。註:action有4種可能 (1)no-op(2) fire left engine(3) fire main engine(4) fire right engine。




```
class ReplayMemory:
    __slots__ = ['buffer']
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)
    def __len__(self):
        return len(self.buffer)
    def append(self, *transition):
        # (state, action, reward, next_state, done)
        self.buffer.append(tuple(map(tuple, transition)))
    def sample(self, batch_size, device):
        '''sample a batch of transition tensors'''
        transitions = random.sample(self.buffer, batch_size)
        return (torch.tensor(x, dtype=torch.float, device=device)
                for x in zip(*transitions))
```

Replay Buffer(ReplayMemory) :
從Q-learning的原始公式和算法流程來看，每次更新Q值的樣本都只能用一次，而且在連續獲取遊戲畫面的情景下，狀態樣本存在極高的相關性。針對這兩個問題，可以使用一個較大的buffer來儲存這些樣本，每次隨機均勻採樣，既能多次使用樣本，還能打破樣本之間的相關性。
因此，此段程式的目的是把過去的數據從一個緩存中又拿出來用，能比較好地解決了困擾Q-learning算法的樣本效率以及相關性問題。

```
def select_action(self, state, epsilon, action_space):
    if random.random() < epsilon:  # explore
        return action_space.sample()
    else:
```

```
        with torch.no_grad():
            return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)
[1].item()
```

epsilon-greedy:
選擇行動時，以epsilon的機率隨機探索（隨機從action space中取樣），否則則是以過去的經驗選擇最好的action。大多數時候都會選擇當前最佳選項（“貪婪”），但有時選擇概率很小的隨機選項(用來探索)。

```
def _update_behavior_network(self, gamma):
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    q_value = self._behavior_net(state).gather(dim=1,index=action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

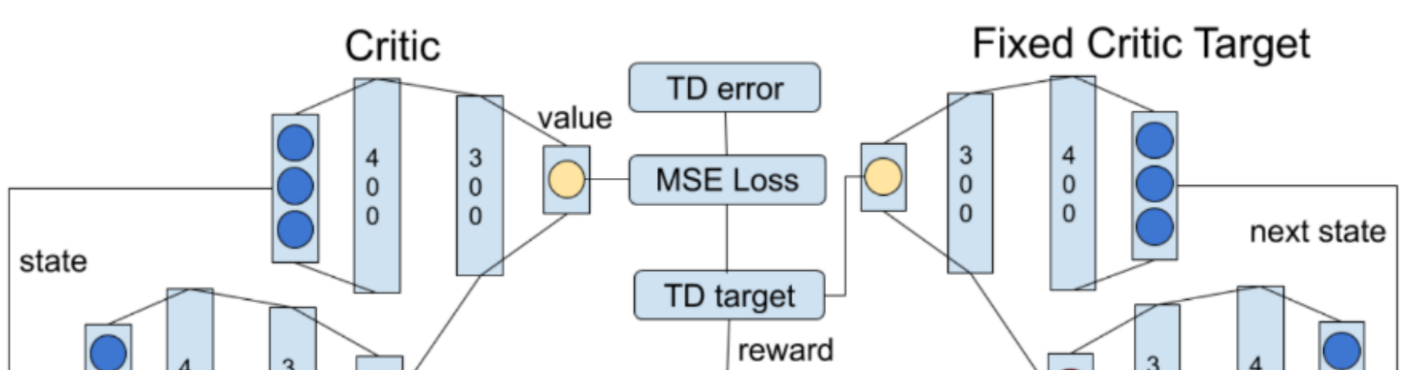
update behavior network是由replay memory去sample一些遊戲的過程，(state, action, reward,next_state, done)做td-learning，在對q_value跟q_target(reward+gamma*max Q'(s',a'))做MSELoss

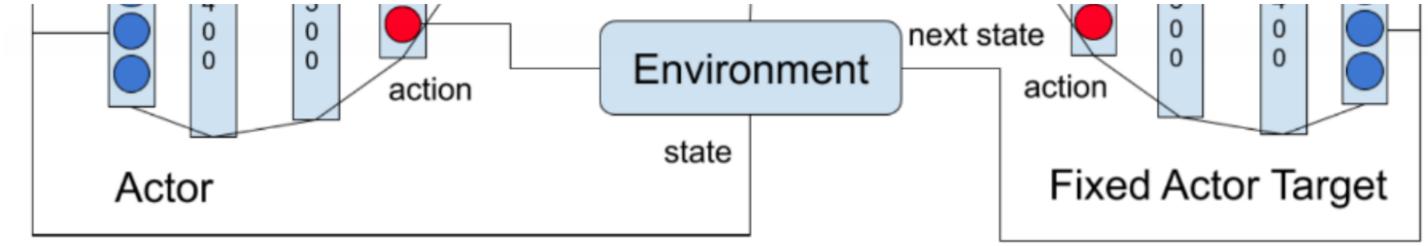
---

```
def _update_target_network(self):
     self._target_net.load_state_dict(self._behavior_net.state_dict())
```

在 update target network ， behavior network 每隔一段時間會更新取代 target network。

- DDPG





DDPG 需要同時學習 2 個網路：actor 和 critic。DDPG可以看成是DQN的擴展版，不同的是，以往的DQN在最終輸出的是一個動作向量，對於DDPG是最終確定地只輸出一個動作。而且，DDPG讓 DQN 可以擴展到連續的動作空間。

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(300, 300)):
        super(ActorNet, self).__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim,hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()
        self.tanh=nn.Tanh()
    def forward(self, x):
        ## TODO ##
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.tanh(self.fc3(x))
        return x
```

actor network可以依據目前 state 決定要執行哪個 action ，由三層 fully-connected layer 所構成。由於 action 有兩種(main engine:-1 ~ +1, left right engine: -1 ~ +1)，所以最後一層有 2 個 neuron

---

```
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(300, 300)):
        super(CriticNet, self).__init__()
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, hidden_dim[0]),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(hidden_dim[0], hidden_dim[1]),
            nn.ReLU(),
            nn.Linear(hidden_dim[1], 1),
        )
    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)
```

Critic Network用來預估Q(s,a)。輸出的是純量，所以最後一層 neuron 數為 1。

```
def select_action(self, state, noise=True):
        '''based on the behavior (actor) network and exploration noise'''
        ## TODO ##
        with torch.no_grad():
            if noise:
                re =
self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))+torch.from_numpy(self._action_noise.sa
mple()).view(1,-1).to(self.device)
            else:
                re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
        return re.cpu().numpy().squeeze()
```

在episode中，由Actor Network選擇action，但在 actor 回傳 action 給環境之前，會先將雜訊加到 action 中，以達 到探索的目的。

```
def _update_behavior_network(self, gamma):
        actor_net, critic_net, target_actor_net, target_critic_net = self._actor_net, self._critic_net,
self._target_actor_net, self._target_critic_net
        actor_opt, critic_opt = self._actor_opt, self._critic_opt
        state, action, reward, next_state, done = self._memory.sample(
            self.batch_size, self.device)
        q_value = self._critic_net(state,action)
        with torch.no_grad():
           a_next = self._target_actor_net(next_state)
           q_next = self._target_critic_net(next_state,a_next)
           q_target = reward + gamma*q_next*(1-done)
        criterion = nn.MSELoss()
        critic_loss = criterion(q_value, q_target)
        actor_net.zero_grad()
        critic_net.zero_grad()
        critic_loss.backward()
        critic_opt.step()
        action = self._actor_net(state)
        actor_loss = -self._critic_net(state,action).mean()
        actor_net.zero_grad()
        critic_net.zero_grad()
        actor_loss.backward()
        actor_opt.step()
```

在Critic的更新中與 DQN 類似。將 Actor 的輸出傳給 Critic，然後將 Critic 的負輸出當成 Loss，反向傳播完成網路的更新。

```
 @staticmethod
    def _update_target_network(target_net, net, tau):
        '''update target network by _soft_ copying from behavior network'''
        for target, behavior in zip(target_net.parameters(), net.parameters()):
            ## TODO ##
```

```
            target.data.copy_((1-tau)*target.data + tau*behavior.data)
```

DDPG 中 的target_network 和 buffer 的使用技巧也和 DQN 類似。

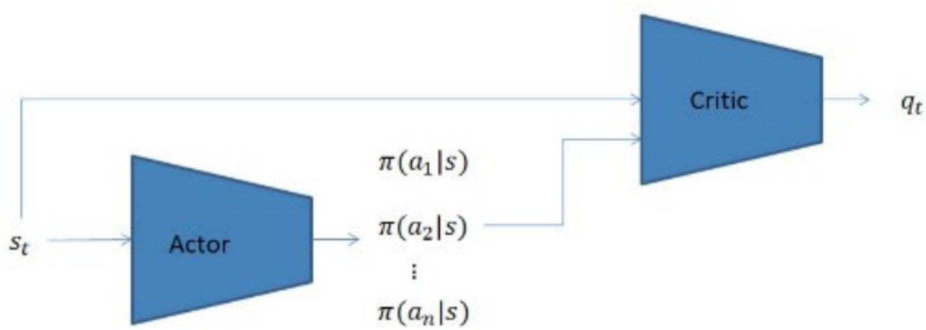
■ Describe differences between your implementation and algorithms. (10%)

在一開始 training 時，由於網路還沒有足夠的經驗與精準的參數，開始的時候會隨機探索，計原理類似 epsilon-greedy（隨機從 action space 中取樣），並將 transition 存進 buffer。

在 DQN中，也並不是每個 iteration 都要更新 behavior network，而是一段時間才會更新一次(ex: 4 個 iteration)。

在DDPG中，網路開始時候是隨機的，所以一開始評委(Q 網路 Critic)亂打分，演員(策略網路 Actor)亂表演，然後根據觀眾的回饋 reward ，Critic 的打分會越來越準確，進一步推動 Actor 的表現越來越好。

■ Describe your implementation and the gradient of actor updating. (10%)



Update the actor policy using the sampled gradient:

$$\nabla_{\theta^\mu}\mu|s_i \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$$

```
action = self._actor_net(state)
        actor_loss = -self._critic_net(state,action).mean()
```

更新 actor net:將 Actor 的輸出傳給 Critic，然後將 Critic 的負輸出當成 Loss(加負號是因為要讓他反向去變 化 parameter)，，反向傳播完成網路的更新。目標是最小化 Critic 回傳的負值。

■ Describe your implementation and the gradient of critic updating. (10%)

根據 critic 的損失函式來更新 crtic 網路:

$$\text{Update critic by minimizing the loss: } L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$$

```
## update critic ##
# critic loss
## TODO ##
q_value = self._critic_net(state,action)
with torch.no_grad():
   a_next = self._target_actor_net(next_state)
   q_next = self._target_critic_net(next_state,a_next)
   q_target = reward + gamma*q_next*(1-done)
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
```

把 next_action) 輸入target_network ，計算q_next，加上 reward 以後就可以求的 Q_target，然後通過 behavior network預測Q_value，Q_target和Q_value的均方差即為 loss，用反向傳播來更新Q Network(critic network)。

■ Explain effects of the discount factor. (5%)

$$G_t = R_{t+1} + \lambda R_{t+2} + \ldots = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

λ是discount factor。未來所給 reward 影響是愈小的，而當下的 reward 影響是最大 的。如果 discount factor 愈大，表示愈看中較遠的 future，如果愈小，代表比較看重較近的 future。

■ Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)

目的是為了在 explore 和 exploit 之間取得平衡。

剛開始訓練時，Q 可能還不是很好，會造成在某些state下 ，agent會一直被困在錯誤行動中，因此此時需要不斷的嘗試新的 action，藉由多一些隨機探索，建立更完整的 transition 資訊，讓他可以有機會了解不同的 action 會帶來什麼樣的結果。

而隨著不斷訓練，agent的判斷變得成熟且精準，可以選出較優的action。這時就不應該浪費時間在反覆隨機嘗試已經試過且知道結果的 action，需要改用 Q 來決定 action。

epsilon-greedy 就是這兩種情形的混合解法。一開始的 epsilon 可能很大，但可用 eps_decay逐漸調降。

■ Explain the necessity of the target network. (5%)

在target network 與 behavior network 的搭配下，可以使training更穩定。

產生Q_target的Target Network更新得比較慢，每隔一段時間(EX: 1000個iterations)才會被behavior network更新。

■ Explain the effect of replay buffer size in case of too large or too small. (5%)

replay buffer size 愈大， training 的過程愈穩定，但training 的速度會隨著replay buffer size變大而較慢(有較多不新鮮的資料，需要更長時間才能收斂)，佔用的 memory 空間亦會較大。

較多不新鮮的資料，需要更長時間才能收斂)，佔用的 memory 空間亦會較大。
replay buffer size 愈小，會一直著重在最近的 episode 上(只會考慮到最近的 data)。容易造成 overfitting。

Report Bonus (25%)

■ Implement and experiment on Double-DQN (10%)

DDQN和Nature DQN一樣，也有一樣的兩個Q網絡結構。除了目標Q值的計算方式以外，DDQN算法和 Nature DQN的算法流程沒什麼不相同。

DDQN不是直接在目標Q網絡裡面找各個動作中最大Q值，而是先在當前Q網絡中先找出最大Q值對應的動作，然後利用這個選擇出來的動作在目標網絡裡面去計算目標Q值。

```
def _update_behavior_network(self, gamma):
   state, action, reward, next_state, done = self._memory.sample(self.batch_size, self.device)

   q_value = self._behavior_net(state).gather(dim=1,index=action.long())
   with torch.no_grad():
       action_index=self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
       q_next = self._target_net(next_state).gather(dim=1,index=action_index.long())
       q_target = reward + gamma*q_next*(1-done)
   criterion = nn.MSELoss()
   loss = criterion(q_value, q_target)
   self._optimizer.zero_grad()
   loss.backward()
   nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
   self._optimizer.step()
```

和 DQN 不同，DDQN先經由 behavior net 取得 action ，再將其帶到 target net 以取得 q next，最後才獲得 q target。

■ Implement and experiment on TD3 (Twin-Delayed DDPG) (10%)

■ Extra hyperparameter tuning, e.g., Population Based Training. (5%)

Performance (20%)

■ [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30

DQN: 共2000個episode

```
Start Testing
total reward: 241.87
total reward: 239.64
total reward: 258.72
total reward: 293.14
total reward: 264.62
total reward: 268.78
total reward: 290.83
total reward: 293.26
total reward: -99.65
total reward: 267.90
Average Reward 231.91205737650216
```

DDQN: 共2000個episode

```
Start Testing
total reward: 247.30
total reward: 243.94
total reward: 269.15
total reward: 294.10
total reward: 264.71
total reward: 267.09
total reward: 308.55
total reward: 323.95
total reward: 313.02
total reward: 276.19
Average Reward 280.8015424634344
```

■ [LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30

DDPG: 共2000個episode

```
Start Testing
total reward: 108.67
total reward: 237.97
total reward: 261.43
total reward: 273.61
total reward: 261.40
total reward: 268.93
total reward: 300.33
total reward: 198.94
total reward: 159.43
total reward: 267.02
Average Reward 233.772565615283
```