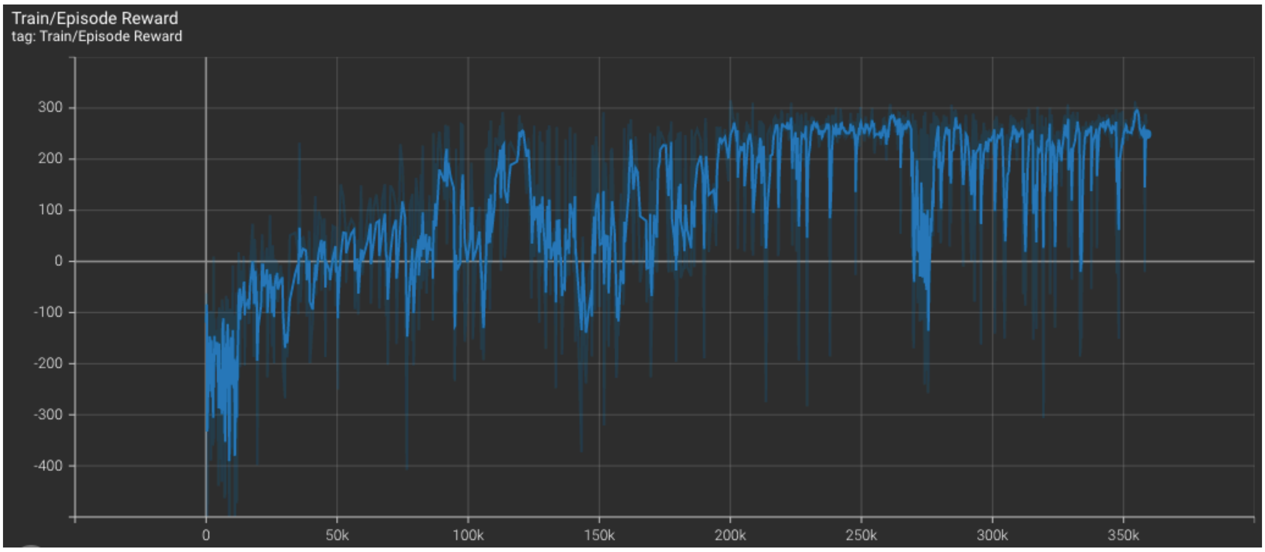
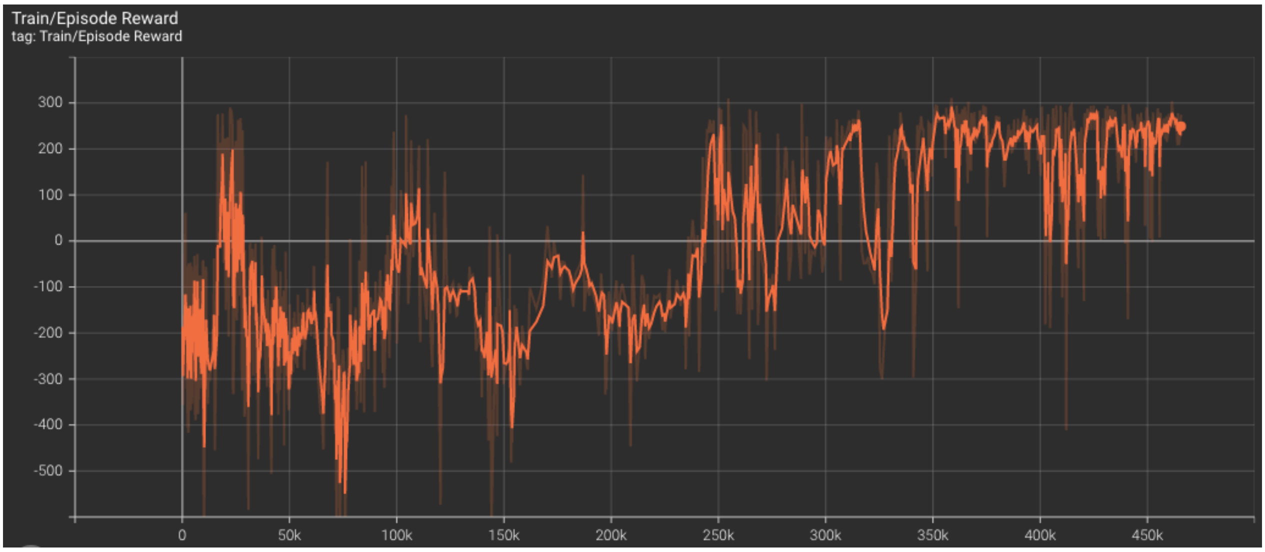
- **Report (80%)**
  - **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%)**



  - **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%)**

    DDPG 與 DQN 都是 off-policy 的 algorithm，且對於 ground truth Q value 都是用 temporal difference 的方法去近似。

Algorithm – DDPG algorithm: Behavior and target network(both actor and critic)

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for $episode = 1, M$ **do**

Initialize a random process N for action exploration

Receive initial observation state s_1

for $t = 1, T$ **do** action drawn from a deterministic policy with exploration

Select action $a_t = \mu(s_t|\theta^\mu) + \boxed{N_t}$ according to the current policy and exploration noise

experience replay

Execute action a_t and observe reward r_t and observe new state s_{t+1}

Store transition (s_t, a_t, r_t, s_{t+1}) in R

Sample random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R

Set $y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$

update the behavior networks(both actor and critic)

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

Update the actor policy using the sampled gradient:

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

Update the target networks: update the target network softly

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'}$$

end for

end for

Algorithm – Deep Q-learning with experience replay:

Initialize replay memory D to capacity N behavior and target network
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do** epsilon-greedy based on behavior network
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$ experience replay
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D update the behavior network
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$ update the target network periodically
  End For
End For

- **Network Architecture**

  DDPG 為 policy based 的 algorithm，所以有兩個 network 分別為 critic network 與 actor network。

```
#DDPG: actor network & critic network
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 = hidden_dim
        self.main = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, action_dim),
            nn.Tanh()
        )
        #raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        return self.main(x)
        #raise NotImplementedError

class CriticNet(nn.Module):
```

```
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, 1),
        )

    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)
```

DQN 為 value based 的 algorithm，所以只有一個用來衡量 Q value 的 network。

```
#DQN: Q value network
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(400, 300)):
# hidden_dim change from (32, 32) to (400, 300)
        super().__init__()
        ## TODO ##
        self.main = nn.Sequential(
            #first layer
            nn.Linear(state_dim, hidden_dim[0]),
            nn.ReLU(),
            #second layer
            nn.Linear(hidden_dim[0], hidden_dim[1]),
            nn.ReLU(),
            #third layer
            nn.Linear(hidden_dim[1], action_dim)
        )
        #raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        return self.main(x)
        #raise NotImplementedError
```

- **Optimizer**

  DDPG 與 DQN 都是對 behavior network 去做 update 而非 target network，其中不同的是 DDPG 有 actor 與 critic 兩種 network，而 DQN 只需要一個衡量 Q value 的 network 即可。

```
#DDPG: optimizer
## TODO ##
self._actor_opt = torch.optim.Adam(self._actor_net.parameters(), lr=1e-3)
# gradient descent toward behavior network instead of target network
self._critic_opt = torch.optim.Adam(self._critic_net.parameters(), lr=1e-
3) # gradient descent toward behavior network instead of target network
#raise NotImplementedError
```

```
#DQN: optimizer
## TODO ##
self._optimizer =
torch.optim.Adam(self._behavior_net.parameters(),lr=args.lr) # gradient
descent toward behavior network instead of target network
#raise NotImplementedError
```

- **Select Action**

  DDPG 的 action space 是 continuous 的，因此會有一定機率直接選擇 behavior actor network 建議的 deterministic action，一定機率選擇 behavior actor network 給出的 deterministic action 上又再加上一些雜訊。

```
#DDPG: select action
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    if noise:
        eps =
torch.from_numpy(self._action_noise.sample()).to(self.device)
    else:
        eps = torch.from_numpy(np.zeros(2)).to(self.device)
    return self._actor_net(torch.from_numpy(state).to(self.device)) + eps
    #raise NotImplementedError
```

  DQN 的 action space 是 discrete 的，因此會有一定機率去選擇衡量 Q value 的 behavior network 中能使 Q value 最大的 action，另外一定機率選擇 random action。

```
#DQN: select action
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    use_exploration = random.random() < epsilon
    if use_exploration:
        return action_space.sample() # select a random action
    else:
        return
self._behavior_net(torch.from_numpy(state).to(self.device)).argmax().item(
) # select the best action according to target network
    #raise NotImplementedError
```

- **Update Behavior Network**

DDPG 的 ground truth Q value 利用 temporal difference 、 target actor network 與 target critic network 近似得出，接著透過 mean square error 看 behavior critic network 估計出的值與 ground truth Q value 差距有多少，希望能盡可能縮短他們之間的差距；behavior actor network 希望能在給定一個 state 下輸出**最大的** Q value，亦即希望能夠**最小化負的** behavior actor network 輸出出來的 Q value。

```
#DDPG: update behavior network
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

    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## 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)
    #raise NotImplementedError
    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()
```

```
    ## update actor ##
    # actor loss
    ## TODO ##
    action = self._actor_net(state)
    actor_loss = -self._critic_net(state,action).mean()
    #raise NotImplementedError
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

DQN 希望 behavior network 能夠在給定 state 與 action 的情況下輸出最大的 Q value，而 ground truth Q value （代表在給定 state 與 action 的情況下輸出的最大 Q value） 由 temporal difference 與 target network 近似而來，因此就直接計算 behavior network 輸出的 Q value 與 ground truth Q value 的 mean square error，希望能夠最小化這個 error 來減少 behavior network 輸出的 Q value 與 ground truth Q value 之間的差距，讓 behavior network 盡可能能夠在給定 state 與 action 的情況下輸出最大的 Q value。

```
#DQN: update behavior network
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    index = action.long()
    q_value = torch.gather(self._behavior_net(state), dim=1, index=index)
    with torch.no_grad():
        q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1) #
DQN is a VI-based algorithm, not PI-based. hence we need to pick the
maximum Q value instead of the action with maximum Q value.
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
    #raise NotImplementedError
    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

- **Update Target Network**

  DDPG 採用 softly update 的方式，會每個 step 都 update 一點點 (control by ratio τ) 的 behavior network 的參數資訊至 target network。

```python
#DDPG: update target network softly
def update(self):
    # update the behavior networks
    self._update_behavior_network(self.gamma)
    # update the target networks
    self._update_target_network(self._target_actor_net, self._actor_net,
self.tau)
    self._update_target_network(self._target_critic_net, self._critic_net,
self.tau)

@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)
        #raise NotImplementedError
```

DQN 採用 periodically update 的方式，會在定期在跑了幾個 step 後，將整個 behavior network 的參數資訊覆蓋至 target network。

```python
#DAN: update target network periodically
def update(self, total_steps):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma)
    if total_steps % self.target_freq == 0:
        self._update_target_network()

def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
    #raise NotImplementedError
```

- **Test**

  DDPG 與 DQN 做 testing 的方式基本上一樣，唯一不同的是 DDPG 的 select_action function reutrn 回來的東西是一個在 GPU 上的 tensor，因此要把它轉回 NumPy 型態再丟給 environment 做互動，而 DQN 的 select_action function reutrn 回來的東西是一個 scalar，因此不需要做任何除裡就可以直接丟給 environment 做互動。

```python
#DDPG: test
def test(args, env, agent, writer):
    print('Start Testing')
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
```

```
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        for t in itertools.count(start=1):
            # select action
            action = agent.select_action(state)
            action = action.detach().cpu().numpy()
            # execute action
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            if args.render:
                env.render()
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward,
n_episode)
                print(
                 'Episode: {}\tLength: {:3d}\tTotal reward: {:.2f}\t'
                      .format(n_episode, t, total_reward))
                rewards.append(total_reward)
                break
        #raise NotImplementedError
    print('Average Reward', np.mean(rewards))
    env.close()
```

```
#DQN: test
def test(args, env, agent, writer):
    print('Start Testing')
    action_space = env.action_space
    epsilon = args.test_epsilon
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        for t in itertools.count(start=1):
            action = agent.select_action(state, epsilon, action_space)
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            if args.render:
                env.render()
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward,
n_episode)
```

```
                    print(
                        'Episode: {}\tLength: {:3d}\tTotal reward: {:.2f}\t'
                            .format(n_episode, t, total_reward))
                    rewards.append(total_reward)
                    break
            #raise NotImplementedError
    print('Average Reward', np.mean(rewards))
    env.close()
```

- **Replay Memory Sampling**

  DDPG 與 DQN sample transition 的方式相同，只是在 DDPG 裡面用了不同的寫法而已，都一樣要 output 在 replay buffer 裡 sample 出 batch_size 個的 (state, action, reward, next_state, done) 這五個東西。

```
#DDPG: replay memory sampling
def sample(self, batch_size, device):
    '''sample a batch of transition tensors'''
    ## TODO ##
    transitions = random.sample(self.buffer, batch_size)
    states = torch.Tensor([list(x) for x in np.asarray(transitions)[:,
0]]).to(device)
    actions = torch.Tensor([list(x) for x in np.asarray(transitions)[:,
1]]).to(device)
    rewards = torch.Tensor([list(x) for x in np.asarray(transitions)[:,
2]]).to(device)
    next_states = torch.Tensor([list(x) for x in np.asarray(transitions)
[:, 3]]).to(device)
    dones = torch.Tensor([list(x) for x in np.asarray(transitions)[:,
4]]).to(device)
    return states, actions, rewards, next_states, dones
    #raise NotImplementedError
```

```
#DQN: replay memory sampling
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))
```

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

  對於 DDPG 所有參數都沒有改動，而對於 DQN 則有改動 hidden_dim，(第一層 hidden layer dimension, 第二層 hidden layer dimension) 從 (32,32) 變成 (400,300)，讓 DQN network 的架構長的跟 DDQN critic network 的架構一樣，增加網路的參數來試著提高 reward。

  另外 DQN 更新 epsilon 的地方也變了，從每個 step 都更新一次更改為每個 epsisode 更新一次，讓它不要下降那麼快，在一開始多做一些 expolaration 避免卡在局部最佳 reward 上。

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

  actor network要做的事情就是給定 state，要找出哪個 action 能夠使 critic network 算出的 Q-value 最大，因此 actor loss 就被定義成以下的樣子：

$$\text{optimal action} = \arg\max_{\mu} Q(s, \mu(s|\theta_{\text{behavior\_actor}})|\theta_{\text{behavior\_critic}})$$
$$\Rightarrow \text{actor loss} = -Q(s, \mu(s|\theta_{\text{behavior\_actor}})|\theta_{\text{behavior\_critic}})$$
$$\Rightarrow \frac{\partial \text{ actor loss}}{\partial \theta_{\text{behavior\_actor}}} = \frac{\partial(-Q(s, \mu(s|\theta_{\text{behavior\_actor}})|\theta_{\text{behavior\_critic}}))}{\partial \mu(s|\theta_{\text{behavior\_actor}})} \frac{\partial \mu(s|\theta_{\text{behavior\_actor}})}{\theta_{\text{behavior\_actor}}}$$

  程式碼的部分在第三點 "Describe your major implementation of both algorithms in detail." 已提過。

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

  critic network 要做的事情就是盡可能讓 critic network 預估的 Q value 接近 ground truth Q value，而在 DDPG 裡 ground truth Q value 是利用 temporal difference 的方式搭配 target network去近似的，因此 critic loss 就透過 mean square error 被定義成以下的樣子：

$$\text{ground truth Q value} \approx \text{TD target} = r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta_{\text{target\_actor}})|\theta_{\text{target\_critic}})$$
$$\Rightarrow \text{critic loss} = \frac{1}{N}\sum(\text{TD target} - Q(s_t, a_t|\theta_{\text{behavior\_critic}}))^2$$

  程式碼的部分在第三點 "Describe your major implementation of both algorithms in detail." 已提過。

- **Explain effects of the discount factor. (5%)**

$$\text{Given a trajectoray } \tau, G_t(\tau) = \sum_{m=t}^{\infty} \gamma^m r_{m+1}$$

  代表著對於未來報酬的期望值會以近期得到的 reward 為主，而遠期得到的 reward 對於期望值的影響較低（折現(discount)較多）。

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

  reinforcement learning 最常遇到的問題就是要在 expolaration 與 expolitation 間作抉擇，若只做 greedy action selection 將有可能會卡在局部最佳解中，因為 expolaration 做得不夠多就有可能會沒有遇到那些比較好的 action 的結果，因此才會需要在原本的 greedy action 外再設一定機率會去做 random action 讓 agent 可以去做 expolaration，即為 epsilon-greedy action selection，這可以提高跳出局部最佳解的機率。

- **Explain the necessity of the target network. (5%)**

  DDPG 與 DQN 都是利用 temporal difference 的方法來去近似 ground truth Q value：

```
q_target = reward + gamma*q_next*(1-done)
```

  其中對於 DDPG 來說，target actor network 用來決定 a_next 的值，這個 a_next 會再與 target critic network 一起決定 q_next 的值，但每個 step 對於 critic network 的更新是針對 behavior critic network 來更新，對於 behavior actor network 則是要想辦法最大化 Q value，因此每個 step 對於 actor network 的更新是針對 behavior actor network 來更新，而 target network 更新的時候則是每個 step 都直接把 behavior network 的網路參數根據 τ 這個比率來複製到 target network 的網路參數

上，每次更新一點點；而對於 DQN 來說，target network 是用來決定 q_next 的值的，但每個 step 的更新也同樣是針對 behavior network 來更新，而 target network 更新的時候則是直接把 behavior network 的參數複製過來，在本實驗中是設約 1000 step 會更新一次 target network。亦即，target network 基本上就只是用來取值近似 ground truth Q value，再讓 behavior network 根據這個近似的 ground truth Q value 來在每個 step 更新自身的網路參數，並非直接更新 target network，這樣有個好處就是每次近似出來的 ground truth Q value 不會每經過一個 step 就變動的很劇烈，而是相對平緩地在改變；若只有一個網路的話，那麼用來近似值、更新的網路都將會是同一個，近似的 ground truth Q value 每經過一個 step 就會大幅改變，這會增加訓練的不穩定性。

> reference: [Why is a target network required?]()

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

  若 replay buffer 太大，當然能再利用的經驗很多，但就是因為太多的經驗需要過多記憶體，使得 training 速度下降；若 replay buffer 太小，代表能再利用的經驗很少，很容易過於專注於環境中的某個特定現象，導致 overfitting 的情況。

  > reference: [How large should the replay buffer be?]()

- **Report Bonus (20%)**
  - **Implement and experiment on Double-DQN (10%)**

    Dobule-DQN 整體架構跟 DQN 一樣，唯一有改動的地方是計算 q_next 時並非是取 target network 的最大 Q value，而是取能在 behavior network 取得最大 Q value 的 action，再將這個 action 餵給 target network 看對應的 Q value 是多少，用它來當 q_next。如此改動便會使原本 DQN over-estimate 的問題被改善，因為每次更新不再是取對於 target network 絕對是最大值的 Q value，而是取對於 target network 相對大的 Q value（透過 behavior network 來決定）。

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

    ## TODO ##
    buffer_action_index = action.long()
    q_value = torch.gather(self._behavior_net(state), dim=1,
index=buffer_action_index)
    with torch.no_grad():
        # different from DQN directly extract the maximal Q value based on
target network
        # DDQN extract the action with maximal Q value based on behavior
network, and see the corresponding Q value on target network when using the
particular action
        behavior_action_index =
self._behavior_net(next_state).argmax(dim=1).long().view(-1,1)
        q_next = torch.gather(self._target_net(next_state), dim=1,
index=behavior_action_index)
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

```
        #raise NotImplementedError
        # optimize
        self._optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
        self._optimizer.step()
```

DDQN 結果明顯比 DQN 還要好。

```
(base) ubuntu@ec037-109:~/DLlabs_temporary/lab6$ python ddqn.py --test_only
Start Testing
Episode: 0      Length: 152     Total reward: 256.81
Episode: 1      Length: 154     Total reward: 282.45
Episode: 2      Length: 165     Total reward: 285.24
Episode: 3      Length: 182     Total reward: 279.94
Episode: 4      Length: 287     Total reward: 292.95
Episode: 5      Length: 415     Total reward: 267.36
Episode: 6      Length: 230     Total reward: 299.04
Episode: 7      Length: 166     Total reward: 297.08
Episode: 8      Length: 231     Total reward: 312.58
Episode: 9      Length: 211     Total reward: 301.99
Average Reward 287.54505443570196
```

- **Performance (20%)**
  - **[LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30**

```
(base) ubuntu@ec037-109:~/DLlabs_temporary/lab6$ python dqn.py --test_only
Start Testing
Episode: 0      Length: 160     Total reward: 249.79
Episode: 1      Length: 169     Total reward: 240.59
Episode: 2      Length: 195     Total reward: 276.32
Episode: 3      Length: 190     Total reward: 279.55
Episode: 4      Length: 188     Total reward: 305.74
Episode: 5      Length: 534     Total reward: 267.11
Episode: 6      Length: 231     Total reward: 303.20
Episode: 7      Length: 214     Total reward: 291.42
Episode: 8      Length: 237     Total reward: 303.21
Episode: 9      Length: 527     Total reward: 268.96
Average Reward 278.5871336767776
```

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

```
(base) ubuntu@ec037-109:~/DLlabs_temporary/lab6$ python ddpg.py --test_only
Start Testing
Episode: 0      Length: 164     Total reward: 258.19
Episode: 1      Length: 132     Total reward: 292.61
Episode: 2      Length: 164     Total reward: 278.98
Episode: 3      Length: 169     Total reward: 285.34
Episode: 4      Length: 760     Total reward: 252.80
Episode: 5      Length: 233     Total reward: 263.45
Episode: 6      Length: 178     Total reward: 307.46
Episode: 7      Length: 144     Total reward: 289.89
Episode: 8      Length: 188     Total reward: 312.22
Episode: 9      Length: 192     Total reward: 230.67
Average Reward 277.1617734601881
```