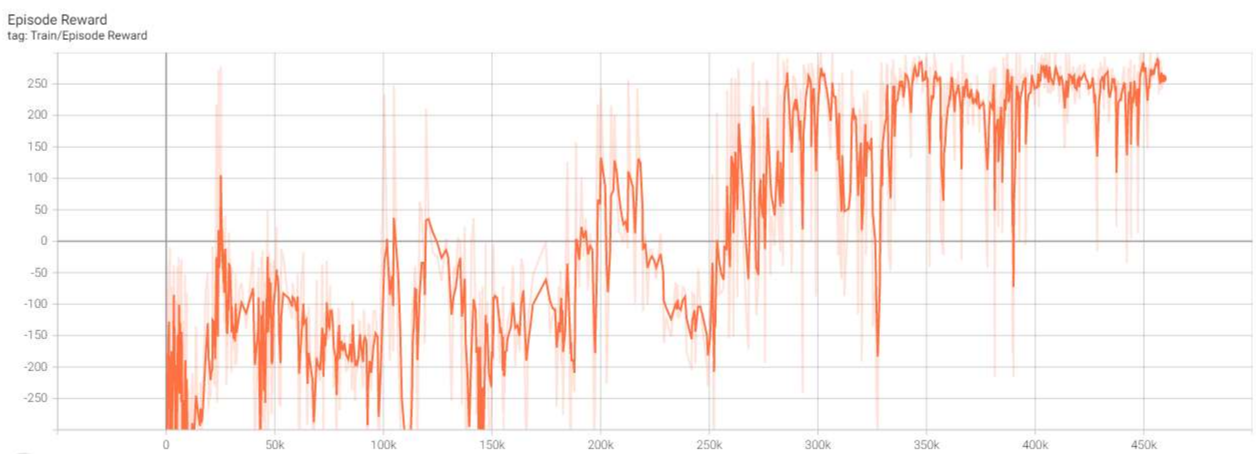
Report (80%)

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:

首先我們根據 specification 來定義我們的網絡，輸入是 8 個 state，輸出是 4 個 action。

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        ## TODO ##
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

在 DQN 裡選用 optimizer 為 Adam，因為 Adam 是比較常用和相對快收斂的 loss function

```
class DQN:
    def __init__(self, args):
        ## TODO ##
        self._optimizer =
torch.optim.Adam(self._behavior_net.parameters(), lr=args.lr)
```

我們選用了一個慢慢減少的 epsilon，一開始先隨機的行動，然後得到 reward，一段時間後慢慢把 epsilon 降低，讓 action 根據 Q-value 來決定下一個動作。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        action = action_space.sample()
    else:
        action =
torch.argmax(self._behavior_net(torch.tensor(state).to(self.device))).
item()  # get propability of output and take the largest one as action
    return action
```

更新網絡行為網絡的參數，先從 memory 抽樣出環境參數，然後根據當前 state 在 behavior_network 的輸出作為 q_value，然後如下公式：用 rewrad 加上 gamma 乘以下一個 state 的最大 Q_value 來作為 Q_target，用以和原來 state 的 q_value 作比較，從而得出 loss 並更新網絡參數。

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

```
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 ##
    q_value = self._behavior_net(state).gather(1, action.long())  # gather Q_value(ppb) from NET according to selected action, transfer to long(int) for .gather
    with torch.no_grad():
       q_next = torch.max(self._target_net(next_state), dim=1)[0].view(-1, 1)  # get target Q_max(dim=1) value[0] from next state and convert to (-1,1) dimension
       q_target = reward + gamma * q_next * (1 - done)  # if done(end game) q_target = 0
    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()
```

更新 target_network 就只是簡單的把 behavior_network 的參數抄去把 target_network 更新，我們可以根據參數來調節更新的頻率。

```
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())
```

這邊基本上和 train network 時要做的事差不多，把模型選擇的 action 作為實際的行動，並根據環境的 feedback 來記錄 reward 的分數。

```
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):
            # select action
            action = agent.select_action(state, epsilon, action_space)
            # execute action
            next_state, reward, done, _ = env.step(action)  # done -
end game
            # store transition
            agent.append(state, action, reward, next_state, done)

            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward,
n_episode)
                print('Episode: {}\tLength: {:3d}\tTotal reward:
{:.2f}\tEpsilon: {:.3f}'.format(n_episode, t, total_reward, epsilon))
                rewards.append(total_reward)
                break
```

DDPG:

Replay memory 就是記錄之前的環境參數，裡面的 sample 是方便我們從中抽取一個作為環境參數，根據輸入的 batch size 來決定要抽出多少個 sample 然後解壓成我們需要的樣式。

```
class ReplayMemory:
    __slots__ = ['buffer']

    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)
```

```python
    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'''
        ## TODO ##
        transitions = random.sample(self.buffer, batch_size)  # 
sample: (state(8), action(2), reward(1), next_state(8), done(1)) * 
batch_size
        # unzip transitions to state_vector, action_vector, 
reward_vector, next_state_vector, done_vector
        return (torch.tensor(out, dtype=torch.float, device=device) 
for out in zip(*transitions))  # transfer from double to torch.float
        # raise NotImplementedError
```

ActorNet 是 LunarLanderContinuous-v2 中用的網絡，大體上和 dqn 差不多，輸入為 8，輸出為 2 都在範圍-1 到 1 的中，所以最後把輸出接上 Tanh 方便直接使用。

CriticNet 則是根據 state 和 action 來輸出 Q_value 作為評分的網絡，這也是和 DQN 單純一個 Net 不一樣的地方。

```python
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.actor = 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.actor(x)
```

和 DQN 一樣選擇 Adam 作為 Optimizer，然後設定不一樣的 learning rate。

```
class DDPG:
    def __init__(self, args):
        # behavior network
        self._actor_net = ActorNet().to(args.device)
        self._critic_net = CriticNet().to(args.device)
        # target network
        self._target_actor_net = ActorNet().to(args.device)
        self._target_critic_net = CriticNet().to(args.device)
        # initialize target network
        self._target_actor_net.load_state_dict(self._actor_net.state_dict())
        self._target_critic_net.load_state_dict(self._critic_net.state_dict())
        ## TODO ##
        self._actor_opt = torch.optim.Adam(self._actor_net.parameters(),
lr=args.lra)
        self._critic_opt = torch.optim.Adam(self._critic_net.parameters(),
lr=args.lrc)
        # action noise
        self._action_noise = GaussianNoise(dim=2)
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)
        ## config ##
        self.device = args.device
        self.batch_size = args.batch_size
        self.tau = args.tau
        self.gamma = args.gamma
        self.test_only = args.test_only
```

根據公式在 select action 上加上 noise 幫助 network 更好的探索，讓 action 不要一直只選擇最優解，有一定機率觸發到不一樣的 action，但基本上類似的 action 應該是有差不多的 q_value。

$$\text{Select action } a_t = \mu(s_t|\theta^\mu) + \boxed{N_t} \quad \text{A noise process}$$

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.tensor(state).to(self.device)
    action = self._actor_net(state)
    if noise:
        action +=
torch.tensor(self._action_noise.sample()).to(self.device)
    return action.cpu().detach().numpy()
```

更新方面，如上所述會以 criticNet 作為 q_value 的評分，然後用 q_target 和 q_value 做 MSE 取得 loss，更新 critic 的參數，actor 則用 minimize – q_value 來作為 loss 更新參數，由於 max Q_value 的功效等價於 min Q_value。

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

```
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 = critic_net(state, action)
    with torch.no_grad():
       a_next = target_actor_net(next_state)
       q_next = target_critic_net(next_state, a_next)
       q_target = reward + gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    # print(q_value, q_target)
    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 = actor_net(state)
    actor_loss = -critic_net(state, action).mean()  # min q_value:
select max = -min
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

更新 target_network 則根據 target network 的公式，用 average 的方式更新。

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

```
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_(tau * behavior + (1 - tau) * target)
```

test function 基本上和 DQN 一樣，但要記得把 noise 關掉。

```
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, noise=False)
            # execute action
            next_state, reward, done, _ = env.step(action)  # done -
end game
            # store transition
            agent.append(state, action, reward, next_state, done)

            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward,
n_episode)
                print('Episode: {}\tLength: {:3d}\tTotal reward:
{:.2f}'.format(n_episode, t, total_reward))
                rewards.append(total_reward)
                break
```

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

主要是三個點：

1. 模型開始時的隨機 action，根據 warmup 的大小來先跑出環境參數並存入 replay memory 中。
2. 在更新 target_network 時，實際並不是每次都更新，可以利用 freq 來延遲更新。
3. Actor_loss 用-min Q_value 來實行 max Q_value，藉此減少 loss。

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

因為我們希望令 Q_value 最大化，而用上最少化的-Q 其實也是同樣效果，用 Q_value 的值來更新的網絡。

```
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()  # min q_value: select max = -min

# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

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

Critic 更新利用 target_net 預測的 action 來作為下一個 action's q_value 的預測，以前來得出 q_target 和原先的 q_value 比較得出 loss 更新 critic 的參數。

```
q_value = critic_net(state, action)
with torch.no_grad():
   a_next = target_actor_net(next_state)
   q_next = target_critic_net(next_state, a_next)
   q_target = reward + gamma * q_next * (1 - done)
criterion = nn.MSELoss()
# print(q_value, q_target)
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()
```

Explain effects of the discount factor. (5%)

因為 gamma 和 q_next 相乘的，所以可以解釋成 q_target 考慮下一步 q_value 的比例，以此來加入預測未來分數的考量，而不單單只考慮當前 action 的 reward。

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

Epsilon 其實就是讓模型多考量不一樣的 action，不要單單只看 q_value 來決定 action，若模型因為 network 一直選擇同一個 action，可能會導致有更好的 action 但沒機會採用，多探索不一樣的路，讓模型不會止步不前。在初始化也是很有用，能夠讓模型得到一定的 q_value，知道哪個 action 較好，在 DQN 上還將 epsilon 遞減，以此讓模型漸漸依賴 q_value 判定 action。

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

Target_network 可以使更新時更加穩定，不會因為只有一個網絡導致更新時數值一直變動，容易有很大的變化。

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

1. 太大的話 training 時間會變得很長，亦容易得到關聯性較低的 sample，反之考慮亦是較為全面的訊息。
2. 太小的話會一直 sample 類同甚至一樣的環境參數，buffer 就沒有那麼有效，且可能會造成 overfitting。

Report Bonus (25%)

Implement and experiment on Double-DQN (10%)

DQN 和 Double-DQN 最大的差別在於計算 Q_next 的方法(如下所示)，DQN 會先 target_network 裡的最大 Q 值作為 Q_next 用來計算 Q_target，而 DDQN 則是把 action 裡擁有最大 Q 值的動作找出來作為 Q_target 的 action。這樣可以減少最大 Q 值作為 Q_target 的差距，以貼近實際 Q 值。

$$Y_t^{Q} = r_{t+1} + \gamma \max_a Q(S_{t+1}, \boxed{a} | \theta^-)$$

$$Y_t^{DoubleQ} = r_{t+1} + \gamma Q\left(S_{t+1}, \boxed{\operatorname*{argmax}_a Q(S_{t+1}, a|\theta)} |\theta^-\right)$$

實際的 coding 如下所示，先找出 behavior_net 裡的最大值作為 action_next，然後根據此 action 實際的 Q 值作為 Q_next，最後運算 Q_target。

```
q_value = self._behavior_net(state).gather(1, action.long())
with torch.no_grad():
    action_next = torch.argmax(self._behavior_net(next_state),
dim=1).view(-1, 1)  # take max q_value action as next action index
    q_next = self._target_net(next_state).gather(dim=1,
index=action_next.long())  # take t_net q_value according to b_net's
predict action
    q_target = reward + gamma * q_next * (1 - done)
```

Tensorboard plot of DDQN episode rewards:

Testing Reward of DDQN:

```
Episode: 0  Length: 221 Total reward: 250.02    Epsilon: 0.001
Episode: 1  Length: 288 Total reward: 243.71    Epsilon: 0.001
Episode: 2  Length: 315 Total reward: 267.73    Epsilon: 0.001
Episode: 3  Length: 205 Total reward: 312.81    Epsilon: 0.001
Episode: 4  Length: 292 Total reward: 267.89    Epsilon: 0.001
Episode: 5  Length: 212 Total reward: 314.26    Epsilon: 0.001
Episode: 6  Length: 218 Total reward: 311.56    Epsilon: 0.001
Episode: 7  Length: 214 Total reward: 281.35    Epsilon: 0.001
Episode: 8  Length: 285 Total reward: 282.65    Epsilon: 0.001
Episode: 9  Length: 247 Total reward: 252.66    Epsilon: 0.001
Average Reward: 278.4646082316626, Average/30: 9.282153607722087
```

Reward 比原來的 DQN 更好，我猜想是因為選到的動作較原來的 DQN 準確，導致更新的值更為直接，更容易重現以前選對的動作。

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

TD3 主要有 3 個 trick：

1. Clipped Double-Q Learning：用 2 個 Q-learning network(實驗裡則為 2 個 critic_net)來取代原本只有 1 個 network 生成的 Q_next，然後再取 2 個 network 中較為小的 q_value 作為 q_next，最後計算出 1 個 target_net 的 q_target

$$y = r + \gamma Q_{\theta'}(s', \pi_\phi(s'))$$

$$\boxed{y} = r + \gamma \min[Q_{\theta'_1}(s', \boxed{\pi_\phi}(s')), Q_{\theta'_2}(s', \boxed{\pi_\phi}(s'))]$$

Only one Q target

Only one actor

2. Delay Policy Updates：顧名思義就是延遲更新 Behavior Actor 和 Target Network 的頻率。

```
initial
for episode = 1~M do
   for t = 1~T do
      ...
      Update Behavior Critic
      Update Behavior Actor
      Update Targets Networks
```

```
initial
for episode = 1~M do
   for t = 1~T do
      ...
      Update Behavior Critic
      if t mod d then
         Update Behavior Actor
         Update Targets Networks
```

Hyperparamete

3. Target Policy Smoothing：也是對 Target policy 著手，在 action 加上 noise，期望 action 不會被卡在一個點，因為類似的 action 會有類似的 reward。

$$y = r + \gamma Q(s', \pi(s') + \epsilon), \epsilon \sim clip(\mathcal{N}(0, \sigma), \underline{-c, c})$$

Hyperparameters

Implementation:

CriticNet 改成 2 個 network 有 2 個輸出 q_value，同時新加 1 個單輸出的 criticNet 作為 target_net(如下)：

```python
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=256):
        super(CriticNet, self).__init__()
        self.relu = nn.ReLU()
        # critic1
        self.q1_fc1 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.q1_fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.q1_fc3 = nn.Linear(hidden_dim, 1)

        # critic2
        self.q2_fc1 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.q2_fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.q2_fc3 = nn.Linear(hidden_dim, 1)

    def forward(self, state, action):
        x = torch.cat([state, action], 1)
        x1 = self.q1_fc1(x)
        x1 = self.relu(x1)
        x1 = self.q1_fc2(x1)
        x1 = self.relu(x1)
        x1 = self.q1_fc3(x1)

        x2 = self.q2_fc1(x)
        x2 = self.relu(x2)
        x2 = self.q2_fc2(x2)
        x2 = self.relu(x2)
        x2 = self.q2_fc2(x2)
        return x1, x2  # output critic1 and 2

    def single_q(self, state, action):
        x = torch.cat([state, action], 1)

        x = self.relu(self.q1_fc1(x))
        x = self.relu(self.q1_fc2(x))
        x = self.q1_fc3(x)
        return x  # output single critic
```

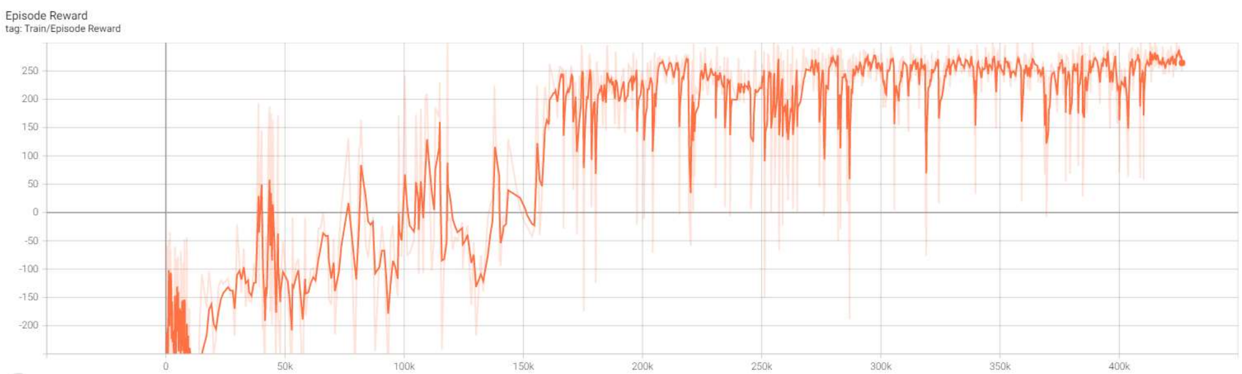
update_behavior_net 的部份，3 個 trick 都在裡面實現了，一開始把前面提到的第 3 點 noise 加到 next_action 裡面，然後 clip 來限制 noise 不會導致 action out of boundary。然後把 next_action 輸入到上面第一點提到的 clip double Q-learning，取出最小的那個 Q 值作為 q_next，用來更新 q_target，最後根據現在的 literation step 來決定 delay policy update 的時機，分別更新 actor、target_net。

Coding 如下：

```
def _update_network(self, current_step):
    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 ##
    with torch.no_grad():
        noise = (torch.randn_like(action) * self.noise).clamp(-
self.cp, self.cp)  # 3nd: select action and clip noise
        next_action = (target_actor_net(next_state) + noise).clamp(-
self.max_action, self.max_action)

        # Compute q_target value
        q1_next, q2_next = target_critic_net(next_state, next_action)
        q_next = torch.min(q1_next, q2_next)  # 2nd: min(double
critic)
        q_target = reward + gamma * q_next * (1 - done)

    q1_value, q2_value = critic_net(state, action)  # get q_value from
both critic
    critic_loss = F.mse_loss(q1_value, q_target) +
F.mse_loss(q2_value, q_target)

    # optimize critic
    critic_opt.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    # 2nd Delayed updates
    if current_step % self.update_step == 0:
        ## update actor ##
        # actor loss
        ## TODO ##
        actor_loss = -critic_net.single_q(state,
actor_net(state)).mean()

        # optimize actor
        actor_opt.zero_grad()
        actor_loss.backward()
        actor_opt.step()
```

```
        '''update target network by _soft_ copying from behavior
network'''
        ## TODO ##
        for param, target_param in zip(critic_net.parameters(),
target_critic_net.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)

        for param, target_param in zip(actor_net.parameters(),
target_actor_net.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 -
self.tau) * target_param.data)
```

Tensorboard plot of TD3 episode rewards:



很明顯 TD3 有著更穩定的 training reward，一直都很平穩，沒有太大起伏，也有了更多的 hyper parameter 可以調整。

Testing Reward of TD3:

```
Episode: 0  Length: 200 Total reward: 256.63
Episode: 1  Length: 210 Total reward: 252.01
Episode: 2  Length: 249 Total reward: 281.81
Episode: 3  Length: 243 Total reward: 299.89
Episode: 4  Length: 241 Total reward: 278.59
Episode: 5  Length: 249 Total reward: 301.34
Episode: 6  Length: 254 Total reward: 308.53
Episode: 7  Length: 229 Total reward: 284.61
Episode: 8  Length: 273 Total reward: 283.56
Episode: 9  Length: 191 Total reward: 260.29
Average Reward: 280.7250182206002, Average/30: 9.357500607353339
```

我認為 TD3 應該可以有更高的 Reward，所以我調大了 epoch，再 train，得到 291 的 reward，證明 TD3 是穩定而且有能力達到更高的 reward。

```
Episode: 0  Length: 150 Total reward: 267.84
Episode: 1  Length: 155 Total reward: 261.89
Episode: 2  Length: 180 Total reward: 283.87
Episode: 3  Length: 164 Total reward: 319.30
Episode: 4  Length: 177 Total reward: 287.54
Episode: 5  Length: 158 Total reward: 316.59
Episode: 6  Length: 149 Total reward: 319.28
Episode: 7  Length: 173 Total reward: 288.22
Episode: 8  Length: 183 Total reward: 299.83
Episode: 9  Length: 171 Total reward: 266.90
Average Reward: 291.1246286312855, Average/30: 9.704154287709516
```

Performance (20%)

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

Testing Reward of DQN：

```
Episode: 0  Length: 1000    Total reward: 103.58    Epsilon: 0.001
Episode: 1  Length: 249 Total reward: 250.35    Epsilon: 0.001
Episode: 2  Length: 292 Total reward: 276.03    Epsilon: 0.001
Episode: 3  Length: 297 Total reward: 290.87    Epsilon: 0.001
Episode: 4  Length: 251 Total reward: 276.95    Epsilon: 0.001
Episode: 5  Length: 366 Total reward: 290.09    Epsilon: 0.001
Episode: 6  Length: 279 Total reward: 292.80    Epsilon: 0.001
Episode: 7  Length: 297 Total reward: 240.46    Epsilon: 0.001
Episode: 8  Length: 247 Total reward: 297.18    Epsilon: 0.001
Episode: 9  Length: 201 Total reward: 253.83    Epsilon: 0.001
Average Reward: 257.21320171359014, Average/30: 8.573773390453004
```

Testing Reward of DDPG：

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

```
Episode: 0  Length: 185 Total reward: 271.27
Episode: 1  Length: 195 Total reward: 256.73
Episode: 2  Length: 207 Total reward: 289.43
Episode: 3  Length: 150 Total reward: 305.60
Episode: 4  Length: 259 Total reward: 287.30
Episode: 5  Length: 206 Total reward: 309.00
Episode: 6  Length: 130 Total reward: 305.55
Episode: 7  Length: 228 Total reward: 287.52
Episode: 8  Length: 209 Total reward: 298.01
Episode: 9  Length: 162 Total reward: 269.77
Average Reward: 288.01745036154045, Average/30: 9.600581678718015
```

在 1200 Episode 訓練中，DDPG 是 Reward 最高的，有 288 分，其次就是 TD3，280 分，然後是 DDQN 278 分，DQN 257 分。但調大 Episode 到 5000 後，TD3 仍能平穩的提升，並獲得最高的 reward。整體來說還是 TD3 較為可控穩定，可以根據情況調整需要探索的幅度，我的例子是相對平穩，但其實也可以把參數調整成比較注重探索的 TD3。DQN 其實由於預測每一步都是最大的 reward 導致可能出現選的 action 不是最好，但模型卻認為這個 action 的 reward 最大的情況，所以會出現 Episode 越大反而不一定學好的情況。DDQN 相對的改善了這部份，可以看到相對於 DQN 來說變得更平穩了，學習更為正確。而 DDPG 就提出 criticNet 來幫助 action 的評分和加入 noise 幫助探索，所以結果亦相對更好，但探索越多就代表越不穩定，而 TD3 就相對穩定而且還保有探索的部份。下面是各個模型在 epochs=5000 的 Episode Reward 的比較，同樣得出 TD3 相對穩定的結論。