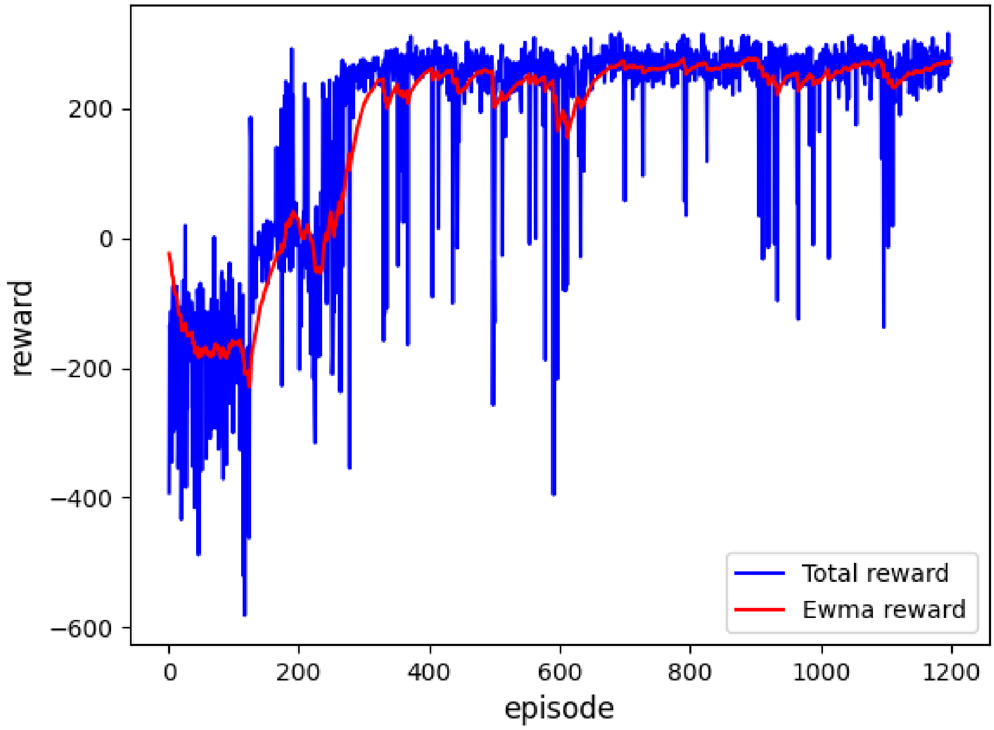
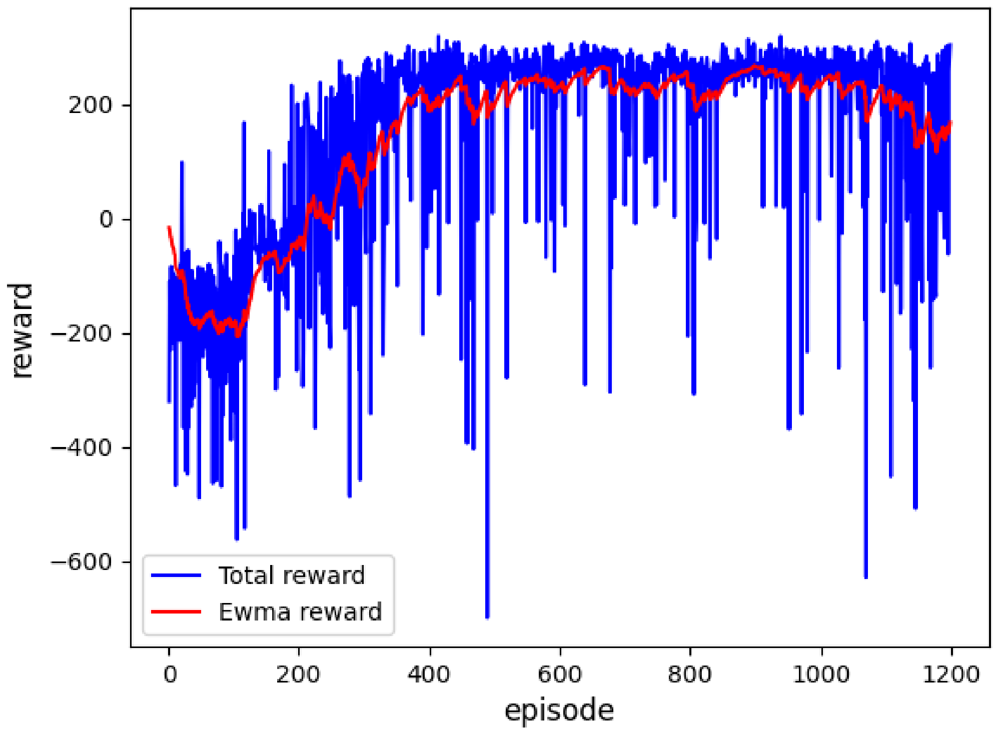
Lab6 Report

- **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2.**

DQN:

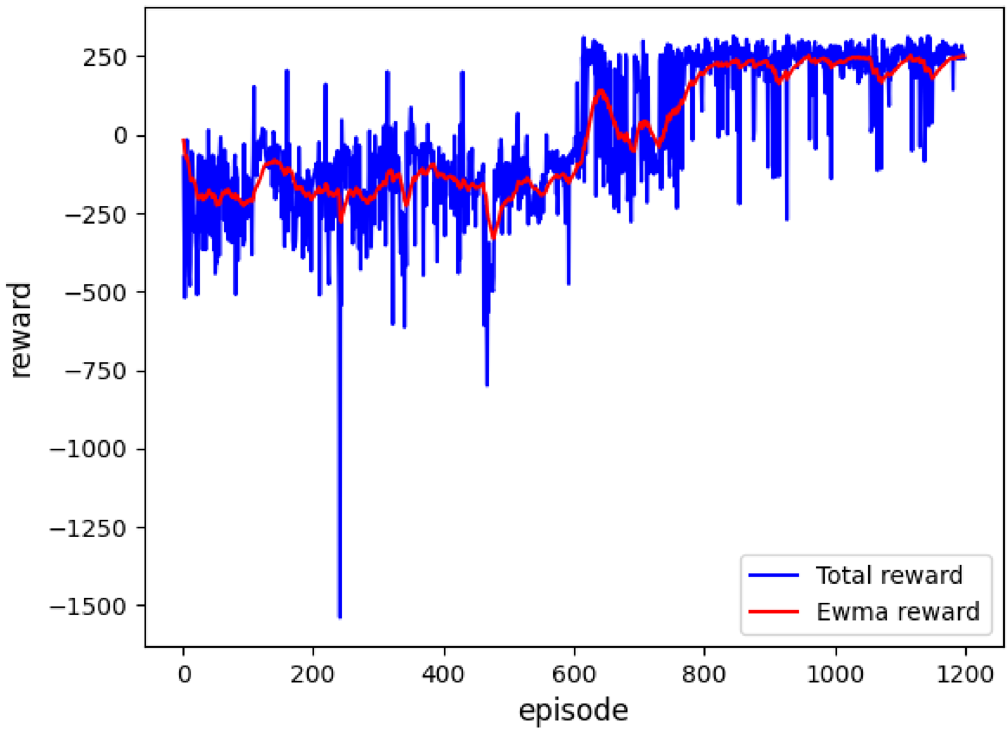


DDQN:



- **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2.**

DDPG:



- **Describe your major implementation of both algorithms in detail.**

DQN:

Algorithm – Deep Q-learning with experience replay:

Initialize replay memory D to capacity N
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**
    With probability ε select a random action a_t
    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $\phi_{t+1} = \phi(s_{t+1})$
    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
    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 $\hat{Q} = Q$
  End For
End For

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    # probability < epsilon, random choose
    if random.random() < epsilon:
        return action_space.sample()
    # probability >= epsilon, choose the max Q from behavior net
    else:
        return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()
```

上圖框框是 epsilon greedy 的部分。有 epsilon 的機率會隨機選擇一個 action (紅色框)。用 OpenAI gym 環境提供的 action_space.sample() 函式可以從環境的 action space 隨機 sample 一個 action 出來；反之，要從 behavior net 中找出 Q 值最大的 action (藍色框)。

Algorithm – Deep Q-learning with experience replay:

Initialize replay memory D to capacity N
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**
    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 $\phi_{t+1} = \phi(s_{t+1})$
    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
    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
    Every C steps reset $\hat{Q} = Q$
  End For
End For

```
# select action
if total_steps < args.warmup:
    action = action_space.sample()
else:
    action = agent.select_action(state, epsilon, action_space)
    epsilon = max(epsilon * args.eps_decay, args.eps_min)
# execute action
next_state, reward, done, _ = env.step(action)
# store transition
agent.append(state, action, reward, next_state, done)
if total_steps >= args.warmup:
    agent.update(total_steps, args.ddqn)
```

接著是上圖紅色框執行 action 得到 next state，並將整個 transition 存到 replay buffer 裡。用 gym 的 step 函式可以得到當下這個 state 執行 action 得到的 reward 和 next state，並且 done 告訴我們這是不是 terminal state。最後是藍色框的 update 部分，以下圖來進行詳細解釋:

Algorithm – Deep Q-learning with experience replay:

Initialize replay memory D to capacity N
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**
    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 $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    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
    Every C steps reset $Q = Q$
  End For
End For

```python
def update(self, total_steps, DDQN):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma, DDQN)
    if total_steps % self.target_freq == 0:
        self._update_target_network()

def _update_behavior_network(self, gamma, DDQN):
    # 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(dim=1, index=action.long()) # gather: change the index
    with torch.no_grad():
        if DDQN:
            # choose the best action from behavior net
            action_index = self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
            # choose related Q from target net
            q_next = self._target_net(next_state).gather(dim=1, index=action_index.long())
        else:
            # choose max Q(s', a') from target net
            q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)

        q_target = reward + gamma * q_next * (1 - done) # done == 1: final state
    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()
```

先從 replay buffer sample 一個 minibatch 的 transition 出來 (橘色框)，接著找到 q value 和 q target (紅色框)，並讓兩者差距越小越好(藍色框)。q value 直接從 behavior net 取出，取法實作上是用 pytorch 中的 gather 函式置換 index，讓 action 成為 index 置換進去 behavior net，回傳的結果就會是 q value。q target 概念上與 TD target 相似，這裡先說明 DQN 部分。將 next state 放進 target net 取得 q next 後，q target 就是 reward 加上 discount factor gamma 乘以 q next 乘以(1-done)。多乘一個(1-done)是因為若 next state 是 terminal state，則 done=1，q target 就會直接等於 reward；反之，則 done=0，乘上 1 不影響結果。然後是 backpropagate 的部分(藍

色框)，因為我們的目標是讓 q value 和 q target 越接近越好，直接取兩者的 mean square error 做為 loss 來進行 backpropagate 即可。

最後是 update 頻率，也就是綠色框部分。

DDPG:

(與 DQN 雷同部分不再詳細說明)

Algorithm – DDPG algorithm:

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**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    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 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:
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'}$$
  end for
end for

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            sampled_noise = torch.from_numpy(self._action_noise.sample()).view(1,-1).to(self.device)
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
            action = action + sampled_noise
        else:
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))

    return action.cpu().numpy().squeeze()
```

首先是 select action 的部分。在 DQN 中的 epsilon greedy，在 DDPG 中直接加上一個高斯雜訊做為擾動，以達到 exploration 的效果。實作上因為 test 不需要擾動，因此分成當 noise 為 true 時，action = action + sampled_noise；為 false 時直接回傳 action。

Algorithm – DDPG algorithm:

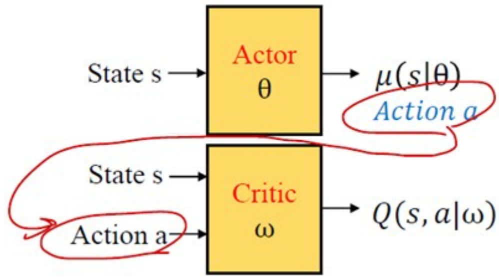
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for $episode = 1, M$ **do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    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 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:

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

  end for
end for




```python
## 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)  # done == 1: final state

criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

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

接著是 update critic 的部分。一樣分為 q value 和 q target。把當前的 state 和 action 丟進 critic net 的回傳值即是 q value；而 q target 則要將 next state 傳入 target_actor net，得到的 a next 再和 next state 一起傳入 target_critic net 得到 q next。流程如上圖右上角所示。接著 q_target = reward + gamma*q_next*(1-done) 和取 MSE loss 則和 DQN 一樣，這裡就不再多加贅述。

Algorithm – DDPG algorithm:

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**

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

    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 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:

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

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

  end for

end for

```
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = (self._critic_net(state, action).mean()) * -1  # gradient ascend

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

```
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)
```

update 完 critic 之後就要來 update actor。policy gradient 針對 actor 的部分是要 maximize objective function，也就是上圖紅色框框部分。我們將當前 state 傳入 actor net 得到 action (這裡的 action 和一開始從 transition 取出來的不同)，再將 action 和 state 傳入 update 過後的 critic，並試圖讓它給出的 q 值越大越好。實作上我們將得到的 q 值取平均再加上負號來當作 loss，這樣進行 backpropagate 就可以達到 gradient ascend 的效果。最後是綠色框框部分的 soft target update。這裡我們的 tau 是 0.005，也就是說 target = 0.995 * target + 0.005 * new，可以理解成 target 幾乎和原本一樣，一次就只改動一點點。這樣能夠使 target 較為穩定，不會一次就發生過大

的改動造成值出現過大的浮動。

- **Describe differences between your implementation and algorithms.**

根據上述內容，我整理出 3 個實作中的細節是演算法中沒有的:

1. epsilon decay in DQN:

   在 DQN 中有 epsilon greedy，epsilon 在算法裡是固定的，而在 training 的過程中是由大至小的。這是因為 training 前期要有較多的 exploration，epsilon 就要設計得較大；後期要有更多的 exploitation，epsilon 就要設計得較小。

2. warm up:

   在算法裡沒有提到，但實作上會先讓 replay buffer 存到一定量的 transition，再拿出來 update。若 replay buffer 太小，random sample 就會沒有意義，也會有太容易拿到相同 transition 的問題。

3. update frequency in DQN:

   算法中只有提到 every C step 將 behavior 複製給 target。但實作上 behavior 也不會每個 step 都進行 update，而是每 4 個 step 做一次 update。

- **Describe your implementation and the gradient of actor updating.**

如前面所述，DDPG 中的 actor update 是要 maximize 下面這條 objective function:

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

因為在 backpropagation 的過程中，critic 會是先完成 update 的，所以 actor 的目標可以理解成要讓已經 update 完成的 critic 對(state, action)輸出的 q 值越大越好。實作上，將當前 state 傳入 actor 得到 action 再將 action 和 state 傳入 update 過後的 critic，並試圖讓輸出的 q 值越大越好。又因為我們要進行的是 gradient ascend，將輸出值取平均乘上-1 當作 loss 來進行 backpropagate 可以達成此目的。

將實作對應回公式就會如下圖:

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

```
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = (self._critic_net(state, action).mean()) * -1  # gradient ascend

# 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.**

critic 的 update 方式前面也有說到，是用和 TD error 相似的概念。

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

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

把當前的 state 和 action 丟進 critic net 的回傳值即是 q value；而 q target 則要將 next state 傳入 target_actor net，得到的 a next 再和 next state 一起傳入 target_critic net 得到 q next。接著 q_target = reward + gamma*q_next*(1-done) 和 q value 取 MSE loss。

```
## 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)  # done == 1: final state

criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

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

- **Explain effects of the discount factor.**

discount factor 是指離現在越遠、越未來的 TD error 對現在的影響應該要越小。此 lab 只用到 one step TD error，discount factor 作用

並不明顯。若是 k step 則如下圖(取自講義):

- Advantage Actor-critic (A2C or A3C) policy gradient uses the (k+1)-step TD error $=A^{(k+1)}$

$$\Delta\theta = \alpha(\delta_t + \gamma\delta_{t+1} + \cdots + \gamma^k\delta_{t+k})\nabla_\theta \log \pi_\theta(s_t, a_t)$$

r 就是 discount factor，並會小於 1。因此 r 的越高次方值越小，乘上越未來的 TD error 也就會越小。

- **Explain benefits of epsilon-greedy in comparison to greedy action selection.**

在 RL 中，我們永遠要在 exploration 和 exploitation 之間取得平衡。而 epsilon greedy 就是一種方法。如果我們每次都用 greedy 選最好、有最大 q 值的 action，那我們將永遠不會知道是不是有更好的 action 是我們沒有發現、沒有嘗試過的。因此，要有一定的比例選擇最好的(exploitation)，也要有一定的比例隨機選擇最好之外的(exploration)。

- **Explain the necessity of the target network.**

使用 target network 是為了避免 behavior network 每次都要更新，取出來的值會一直浮動。從 target network 取值可以讓取出來的值更加穩定。

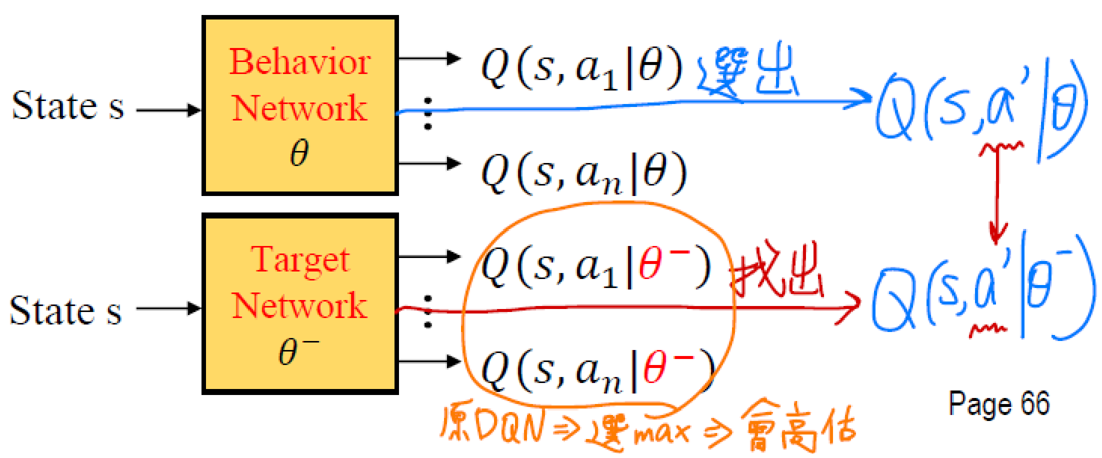
- **Explain the effect of replay buffer size in case of too large or too small.**

若 replay buffer 太小，則每次 sample 出來的 transition 會有高機率重複，不但會有許多狀態不會被存到，一直 update 重複的 transition 還會有 overfitting 的狀況產生。若 replay buffer 太大，雖然 training 會較為穩定，但就比較不容易達到收斂，訓練的效率較差。

Bonus

- **Implement and experiment on Double-DQN.**

DDQN 的概念如下圖 (取自講義):



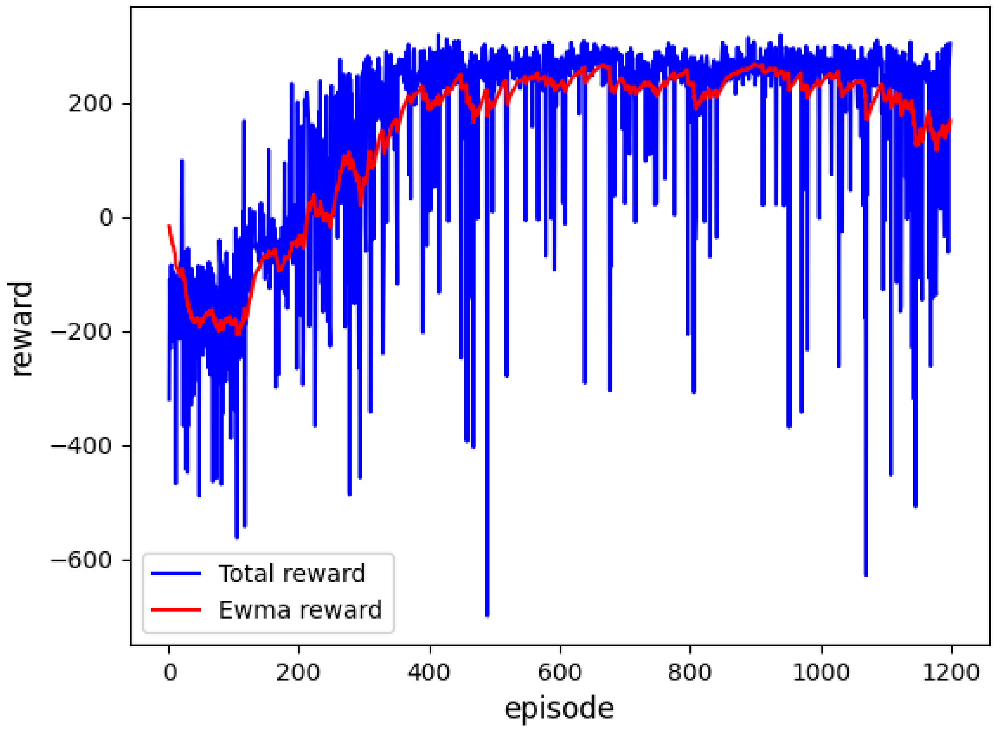
原本的 DQN 是從 target network 選出最大的 q 值來當 q next，而 DDQN 改從 behavior network 選，再從 target network 找出對應的 q 值來當 q next。這是為了避免從 target network 直接選 max 的 q 值會高估。實作上將 next state 傳入 behavior net 得到 action index，再用 gather 函式將 target net 的 index 置換掉，就可以得到

對應的 q next。如下圖:

```python
def _update_behavior_network(self, gamma, DDQN):
    # 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(dim=1, index=action.long()) # gather: change the index
    with torch.no_grad():
        if DDQN:
            # choose the best action from behavior net
            action_index = self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
            # choose related Q from target net
            q_next = self._target_net(next_state).gather(dim=1, index=action_index.long())
        else:
            # choose max Q(s', a') from target net
            q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)

        q_target = reward + gamma * q_next * (1 - done) # done == 1: final state

    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()
```

接下來是 experiment result。

training reward:



best testing performance:

```
In [29]: runfile('C:/Users/user/Desktop/DLP_HW/lab6/dqn-example.py', wdir='C:/Users/user/Desktop/
DLP_HW/lab6', args='--test_only --ddqn')
Start Testing
total reward: 247.10920381804942
total reward: 280.3502593031105
total reward: 275.23809328356754
total reward: 280.85855015535697
total reward: 299.520254359205
total reward: 269.6002969593901
total reward: 296.1735268591593
total reward: 289.9961031715884
total reward: 296.1489678829863
total reward: 301.77346776537945
Average Reward 283.6768723557792
```

- **Extra hyperparameter tuning.**

原本 DQN hidden layer 的設計是 32*32。

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

但我發現這樣的設定成效不佳，testing 結果如下圖:

```
Start Testing
total reward: -151.67857334873335
total reward: -119.65855783063292
total reward: -122.17753032877808
total reward: -130.08399125673247
total reward: -214.86913150018628
total reward: -128.6818821221234
total reward: -100.58185988523492
total reward: -170.00295882269927
total reward: -93.45949236149752
total reward: -94.25199524767042
Average Reward -132.54459727042885
```

因此我改用和 continuous 相同的 400*300:

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(8, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 4)
        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
```

testing 效果得到顯著的提升:

```
In [58]: runfile('C:/Users/user/Desktop/DLP_HW/lab6/dqn-example.py', wdir='C:/Users/user/Desktop/
DLP_HW/lab6', args='--test_only')
Start Testing
total reward: 239.6778258136213
total reward: 279.32579932595434
total reward: 269.37849748889107
total reward: 270.12305626819807
total reward: 296.48074145076066
total reward: 262.42717255069755
total reward: 299.1819637967591
total reward: 283.4329270730618
total reward: 309.9783225957957
total reward: 271.41329229344467
Average Reward 278.14195986571843
```

另外，我有用 Ray Tune 試圖找到 DDPG 更好的 hidden layer 設計。

Ray Tune 會在 config 裡 sample 好幾組 hyperparameter 出來同步執行並比較，並且將表現較差的組別的 process 提前 kill 掉。

```python
# apply ray tune
config = {
    "first_hidden": tune.sample_from(lambda _: np.random.randint(32, 512)),
    "second_hidden": tune.sample_from(lambda _: np.random.randint(32, 512)),
}
scheduler = ASHAScheduler(
    metric = "total_reward",
    mode = "max",
    max_t = 1200,
    grace_period = 200,
    reduction_factor = 2
)
reporter = CLIReporter(metric_columns=["total_reward", "ewma_reward", "training_iteration"])
result = tune.run(
    train,
    #resources_per_trial={"cpu": CPU, "gpu": GPU},
    config = config,
    num_samples = 5,
    scheduler = scheduler,
    local_dir = './outputs/raytune_result',
    keep_checkpoints_num = 1,
    #checkpoint_score_attr = 'max_total_reward',
    progress_reporter = reporter
)
# Extract the best trial run from the search.
best_trial = result.get_best_trial(
    'total_reward', 'max', 'last'
)
print(f"Best trial config: {best_trial.config}")
print(f"Best trial total reward: {best_trial.last_result['total_reward']}")
print(f"Best trial ewma reward: {best_trial.last_result['ewma_reward']}")
```

如上圖，config 中讓 ray tune 隨機 sample hidden layer 的維度，範圍都在 32 到 512 之間。接著 tune.run 會同步執行 num_samples 組 train，ASHAScheduler 會將其中較差的組別提前終止。執行結果如下圖:

```
== Status ==
Memory usage on this node: 14.7/31.8 GiB
Using AsyncHyperBand: num_stopped=5
Bracket: Iter 800.000: 287.3433263507781 | Iter 400.000: -167.34628862711693 | Iter 200.000: -82.65667717526199
Resources requested: 0/12 CPUs, 0/2 GPUs, 0.0/10.86 GiB heap, 0.0/5.43 GiB objects
Result logdir: C:\Users\user\Desktop\DLP_HW\lab6\outputs\raytune_result\train_2022-05-20_17-08-42
Number of trials: 5/5 (5 TERMINATED)
+-------------------+------------+-------+----------------+-----------------+----------------+---------------+----------------------+
| Trial name        | status     | loc   |   first_hidden |   second_hidden |   total_reward |   ewma_reward |   training_iteration |
|-------------------+------------+-------+----------------+-----------------+----------------+---------------+----------------------|
| train_71e7d_00000 | TERMINATED |       |            436 |             282 |       -173.072 |      -109.225 |                  400 |
| train_71e7d_00001 | TERMINATED |       |            451 |              56 |        275.74  |       208.04  |                  800 |
| train_71e7d_00002 | TERMINATED |       |            270 |             347 |       -260.572 |      -132.798 |                  200 |
| train_71e7d_00003 | TERMINATED |       |            116 |             308 |       -205.113 |      -147.645 |                  400 |
| train_71e7d_00004 | TERMINATED |       |            355 |             168 |        263.715 |       256.096 |                 1200 |
+-------------------+------------+-------+----------------+-----------------+----------------+---------------+----------------------+
```

可以看到 5 個 training process 都已經在 terminate 狀態。每組會有各自的 hidden layer，且 terminate 時的 training iteration 有所不同。但可惜最後並沒有找到 testing result 比 400*300 更好的 hidden layer。

有可能的原因是這個 train 的 variance 很大，單純用幾個 episode 的 reward 來判斷哪組 hyperparameter 比較好並不準確。有可能有一組是剛好有一段時間表現較差就被提前終止掉。另外還有一些可能的問題，例如 sample 組數不夠多、沒辦法 sample 出夠多組的 hyperparameter；又或者是要有更多 hyperparameter 配合參與，例如 batch size、learning rate 等，才會有效果。總而言之還有很多這方面的 issue 可以進行研究。

Testing Performance

- **Best performance in DQN (LunarLander-v2):**

```
In [58]: runfile('C:/Users/user/Desktop/DLP_HW/lab6/dqn-example.py', wdir='C:/Users/user/Desktop/
DLP_HW/lab6', args='--test_only')
Start Testing
total reward: 239.6778258136213
total reward: 279.32579932595434
total reward: 269.37849748889107
total reward: 270.12305626819807
total reward: 296.48074145076066
total reward: 262.42717255069755
total reward: 299.1819637967591
total reward: 283.4329270730618
total reward: 309.9783225957957
total reward: 271.41329229344467
Average Reward 278.14195986571843
```

Average Reward of 10 testing episodes: 278.14

- **Best performance in DDQN (LunarLander-v2):**

```
In [29]: runfile('C:/Users/user/Desktop/DLP_HW/lab6/dqn-example.py', wdir='C:/Users/user/Desktop/
DLP_HW/lab6', args='--test_only --ddqn')
Start Testing
total reward: 247.10920381804942
total reward: 280.3502593031105
total reward: 275.23809328356754
total reward: 280.85855015535697
total reward: 299.520254359205
total reward: 269.6002969593901
total reward: 296.1735268591593
total reward: 289.9961031715884
total reward: 296.1489678829863
total reward: 301.77346776537945
Average Reward 283.6768723557792
```

Average Reward of 10 testing episodes: 283.68

- **Best performance in DDPG (LunarLanderContinuous-v2):**

```
In [23]: runfile('C:/Users/user/Desktop/DLP_HW/lab6/ddpg-example.py', wdir='C:/Users/user/Desktop/
DLP_HW/lab6', args='--test_only')
Start Testing
total reward: 245.62997203849835
total reward: 278.1958161160608
total reward: 274.99716770969417
total reward: 267.6146542033257
total reward: 297.72172560396484
total reward: 265.9000484309872
total reward: 288.0134844987243
total reward: 286.7574307829075
total reward: 298.9933796629788
total reward: 270.9453905372832
Average Reward 277.4769069584425
```

Average Reward of 10 testing episodes: 277.48