Lab6 : Deep Q-Network and Deep Deterministic Policy Gradient

IOC/資科工碩

學號: 310551031

張皓雲

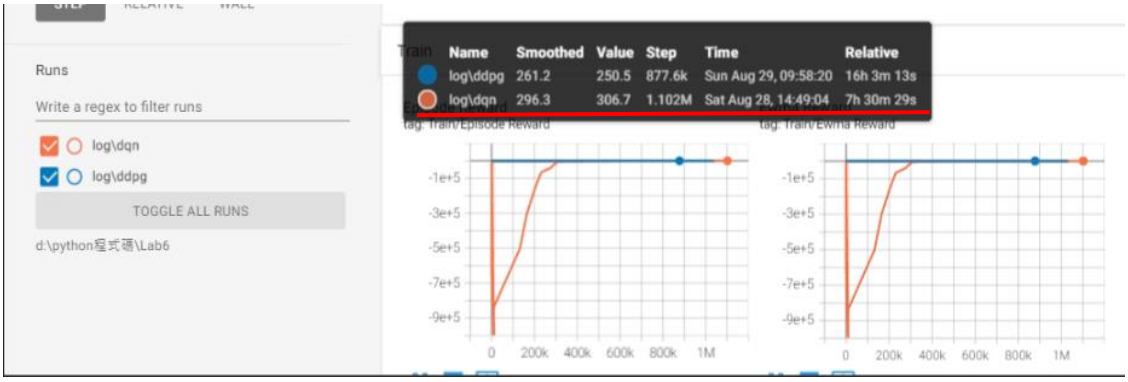
目錄

1. Introduction

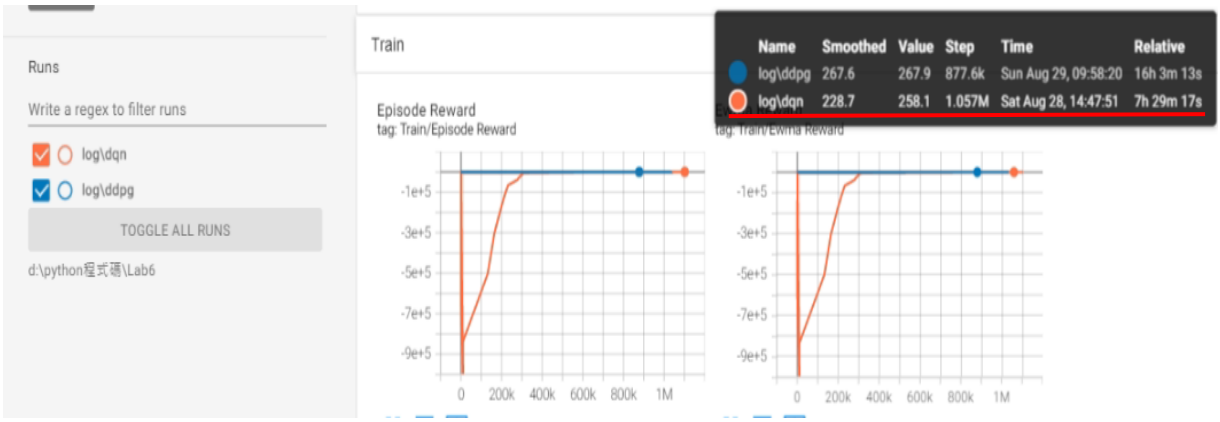
本實驗專注在實作 DQN 及 DDPG。其中 DQN 是應用在 LunarLander-v2，DDPG 則是應用在 LunarLanderContinuous-v2。本實驗在 **LunarLander-v2** 測試的最好的平均 10 個 reward 的結果為 **270.02**，本實驗在 **LunarLanderContinuous-v2** 測試的最好的平均 10 個 reward 的結果為 **285.22**。本實驗分別針對 LunarLander-v2 及 LunarLanderContinuous-v2 進行測試並拍下影片上傳至 youtube 以供證明。DQN 測試 LunarLander-v2 的 youtube 連結為 https://youtu.be/Kko3koeVd18，DDPG 測試 LunarLanderContinuous-v2 的 youtube 連結為 https://youtu.be/JWXbZfipZzw。

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

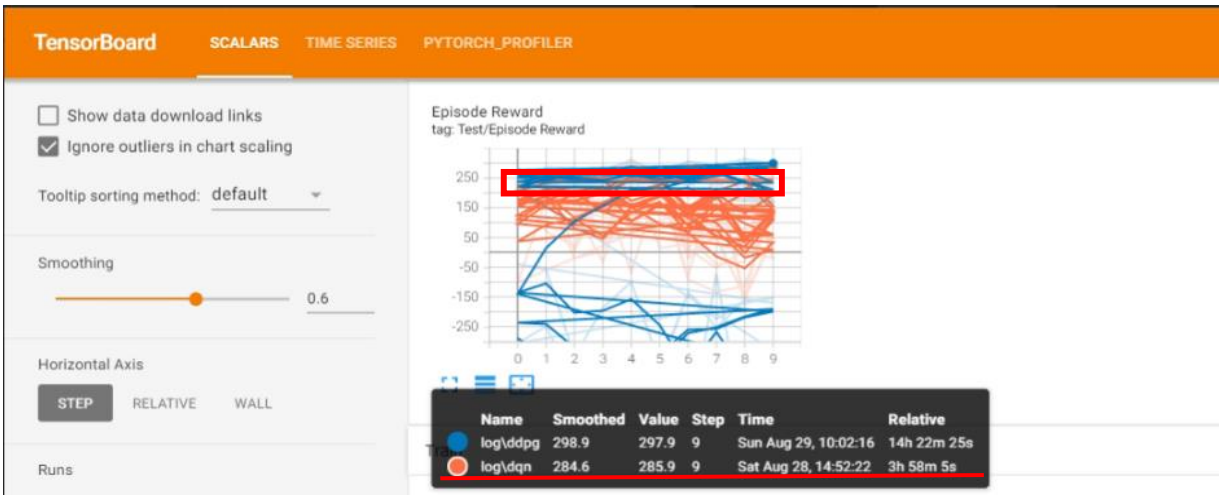
在第一部份中，本實驗是使用 DQN 進行 LunarLander-v2 的實作，訓練實作分別得到的 total_reward 及 ewma_reward 方別可以對應到**圖一**及**圖二**的橘色曲線圖。可以看到雖然 total_reward 與 ewma_reward 在一開始的值都是負的，並在-200 左右。但是隨著 steps 持續增加，模型開始知道遊戲角色的操控要怎麼做才會適當。因此直到 400000 個 steps 之後，total_reward 與 ewma_reward 逐漸穩定，並可以看到**圖一**及**圖二**中得知 total_reward 與 ewma_reward 皆為正值，其中 total_reward 落在 300 左右，ewma_reward 落在 258 左右。然而測試得到的 reward 則可以看到**圖三**。在**圖三**中，可以發現到測試 DQN 得到的 10 個 total_reward 數值會落在 260~280 之間，**平均的 10 個 total_reward 測試數值最後落在 270.207 左右**(本實驗在 DQN 中最好的數值，在後續部份有提供 youtube 影片連結證明)。由於本實驗做了許多實驗，因此**每個實驗的圖幾乎都是混在一起的，因此可能會看到前面實驗的數值(Source code 裡面有付 log 日誌可以參考)**。其中在這次實驗中，本實驗將 DQN 的訓練 episode 改成 2000，讓小機器人可以學到多一點的資訊。



圖一、DQN_Total_Reward 在訓練時的曲線圖(看到橘色曲線就好)



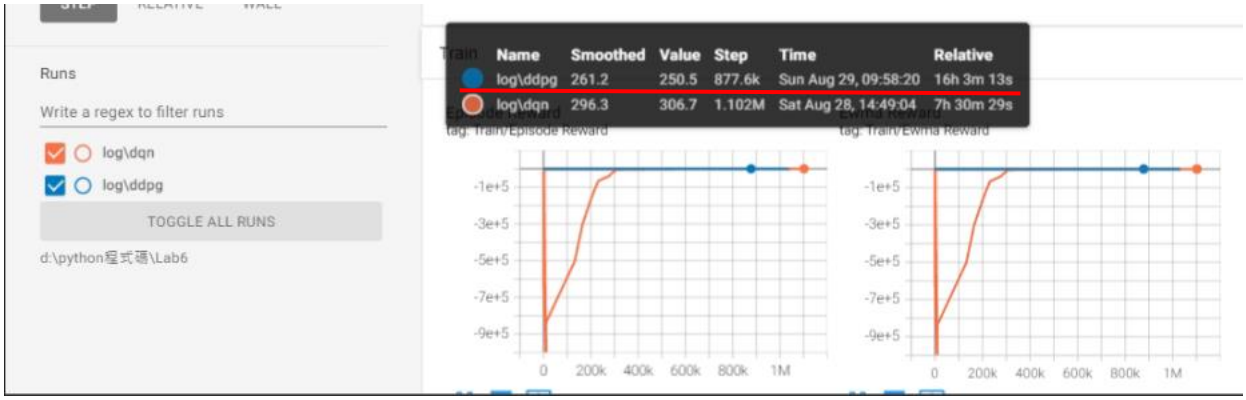
圖二、DQN_Ewma_Reward 在訓練時的曲線圖(看到橘色曲線就好)



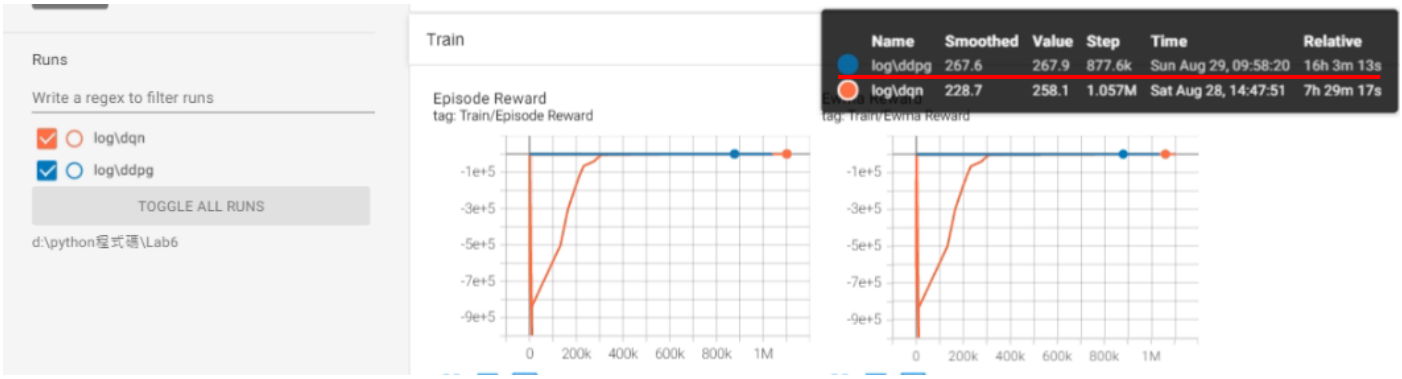
圖三、DQN_Total_Reward 在測試時的曲線圖(看到橘色曲線就好)

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

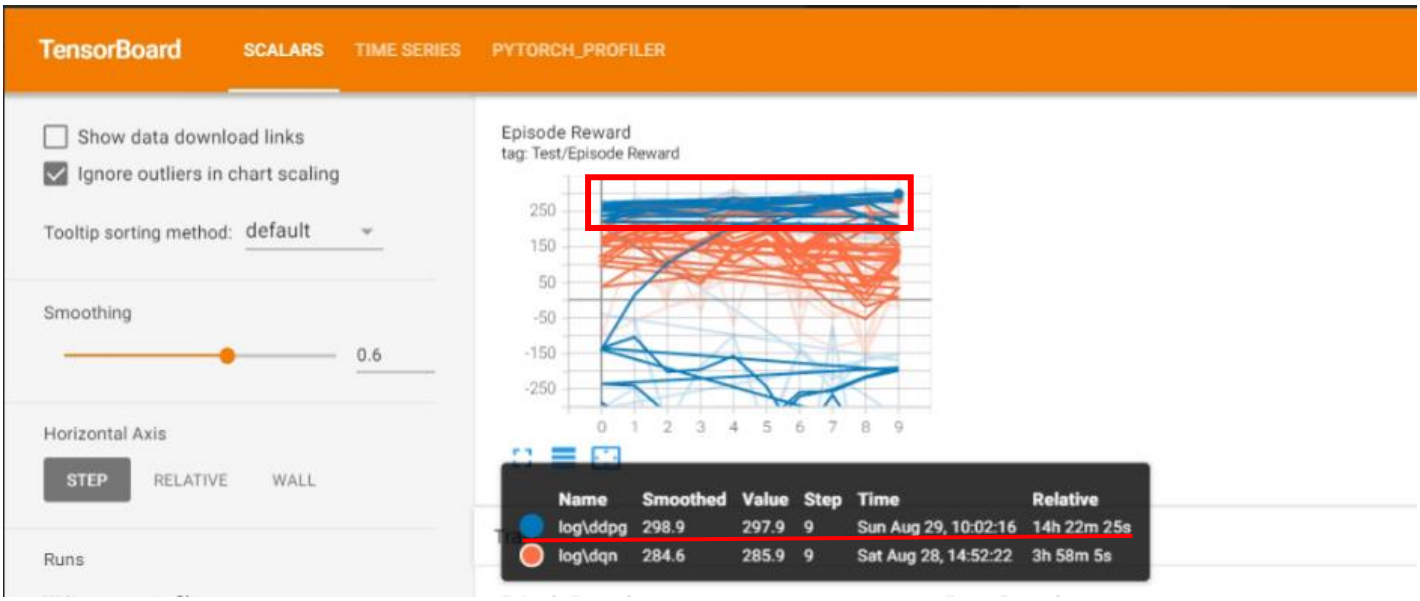
在第二部份中，本實驗是使用 DDPG 進行 LunarLanderContinuous-v2 的實作，訓練實作分別得到的 total_reward 及 ewma_reward 方別可以對應到**圖四**及**圖五**的藍色曲線圖。可以看到雖然 total_reward 與 ewma_reward 在一開始的值都是負的，並在-200 左右。但是隨著 steps 持續增加，模型開始知道遊戲角色的操控要怎麼做才會適當。因此 total_reward 與 ewma_reward 再後續逐漸增加之後趨於穩定，並可以看到**圖四**及**圖五**後發現 total_reward 與 ewma_reward 皆為正值，其中 total_reward 落在 250 左右，ewma_reward 落在 270 左右。然而測試得到的 reward 則可以看到**圖六**。在**圖六**中，可以發現到使用 DDPG 得到的 10 個 total_reward 數值會落在 285 附近，**平均的 10 個 total_reward 測試數值最後落在 285.22 左右**(本實驗在 DDPG 中最好的數值，在後續部份有提供 youtube 影片連結證明)。由於本實驗做了許多實驗，因此**每個實驗的圖幾乎都是混在一起的，因此可能會看到前面實驗的數值(Source code 裡面有付 log 日誌可以參考)**。其中本實驗針對原始的些微參數進行修改，分別是 warm_up 數量、訓練 episode 數量及 batch_size 進行修改。後續這三個數值分別被改為 50000、2800 及 128。



圖四、DDPG_Total_Reward 在訓練時的曲線圖(看到藍色曲線就好)



圖五、DDPG_Ewma_Reward 在訓練時的曲線圖(看到藍色曲線就好)



圖六、DDPG_Total_Reward 在測試時的曲線圖(看到藍色曲線就好)

4. Describe your major implementation of both algorithms in detail.

在此部份中，本實驗會分為 DQN 及 DDPG 兩個實驗的實作進行個別的介紹，其主要內容是在講述 pseudocode 與實際使用 python 語法之間的對照。其中 DQN 是用來實作 LunarLander-v2，DDPG 則是用來實作 LunarLanderContinuous-v2。在實作兩種方法時，皆是使用到 Adam optimizers。

4.1. DQN

4.1.1. Network 的設計

Network 的設計可以先看到**圖七**這張圖，**圖七**中說明 Network 的設計方法以及細部細節需要怎麼處理(包含 input、output 及 hidden 數量等)，因此本實驗會依照**圖七**進行 Network 的實作，**圖八**為 Network 的實作結果，可以看到每一層的 Network 都是按照**圖七**來完成的。最後這個 Network 將會成為 Behavior_Net 與 Target_Net 兩種 Net 的基礎架構。

Network Architecture

- Input: an 8-dimension observation (not an image)
- First layer: fully connected layer (ReLU)
  - input: 8, output: 32
- Second layer: fully connected layer (ReLU)
  - input: 32, output: 32
- Third layer: fully connected layer
  - input: 32, output: 4

圖七、Network 架構的文字說明

```python
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)

    def forward(self, x):
        ## TODO ##

        A1 = F.relu(self.fc1(x))
        A2 = F.relu(self.fc2(A1))
        action_pros = self.fc3(A2)

        return action_pros
```

圖八、Network 的實作結果

4.1.2. Select_Action 的實踐

在實作 Select_Action 的動作前，可以先參考到**圖九**紅色底線的說明。紅色底線說在實驗中，我們必須要使用一個 epilson 的值去決定一個隨機的 t 時間點下的 action，否則就需使用到該時間點下的 state 及 behavior_net 去決定 t 時間點下的 action。因此不難看出，action 這個值主要是由 eplison 這個值決定要用隨機的 action 或是要由 behavior_net 決定下一個 action。因此對應到**圖十**的實作內容，**圖十**中的紅色框框就是使用 eplison 來決定這一輪的 action 是要隨機的還是由 behavior_net 來決定，如果該輪的 action 是隨機決定的，此處就會從上、下、左、右四種 action 中，隨機挑一個使用。如果該輪的 action 是由 behavior_net 來決定的，那就將這一輪的 state 輸入至 behavior_net 來決定 action 要使用哪一個，由 behavior_net 決定的部份可以對應至圖十的橘色框框。

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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

圖九、Select Action 的 pseudocode

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)

    self._behavior_net.eval()
    with torch.no_grad():
        action_values = self._behavior_net(state)
    self._behavior_net.train()

    if random.random() > epsilon:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.freq))
```

圖十、Select Action 的實作內容

4.1.3. Update_Behavior_Network 的實踐

在實作 Update_Behavior_Network 的動作前，可以先參考至**圖十一** pseudocode 紅色框框，在紅色框框中主要可以分為三個步驟。第一個步驟是從 memory 中隨機抽出 state、action、reward、next_state 及 done 等五個參數。第二個步驟則是算出 Q 估計值及 Q 實際值。第三個步驟則是將 Q 估計值與 Q 實際值做 Loss 的運算，最後再將此 Loss 值做 backward。因此接下來看到**圖十二**的使用 python 語法實作 pseudocode 的部份。**圖十二**中的紅色框部份就是隨機抽取模型訓練需要的 state、action、reward、next_state 及 done 五個參數。**圖十二**的黃色框部份就是算出 Q 估計值及 Q 實際值，主要就是想要透過**圖十二**中的 q_value 跟 q_target 去做 Loss 值的運算。其中 q_target 中的 gamma 就是想要衰減多少 reward 的意思，q_value 主要是從 behavior_net 來的，q_target 主要是從 target_net 來的。**圖十二**的綠

色框部份就是 Loss 值的運算，本實驗在此階段使用到 MSE 這個 Loss Function 進行 Q 估計值與 Q 實際值的 Loss 值運算，最後再將 Loss 結果做 backward。在此步驟中，本實驗是每 4 個 step 才更新一次 Behavior_Network。

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

圖十一、Update_Behavior_Network 的 pseudocode

```
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)

    state = state.to(self.device).float()
    action = action.to(self.device).long()
    reward = reward.to(self.device).float()
    next_state = next_state.to(self.device).float()
    done = done.to(self.device).float()
    self._behavior_net = self._behavior_net.to(self.device)
    self._target_net = self._target_net.to(self.device)

    ## TODO ##
    q_value = self._behavior_net(state).gather(1, action)
    with torch.no_grad():
        q_next = self._target_net(next_state).detach().max(dim=1)[0].unsqueeze(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()
```

圖十二、Update_Behavior_Network 的實作

4.1.4. Update_Target_Network 的實踐

在實作 Update_Target_Network 的動作前，可以先參考到**圖十三**紅色底線的說明。紅色底線的意思就是將 behavior_net 的 parameters 的所有 data 傳給 target_net。因此在這部份的實作可以看到**圖十四**的 for 迴圈來進行，在**圖十四**中可以看到需要進行的動作就是將 behavior_net 的 parameters 取出，並傳給 target_net 的每一層。在此步驟中，本實驗是每 10000 個 step 才更新一次 Target_Network。

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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

圖十三、Update_Target_Network 的 pseudocode

```
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    for target_param, local_param in zip(self._target_net.parameters(), self._behavior_net.parameters()):
        target_param.data.copy_(local_param.data)
```

圖十四、Update_Target_Network 的實作

4.2. DDPG

4.2.1. Reply Buffer Sample 的實踐

如**圖十五**的紅色底線所示。紅色底線說明，在實作過程中，需要依據給定的 minibatch N 來抽取給模型訓練使用的 transitions。**圖十五**的實作部份如**圖十六**所示，在**圖十六**中，transitions 都被儲存在 self.buffer 裡面，因此指需要使用 random.sample 就可以抽出指定 batch size 大小的 transitions，並將此 transitions 用在模型訓練中。

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

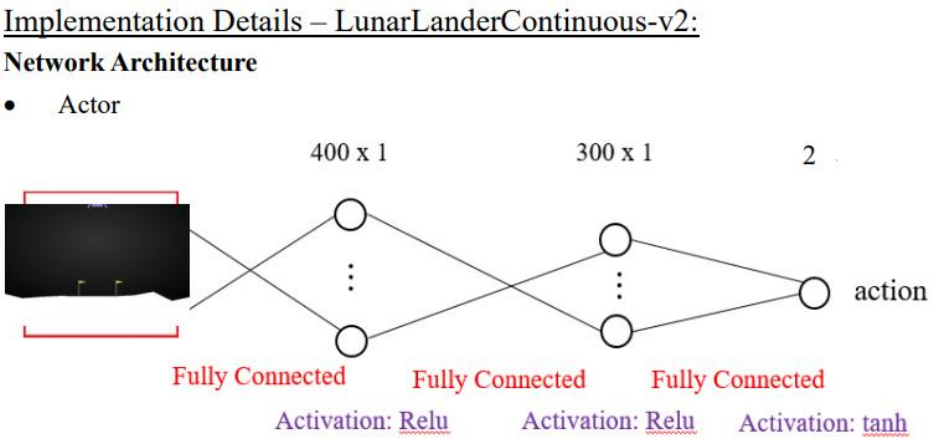
圖十五、Reply Buffer Sample 的 pseudocode

```
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'''
        ## TODO ##
        transitions = random.sample(self.buffer, batch_size)
        return (torch.tensor(x, dtype=torch.float, device=device)
                for x in zip(*transitions))
```

圖十六、Reply Buffer Sample 的實作

4.2.2. ActorNet 的實踐

ActorNet 的架構可以對應至**圖十七**，在圖十七中，可以看到模型總共有六層，分別是由 FullyConnected、ReLU 及 Tanh 組成的，因此模型僅需依據**圖十七**進行模型的建構即可。**圖十八為**建構後的模型的樣子。其中特別需要注意的有兩個地方，第一個地方在於最後一層的輸出要是 Tanh，由於在 LunarLanderContinuous-v2 的遊戲中，控制遊戲角色的數值位於-1~1 之間，因此最後的輸出會需要使用 Tanh。第二個地方在於遊戲角色的 action 是一次兩個值，例如[xxx,xxx]，這才能操控一次角色的動作，因此最後輸出的維度要是 2。



圖十七、ActorNet 的架構

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.actor_head = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
        )
        self.actor = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
        )
        self.actor_output = nn.Sequential(
            nn.Linear(h2, action_dim),
            nn.Tanh(),
        )

    def forward(self, x):
        x = self.actor_head(x)
        x = self.actor(x)
        x = self.actor_output(x)
        return x
```

圖十八、ActorNet 的實作

4.2.3. Select_Action 的實踐

Select_Action 的 pseudocode 可以對應至**圖十九**的紅色底線。**圖十九**的紅色底線說明 action 的選擇是需要依據 actor network 以及 gaussian noise 來決定。因此對應的實作內容可以對應至**圖二十**，在**圖二十**中可以發現到，在此處的實作中會需要使用 gaussian noise，而且依據 pseudocode 的解釋是直接將此 gaussian noise 直接加上 actor net 的分類結果，因此實作步驟總共可以分為三點，分別是先使用 state 作為 actor net 的輸入分類出一個 action，再將此 action 與 gaussain noise 相加，最後由於遊戲的 action 數值是位於-1~1 之間，因此再將最後結果壓在-1~1 之間。

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

圖十九、Select_Action 的 pseudocode

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##

    state = torch.FloatTensor(state.reshape(1, -1)).to(self.device)
    with torch.no_grad():
        selected_action = self._actor_net(state).cpu().detach().numpy().flatten()

    if noise:
        add_noise = self._action_noise.sample()
        selected_action = np.clip(selected_action + add_noise, -1.0, 1.0)

    return selected_action
```

圖二十、Select_Action 的實作

4.2.4. Update_Behavior_Network 的實踐

Update_Behavior_Network 的 pseudocode 可以對應至**圖二十一**的紅色框框。主要可以分為三個步驟需要處理。第一個步驟是從 memory 抽出需要使用的 transitions，第二個步驟是 update critic net。第三個步驟是 update actor net。**圖二十二**的紅框、綠框及黃框就分別對應上述的三個動作。看到**圖二十二**的綠色框框可以得知在更新 critic net 的時候，target_actor net 與 target_critic net 使用到的輸入分別都是 next_state 與 next_action，只有 critict net 的輸入是這一輪的 state 跟 action。因此在綠色框框中的步驟主要就是希望得到 target_critic net 的結果，並將這個結果與 reward 等參數作結合，再透過 gamma 衰減 rewrad 值，最後再將此結果(q_target)與 critict net 的輸出(q_value)做 Loss 值的計算，其中使用到的 Loss Function 為 MSE。看到**圖二十二**的黃色框框可以得知在更新 actor net 的時候，則是使用 actor net 吐出的 action，並將此 action 與當前的 state 輸入 critic，得到一個輸出結果，由於輸入的資料是一個 batch，因此最後再使用 torch.mean 進行結果的運算。

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

圖二十一、Update_Behavior_Network 的 pseudocode

```python
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 = target_actor_net(next_state).detach()
        q_next = target_critic_net(next_state, a_next).detach()
        q_target = reward + gamma * (1-done) * q_next
    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()

    ## update actor ##
    # actor loss
    ## TODO ##
    action = actor_net(state)
    actor_loss = -torch.mean(critic_net(state, action))

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

圖二十二、Update_Behavior_Network 的實作

4.2.5. Update_Target_Network 的實踐

Update_Target_Network 的 pseudocode 可以對應至**圖二十三**的紅色框框，紅色框框中使用到的技術為 Soft update。傳統的模型參數更新方式為 hard update，這種網路參數更新方式將會造成學習的不穩定。而 DDPG 中需要更新的 target network 主要有兩種。第一種是將 actor net 的參數更新到 target_actor net。第二種是將 critic net 的參數更新到 target_critic net。更新這兩種 target network 的方法實作可以參考至**圖二十四**。在**圖二十四**中可以發現到 Soft update 的更新方式需要有一個參數 tau 去做支持，在此處的 tau 數值為 0.005，使用這個參數的目的是為了不讓所有的參數都更新 target network 上。因此此方法將會保持一些原有 target network 的參數，目的是為了能夠提高學習的穩定性。

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

圖二十三、Update_Target_Network 的 pseudocode

```
@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_(tau * behavior.data + (1.0 - tau) * target.data)
```

圖二十四、Update_Target_Network 的實作

5. Describe differences between your implementation and algorithms.

在此部份中，本實驗主要是在講述，在實作中有哪些是額外需要處理的項目。其中 DQN 是用來實作 LunarLander-v2，DDPG 則是用來實作 LunarLanderContinuous-v2。

5.1. Warmup 的設定

在 algorithm 中並沒有提及要使用 warmup。但是在實作中為了讓模型能夠抽出 transition 以供模型去進行使用，因此在實作時，本實驗會需要隨機先將 state、action、reward 等資料儲放在 memory 中，以便後續在訓練模型的時候，可以將資料從 memory 中抽取出來。

5.2. ewma_reward 的使用

在 algorithm 中並沒有提及要使用 ewma_reward。但是在實作中為了讓使用者能夠同時考慮到在這一輪中的 total_reward 跟上一輪的 total_reward 是不是呈現正向發展(上一輪的 total_reward 增加，這一輪的 total_reward 比上一輪的 total_reward 還要大)的關係，因此在實作的部分中，需要將 ewma_reward 加入實驗一併進行考慮，就能知道訓練完的模型是不是真的適合應用在這個遊戲內。

5.3. DQN 中 eplison 數值的更新

在實作 DQN algorithm 中，pseudocode 僅有提到要依據 eplison 數值去選擇使用 action。但是在實作過程中，本實驗卻需要設定 eplison decay 去讓 eplison 的數值得以下降，或是設定 eplison minimum eplison 的數值。使用 eplison decay 及 eplison minimum 數值的原因是因為，在一開始訓練得到的 DQN 模型是不穩定的，因此會需要先隨機使用 action 來訓練。但隨著 DQN 模型越來越穩定，就可以漸漸開始使用 behavior net 得到的 action 來進行 state 的回饋，就能夠得到較好的訓練結果。主要的概念就是希望能夠在 exploration 和 exploitation 之間進行切換。

5.4. DQN 中使用 clip_grad_norm

在實作 DQN algorithm 中，pseudocode 沒有提到要使用 clip_grad_norm。但在訓練 DQN 模型時，使用 clip_grad_norm 將會成為測試結果變得更高的一個關鍵。在實作 clip_grad_norm 時，可以將其想像成是在模型裡面使用 Dropout 那樣，clip_grad_norm 就相當於是會將梯度限制在使用者指定的範圍內，因此這樣就可以減少過度擬合的問題產生。因為在訓練 DQN 模型時，可能會一直不斷的吃到重複的資料，因此 clip_grad_norm 就將會是解決此問題的好方法。

6. Describe your implementation and the gradient of actor updating.

Actor 的 gradient 更新的公式可以對應到**式(1)**到**式(3)**，其中**式(2)**的 Gai 為 critic net output 的 gradient，在此的 critic net input 又為 actor net 計算得到的 action。**式(3)**的 Gμi 為 actor net output 的 gradient。最後兩個 gradient 都會被用來 evaluate observation Si。**圖二十五**為本實驗實作的 gradient of actor updating，可以看到在 actor 的 gradient 中總共會需要 actor net 跟 critic net，首先先將這一輪的 state 輸入至 actor net 中，並讓其回傳一個 action。接著再將此 action 與這一輪的 state 讓 critit net 做決斷得到一個數值。最後由於輸入的資料是一個 batch 的資料，因此使用 torch.mean 將其作平均並使用負號最小化這個結果。

$$\nabla_{\theta_\mu} J \approx \frac{1}{M}\sum_{i=1}^{M} G_{ai} G_{\mu i} \qquad \textbf{式(1)}$$

$$G_{ai} = \nabla_A Q(S_i, A|\theta_Q) \quad \text{where } A = \mu(S_i|\theta_\mu)$$

式(2)

$$G_{\mu i} = \nabla_{\theta_\mu} \mu(S_i|\theta_\mu)$$ 式(3)

```
## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = -torch.mean(critic_net(state, action))

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

圖二十五、gradient of actor updating

7. Describe your implementation and the gradient of critic updating.

Critic gradient 的更新公式可以對應至**式(4)**及**式(5)**，**式(4)**表示的是 target critic net 的 Q 數值，**式(5)**表示的是 critic net 的 Q 數值，為了讓 reward 能夠更接近真實的情況，target critic net 將 reward 的值乘上 gamma，希望能讓 reward 的值不要那麼高。另外對應至**圖二十六**可以發現到，target critic net 使用的 input 都是 next 的 state 或是 next 的 action，而 critic net 使用的 input 都是這一輪的 state 及這一輪的 action，希望能夠最小化 target critic net 的 Q 數值與 critic net 的 Q 數值之間的差異，以優化 critic net 的結果。

$$y_i = r_i + \gamma Q'\left(s_{t+1}, \mu'\left(s_{t+1}|\theta^{\mu'}\right)|\theta^{Q'}\right)$$ 式(4)

$$L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$$ 式(5)

```
# 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 = target_actor_net(next_state).detach()
    q_next = target_critic_net(next_state, a_next).detach()
    q_target = reward + gamma * (1-done) * q_next
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()
```

圖二十六、gradient of critic updating

8. Explain effects of the discount factor.

discount factor 的使用時機如**式(6)**所示，在**式(6)**中 R(sx,ax)表示的是在第 x 輪的總共 reward 是多少，但如果將 discount factor(gamma，假設其數值為 0.0001)加進式子後就可以發現到在越後面的 reward，需要乘上的 gamma 數值會越小，因此那一輪的 reward 的值會越來越小。慢慢將式子進行疊加之後就可以發現到，越後面的 reward 數值就越容易被忽略(除非 reward 本身的數值就很大)。因此較為前面的 reward 就會成關鍵。但如果將 discount factor(gamma)的值接近 1 的話，後面的 reward 也會被考慮到。

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots$$ **式(6)**

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

使用 epsilon-greedy 的好處是為了取得 exploration 與 exploitation 之間的平衡，可以先短暫看到**圖二十七**的 Eplison-Greedy Action Selection 的內容，如果 eplison 的值過小，當前就會選擇使用 exploration。使用 exploration 的概念就是可以隨機選擇 action 去學習，這樣模型就能夠學習到新的東西。但如果 eplison 的值過大，當前就會選擇使用 exploitations。這時模型就會依據之前的記憶選擇一個最佳的 action。因此如果能同時平衡學習新東西及依據之前的記憶尋找最佳解，這樣就能達成一個好的循環。

Algorithm 2: Epsilon-Greedy Action Selection

Data: Q: Q-table generated so far, : a small number, S: current state

Result: Selected action

Function *SELECT-ACTION(Q, S, ϵ)* **is**

n ← uniform random number between 0 and 1;

if $n < \epsilon$ **then**

A ← random action from the action space;

else

A ← maxQ(S,.);

end

return selected action A;

end

圖二十七、Eplison-Greedy Action Selection

10. Explain the necessity of the target network.

使用 target network 主要可以解決 Bootstrapping 的問題，緩解 overestimate(也就是高估 Q 值的意思)，但並不能完全解決。因此為了解決這些問題，target network 主要是用來計算 TD target，並使用 target network 的預估值來代替原本網路的預估值。

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

replay buffer size 過大會造成 memory 中的 transition 樣本過多，因此如果現在隨機抽取一個 transition 來訓練，下一輪隨機取到的 transition 可能會跟這次使用的 transition 差很多，模型可能會需要更多的 step 或是 batch_size 來將 memory 中的 transition 都學過一遍。但如果模型真的將所有的 transition 都學完之後，模型就會變得超強。replay buffer size 如果過小的話會造成 memory 中的 transition 樣本過少。雖然模型能夠學得很快，但當模型用在測試的時候，可能會因為沒辦法考慮到更多的 state 而造成 action 無法對應到測試的 state。

12. Performance

12.1. [LunarLander-v2] Average reward of 10 testing episodes

圖二十八在 DQN 上的 10 個 reward 結果的平均，其結果為 270.207。**圖二十九**為本實驗使用 2000 個 episode 的訓練過程。本實驗在 DQN 上修改的部份僅有針對 eposide 的數量，其餘的 hyperparameters 都與實驗的初始設定數值相同。測試的實際 demo 可以參考至連結: https://youtu.be/Kko3koeVd18。

```
(Summer) D:\python程式碼\Lab6>python dqn-example2.py
Start Testing
Average Reward 270.2077245782799
```

圖二十八、DQN 測試結果(270.207)

```
Step: 1095034   Episode: 1976   Length: 283    Total reward: 278.58   Ewma reward: 244.56   Epsilon: 0.010
Step: 1095275   Episode: 1977   Length: 241    Total reward: 288.88   Ewma reward: 246.77   Epsilon: 0.010
Step: 1095589   Episode: 1978   Length: 314    Total reward: 286.96   Ewma reward: 248.78   Epsilon: 0.010
Step: 1095859   Episode: 1979   Length: 270    Total reward: 242.55   Ewma reward: 248.47   Epsilon: 0.010
Step: 1096123   Episode: 1980   Length: 264    Total reward: 231.80   Ewma reward: 247.64   Epsilon: 0.010
Step: 1096394   Episode: 1981   Length: 271    Total reward: 273.32   Ewma reward: 248.92   Epsilon: 0.010
Step: 1096669   Episode: 1982   Length: 275    Total reward: 258.47   Ewma reward: 249.40   Epsilon: 0.010
Step: 1096973   Episode: 1983   Length: 304    Total reward: 263.98   Ewma reward: 250.13   Epsilon: 0.010
Step: 1097221   Episode: 1984   Length: 248    Total reward: 226.47   Ewma reward: 248.94   Epsilon: 0.010
Step: 1097571   Episode: 1985   Length: 350    Total reward: 267.33   Ewma reward: 249.86   Epsilon: 0.010
Step: 1097843   Episode: 1986   Length: 272    Total reward: 250.27   Ewma reward: 249.88   Epsilon: 0.010
Step: 1098157   Episode: 1987   Length: 314    Total reward: 229.67   Ewma reward: 248.87   Epsilon: 0.010
Step: 1098396   Episode: 1988   Length: 239    Total reward: 303.85   Ewma reward: 251.62   Epsilon: 0.010
Step: 1098729   Episode: 1989   Length: 333    Total reward: 275.17   Ewma reward: 252.80   Epsilon: 0.010
Step: 1099032   Episode: 1990   Length: 303    Total reward: 283.13   Ewma reward: 254.32   Epsilon: 0.010
Step: 1099313   Episode: 1991   Length: 281    Total reward: 266.61   Ewma reward: 254.93   Epsilon: 0.010
Step: 1099637   Episode: 1992   Length: 324    Total reward: 282.34   Ewma reward: 256.30   Epsilon: 0.010
Step: 1099937   Episode: 1993   Length: 300    Total reward: 284.43   Ewma reward: 257.71   Epsilon: 0.010
Step: 1100229   Episode: 1994   Length: 292    Total reward: 297.09   Ewma reward: 259.68   Epsilon: 0.010
Step: 1100504   Episode: 1995   Length: 275    Total reward: 248.49   Ewma reward: 259.12   Epsilon: 0.010
Step: 1100762   Episode: 1996   Length: 258    Total reward: 276.06   Ewma reward: 259.96   Epsilon: 0.010
```

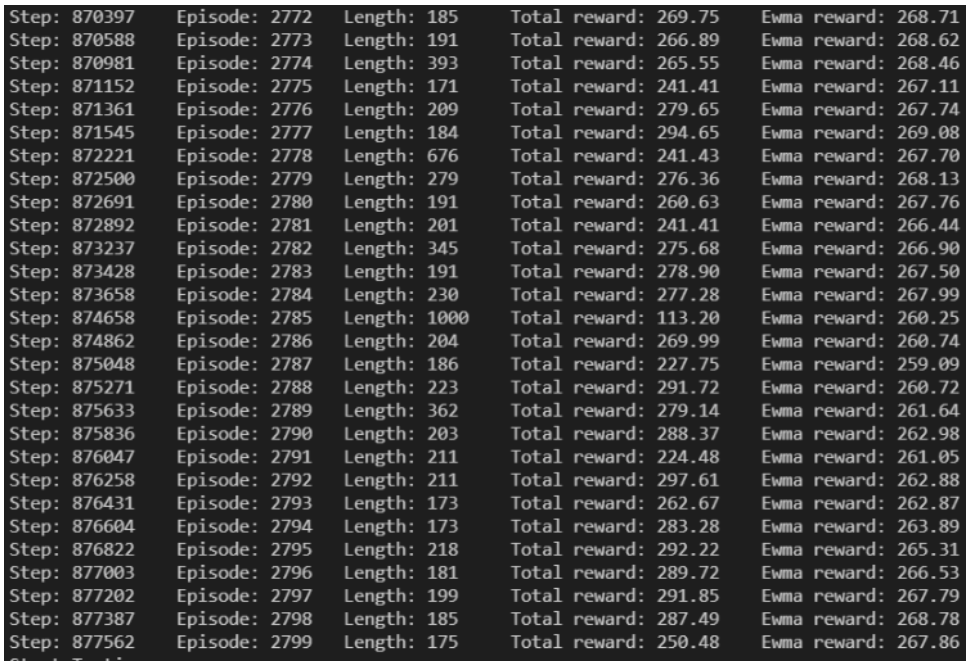
圖二十九、DQN 訓練過程

12.2. [LunarLanderContinuous-v2] Average reward of 10 testing episodes

圖三十在 DDPG 上的 10 個 reward 結果的平均，其結果為 285.22。**圖三十一**為本實驗使用 3000 個 episode、warmup 為 50000、batch_size 為 128 的訓練過程(原本的 episode 為 1200、warmup 為 10000、batch_size 為 64)，其餘的 hyperparameters 都與實驗的初始設定數值相同。本實驗將這三項參數更改為這些參數的原因為在測試時，機器人永遠只會用同樣的方式降落，因此本實驗希望透過增加樣本數的概念增強模型的 performance。測試的實際 demo 可以參考至連結: https://youtu.be/JWXbZfipZzw。

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

Step: 876822    Episode: 2795   Length: 218    Total reward: 292.22   Ewma reward: 265.31
Step: 877003    Episode: 2796   Length: 181    Total reward: 289.72   Ewma reward: 266.53
Step: 877202    Episode: 2797   Length: 199    Total reward: 291.85   Ewma reward: 267.79
Step: 877387    Episode: 2798   Length: 185    Total reward: 287.49   Ewma reward: 268.78
Step: 877562    Episode: 2799   Length: 175    Total reward: 250.48   Ewma reward: 267.86
Start Testing
Average Reward 285.2266716391417
```

圖三十、DDPG 測試結果(285.22)



圖三十一、DDPG 訓練過程