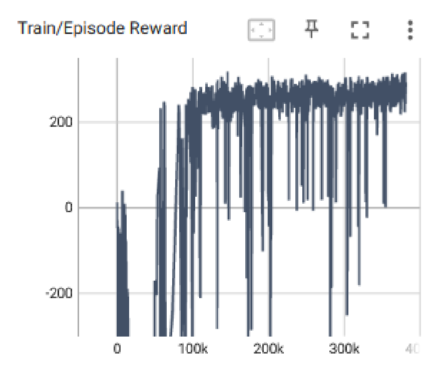
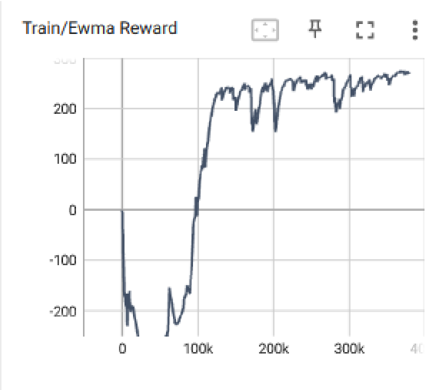
311551069 余忠旻 Lab6 : DQN and DDPG

◆ Experimental Results

Your screenshot of tensorboard and testing results on LunarLander-v2 using DQN.

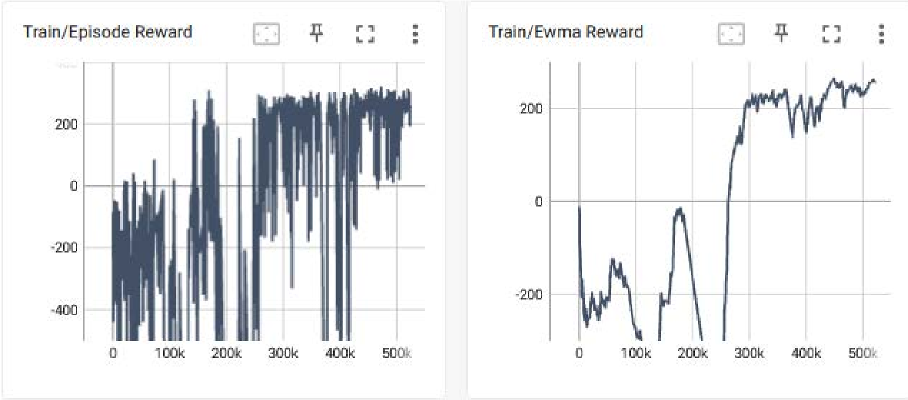
```
pp037@ec037:~/DLP/lab6$ python3 dqn.py --test_only
Start Testing
/home/pp037/.local/lib/python3.8/site-packages/gym/
 deprecated alias for `np.bool_`.  (Deprecated NumP
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 264.02
episode 2: 237.25
episode 3: 257.86
episode 4: 298.76
episode 5: 270.63
episode 6: 315.03
episode 7: 311.92
episode 8: 282.32
episode 9: 277.53
episode 10: 262.62
Average Reward 277.7947256785025
```





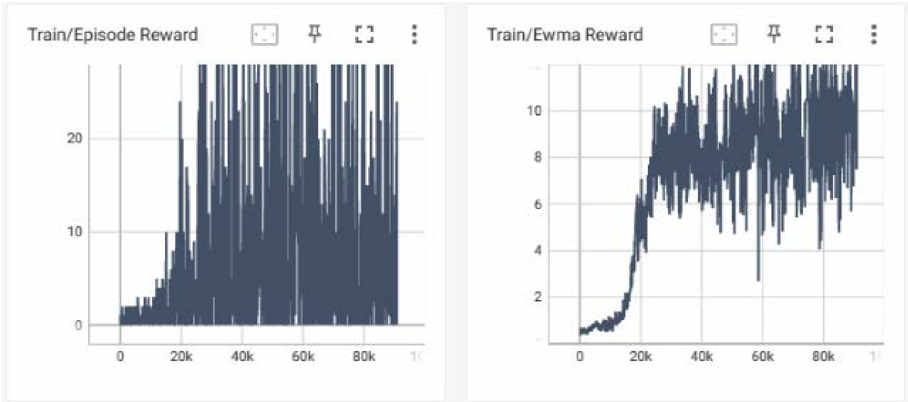
Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2 using DDPG.

```
pp037@ec037:~/DLP/lab6$ python3 ddpg.py --test_only
Start Testing
/home/pp037/.local/lib/python3.8/site-packages/gym/u
 deprecated alias for `np.bool_`.  (Deprecated NumPy
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 264.83
episode 2: 215.84
episode 3: 209.07
episode 4: 318.63
episode 5: 252.21
episode 6: 314.62
episode 7: 316.99
episode 8: 289.42
episode 9: 237.63
episode 10: 260.75
Average Reward 268.0003020433581
```



Your screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4.

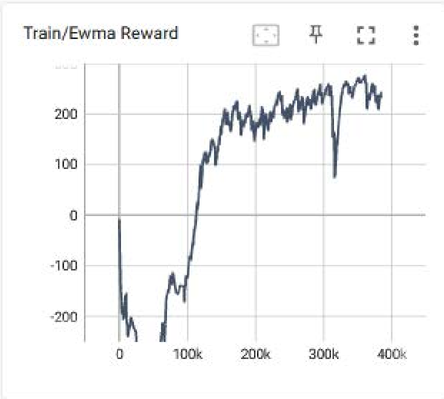
```
(pytorch-gpu) [redacted]\DLP_lab6>python dqn_breakout.py --test_only
Start Testing
episode 1: 419.00
episode 2: 791.00
episode 3: 471.00
episode 4: 341.00
episode 5: 397.00
episode 6: 419.00
episode 7: 423.00
episode 8: 455.00
episode 9: 416.00
episode 10: 413.00
Average Reward: 454.50
```



(Bonus)

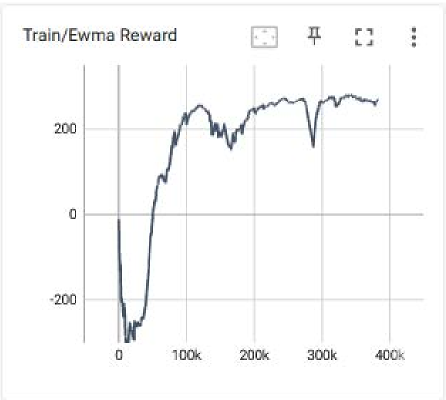
Your screenshot of tensorboard and testing results on LunarLander-v2 using DDQN.

```
pp037@ec037:~/DLP/lab6$ python3 ddqn.py --test_only
Start Testing
/home/pp037/.local/lib/python3.8/site-packages/gym/
 deprecated alias for `np.bool_`.  (Deprecated NumPy
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 262.63
episode 2: 250.85
episode 3: 215.90
episode 4: 308.46
episode 5: 277.35
episode 6: 305.38
episode 7: 313.41
episode 8: 284.96
episode 9: 287.07
episode 10: 263.56
Average Reward 276.9566147815968
```



Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2 using TD3.

```
pp037@ec037:~/DLP/lab6$ python3 td3.py --test_only
Start Testing
/home/pp037/.local/lib/python3.8/site-packages/gym/
 deprecated alias for `np.bool_`.  (Deprecated NumP
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 253.77
episode 2: 256.93
episode 3: 257.71
episode 4: 309.51
episode 5: 271.75
episode 6: 311.09
episode 7: 310.58
episode 8: 281.11
episode 9: 282.83
episode 10: 254.88
Average Reward 279.0165834476812
```



◆ Questions

1. Describe your major implementation of both DQN and DDPG 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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 ##
    # With probability eps select a random action
    if random.random() < epsilon:
        return action_space.sample() # from OpenAI gym
    # With probability (1-eps) select a max Q from behavior net
    else:
        # convert state to one row, find the maximum Q in the row and return corresponding index
        return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()
```

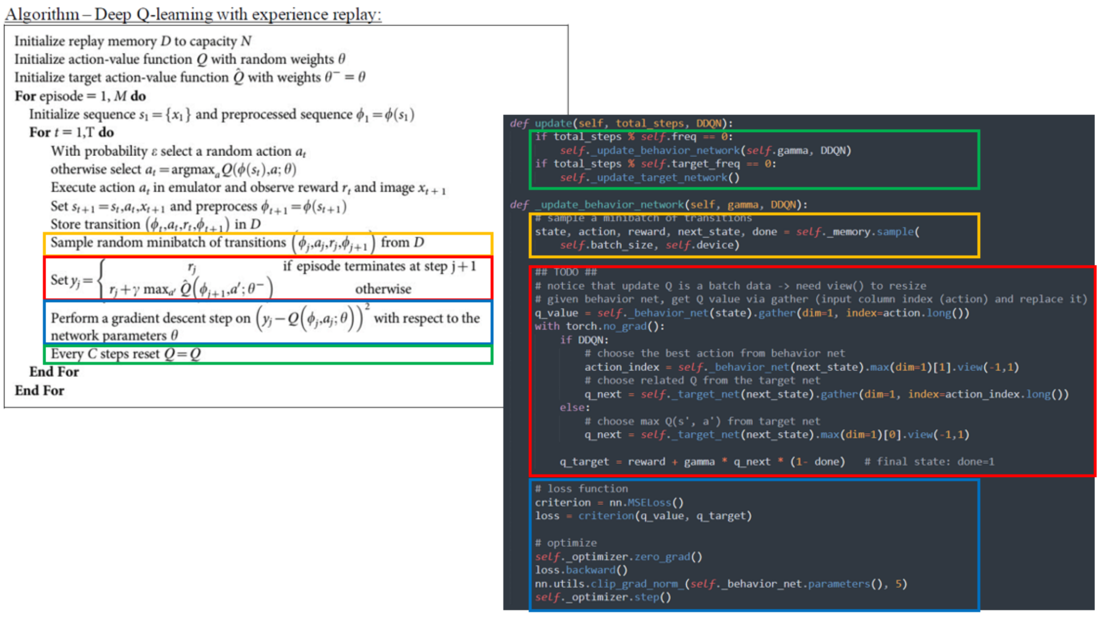
上圖框框是 epsilon greedy 的部分。有 epsilon 的機率會隨機選擇一個 action (紅色框)。用 OpenAI gym 環境提供的 action_space.sample() 函式可以從環境的 action space 隨機 sample 一個 action 出來；反之，要從 behavior net 中找出 Q 值最大的 action (藍色框)。

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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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
# 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)
```

接著是上圖淺藍色框執行 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。



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

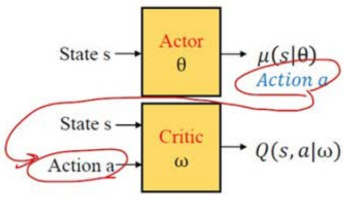
```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            sample_noise = torch.from_numpy(self._action_noise.sample()).view(1,-1).to(self.device)
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
            action = action + sample_noise
        else:
            # convert state to one row and feed on action net. convert tensor to 1-D numpy array.(via squeeze)
            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)   # final state: done=1

# critic loss function
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
# select action a from behavior actor network (a is different from sample transition's action)
# get Q from behavior critic network, mean Q value -> objective function
# maximize (objective function) = minimize -1 * (objective function)
## TODO ##
action = self._actor_net(state)
actor_loss = -1 * (self._critic_net(state, action).mean())

# 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，也就是上圖橘色框框部分。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。

2. 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:

- 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。

3. 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)。

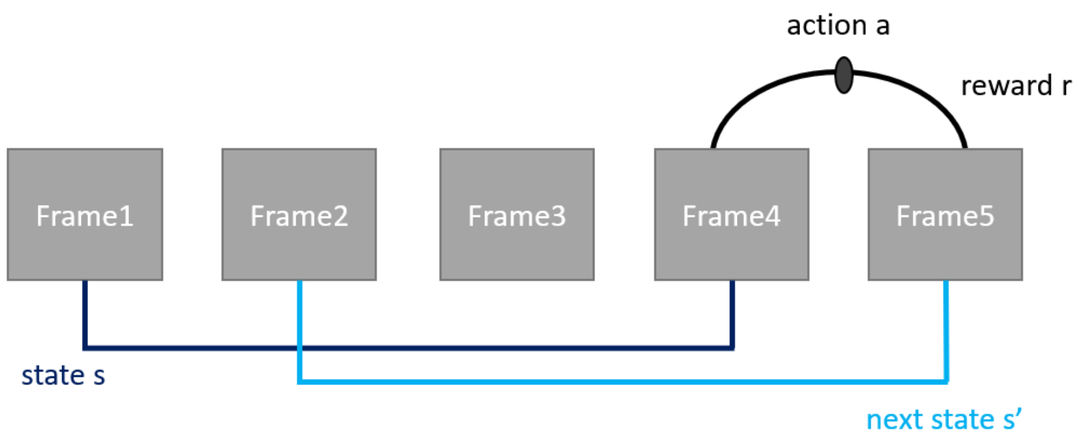
4. Explain the necessity of the target network.

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。

5. Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander.

Breakout 實作部分和 LunarLander 差異最大的部分就是它是吃整個圖片進去，因此我們需要 Stack a sequence of four frames together，這樣才能觀察到球的運動軌跡用來訓練，因此我們需要在 ReplayMemory 上多增加一些 trick。

我們在 ReplayMemory 要存 state, action, reward, next_state, done，而一個 state 是由 four stacked frames 組成的。因此我用 deque(maxlen=5) 用來暫存 frame，再把 deque 中的 frame 連同 action, reward, done 放進 ReplayMemory 處理，ReplayMemory 會將這五個 frame 拆成前四和後四個 frames 當成 state 和 next_state，如下圖所示:



另外，在遊戲一開始 deque 中還沒有 frame，因此我們會先讓它做幾次 no-op 去蒐集 frame(也就是填滿 deque)。而在 select action 時，deque 中後四個 frame 是現在的 state，我們會依據這個 state 去挑選 action。其餘部分和 LunarLander 實作部分大致相同，我就不加以贅述。