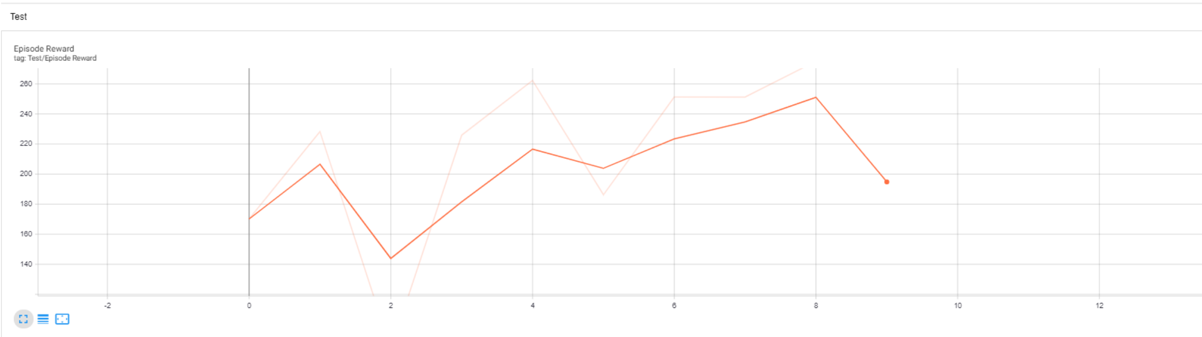
DLP Lab6
309553012
黃建洲

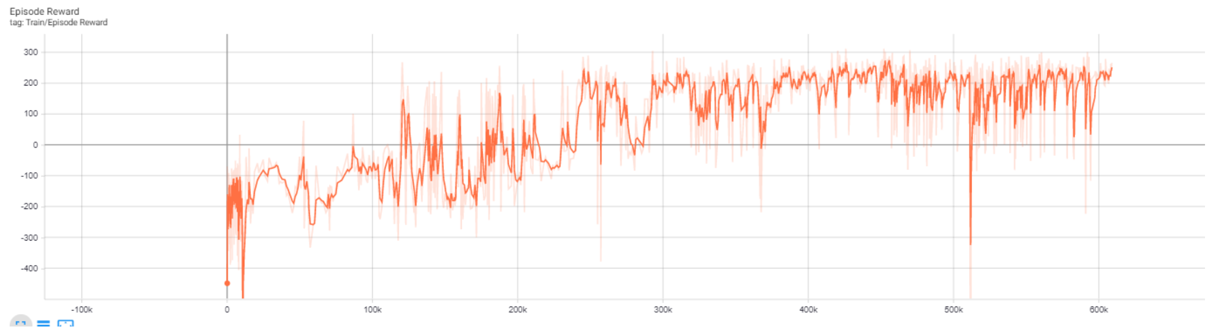
1. Result

dqn

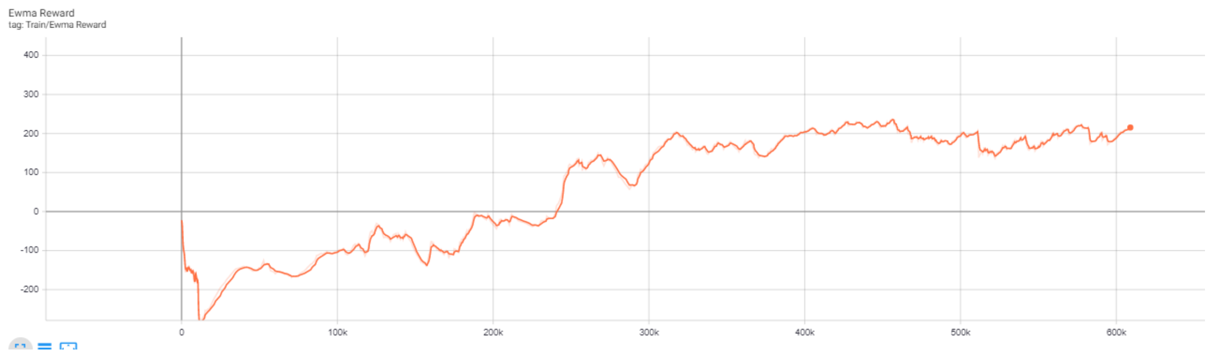
(1) Test



(2) Episode Reward



(3) Elwa Reward

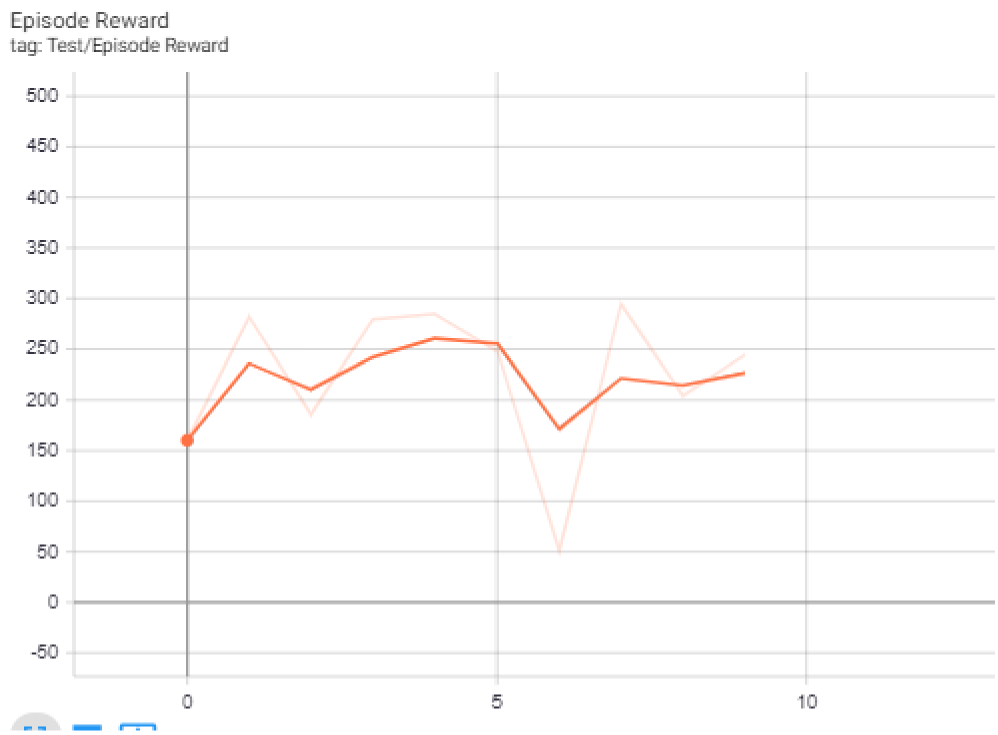


(4) Test Average Reward: 204.56

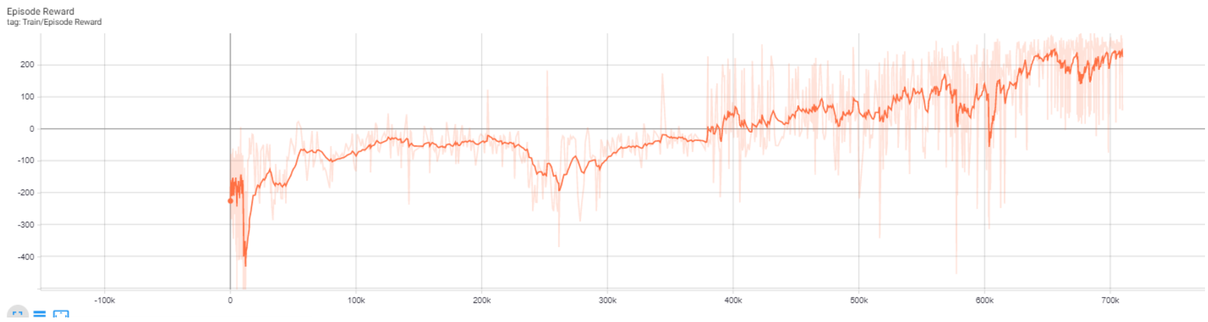
```
Step: 608990    Episode: 1199    Length: 366    Total reward: 265.95    Ewma reward: 218.18    Epsilon: 0.010
Start Testing
Average Reward 204.56857582614546

(pytorch) D:\desktop\dl_lab6>
```

ddqn

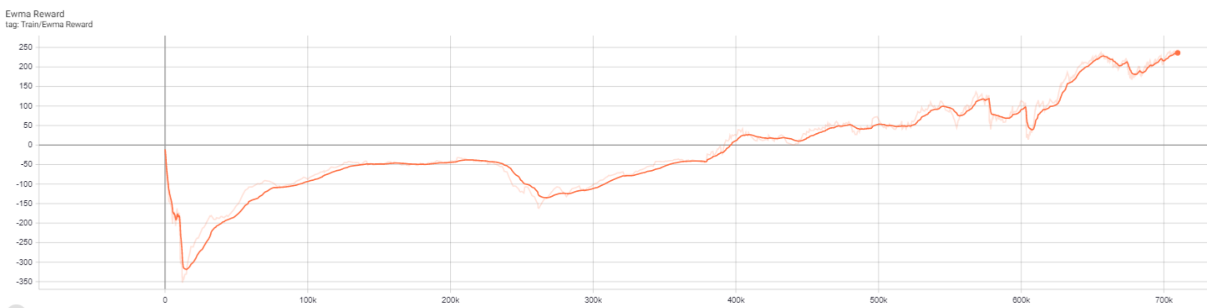
(1) Test



(2) Episode Reward



(3) Elwa Reward

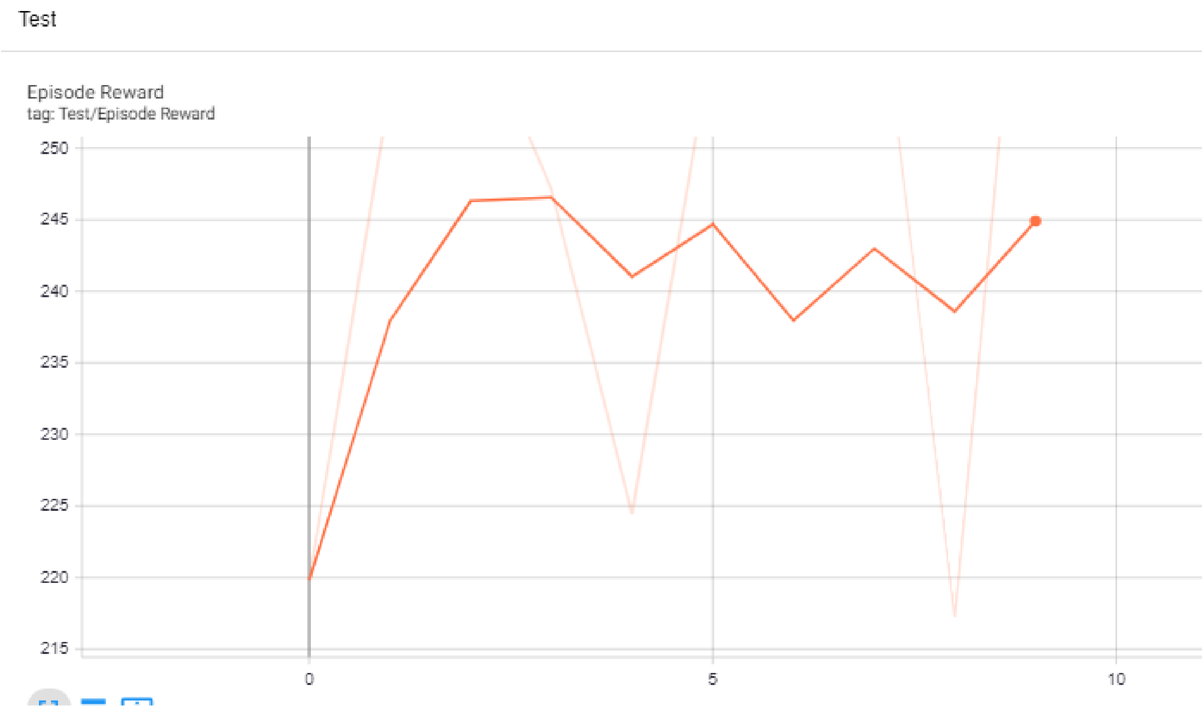


(4) Test Average Reward: 223.42

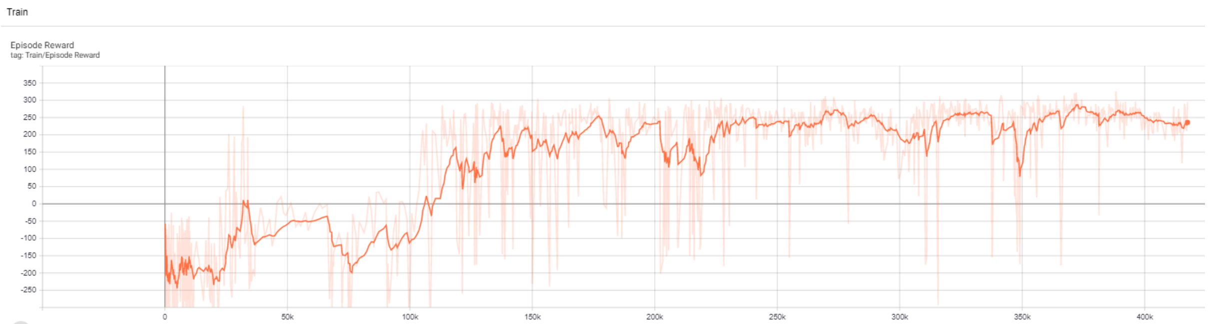
```
Step: 709679    Episode: 1199   Length:
Start Testing
Average Reward 223.42262112369548
```

ddpg:

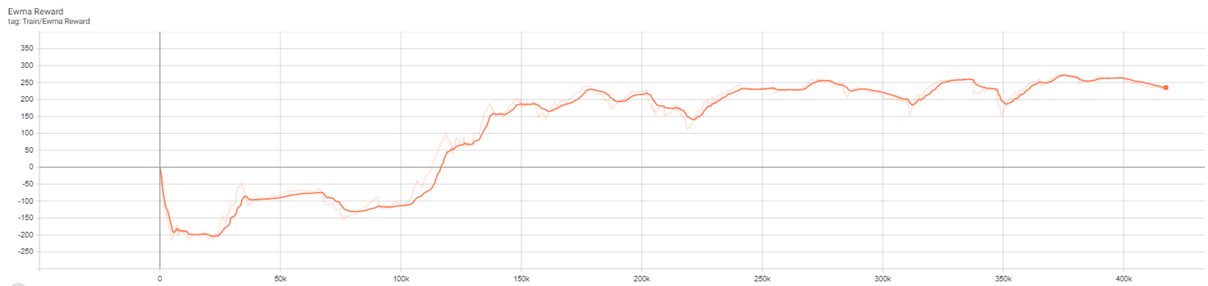
(1) Test:



(2) Episode Reward



(3) Elwa Reward



(4) Test Average Reward: 243.47

```
Step: 417490    Episode: 1199   Length: 279     Total reward: 294.88    Ewma reward: 235.49
Start Testing
[219.82496490038534, 254.00719274787846, 260.3636954112194, 247.15751532728504, 224.4380304889064, 257.67706832683746,
10.74288756409086, 265.4936725403695, 217.24635266796687, 277.78808969161344]
Average Reward 243.4739469666553
```

2. Describe your major implementation of both algorithms in detail

DQN

(1) Network部分:

Implementation Details – LunarLander-v2:

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

Training Hyper-Parameters

- Memory capacity (experience buffer size): 10000
- Batch size: 128
- Warmup steps: 10000
- Optimizer: Adam
- Learning rate: 0. 0005
- Epsilon: 1 → 0.1 or 1 → 0.01
- Gamma (discount factor): 0.99
- Update network evert 4 iterations
- Update target network every 100 iterations

根據PDF給予的網路架構，建構三層帶有ReLU的全連接層。
程式碼如下:

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        self.firstLayer = nn.Linear(state_dim, hidden_dim)
        self.secondLayer = nn.Linear(hidden_dim, hidden_dim)
        self.fcLayer = nn.Linear(hidden_dim, action_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.firstLayer(x)
        x = self.activation(x)
        x = self.secondLayer(x)
        x = self.activation(x)
        x = self.fcLayer(x)
        return x
```

(2) Select Action部分:

Deep Q-Network (DQN)

Algorithm 1 – 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

根據第一個藍框，將select action依照episilon的機率分成兩種:
第一種是隨機在4個動作中選取一個進行。
第二種則是透過behavior net來選擇動作。

(3) Update behavior network

根據第二個藍框，最主要我們要計算由behavior net和currenct state所得到的Q值，與target net和next state所得到的Q'值(加上reward和各項係數)之間的MSE LOSS，並利用這個Loss進行back propagation和update

(4) Testing部分:

設定Max step為1000，在每一個iteration中進行三個步驟，第一是讓agent進行select action，第二是根據選擇出的action對環境進行更新，最後計算reward並累加。

```
for n_episode, seed in enumerate(seeds):
    total_reward = 0
    env.seed(seed)
    state = env.reset()
    ## TODO ##
    # ...
    #     if done:
    #         writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
    #         ...
    for i in range(1000):
        action = agent.select_action(state,0,action_space)
        state,reward,done,_ = env.step(action)
        total_reward += reward
        if done:
            writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
            break
    rewards.append(total_reward)
print('Average Reward', np.mean(rewards))
env.close()
```

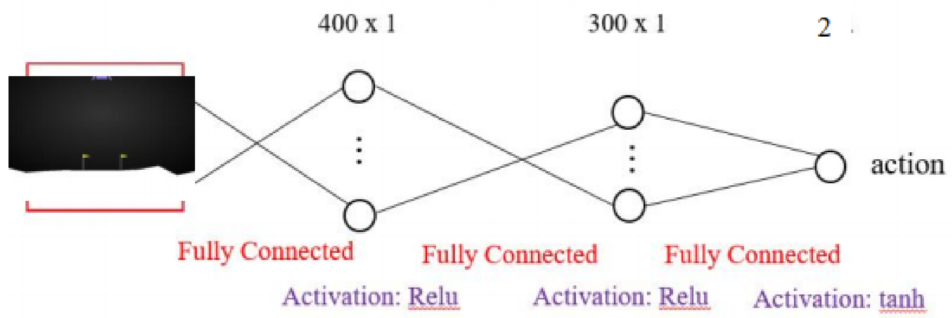
DDPG

(1) Network部分:

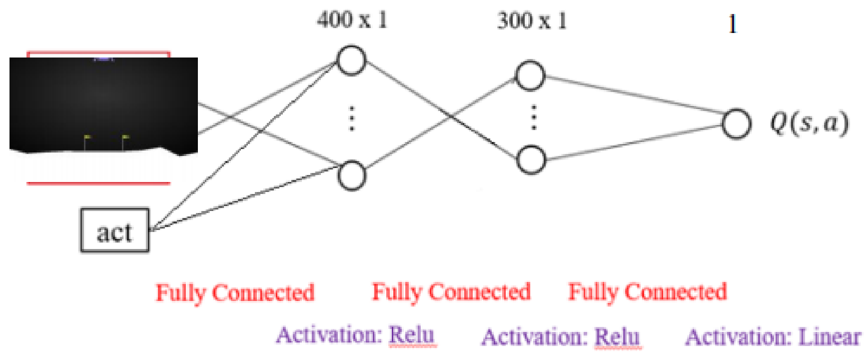
Implementation Details – LunarLanderContinuous-v2:

Network Architecture

- Actor



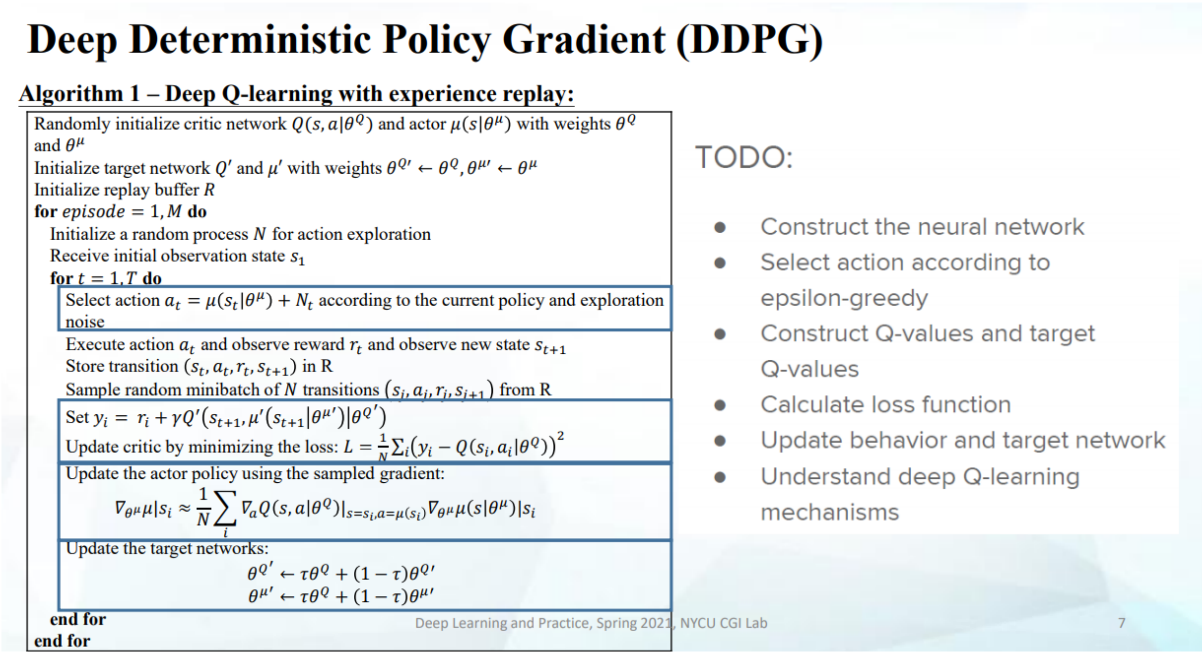
- Critic



根據PDF，與DQN相比網路部分多了一層hidden layer，並調整neural數量。且分為Actor Net和Critic Net(我們需要更動的是Actor Net)

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.firstLayer = nn.Linear(state_dim, 400)
        self.hiddenLayer = nn.Linear(400,300)
        self.fcLayer = nn.Linear(300, action_dim)
        self.activation = nn.ReLU()
        self.fcAct = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        x = self.firstLayer(x)
        x = self.activation(x)
        x = self.hiddenLayer(x)
        x = self.activation(x)
        x = self.fcLayer(x)
        x = self.fcAct(x)
        return x
```

(2) Select Action部分:



根據第一個藍框，由Actor Net來進行動作的選擇，並決定是否加上noise來促進探索的效果。

(3) Update behavior network

根據第二、三個藍框，首先從critic Net與current state, action計算出Q value後，我們需要分別從target action net和target

critic net取得next action和next q value，並利用next q value與reward和一些超參數計算出q target。
接著利用Q value與Q target計算出MSE Loss後進行critic net的back propagation和update。
action net的部分則首先由action net與current state決定一個action後，將這個state與action交由critic net進行評估，此評估值作為action net的Loss進行back propagation和update(Action Net)

```
# q_value = ?
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()
critic_loss = criterion(q_value, q_target)
# with torch.no_grad():
#     a_next = ?
#     q_next = ?
#     q_target = ?
# criterion = ?
# 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_loss = ?
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
#print(f"actor loss: {actor_loss}")
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

(4) Update Target Network部分:

每次進行update behavior network時將部分的值copy進target Network，避免target一直改動但又可以一點點對其進行更

新。

```python
@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)
```

(5) Testing部分:

實作上基本與DQN的相同。

3. Describe differences between your implementation and algorithms

我沒有在DQN與DDPG中實作與演算法不同的部分，但在加分題的DDQN中我將DDPG裡面update target network的做法搬進來使用，作為soft update的用途。

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

實作的部分於第2-3點已提過，因此這邊只說明gradient根據下列公式

Update the actor policy using the sampled gradient:

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

其中可以看到我們取gradient的目標應是中間的Q(s,a|cQ)。並且前面有1/N * Sum的標記說明我們使用的是mean的部分，因此code如下

```python
actor_loss = -critic_net(state, action).mean()
```

我們可以使用該loss來計算出gradient

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

Critic updating實作和gradient如何取用的部分已經於第2-3點提過了，因此這裡不再重打一次

6. Explain effects of the discount factor

discount factor的用途主要是用來決定未來的資訊對於現在的影響程度，若discount factor為1，則代表未來的所有資訊都跟現在一樣重要，若discount factor小於1，則每過一個iteration就將得到的資訊乘上一次discount factor後feedback回該時間點。

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

epsilon-greedy的用途主要在解決eploration and exploitation，若是一直依照Q值來進行動作的選擇，可能會因此而找不到更好的方案。因此epsilon-greedy的應用讓行為出現了一些隨機性，可以在各種狀況下進依照一定機率進行探索。

8. Explain the necessity of the target network.

不使用target network的話，behavior network和target network就相當於是同樣的東西，對network進行調整的話會同時影響到target和prediction，讓學習變得不穩定。使用target network的話會讓target在數個iteration才更新一次，預測可以慢慢逼近這個目標，使訓練過程相對穩定。

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

Replay buffer決定整個網路要保留多久以前的記憶，到達上限時從最老的開始刪除，因此若replay buffer size太大，則有可能保存到太老的記憶，這些記憶好一點可能沒有參考價值，慘一點甚至會影響到model的良率。而replay buffer size太小則可能導致存到的東西都是高度相關(同一場遊戲裡面)，不能學到一些比較沒有關聯性的經驗。