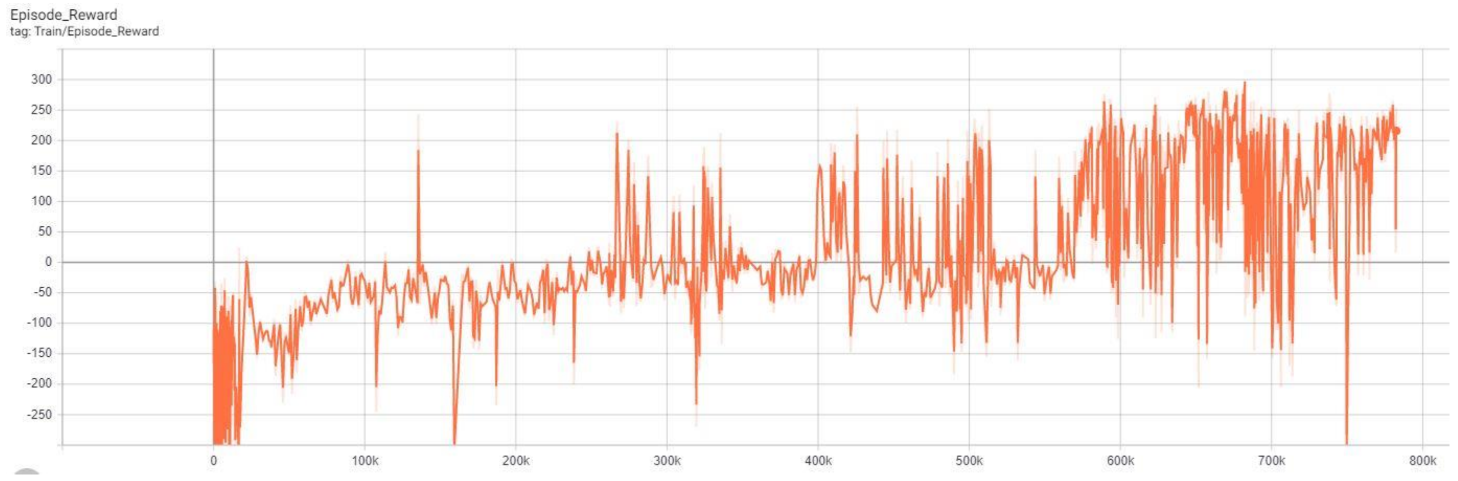
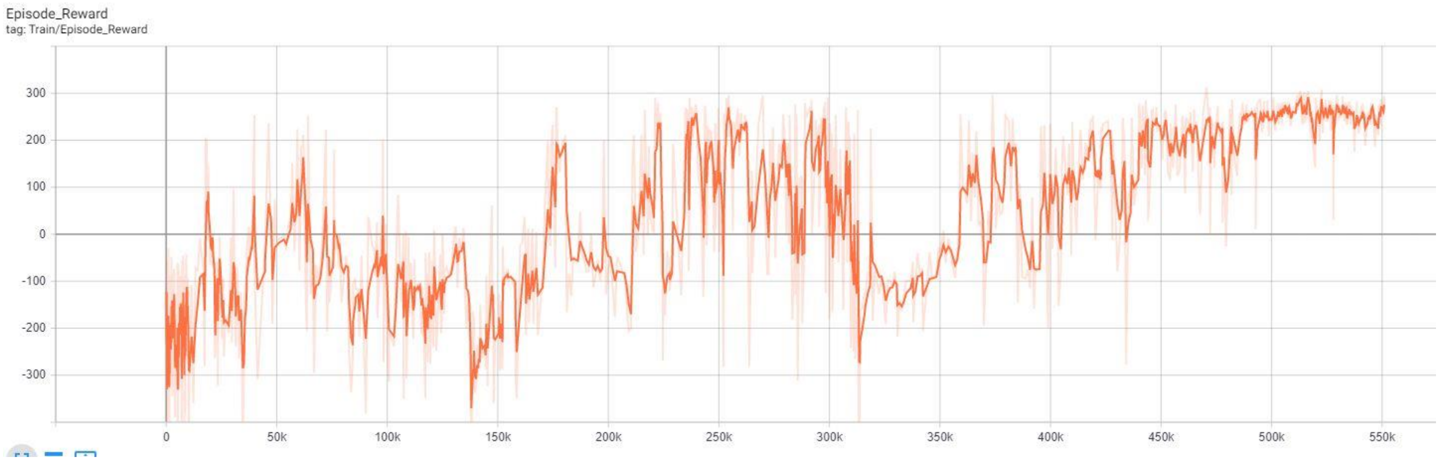
Deep Learning and Practice

#Lab06 DQN-DDPG **309505002 鄭紹文**

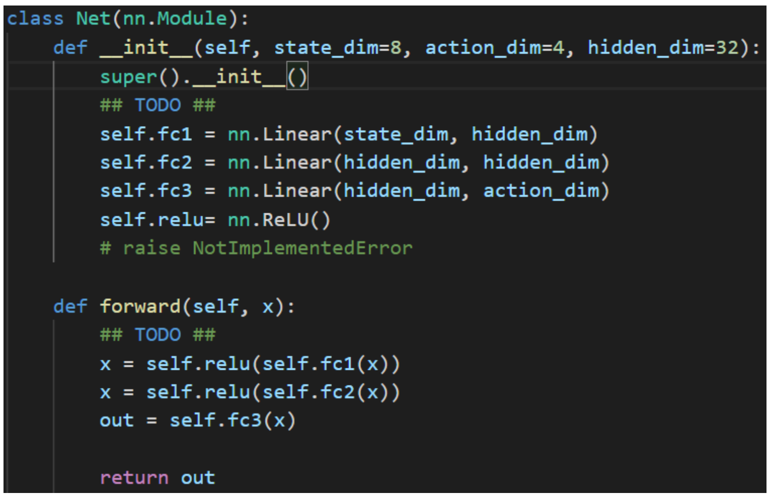
- **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2**



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

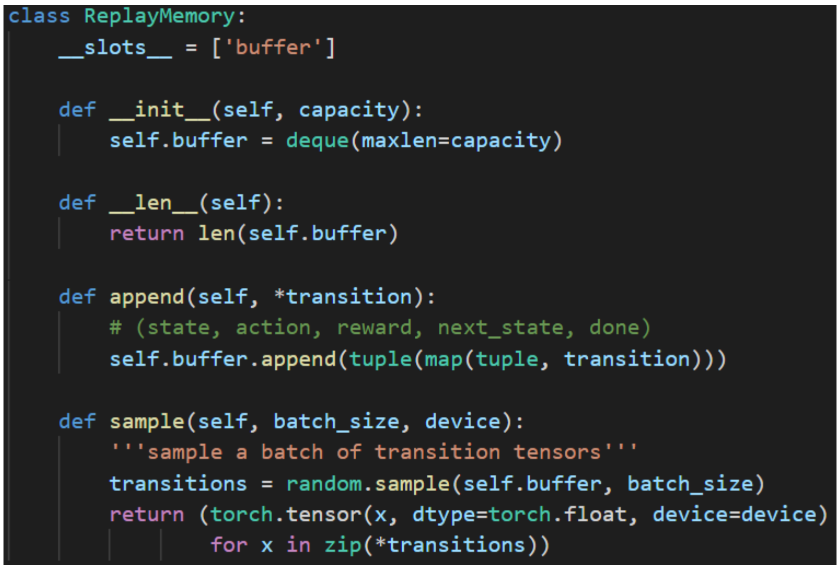


- **Describe your major implementation of both algorithms in detail.**
  - **For DQN：**



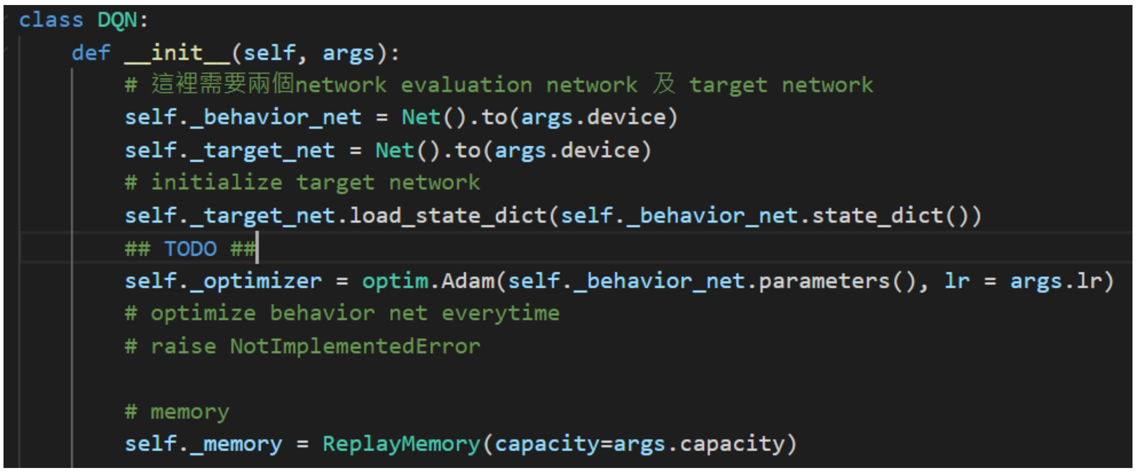
```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)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

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

        return out
```

Net 架構部分根據助教所給的 Network Architecture 所建構，三層 fully connected layer，搭配 ReLU layer，輸入 input 後，在得到相對應的 action 作為 output。



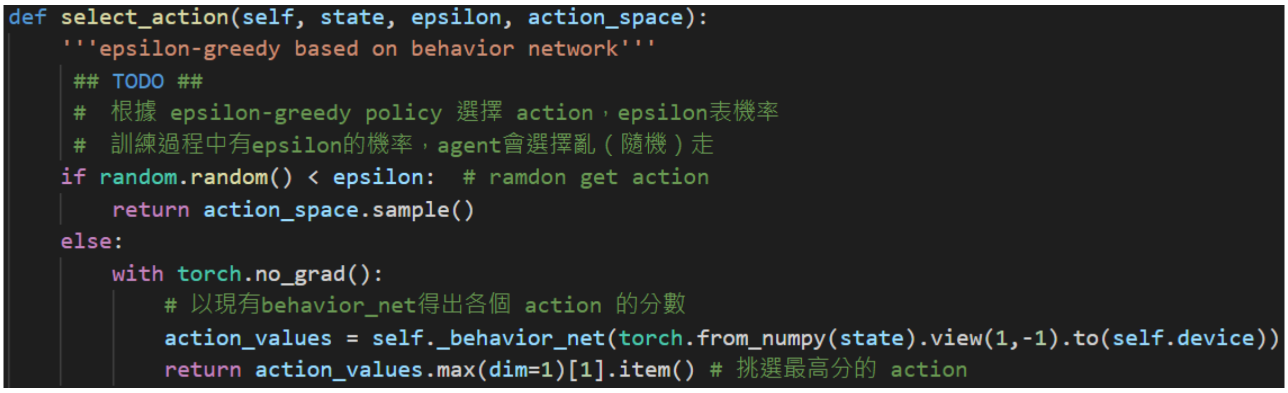
```python
class ReplayMemory:
    __slots__ = ['buffer']

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

Replay memory 用來儲存要進行很多次遊戲的 memory，才得以進行 experience replay(需要有 memory pool)。



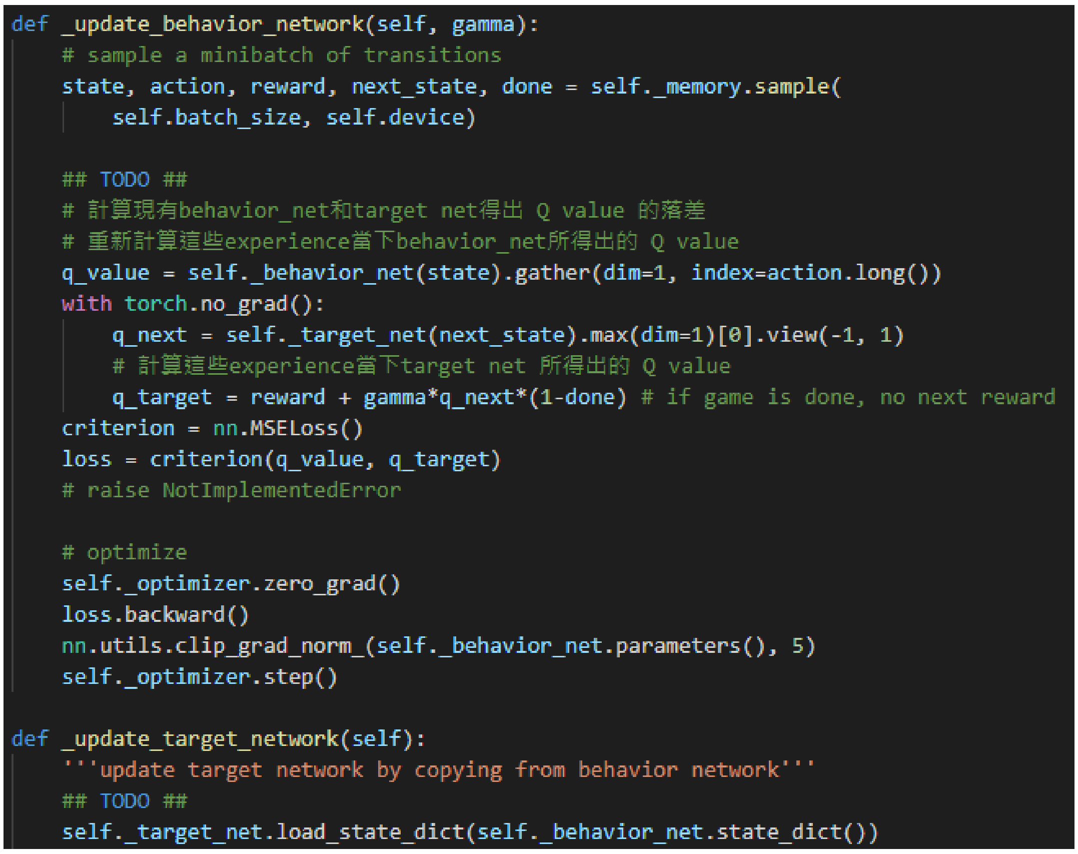
```python
class DQN:
    def __init__(self, args):
        # 這裡需要兩個network evaluation network 及 target network
        self._behavior_net = Net().to(args.device)
        self._target_net = Net().to(args.device)
        # initialize target network
        self._target_net.load_state_dict(self._behavior_net.state_dict())
        ## TODO ##
        self._optimizer = optim.Adam(self._behavior_net.parameters(), lr = args.lr)
        # optimize behavior net everytime
        # raise NotImplementedError

        # memory
        self._memory = ReplayMemory(capacity=args.capacity)
```

根據 spec 所提供的 training Hyper-Parameters：optimizer：Adam，learning rate：0.0005



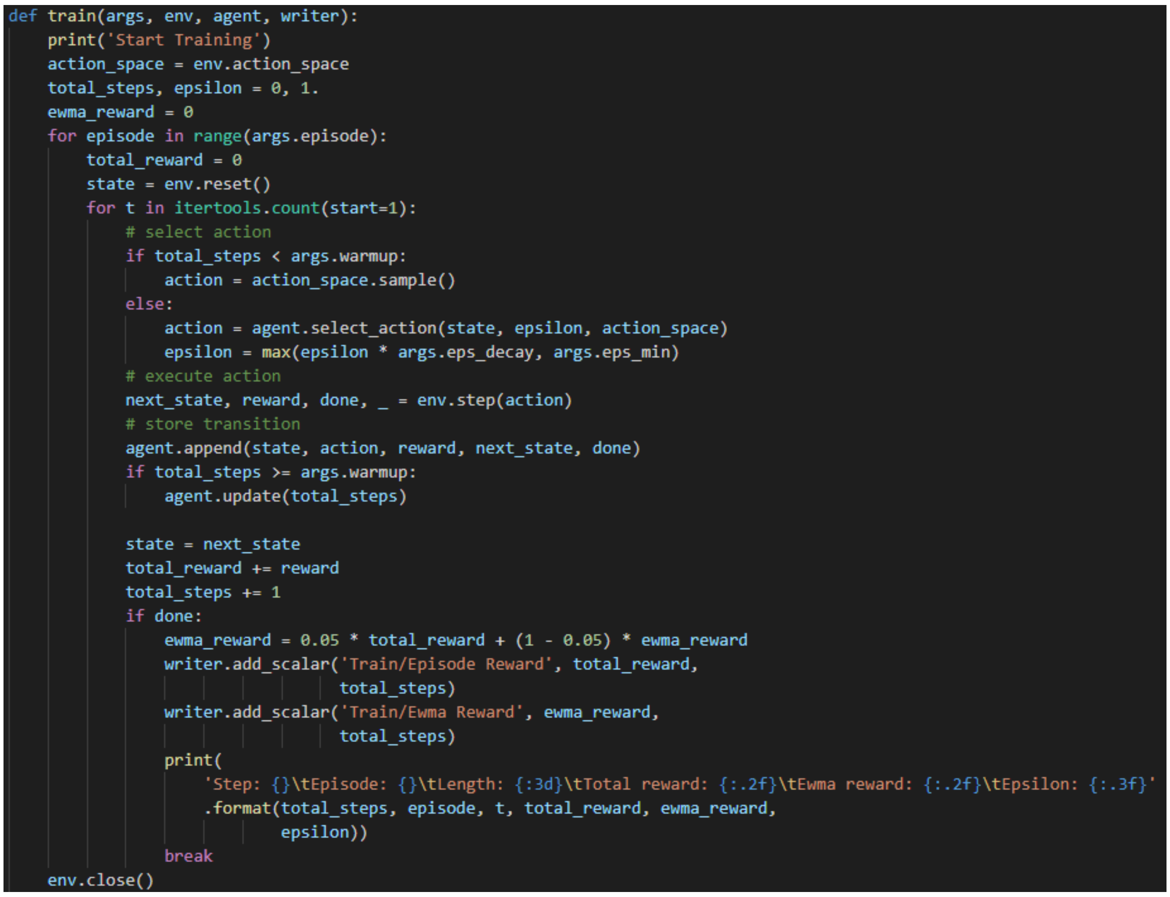
```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    #  根據 epsilon-greedy policy 選擇 action，epsilon表機率
    #  訓練過程中有epsilon的機率，agent會選擇亂（隨機）走
    if random.random() < epsilon:  # ramdon get action
        return action_space.sample()
    else:
        with torch.no_grad():
            # 以現有behavior_net得出各個 action 的分數
            action_values = self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device))
            return action_values.max(dim=1)[1].item() # 挑選最高分的 action
```

training 過程中，選擇動作的策略是使用 epsilon-greedy action，除了會選擇最大 reward 的動作也有機會選到不是的動作，這樣也許會透過嘗試其他可能性發現新的有更大 reward 的動作。



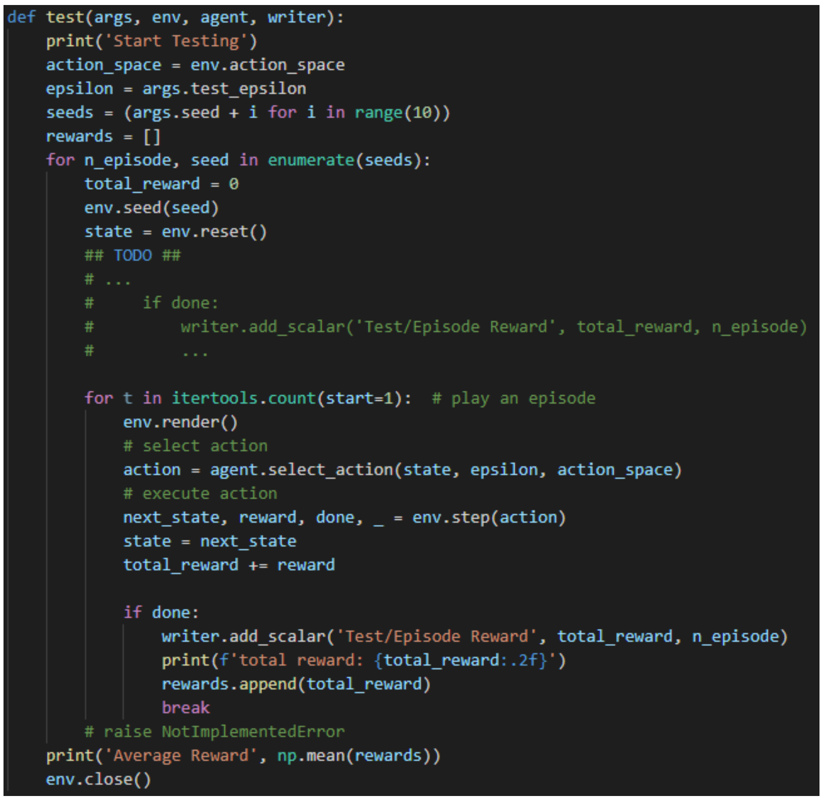
```python
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    # 計算現有behavior_net和target net得出 Q value 的落差
    # 重新計算這些experience當下behavior_net所得出的 Q value
    q_value = self._behavior_net(state).gather(dim=1, index=action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state).max(dim=1)[0].view(-1, 1)
        # 計算這些experience當下target net 所得出的 Q value
        q_target = reward + gamma*q_next*(1-done) # if game is done, no next reward
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
    # raise NotImplementedError

    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()

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())
```

行為網路會每次都更新參數但是目標網路並不會這麼做，這樣拆成兩個網路的好處是在一段時間裡使目標Q值保持不變，能在一定程度上降低當前Q值和目標Q值的相關性，提高算法的穩定性，並在一定次數後再將行為網路的參數更新到目標網路中。



```python
def train(args, env, agent, writer):
    print('Start Training')
    action_space = env.action_space
    total_steps, epsilon = 0, 1.
    ewma_reward = 0
    for episode in range(args.episode):
        total_reward = 0
        state = env.reset()
        for t in itertools.count(start=1):
            # 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)

            state = next_state
            total_reward += reward
            total_steps += 1
            if done:
                ewma_reward = 0.05 * total_reward + (1 - 0.05) * ewma_reward
                writer.add_scalar('Train/Episode Reward', total_reward,
                                  total_steps)
                writer.add_scalar('Train/Ewma Reward', ewma_reward,
                                  total_steps)
                print(
                    'Step: {}\tEpisode: {}\tLength: {:3d}\tTotal reward: {:.2f}\tEwma reward: {:.2f}\tEpsilon: {:.3f}'
                    .format(total_steps, episode, t, total_reward, ewma_reward,
                            epsilon))
                break
    env.close()
```

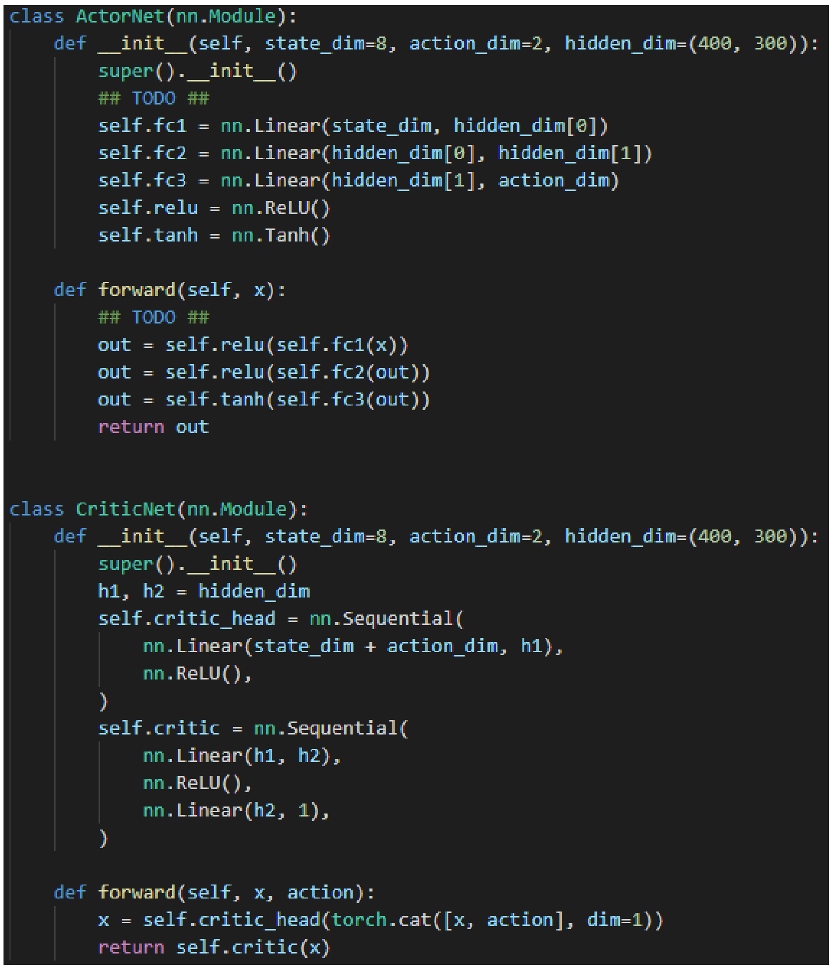
Training時分成兩步驟，先蒐集遊戲data memory放進先前提到的memory內，再根據這些data來update net，一直到遊戲結束就進入下一個epoch。



```python
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 ##
        # ...
        #     if done:
        #         writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
        #         ...

        for t in itertools.count(start=1):  # play an episode
            env.render()
            # select action
            action = agent.select_action(state, epsilon, action_space)
            # execute action
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                print(f'total reward: {total_reward:.2f}')
                rewards.append(total_reward)
                break
        # raise NotImplementedError
    print('Average Reward', np.mean(rewards))
    env.close()
```

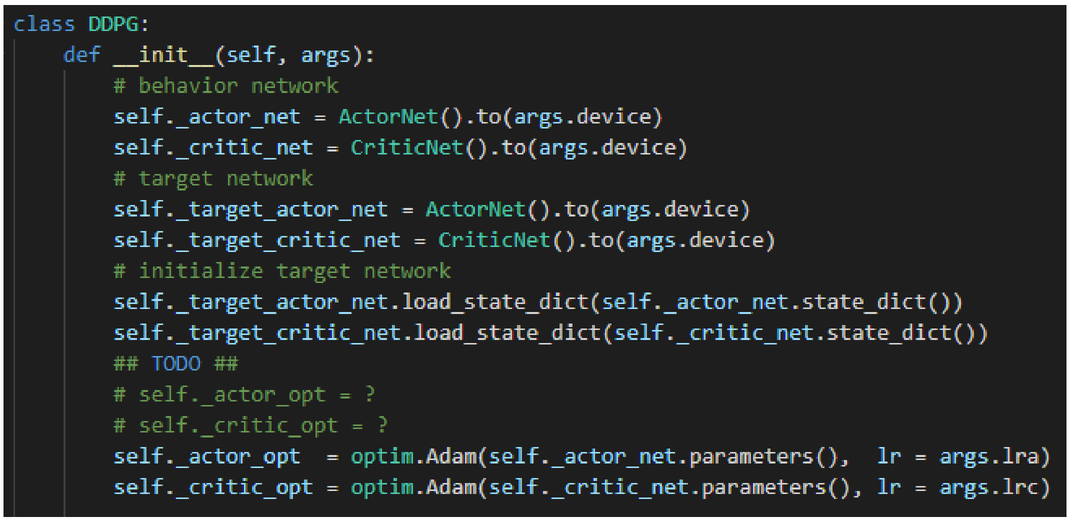
Testing時，根據select_action function來決定下一個action。

- **For DDPG：**



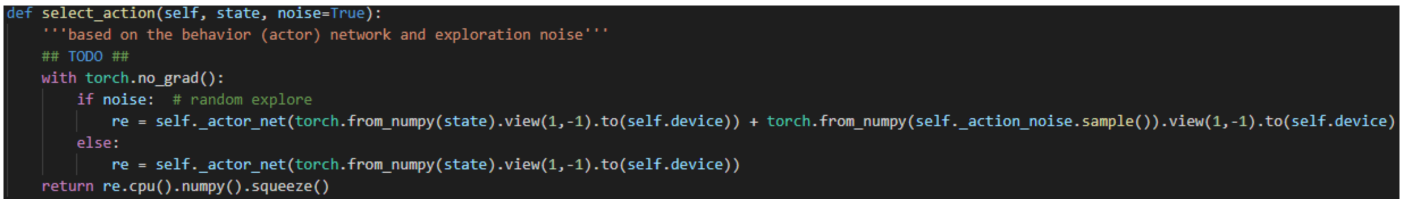
```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim, hidden_dim[0])
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.fc3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

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


class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, 1),
        )

    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)
```

主要由 actor 和 critic 兩個部分組成，分別有對應的網路和目標，兩者分別在最後輸出使用的激勵函數不一樣。而 Actor 只負責輸出動作，而判斷動作的好壞由 critic 來判斷(輸出 Q value)，目標的更新依據 critic 對下一個狀態和動作計算出的 Q value



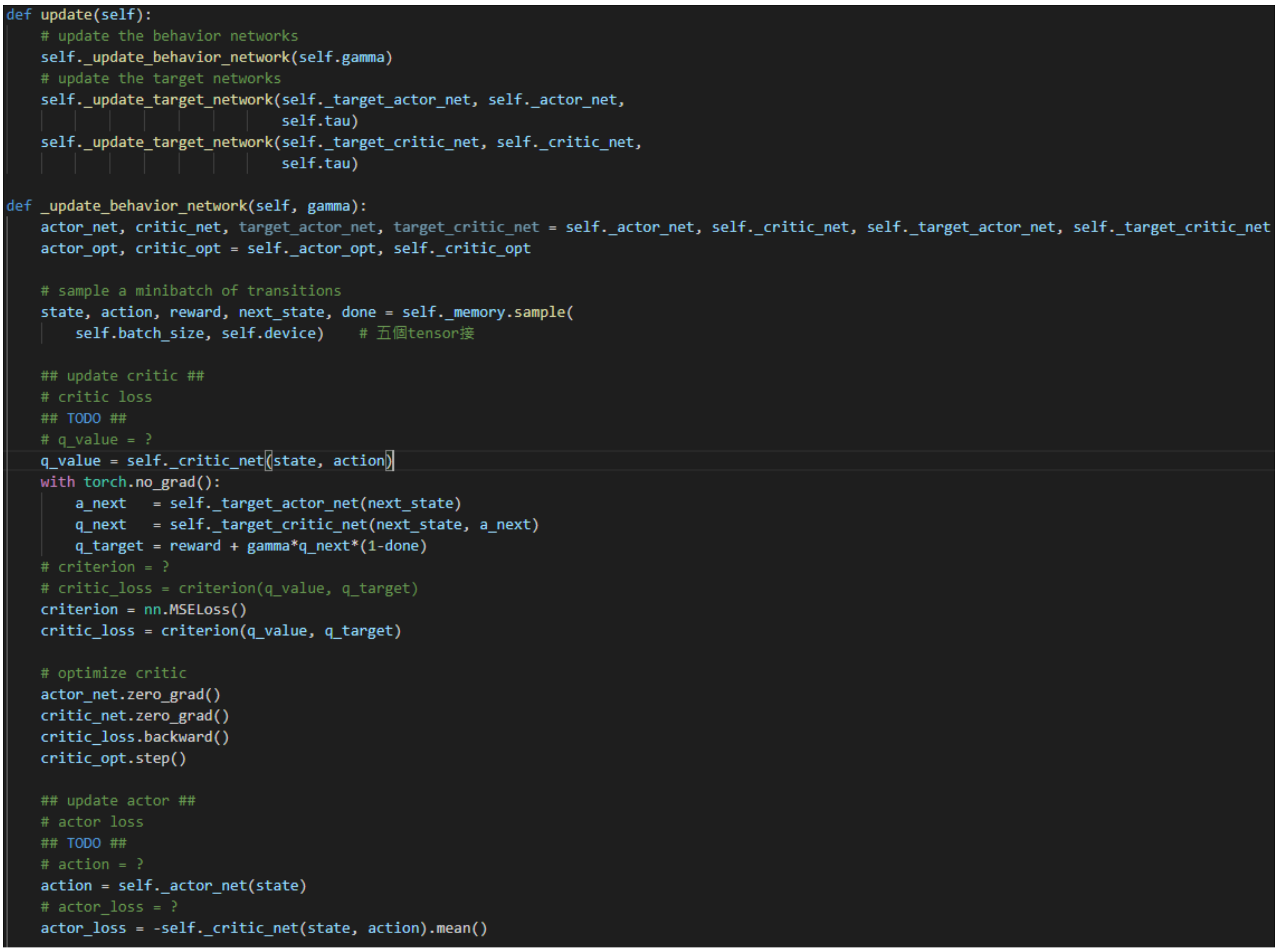
```
class DDPG:
    def __init__(self, args):
        # behavior network
        self._actor_net = ActorNet().to(args.device)
        self._critic_net = CriticNet().to(args.device)
        # target network
        self._target_actor_net = ActorNet().to(args.device)
        self._target_critic_net = CriticNet().to(args.device)
        # 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 = ?
        # self._critic_opt = ?
        self._actor_opt  = optim.Adam(self._actor_net.parameters(),  lr = args.lra)
        self._critic_opt = optim.Adam(self._critic_net.parameters(), lr = args.lrc)
```

根據 spec 所提供的 training Hyper-Parameters：optimizer：Adam，learning rate：0.003

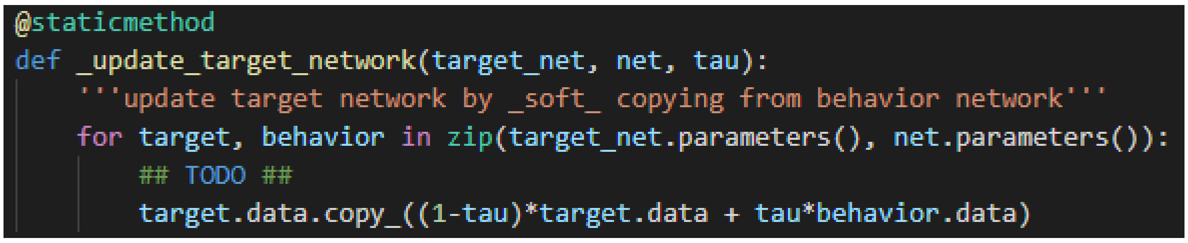


```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:  # random explore
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device)) + torch.from_numpy(self._action_noise.sample()).view(1,-1).to(self.device)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

選擇動作上加入noise使其可以探索不同環境、其他沒有嘗試過的動作以找到最佳的reward



```python
def update(self):
    # update the behavior networks
    self._update_behavior_network(self.gamma)
    # update the target networks
    self._update_target_network(self._target_actor_net, self._actor_net,
                                self.tau)
    self._update_target_network(self._target_critic_net, self._critic_net,
                                self.tau)

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)    # 五個tensor接

    ## update critic ##
    # critic loss
    ## TODO ##
    # q_value = ?
    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)
    # criterion = ?
    # critic_loss = criterion(q_value, q_target)
    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 = ?
    action = self._actor_net(state)
    # actor_loss = ?
    actor_loss = -self._critic_net(state, action).mean()
```



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

先通過 minimum loss 來更新 online target network，再透過 PG 來更新 online actor network，最後使用 soft update 每次更新參數一點點(更新 target network)。

$$\boldsymbol{\theta^{Q'} = \Gamma\theta^{Q} + (1 - \Gamma)\theta^{Q'}}$$

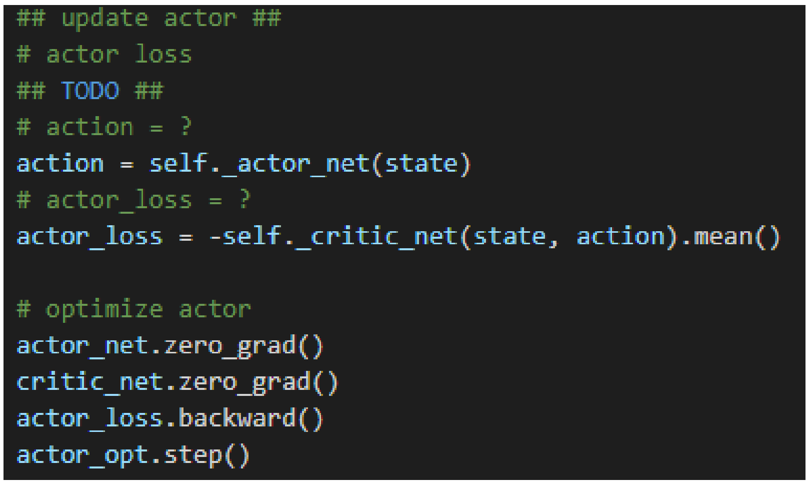
$$\boldsymbol{\theta^{\mu'} = \Gamma\theta^{\mu} + (1 - \Gamma)\theta^{\mu'}}$$

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

  在一開始會先進行初始化，warm up時間的動作等進行了一定次數的遊戲後再開始進行更新參數，和在原本算法中是同步更新行為網路和目標網路，實作中設定了一定次數才更新。

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

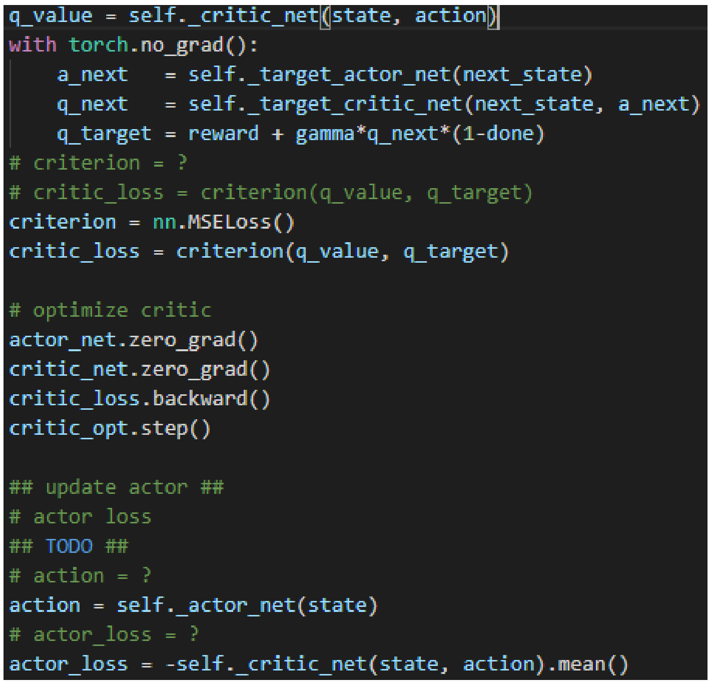
  為了使actor net所獲得的reward越多越好，先將state放入actor net中產生動作再將產生的動作和state放入critic中產生actor的loss取平均(因為要maximize，故此處加上負號)，再利用back propagation，計算後更新參數。



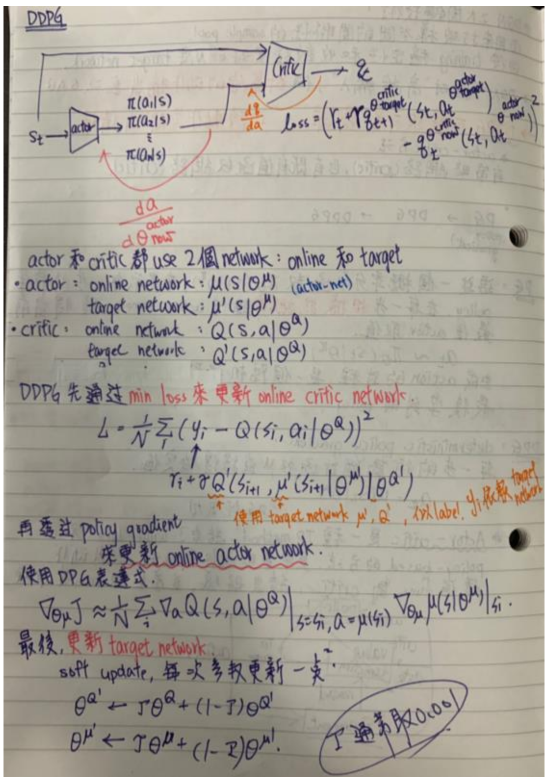
```
## update actor ##
# actor loss
## TODO ##
# action = ?
action = self._actor_net(state)
# actor_loss = ?
actor_loss = -self._critic_net(state, action).mean()

# 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**

  使用actor的目標網路計算下一個狀態的動作a_next，再用critic的目標網路算出q_next，乘上gamma算出q_target，計算loss的方式就是將q_value和target做MSE_loss。



```
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)
# criterion = ?
# critic_loss = criterion(q_value, q_target)
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 = ?
action = self._actor_net(state)
# actor_loss = ?
actor_loss = -self._critic_net(state, action).mean()
```



S_t
actor
$\pi(a_1|s)$
$\pi(a_2|s)$
$\pi(a_N|s)$
critic
q_t
$\frac{dq}{da}$
$\frac{da}{d\theta^{actor}_{now}}$
$loss = \left(r_t + \gamma q_{t+1}^{\theta^{critic}_{target}}(s_t, a_t^{\theta^{actor}_{target}}) - q_t^{\theta^{critic}_{now}}(s_t, a_t^{\theta^{actor}_{now}})\right)^2$

DDPG

actor 和 critic 都 use 2個 network：online 和 target

- actor： online network：$\mu(s|\theta^{\mu})$ (actor-net)
  target network：$\mu'(s|\theta^{\mu})$
- critic： online network：$Q(s,a|\theta^{Q})$
  target network：$Q'(s,a|\theta^{Q})$

DDPG 先通过 min loss 來更新 online critic network.

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

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

使用 target network μ', Q'，作 label. y_i 依賴 target network

再透过 policy gradient 來更新 online actor network.

使用 DPG 表達式.

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

最後，更新 target network.

soft update，每次參數更新一点.

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

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

τ 通常取 0.001

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

$$G_t = R_{t+1} + \lambda R_{t+2} + \ldots = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

從公式中可以看出，Gt是未來每一個時間點的reward相加，λ(通常介於0到1之間)稱之為discount factor，由於越遙遠的未來相對於現在而言是較不確定的，所以reward的影響是越小，而當下的reward是影響最大的。

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

epsilon-greedy結合了exploration以及exploitation，在訓練初期epsilon-greedy是比較好的選擇，因為在一開始model沒有任何經驗，所以model需要能去有彈性的、大量的嘗試不同的動作所帶來的影響，能去有彈性的嘗試其他可能，我們會設一個參數ε，小於ε時會隨機嘗試下一個action，而大於ε則是選取behavior net的output，其為目前已知reward最大的action，而漸漸訓練到後期之後我們就可以從之前嘗試的動作選取較佳的動作來得到比較好的reward，這時候隨機探索所帶來的效益就會比較差。

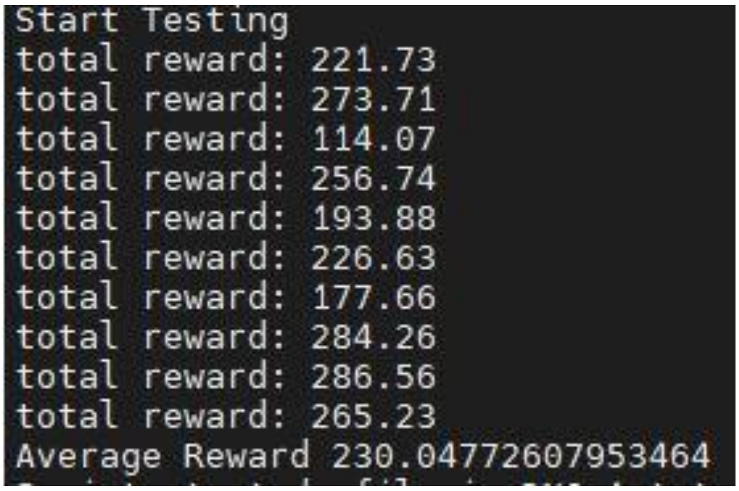
- **Explain the necessity of the target network.**

在實作中進行網路的訓練時，若只用一個網路，在每次訓練完後更新會導致目標也跟著變動，選擇action以及計算action reward的net相同，會導致因為目標計算與當前有超度相關，而結果無法收斂，所以在Q learnimg中為了要降低當前Q值和目標Q值的相關性，會拆成兩個網路來訓練，以提高穩定性和收斂性。

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

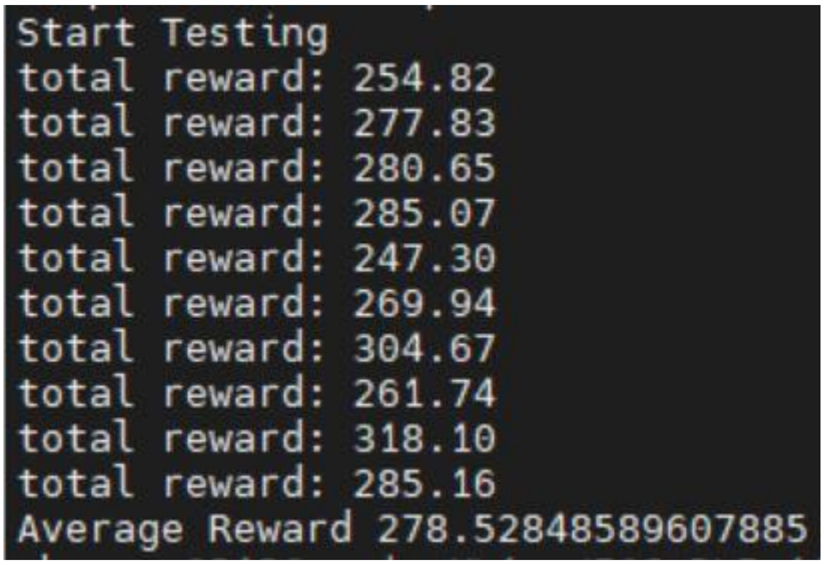
在訓練的過程中獲得的經驗可以不斷重複使用，抽取定量數據採隨機抽樣更新幫助網路訓練得更好，如果將size設定的太小變成過去經驗太少沒有足夠的資料能夠參考幫助訓練，但是如果設定成太大可能會導致陷入局部的最佳解而找不到更好的解，分數無法進步，同時也會增加training時間。

- **Performance：**
  - [LunarLander-v2] Average reward of 10 testing episodes：



```
Start Testing
total reward: 221.73
total reward: 273.71
total reward: 114.07
total reward: 256.74
total reward: 193.88
total reward: 226.63
total reward: 177.66
total reward: 284.26
total reward: 286.56
total reward: 265.23
Average Reward 230.04772607953464
```

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



```
Start Testing
total reward: 254.82
total reward: 277.83
total reward: 280.65
total reward: 285.07
total reward: 247.30
total reward: 269.94
total reward: 304.67
total reward: 261.74
total reward: 318.10
total reward: 285.16
Average Reward 278.52848589607885
```