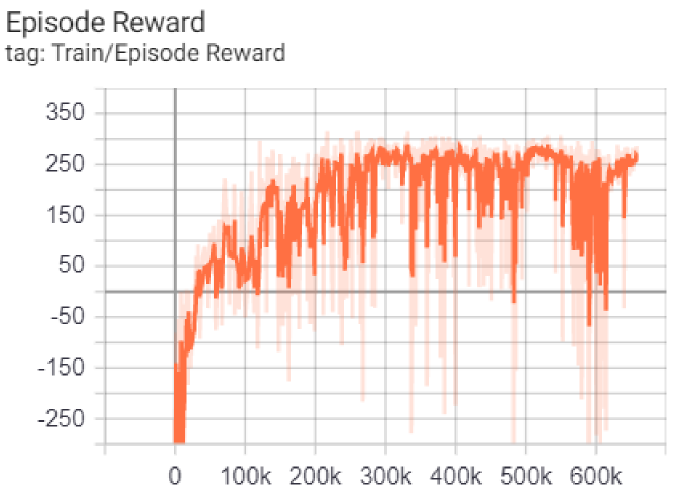
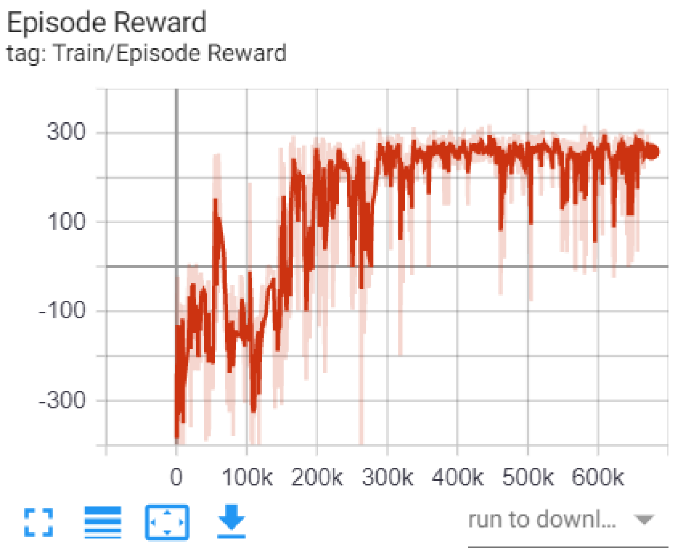
Lab 6 Deep Q-Network and Deep Deterministic Policy Gradient

309553005 賴佑家

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



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



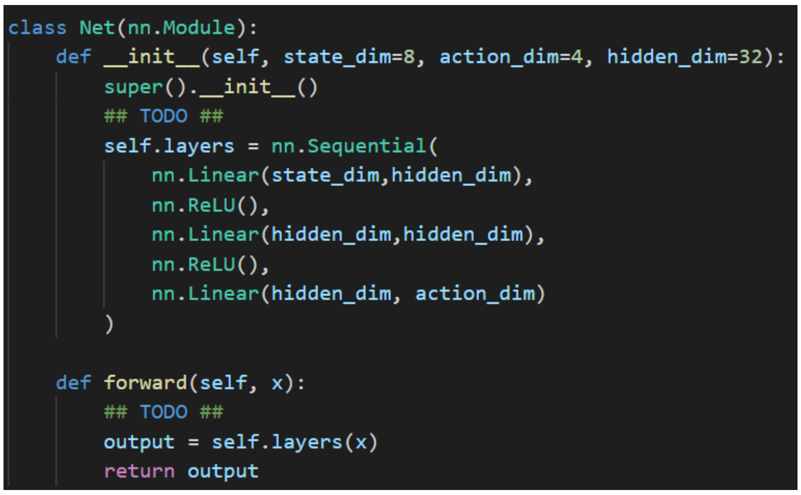
3. Describe your major implementation of both algorithms in detail

- **DQN**

a. Network architecture

DQN是利用一個neural network來預測Q value Q(s,a)，而在LunarLander-v2中總共有四個action，所以在設計神經網路時經過兩層fully connected layer後，最後一層fully connected要output 4個action的value

(0 (No-op), 1 (Fire left engine), 2 (Fire main engine), 3 (Fire right engine))

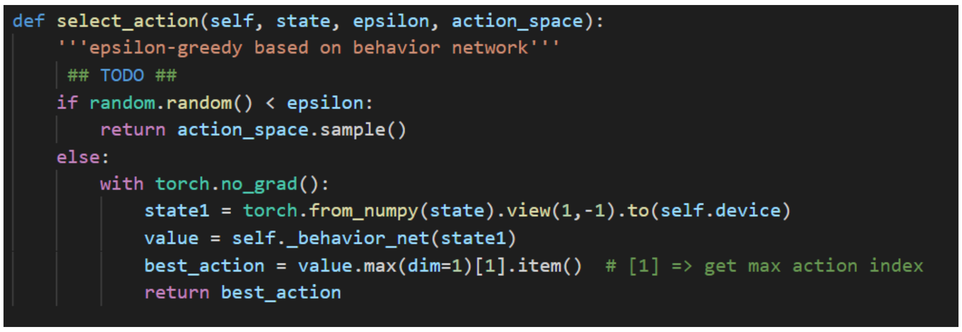


```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.layers = nn.Sequential(
            nn.Linear(state_dim,hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim,hidden_dim),
     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

    def forward(self, x):
        ## TODO ##
        output = self.layers(x)
        return output
```

b. Select action

那有了network產生的action value function後，我們便可以選擇當前擁有最大值的value選擇為最佳的action，而因為epsilon-greedy的關係，有一定的機率會隨機選擇一個action



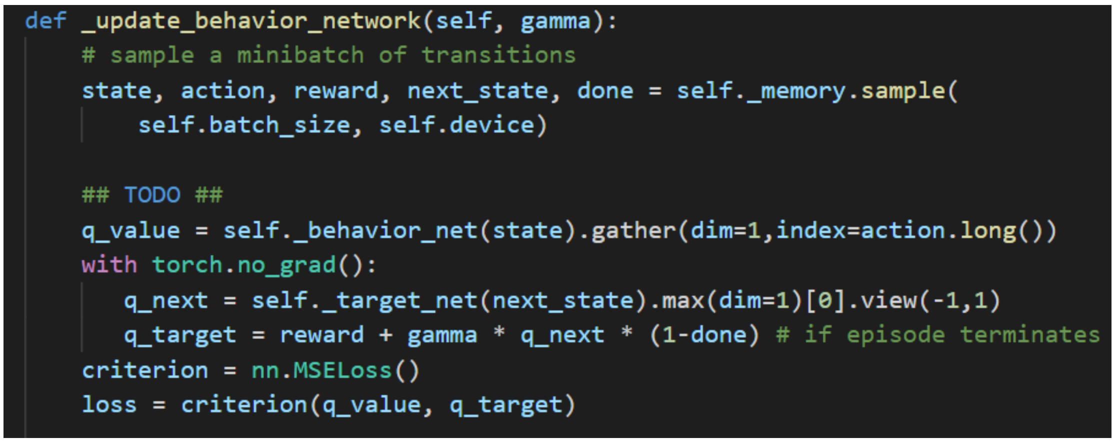
```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            state1 = torch.from_numpy(state).view(1,-1).to(self.device)
            value = self._behavior_net(state1)
            best_action = value.max(dim=1)[1].item()  # [1] => get max action index
            return best_action
```

c. Update network

■ behavior network

首先從replay buffer sample資料出來，然後我們要算我們的Q target value y，那我們就先把next state放入target network算出Q’，再選出Q’的最大值乘上discount factor加上reward就得到我們的y，而如果這個step即將要結束這個episode的話，y就會直接等於reward

那有了target value y後，我們就可以把我的target value與我們behavior network算出的Q value算loss，之後經過back-propagation就可以更新我們的網路了



```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 ##
    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)
        q_target = reward + gamma * q_next * (1-done) # if episode terminates
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

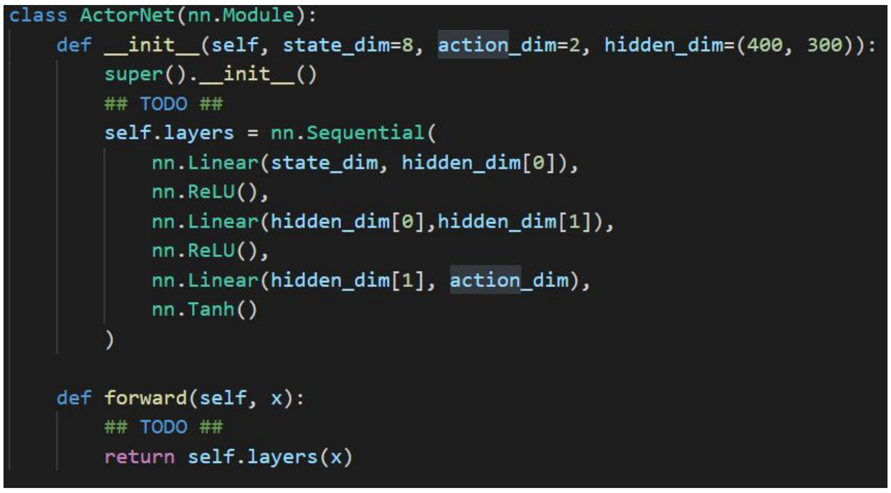
$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the

- target network
  經過一段時間後，把behavior network的參數複製給他

- **DDPG**

  a. Actor network
  建立一個根據目前state就可以知道要做哪個action的neural network，網路的架構為經過兩層fully connected layers後，最後一層fully connected後output兩個action的值（Main engine: -1 ~ 1, Left-right engine -1 ~ 1)

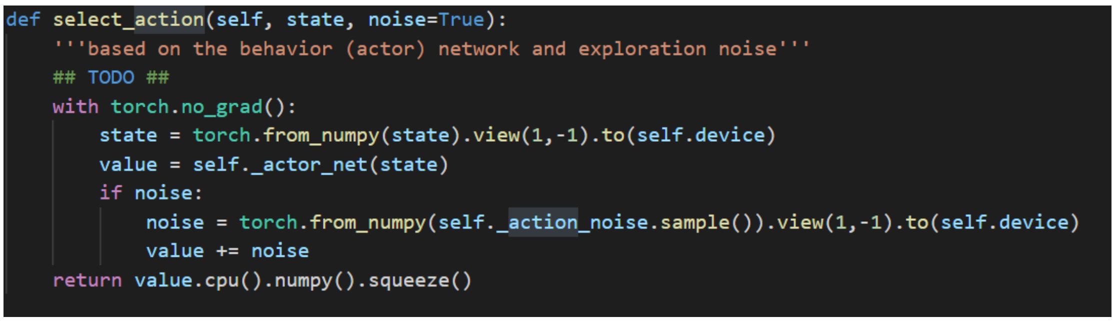


```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.layers = nn.Sequential(
            nn.Linear(state_dim, hidden_dim[0]),
            nn.ReLU(),
            nn.Linear(hidden_dim[0],hidden_dim[1]),
            nn.ReLU(),
            nn.Linear(hidden_dim[1], action_dim),
            nn.Tanh()
        )

    def forward(self, x):
        ## TODO ##
        return self.layers(x)
```

b. Select action

直接將state actor後就可以得到我們的action，而參數noise決定是否要加入exploration noise



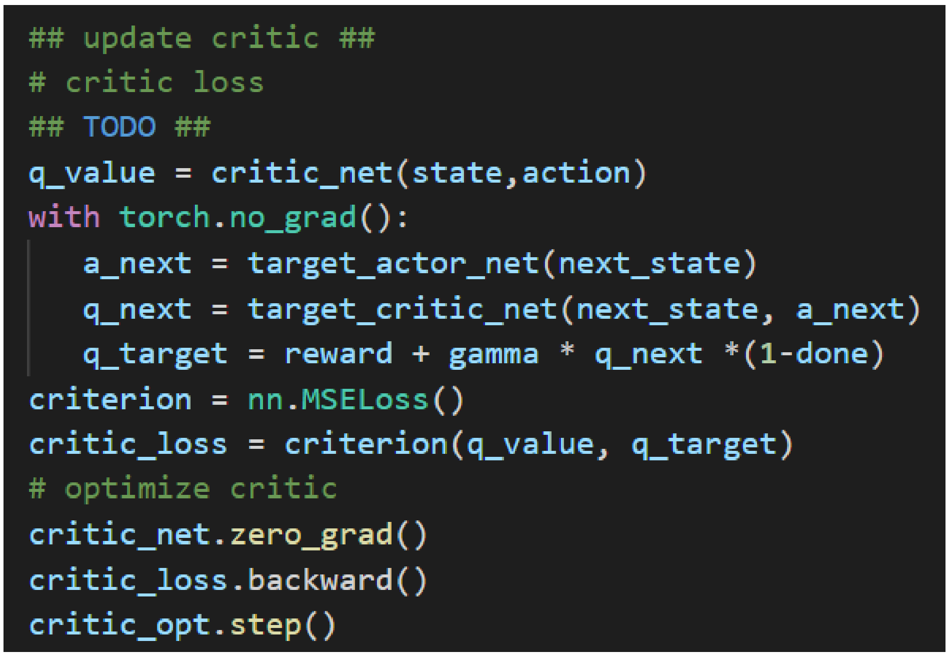
```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        state = torch.from_numpy(state).view(1,-1).to(self.device)
        value = self._actor_net(state)
        if noise:
           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

c. Update network

更新critic與actor兩個網路，更新的細節在第五點與第六點有說明，在這邊大致說明一下

- Critic

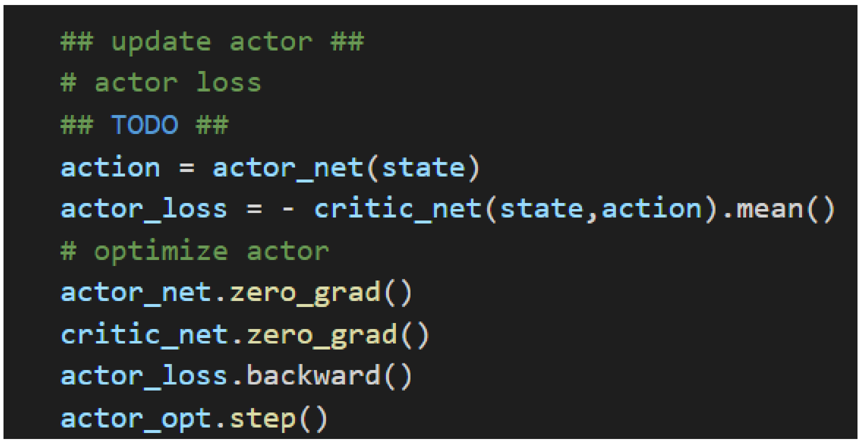
主要就是將critic predict的value與target value算loss然後做back-propagation



```python
## update critic ##
# critic loss
## TODO ##
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)
# optimize critic
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

- Actor

計算基於actor的action經過critic算出預測的value function後取平均，之後再經過back-propagation算出gradient來更新網路

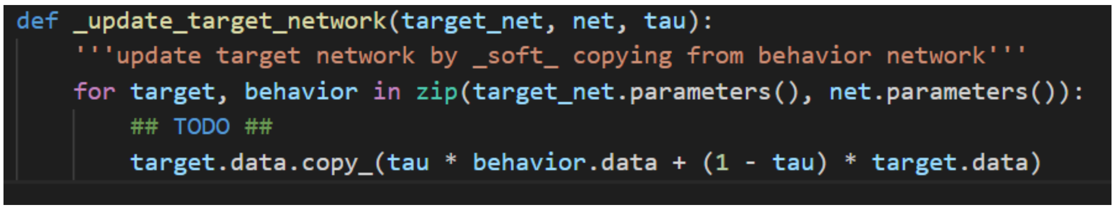


```python
## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = - critic_net(state,action).mean()
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

- Target network
  使用一個parameter tau來控制要從behavior network拿多少比例的參數來更新target network

  Update the target networks:

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



```python
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 - tau) * target.data)
```

4. Describe differences between your implementation and algorithms
   - Discrete and continuous
   DQN主要是處理discrete的environment，而DDPG是處理continuous的environment
   - Architecture
   DQN state經過network後便會產生action value Q，而DDPG會有兩個network actor與critic，state經過actor產生action後，再將state與actor產生出來的action放入critic後，才會產生action value Q
   - Select action
   DQN在選擇action時，會將所有的action value function算出來，取最大值的action當作最佳的action。而DDPG的environment是continuous，所以它是通過actor後，即產生action的值(主引擎與左右引擎要給予多少能量)

5. Describe your implementation and the gradient of actor updating
   首先用state經過actor產生action u，再把state與剛算出來的action放入critic產

生Q value，然後再取平均，之後就是在back-propagation去計算gradient，然後再用去更新actor的參數

Update the actor policy using the sampled gradient:

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

6. Describe your implementation and the gradient of critic updating
先從replay buffer sample一個minibatch出來，之後再把這個batch的action放進critic算出對應的Q value，而loss的計算必須要有Q value以及target value，有了Q value後，我們還需要target value

那target value就是下圖的y值，他的算法就是，先將next state s_{t+1}放入target actor network產生next state的action u'，之後再將s_{t+1}與u'放入target critic network，產生next state的Q value Q'
那再套用下面的公式，利用我們算出的Q'乘上discount factor加上reward就得到我們的y了

那有了Q value以及target我們就可以拿這兩個值進行MSE的計算，之後再透過back-propagation就可以更新我們的critic參數了

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

7. Explain effects of the discount factor
底下為Q learning的演算法，裡面的gamma就是discount factor，當discount factor越小時，agent就會越注重目前可獲得的reward，當discount factor越大時，agent就會比較重視未來獲得的長期獎勵

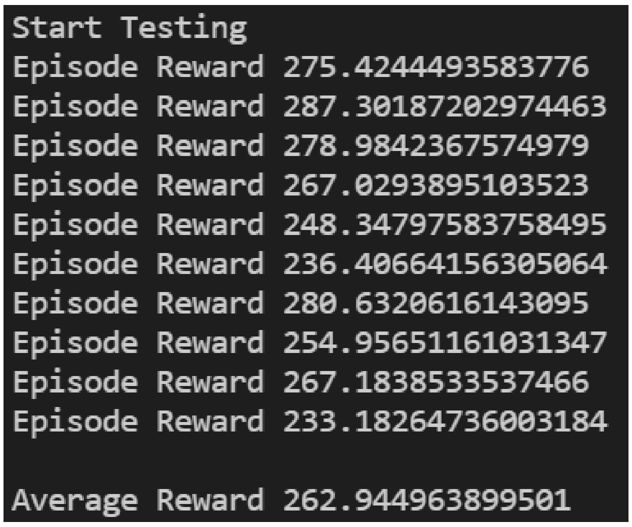
$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

8. Explain benefits of epsilon-greedy in comparison to greedy action selection
如果每次都選擇當前最好的action，其最後的結果不一定是最好的，所以epsilon greedy就設一個機率epsilon，有一定的機率會選擇隨機的action

9. Explain the necessity of the target network
如果沒有target network，就會同時計算target Q跟預測的Q，然後兩者同時更新，這樣就會造成自己在跟自己的target Q進行比較，如果多了target network，隔了一段時間會將訓練的參數複製給target network，就可以解決這個問題

10. Explain the effect of replay buffer size in case of too large or too small
如果replay buffer size大的話，那它涵蓋的sample數就會比較大，這樣會讓training比較穩定，但這樣會導致訓練速度比較慢以及記憶體消耗的問題
如果replay buffer size太小的話，這樣就只會著重在最近玩得episode狀況，導致performance不好

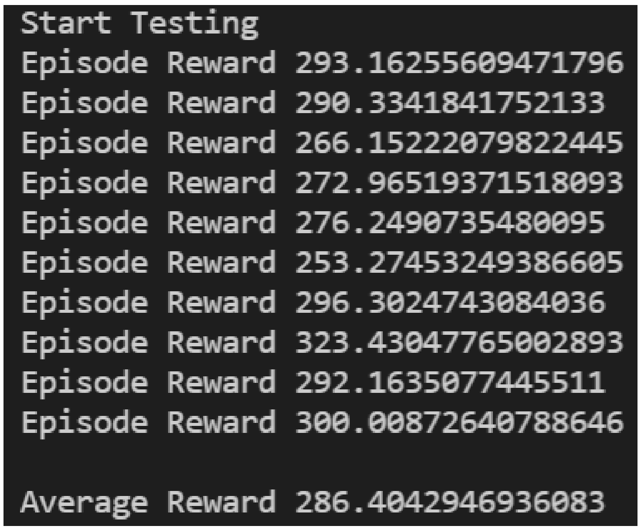
11. Performance
- DQN average reward 262.94496



```
Start Testing
Episode Reward 275.4244493583776
Episode Reward 287.30187202974463
Episode Reward 278.9842367574979
Episode Reward 267.0293895103523
Episode Reward 248.34797583758495
Episode Reward 236.40664156305064
Episode Reward 280.6320616143095
Episode Reward 254.95651161031347
Episode Reward 267.1838533537466
Episode Reward 233.18264736003184

Average Reward 262.944963899501
```

- DDPG average reward 286.40429



```
Start Testing
Episode Reward 293.16255609471796
Episode Reward 290.3341841752133
Episode Reward 266.15222079822445
Episode Reward 272.96519371518093
Episode Reward 276.2490735480095
Episode Reward 253.27453249386605
Episode Reward 296.3024743084036
Episode Reward 323.43047765002893
Episode Reward 292.1635077445511
Episode Reward 300.00872640788646

Average Reward 286.4042946936083
```