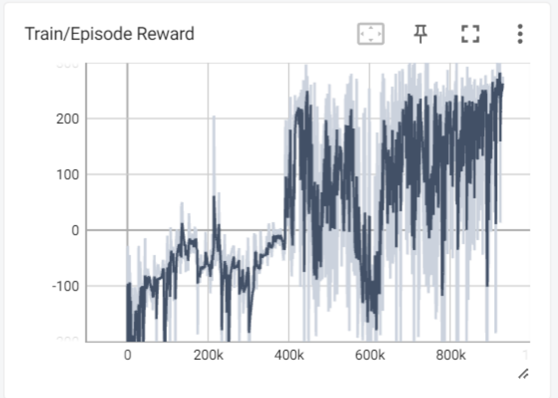
DL_LAB5_Deep Q-Network and Deep Deterministic Policy Gradient

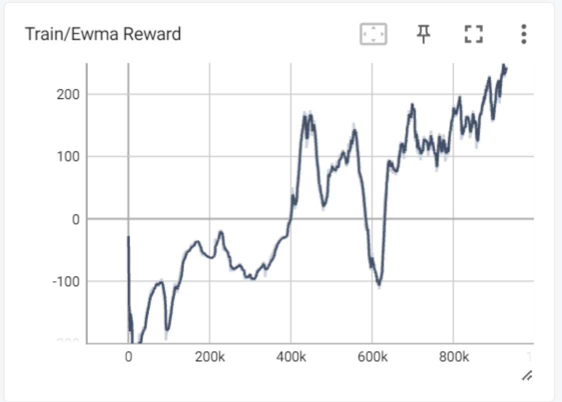
學號：312554004　姓名：林垣志

(1) Results of experiment:

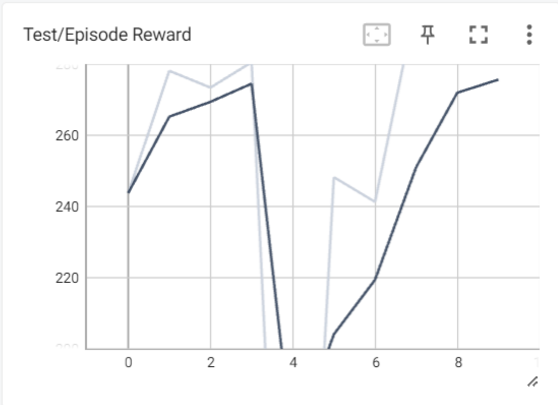
(a) LunarLander-v2 using DQN

Train





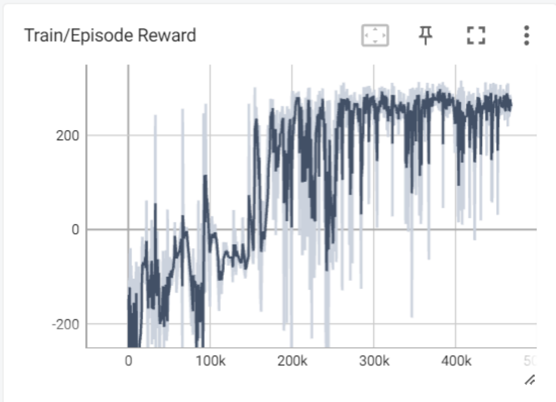
Test




```
Start Testing
Episode: 0      \Length: 205    Total Reward: 243.76
Episode: 1      \Length: 533    Total Reward: 278.10
Episode: 2      \Length: 247    Total Reward: 273.40
Episode: 3      \Length: 199    Total Reward: 280.17
Episode: 4      \Length:  93    Total Reward: 38.56
Episode: 5      \Length: 340    Total Reward: 251.52
Episode: 6      \Length: 238    Total Reward: 241.30
Episode: 7      \Length: 223    Total Reward: 297.48
Episode: 8      \Length: 237    Total Reward: 301.72
Episode: 9      \Length: 337    Total Reward: 281.12
Average Reward 248.71302242610136
```

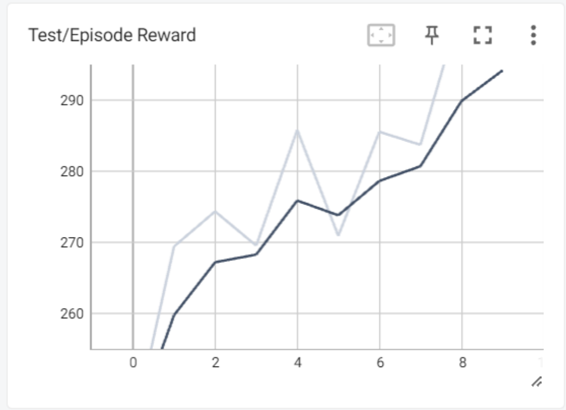
(b) LunarLanderContinuous-v2 using DDPG

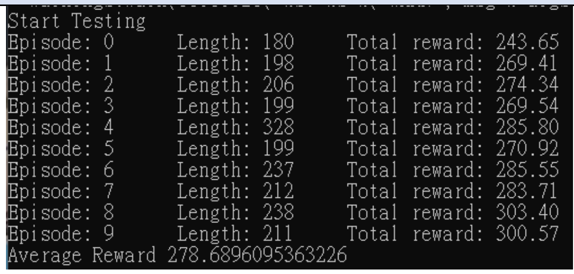
Train



(b) LunarLanderContinuous-v2 using DDPG

Test

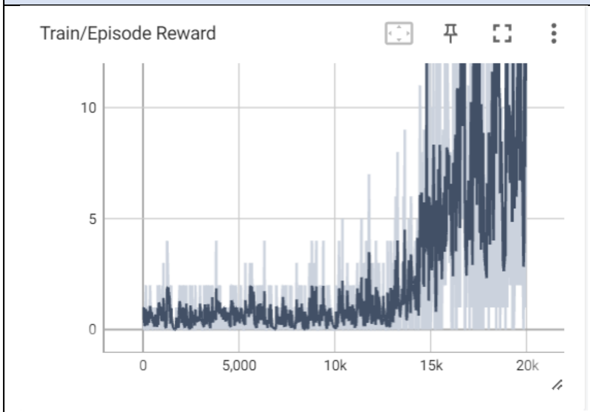


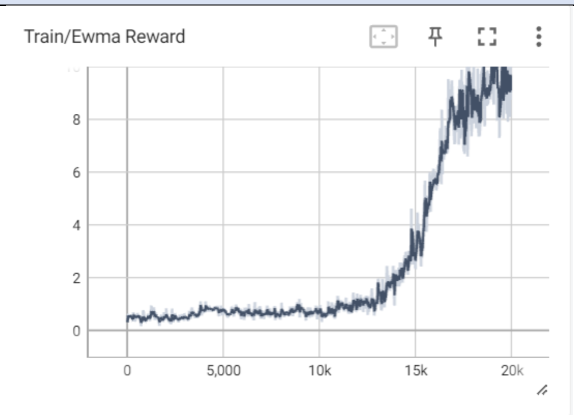


```
Start Testing
Episode: 0      Length: 180     Total reward: 243.65
Episode: 1      Length: 198     Total reward: 269.41
Episode: 2      Length: 206     Total reward: 274.34
Episode: 3      Length: 199     Total reward: 269.54
Episode: 4      Length: 328     Total reward: 285.80
Episode: 5      Length: 199     Total reward: 270.92
Episode: 6      Length: 237     Total reward: 285.55
Episode: 7      Length: 212     Total reward: 283.71
Episode: 8      Length: 238     Total reward: 303.40
Episode: 9      Length: 211     Total reward: 300.57
Average Reward 278.6896095363226
```

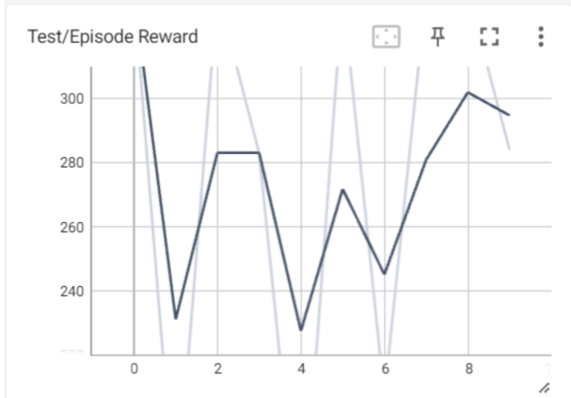
(c) BreakoutNoFrameskip-v4 using DQN

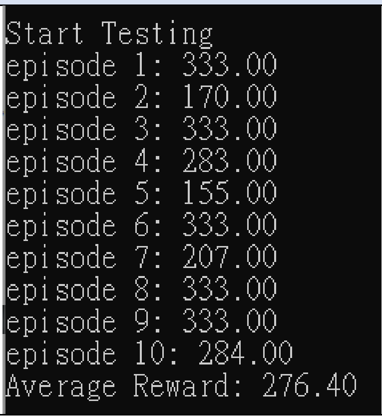
Train





Test



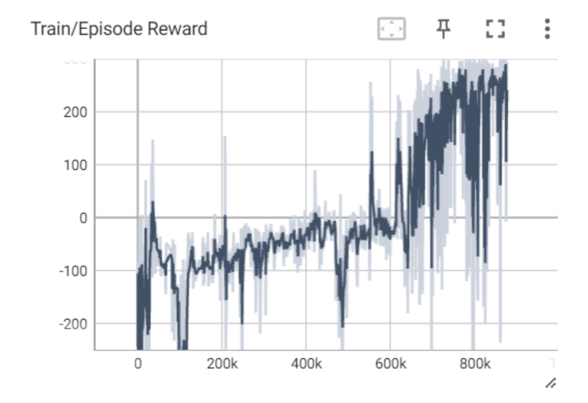


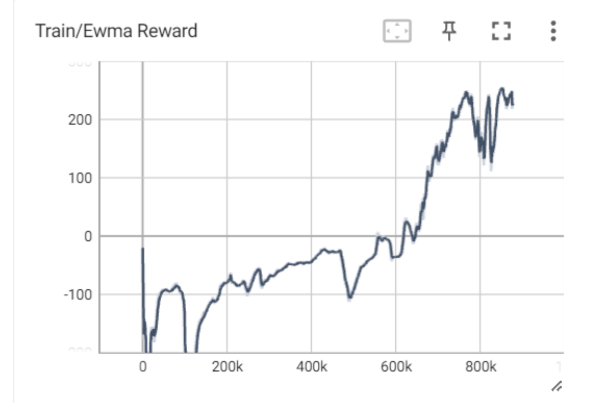
```
Start Testing
episode 1: 333.00
episode 2: 170.00
episode 3: 333.00
episode 4: 283.00
episode 5: 155.00
episode 6: 333.00
episode 7: 207.00
episode 8: 333.00
episode 9: 333.00
episode 10: 284.00
Average Reward: 276.40
```

(2) Bonus:

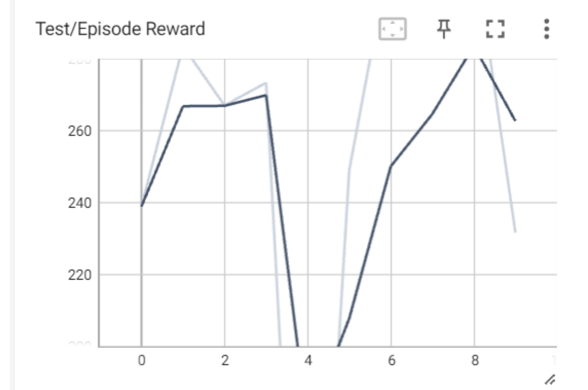
LunarLander-v2 using DDQN

Train





Test




```
Start Testing
Episode: 0      \Length: 241    Total Reward: 238.96
Episode: 1      \Length: 226    Total Reward: 283.52
Episode: 2      \Length: 267    Total Reward: 267.01
Episode: 3      \Length: 265    Total Reward: 273.32
Episode: 4      \Length: 154    Total Reward: 58.19
Episode: 5      \Length: 291    Total Reward: 248.98
Episode: 6      \Length: 237    Total Reward: 310.50
Episode: 7      \Length: 238    Total Reward: 285.83
Episode: 8      \Length: 249    Total Reward: 311.48
Episode: 9      \Length: 531    Total Reward: 231.65
Average Reward 250.9436356482252
```

(3) Questions:

(a) Your implementation of Q network updating in DQN.

According to the algorithm, I will sample random minibatch of transition $(\phi_t, a_t, r_t, \phi_{t+1})$ from replay memory, then compute target value, and after that using MSE as loss function to update behavior net. In the update_target_network function, it's copy weights of behavior net to update target network every C steps.

```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 = ?
    # with torch.no_grad():
    #    q_next = ?
    #    q_target = ?
    # criterion = ?
    # loss = criterion(q_value, q_target)
    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), 1)[0].view(-1, 1)
        q_target = reward + q_next * gamma * (1.0 - 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()

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())
```

(b) Your implementation and the gradient of actor updating in DDPG

According to the algorithm, it's same as DQN for the first, and let behavior actor network predict an action which can get maximum q value by critic network, using actor network to generate actions, and then get the q value by critic network, and after that computing the mean q value and back-propagation to update actor network. Therefore, we define actor loss as follows.

(c) Your implementation and the gradient of critic updating in DDPG

Like DQN, I will sample random minibatch from replay memory, and use actions of batch to calculate corresponding q value by critic network, also need to calculate target value. And after that, get action of next state by target actor network, generate the q value of next state by target critic network, multiply discount factor and plus reward, we can get the target q value. Finally, we can calculate MSE loss by q value and target q value.

```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 = ?
    # with torch.no_grad():
    #    a_next = ?
    #    q_next = ?
    #    q_target = ?
    # criterion = ?
    # critic_loss = criterion(q_value, q_target)
    q_value = critic_net(state, action)
    with torch.no_grad():
        q_next = target_critic_net(next_state, target_actor_net(next_state))
        q_target = reward + gamma * q_next * (1 - done)

    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_loss = ?
    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()
```