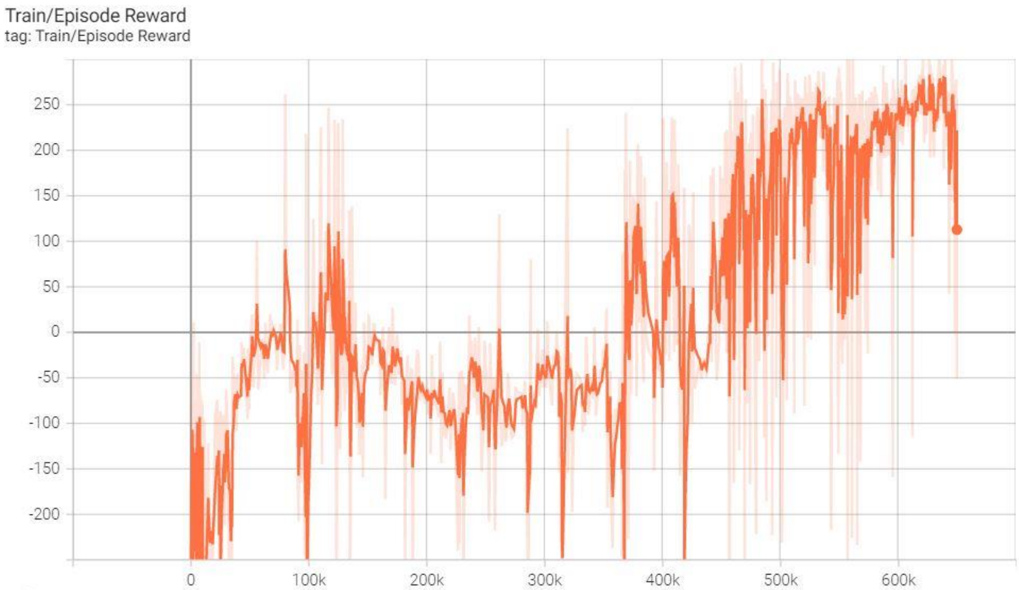
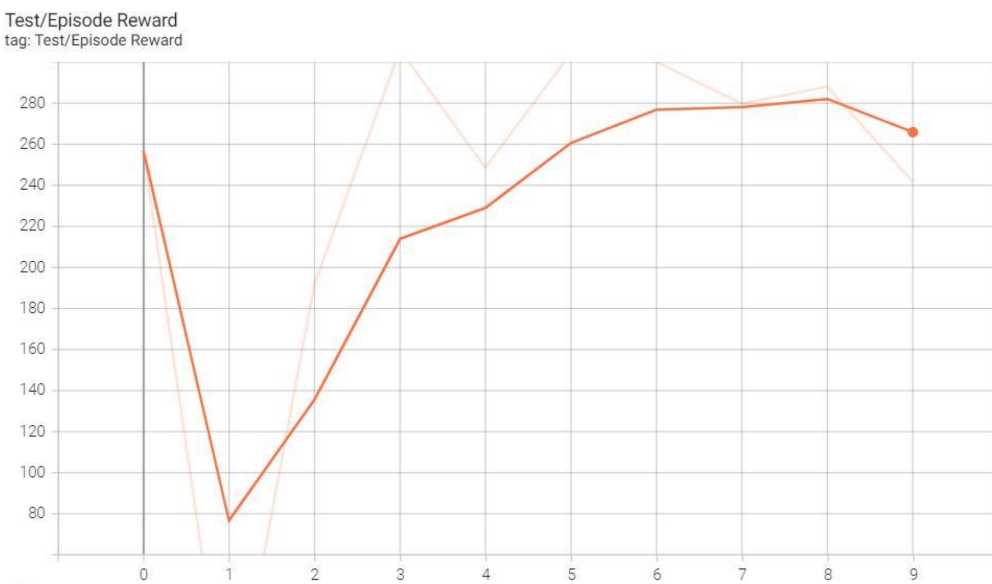
1. **Show episode rewards in LunarLander-v2**

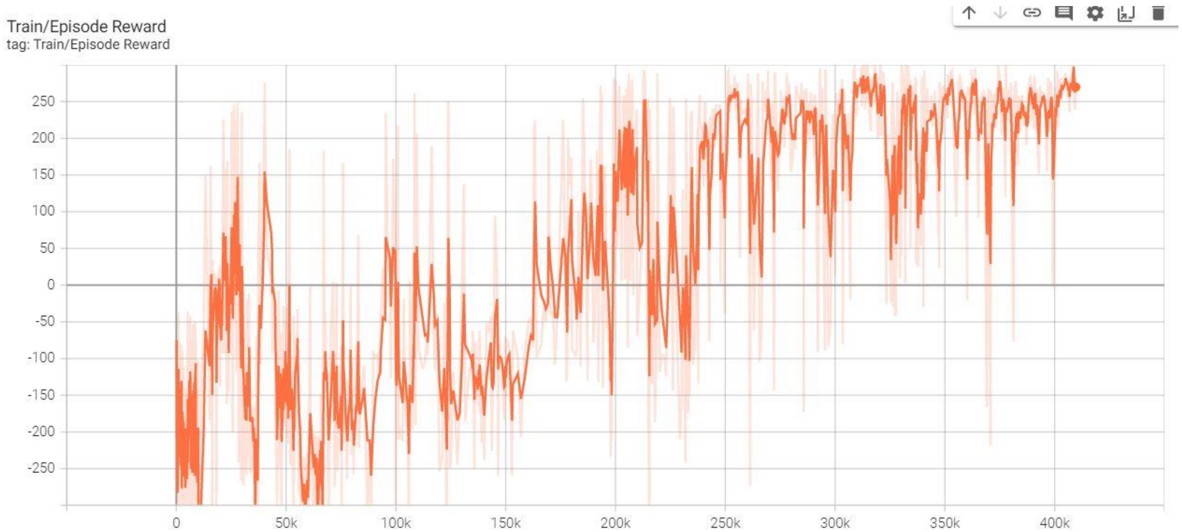
```
Average Reward 216.7
```

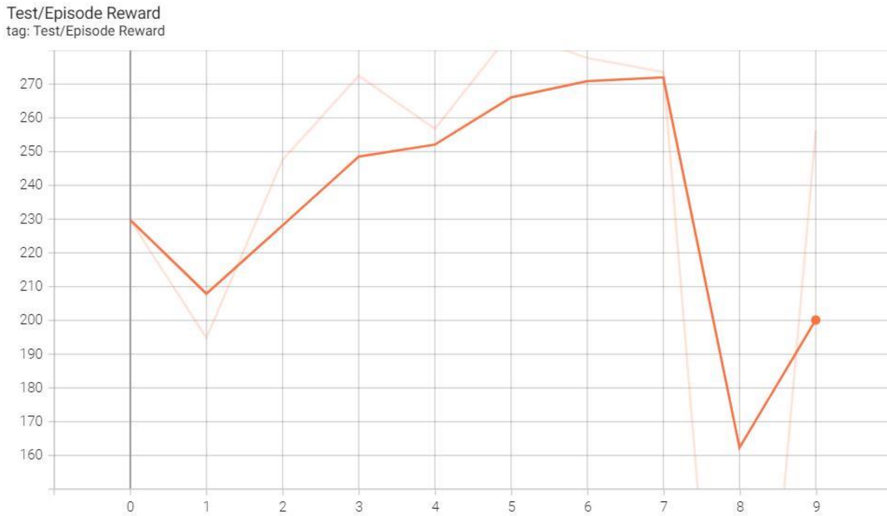




2. Show episode rewards in LunarLanderContinuous-v2

```
Average Reward 229.49
```





3. Describe your major implementation of both algorithms in detail.

DQN

a. Reply buffer 中，用來存取曾經 sample 過的結果，讓 network 可以一次抽取 n 筆資料，且可以更有效率地做訓練。

```
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))
```

b. 選擇 action：有ε的的機率選擇到任意的 action，其餘則是選擇 q value 最大的 action。

```
    def select_action(self, state, epsilon, action_space):
        '''epsilon-greedy based on behavior network'''
         ## TODO ##
        if random.random() < epsilon:
            random_action = action_space
            return random_action.sample()
        with torch.no_grad():
            every_probability = self._behavior_net(state)
            best_action = torch.argmax(every_probability)
            return best_action.item()
```

c. 將 action 丟入環境中，得到 reward 跟新的 state。

```
# execute action
next_state, reward, done, _ = env.step(action)
```

d. behavior network：每走 freq 步更新一次 behavior network。

```
def _update_behavior_network(self, args, 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(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.to(args.device) * gamma * (1.0 - done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

e. target network：每走 target_freq 步再更新一次。

```
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())
```

DDPG

a. 有與 DQN 相同架構的 Replay Buffer
b. 選擇 Actor network 與 Critic network 的架構。
c. 選擇 action：在 action 上任意的加上高斯雜訊，以提升最後的 performance

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    if noise:
        random_noise = self._action_noise.sample()
        action = self._actor_net(state).detach().cpu().numpy() + random_noise
        return action
    else:
        action = self._actor_net(state).detach().cpu().numpy()
        return action
```

d. 將 action 丟入環境中，得到 reward 跟新的 state。
e. 從 Replay Buffer 中取得 N 筆資料
f. 更新 behavior network：
Critic loss 為公式(1)，也就是使用 Mean square error 取其 loss。

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

Actor loss 為公式(2)，實作的部分則是根據公式(3)取其負值計算 loss。

$$\nabla_{\theta^\mu}\mu|s_i \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i \tag{2}$$

$$\mathrm{J} = \sum_n Q(s_n, \pi(s_n)) \tag{3}$$

```
    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 = critic_net(state, action)
```

```python
with torch.no_grad():
    a_next = target_actor_net(next_state)
    q_next = target_critic_net(next_state, a_next)
    q_target = reward + q_next * gamma * (1.0 - 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_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()
```

g. 更新 target network

```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.

DQN

a. 丟進 Net 中的 state 都需要轉換成 tensor 和 cuda 使用
b. 在選擇 batch size 中最好的 action 時，需保持在 tensor，若中途轉換成 numpy 會使訓練的結果較差。
c. Algorithms 中有，x、s、ϕ三個參數。我的理解中，x 為一張照片，s 為照片中得到的資料，ϕ則是資料轉換成的 state，因此 action 丟入環境 以後還需要再做一些處理，但在本次的實驗中並沒有用到這麼複雜的轉換，而是將 action 丟入環境後，就可以直接的到對應的 state。

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

因為需要將公式(3)最大化，所以以公式(3)的負值為 actor loss。

```
## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
```

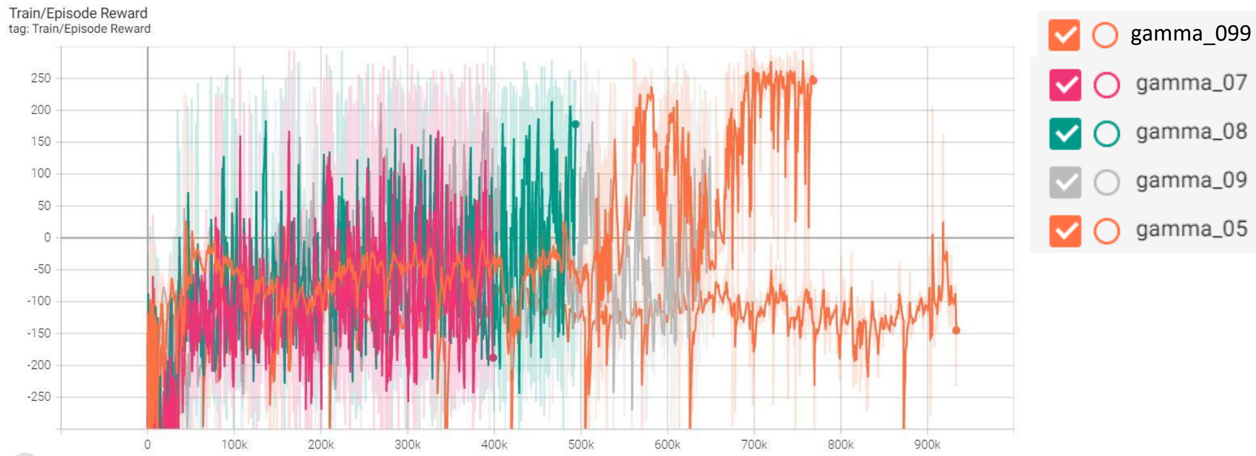
6. Describe your implementation and the gradient of critic updating.

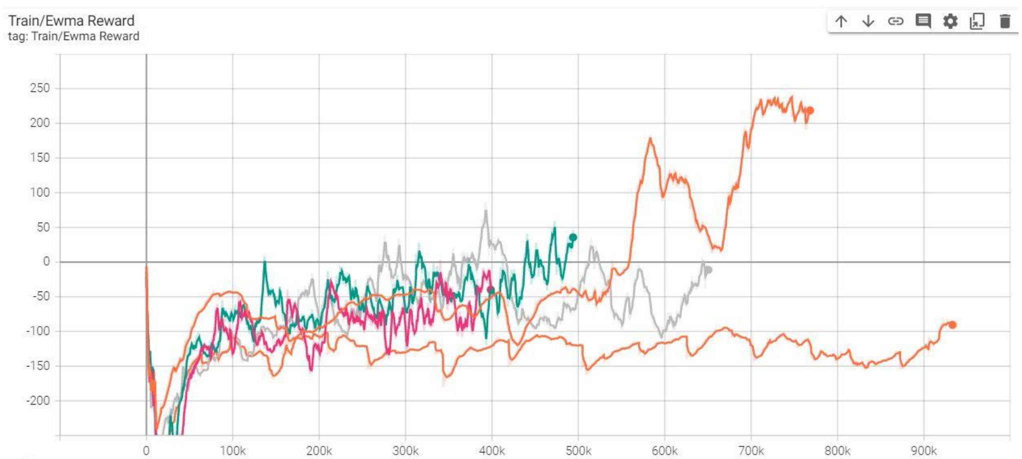
根據公式(1)，用 mean square error 計算 critic loss。

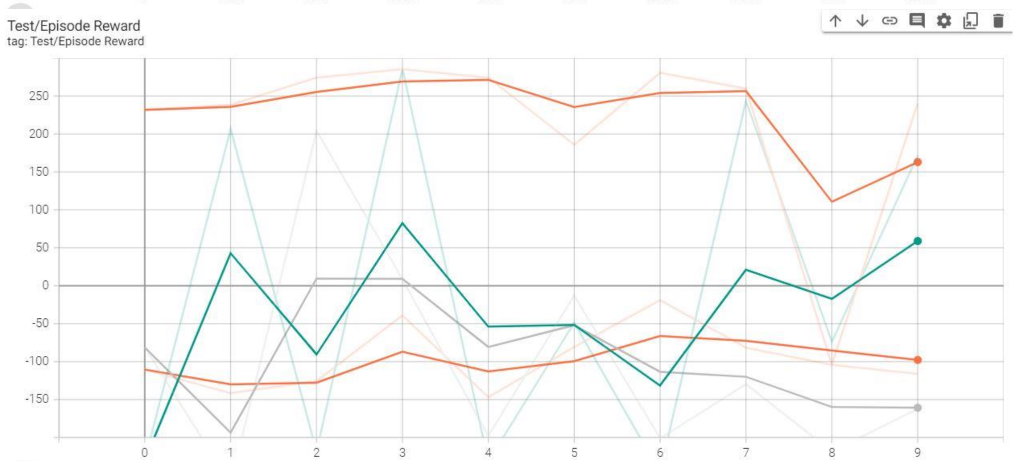
```
## update critic ##
# critic loss
## TODO ##
q_value = critic_net(state, action)
with torch.no_grad():
   a_next = target_actor_net(next_state)
   q_next = target_critic_net(next_state, a_next)
   q_target = reward + q_next * gamma * (1.0 - done)
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
```

7. Explain effects of the discount factor.

Gamma = 0.99、0.9、0.8、0.7、0.5







從最後的 test 來看，並沒辦法說明 gamma 越大就能使其結果越好，但可以從 training 的結果似乎可以發現，當 gamma 越大時，其 total step 的步數似乎也越多。

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

greedy action selection 是選擇最大的 action value 做為下一次的 action，但如果有些沒有被 sample 過的 action，其 action value 會一直降低，因此導致無法被選擇到，而非因為動作產生的結果較差。Epsilon-greedy 就是為了減少這樣子的問題產生，所以會加入一個變數 epsilon，讓選擇 action 時，有部分的機會可以選擇到沒被 sample 過的 action。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        random_action = action_space
        return random_action.sample()
    with torch.no_grad():
        every_probability = self._behavior_net(state)
        best_action = torch.argmax(every_probability)
        return best_action.item()
```

9. Explain the necessity of the target network.

相較於 behavior network，target network 久久才會更新一次。因為在更新 behavior network 的算是中(4)，會需要用到 target network，假設兩個都是使用 behavior network 就會像是追著自己的尾巴一樣，使 network 很難穩定的收斂。

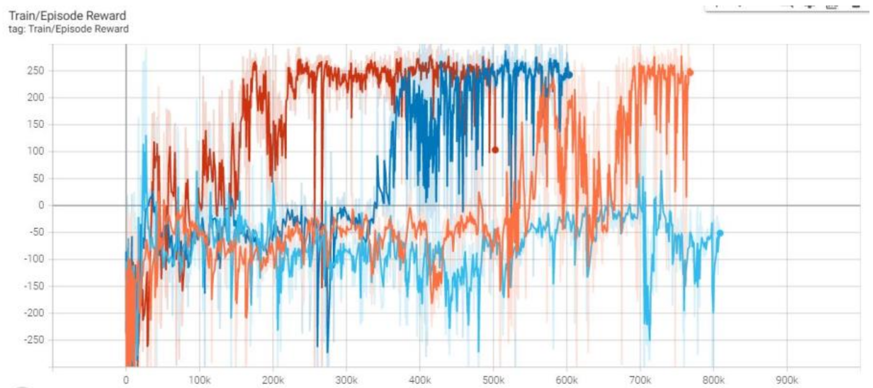
$$Q(s_t, a_t) = r_t + \gamma * Q'(s_{t+1}, a) \tag{4}$$

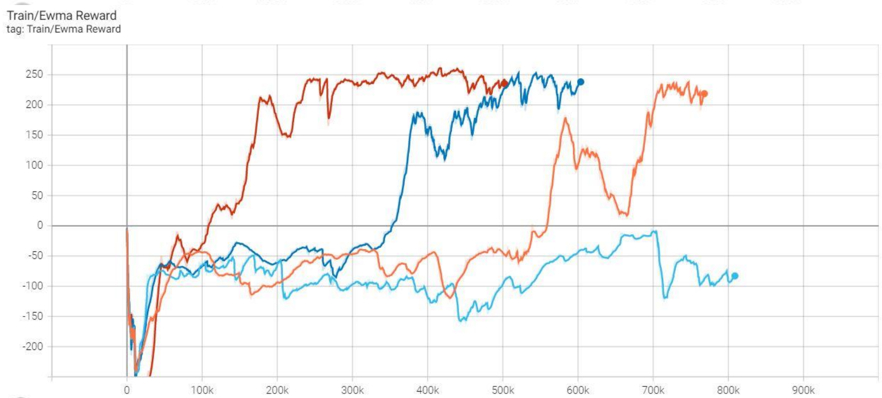
```
def _update_behavior_network(self, args, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    q_value = self._behavior_net(state).gather(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.to(args.device) * gamma * (1.0 - done)
```

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

Buffer size = 1000、10000、100000、1000000



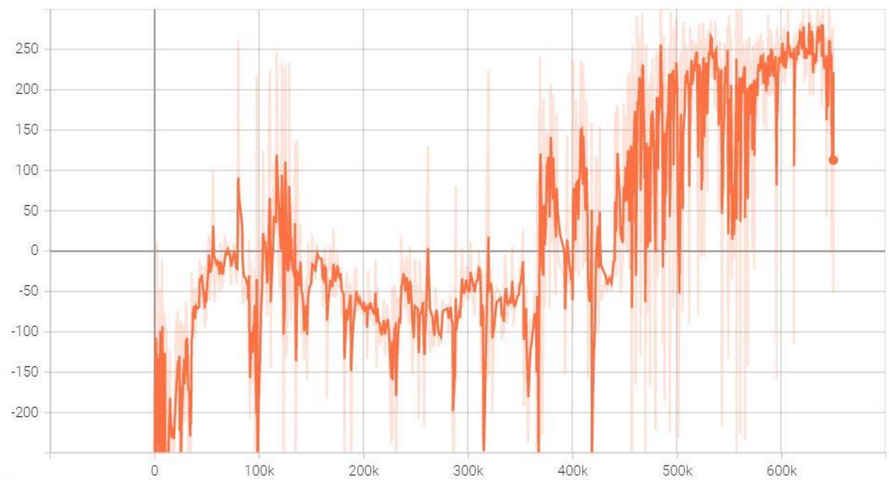


從結果可以看出，當 buffer size 越大時，可以越快訓練到較好的 performance。

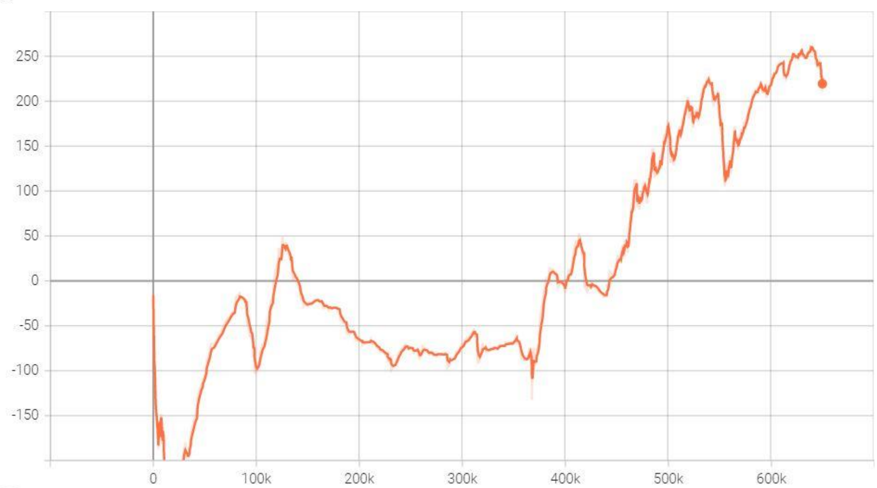
11. Bonus

Double dqn

Train/Episode Reward
tag: Train/Episode Reward



Train/Ewma Reward
tag: Train/Ewma Reward



Test/Episode Reward
tag: Test/Episode Reward