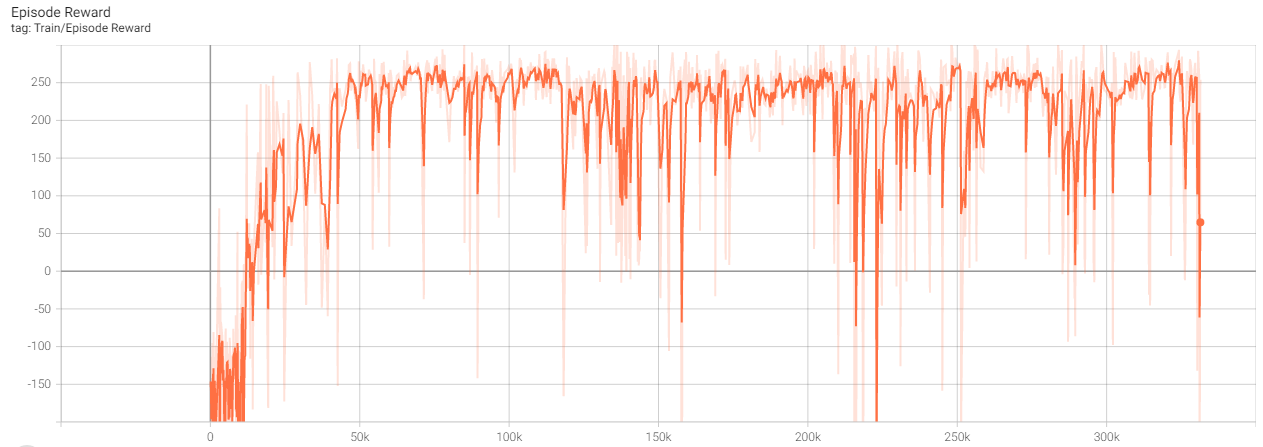
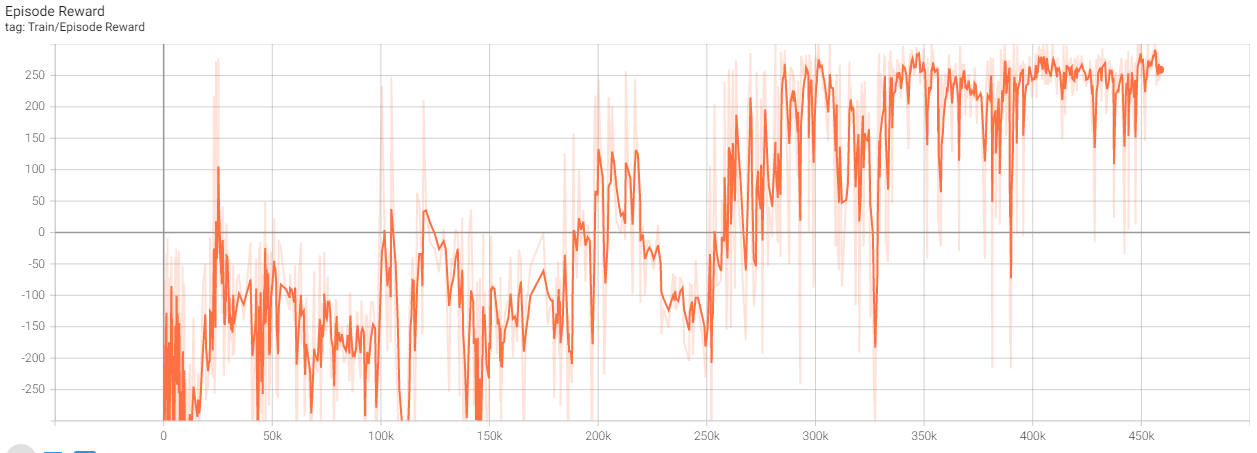
**Report (80%)**

**A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%)**



**A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2 (5%)**



**Describe your major implementation of both algorithms in detail. (20%)**

**DQN:**

首先我們根據specification 來定義我們的網絡，輸入是8個state，輸出是4個action。

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

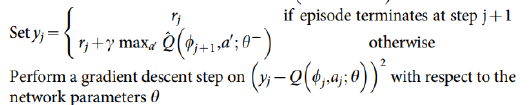
在DQN裡選用optimizer為Adam，因為Adam是比較常用和相對快收斂的loss function

class DQN:  
 def \_\_init\_\_(self, args):  
 ## *TODO ##* self.\_optimizer = torch.optim.Adam(self.\_behavior\_net.parameters(), lr=args.lr)

我們選用了一個慢慢減少的epsilon，一開始先隨機的行動，然後得到reward，一段時間後慢慢把epsilon降低，讓action根據Q-value來決定下一個動作。

def select\_action(self, state, epsilon, action\_space):  
 *'''epsilon-greedy based on behavior network'''* ## *TODO ##* if random.random() < epsilon:  
 action = action\_space.sample()  
 else:  
 action = torch.argmax(self.\_behavior\_net(torch.tensor(state).to(self.device))).item() # get propability of output and take the largest one as action  
 return action

更新網絡行為網絡的參數，先從memory 抽樣出環境參數，然後根據當前state在behavior\_network的輸出作為q\_value，然後如下公式︰用rewrad加上gamma 乘以下一個state的最大Q\_value來作為Q\_target，用以和原來state的q\_value作比較，從而得出loss並更新網絡參數。



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(1, action.long()) # gather Q\_value(ppb) from NET according to selected action, transfer to long(int) for .gather  
 with torch.no\_grad():  
 q\_next = torch.max(self.\_target\_net(next\_state), dim=1)[0].view(-1, 1) # get target Q\_max(dim=1) value[0] from next state and convert to (-1,1) dimension  
 q\_target = reward + gamma \* q\_next \* (1 - done) # if done(end game) q\_target = 0  
 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()

更新target\_network就只是簡單的把behavior\_network的參數抄去把target\_network更新，我們可以根據參數來調節更新的頻率。

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())

這邊基本上和train network時要做的事差不多，把模型選擇的action作為實際的行動，並根據環境的feedback來記錄reward的分數。

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 ##* for t in itertools.count(start=1):  
 # select action  
 action = agent.select\_action(state, epsilon, action\_space)  
 # execute action  
 next\_state, reward, done, \_ = env.step(action) # done - end game  
 # store transition  
 agent.append(state, action, reward, next\_state, done)  
  
 state = next\_state  
 total\_reward += reward  
  
 if done:  
 writer.add\_scalar('Test/Episode Reward', total\_reward, n\_episode)  
 print('Episode: {}\tLength: {:3d}\tTotal reward: {:.2f}\tEpsilon: {:.3f}'.format(n\_episode, t, total\_reward, epsilon))  
 rewards.append(total\_reward)  
 break

**DDPG:**

Replay memory就是記錄之前的環境參數，裡面的sample是方便我們從中抽取一個作為環境參數，根據輸入的batch size來決定要抽出多少個sample然後解壓成我們需要的樣式。

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'''* ## *TODO ##* transitions = random.sample(self.buffer, batch\_size) # sample: (state(8), action(2), reward(1), next\_state(8), done(1)) \* batch\_size  
 # unzip transitions to state\_vector, action\_vector, reward\_vector, next\_state\_vector, done\_vector  
 return (torch.tensor(out, dtype=torch.float, device=device) for out in zip(\*transitions)) # transfer from double to torch.float  
 # raise NotImplementedError

ActorNet是LunarLanderContinuous-v2中用的網絡，大體上和dqn差不多，輸入為8，輸出為2都在範圍-1到1的中，所以最後把輸出接上Tanh方便直接使用。

CriticNet則是根據state和action來輸出Q\_value作為評分的網絡，這也是和DQN單純一個Net不一樣的地方。

class ActorNet(nn.Module):  
 def \_\_init\_\_(self, state\_dim=8, action\_dim=2, hidden\_dim=(400, 300)):  
 super().\_\_init\_\_()  
 ## *TODO ##* h1, h2 = hidden\_dim  
 self.actor = nn.Sequential(  
 nn.Linear(state\_dim, h1),  
 nn.ReLU(),  
 nn.Linear(h1, h2),  
 nn.ReLU(),  
 nn.Linear(h2, action\_dim),  
 nn.Tanh(),  
 )  
  
 # raise NotImplementedError  
  
 def forward(self, x):  
 ## *TODO ##* return self.actor(x)

和DQN一樣選擇Adam作為Optimizer，然後設定不一樣的learning rate。

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 = torch.optim.Adam(self.\_actor\_net.parameters(), lr=args.lra)  
 self.\_critic\_opt = torch.optim.Adam(self.\_critic\_net.parameters(), lr=args.lrc)  
 # action noise  
 self.\_action\_noise = GaussianNoise(dim=2)  
 # memory  
 self.\_memory = ReplayMemory(capacity=args.capacity)  
 ## config ##  
 self.device = args.device  
 self.batch\_size = args.batch\_size  
 self.tau = args.tau  
 self.gamma = args.gamma  
 self.test\_only = args.test\_only

根據公式在select action上加上noise幫助network更好的探索，讓action 不要一直只選擇最優解，有一定機率觸發到不一樣的action，但基本上類似的action應該是有差不多的q\_value。



def select\_action(self, state, noise=True):  
 *'''based on the behavior (actor) network and exploration noise'''* ## *TODO ##* state = torch.tensor(state).to(self.device)  
 action = self.\_actor\_net(state)  
 if noise:  
 action += torch.tensor(self.\_action\_noise.sample()).to(self.device)  
 return action.cpu().detach().numpy()

更新方面，如上所述會以criticNet作為q\_value的評分，然後用q\_target和q\_value做MSE取得loss，更新critic的參數，actor則用minimize – q\_value來作為loss更新參數，由於max Q\_value 的功效等價於min Q\_value。



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)  
 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()  
 # print(q\_value, q\_target)  
 critic\_loss = criterion(q\_value, q\_target)  
 # raise NotImplementedError  
 # 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() # min q\_value: select max = -min  
 # optimize actor  
 actor\_net.zero\_grad()  
 critic\_net.zero\_grad()  
 actor\_loss.backward()  
 actor\_opt.step()

更新target\_network則根據target network的公式，用average的方式更新。



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 + (1 - tau) \* target)

test function基本上和DQN一樣，但要記得把noise關掉。

def test(args, env, agent, writer):  
 print('Start Testing')  
 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 ##* for t in itertools.count(start=1):  
 # select action  
 action = agent.select\_action(state, noise=False)  
 # execute action  
 next\_state, reward, done, \_ = env.step(action) # done - end game  
 # store transition  
 agent.append(state, action, reward, next\_state, done)  
  
 state = next\_state  
 total\_reward += reward  
  
 if done:  
 writer.add\_scalar('Test/Episode Reward', total\_reward, n\_episode)  
 print('Episode: {}\tLength: {:3d}\tTotal reward: {:.2f}'.format(n\_episode, t, total\_reward))  
 rewards.append(total\_reward)  
 break

**Describe differences between your implementation and algorithms. (10%)**

主要是三個點︰

1. 模型開始時的隨機action，根據warmup的大小來先跑出環境參數並存入replay memory中。
2. 在更新target\_network時，實際並不是每次都更新，可以利用freq來延遲更新。
3. Actor\_loss用-min Q\_value來實行max Q\_value，藉此減少loss。

Describe your implementation and the gradient of actor updating. (10%)

因為我們希望令Q\_value最大化，而用上最少化的-Q 其實也是同樣效果，用Q\_value的值來更新的網絡。

action = actor\_net(state)  
actor\_loss = -critic\_net(state, action).mean() # min q\_value: select max = -min  
  
# 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. (10%)**

Critic更新利用target\_net預測的action來作為下一個action’s q\_value的預測，以前來得出q\_target和原先的q\_value比較得出loss更新critic的參數。

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()  
# print(q\_value, q\_target)  
critic\_loss = criterion(q\_value, q\_target)  
# raise NotImplementedError  
# optimize critic  
actor\_net.zero\_grad()  
critic\_net.zero\_grad()  
critic\_loss.backward()  
critic\_opt.step()

**Explain effects of the discount factor. (5%)**

因為gamma和q\_next相乘的，所以可以解釋成q\_target考慮下一步q\_value的比例，以此來加入預測未來分數的考量，而不單單只考慮當前action 的reward。

**Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)**

Epsilon其實就是讓模型多考量不一樣的action，不要單單只看q\_value來決定action，若模型因為network一直選擇同一個action，可能會導致有更好的action但沒機會採用，多探索不一樣的路，讓模型不會止步不前。在初始化也是很有用，能夠讓模型得到一定的q\_value，知道哪個action較好，在DQN上還將epsilon遞減，以此讓模型漸漸依賴q\_value判定action。

**Explain the necessity of the target network. (5%)**

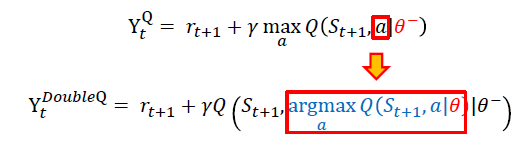
Target\_network可以使更新時更加穩定，不會因為只有一個網絡導致更新時數值一直變動，容易有很大的變化。

**Explain the effect of replay buffer size in case of too large or too small. (5%)**

1. 太大的話training時間會變得很長，亦容易得到關聯性較低的sample，反之考慮亦是較為全面的訊息。
2. 太小的話會一直sample類同甚至一樣的環境參數，buffer就沒有那麼有效，且可能會造成overfitting。

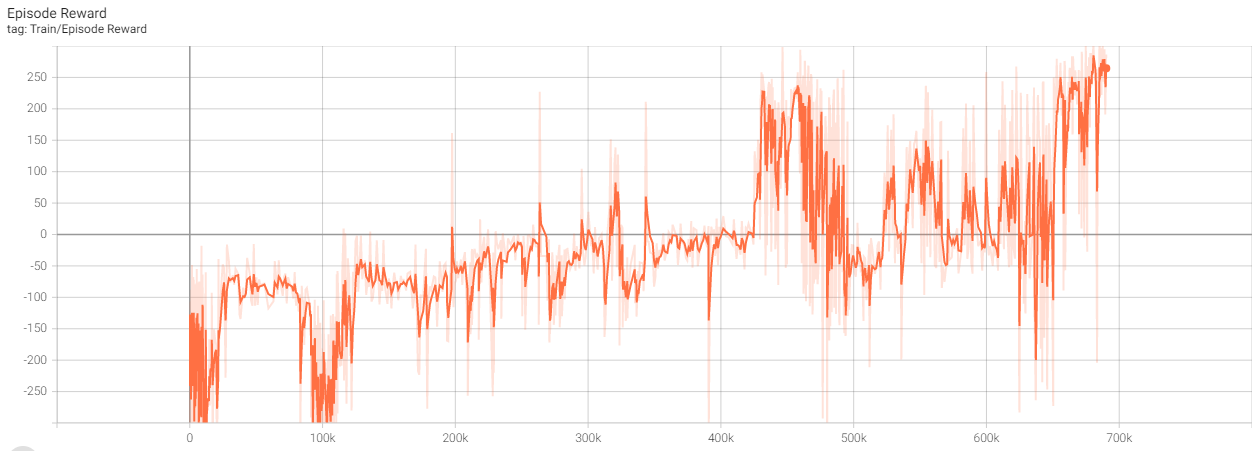
**Report Bonus (25%)**

**Implement and experiment on Double-DQN (10%)**DQN和Double-DQN最大的差別在於計算Q\_next的方法(如下所示)，DQN會先target\_network裡的最大Q值作為Q\_next用來計算Q\_target，而DDQN則是把action裡擁有最大Q值的動作找出來作為Q\_target的action。這樣可以減少最大Q值作為Q\_target的差距，以貼近實際Q值。



實際的coding 如下所示，先找出behavior\_net裡的最大值作為action\_next，然後根據此action實際的Q值作為Q\_next，最後運算Q\_target。

q\_value = self.\_behavior\_net(state).gather(1, action.long())   
with torch.no\_grad():  
 action\_next = torch.argmax(self.\_behavior\_net(next\_state), dim=1).view(-1, 1) # take max q\_value action as next action index  
 q\_next = self.\_target\_net(next\_state).gather(dim=1, index=action\_next.long()) # take t\_net q\_value according to b\_net's predict action  
 q\_target = reward + gamma \* q\_next \* (1 - done)

**Tensorboard plot of DDQN episode rewards:**

**Testing Reward of DDQN:**



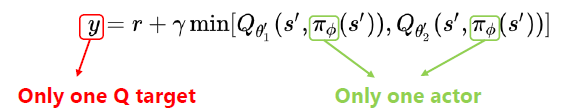
Reward比原來的DQN更好，我猜想是因為選到的動作較原來的DQN準確，導致更新的值更為直接，更容易重現以前選對的動作。

**Implement and experiment on TD3 (Twin-Delayed DDPG) (10%)**

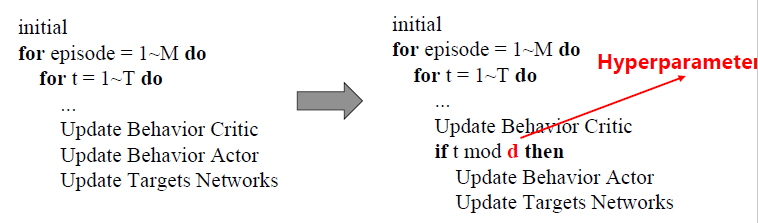
TD3主要有3個trick︰

1. Clipped Double-Q Learning︰用2個Q-learning network(實驗裡則為2個critic\_net)來取代原本只有1個network生成的Q\_next，然後再取2個network中較為小的q\_value作為q\_next，最後計算出1個target\_net的q\_target

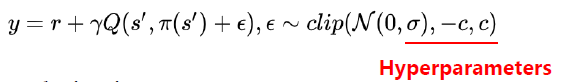




1. Delay Policy Updates︰顧名思義就是延遲更新Behavior Actor和Target Network的頻率。



1. Target Policy Smoothing︰也是對Target policy 著手，在action加上noise，期望action不會被卡在一個點，因為類似的action會有類似的reward。



Implementation:

CriticNet改成2個network有2個輸出q\_value，同時新加1個單輸出的criticNet作為target\_net(如下)︰

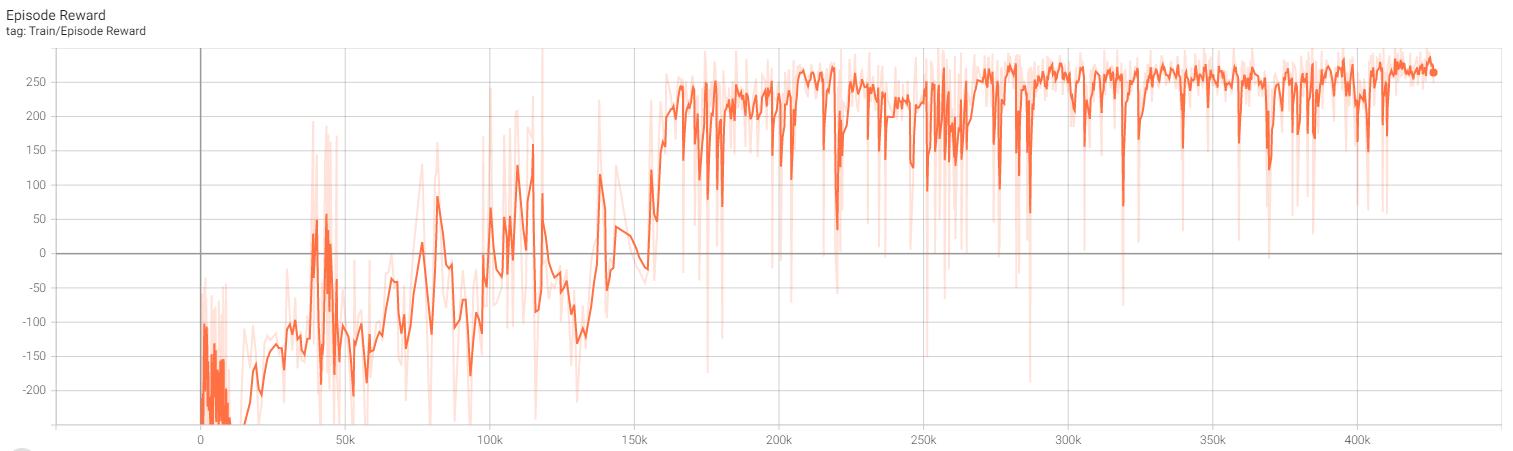
class CriticNet(nn.Module):  
 def \_\_init\_\_(self, state\_dim=8, action\_dim=2, hidden\_dim=256):  
 super(CriticNet, self).\_\_init\_\_()  
 self.relu = nn.ReLU()  
 # critic1  
 self.q1\_fc1 = nn.Linear(state\_dim + action\_dim, hidden\_dim)  
 self.q1\_fc2 = nn.Linear(hidden\_dim, hidden\_dim)  
 self.q1\_fc3 = nn.Linear(hidden\_dim, 1)  
  
 # critic2  
 self.q2\_fc1 = nn.Linear(state\_dim + action\_dim, hidden\_dim)  
 self.q2\_fc2 = nn.Linear(hidden\_dim, hidden\_dim)  
 self.q2\_fc3 = nn.Linear(hidden\_dim, 1)  
  
 def forward(self, state, action):  
 x = torch.cat([state, action], 1)  
 x1 = self.q1\_fc1(x)  
 x1 = self.relu(x1)  
 x1 = self.q1\_fc2(x1)  
 x1 = self.relu(x1)  
 x1 = self.q1\_fc3(x1)  
  
 x2 = self.q2\_fc1(x)  
 x2 = self.relu(x2)  
 x2 = self.q2\_fc2(x2)  
 x2 = self.relu(x2)  
 x2 = self.q2\_fc2(x2)  
 return x1, x2 # output critic1 and 2  
  
 def single\_q(self, state, action):  
 x = torch.cat([state, action], 1)  
  
 x = self.relu(self.q1\_fc1(x))  
 x = self.relu(self.q1\_fc2(x))  
 x = self.q1\_fc3(x)  
 return x # output single critic

update\_behavior\_net的部份，3個trick都在裡面實現了，一開始把前面提到的第3點noise加到next\_action裡面，然後clip來限制noise不會導致action out of boundary。然後把next\_action輸入到上面第一點提到的clip double Q-learning，取出最小的那個Q值作為q\_next，用來更新q\_target，最後根據現在的literation step來決定delay policy update的時機，分別更新actor、target\_net。

Coding如下︰

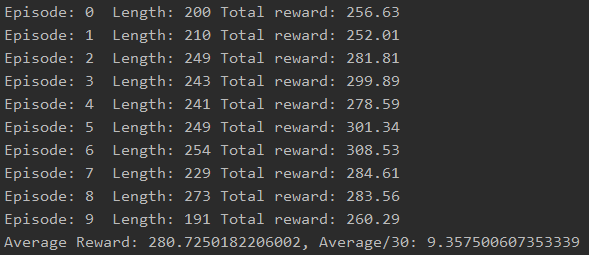
def \_update\_network(self, current\_step):  
 actor\_net, critic\_net, target\_actor\_net, target\_critic\_net = self.\_actor\_net, self.\_critic\_net, self.\_target\_actor\_net, self.\_target\_critic\_net  
 actor\_opt, critic\_opt = self.\_actor\_opt, self.\_critic\_opt  
  
 # sample a minibatch of transitions  
 state, action, reward, next\_state, done = self.\_memory.sample(self.batch\_size, self.device)  
 ## update critic ##  
 # critic loss  
 ## *TODO ##* with torch.no\_grad():  
 noise = (torch.randn\_like(action) \* self.noise).clamp(-self.cp, self.cp) # 3nd: select action and clip noise  
 next\_action = (target\_actor\_net(next\_state) + noise).clamp(-self.max\_action, self.max\_action)  
  
 # Compute q\_target value  
 q1\_next, q2\_next = target\_critic\_net(next\_state, next\_action)  
 q\_next = torch.min(q1\_next, q2\_next) # 2nd: min(double critic)  
 q\_target = reward + gamma \* q\_next \* (1 - done)  
  
 q1\_value, q2\_value = critic\_net(state, action) # get q\_value from both critic  
 critic\_loss = F.mse\_loss(q1\_value, q\_target) + F.mse\_loss(q2\_value, q\_target)  
  
 # optimize critic  
 critic\_opt.zero\_grad()  
 critic\_loss.backward()  
 critic\_opt.step()  
  
 # 2nd Delayed updates  
 if current\_step % self.update\_step == 0:  
 ## update actor ##  
 # actor loss  
 ## *TODO ##* actor\_loss = -critic\_net.single\_q(state, actor\_net(state)).mean()  
  
 # optimize actor  
 actor\_opt.zero\_grad()  
 actor\_loss.backward()  
 actor\_opt.step()  
  
 '''update target network by \_soft\_ copying from behavior network'''  
 ## *TODO ##* for param, target\_param in zip(critic\_net.parameters(), target\_critic\_net.parameters()):  
 target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)  
  
 for param, target\_param in zip(actor\_net.parameters(), target\_actor\_net.parameters()):  
 target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)

**Tensorboard plot of TD3 episode rewards:**

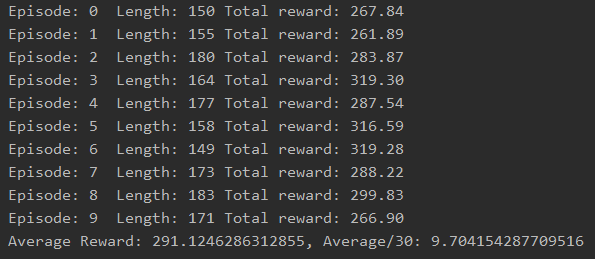


很明顯TD3有著更穩定的training reward，一直都很平穩，沒有太大起伏，也有了更多的hyper parameter可以調整。

**Testing Reward of TD3:**



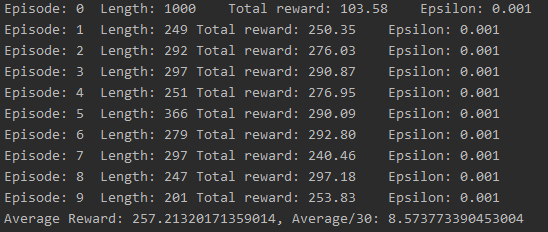
我認為TD3應該可以有更高的Reward，所以我調大了epoch，再train，得到291的reward，證明TD3是穩定而且有能力達到更高的reward。



**Performance (20%)**

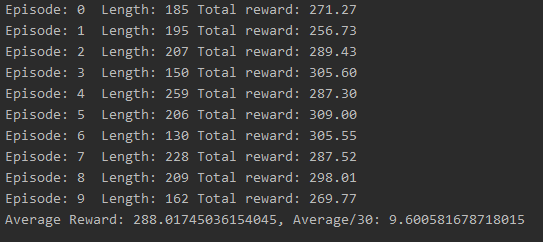
[LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30

**Testing Reward of DQN︰**



**Testing Reward of DDPG︰**

[LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30



在1200 Episode訓練中，DDPG是Reward最高的，有288分，其次就是TD3，280分，然後是DDQN 278分，DQN 257分。但調大Episode到5000後，TD3仍能平穩的提升，並獲得最高的reward。整體來說還是TD3較為可控穩定，可以根據情況調整需要探索的幅度，我的例子是相對平穩，但其實也可以把參數調整成比較注重探索的TD3。DQN其實由於預測每一步都是最大的reward導致可能出現選的action不是最好，但模型卻認為這個action的reward最大的情況，所以會出現Episode越大反而不一定學好的情況。DDQN相對的改善了這部份，可以看到相對於DQN來說變得更平穩了，學習更為正確。而DDPG就提出criticNet來幫助action的評分和加入noise幫助探索，所以結果亦相對更好，但探索越多就代表越不穩定，而TD3就相對穩定而且還保有探索的部份。下面是各個模型在epochs=5000的Episode Reward的比較，同樣得出TD3相對穩定的結論。

