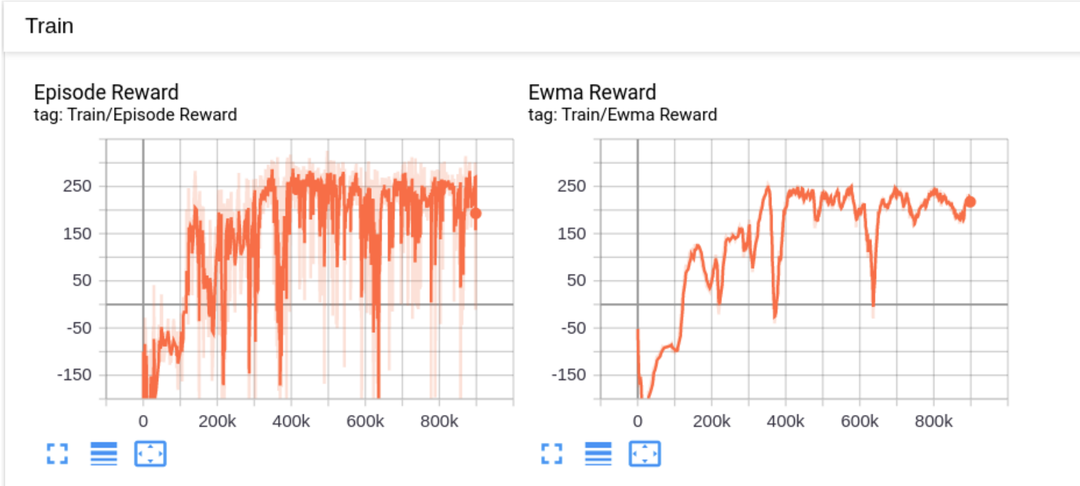
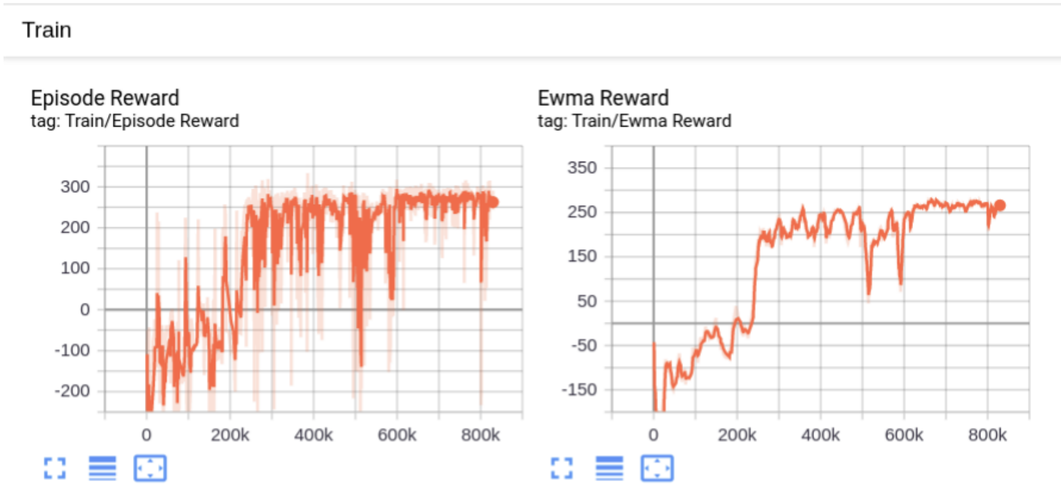
Report(120%)

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



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



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

DQN:

Network發現hidden size 32不太好avg大約240左右，後來調整程(500,400)

select_action的部份實做epsilon-greedy，作法是透過從uniform[0,1]中random sample出一個值，若這個值小於epsilon則做exploration反之則做greedy。

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    state = torch.from_numpy(state).float().to(self.device)
    flag = np.random.uniform(low=0.0, high=1.0)
    if(flag <= epsilon):
        action = action_space.sample()
    else:
        Q = self._behavior_net(state)
        action = torch.argmax(Q)
        action = action.item()
    return action
```

update_behavior的部份因為behavior_net的output是128*4，透過gather從dim=1的地方透過action作為index填入並回傳值，也就是對這mini batch採樣到的128個state中根據它真正做的action把Q值回傳，得到我們的q_value。
接著把next-state傳入targe_net去對每個next-state選讓Q值最大的action並透過view把它攤成128*1的，得到我們的q_next。
根據algorithm透過(1-done)判斷是否為terminal state，若是的話q_next為0，透過reward加上q_next來當作我們的target來預測這個state的q_target。
接著透過預測的q_target和q_value做loss並更新。

```
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) #[0] means tensro [1] means indice
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

target_net的更新僅是copy當前的behavior_net的weight

```
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:
sample的部份參考DQN的sample code，network及optim依據spec。
select_action的時候根據actor_net輸出的action再加入sample code提供的gaussian noise來作選擇，加入if(noise)條件來讓inference的時候可以選擇不加入雜訊來提高test品質，return type是由於gym在吃action的時候要是numpy的原因。

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.from_numpy(state).float().to(self.device)
    action = self._actor_net(state)
    if(noise):
        action = action + torch.from_numpy(self._action_noise.sample()).to(self.device)

    return action.cpu().detach().numpy()
```

更新的部份分兩部份 critic_net及actor_net:（解釋和report的5跟6重複，這邊可先略）
crtic的部份透過critic_net得到當前state執行action之後得到q_value，把next_state透過target_actor_net得到a_next，把next_sate跟a_next經過target_critic_net預測q_next再和DQN一樣透過reward+q_next來當作預測的q_target，接著透過q_value跟q_target來做loss並更新critic_net。

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

```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)
```

actor的部份參考algorithm，先把sate經過actor_net得到action，再把透過actor_net得到的action和state經過critic_net中後取mean作為actor_loss，所以在做gradient時可以透過偏微分來得到critic的gradient以及actor的gradient的乘積，且這邊僅是透過crtic算Q值而已，只有更新actor_net的參數而已，並沒有更新critic_net的，我們是透過把loss值加上一個負號來做ascent，因為我們希望透過更新actor的參數來使Q值越大越好。

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

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

在更新target_net的時候是採用soft update，並不會一次把整個behavior的weight替換上去，而是按照一定的比例替換

```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)
```

再做test的時候DQN跟DDPG大同小異，只是DDPG再做test的時候可以把noise關掉，直接透過actor_net去選，因為已經模型是訓練好的了，所以這樣選action會比較穩定

```python
        ## TODO ##
        for t in itertools.count(start=1):  # play an episode
#            env.render()
            # select action
            action = agent.select_action(state,noise=False)
            # 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
```

4. Describe differences between your implementation and algorithms. (10%)
   1.sample code中有提供warmup，在這個階段action就是隨機sample，去做探索並放到replay buffer中，且在實做DQN時並沒有像algorithm中對state去做preprocess，原因是因為gym給的state已經可以拿來訓練了不需要做預處理
   2.不是像algorithm中每個epsiode就更新一遍behavior而是每4次更新一次，還有個細微的差異是algorithm中只有對SE做gradient descent，我實做時是用MSE
   3.有發現episode最後訓練的結果可能會比中途的結果還來的差一點，所以有新增一小段程式碼把最好的checkpoint紀錄下來。

```
if(ewma_reward > best ):
    best = ewma_reward
    agent.save("DQN_best.pth")
```

5. Describe your implementation and the gradient of actor updating. (10%)
   actor的部份參考algorithm，先把sate經過actor_net得到action，再把透過actor_net得到的action和state經過critic_net中後取mean作為actor_loss，所以在做gradient時可以透過偏微分來得到critic的gradient以及actor的gradient的乘積，且這邊僅是透過crtic算Q值而已，只有更新actor_net的參數而已，並沒有更新critci_net的，我們是透過把loss值加上一個負號來做ascent，因為我們希望透過更新actor的參數來使Q值越大越好。

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

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

6. Describe your implementation and the gradient of critic updating. (10%)
   crtic的部份透過critic_net得到當前state執行action之後得到q_value，把next_state透過target_actor_net得到a_next，把next_sate跟a_next經過target_critic_net預測q_next再透過reward+q_next來當作預測的q_target，接著透過q_value跟q_target計算loss後並做gradient更新critic_net。

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

```
## 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)
```

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

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

根據total reward的公式，discount factor介於0到1之間，可以發現越未來的reward隨著dicount factor乘的越多次，所以考慮的比重越少，discount factor設的越大代表考慮未來state能獲得的reward越多，設的越小代表越專注於現在這個state的reward。

8. Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)
透過epsilon-greedy可以讓我們在選擇action的時候能夠把所有的狀況更完整的考慮進去，因為有時候Q值的初始化結果不好的情況下若只做greedy選法，可能會導致它只看到目前最好的，沒去嘗試也許一開始雖然差一點但之後可以獲得更多reward的動作，而導致agent沒去發現能產生更大reward的action而讓學習結果不良。

9. Explain the necessity of the target network. (5%)
透過target network能讓訓練時更穩定，因為target network是一段時間才更新一次，這樣這段期間在透過target network計算target並做gradient時都會是同樣parameter的network，不會變動這麼大，若不用target network的話可能會因為bootsrapping而導致overestimation使得網路變得很不穩定。

10. Explain the effect of replay buffer size in case of too large or too small. (5%)
若replay buffer很大的話可以確保訓練時有足夠的多樣性，可以考慮到更多更全面的transition，使得訓練更加穩定，但缺點是會花費更多的記憶體空間以及計算時間，若太小的話會導致replay buffer只有最近玩過的資料，使得訓練可能會有overfitting而train壞掉。

11. Report Bonus (20%)
(i)Implement and experiment on Double-DQN (10%)
DDQN跟DQN只有一個實做上的差別，DQN在計算q_target的時候是透過target network去選所有action裡面最大的q值來當作q_next，而DDQN在計算q_targe時會先透過behavior network選出所有action裡面最好的behavior_action，之後指定target network走這一個behavior_action去產生出q_next，再透過這個q_next來計算q_target，這樣做可以避免overestimation的問題。

```
    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():
            behavior_action = self._behavior_net(next_state).max(dim=1)[1].view(-1,1)#[0] means tensor [1] means indice
            q_next = self._target_net(next_state).gather(dim=1,index=behavior_action.long()) #[0] means tensor [1] means indice
            q_target = reward + gamma*q_next*(1-done)
        criterion = nn.MSELoss()
        loss = criterion(q_value, q_target)
```

inference結果：

```
Start Testing
total reward: 254.64
total reward: 283.41
total reward: 283.11
total reward: 273.95
total reward: 313.09
total reward: 258.27
total reward: 306.05
total reward: 290.81
total reward: 317.89
total reward: 300.62
Average Reward 288.18381039551696
```

(ii)Extra hyperparameter tuning, e.g., Population Based Training. (10%)
有調整episode的大小到2500來讓增加訓練次數，以及調整DQN的network從原本sample code的hidden size=32，改成hidden size = (500,400)來增加學習效果，從原本的avg:240到avg:287

12. Performance (20%)
(i)[LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30
DQN＆DDQN結果：
DQN: hidden=(500,400),episode=2500
```
Start Testing
total reward: 251.62
total reward: 280.75
total reward: 275.47
total reward: 278.40
total reward: 310.79
total reward: 274.90
total reward: 304.38
total reward: 294.95
total reward: 305.96
total reward: 295.00
Average Reward 287.2227931330424
```
DDQN:hidden=(400,300),episode=2500
```
Start Testing
total reward: 254.64
total reward: 283.41
total reward: 283.11
total reward: 273.95
total reward: 313.09
total reward: 258.27
total reward: 306.05
total reward: 290.81
total reward: 317.89
total reward: 300.62
Average Reward 288.18381039551696
```
(i)[LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30
DDPG:hidden=(400,300),episode=2500
```
Start Testing
total reward: 245.79
total reward: 275.79
total reward: 273.40
total reward: 268.73
total reward: 299.90
total reward: 265.59
total reward: 295.52
total reward: 287.93
total reward: 304.80
total reward: 257.96
Average Reward 277.5407738175004
```