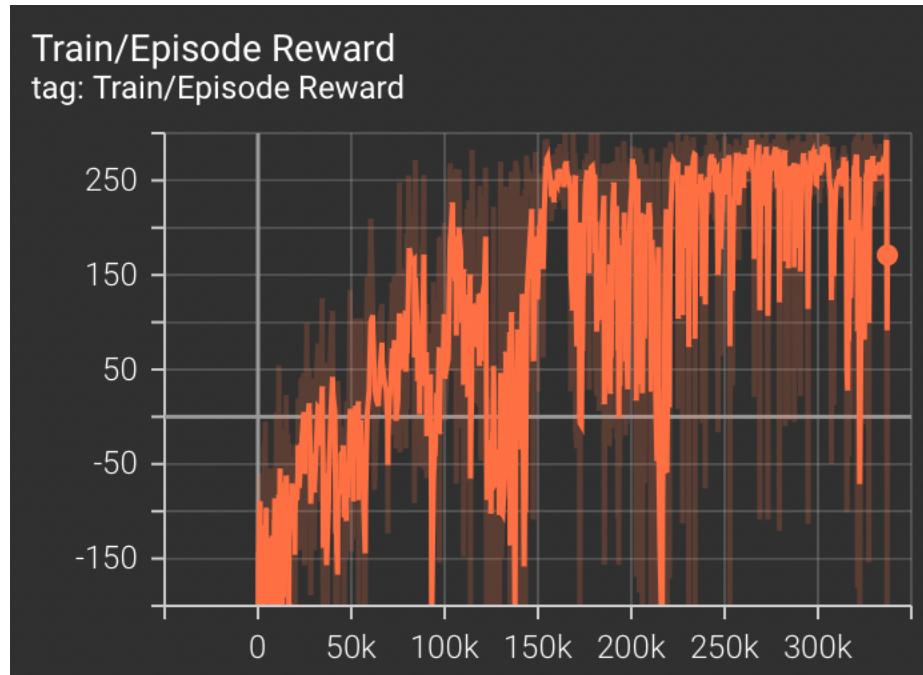
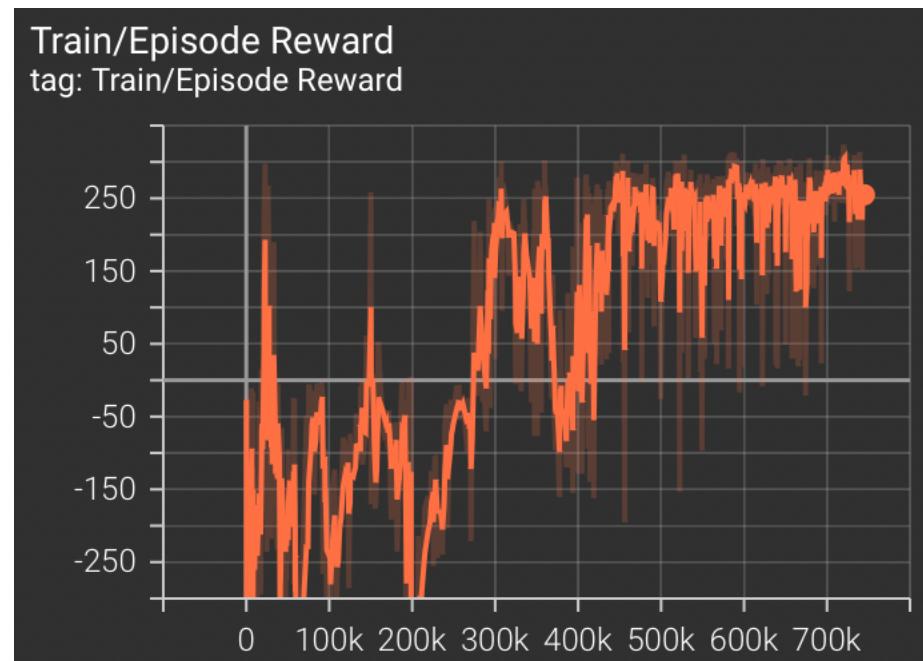


1. A tensor board plot shows episode rewards of at least 800 training episodes in LunarLander-v2.



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



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

## Deep Q-Learning

當 state 狀態是在連續空間的數值時，要建造一個 Q-table 並跟新是一件非常耗費空間的事情，所以我們會使用一個神經網路來代替我們的 Q-table，透過跟新網路參數的方式就可以達到跟新 Q-table 的效果。

而我們在 action space 只有四種可能的情況下，網路輸出只會有四種可能，我們將 state 當成網路輸入時就會得到有四個值的網路輸出，此時我們只要選擇 Q-value 最大的當成當前 state 的 action 即可。

```
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.activation = nn.ReLU()

    def forward(self, x):
        ## TODO ##
        output = self.activation(self.fc1(x))
        output = self.activation(self.fc2(output))
        output = self.fc3(output)
        return output
```

當我們在選擇 action 時，我們也不一定總是會選擇最大 Q-value 的 action，所以在程式實作時，我們訓練前段時會有比較高的機率，讓 agent 隨機探索不同的 action，到了訓練後半段才會有比較高的機率，選擇最大的 Q-value 當成 action。至於這麼做的原因後面會詳細解釋。

```
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():
            q_value = self._behavior_net(torch.from_numpy(state) \
                                         .view(1, -1).to(self.device))
        return q_value.max(dim=1)[1].item()
```

當我們選好當前 state 的 action 之後就可以丟進去環境裡進行互動，以得到當前 action 會獲得的 reward 以及下一個 state。我們這時要將蒐集到的 state, next\_state, action, reward 存進去我們事先建立的 buffer，在訓練時，我們會從 buffer 裡面隨機 sample 出資料，這是為了能打破連續資料之間的 dependency。

```
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))
```

在訓練過程中，我們會使用 target 以及 behavior 兩個網路，主要會跟新 behavior 這個網路裡面的參數，一段時間後我們才會將 behavior 的網路跟新至 target\_network，這麼做可以讓訓練較穩定。

我們會從 Buffer 中取得 batch\_size 大小的資料，將 state 送給 behavior 計算出 Q-value 並根據當初選擇的 action 得到當初估計的 Q-value，然後將 next\_state 送給 target\_network 並根據當前網路內部的參數以及 Bellman equation 算出預測的 Q-value，此時透過 Temporal-difference 去計算預測與前一步估計的 Q-value 之間的 MSELoss，就可以使用 Back propagation 更新網路參數。

```

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 = self._behavior_net(state)
    q_value = q.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)
    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()

```

下方是整體的訓練過程：Agent 得到當前環境的 state → 透過此 state 並根據我們目前的 policy 去選擇 action → 使用此 action 與環境互動得到 reward, next\_state → 將蒐集到的 state, next\_state, reward, action 存入 buffer 裡面 → 存 buffer 取出資料計算 Q-value 的預測值以及估計值 → 計算 Loss 以跟新網路參數。

```

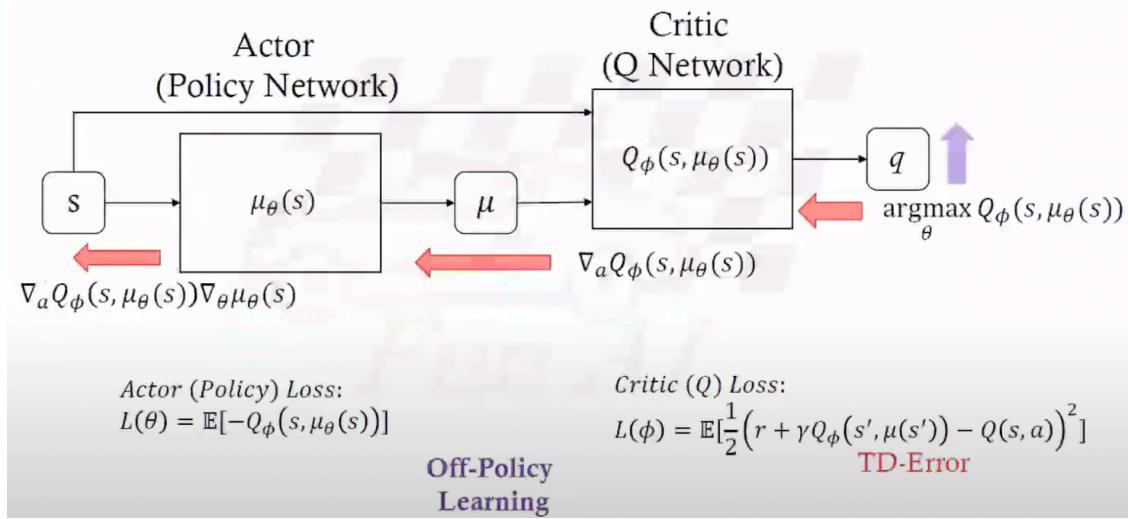
for episode in range(args.episode):
    total_reward = 0
    state = env.reset()
    for t in itertools.count(start=1):
        # select action
        if total_steps < args.warmup:
            action = action_space.sample()
        else:
            action = agent.select_action(state, epsilon, action_space)
            epsilon = max(epsilon * args.eps_decay, args.eps_min)
        # execute action
        next_state, reward, done, _ = env.step(action)
        # store transition
        agent.append(state, action, reward, next_state, done)
        #update behavior network
        if total_steps >= args.warmup:
            agent.update(total_steps)

        state = next_state
        total_reward += reward
        total_steps += 1

```

## Deep Deterministic Policy Gradient

DDPG 是透過同時計算 Q-value 以及 policy 來訓練模型，與 Q-Learning 一樣我們也需要利用神經網路來計算，分別是 critic 以及 actor。一開始我們會拿到初始的 state，Actor Network 會將 state 當成輸入，並根據當前網路內部的參數（也就是當前的 policy）計算出對應的 action 作為網路輸出，而 Critic Network 會同時將 state 以及 actor 輸出的 action 當成網路輸入，根據 critic 網路內部的參數計算出對應的 Q-value。Actor 網路會根據計算出的 Q-value 調整自己的 policy，而 Critic 網路也會根據 TD-error 去更新自身內部的參數，如此維持一個相輔相成的關係。



Actor Network 會根據 state 計算對應的 action，因為 Actor 的輸出直接就是 action，所以網路輸出的維度會是 action 的維度。

而 Critic Network 會根據 state 以及 action 計算對應的 Q-value，因此 Critic 輸出會是一維的數值。

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim,hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()
        self.tanh=nn.Tanh()

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

```

class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, 1),
        )

    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)

```

在訓練階段選擇 action 時，會加入 noise 藉此來達到 exploration 的效果。

```

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

```

選好 action 後我們一樣會拿去跟環境進行互動，得到對應的 next\_state, reward  
並將蒐集到的 state, action 也一併存入 buffer 裡面。

```

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))

```

訓練過程中的更新過程其實跟 DQN 大同小異，最不一樣的點在於 DDPG 會同時計算 action 以及 Q-value · 利用 Q-value 去計算 TD-error 以跟新 critic network · 並將 Q-value 取負號當成 Loss 去跟新 actor network · 畢竟 Q-value 在本質上就是 expected value · 也因為 critic network 會吃 actor 輸出的 action · 所以這邊可以想像成是 actor 希望調整自己的 policy · 讓 critic 輸出的 expected value 能夠越大越好。

而 DDPG 中也會使用到 DQN 中的 target\_network 以及 behavior\_network 技巧 · 使整體訓練更穩定。也是每隔一段時間就會將 behavior\_network 當中的參數 · 直接拷貝進去 target\_network 。

Buffer 的使用方法也是一樣的 · 我們將蒐集到的 state, next\_state, reward, action 存入 buffer 中 · 訓練時隨機 sample 出來以更新網路參數 · 降低資料之間的 dependency 關係 。

```

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 = self._critic_net(state,action)
    with torch.no_grad():
        a_next = self._target_actor_net(next_state)
        q_next = self._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)
    # bp
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()
    ## update actor ##
    # actor loss
    ## TODO ##
    action = self._actor_net(state)
    actor_loss = -self._critic_net(state,action).mean()
    # bp
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()

```

#### 4. Describe differences between your implementation and algorithms.

整體流程都是照著原演算法進行實作，但一開始在訓練的時候，如果網路只使用原本的 hidden\_dim=32 似乎有點太過於簡單，網路參數不夠多，會導致整個網路沒辦法很好的估計 Q-value，訓練了幾個 episode 之後會發現 reward 並沒有獲得很好的改善，於是後來有將 hidden\_dim 加大成(400, 300)，改完之後在前幾輪 episode 就可以發現 reward 獲得很好的進步。

我在想是原本的參數設定太過於簡單，呈現出 underfitting 的情況，模型沒辦法很好的擬合。

並且我發現原本設定的 epsilon 下降的速度太快，原本是每一步選擇 action 後都會進行 epsilon 跟新，所以我將 epsilon-greedy 的參數 epsilon 更改成每一輪 episode 結束後才會進行跟新。

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

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

Formula 1:

$$\nabla_a Q_\phi(s, \mu_\theta(s)) \nabla_\theta \mu_\theta(s)$$

Formula 2:

在跟新 critic 時會計算 Formula 2 中的 gradient Q。

利用 sample 的方式來逼近 expected value，我們將 critic network 的輸出取負號當成 Loss，在 Formula 2 中將 gradient Q 跟 gradient mu 根據 chain rule 執行 back propagation 跟新 actor network。

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

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

因為 critic 的輸出是一個 deterministic 的數值，利用 target\_network 得到的  $Q(s', a')$  以及 behavior\_network 得到的  $Q(s, a)$ ，我們可以透過計算 TD-error 來進行 Back propagation 來跟新 critic network 內部的網路參數。

```

q_value = self._critic_net(state,action)
with torch.no_grad():
    a_next = self._target_actor_net(next_state)
    q_next = self._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)

# bp
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()

```

## 7. Explain effects of the discount factor.

Discounted  
Total Reward  
(Return)

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} \dots + \gamma^{t+n} r_{t+n} + \dots$$
 $\gamma$ : discount factor;  $0 < \gamma < 1$

想想看哪一種給 reward 的方式比較合理 · 一百步以前獲得的 reward 以及上一步所獲得的 reward · 應該是近期獲得的 reward 才是影響這一步的主要原因 · 所以我們多一個 discount factor 是為了讓那些比較遠、比較不相干的 reward 影響的幅度可以降低一點。

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

我們在 RL 當中其實不知道 Transition probability · 我們只能用 sample 的方式去近似 expected value · 所以在一開始資料量還不夠大的情況下 · 計算出的 Q-value 不一定非常的準確 · 如果 agent 一開始就照著 Q-value 的方式去選擇 action · 實際很有可能會沒有辦法探索到最佳的解 · 所以我們在一開始會先讓 agent 能夠去探索各種不同的可能 · 等到一定次數之後才會逐漸相信 Q-value 的值並根據其做選擇。

## 9. Explain the necessity of the target network.

根據 Bellman equation 估計 expected value 以及更新網路參數時 · 我們會需要  $Q(s', a')$  跟  $Q(s, a)$  · 但這個時候一旦我們跟新網路參數 · 將會使得計算出的  $Q(s', a')$  有些微的改變 · 所以我們應該使用一個 target\_network 去存放跟新以前的參

數，使得這個  $Q(s', a')$  可以根據舊的網路參數去計算 Q-value，每隔一段時間再去跟新這個 target\_value 使得整體訓練更穩定。

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

如果 buffer 紿太大，會有太多舊有的資料在裡面，雖然這時 sample 出來的資料彼此不會有太大的相依性，但網路參數會需要更長的時間來達到收斂。如果給的太小，就算我們隨機 sample 資料，sample 出的資料之間相依性還是很高，訓練時容易導致 over-fitting。

11. Experiment results.

DQN:

```
total reward: 271.10
total reward: 229.82
total reward: 234.29
total reward: 318.66
total reward: 261.92
total reward: 239.33
total reward: 307.90
total reward: 280.36
total reward: 154.79
total reward: 250.97
Average Reward 254.9135046503621
```

DDPG:

```
total reward: 264.03
total reward: 257.41
total reward: 251.35
total reward: 297.28
total reward: 280.89
total reward: 307.73
total reward: 309.86
total reward: 281.20
total reward: 266.31
total reward: 262.04
Average Reward 277.8096715810133
```