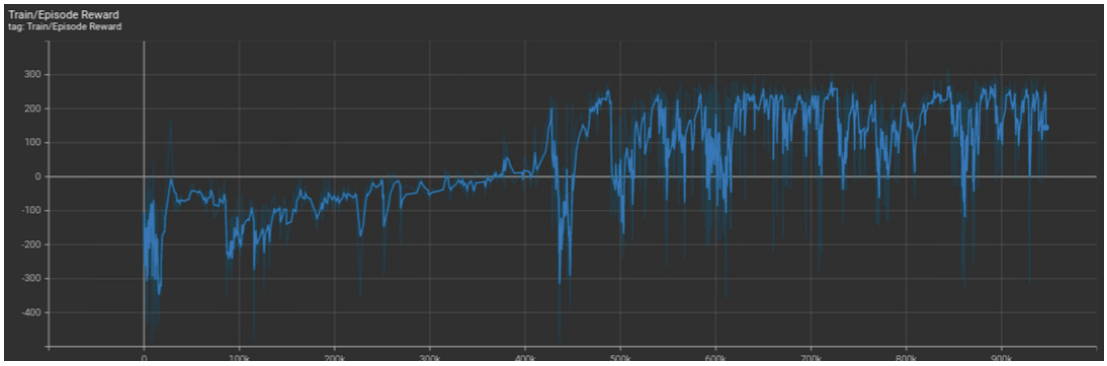
DLP – LAB06: Deep Q-Network and Deep Deterministic Policy Gradient

學號：310611008 姓名：張祐誠

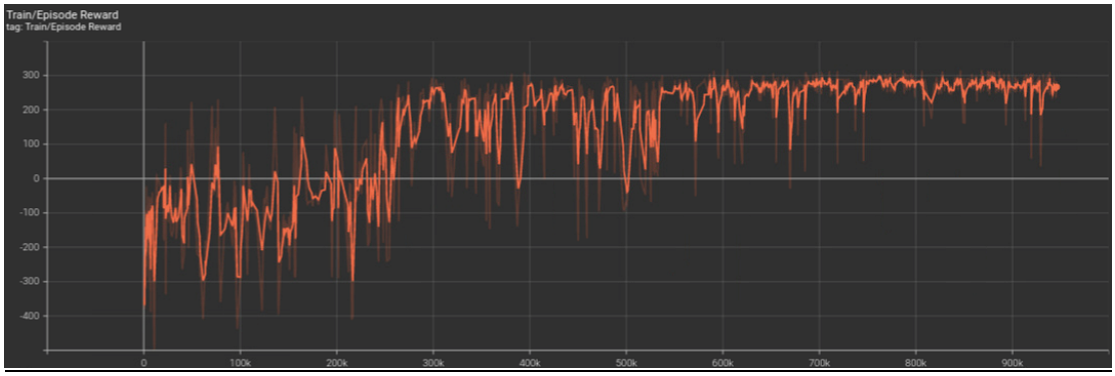
A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2:

Plots of reward trained in 2000 episodes:



A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2:

Plots of reward trained in 2000 episodes:



Describe your major implementation of both algorithms in detail

DQN:

不同於單純的 Q table，這次所用來決定 action 的是由 Network 建立的 Q function，由 8 項可以觀測的數值，output 出 4 個可以進行的動作，本次實驗 Network 由兩個 hidden size = 32 的 fully connected layer 組成，架構如下：

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        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):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        out = self.fc3(x)
        return out
```

而本次 lab 進行動作選擇的方式是使用 epsilon-greedy 的方式，我們希望在每次都選最大 reward 的動作，也就是所謂的 greedy，但也擔心會因此錯過最好的 solution，所以引進了 epsilon 的概念，在 epsilon 的機率下，會隨機進行探索，而在 1- epsilon 的機率下，會選擇最大 reward 的 action，implementation 如下：

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if random.random()<epsilon:
        action = action_space.sample()

    else:
        with torch.no_grad():
            action = torch.argmax(self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)), dim=1).item()

    return action
```

在 update network 的部分所使用的手法，是從 memory 裡面隨機選取多個 transitions，使用 behavior network 推論決定動作後，再使用 target network 計算動作價值，算 mse loss 後再更新。另外 behavior network 4 個 step 更新一次，target network 150 steps 更新一次。implementation 如下：

```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 = self._behavior_net(state).gather(dim=1, index = action.long())
    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), 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()

def _update_target_network(self):
    '''update target network by copying from behavior network'''
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

DDPG:

對於連續的 action，使用 DQN 來解決會變得相當困難，所以第二個環境所使用的是 DDPG。首先我們要先建立 Critic net 跟 Actor net，建構如下：

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        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):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        out = self.tanh(x)
        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)
```

能夠處理連續動作是因為 policy gradient(actor)本身就是一個 normal distribution，然而如果單純使用 policy gradient，因為是屬於回合更新，學習的效率相當緩慢；這個時候 critic 就發揮作用了，她告訴 actor 不需要這麼不確定，繼承了 DQN 的意志，去選擇連續空間中的單一一個動作，這樣的概念也就完成了 DDPG。

在動作選擇上，我們根據 actor network 找尋單一且最佳的動作，詳細來說，actor 會利用 policy gradient 的方法，進行 gradient ascent，由 Critic 来告诉他, 这次的 Gradient ascent 是不是一次正確的 ascent, 如果這次的得分不好, 那就不要 ascent 那麼多。另外我們也會增加高斯雜訊，其功用也是為了要更完整探索所有的解，implementation 如下：

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

如同前面的 DQN，我們的兩 actor 跟 critic 也個別都有兩個 network，其中一個是 eval net，另一個是 target net，這邊個更新概念與 DQN 類似，下面直接將程式碼附上：

```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 = 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)

    # 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()
```

Describe differences between your implementation and algorithms：

第一個是我們 DQN 中 update network 的頻率不是每個 iteration 都更新一次，eval network 是 4 個 iteration 更新一次，而 target network 是 1200 iteration 更新一次，減少 model 過度從比較近的經驗學習，造成 training 不穩定：

```python
def update(self, total_steps):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma)
    if total_steps % self.target_freq == 0:
        self._update_target_network()
```

第二個是我們在 DDPG 還有 DQN 都有做一個 warmup 的動作，在到達 warmup step 前先隨機探索，讓我們開始進行學習更新時，有比較好 prior knowledge :

```python
if total_steps < args.warmup and not args.load_model:
    action = env.action_space.sample()
```

```python
if total_steps >= args.warmup:
    agent.update()
```

Describe your implementation and the gradient of actor updating

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

如同前面題目所提，actor 的更新參數會涉及 critic，也就是透過 critic 得知要怎麼動作才可以獲得最大的 Q value，而算式後面的部分則是在告訴我們說，這個 actor 要怎麼改才能做這個動作，所以可以理解為 actor 要朝 Q 期望值最大的方向移動。Implementation 如下：

```
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
```

Describe your implementation and the gradient of critic updating

Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

這邊可以看到了 DQN 的精神，由 Q target 下一狀態，用 actor target 選擇下一個動作，然後由 TD learning 的方式，計算並 minimize 這一項的 mse loss，Implementation 如下：

```
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)
```

Explain effects of the discount factor

Discount factor = 0 是僅考慮下一步 reward 進行更新，這點與蒙地卡羅算法（TD(1)）是相對應的，蒙地卡羅是將整個 episode 做完之後在評估與更新，所以這一項是我們在學習時，考慮當前可以立刻得到的 reward 與未來一段期間內可以得到的 reward 的一個權重。

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

如同前面講述 epsilon 的概念，在 epsilon 的機率下，會隨機進行探索，而在 1- epsilon 的機率下，會選擇最大 reward 的最為選擇的 action；如果只做 greedy 的話會有可能造成一些動作沒被探索到，例如一開始就只會往左等等的。

Explain the necessity of the target network

從結論上來說，就是要讓 model 在 training 上更穩定。如果我們每個 iteration 都對 target network 做一次更新，那我們每次得到的 network 就會很接近，而且會頻繁更新不穩定，甚至於高估我們得到的 value，因次我們需要一個長時間固定的 target network，即 $r_t+Q\pi(s_{t+1},\pi(s_{t+1}))$是固定的，打斷了時間的關聯性，讓我們可以輕易的做 fitting 的問題。

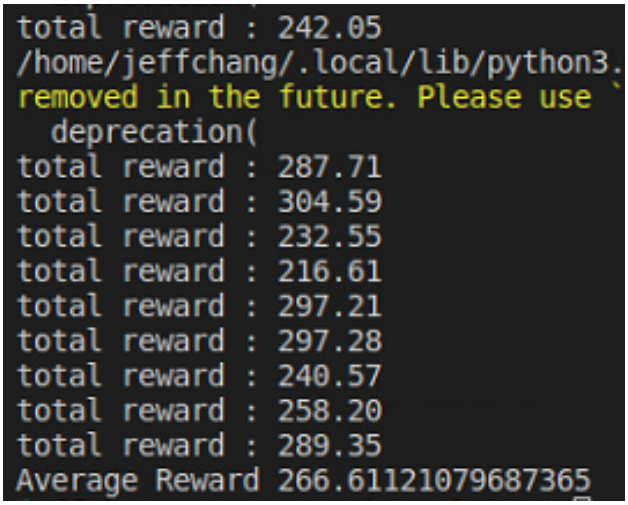
Explain the effect of replay buffer size in case of too large or too small

我們會從 buffer 裡面抽取資料來訓練，希望透過這樣可以減少與環境互動的互動次數，並盡量讓 batch 的資料盡可能離散。所以當我們的 buffer size 越小時，越容易發生訓練過程不穩定的狀況；但當 buffer size 太大的話，雖然 model 似乎會比較穩定，但更可能使學習上效率很慢。

Performance:

DQN:

(average Reward: 266.61)



DDPG:

(average Reward: 280.04)

```
Start Testing
/home/jeffchang/.local/lib/python3.
removed in the future. Please use `
  deprecation(
total reward: 255.71
total reward: 286.13
total reward: 305.20
total reward: 274.47
total reward: 310.24
total reward: 299.11
total reward: 262.96
total reward: 303.63
total reward: 254.50
total reward: 248.51
Average Reward 280.0466135904348
```

Implement and experiment on Double-DQN

DDQN 是由 DQN 變化而來，我們在 DQN 計算目標 Q_value 所使用的是所找到最大的 Q 值，正如同我們前面提到的，這個值往往是被高估的，並不是最好的 Q 值；DDQN 在這個時候對這方面進行改進，一樣先使用 main net 找到動作，再從 target net 找到這個 action 真正應該要有的 Q 值，以選取到真正最好的 action。

```
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():
        action_index = torch.argmax(self._behavior_net(next_state),dim = 1).view(-1,1)
        q_next = self._target_net(next_state).gather(dim=1, index = action_index.long())
        q_target = reward+gamma*q_next*(1-done)

    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

Performance:

```
Start Testing
/home/jeffchang/.local/lib/python3.8/s
removed in the future. Please use `env
  deprecation(
total reward : 261.62
total reward : 283.13
total reward : 276.93
total reward : 147.71
total reward : 154.10
total reward : 288.47
total reward : 287.70
total reward : 151.96
total reward : 279.08
total reward : 278.98
Average Reward 240.96921663680845
```

Implement and experiment on TD3

TD3 的手法是將兩個 critic net 相互作用，這邊一樣要解決 Q_value over estimate 的問題，而在原始論文中有提到說，actor critic 利用 DDQN 的概念來避開最大 Q value 造成的 over estimation 是比較沒效果的，因為本身更新速度比較緩慢。所以作者的想法就是利用兩個不同的 critic net 來估計，然後找算出來比較小的 Q value 作為我們的 target value。作者同時也提到，DDPG 本身在 hyperparameter 的調整上比較脆弱一點，td3 會給 target action clip noise，避開一些不正倔 target action 的 peak。

```
        h3, h4 = hidden_dim
        self.critic_head2 = nn.Sequential(
            nn.Linear(state_dim + action_dim, h3),
            nn.ReLU(),
        )
        self.critic2 = nn.Sequential(
            nn.Linear(h3, h4),
            nn.ReLU(),
            nn.Linear(h4, 1),
        )

    def forward(self, x, action):
        x1 = self.critic_head1(torch.cat([x, action], dim=1))
        x2 = self.critic_head2(torch.cat([x, action], dim=1))

        return self.critic1(x1), self.critic2(x2)
```

```
with torch.no_grad():
    noise = torch.FloatTensor(self._policy_noise.sample()).to(self.device)
    noise = noise.clamp(-self.noise_clip, self.noise_clip)
    a_next = (target_actor_net(next_state)+noise).clamp(-self.max_action, self.max_action)

    target_Q1, target_Q2 = target_critic_net(next_state, a_next)
    target_Q = torch.min(target_Q1, target_Q2)
    q_target = reward + gamma*target_Q*(1 - done)

current_Q1, current_Q2 = critic_net(state, action)
criterion = nn.MSELoss()
critic_loss = criterion(current_Q1, q_target) + criterion(current_Q2, q_target)
```

Performance:

```
/home/jeffchang/.local/lib/python3.8
removed in the future. Please use `e
  deprecation(
total reward: 262.19
total reward: 291.52
total reward: 320.80
total reward: 292.46
total reward: 320.30
total reward: 322.78
total reward: 285.84
total reward: 306.56
total reward: 267.83
total reward: 286.89
Average Reward 295.7178736219879
```

Appendix: EWMA reward curve of DQN/DDPG/DDQN/TD3