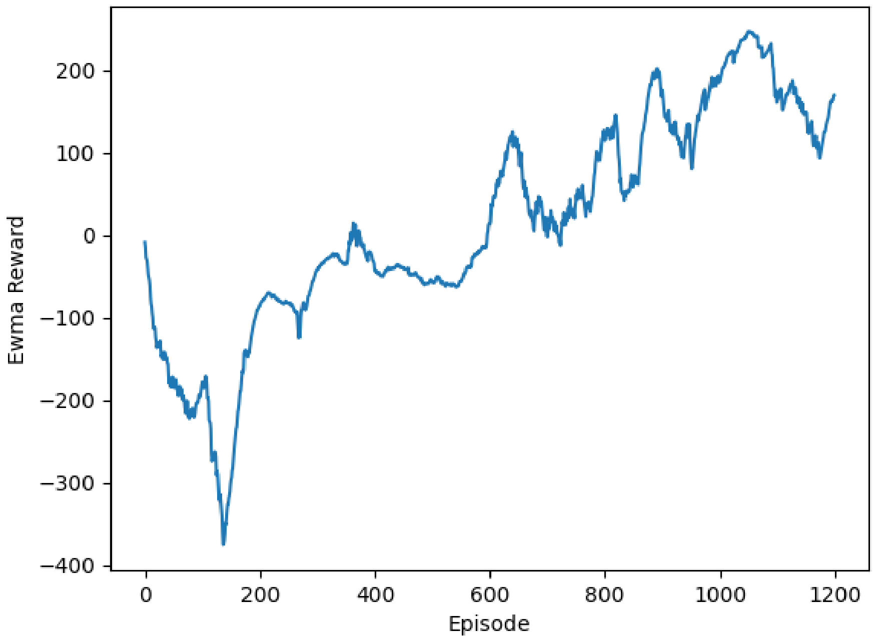
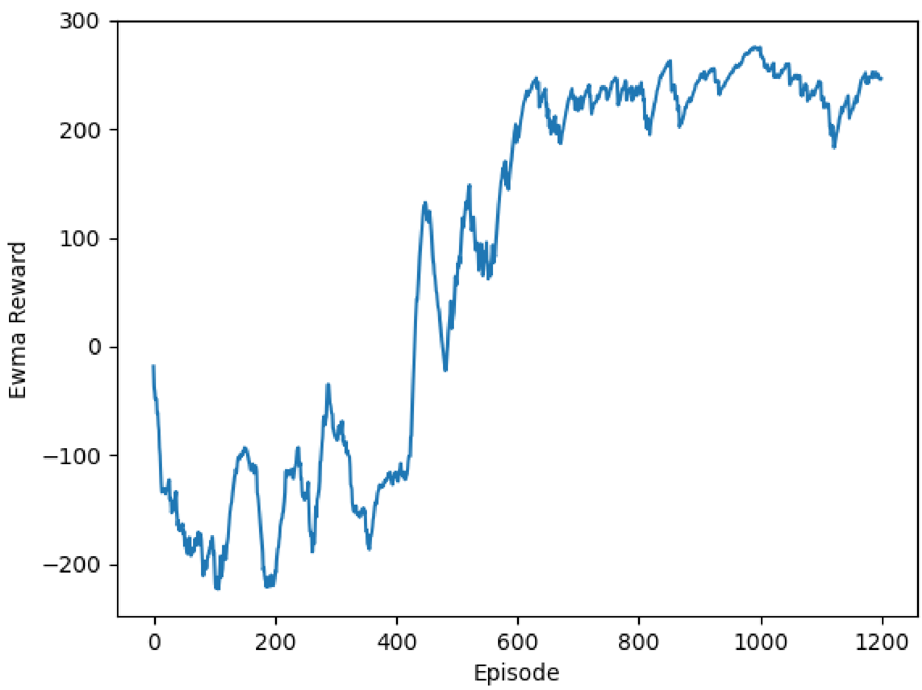
GLP HW6

309553008

- episode rewards of training episodes in LunarLander-v2



- episode rewards of training episodes in LunarLanderContinuous-v2



- Describe your major implementation of both algorithms in detail

  DQN:

  根據 TA 給的架構，這是一個預測當下 state 做出何種 action 是最適合的 Net，所以 input 是含有 8 個資訊的 Observation(state)，output 是 4 種的 action (No-op, Fire left engine, Fire main engine,　Fire right engine)

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.action = nn.Sequential(
                    nn.Linear(state_dim, hidden_dim),
                    nn.ReLU(inplace=True),
                    nn.Linear(hidden_dim, hidden_dim),
                    nn.ReLU(inplace=True),
                    nn.Linear(hidden_dim, action_dim),
                )
        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        return self.action(x)
        # raise NotImplementedError
```

在選擇 action 的部分有給定一個 epsilon 的閥值當作機率，所以 epsilon 的機率 random 的選擇 action；反之選擇預測機率最大的 action。

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if np.random.random() > epsilon:
        return self._behavior_net(torch.from_numpy(state).view(1, -1).to(self.device)).max(dim=1)[1].item()
    else:
        return action_space.sample()
    # raise NotImplementedError
```

Update network 的方法是先從 memory 中抽出 batch size 數量的(state, action, reward, next state)資料與是否成功降落(done=True)來做 TD-Learning，再用 Q(s, a)與 r + gamma*max(Q’(s’, a’))的差做 MSELoss

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

```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)

    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)
        q_target = reward + gamma*q_next*(1-done)

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

    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
```

因為 DQN 的核心概念就是有兩個初始狀態一樣的 Net，然後更新時間不一樣，weight 因時間差更新所有不同，較常更新的是 Q 估計(behavior)，另一個則是 Q 現實(target)，以此作業給的參數是 Q -behavior 每 4 個動作更新一次，Q-target 則是 1000 次更新一次，藉由 Fixed Q-target 可以打亂每一次 action 的關聯性(前面 sample memory 也有此意)，已提升預測 Q-behavior 的 action。

```python
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())
    # raise NotImplementedError
```

DDPG:

相較於 DQN 更適合用來處理連續的資料

與 DQN 差不多，都是預測當下 state 做出何種 action 是最適合的 Net，不過因為 LunarLanderContinuous-v2 的 action 是兩個(Main engine, Left-right)，所以最後一層是 2 維。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.action = nn.Sequential(
                        nn.Linear(state_dim, hidden_dim[0]),
                        nn.ReLU(inplace=True),
                        nn.Linear(hidden_dim[0], hidden_dim[1]),
                        nn.ReLU(inplace=True),
                        nn.Linear(hidden_dim[1], action_dim),
                        nn.Tanh()
                    )
        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        return self.action(x)
        # raise NotImplementedError
```

建立一個估計 Q(s, a)的 Net。

```python
class CriticNet(nn.Module): # output Q(a, s) -> evaluate value
    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)
```

把當前 state 加上 noise，餵進 actor net，決定下一個 action。

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            action = 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:
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))

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

跟 DQN 差不多，只是不是用 Q -target action 的 value 來估算了，取而代之的是用 CriticNet 將 Q- behavior 與 Q-target 所 output 出來的 action 估值之後再帶進 MSELoss。

```python
q_value = self._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)
```

我們要使 Q(s, a)輸出越大越好，所以 Loss 定義為 E[-Q(s, action(s))]

```python
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()
```

- Describe differences between your implementation and algorithms.

  在 training 的時候，一開始會有 warmup，在 warmup(10000)個步驟前都不會 update Net，只會存進 Memory，用於以後做 sample，而 DQN 跟 DDPG 更新 weight 的時機不一樣，DDPG 每一次 iteration 都會更新一次，DQN 則跟之前講得一樣 4/1000 更新一次。

- Describe your implementation and the gradient of actor updating

  我們要使 Q(s, a)輸出越大越好，所以 Loss 定義為 E[-Q(s, action(s))]

```
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()

actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

- Describe your implementation and the gradient of critic updating.

  根據

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

  將 Q- behavior 與 Q-target 的 Q(s, a)做 MSELoss

```
q_value = self._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(gamma)

  以 DDPG 為例:

  下一步的 Q(s’, a’)會乘上 discount factor(gamma)，也就是說越是下一步(與當下隔越遠)的權重越低，影響力越小。

```
q_value = self._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 benefits of epsilon-greedy in comparison to greedy action selection.
  如果單純地用 greedy，我們很可能只能得到部分最佳解，所以為了跳脫區域，我們需要用 smaple()改變方向(區域)。

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if np.random.random() > epsilon:
        return self._behavior_net(torch.from_numpy(state).view(1, -1).to(self.device)).max(dim=1)[1].item()
    else:
        return action_space.sample()
```

- Explain the necessity of the target network.
  與前面有提到的一樣，因為兩邊 weight 不一樣，所以可以藉由 Fixed Q-target 打亂每一次 action 的關聯性，藉此來提升整個 Network 的表現、使 training 更加穩定。
- Explain the effect of replay buffer size in case of too large or too small.
  若 buffer size 太大會使 training 更穩定收斂，只是會降低 training 的速度；太小則會每次 sample 都是最近的狀況，容易造成 overfitting 或導致 train 出來的 model 表現差強人意。
- Implement and experiment on Double-DQN
  DDQN 跟 DQN 基本上是完全一樣的，只差在 update behavior network 的部分，DDQN 在決定 Q-target 的時候不是直接拿 Q'(s, a)。而是用 Q(s, a)最大值當作 index 去找 Q'(s, a)。
  DQN:

```python
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)
    q_target = reward + gamma*q_next*(1-done)
```

  DDQN:

```python
q_value = self._behavior_net(state).gather(dim=1,index=action.long())

with torch.no_grad():
    idx = self._behavior_net(next_state).max(dim=1)[1].view(-1, 1)
    q_next = self._target_net(next_state).gather(dim=1,index=idx.long())
    q_target = reward + gamma*q_next*(1-done)
```

- Performance

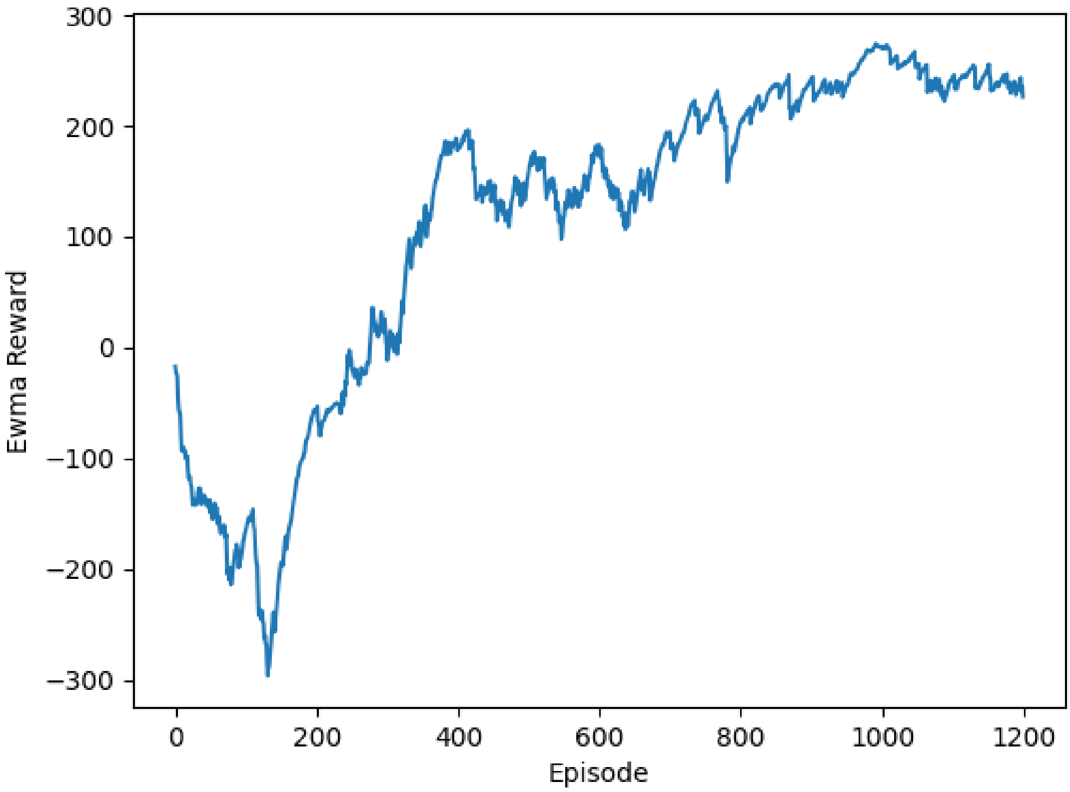
DQN

```
Average Reward 207.8983165751878
```

DDPG

```
Average Reward 281.4128822053839
```

DDQN:




```
Average Reward 211.5208737695606
```

- 参考:

https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/intro-DQN/

https://wanjun0511.github.io/2017/11/19/DDPG/