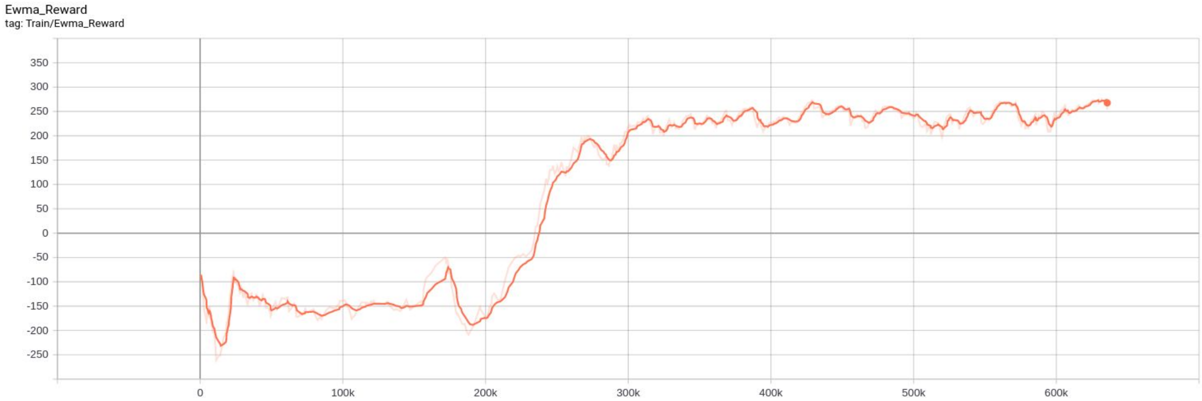
Lab8

資工碩一 吳承翰 0856105

1. LunarLander-v2 (action離散) episode rewards

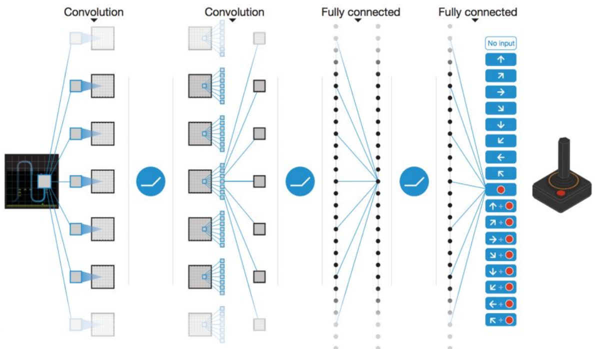
2. LunarLanderContinuous-v2 (action連續) episode rewards



3. Describe your major implementation of both algorithms in detail

DQN:

建立一個network來預測Q(s,a)的value，這裡的action4有種可能(No-op,Fire left engine,Fire main engine,Fire right engine)，所以網路最後一層為4個neuron



```python
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 ##
        out=self.relu(self.fc1(x))
        out=self.relu(self.fc2(out))
        out=self.fc3(out)
        return out
```

在episode中(玩遊戲的過程中)，選擇最大Q(s,a_i)的a_i或者有一定的機率ε隨機選擇action (called ε-greedy)

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:  # explore
        return action_space.sample()
    else: # exploit
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()
```

update network的方法是由replay memory中sampling一些遊戲的過程: (state,action,reward,next_state,done)來做td-learning，再用Q(s,a)與r + gamma * $\max_{a'}$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)
    ## 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)
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

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

每隔一段時間，就用behavior network取代target network

```
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:

建立一個可以依據目前state決定要執行哪個action的Actor Network, 由於action有2個(Main engine:-1~+1,Left-right-engine:-1~+1)，所以最後一層有2個neuron

```
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
```

建立一個可以預估Q(s,a)的Critic Network，由於輸出的是一個純量，所以最後一層neuron數為1

```
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episode中(玩遊戲的過程中)， 由Actor Network選擇action並加上一noise

```python
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()
```

在episode中(玩遊戲的過程中)， 也要更新Behavior的Actor Network μ, Critic Network Q, Target的Actor Network μ’, Critic Network Q’。
利用Target Network生出的q_target與Behavior Network生出的q_value做MSELoss更新Q。

```python
# 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()
```

利用Behavior Network的Actor Network μ與Critic Network Q可以求出Q(s,a)， 我們想要更新μ來使輸出的Q(s,a)越大越好， 因此定義Loss Value = E[-Q(s, μ(s))]， 並透過backpropagation更新。

```python
## 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

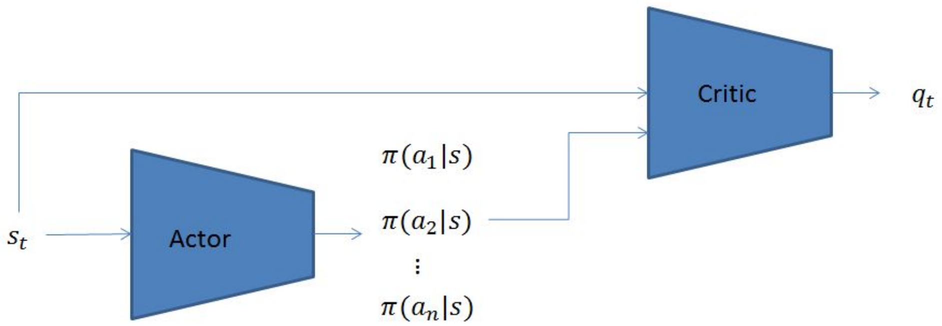
在training的時候，一開始會有一段warmup的時間，在這段時間中，不會去update network的參數，只會隨便亂玩(隨機選擇action)，並把遊戲過儲存到replay memory裡。另外在DQN的部份，並不是每個iteration都要更新Behavior Network，而是每隔一段時間(Ex: 4個iteration)才會更新一次。

5. Describe your implementation and the gradient of actor updating in DDPG

利用Behavior Network的Actor Network μ與Critic Network Q可以求出Q(s,a)，我們想更新Actor Network μ來使輸出的Q(s,a)越大越好，因此定義Loss Value = -Q(s, μ(s))，backpropagation的時候不更新Critic、只更新Actor。

$$L = -Q(s, a|\theta_Q),\ a = u(s|\theta_u)$$

$$\begin{aligned}\frac{\nabla L}{\nabla \theta_u} &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla a}\frac{\nabla a}{\nabla u(s|\theta_u)}\frac{\nabla u(s|\theta_u)}{\nabla \theta_u} \\ &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla u(s|\theta_u)}\frac{\nabla u(s|\theta_u)}{\nabla \theta_u}\end{aligned}$$




```
## 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()
```

6. Describe your implementation and the gradient of critic updating in DDPG

利用Target Network生出的Q_{target}與Behavior Network生出的Q(s,a)做mean square error來更新Q Network。

$$L = \frac{1}{N}\sum(Q_{target} - Q(s_t, a_t|\theta_Q))^2$$

```python
# 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()
```

7. Explain effects of the discount factor

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

λ就是discount factor，意思就是說，越是未來所給的reward影響是越來越小的，當下的reward是最大的。

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

我們必須在explore與exploit之間取得平衡，因此在greedily choosing action的基礎上，必須偶爾選擇其他的action來explore那些未知但可能是最佳的action。

9. Explain the necessity of the target network

有Target Network與Behavior Network的搭配可以使training的時候更穩定， 因為生出Q_target的Target Network每隔一段時間才會改變一次。

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

如果replay buffer size越大，training過程可以更穩定，但會降低training的速度。如果replay buffer size越小， 會一直著重於最近玩的episode的狀況， 容易造成overfitting、甚至整個train壞掉。

11. Implement and experiment on Double-DQN

DDQN與DQN其實差不多， 就只差在update Behavior Network時是如何決定q_target的，DDQN在決定q_target時， 不是直接取max $Q'(s,a_i)$， 而是用$Q(s,a_i)$中最大值的i作為t查找$Q'(s,a_i)$的index。

```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():
        action_index=self._behavior_net(next_state).max(dim=1)[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)

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

train出來的結果跟DQN差不多。(train 2000個episode)

```
total reward: 243.89
total reward: 264.59
total reward: 279.63
total reward: 270.68
total reward: 277.46
total reward: 268.63
total reward: 272.15
total reward: 277.39
total reward: 295.00
total reward: 295.44
Average Reward 274.48658371688487
```

12. Performance

DQN:

共train 2000個episode

```
total reward: 244.99
total reward: 285.70
total reward: 271.34
total reward: 270.52
total reward: 313.91
total reward: 268.36
total reward: 307.92
total reward: 298.52
total reward: 311.83
total reward: 297.35
Average Reward 287.0437232289645
```

DDPG:

共train 2000個episode

```
total reward: 240.84
total reward: 296.34
total reward: 249.44
total reward: 285.31
total reward: 288.56
total reward: 259.16
total reward: 311.15
total reward: 304.35
total reward: 303.36
total reward: 290.88
Average Reward 282.94019903086956
```