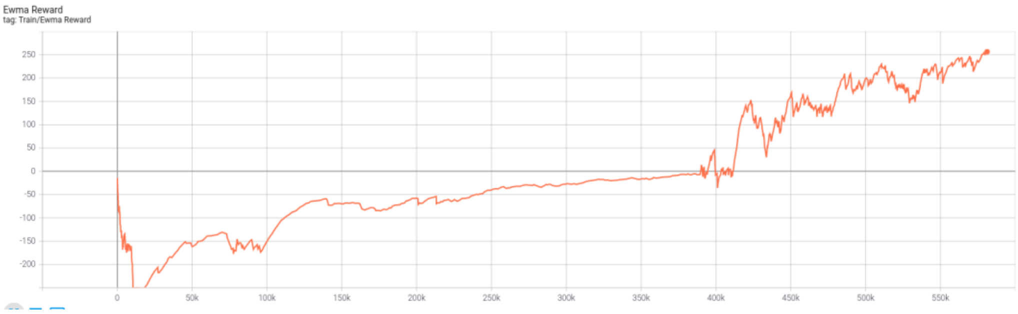
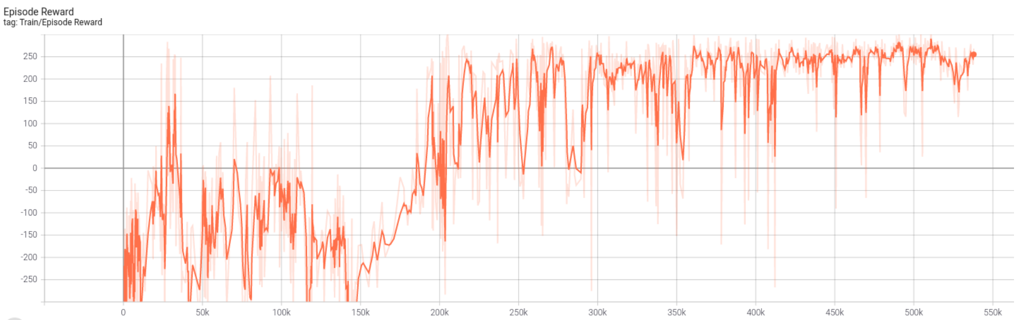
LAB6 Deep Q-Network and Deep Deterministic Policy Gradient

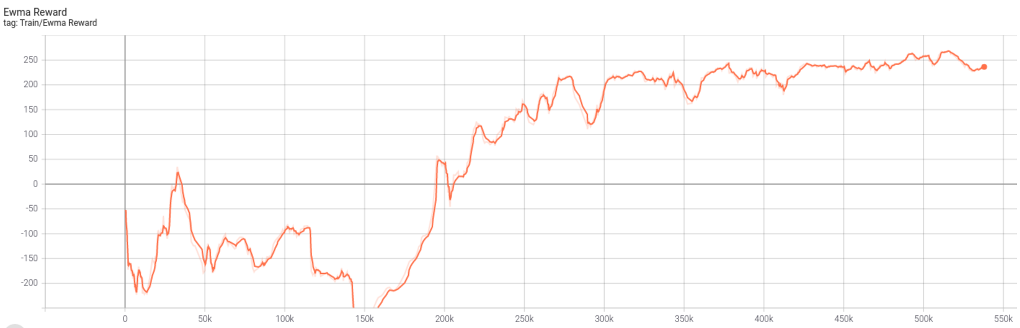
1. Tensorboard

   LunarLander-v2 (DQN)



   LunarLanderContinuous-v2 (DDPG)





2. Implement detail

1. Describe your major implementation of both algorithms in detail. (TODO)

DQN :

Create a network to generate distribution of four action .

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(8, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 32)
        self.out = nn.Linear(32, 4)

    def forward(self, x):
        ## TODO ##
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        action_distribution = self.out(x)

        return action_distribution
```

Set the optimizer and loss function .

```python
self._optimizer = torch.optim.Adam(self._behavior_net.parameters(),lr = 0.0005)
self.criterion = nn.MSELoss()
```

When training , use the action of the highest possibility Q(S, a_i) or randomly choose action with the epsilon .

With probability ε select a random action a_t
otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$

```python
    def select_action(self, state, epsilon, action_space):
        '''epsilon-greedy based on behavior network'''
        x = torch.unsqueeze(torch.FloatTensor(state),0).cuda()
        if random.random() < epsilon:
            action_ = action_space.sample()
        else:
            with torch.no_grad():
                action_value = self._behavior_net(x)
                action_ = torch.argmax(action_value).cpu().numpy()

        return action_
```

Update the behavior network with the algorithm :

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

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

Every C steps reset $\hat{Q} = Q$

End For

End For

Deep Learning and Practice, Spring 2021,

```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)

        self._optimizer.zero_grad()
        q_value = self._behavior_net(state).gather(1, action.long())
        with torch.no_grad():
            q_next = self._target_net(next_state)
            q_target = reward + (1-done)*gamma*torch.max(q_next, dim=1)[0].view(self.batch_size, 1)

        loss = self.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()
```

Update the target network with behavior network every 4 steps .

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

My DQN training parameters are using default .

DDPG :

Create a actor network to generate value of “Main engine” and “Left-Right engine” , so the num of the output is two .

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 = hidden_dim
        self.fc1 = nn.Linear(state_dim, h1)
        self.fc2 = nn.Linear(h1, h2)
        self.out = nn.Linear(h2, 2)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        action_value = torch.tanh(self.out(x))

        return action_value
```

Set the optimizer and loss function .

```
        self._actor_opt = torch.optim.Adam(self._actor_net.parameters(), lr=args.lra)
        self._critic_opt = torch.optim.Adam(self._critic_net.parameters(), lr=args.lrc)
        self.criterion = nn.MSELoss()
```

When training , generate action values and add the noise .

Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise

```
    def select_action(self, state, noise=True):
        '''based on the behavior (actor) network and exploration noise'''
        ## TODO ##
        with torch.no_grad():
            x = torch.unsqueeze(torch.FloatTensor(state),0).cuda()
            action = self._actor_net(x)
            action = action.detach().cpu().numpy()[0]
            if noise:
                action = action + self._action_noise.sample()

        return action
```

Update the behavior network with the algorithm :

Sample random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

Set $y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$

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

Update the actor policy using the sampled gradient:

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

```
    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)

        critic_loss = self.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 = (-1.0)*critic_net(state, action).mean()
        # optimize actor
        actor_net.zero_grad()
        critic_net.zero_grad()
        actor_loss.backward()
        actor_opt.step()
```

Update the target networks .

Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{\mu'}$$

```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_(tau*behavior.data + (1.0 -tau)*target.data)
```

2. Describe differences between your implementation and algorithms.

When start training , in order to have enough sample for training , we don't update the network within warmup step , just randomly choose action to play the game and store the result in the replay memory . And in the DQN , we update the target network every few iteration , it can reduce the correlation between target and behavior network .

3. Describe your implementation and the gradient of actor updating.

```python
            action = actor_net(state)
            actor_loss = (-1.0)*critic_net(state, action).mean()
            # optimize actor
            actor_net.zero_grad()
            critic_net.zero_grad()
            actor_loss.backward()
            actor_opt.step()
```

for actor :

expect large $Q(s,a)$,

so make $J(\theta^\mu) = E[Q(s,a)]$

$$\Rightarrow \nabla_{\theta^\mu} J(\theta^\mu) = E\left[\frac{\partial Q(s,a|\theta^Q)}{\partial \theta^\mu}\right], \text{ with } a = \mu(s|\theta^\mu)$$

$$= E\left[\frac{\partial Q(s,a|\theta^Q)}{\partial a} \cdot \frac{\mu(s|\theta^\mu)}{\partial \theta^\mu}\right]$$

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

So we can define loss function as :

$$\text{Actor loss} = -\nabla_{\theta^\mu} J(\theta^\mu)$$

We only update the actor network .

4. **Describe your implementation and the gradient of critic updating.**

   Compute the MSELoss with Q_target from target network and Q_value from behavior network .

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

```python
        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)

        critic_loss = self.criterion(q_value, q_target)
        # optimize critic
        actor_net.zero_grad()
        critic_net.zero_grad()
        critic_loss.backward()
        critic_opt.step()
```

5. **Explain effects of the discount factor.**

   As the training step is longer , the sum of the reward will become infinity , so the discount factor can reduce the reward in the later step , in other words , it reduce the correlation of the later step . In this lab , only using the one next step , but we still give the 0.99 as discount factor .

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

   The epsilon-greedy make additional opportunity to select the best action , because the action selected by the network may not be the best selection , and with the episode growing , we increase the epsilon . In conclusion , it can improve the exploratory of the network in the early step .

7. **Explain the necessity of the target network.**

   We use the periodically update network (target network) to improve the stability of training .

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

   The higher replay buffer size can make training more stable but need more time to train . And if too small , it is easier to occur overfitting .

3. Bonus

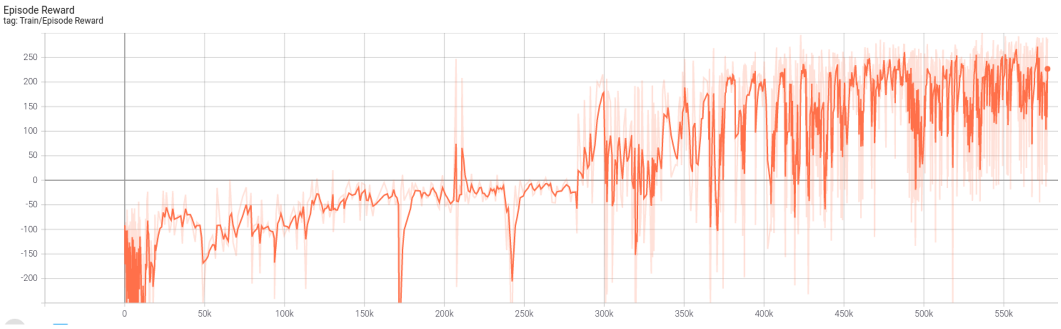
DDQN

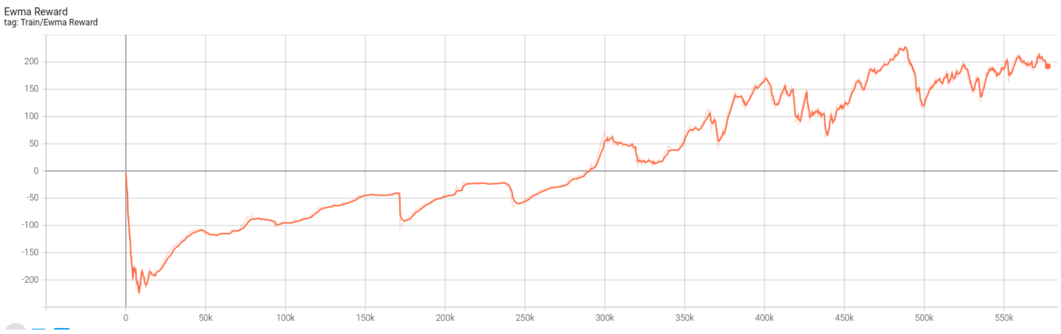
The difference between DQN and DDQN is that in DDQN case , when compute the q_target , it don't use the max of $Q'(s_i, a_i)$, but use the index of the max value in $Q(s_i, a_i)$ as the index of $Q'(s_i, a_i)$, that mean in DDQN , the action selection and evaluation are generate from the different function .

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

loss = self.criterion(q_value, q_target)
```

Tensorboard :





4. Performance (Average reward of 10 games)

DQN (train 1200 episode) :

```
Average Reward 274.02778984625076
```

DDPG (train 1500 episode) :

```
Average Reward 230.17661800337765
```

DDQN (train 1200 episode) :

```
Average Reward 240.21417588222707
```