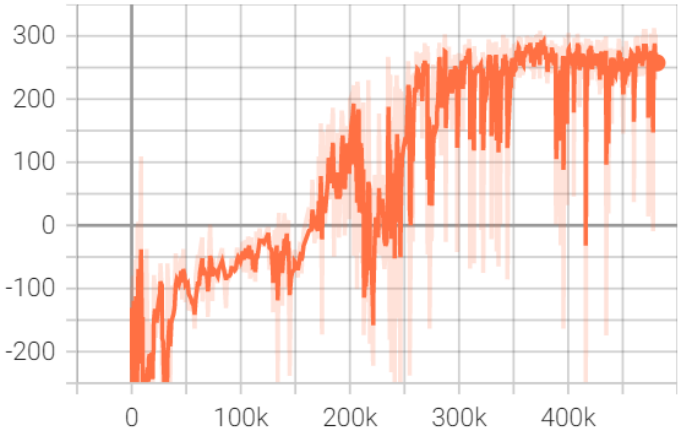
Deep Learning and Practice Lab 6

309554005 黃睿宇

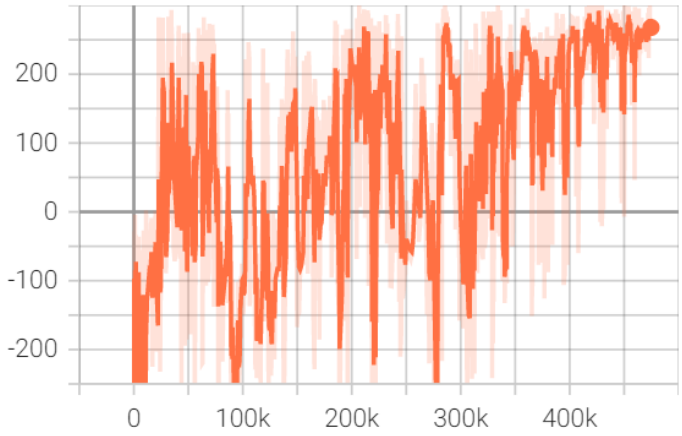
1 Introduction

In this lab, we implement two deep reinforcement algorithms by completing the following two tasks: (1) solve LunarLander-v2 by using deep Q-network (DQN), and (2) solve LunarLanderContinuous-v2 by using deep deterministic policy gradient (DDPG).

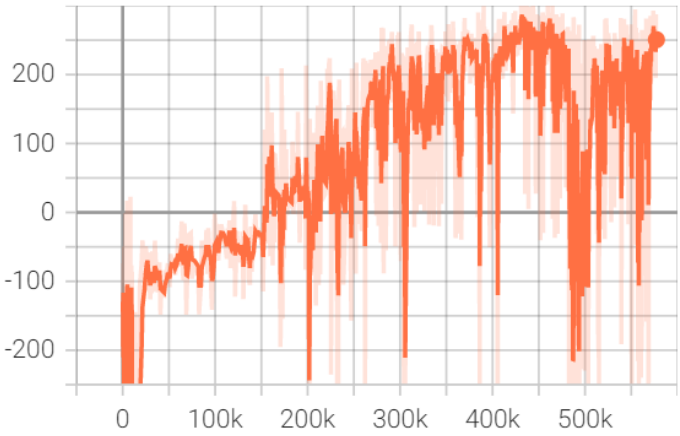
2 Tensorboard of Episode Rewards of DQN in LunarLander-v2



3 Tensorboard of Episode Rewards of DDPG in LunarLanderContinuous-v2



4 Tensorboard of Episode Rewards of DDQN in LunarLander-v2



5 Average reward of DQN in LunarLander-v2

```
Start Testing
ep 0 | Total reward: 254.11769136420787
ep 1 | Total reward: 291.159052434358
ep 2 | Total reward: 284.38176793670044
ep 3 | Total reward: 285.60449093698014
ep 4 | Total reward: 316.92826214159004
ep 5 | Total reward: 267.06623625417393
ep 6 | Total reward: 305.5958333777045
ep 7 | Total reward: 306.06670513861945
ep 8 | Total reward: 314.93549830316783
ep 9 | Total reward: 295.0337411417647
Average Reward 292.0889279029267
```

6 Average reward of DDPG in LunarLanderContinuous-v2

```
Start Testing
ep 0 | Total reward: 252.3465097408421
ep 1 | Total reward: 291.3438386161554
ep 2 | Total reward: 236.86147013224814
ep 3 | Total reward: 276.8793063835286
ep 4 | Total reward: 291.286585602035
ep 5 | Total reward: 225.14790863053497
ep 6 | Total reward: 275.0930705985057
ep 7 | Total reward: 293.3147359563233
ep 8 | Total reward: 293.94107724068215
ep 9 | Total reward: 255.41956738751688
Average Reward 269.16340702883724
```

7 Average reward of DDQN in LunarLander-v2

```
Start Testing
ep 0 | Total reward: 250.53281199340233
ep 1 | Total reward: 235.42573113935833
ep 2 | Total reward: 277.0653826809082
ep 3 | Total reward: 257.353910306314
ep 4 | Total reward: 312.7955523422477
ep 5 | Total reward: 230.36167463110667
ep 6 | Total reward: 295.91171876159046
ep 7 | Total reward: 290.5370413488753
ep 8 | Total reward: 298.6244118077086
ep 9 | Total reward: 196.3874378243332
Average Reward 264.49956728358444
```

8 Implement of DQN

8.1 Network

The Network is composed of three fully connected layers.

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(64, 64)):
        super().__init__()
        self.l1 = nn.Linear(state_dim, hidden_dim[0])
        self.l2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.l3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        a = self.relu(self.l1(x))
        a = self.relu(self.l2(a))
        a = self.l3(a)
        return a
```

8.2 Action

We select action by using ϵ-greedy algorithm. The agent will choose the action with the highest Q-value with probability $1-\epsilon$ or explore a random action with probability ϵ. The value of ϵ decreases when episode increases.

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

8.3 Behavior network

We update the behavior network by setting

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

```
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 + (1 - done) * gamma * q_next
    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()
```

8.4 Target network

We update target network by copying from behavior network.

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

9 Implement of DDPG

9.1 Actor network

The Network is composed of three fully connected layers and output with hyperbolic tangent function.

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        self.l1 = nn.Linear(state_dim, hidden_dim[0])
        self.l2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.l3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        a = self.relu(self.l1(x))
        a = self.relu(self.l2(a))
        a = self.tanh(self.l3(a))
        return a
```

9.2 Action

We add a gaussian noise to the predicted action.

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    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()
```

9.3 Behavior network

We update the behavior network via updating critic by minimizing the loss and updating actor policy by using the sampled gradient. The loss of actor is $-Q(s, a)$.

```
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
    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 + (1 - done) * gamma * q_next
    criterion = nn.MSELoss()
    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
    action = self._actor_net(state)
    actor_loss = self._critic_net(state, action).mean()
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

9.4 Target network

We update target network by soft copying from behavior network.

```
def _update_target_network(targrt_net, net, tau):
    '''update target network by _soft_ copying from behavior network'''
    for target, behavior in zip(
        target_net.parameters(), net.parameters()):
        target.data.copy_((1 - tau) * target.data + tau * behavior.data)
```

10 Implement of DDQN

The only difference between DQN and DDQN is in finding the Q-value. Since DQN have overestimating problem, DDQN selects the action from the behavior network instead of target network. The target Q-value is selected as

$$y_t = r_{t+1} + \gamma Q\left(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(s_{t+1}, a|\theta)\,|\theta^-\right)$$

```
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():
        # double DQN
        action_idx = self._behavior_net(next_state).max(dim=1)[1]
            .view(-1, 1)
        q_next = self._target_net(next_state).gather(dim=1),
            index=action.long())
        q_target = reward + (1 - done) * gamma * q_next
    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()
```

11 Differences Between Implementation and Algorithms

Basically, the implementation is similar to the algorithms. The difference is that the gradient is clipped when we update the network parameters.

12 Gradient of Actor Updating

We update actor policy by using the sampled gradient

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

```
# actor loss
action = self._actor_net(state)
actor_loss = self._critic_net(state, action).mean()
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

13 Gradient of Critic Updating

We set

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

and update critic by minimizing the loss

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

```
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
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 + (1 - done) * gamma * q_next
criterion = nn.MSELoss()
loss = criterion(q_value, q_target)
# optimize critic
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

14 Effects of Discount Factor

The discount factor determines how much the agents cares about rewards in the distant future relative to those in the immediate future. If $\gamma = 0$, the agent will only learn about actions that produce an immediate reward. If $\gamma = 1$, the agent will evaluate its actions based on the future rewards.

15 Benefits of Epsilon-greedy

The epsilon-greedy approach will select the action with the highest estimated reward most of the time. With a small probability of ϵ, agent chooses to explore, i.e., the action is selected randomly. Thus, if we make infinite trials, each action is taken an infinite number of times. Hence, the epsilon-greedy action selection policy will discover the optimal actions for sure.

In comparison with greedy action selection, there isn't exploration in greedy action selection. Therefore, some better actions may not have the chance to be selected if their initial action value is small.

16 Necessity of Target Network

Since we update many states and actions at each timestep, the action values may be affected for the very next state. If we don't have a target network, every time we update, the target $Q(s', a')$ will also change as well. Thus, it will not be stable.

17 Effects of Replay Buffer Size

If the replay buffer is too small, it's likely that we sample correlated elements, hence the training will not be stable. In addition, the model will train with recent steps if the replay buffer is too small, which may loss the good steps learned before. However, if the replay buffer is too large, there is likely that we couldn't sample new steps, and it also requires a lot of memory and might slow down the training.