Deep Learning and Practice Report

Lab6: Deep Q-Network and Deep Deterministic Policy Gradient

1 A tensorboard plot showing episode rewards of at least 800 training episodes in LunarLander-v2

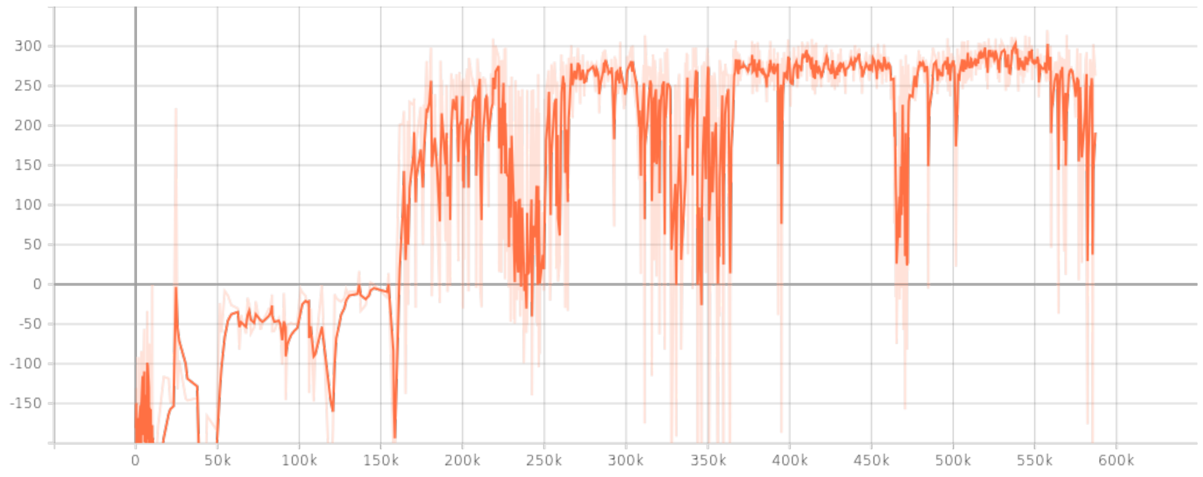


Figure 1: TensorBoard plot of DQN in LunarLander-v2.

2 A tensorboard plot showing episode rewards of at least 800 training episodes in LunarLanderContinuous-v2

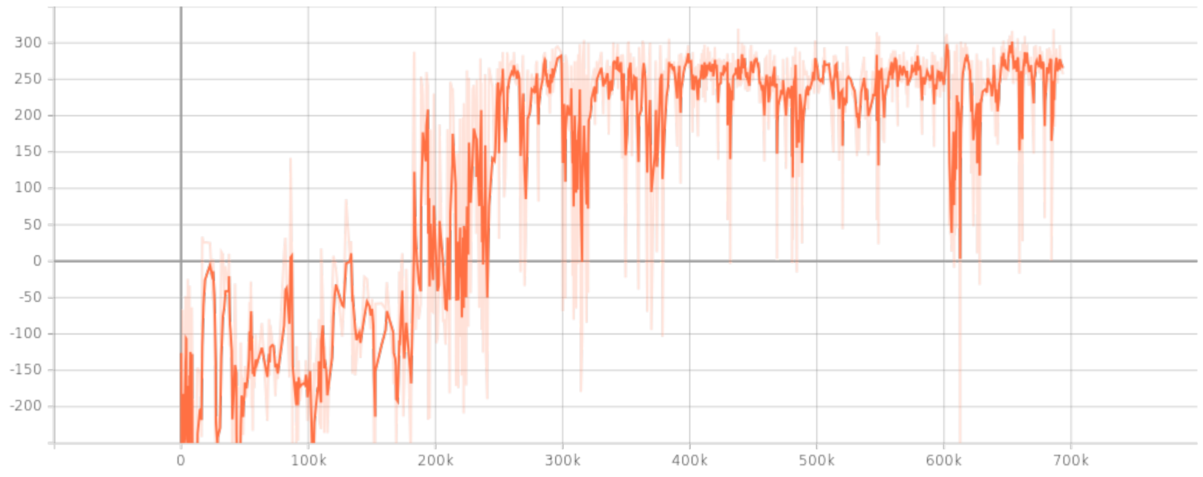
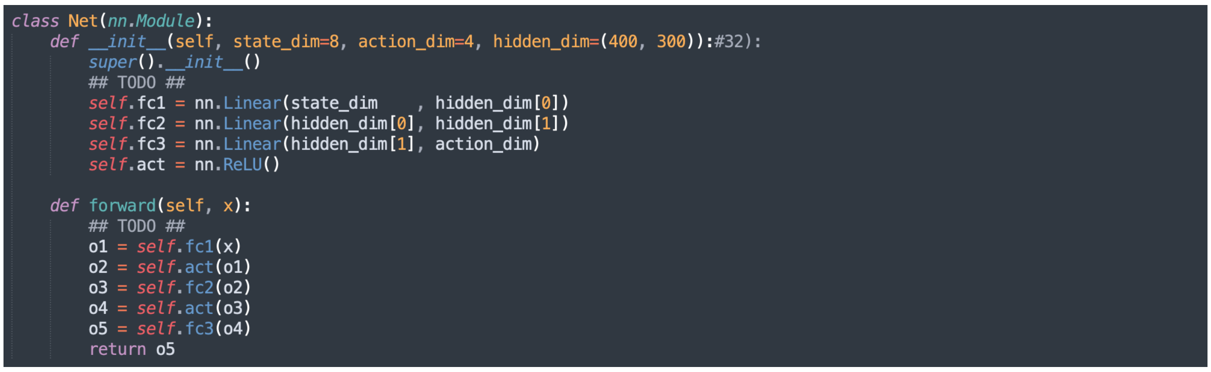


Figure 2: TensorBoard plot of DDPG in LunarLanderContinuous-v2.

3 Describe your major implementation of both algorithms in detail.

In this section, I will discuss my code implementation details of both **d**eep **Q**-**n**etwork (**DQN**) and **d**eep **d**eterministic **p**olicy **g**radient (**DDPG**). Specifically, I use Adam optimizer for both algorithms. The learning rate for DQN is 0.0005 and the learning rate for both actor and critic networks in DDPG are set to 0.001. Both algorithms warm up for 10000 iterations and are trained for 2000 episodes in total. The discount factors are set to 0.99. The batch size for DQN is 128 and the batch size for DDPG is 64.

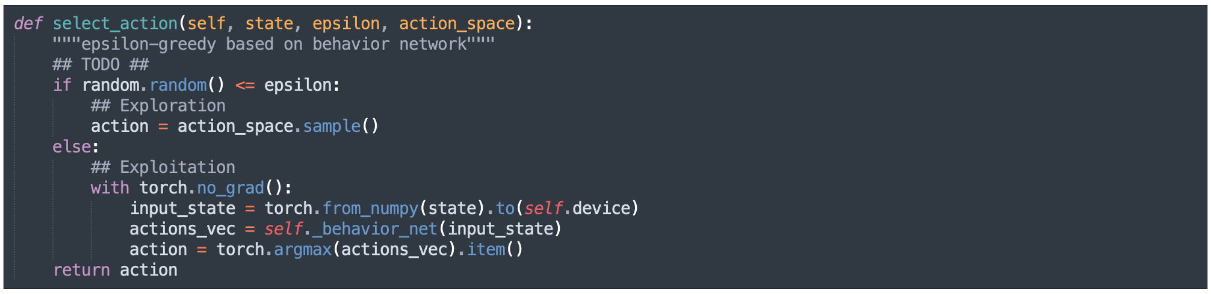
3.1 DQN



```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(400, 300)):#32):
        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.act = nn.ReLU()

    def forward(self, x):
        ## TODO ##
        o1 = self.fc1(x)
        o2 = self.act(o1)
        o3 = self.fc2(o2)
        o4 = self.act(o3)
        o5 = self.fc3(o4)
        return o5
```

Figure 3: Code implementation of value network.

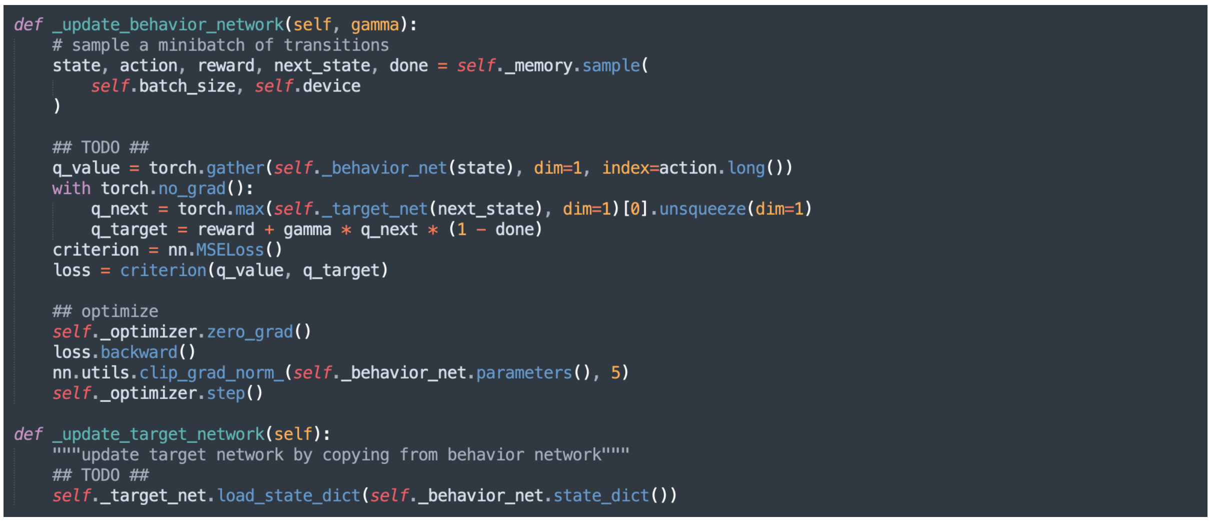
Figure 3 shows my code implementation of the value network. We can see that the network is simply composed of 3 fully connected layers, each of which is followed by a ReLU activation except for the last one. The network takes a state vector with dimension 8 as input and transform it into hidden representation with higher dimension, it then output a vector with dimension 4 which is the default cardinality of LunarLander-v2's action space. It is worthy to note that the hidden dimension is set to 32 by default. With this setting, I found that the agent starts to make horrible decisions and thus receiving low rewards in the later stage during training. Therefore, I conjecture that this is due to the low hidden dimension and I change the dimension of the first hidden vector to 400 and 300 for the second one.



```
def select_action(self, state, epsilon, action_space):
    """epsilon-greedy based on behavior network"""
    ## TODO ##
    if random.random() <= epsilon:
        ## Exploration
        action = action_space.sample()
    else:
        ## Exploitation
        with torch.no_grad():
            input_state = torch.from_numpy(state).to(self.device)
            actions_vec = self._behavior_net(input_state)
            action = torch.argmax(actions_vec).item()
    return action
```

Figure 4: Code implementation of ϵ-greedy action selection.

Figure 4 shows my code implementation of ϵ-greedy action selection. With probability ϵ, I let the agent randomly choose an action from the action space, which is also called exploration that encourages the model to explore different choices at unseen states. On the other hand, the agent would choose a greedy action based on the estimated value returned by the behaviour network with probability $1 - \epsilon$.



```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 = torch.gather(self._behavior_net(state), dim=1, index=action.long())
    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), dim=1)[0].unsqueeze(dim=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"""
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

Figure 5: Code implementation of updating both behaviour and target networks.

Figure 5 shows my code implementation of how the behaviour and target networks are updated. I first sample a mini-batch of M state transition observations from the replay memory, this approach is called experience replay. Given the i-th observation (s_i, a_i, r_i, s_{i+1}), the target value is written as follows,

$$t_i = \begin{cases} r_i, & \text{if episode terminates at step } i+1 \\ r_i + \gamma \max_{a'} Q_t(s_{i+1}, a'; \theta_t), & \text{otherwise} \end{cases} \tag{1}$$

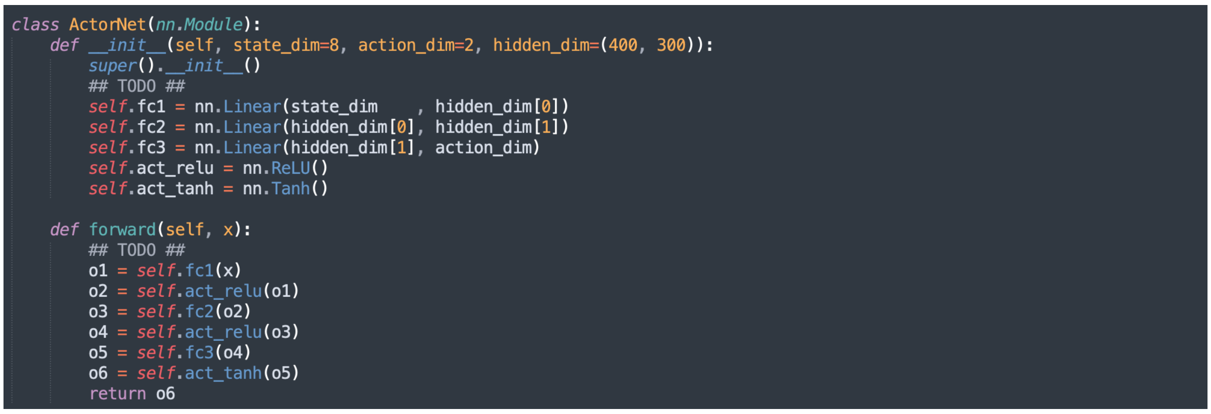
where γ is the discount factor, Q_t denotes the target network with parameters θ_t and a' belongs to the action space. Since we want to minimize the error between the estimated value and the target value t_i, we can define the loss function as follows,

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} (t_i - Q(s_i, a_i; \theta))^2 \tag{2}$$

which is the well-known **m**ean-**s**quare **e**rror (**MSE**) and Q denotes the behaviour network with parameters θ. Note that the behaviour network is updated every 4 iterations, and the target network is only updated every 1000 iterations.

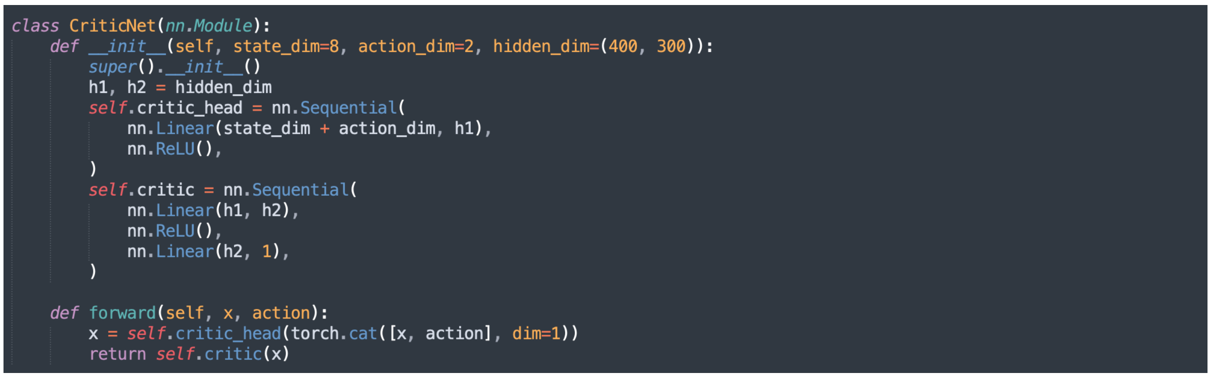
3.2 DDPG

Figure 6 and 7 shows the code implementation of the actor network and critic network respectively. We can see that both networks consist of 3 fully connected layers with first hidden dimension set to 400 and 300 for the second one. Both networks have a ReLU function followed by the first 2 layers. The activation of actor network's last layer is tanh. Given the current state, the action network first predicts an action from the action space of LunarLanderContinuous-v2, and the critic network estimates the Q-value (which represents the sum of current and future rewards) based on current state and the predicted action. Specifically, the actor network takes the state vector of dimension 8 as input, and it outputs the action vector with dimension 2. The critic network takes the state vector and the selected action as inputs simultaneously, and its output is a 1-dimensional value that represents the estimated Q-value.



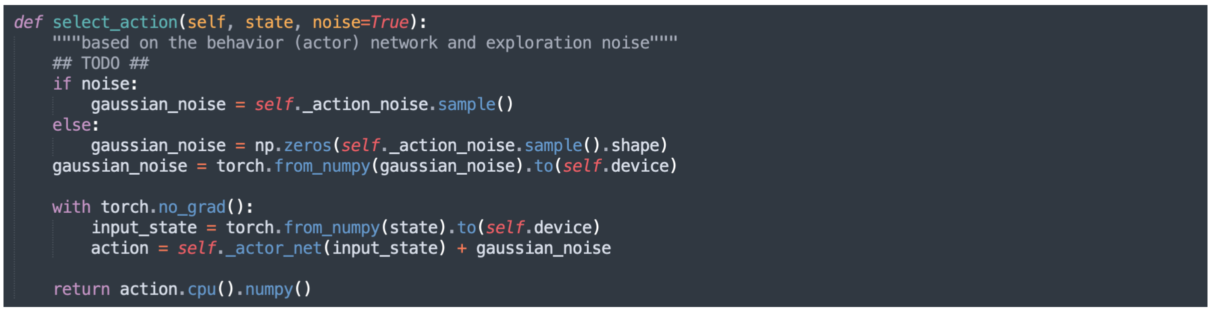
```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim    , hidden_dim[0])
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.fc3 = nn.Linear(hidden_dim[1], action_dim)
        self.act_relu = nn.ReLU()
        self.act_tanh = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        o1 = self.fc1(x)
        o2 = self.act_relu(o1)
        o3 = self.fc2(o2)
        o4 = self.act_relu(o3)
        o5 = self.fc3(o4)
        o6 = self.act_tanh(o5)
        return o6
```

Figure 6: Code implementation of actor network.



```python
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)
```

Figure 7: Implementation of critic network from sample code.



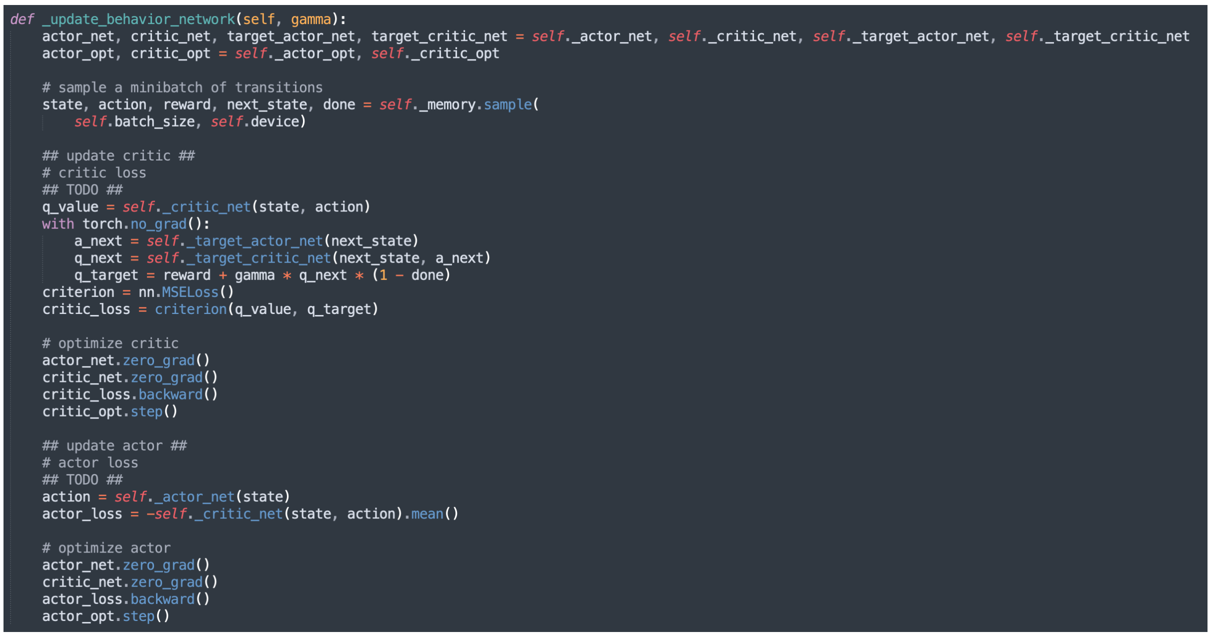
```python
def select_action(self, state, noise=True):
    """based on the behavior (actor) network and exploration noise"""
    ## TODO ##
    if noise:
        gaussian_noise = self._action_noise.sample()
    else:
        gaussian_noise = np.zeros(self._action_noise.sample().shape)
    gaussian_noise = torch.from_numpy(gaussian_noise).to(self.device)

    with torch.no_grad():
        input_state = torch.from_numpy(state).to(self.device)
        action = self._actor_net(input_state) + gaussian_noise

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

Figure 8: Code implementation of action selection.

Figure 8 shows my code implementation of action selection. In DDPG, since we are dealing with continuous action space, greedy action selection is no longer applied. Here, we select an action by the actor network mentioned above. Also, to encourage the agent to explore more possible choices, we add an exploration noise sampled from a Gaussian distribution. A formal expression of this procedure can be written as follows,

$$a_t = \mu(s_t|\theta_\mu) + N_t \tag{3}$$

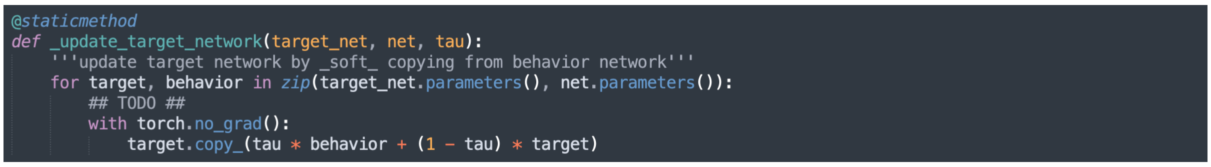
where a_t is the action predicted by the actor network μ with parameters θ_μ based on the current state s_t, and N_t denotes the sampled Gaussian exploration noise.



```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 = 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)

    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    ## update actor ##
    # actor loss
    ## TODO ##
    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()
```

Figure 9: Code implementation of updating behaviour networks.

Figure 9 shows my code implementation of how the behaviour networks are updated. Further derivations of the model's gradient and how the loss function is defined are discussed in section 5 and 6.



```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 ##
        with torch.no_grad():
            target.copy_(tau * behavior + (1 - tau) * target)
```

Figure 10: Code implementation of updating target networks.

Figure 10 shows my code implementation of how the target networks are updated. It is noteworthy that we apply **soft** target updating in DDPG, which is quite different from DQN. Here, we separately update each parameters set of each network. The formal expression of this procedure can be written as follows,

$$\theta_t = \tau\theta + (1 - \tau)\theta_t \tag{4}$$

where $\tau \ll 1$ is a hyperparameter that determines the portion of original parameters' value that we want to retain, θ and θ_t denotes the parameters set of an arbitrary behaviour network and target network respectively.

4 Describe differences between your implementation and algorithms.

For both DQN and DDPG, the first 10000 iteration of the game is set as warm up steps. The warm up steps aims to make the agent explore every possible actions at any situation in the beginning. During this interval, we won't update the networks' parameters, and the agent's action is randomly sampled from the action space. All the state transitions are stored into the replay memory. One more difference from the algorithm when training DQN is that we only update the behavior network every 4 iterations.

5 Describe your implementation and the gradient of actor updating.

Define the actor network as μ with parameters θ_μ and the critic network as Q with parameters θ_Q. The actor network should update its parameters (policy) in the direction suggested by the critic network. Firstly we sample a random mini-batch of M transitions (s_i, a_i, r_i, s_{i+1}) from the replay memory buffer. Given current state s_i, we first obtain the action $\mu(s_i|\theta_\mu)$ via the actor network. Next, we calculate the Q-value $Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q)$ by the critic network. The goal of the actor network is to select an action that maximizes the value returned by the critic network, hence we can define the loss function as

$$\mathcal{L}_\mu = -\frac{1}{M}\sum_i Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q) \tag{5}$$

, where we calculate the average over all M samples. Further derivation of the actor network's gradient is shown as follows,

$$\begin{aligned}\nabla_{\theta_\mu}\mathcal{L}_\mu &= -\frac{1}{M}\sum_i \nabla_{\theta_\mu} Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q) \\ &= -\frac{1}{M}\sum_i \nabla_a Q(s, a|\theta_Q)\Big|_{s=s_i, a=\mu(s_i|\theta_\mu)} \nabla_{\theta_\mu}\mu(s|\theta_\mu)\Big|_{s=s_i}\end{aligned} \tag{6}$$

Code implementation can be referred to figure 9 in section 3.

6 Describe your implementation and the gradient of critic updating.

Define the target network of critic as Q_t with parameters θ_{Q_t} and the target network of actor as μ_t with parameters θ_{μ_t}. For the i-th transition sampled from the mini-batch mentioned in previous section, the target Q-value for the critic network is set as

$$t_i = r_i + \gamma Q_t(s_{i+1}, \mu_t(s_{i+1}|\theta_{\mu_t})|\theta_{Q_t}) \tag{7}$$

, where γ is the discount factor. The goal of the critic network is to minimize the difference between the Q-value $Q(s_i, a_i|\theta_Q)$ and the target value t_i over all M transitions. Therefore, the loss function for the critic network is written as follows,

$$\mathcal{L}_Q = \frac{1}{M}\sum_i (t_i - Q(s_i, a_i|\theta_Q))^2 \tag{8}$$

which is known as the **m**ean-square **e**rror (**MSE**). Further derivation of the critic network's gradient is shown as follows,

$$\begin{aligned}\nabla_{\theta_Q}\mathcal{L}_Q &= \frac{1}{M}\sum_i \nabla_{\theta_Q}(t_i - Q(s_i, a_i|\theta_Q))^2 \\ &\approx (r_i + \gamma Q(s_{i+1}, \mu(s_{i+1}|\theta_{\mu_t})|\theta_{Q_t})) - Q(s_i, a_i|\theta_Q)\nabla_{\theta_Q} Q(s_i, a_i|\theta_Q)\end{aligned} \tag{9}$$

Code implementation can be referred to figure 9 in section 3.

7 Explain effects of the discount factor.

A general form of an agent's discounted reward G_t can be written as follows,

$$\begin{aligned} G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots \\ &= R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \cdots) \end{aligned} \tag{10}$$

where $\gamma < 1$ is the discount factor. The discount factor determines the importance of estimated future rewards, i.e., the second term in the second line of equation 10. Smaller discount factor encourages the agent focus more on current rewards, and larger discount factor reduces the impact of long-term rewards. In DQN and DDPG, the estimated future rewards are often represented by the Q-value returned from the deep neural network. An example is the critic network's target Q-value of DDPG written in equation 7 where $Q_t(S_{i+1}, \mu_t(s_{i+1}|\theta_{\mu_t})|\theta_{Q_t})$ represents the estimated future rewards.

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

ϵ-greedy is a simple method that balances exploration and exploitation. At each time step, the agent selects an action at random with probability ϵ and selects the greedy action based on its learned policy with probability $1 - \epsilon$. The reason of adopting ϵ-greedy action selection is that if we only select the best action, there is a chance that we miss a real better one.

9 Explain the necessity of the target network.

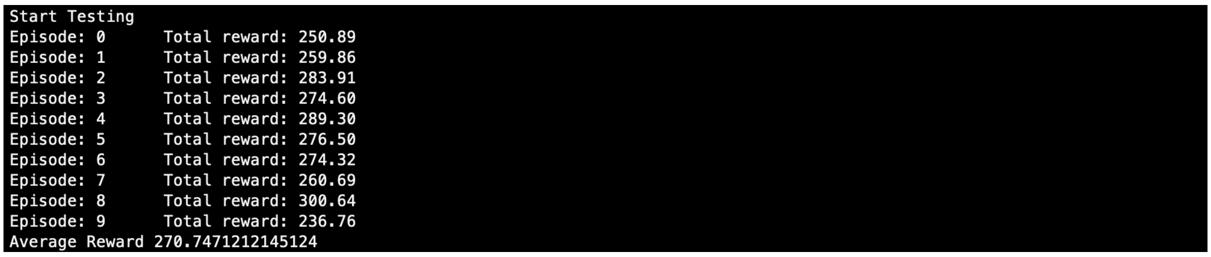
We know that when training DQN and DDPG, the Q-value of each state is estimated by a Q-network / critic network. Different from ordinary Q-learning, the exact value function has been replaced by a function approximator formed by a deep neural network now. In such manner, Q-values of the same state may be different even after a single optimization step on the Q-network, which can possibly make the agent's learning process unstable. Hence, using a stable target network should somehow alleviate this problem. A typical solution is to predict the value of each state by a fixed network, and we only update its parameters after a pre-defined period.

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

The larger the replay buffer size, the less likely we will sample correlated elements. Hence, the more stable the training process should be. However, larger buffer size induces more memory usage, which can slow down the training process. On the other hand, small replay buffer size may increase the chance of sampling correlated elements. Under this situation, the training process may be more unstable though it requires less training time.

11 Performance

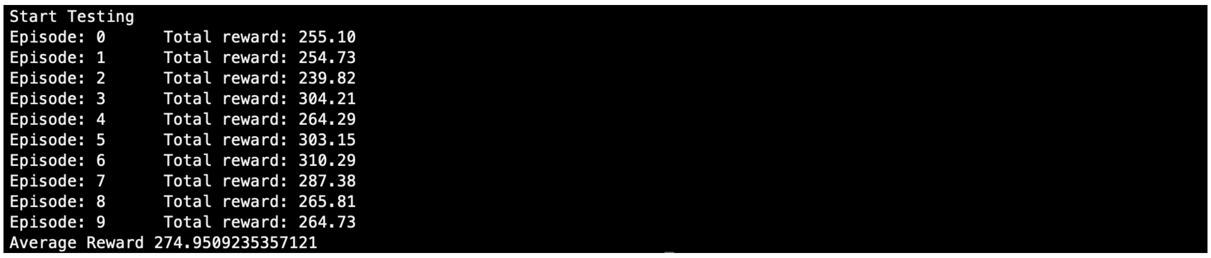
11.1 [LunarLander-v2] Average reward of 10 testing episodes



```
Start Testing
Episode: 0      Total reward: 250.89
Episode: 1      Total reward: 259.86
Episode: 2      Total reward: 283.91
Episode: 3      Total reward: 274.60
Episode: 4      Total reward: 289.30
Episode: 5      Total reward: 276.50
Episode: 6      Total reward: 274.32
Episode: 7      Total reward: 260.69
Episode: 8      Total reward: 300.64
Episode: 9      Total reward: 236.76
Average Reward 270.7471212145124
```

Figure 11: Average reward of 10 testing episodes by DQN in LunarLander-v2.

11.2 [LunarLanderContinuous-v2] Average reward of 10 testing episodes



```
Start Testing
Episode: 0      Total reward: 255.10
Episode: 1      Total reward: 254.73
Episode: 2      Total reward: 239.82
Episode: 3      Total reward: 304.21
Episode: 4      Total reward: 264.29
Episode: 5      Total reward: 303.15
Episode: 6      Total reward: 310.29
Episode: 7      Total reward: 287.38
Episode: 8      Total reward: 265.81
Episode: 9      Total reward: 264.73
Average Reward 274.9509235357121
```

Figure 12: Average reward of 10 testing episodes by DDPG in LunarLanderContinuous-v2.