Summer 2023 DLP Lab5

312552017 董家鳴

1. Experimental Results (100%)

A. Screenshot of tensorboard and testing results on LunarLander-v2 (30%)

Figure 1: Testing results of LunarLander-v2 using DQN

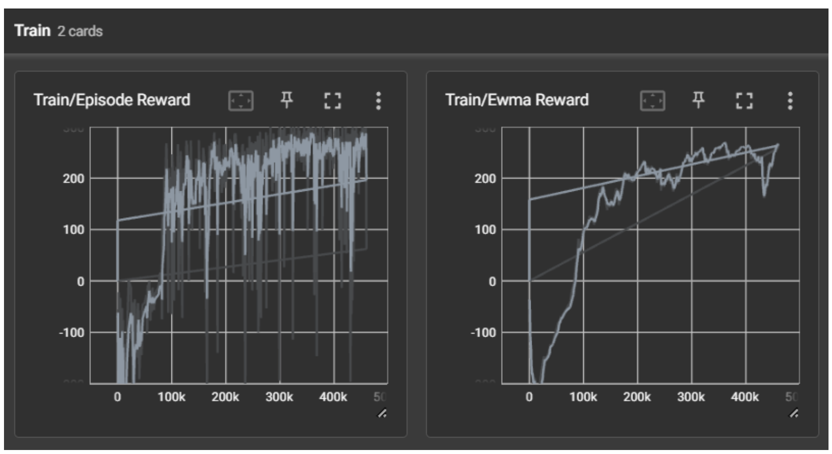


Figure 2: Tensorboard of LunarLander-v2 using DQN

B. Screenshot of tensorboard and testing results on LunarLanderContinuous-v2 (30%)

Figure 3: Testing results of LunarLanderContinuous-v2 using DDPG

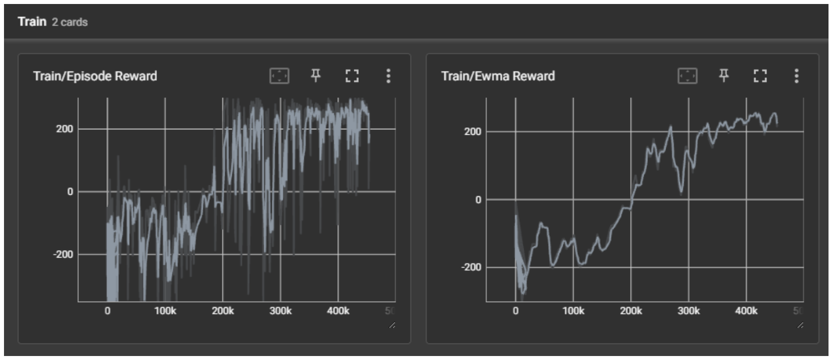
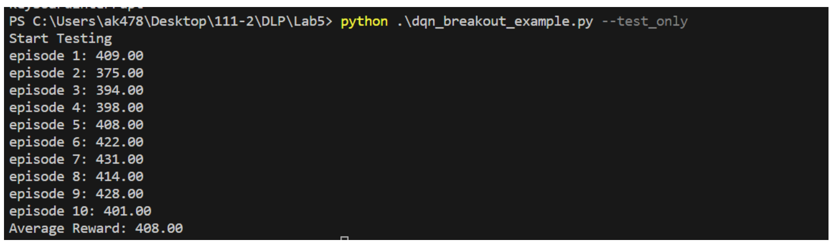


Figure 4: Tensorboard of LunarLanderContinuous-v2 using DDPG

C. Screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4 (40%)



```
PS C:\Users\ak478\Desktop\111-2\DLP\Lab5> python .\dqn_breakout_example.py --test_only
Start Testing
episode 1: 409.00
episode 2: 375.00
episode 3: 394.00
episode 4: 398.00
episode 5: 408.00
episode 6: 422.00
episode 7: 431.00
episode 8: 414.00
episode 9: 428.00
episode 10: 401.00
Average Reward: 408.00
```

Figure 5: Testing results of BreakoutNoFrameskip-v4 using DQN

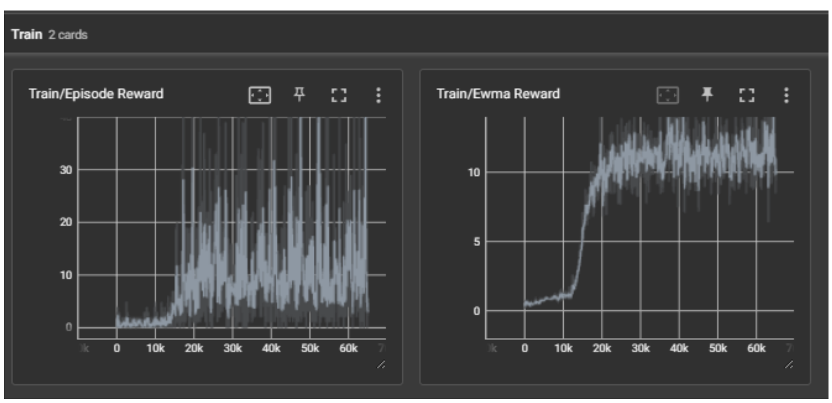


Figure 6: Tensorboard of BreakoutNoFrameskip-v4 using DQN

2. Experimental Results of bonus parts (DDQN, TD3) (15%)

A. Screenshot of tensorboard and testing results on LunarLander-v2 by DDQN (5%)

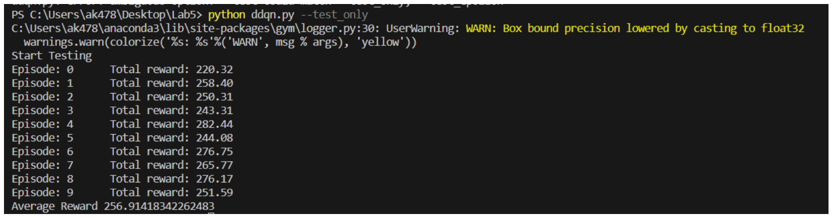


Figure 7: Testing results of LunarLander-v2 using DDQN

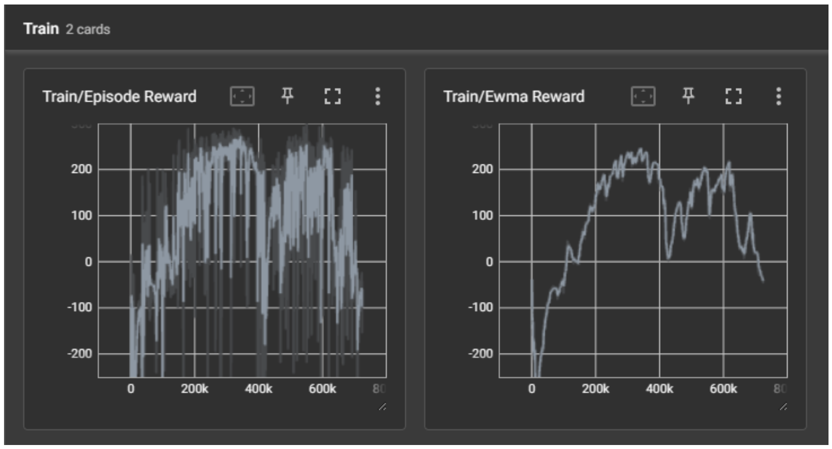


Figure 8: Tensorboard of LunarLander-v2 using DDQN

3. Questions (10%)

A. Major implementation of both DQN and DDPG (5%)

- **Q network updating in DQN** We have two networks in DQN. One is the behavior network and the other is the target network as depicted in figure 9 [1]. The target network is used to obtain the score for the next state and the next action, while the behavior network is used to obtain the action for the current state. We can train the behavior network to choose better actions for the future (i.e., actions with higher scores) by fixing the target network. In practice, the target network is a copy of the behavior network after a period of training. The reason for copying is that after a period of training, the behavior network becomes more proficient, so the scores for the next state and the next action will also change. The figure 10 shows how to implement it.

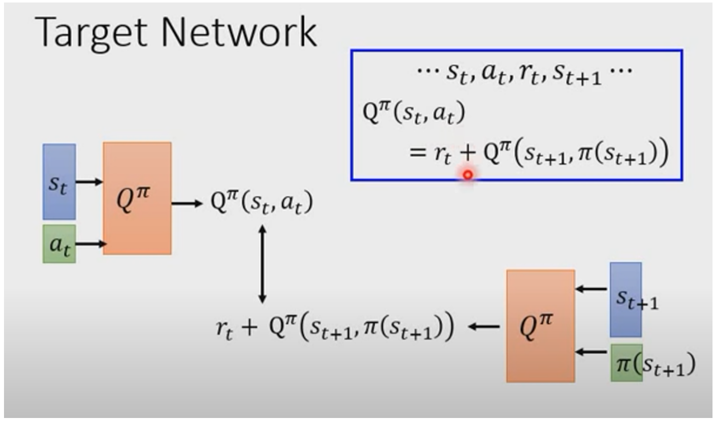
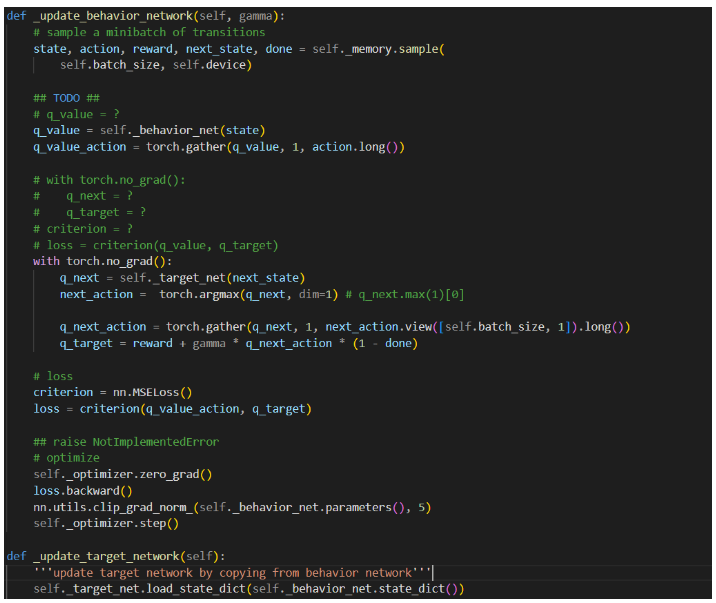


Figure 9: Target network [1]



```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 = ?
    q_value = self._behavior_net(state)
    q_value_action = torch.gather(q_value, 1, action.long())

    # with torch.no_grad():
    #    q_next = ?
    #    q_target = ?
    # criterion = ?
    # loss = criterion(q_value, q_target)
    with torch.no_grad():
        q_next = self._target_net(next_state)
        next_action =  torch.argmax(q_next, dim=1) # q_next.max(1)[0]

        q_next_action = torch.gather(q_next, 1, next_action.view([self.batch_size, 1]).long())
        q_target = reward + gamma * q_next_action * (1 - done)

    # loss
    criterion = nn.MSELoss()
    loss = criterion(q_value_action, q_target)

    ## raise NotImplementedError
    # 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'''
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

Figure 10: Target network implement

- **The gradient of actor updating in DDPG** We have two networks in DDPG. One is the actor network and the other is the critic network as. The critic network estimates the Q-value of the state-action pair. The actor network is responsible for determining the actions by using critic network to take given a current state. It maps the current state to a current action in the continuous action space. The gradient update for the actor's current action is performed in the direction that increases the Q-value, leading the actor to choose actions that are expected to result in higher returns at the next state(the negative sign is used because we want to maximize the Q-values). The figure 11 shows how to implement it.
- **The gradient of critic updating in DDPG** For the critic updating, it's trained by considering the Q value in the next state and current state as depicted in figure 12 so that we can judge the actor network.
- **Implementation of DDQN** By following [2], the only difference between DQN and

```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 = ?
    # with torch.no_grad():
    #    a_next = ?
    #    q_next = ?
    #    q_target = ?
    # criterion = ?
    # critic_loss = criterion(q_value, q_target)
    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)
    ## raise NotImplementedError
    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()
    ## update actor ##
    # actor loss
    ## TODO ##
    # action = ?
    # actor_loss = ?
    action = self._actor_net(state)
    actor_loss = -self._critic_net(state, action).mean()

    ## raise NotImplementedError
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

Figure 11: Update the actor

```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 = ?
    # with torch.no_grad():
    #    a_next = ?
    #    q_next = ?
    #    q_target = ?
    # criterion = ?
    # critic_loss = criterion(q_value, q_target)
    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)
    ## raise NotImplementedError
    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()
    ## update actor ##
    # actor loss
    ## TODO ##
    # action = ?
    # actor_loss = ?
    action = self._actor_net(state)
    actor_loss = -self._critic_net(state, action).mean()

    ## raise NotImplementedError
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

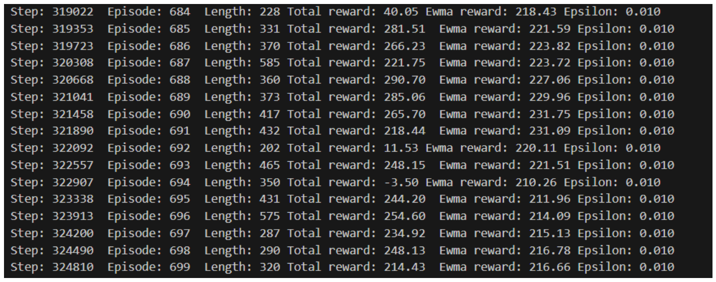
Figure 12: Update the critic

DDQN is how to calculate the target Q value. First, use Q (behavior network) to decide which action to take, then use $Q\prime$ (target network) to calculate the Q value. The implementation is shown in the figure 13.

```python
with torch.no_grad():
    # dqn
    # q_next = self._target_net(next_state)
    # next_action =  torch.argmax(q_next, dim=1) # q_next.max(1)[0]

    # q_next_action = torch.gather(q_next, 1, next_action.view([self.batch_size, 1]).long())
    # q_target = reward + gamma * q_next_action * (1 - done)

    # ddqn
    q_next_behavior = self._behavior_net(next_state)
    next_action_behavior = torch.argmax(q_next_behavior, dim=1)

    q_next_target = self._target_net(next_state)

    q_next_action_prime = torch.gather(q_next_target, 1, next_action_behavior.view([self.batch_size, 1]).long())
    q_target = reward + gamma * q_next_action_prime * (1 - done)
```

Figure 13: How to do DDQN

- **Goal of DDQN** As mentioned in [2], the goal is to avoid the over-estimated problem in the DQN. The figure 14 shows that the estimated reward in DDQN is closest to the actual reward, Ewma reward in the game.



```
Step: 319022  Episode: 684  Length: 228 Total reward: 40.05 Ewma reward: 218.43 Epsilon: 0.010
Step: 319353  Episode: 685  Length: 331 Total reward: 281.51  Ewma reward: 221.59 Epsilon: 0.010
Step: 319723  Episode: 686  Length: 370 Total reward: 266.23  Ewma reward: 223.82 Epsilon: 0.010
Step: 320308  Episode: 687  Length: 585 Total reward: 221.75  Ewma reward: 223.72 Epsilon: 0.010
Step: 320668  Episode: 688  Length: 360 Total reward: 290.70  Ewma reward: 227.06 Epsilon: 0.010
Step: 321041  Episode: 689  Length: 373 Total reward: 285.06  Ewma reward: 229.96 Epsilon: 0.010
Step: 321458  Episode: 690  Length: 417 Total reward: 265.70  Ewma reward: 231.75 Epsilon: 0.010
Step: 321890  Episode: 691  Length: 432 Total reward: 218.44  Ewma reward: 231.09 Epsilon: 0.010
Step: 322092  Episode: 692  Length: 202 Total reward: 11.53 Ewma reward: 220.11 Epsilon: 0.010
Step: 322557  Episode: 693  Length: 465 Total reward: 248.15  Ewma reward: 221.51 Epsilon: 0.010
Step: 322907  Episode: 694  Length: 350 Total reward: -3.50 Ewma reward: 210.26 Epsilon: 0.010
Step: 323338  Episode: 695  Length: 431 Total reward: 244.20  Ewma reward: 211.96 Epsilon: 0.010
Step: 323913  Episode: 696  Length: 575 Total reward: 254.60  Ewma reward: 214.09 Epsilon: 0.010
Step: 324200  Episode: 697  Length: 287 Total reward: 234.92  Ewma reward: 215.13 Epsilon: 0.010
Step: 324490  Episode: 698  Length: 290 Total reward: 248.13  Ewma reward: 216.78 Epsilon: 0.010
Step: 324810  Episode: 699  Length: 320 Total reward: 214.43  Ewma reward: 216.66 Epsilon: 0.010
```

Figure 14: DDQN avoids over-estimate

B. Explain effects of the discount factor (1%)

- The discount factor is commonly denoted as . It will be used in the following formula.

$$q_{\text{target}} = \text{reward} + \gamma \cdot q_{\text{next\_action}} \cdot (1 - \text{done}), \quad 0 < \gamma < 1$$

- The larger the value of γ, the more the critic considers future rewards. Conversely, the more it only considers the immediate reward, this will affect whether the actor's action has considered the future reward for the current state.

C. Explain benefits of epsilon-greedy in comparison to greedy action selection (1%)

- First, we must understand the concepts of exploration and exploitation. Exploration is randomly choosing an action from all possible actions, while exploitation is selecting an action that has previously yielded the maximum reward. The epsilon-greedy strategy allows the agent to start with a higher probability of exploration, and as the number of steps increases, the probability of using exploitation (i.e., taking the best action from the past) becomes greater.

D. Explain the necessity of the target network (1%)

- The Q-learning update rule involves both current and next state Q-values:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$

- When using a single network to calculate both Q-values, the Q-learning process can become unstable. The same network simultaneously estimates the current Q-values and the next state Q-values, leading to a moving target problem. As the Q-values are updated, the **target** also changes, causing oscillations and divergence in the training.

E. Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander (2%)

- The Hint: Trick 1 on the spec of this lab tells us while in the training when the agent dies, the game is over. (episode_life=True) While in the LunarLander game, there's no situation is dies.
- The Hint: Trick 2 on the spec of this lab tells us while in the training we're using the image containing the ball as the state. To know the direction of the ball's movement, we must input 4 consecutive frames, allowing the agent to better judge the current state. In LunarLander, there are no moving objects, so there is no need for consecutive input.

References

[1] Hung-yi Lee. *DRL Lecture 3: Q-learning (Basic Idea)*. 2018. URL: `https://youtu.be/azBugJzmz-o?t=746`.

[2] Hung-yi Lee. *DRL Lecture 4: Q-learning (Advanced Tips)*. 2018. URL: `https://youtu.be/2-zGCx4iv_k?t=373`.