Deep Learning Lab5 **Student ID: 312554014 Name: 羅翊軒**

1. Experimental Results

- Testing results

DQN

```
Isadora Lab5 ♥ 20:17 lab 3.9.17 python dqn-example.py --test_only --model dqn2435.pth
Start Testing
Episode: 0      Total reward: 235.48006622953346
Episode: 1      Total reward: 262.8273832263352
Episode: 2      Total reward: 175.6634347108738
Episode: 3      Total reward: 274.5466031383733
Episode: 4      Total reward: 217.08600884058552
Episode: 5      Total reward: 213.83722179659804
Episode: 6      Total reward: 275.9137580755581
Episode: 7      Total reward: 222.34980378245808
Episode: 8      Total reward: 285.85432283120167
Episode: 9      Total reward: 227.3407619309447
Average Reward 239.0899364562462
```

DDPG

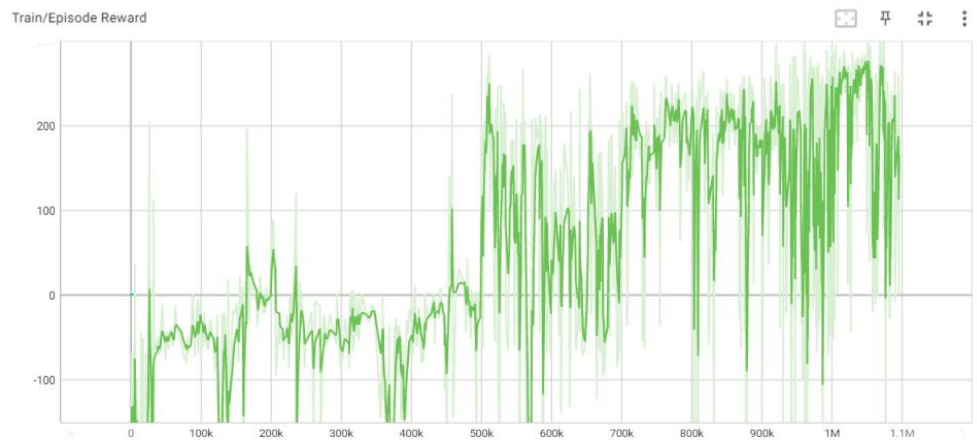
```
Isadora Lab5 ♥ 20:24 lab 3.9.17 python ddpg-example.py --test_only --model ddpg2499.pth
Start Testing
Episode: 0      Total reward: 250.69285137152093
Episode: 1      Total reward: 149.33059683063001
Episode: 2      Total reward: 281.4507361067849
Episode: 3      Total reward: 272.3021491533676
Episode: 4      Total reward: 194.45655805447979
Episode: 5      Total reward: 275.1822844217813
Episode: 6      Total reward: 224.12189666718524
Episode: 7      Total reward: 272.5138581538093
Episode: 8      Total reward: 273.30433485809283
Episode: 9      Total reward: 268.2135038171099
Average Reward 246.15687694347616
```

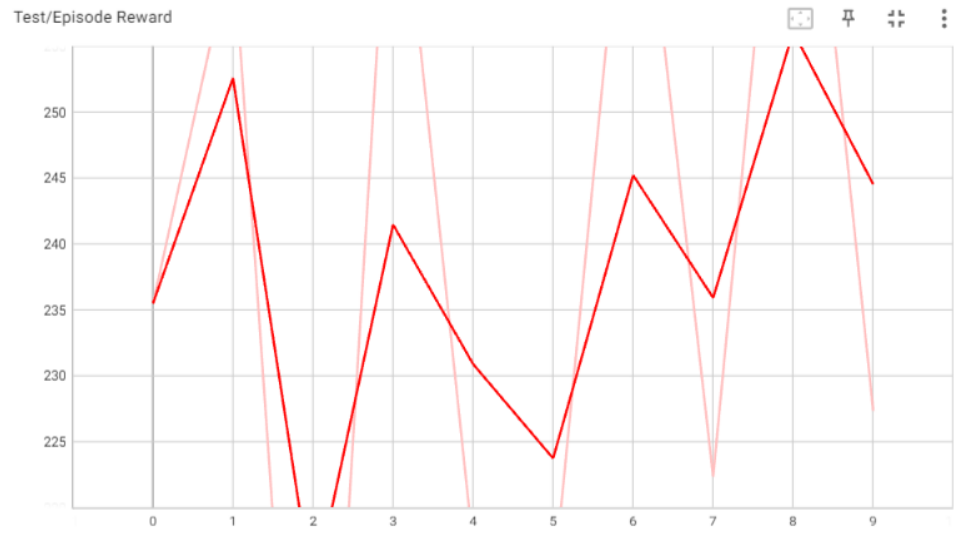
DQN-Breakout

```
Isadora DL_LAB5_312554014_羅翊軒 ♥ 10:15 lab 3.9.17 python dqn_breakout.py --test_only --test_model_path ckp
t/dqn_break_526036.pt
Start Testing
A.L.E: Arcade Learning Environment (version 0.7.5+db37282)
[Powered by Stella]
episode 1: 56.00
episode 2: 52.00
episode 3: 55.00
episode 4: 44.00
episode 5: 39.00
episode 6: 55.00
episode 7: 51.00
episode 8: 49.00
episode 9: 55.00
episode 10: 57.00
Average Reward: 51.30
```

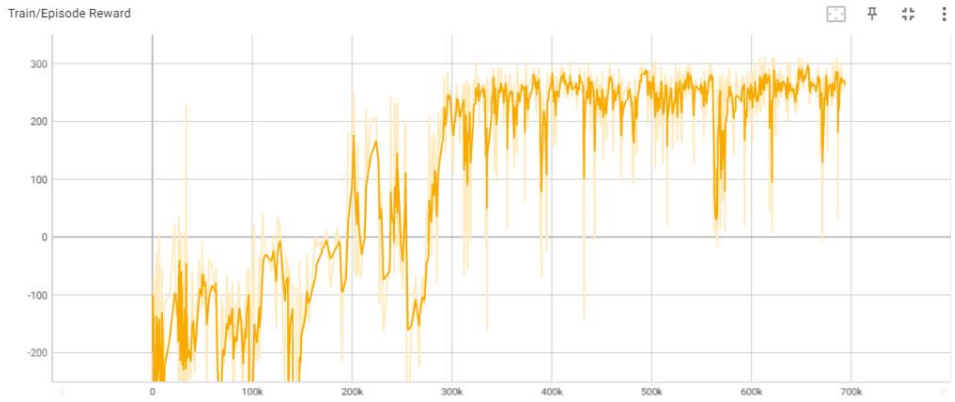
- Tensorboard

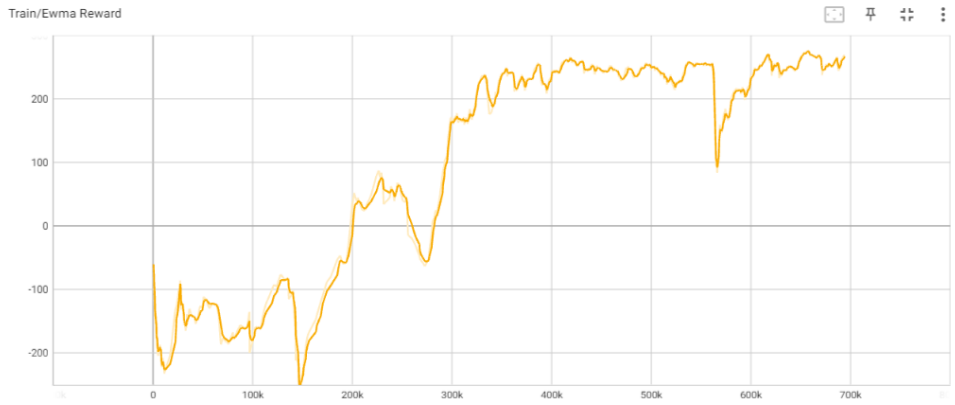
  DQN

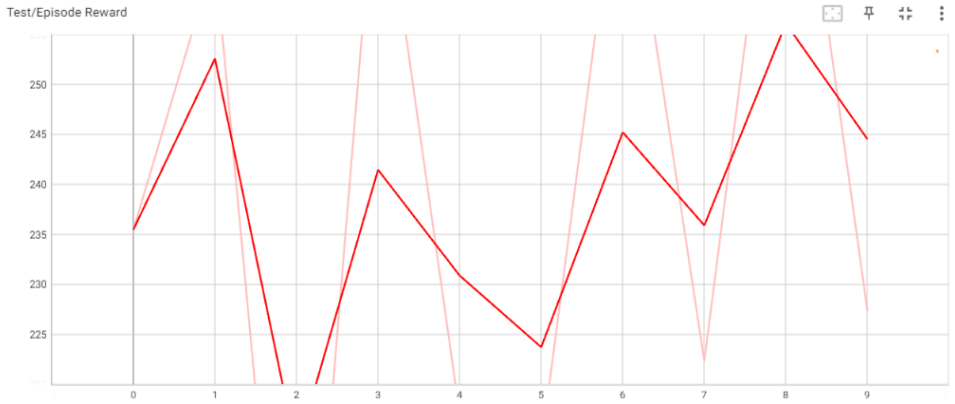




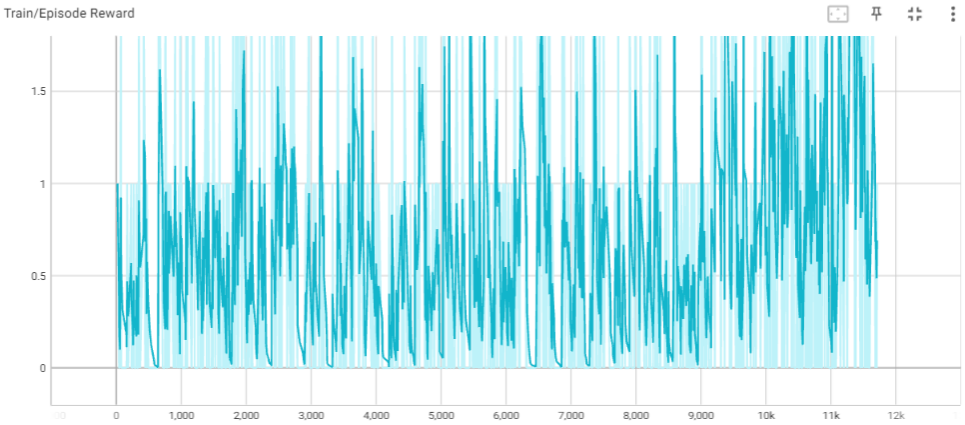
DDPG

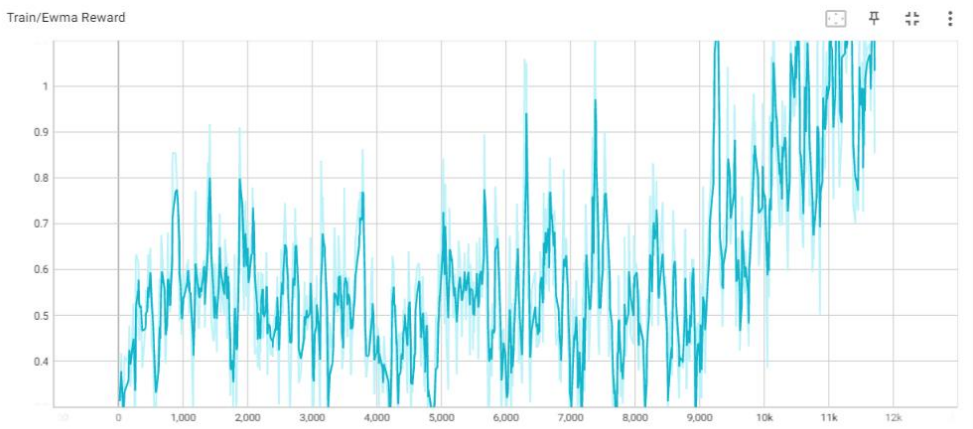

Train/Episode Reward
300
200
100
0
-100
-200
0
100k
200k
300k
400k
500k
600k
700k

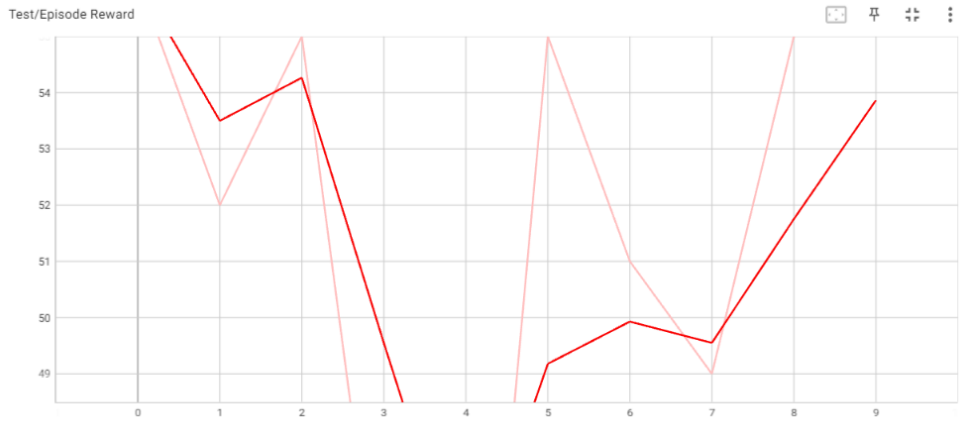

Train/Ewma Reward
200
100
0
-100
-200
0
100k
200k
300k
400k
500k
600k
700k


Test/Episode Reward
250
245
240
235
230
225
0
1
2
3
4
5
6
7
8
9

DQN-Breakout







2. Experimental Results of bonus parts (DDQN, TD3)
3. Questions

- Describe your major implementation of both DQN and DDPG in detail.

  (1) Your implementation of Q network updating in DQN.

```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_values = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state).max(1)[0].unsqueeze(1)
        q_target = reward + gamma * q_next * (1 - done)
    loss = self._criterion(q_values, q_target)

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

The `_update_behavior_network` method is responsible for updating the Q network using a batch of transitions sampled from the replay memory. First, it will sample a batch of transitions, including states, actions, rewards, next states, and done flags, from the replay memory. Then, the behavior network will calculate Q values for the sampled states and actions. The `.gather(1, action.long())` operation selects the Q values corresponding to the chosen actions, and the target Q values are calculated using the target network for the next states. These values are updated and took into account the rewards and whether the episode is done or not. Then, the loss will be calculated between the Q values predicted by the behavior network and the target Q values, and the criterion of the loss is Mean Squared Error loss function. Finally, backpropagation is performed to compute the gradients of the loss with respect to the behavior network's parameters, and the optimizer updates the behavior network's parameters to minimize the calculated loss.

(2) Your implementation and the gradient of actor updating in DDPG.

```
## update actor ##
# actor loss
## TODO ##
new_action = actor_net(state)
actor_loss = -self._critic_net(state, new_action).mean() # Used `-value` to maximize

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

It will first calculate the new action selected by the actor network based on the current state, and then calculate the Q-value associated with the new action by passing both the state and the new action through the critic network. The actor loss is calculated as the negative mean of the Q-values predicted by the critic for the chosen actions, because the goal for the loss function of the actor is to minimize the negative of the critic's assessment. Finally, using the loss calculated from last step to do backpropagation and update the parameters of the network.

(3) Your implementation and the gradient of critic updating in DDPG.

```
## update critic ##
# critic loss
## TODO ##
q_value = self._critic_net(state, action)
with torch.no_grad():
    next_action = target_actor_net(next_state)
    q_next = target_critic_net(next_state, next_action)
    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()
```

In this section, it will first calculate the Q-value predicted by the critic network for the current state-action pair, and then calculate the target Q-value for the current state-action pair based on the reward. The Q-value of the next state-action pair predicted by the target critic network

(parameter q_next), and whether the episode is done (parameter done). After calculating the target Q-value, it will compute the critic loss using the mean squared error loss function, comparing the predicted Q-value (parameter q_value) with the target Q-value (parameter q_target). Then performing backpropagation to compute the gradients of the critic loss, and update the parameter of the networks.

- Explain effects of the discount factor.

  The discount factor (called gamma) in the code determines how much the agent values future rewards compared to immediate rewards. If gamma is small (close to 0), the agent becomes shortsighted and mainly cares about immediate rewards. It tends to make decisions that bring quick gains but might miss out on better long-term rewards. If gamma is big, the agent becomes farsighted and considers future rewards strongly. It looks for actions that lead to higher cumulative rewards over time, even if they involve short-term sacrifices. By adjusting gamma, we can control how much the agent values short-term or long-term gains in the training process.

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

  In the code, using epsilon-greedy action selection to make the agent's decisions is a way to strike a balance between exploration and exploitation.

- Explain the necessity of the target network.

  The target network is a crucial component in the algorithm. Its necessity lies in stabilizing and improving the training process of the algorithm. It allows a balance between exploration and exploitation. The slow update rate of the target network ensures that the learning process has enough time to gather meaningful data for exploitation before exploration strategies are updated. It let the network can achieve smoother and more reliable learning, leading to

improved convergence and better policy optimization.

- Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander.

```
class Net(nn.Module):
    def __init__(self, num_classes=4, init_weights=True):
        super(Net, self).__init__()

        self.cnn = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4),
                                 nn.ReLU(True),
                                 nn.Conv2d(32, 64, kernel_size=4, stride=2),
                                 nn.ReLU(True),
                                 nn.Conv2d(64, 64, kernel_size=3, stride=1),
                                  nn.ReLU(True)
                                        )
        self.classifier = nn.Sequential(nn.Linear(7*7*64, 512),
                                        nn.ReLU(True),
                                        nn.Linear(512, num_classes)
                                        )
        if init_weights:
            self._initialize_weights()
```

The figure shown above is the neural network architecture of Breakout, compare to DQN which uses a few fully connected layers, it is based on a CNN to process the input.

```
def train(args, agent, writer):
    print('Start Training')
    env_raw = make_atari('BreakoutNoFrameskip-v4')
    env = wrap_deepmind(env_raw, episode_life=True, clip_rewards=True, frame_stack=True)
```

Besides, I add some parameters to preprocess the environment for training the agent on the Breakout game. I set `episode_life` this argument to True, and it refers to that the episode will end when the agent loses a life. ` Clip_rewards` to True specifies that rewards will be clipped to a certain range. Reward clipping can help stabilize training by preventing extremely large rewards from causing issues in learning. It ensures that rewards stay within a reasonable range. The last is to assign ` frame_stack` argument to True. It means that multiple consecutive frames will be stacked together to create a single observation. Stacking frames helps the agent perceive motion and understand the dynamics of the game.

```python
def test(args, agent, writer):
    print('Start Testing')
    # env_raw = make_atari('BreakoutNoFrameskip-v4', render_mode='human')
    env_raw = make_atari('BreakoutNoFrameskip-v4')
    env = wrap_deepmind(env_raw, episode_life=False, clip_rewards=False, frame_stack=True)
```

However, during the testing, I will turn `episode_life` argument and `clip_rewards` argument to False to make sure the rewards of the test result is not modified.