DL-Lab6

0816095 Hsuan Wang

May 21, 2023

1 Experimental Results

1.1 DQN for LunarLander-v2

Figure 1: DQN for LunarLander-v2 training process

```
Start Testing
Length: 253     Total reward: 232.36
Length: 383     Total reward: 281.29
Length: 235     Total reward: 266.35
Length: 233     Total reward: 285.12
Length: 201     Total reward: 299.60
Length: 410     Total reward: 230.89
Length: 412     Total reward: 275.75
Length: 312     Total reward: 239.74
Length: 473     Total reward: 225.80
Length: 377     Total reward: 222.71
Average Reward 255.96060836136704
```

Figure 2: DQN for LunarLander-v2 testing

1.2 DDPG for LunarLanderContinuous-v2

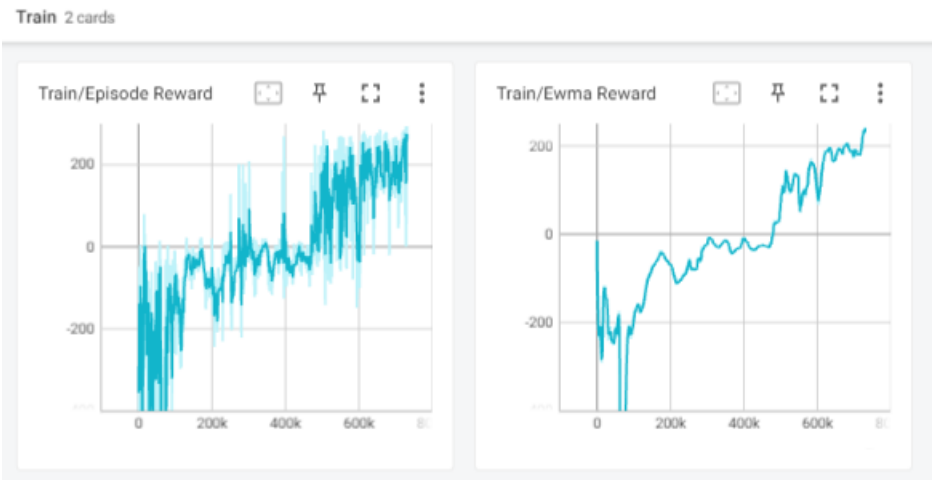

Figure 3: DDPG for LunarLanderContinuous-v2 training process

```
Start Testing
Length: 166     Total reward: 248.03
Length: 151     Total reward: 217.33
Length: 181     Total reward: 277.48
Length: 198     Total reward: 266.94
Length: 303     Total reward: 222.54
Length: 182     Total reward: 264.22
Length: 362     Total reward: 217.87
Length: 648     Total reward: 229.51
Length: 144     Total reward: 13.33
Length: 200     Total reward: 279.32
Average Reward 223.65590825274302
```

Figure 4: DDPG for LunarLanderContinuous-v2 testing

1.3 DQN for BreakoutNoFrameskip-v4

Figure 5: DQN for BreakoutNoFrameskip-v4 training process

Figure 6: DQN for BreakoutNoFrameskip-v4 testing rewards

```
Start Testing
episode 1: 400.00
episode 2: 856.00
episode 3: 455.00
episode 4: 420.00
episode 5: 855.00
episode 6: 857.00
episode 7: 845.00
episode 8: 846.00
episode 9: 431.00
episode 10: 461.00
Average Reward: 642.60
```

Figure 7: DQN for BreakoutNoFrameskip-v4 testing

2 Experimental Results of bonus parts

2.1 DDQN for LunarLander-v2

Figure 8: DDQN for LunarLander-v2 testing

```
Start Testing
Length: 199     Total reward: 224.30
Length: 245     Total reward: 261.84
Length: 223     Total reward: 249.03
Length: 221     Total reward: 280.63
Length: 234     Total reward: 280.53
Length: 197     Total reward: 293.61
Length: 203     Total reward: 308.95
Length: 194     Total reward: 271.04
Length: 190     Total reward: 305.65
Length: 203     Total reward: 296.09
Average Reward 277.1666207265598
```

Figure 9: DDQN for LunarLander-v2 training process

3 Questions

3.1 Describe your major implementation of both DQN and DDPG in detail

3.1.1 Your implementation of Q network updating in DQN

I calculate the q target using $r + \gamma * \max_{a'} Q(s', a')$ where the Q value is obtained from the target network, and update the behavior network towards the q target using mse loss. This is done every 4 steps. Then update the target network by copying the behavior network every 1000 steps.

```
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 = self._behavior_net(state).gather(dim=1, index=action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state)
        max_q_next = torch.max(q_next, dim=1)[0].reshape(-1, 1)
        q_target = reward + gamma*max_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()
```

3.1.2 Your implementation and the gradient of actor updating in DDPG

When training the actor, we don't need the target networks, we just pass the predicted action into the critic and maximize the q value(minimizing -q value). But what we want is to train the actor so we freeze the critic when training the actor.

```
def _update_behavior_network(self, gamma):
    actor_net, critic_net = self._actor_net, self._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 ##
    ## 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 ##
    ## 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()
```

3.1.3 Your implementation and the gradient of critic updating in DDPG

When training the critic, we are going to uset the same concept in the dqn model, where there are an identical pair of target network for critic and actor, just with older parameters. So when we want to train the critic we can freeze the target networks and train it just like how we train dqn.

3.2 Explain effects of the discount factor

The discount factor γ determines how far the agent can see in the future. For example, if we set the discount factor to 0, the agent will only be able to see the immediate reward after every action, and will only be able to choose actions based on this reward. However, if we set the discount factor to 1 the agent will be able make decisions based on the reward of the entire trajectory.

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

When training the agent, we want to use epsilon-greedy, because the agent is still new to the environment, it needs to explore more, if we only use greedy action, it is very easy for the agent to stuck at a local minimum.

3.4 Explain the necessity of the target network

The target network is used to stabilize the training process. Because when we try to update a network towards itself, it might cause oscillation problems where the model oscillates between values and never converge.

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

First, I stack four frames together to form a state for the agent to have more information about the balls direction and speed.

Second, I cropped out the score at the top of the state image because I deemed that unuseful, thus achieving less noise and less computation needed.

Third, I've noticed that the agent is prone to stuck in a loop when testing, but setting a fixed noise to break the loop may effect performance. So I let the noise build up when the agent is getting 0 rewards for a long time.