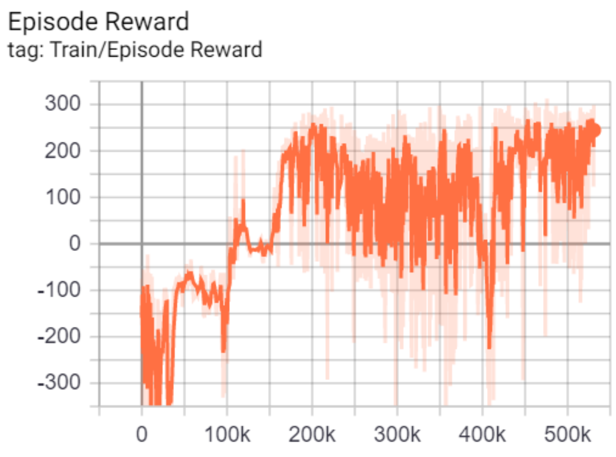
Lab6 Report

0856091 蘇邱弘

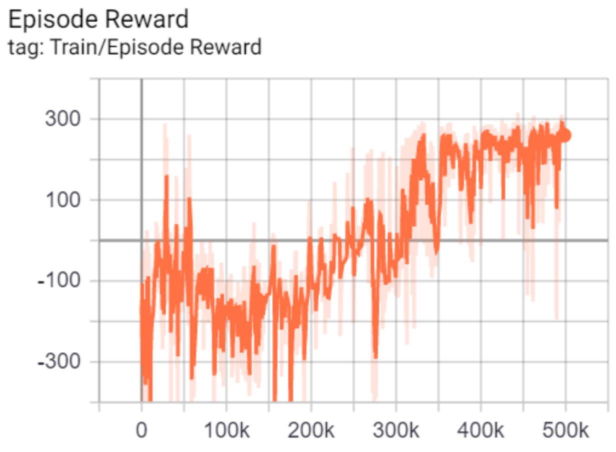
September 2020

1 Tensorboard Plot

The task of LunarLander-v2 is completing by using double deep Q-network (double-DQN). The task of LunarLanderContinuous-v2 is completing by using deep deterministic policy gradient (DDPG). Total training episodes are both 1200.



(a) LunarLander-v2



(b) LunarLanderContinuous-v2

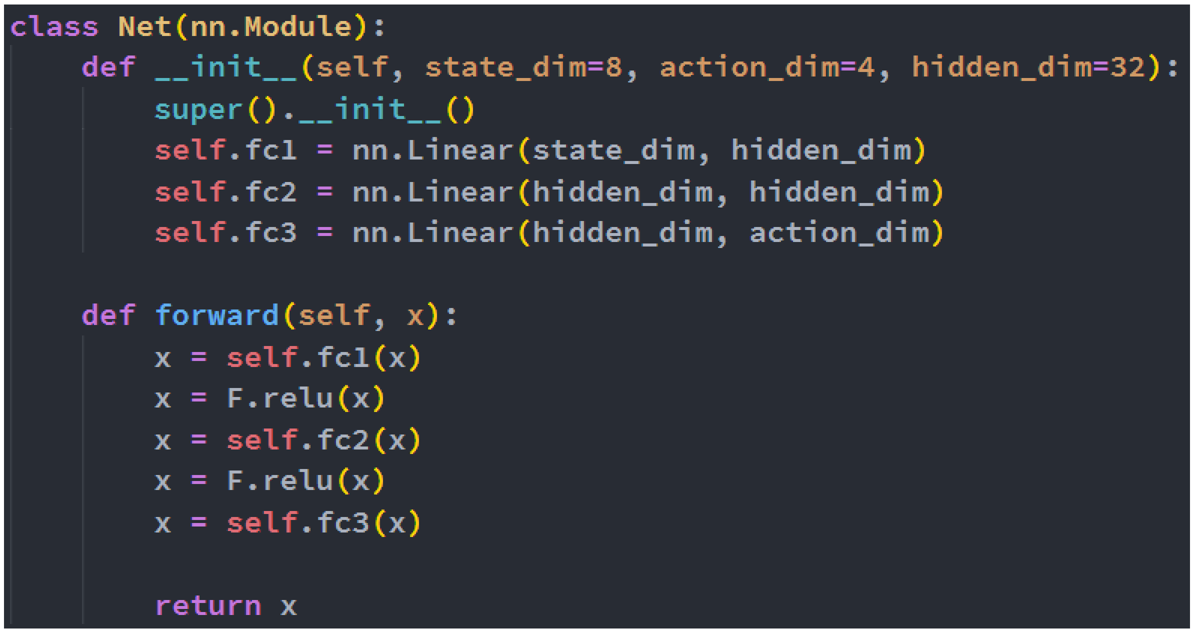
Figure 1: Training episode rewards.

2 Major Implementation

2.1 Network Structure

2.1.1 DQN/Double-DQN

For the network structure, I use 3 fully connected layers to construct it. The input of the network is a 8-dimension vector which indicates the state observation, and the output of the network is a 4-dimension vector which indicates the Q value of the 4 actions. The hidden size of the network is 32.

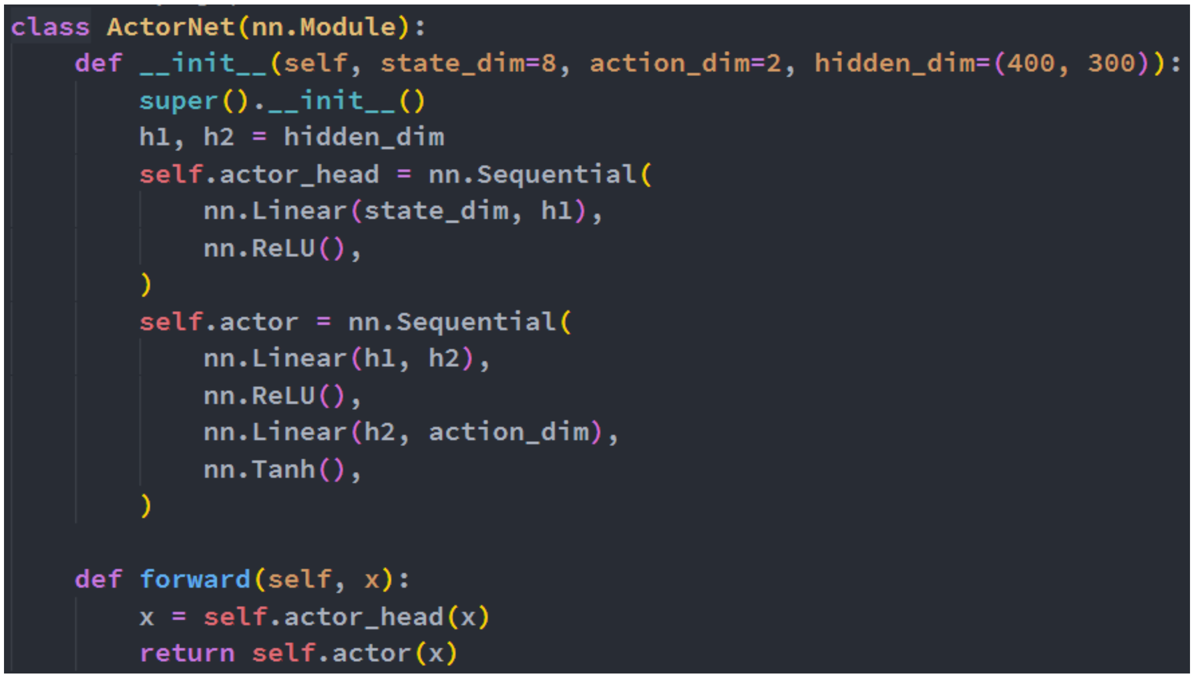


```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)

        return x
```

Figure 2: Code snippet of the network structure in DQN/double-DQN.

2.1.2 DDPG

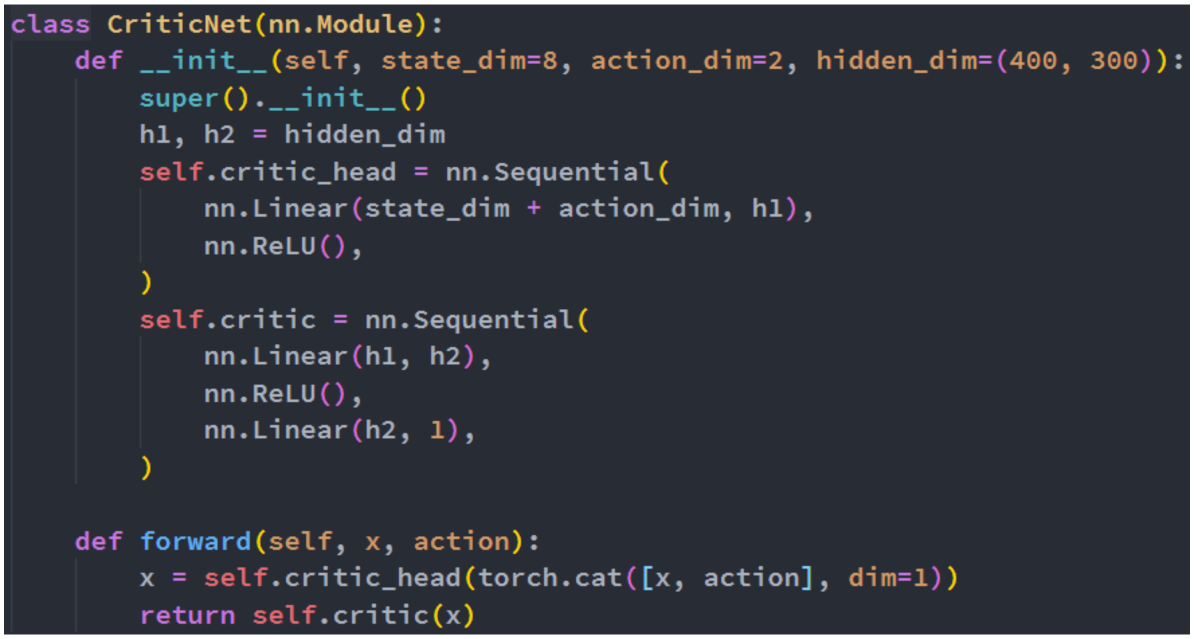
For the network structure of the actor, I use 3 fully connected layers to construct it. The input of the network is a 8-dimension vector which indicates the state observation, and the output of the network is a 2-dimension vector which indicates the value of the 2 actions it take. The hidden size of the hidden layers are 400 and 300, respectively.



```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.actor_head = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
        )
        self.actor = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, action_dim),
            nn.Tanh(),
        )

    def forward(self, x):
        x = self.actor_head(x)
        return self.actor(x)
```

Figure 3: Code snippet of the actor in DDPG.

For the network structure of the critic, I use 3 fully connected layers to construct it. The input of the network is a 10-dimension vector which is the concatenation of the state observation and the actions, and the output of the network is a scalar which indicates the Q value corresponds to the state and the action. The hidden size of the hidden layers are 400 and 300, respectively.



```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 4: Code snippet of the critic in DDPG.

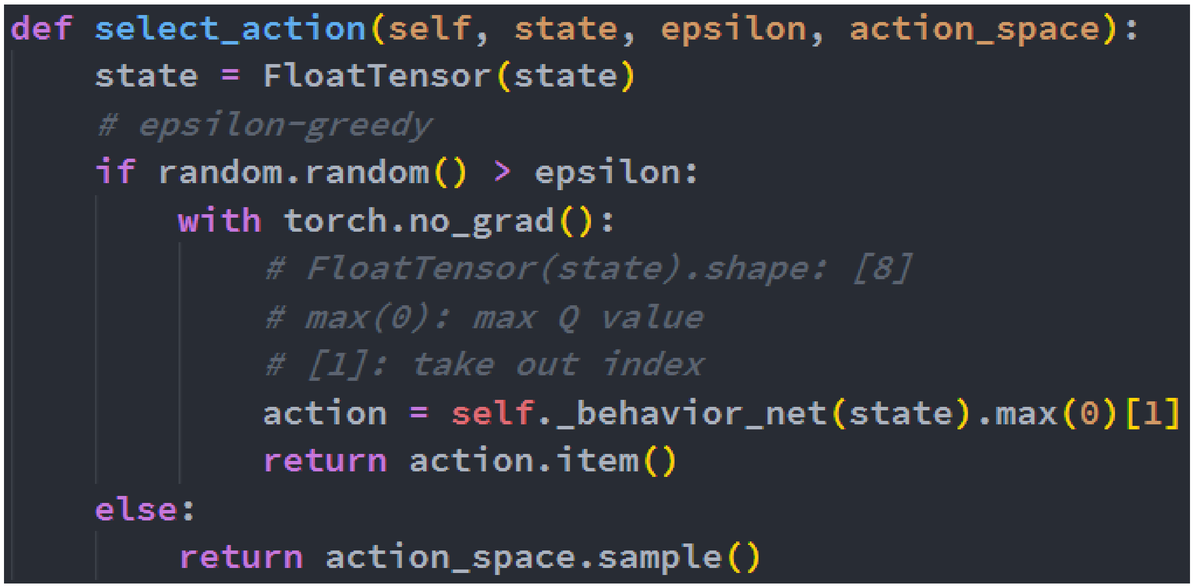
2.2 Training Process

The training process of DQN/double-DQN and DDPG are similar. For each step in an episode, the agent will first select an action based on the current state, execute it to obtain the next state, and then store the transition $(s_t, a_t, r_t, s_{t+1}, done)$ to the replay buffer. After that, the networks will be updated according to the condition, then go into next step until the episode ends. The implementation of the training functions are the same as the sample code that TA provides. The detail of the action selection will be described in the section 2.3, and the detail of the update of the network will be described in section 2.4.

2.3 Action Selection

2.3.1 DQN/Double-DQN

The action selection in DQN/double-DQN is based on epsilon-greedy approach. First, a number in range [0.0, 1.0) is randomly generated. If the number is greater than the epsilon value, the action which can maximize the Q value output from the behavior network will be chosen. Otherwise, the action will be randomly chosen in order to do more exploration.

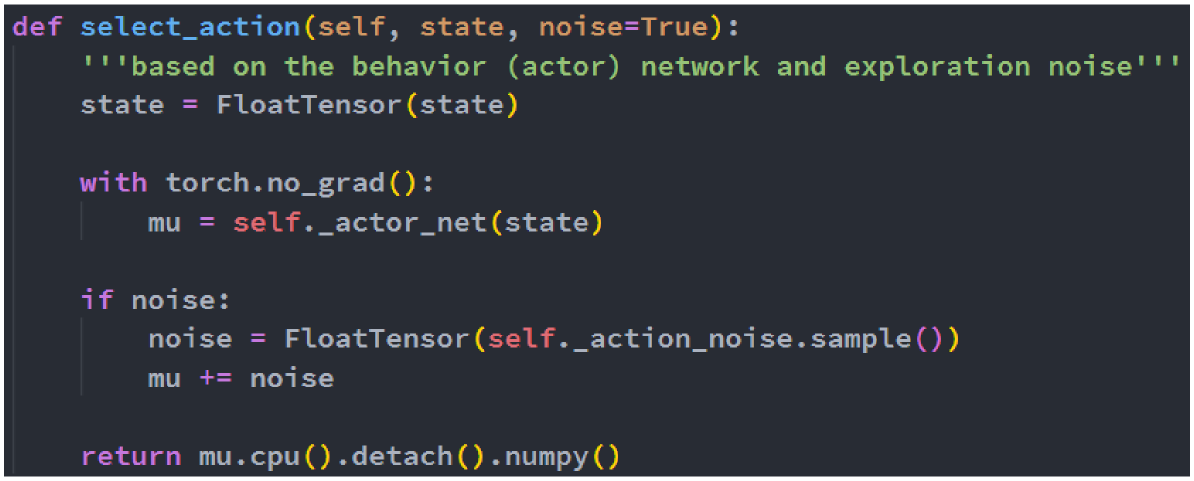


```python
def select_action(self, state, epsilon, action_space):
    state = FloatTensor(state)
    # epsilon-greedy
    if random.random() > epsilon:
        with torch.no_grad():
            # FloatTensor(state).shape: [8]
            # max(0): max Q value
            # [1]: take out index
            action = self._behavior_net(state).max(0)[1]
            return action.item()
    else:
        return action_space.sample()
```

Figure 5: Code snippet of action selection in DQN/double-DQN.

2.3.2 DDPG

The action selection in DDPG is performed by the actor network. First, the values of the two actions will be decided by inputting the state into the actor net. Then, in order to do more exploration, a Gaussian noise is sampled and added to the selected action.



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

    with torch.no_grad():
        mu = self._actor_net(state)

    if noise:
        noise = FloatTensor(self._action_noise.sample())
        mu += noise

    return mu.cpu().detach().numpy()
```

Figure 6: Code snippet of action selection in DDPG.

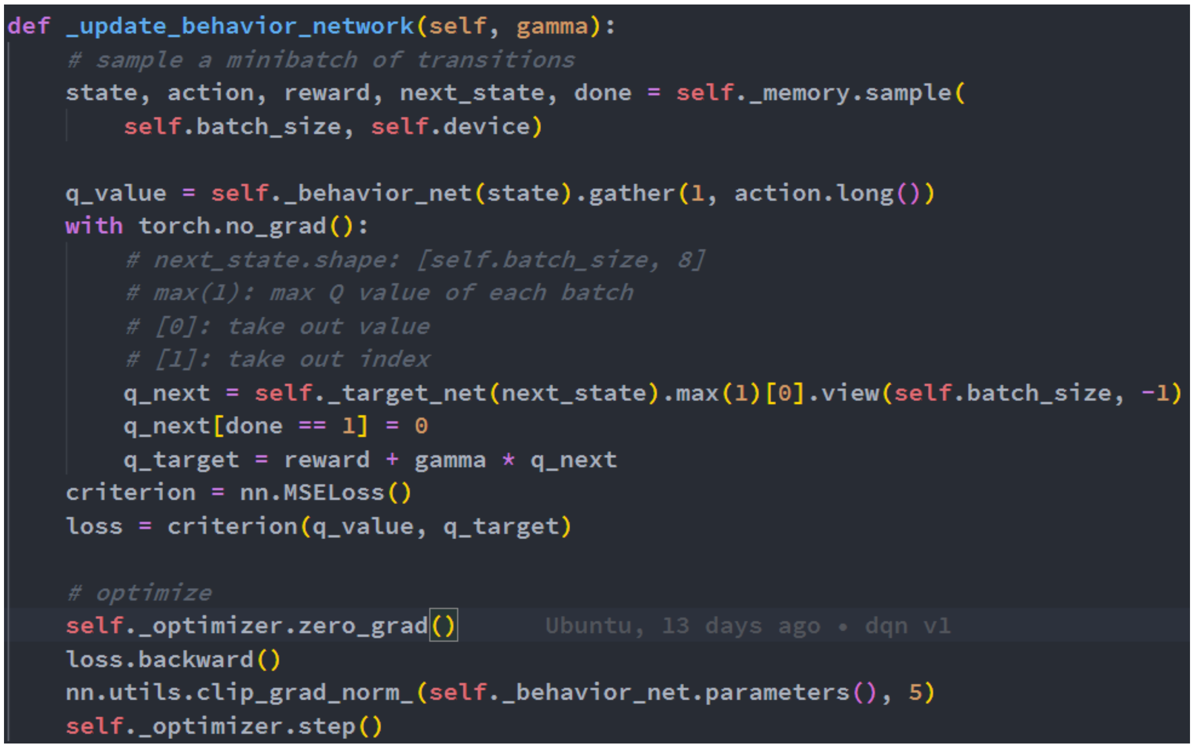
2.4 Network Update

2.4.1 DQN

In DQN, I update the behavior network every 4 iterations, and update the target network every 100 iterations.

For behavior network's update, first a batch of transition will be sampled form the replay buffer, then based on the state s_t and the action a_t, I can get the Q value q_value from the output of the behavior network. Next, I use next state s_{t+1} as target network's input and select the max Q value output from it as q_next, then calculate target Q value by $q\_target = gamma * q\_next$. Finally, I calculate the loss by the MSE between q_value and q_target and then update the behavior network. Notice that for the q_target, if the episode is done, I will set it to zero.

For target network's update, I simply copy the weight from the behavior network.

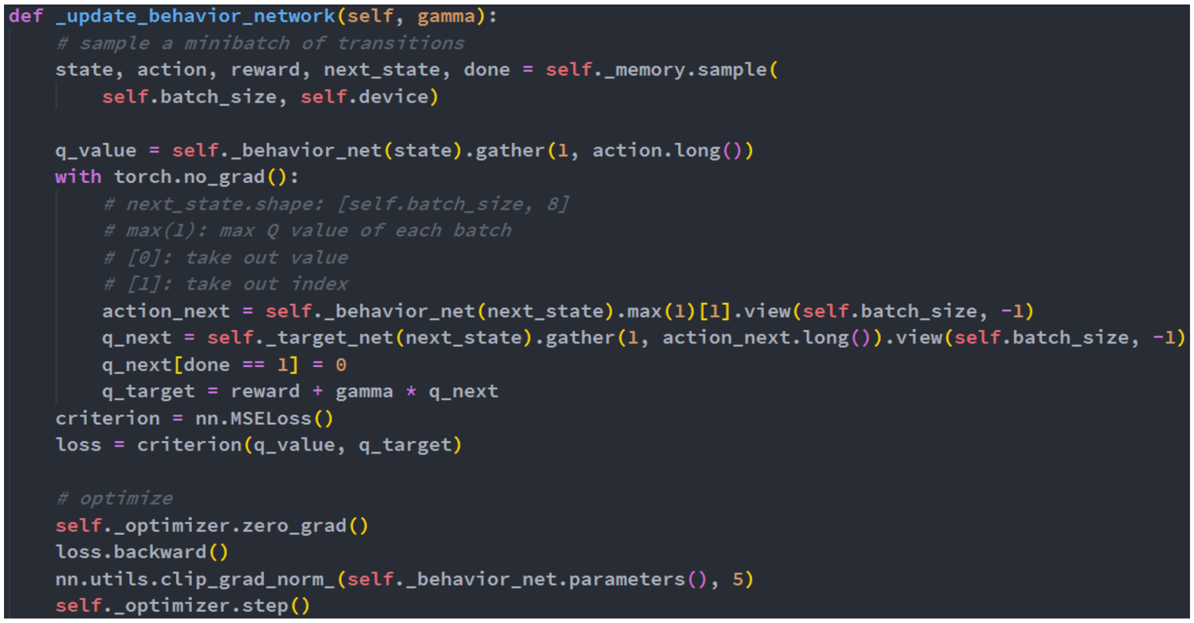


```python
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        # next_state.shape: [self.batch_size, 8]
        # max(1): max Q value of each batch
        # [0]: take out value
        # [1]: take out index
        q_next = self._target_net(next_state).max(1)[0].view(self.batch_size, -1)
        q_next[done == 1] = 0
        q_target = reward + gamma * q_next
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

    # optimize
    self._optimizer.zero_grad()        Ubuntu, 13 days ago • dqn v1
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

Figure 7: Code snippet of network update in DQN.

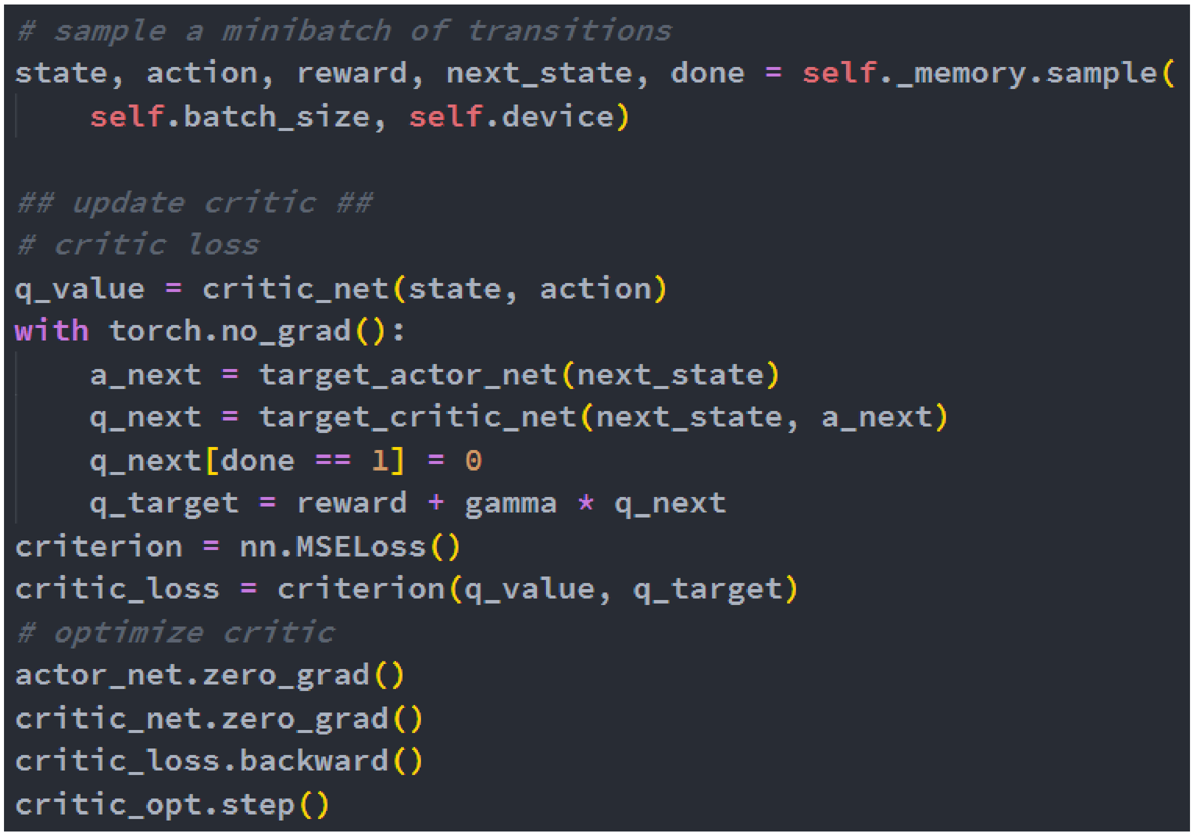
2.4.2 Double-DQN

In double-DQN, almost every procedures are the same as DQN, but the way to obtain q_next is different. For calculating q_next, first I use next state s_{t+1} as behavior network's input, and select the action which can maximize the output of the behavior network. Then I use next state s_{t+1} as taget network's input, and select q_next according to the action obtained previously.


```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)

    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        # next_state.shape: [self.batch_size, 8]
        # max(1): max Q value of each batch
        # [0]: take out value
        # [1]: take out index
        action_next = self._behavior_net(next_state).max(1)[1].view(self.batch_size, -1)
        q_next = self._target_net(next_state).gather(1, action_next.long()).view(self.batch_size, -1)
        q_next[done == 1] = 0
        q_target = reward + 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()
```
Figure 8: Code snippet of network update in double-DQN.

2.4.3 DDPG

Critic Updating For behavior critic network's update, first a batch of transition will be sampled from the replay buffer, then based on the state s_t and the action a_t, I can get the Q value q_value form the output of the behavior critic network. Next, I use next state s_{t+1} as target actor network's input to get the action $action\_next$, and obtain the Q value q_next by input s_{t+1} and $action\_next$ to the target critic network. Then I calculate target Q value by $q\_target = gamma * q\_next$. Finally, I calculate the loss by the MSE between q_value and q_target and then update the behavior critic network. Notice that for the q_target, if the episode is done, I will set it to zero.



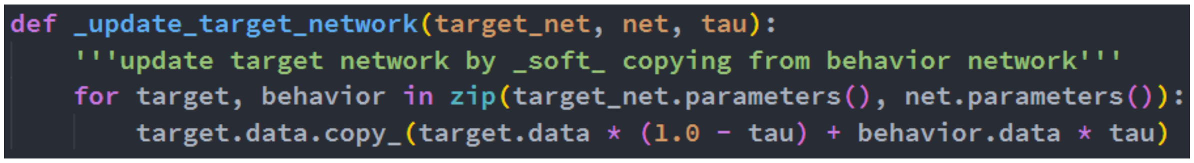
```python
# sample a minibatch of transitions
state, action, reward, next_state, done = self._memory.sample(
    self.batch_size, self.device)

## update critic ##
# critic loss
q_value = critic_net(state, action)
with torch.no_grad():
    a_next = target_actor_net(next_state)
    q_next = target_critic_net(next_state, a_next)
    q_next[done == 1] = 0
    q_target = reward + gamma * q_next
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()
```

Figure 9: Code snippet of behavior critic network's update in DDPG.

For target critic network's update, I apply soft copy from the behavior network. The weights of the target network can be calculated by:

$$target = target * (1 - \tau) + behavior * \tau$$

where τ is set to 0.005.



```python
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()):
        target.data.copy_(target.data * (1.0 - tau) + behavior.data * tau)
```

Figure 10: Code snippet of target network's update in DDPG.

Actor Updating For behavior actor network's update, I input s_t to the behavior actor network to get the action. Next, I input the state s_t and the action obtained previously to the behavior critic network, then calculate the loss by maximizing the output of it and update the actor.

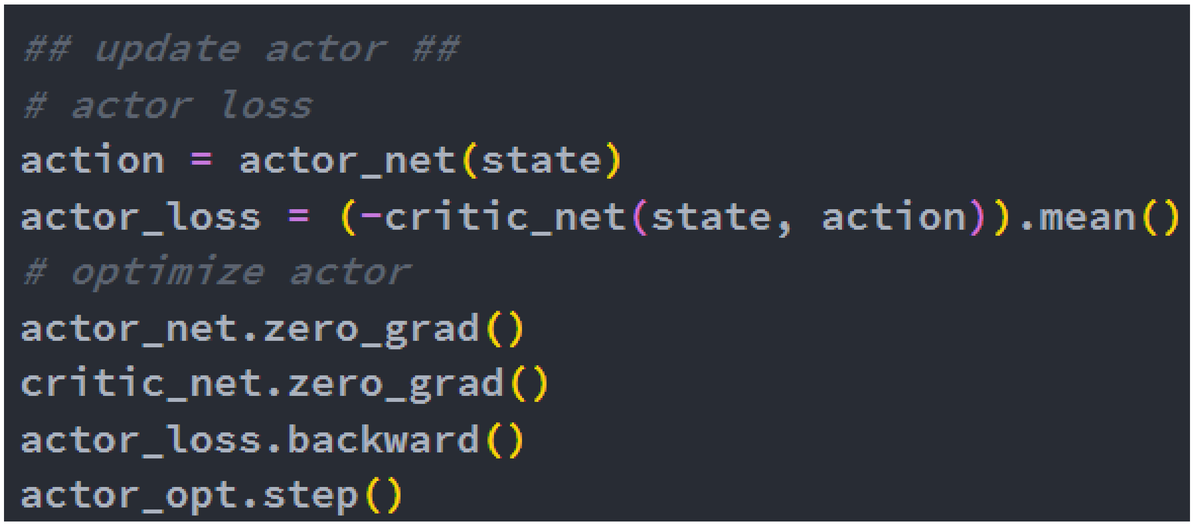

```
## update actor ##
# actor loss
action = actor_net(state)
actor_loss = (-critic_net(state, action)).mean()
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

Figure 11: Code snippet of behavior actor network's update in DDPG.

For target actor network's update, the procedures are the same as the update of target critic network.

3 Difference Between Implementation and Algorithm

3.1 DQN

First, in the algorithm, the input of the Q network are state and action, and then output a scalar as Q value. In the implementation, I only input state into Q network, but output a 4-dimension vector which indicates the Q value corresponding to each action.

Next, we store an additional variable *done* into the replay buffer. Finally, the behavior network doesn't update every iteration, but every 4 iterations.

3.2 DDPG

First, the gradients used to update behavior actor network are different. In the algorithm, the actor is updated using the sampled gradient, while in the implementation the actor is updated by maximizing the critic's output.

Next, as mentioned in DQN, we store an additional variable *done* into the replay buffer.

4 Discussion

4.1 Effects of the Discount Factor

The discount factor is a measure that tells how important rewards are to the current state. A discount factor closer to one means that the agent should also take distant future rewards into accounts. A discount factor closer to zero on the other hand indicates that only rewards in the immediate future are being considered.

Since LunarLander is a game that gives the rewards mainly depending on the final result, the agent should consider the cumulative future reward with higher weight, so the discount factor is set to 0.99. If the discount factor is too small, the agent will only try to maximize the current reward while updating the parameters.

4.2 Benefits of Epsilon-Greedy In Comparison To Greedy Action Selection

With epsilon-greedy, the agent can choose the best action depending on the Q value, and also have a probability to explore more states simultaneously. If we only use greedy action selection, the agent will only choose action based on its history rewards. But if there are better selection that haven't been explore before, it won't have the chance to choose them, which may lead to lower performance.

4.3 Necessity of the Target Network

If we use same Q networks during training, since the model will change after updating the parameters, this will make the model keep chasing a moving target, which make the training unstable. So in order to make training stable, we use an additional target Q network which is fixed and only update the behavior Q network, and this approach can avoid the problem of chasing a moving target.

4.4 Effects of Replay Buffer Size

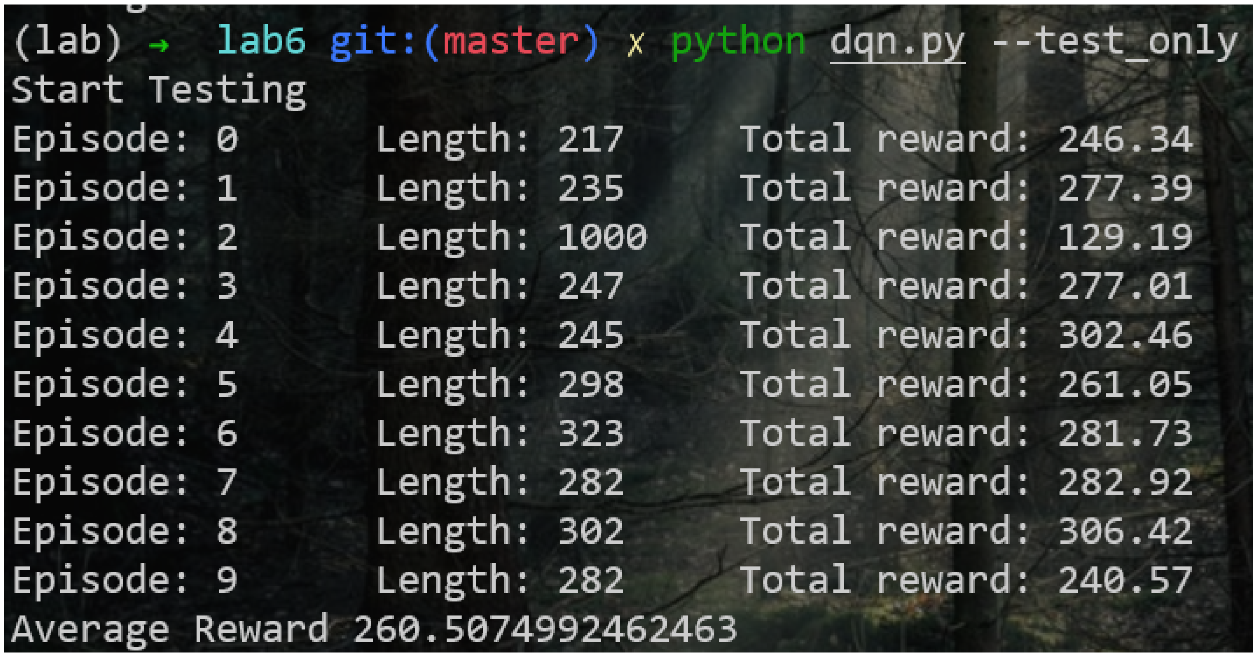
Since the training data is not independent to each other in an episode, we store transitions and randomly sample a mini-batch data for training, which can decorrelate the training data.

If the buffer size is too small, we will have a higher chance to sample correlated elements, which might cause the model to overfit the recent transitions.

On the other hands, if the buffer size is too large, the whole process may require a larger memory and it might slower the training. Also, we will have a higher chance to sample older transitions, which may not be the target we want the model to learn.

5 Performance

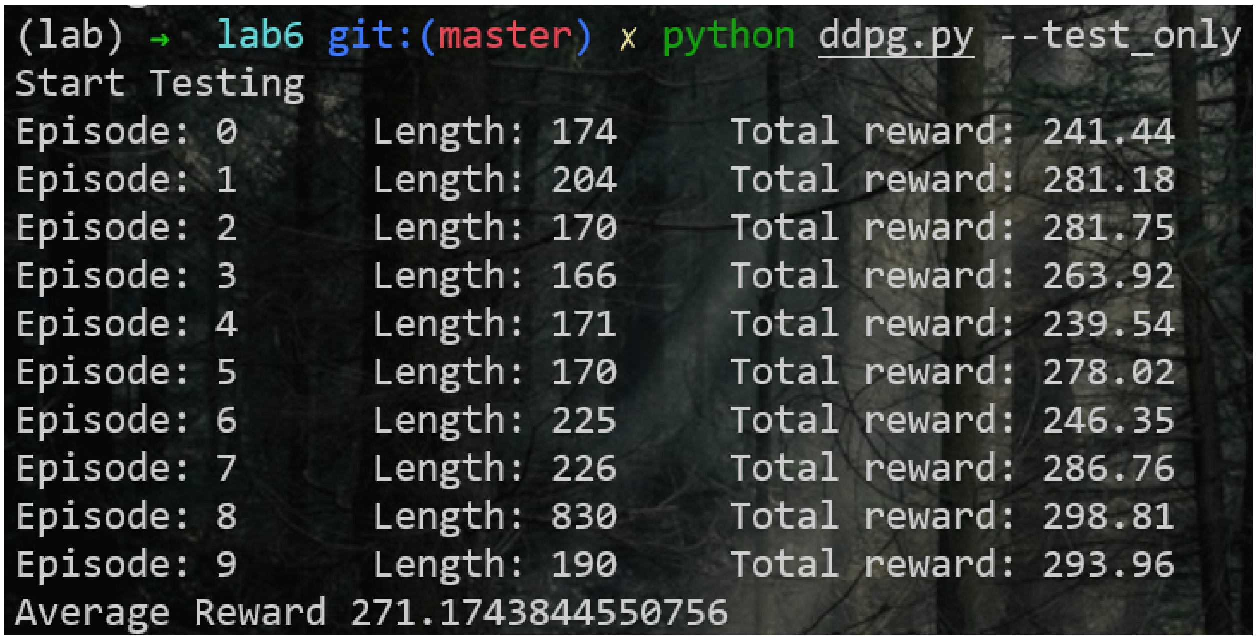
5.1 LunarLander-v2



```
(lab) → lab6 git:(master) ✗ python dqn.py --test_only
Start Testing
Episode: 0      Length: 217     Total reward: 246.34
Episode: 1      Length: 235     Total reward: 277.39
Episode: 2      Length: 1000    Total reward: 129.19
Episode: 3      Length: 247     Total reward: 277.01
Episode: 4      Length: 245     Total reward: 302.46
Episode: 5      Length: 298     Total reward: 261.05
Episode: 6      Length: 323     Total reward: 281.73
Episode: 7      Length: 282     Total reward: 282.92
Episode: 8      Length: 302     Total reward: 306.42
Episode: 9      Length: 282     Total reward: 240.57
Average Reward 260.5074992462463
```

Figure 12: Testing result of LunarLander-v2.

5.2 LunarLanderContinuous-v2



```
(lab) → lab6 git:(master) ✗ python ddpg.py --test_only
Start Testing
Episode: 0      Length: 174     Total reward: 241.44
Episode: 1      Length: 204     Total reward: 281.18
Episode: 2      Length: 170     Total reward: 281.75
Episode: 3      Length: 166     Total reward: 263.92
Episode: 4      Length: 171     Total reward: 239.54
Episode: 5      Length: 170     Total reward: 278.02
Episode: 6      Length: 225     Total reward: 246.35
Episode: 7      Length: 226     Total reward: 286.76
Episode: 8      Length: 830     Total reward: 298.81
Episode: 9      Length: 190     Total reward: 293.96
Average Reward 271.1743844550756
```

Figure 13: Testing result of LunarLanderContinuous-v2.