Lab6: DQN-DDPG

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Experimental results

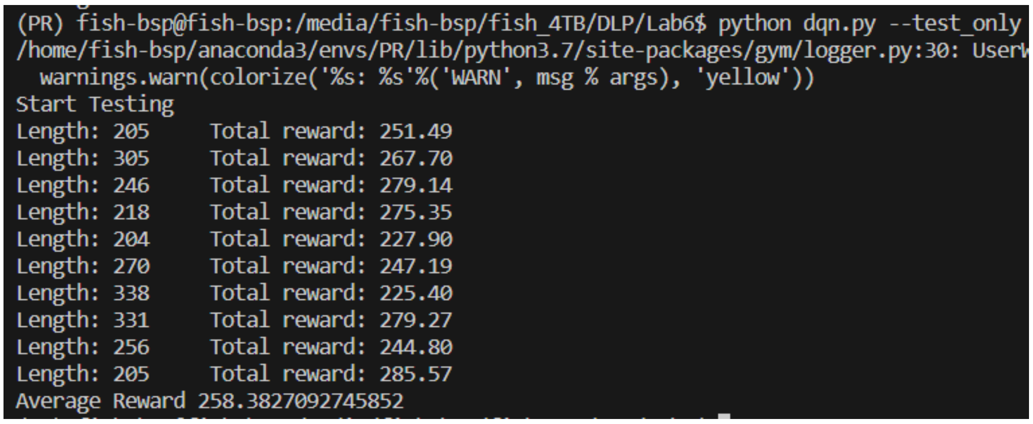
A. Your screenshot of tensorboard and testing results on LunarLander-v2.

A-1. Experiment setup

The hyper-parameters for experiments as shown below.

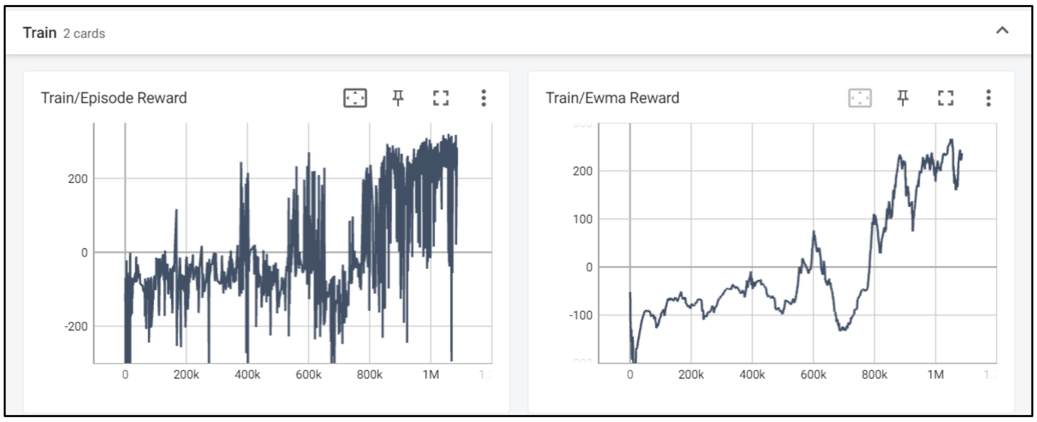
- episode: 2000
- warmup: 10000
- capacity: 10000
- batch size: 128
- learning rate: 0.0005
- eps_decay: 0.995
- eps_min: 0.01
- gamma: 0.99
- freq: 4
- target_freq: 1000

A-2. Testing results



```
(PR) fish-bsp@fish-bsp:/media/fish-bsp/fish_4TB/DLP/Lab6$ python dqn.py --test_only
/home/fish-bsp/anaconda3/envs/PR/lib/python3.7/site-packages/gym/logger.py:30: UserW
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
Start Testing
Length: 205     Total reward: 251.49
Length: 305     Total reward: 267.70
Length: 246     Total reward: 279.14
Length: 218     Total reward: 275.35
Length: 204     Total reward: 227.90
Length: 270     Total reward: 247.19
Length: 338     Total reward: 225.40
Length: 331     Total reward: 279.27
Length: 256     Total reward: 244.80
Length: 205     Total reward: 285.57
Average Reward 258.3827092745852
```

A-3. Tensorboard



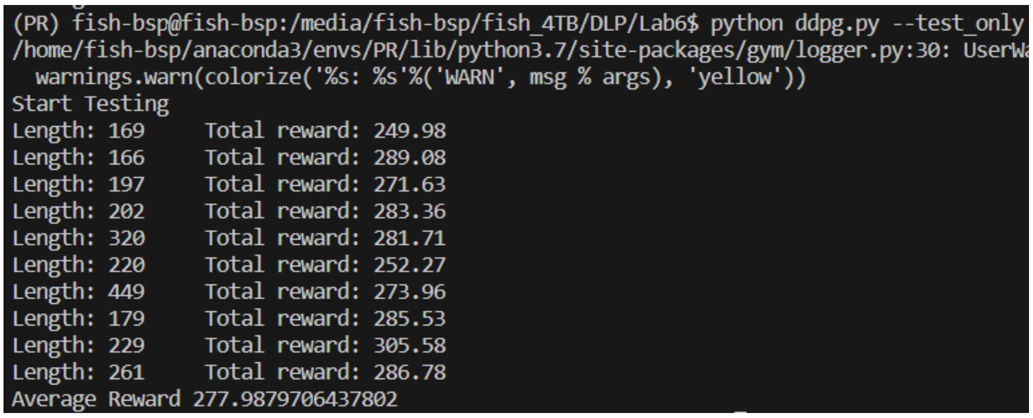
B. Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2.

B-1. Experiment setup

The hyper-parameters for experiments as shown below.

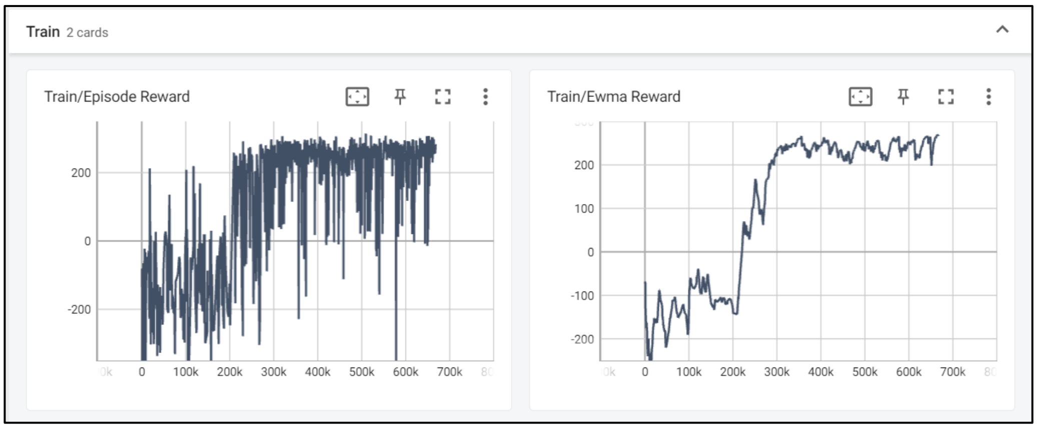
- episode: 2000
- warmup: 10000
- capacity: 500000
- batch size: 64
- learning rate (critic): 0.001
- learning rate (action): 0.001
- gamma: 0.99
- tau: 0.005

B-2. Testing results



```
(PR) fish-bsp@fish-bsp:/media/fish-bsp/fish_4TB/DLP/Lab6$ python ddpg.py --test_only
/home/fish-bsp/anaconda3/envs/PR/lib/python3.7/site-packages/gym/logger.py:30: UserWa
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
Start Testing
Length: 169     Total reward: 249.98
Length: 166     Total reward: 289.08
Length: 197     Total reward: 271.63
Length: 202     Total reward: 283.36
Length: 320     Total reward: 281.71
Length: 220     Total reward: 252.27
Length: 449     Total reward: 273.96
Length: 179     Total reward: 285.53
Length: 229     Total reward: 305.58
Length: 261     Total reward: 286.78
Average Reward 277.9879706437802
```

B-3. Tensorboard



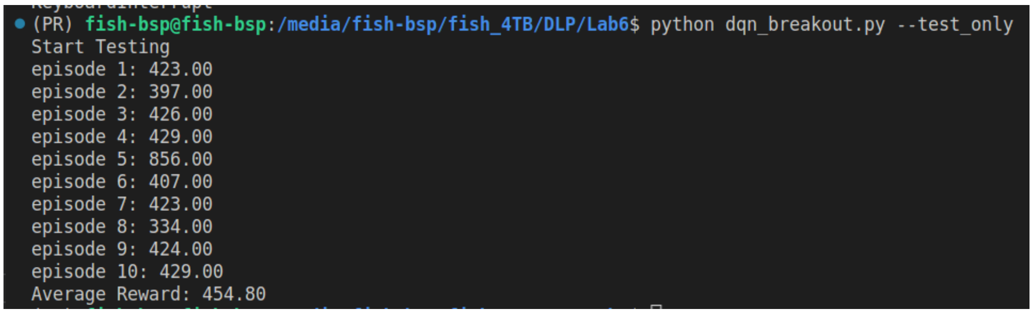
C. Your screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4.

C-1. Experiment setup

The hyper-parameters for experiments as shown below.

- episode: 100000
- warmup: 20000
- capacity: 100000
- batch size: 32
- learning rate: 0.0000625
- eps_decay: 1000000
- eps_min: 0.1
- gamma: 0.99
- freq: 4
- target_freq: 10000
- eval_freq: 20000
- test_epsilon: 0.01

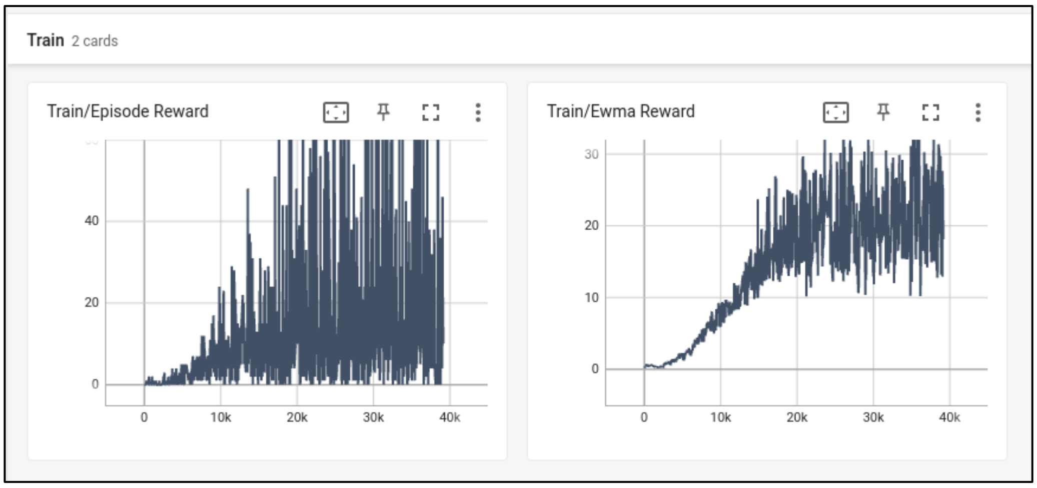
C-2. Testing results



```
(PR) fish-bsp@fish-bsp:/media/fish-bsp/fish_4TB/DLP/Lab6$ python dqn_breakout.py --test_only
Start Testing
episode 1: 423.00
episode 2: 397.00
episode 3: 426.00
episode 4: 429.00
episode 5: 856.00
episode 6: 407.00
episode 7: 423.00
episode 8: 334.00
episode 9: 424.00
episode 10: 429.00
Average Reward: 454.80
```

C-3. Tensorboard

From the previous description, we said that we set episode to 100K, but why the total steps in tensorboard have only about 40K. When training this task, we observed the virtual memory would explode, so we saved the model weights from before the situation of out of memory occurs, and then load the pretrained weights into a new training phase. Repeat the above steps until it achieved the expected result.



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

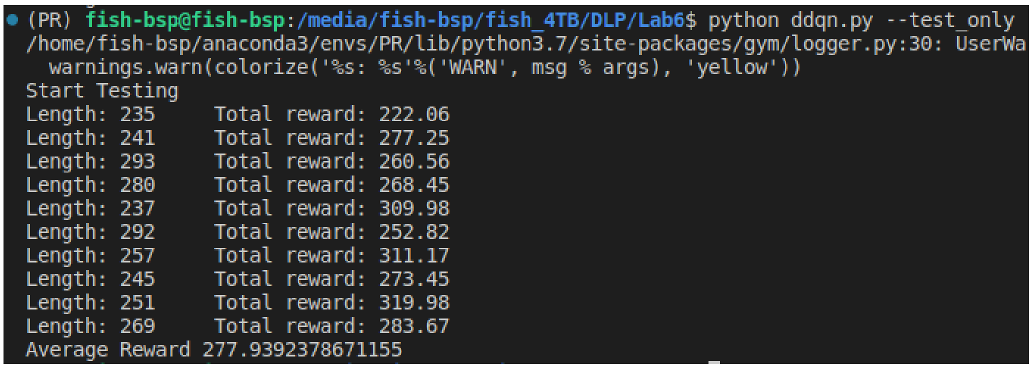
A. Your screenshot of tensorboard and testing results on LunarLander-v2 using DDQN.

A-1. Experiment setup

The hyper-parameters for experiments as shown below.

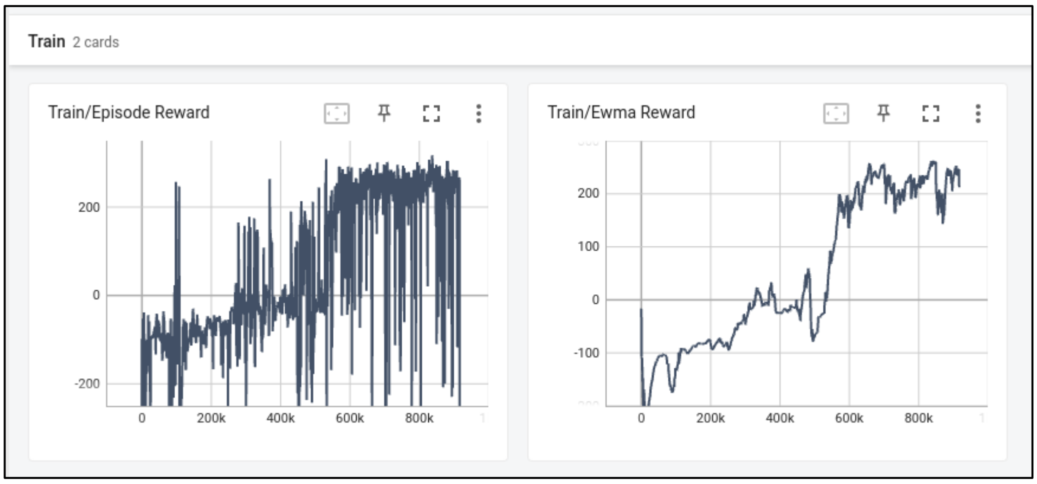
- episode: 2000
- warmup: 10000
- capacity: 10000
- batch size: 128
- learning rate: 0.0005
- eps_decay: 0.995
- eps_min: 0.01
- gamma: 0.99
- freq: 4
- target_freq: 1000

A-2. Testing results



```
● (PR) fish-bsp@fish-bsp:/media/fish-bsp/fish_4TB/DLP/Lab6$ python ddqn.py --test_only
  /home/fish-bsp/anaconda3/envs/PR/lib/python3.7/site-packages/gym/logger.py:30: UserWa
    warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
  Start Testing
  Length: 235     Total reward: 222.06
  Length: 241     Total reward: 277.25
  Length: 293     Total reward: 260.56
  Length: 280     Total reward: 268.45
  Length: 237     Total reward: 309.98
  Length: 292     Total reward: 252.82
  Length: 257     Total reward: 311.17
  Length: 245     Total reward: 273.45
  Length: 251     Total reward: 319.98
  Length: 269     Total reward: 283.67
  Average Reward 277.9392378671155
```

A-3. Tensorboard



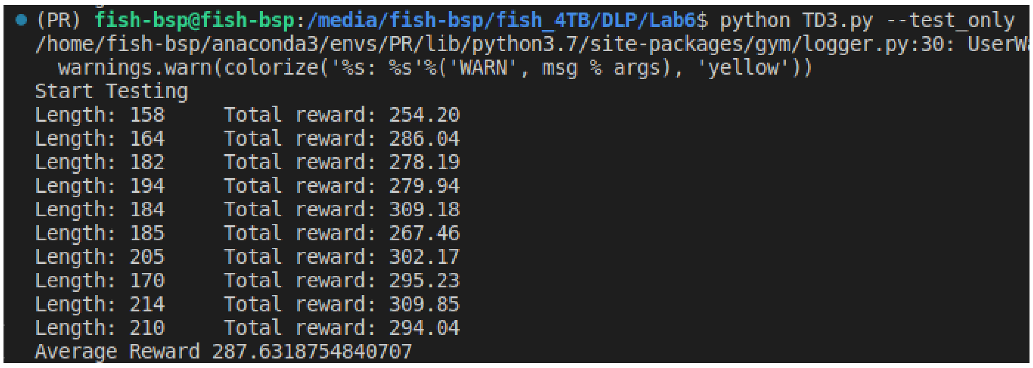
B. Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2 using TD3.

B-1. Experiment setup

The hyper-parameters for experiments as shown below.

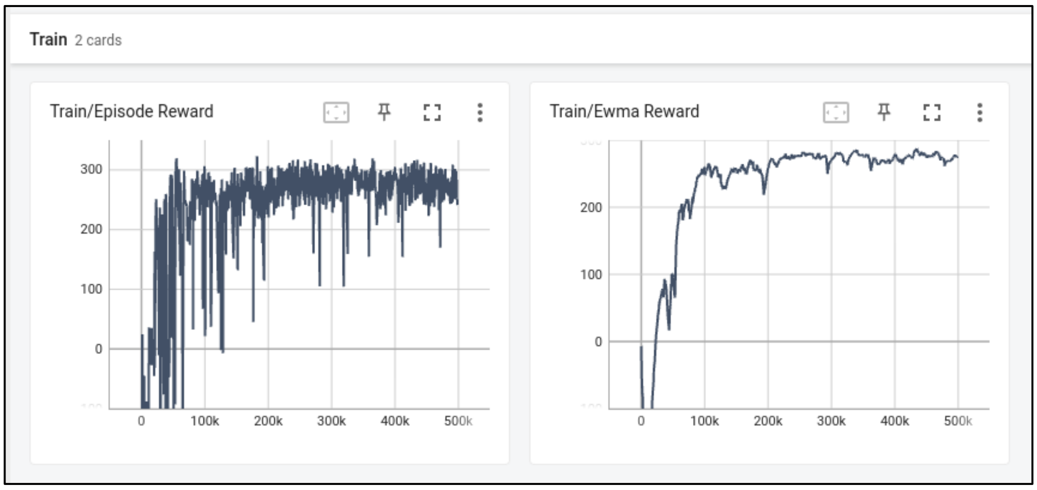
- episode: 2000
- warmup: 10000
- capacity: 500000
- batch size: 64
- learning rate (critic): 0.001
- learning rate (action): 0.001
- gamma: 0.99
- tau: 0.005

B-2. Testing results



```
● (PR) fish-bsp@fish-bsp:/media/fish-bsp/fish_4TB/DLP/Lab6$ python TD3.py --test_only
  /home/fish-bsp/anaconda3/envs/PR/lib/python3.7/site-packages/gym/logger.py:30: UserW
    warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
  Start Testing
  Length: 158     Total reward: 254.20
  Length: 164     Total reward: 286.04
  Length: 182     Total reward: 278.19
  Length: 194     Total reward: 279.94
  Length: 184     Total reward: 309.18
  Length: 185     Total reward: 267.46
  Length: 205     Total reward: 302.17
  Length: 170     Total reward: 295.23
  Length: 214     Total reward: 309.85
  Length: 210     Total reward: 294.04
  Average Reward 287.6318754840707
```

B-3. Tensorboard

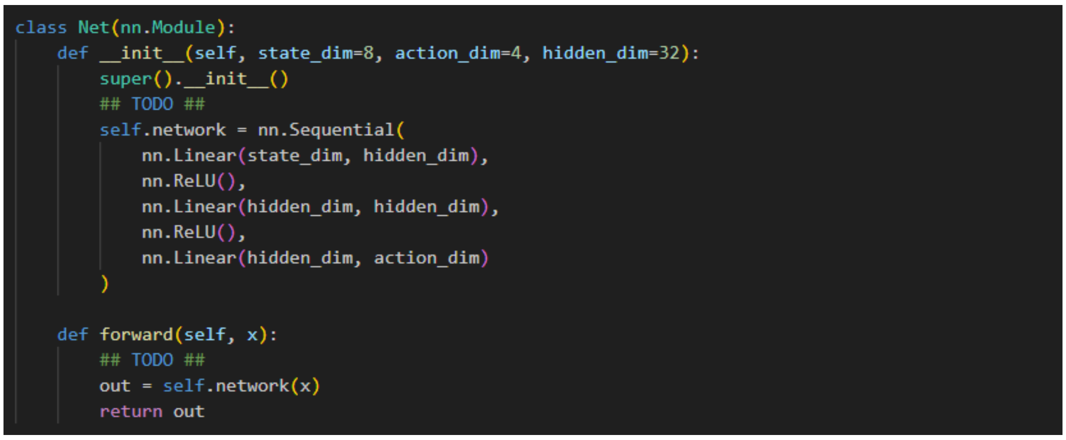


3. Questions

A. Describe your major implementation of both DQN and DDPG in detail. Your description should at least contain three parts

A-1. Your implementation of Q network updating in DQN.

In this section, we would divided DQN into two parts, behavior network and target network. Basically, the two networks have the similar architectures, we design the network contains the input layer, output layer and one hidden layer. The implementation of the network architecture is as follows.



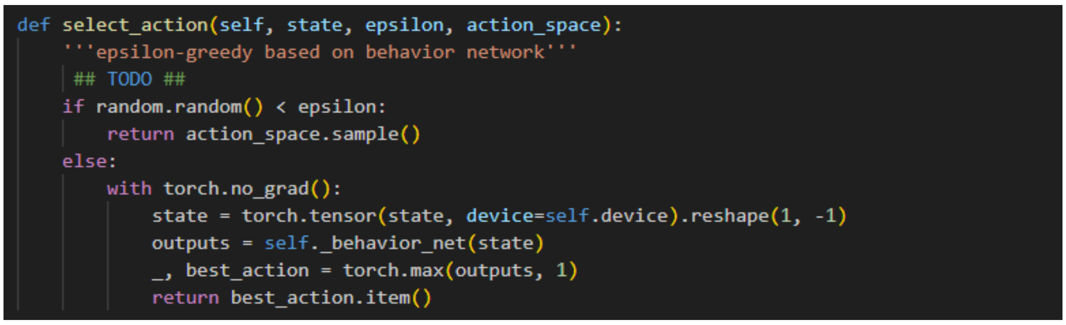
```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
        )

    def forward(self, x):
        ## TODO ##
        out = self.network(x)
        return out
```

About DQN algorithms, we divided it into main three parts, select action, updating behavior network and updating target network.

- First part – select action

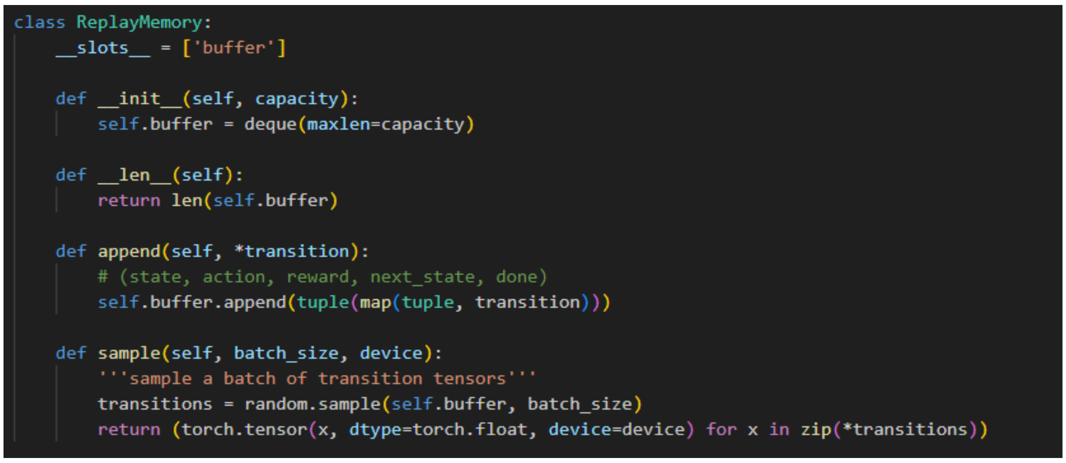
As we can see that we choose the best action with random ϵ or the one with the largest q-value which was obtained by the behavior network. For ϵ, we set it to 1 in default and it will decay after warm-up step by 0.995 until ϵ becomes 0.01.

The process mentioned above is the so-called greedy algorithm.



```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            state = torch.tensor(state, device=self.device).reshape(1, -1)
            outputs = self._behavior_net(state)
            _, best_action = torch.max(outputs, 1)
            return best_action.item()
```

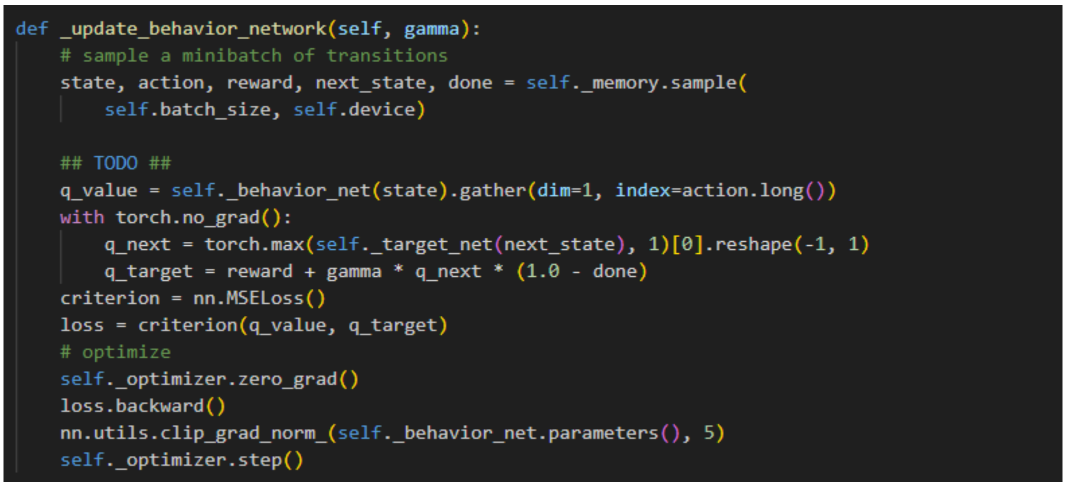
Then, we fed the chosen action to the environment, and the next step, reward and the status will return by the environment. These information was formed to a transition and stored in the replay memory. The implementation of replay memory is as follows.



```python
class ReplayMemory:
    __slots__ = ['buffer']

    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, *transition):
        # (state, action, reward, next_state, done)
        self.buffer.append(tuple(map(tuple, transition)))

    def sample(self, batch_size, device):
        '''sample a batch of transition tensors'''
        transitions = random.sample(self.buffer, batch_size)
        return (torch.tensor(x, dtype=torch.float, device=device) for x in zip(*transitions))
```

- Second part – update behavior network

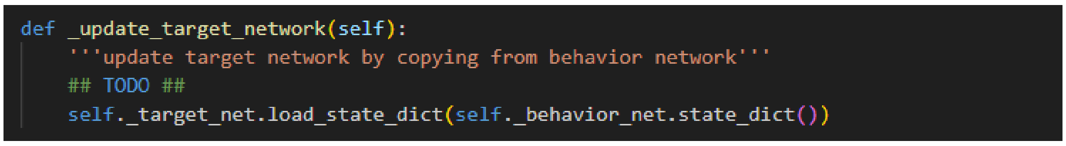
We sample random mini-batch transitions from replay memory in the training phase and calculate the q-value by behavior network using greedy algorithm. Then, we also calculate the next q-value in the same way to get the loss. Here, MSE loss was applied to compare the q-value and the next q-value. The implementation of updating the gradient in the behavior network is as follows.



```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 = self._behavior_net(state).gather(dim=1, index=action.long())
    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), 1)[0].reshape(-1, 1)
        q_target = reward + gamma * q_next * (1.0 - 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()
```

- Third part – update target network

The function reset the target network by copying from the behavior network in every *C* steps.

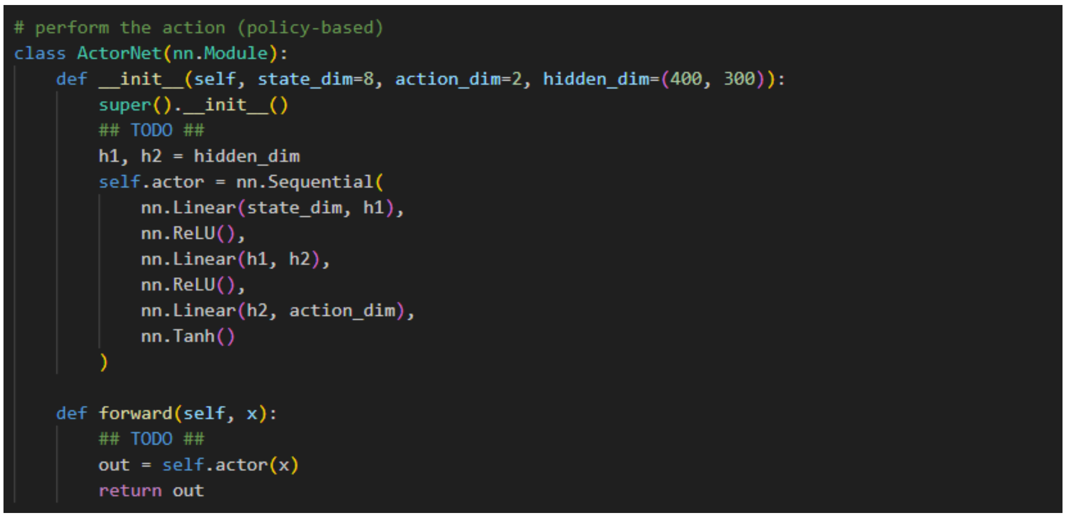


```python
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

Therefore, the architecture of the behavior network and the target network is the same, but the purposes of them are different. The behavior network is the network that actively interacts with the environment and learns to approximate q-values, and the target network provides a stable target for the q-value updates and helps to improve the stability of the training phase.

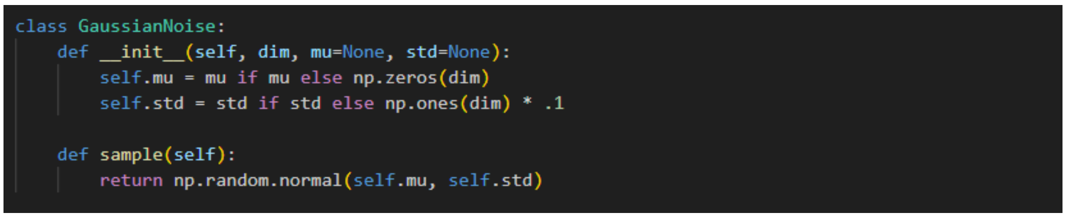
A-2. Your implementation and the gradient of actor updating in DDPG.

DDPG can be divided into the policy-based network and value-based network. Let's start with the policy-based network, which is actor network, it is responsible for performing actions. The implementation of actor network architecture is as follows.

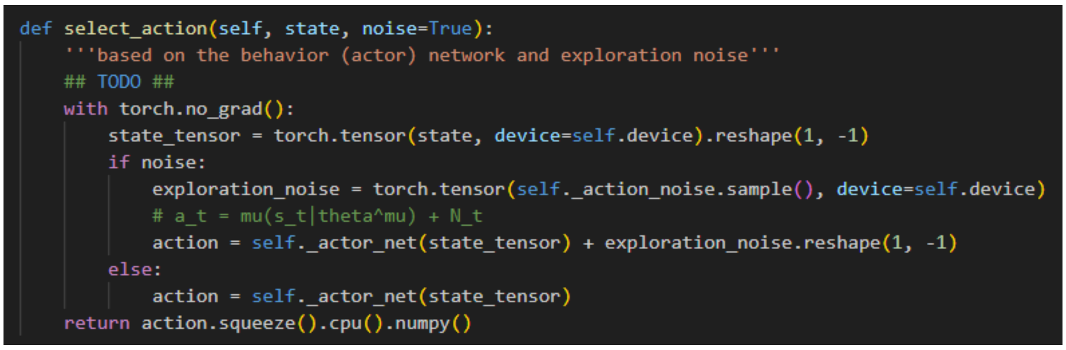


```python
# perform the action (policy-based)
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 = hidden_dim
        self.actor = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, action_dim),
            nn.Tanh()
        )

    def forward(self, x):
        ## TODO ##
        out = self.actor(x)
        return out
```

The behavior network of actor will select the deterministic action according to the current policy and Gaussian noise, this mechanism is defined as $a_t = \mu(s_t|\theta^u) + N_t$. In addition, actor network also has a target network with the same architecture but different parameters, which is used to update the value-based network Critic. Both networks are output actions. The implementation of actor network algorithm to select action is as follows.



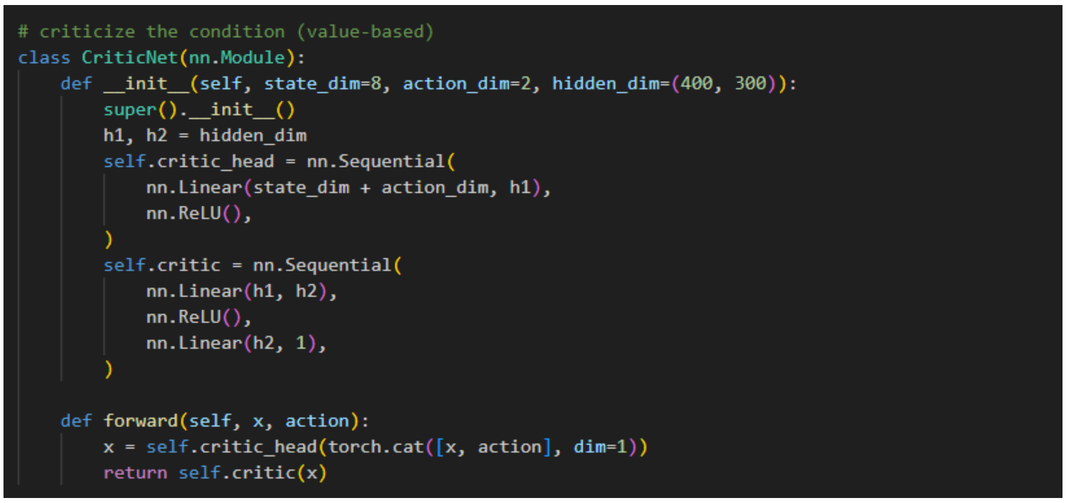
```python
class GaussianNoise:
    def __init__(self, dim, mu=None, std=None):
        self.mu = mu if mu else np.zeros(dim)
        self.std = std if std else np.ones(dim) * .1

    def sample(self):
        return np.random.normal(self.mu, self.std)
```



```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        state_tensor = torch.tensor(state, device=self.device).reshape(1, -1)
        if noise:
            exploration_noise = torch.tensor(self._action_noise.sample(), device=self.device)
            # a_t = mu(s_t|theta^mu) + N_t
            action = self._actor_net(state_tensor) + exploration_noise.reshape(1, -1)
        else:
            action = self._actor_net(state_tensor)
    return action.squeeze().cpu().numpy()
```

A-3. Your implementation and the gradient of critic updating in DDPG.

Critic network is responsible for evaluating the current situation. When we chose an action and fed it to the critic network, it evaluates q-value using the state and action from the actor network and how good the action is. The implementation of critic network architecture is as follows.



```python
# criticize the condition (value-based)
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)
```

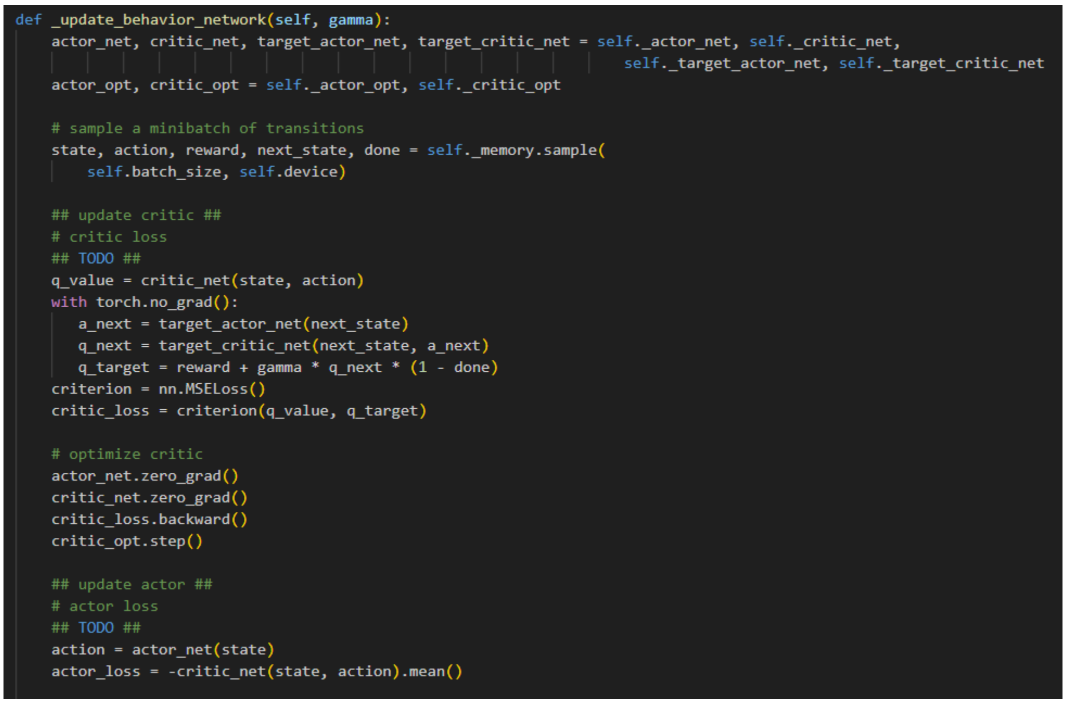
The training and updating processes are similar to that of DQN.

- First part – select action

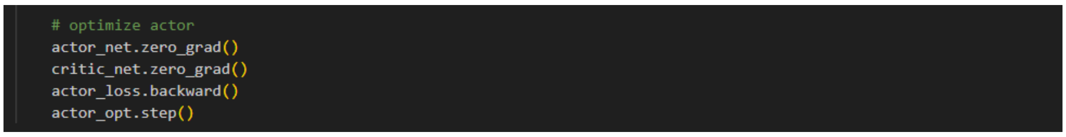
Compare with the algorithm of DQN, DDPG algorithm will select the action according to the current policy and Gaussian noise.

- Second part – update network

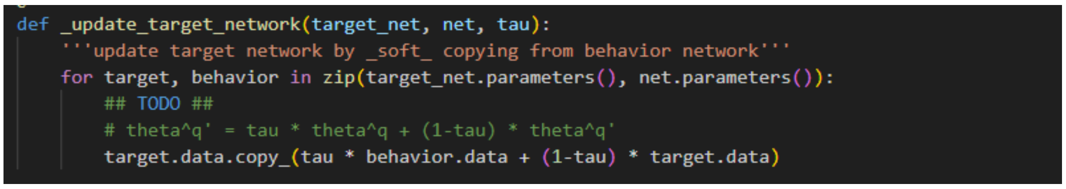
We sample random mini-batch transitions from replay memory in the training phase and calculate the q-value by critic behavior network. Then, we also select the next action using the next state directly by actor target network. The next q-value will be obtained based on the next state and the next action and computed by the critic target network. Finally, we can get the MSE loss to update the critic behavior network. The implementation of updating the gradient in the network is as follows.



```
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 = 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_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 = 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()
```



```
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 ##
        # theta^q' = tau * theta^q + (1-tau) * theta^q'
        target.data.copy_(tau * behavior.data + (1-tau) * target.data)
```

B. Explain effects of the discount factor.

The gamma in the updating behavior network function of previous sections is the discount factor. It determines the degree to which future rewards are valued. A higher discount factor (close to 1) emphasizes long-term rewards and encourages the agent to prioritize delayed rewards over immediate rewards. Delayed rewards refer to rewards that are received by an agent in the future. On the other hand, a lower discount factor (close to 0) places more importance on immediate rewards, leading the agent to focus on short-term gains. By adjusting the discount factor, we can control the balance between immediate and delayed rewards.

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

Epsilon-greedy ensures that the agent continues to explore different actions and states in the environment. By selecting a random action with a certain probability (epsilon), the agent can discover new and potentially optimal actions that may not have been selected otherwise. This exploration is crucial for discovering the optimal policy and avoiding getting stuck in suboptimal solutions.
While epsilon-greedy encourages exploration, it also ensures a certain level of exploitation. The greedy action selection (1 - epsilon) chooses the action with the highest estimated Q-value according to the current policy. This exploitation component allows the agent to focus on actions that have shown to be more rewarding in the past. It strikes a balance between exploring new actions and exploiting the knowledge gained so far.

D. Explain the necessity of the target network.

With Q-learning you are updating exactly one state/action value at each timestep, whereas with DQN or DDPG you are updating many, which you understand. The problem this causes is that you can affect the action values for the very next state you will be in instead of guaranteeing them to be stable as they are in Q-learning. This happens basically all the time with DQN, it will cause the “garbage in, garbage out” situation. So, we need a target network to stabilize our entire network. Conceptually it's like saying, "I have an idea of how to play this well, I'm going to try it out for a bit until I find something better" as opposed to saying "I'm going to retrain myself how to play this entire game after every move". By giving your network more time to consider many actions that have taken place recently instead of updating all the time, it hopefully finds a more robust model before you start using it to make actions.

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

The hint provided by TA in the spec is set episode_life=False, clip_rewards=False while testing. After we understand the meaning of these two parameters, if episode_life=False, we will have multiple lives, but this setting may not help the agent to learn the importance of losing lives. Therefore, we should set episode_life to True in the training phase which means only one life can be consumed per game. Compared with LunarLander, we think there is no significant difference for the overall code. However, in the Breakout case, we need to stack 4 last frames and the shape will be (84, 84, 4). We need to permute the shape to (batch_size, 4, 84,84) before feeding the CNN model. The next trick is loss function, if we use the MSE loss used in LunarLander, the score cannot reach too high (about 200~300). We modify it to SmoothL1Loss, this seems like a good idea. The last trick is reward setting to append to buffer, we get the [reward/10] per game in LunarLander, but for the Breakout, the reward every time we hit a brick is much lower than LunarLander, if we set [reward/10] that may make the agent difficult to train. As a result, we think reward should not be divided by 10.