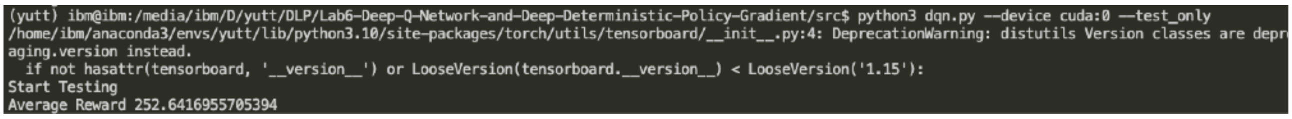
Lab 6 Deep Q–Network and Deep Deterministic Policy Gradient

Deep Learning and Practice

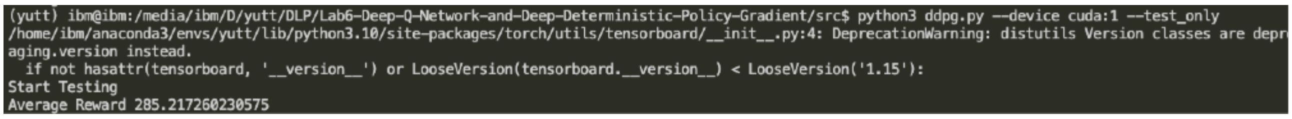
資科工碩1 游騰德 310551054

Performance: LunarLander–v2



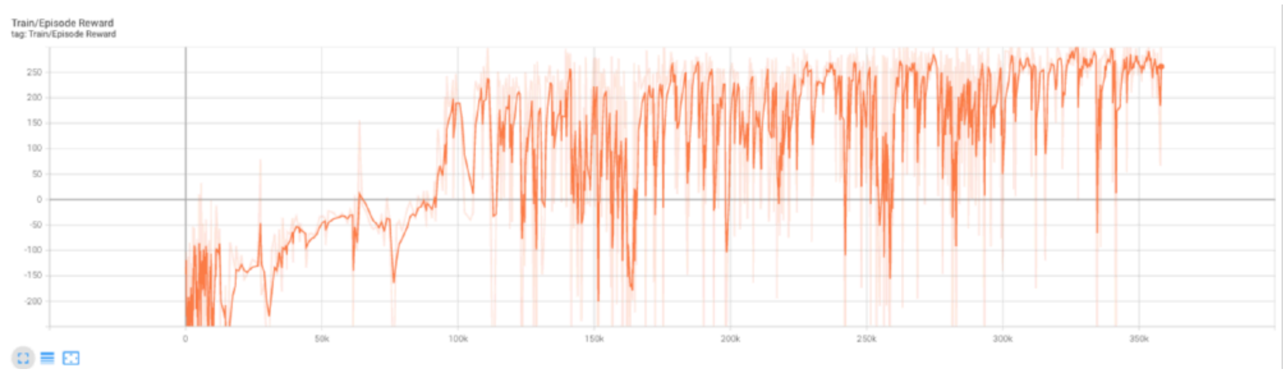
```
(yutt) ibm@ibm:/media/ibm/D/yutt/DLP/Lab6-Deep-Q-Network-and-Deep-Deterministic-Policy-Gradient/src$ python3 dqn.py --device cuda:0 --test_only
/home/ibm/anaconda3/envs/yutt/lib/python3.10/site-packages/torch/utils/tensorboard/__init__.py:4: DeprecationWarning: distutils Version classes are depr
aging.version instead.
  if not hasattr(tensorboard, '__version__') or LooseVersion(tensorboard.__version__) < LooseVersion('1.15'):
Start Testing
Average Reward 252.6416955705394
```

Performance: LunarLanderContinuous–v2

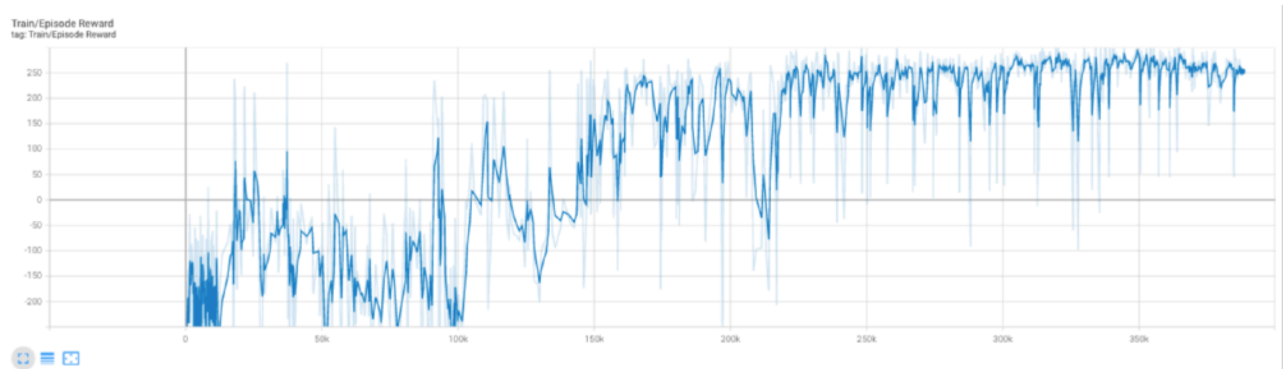


```
(yutt) ibm@ibm:/media/ibm/D/yutt/DLP/Lab6-Deep-Q-Network-and-Deep-Deterministic-Policy-Gradient/src$ python3 ddpg.py --device cuda:1 --test_only
/home/ibm/anaconda3/envs/yutt/lib/python3.10/site-packages/torch/utils/tensorboard/__init__.py:4: DeprecationWarning: distutils Version classes are depr
aging.version instead.
  if not hasattr(tensorboard, '__version__') or LooseVersion(tensorboard.__version__) < LooseVersion('1.15'):
Start Testing
Average Reward 285.217260230575
```

Tensorboard plot: LunarLander–v2



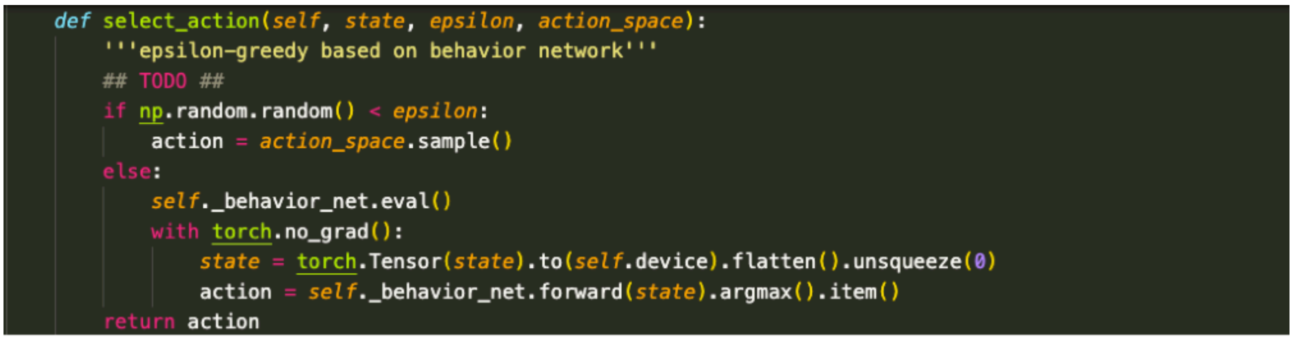
Tensorboard plot: LunarLanderContinuous–v2



Implementation of algorithms

DQN

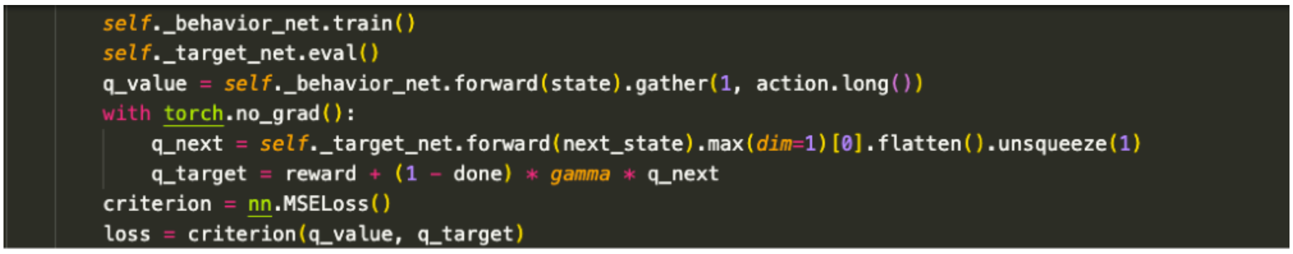
Select action



```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if np.random.random() < epsilon:
        action = action_space.sample()
    else:
        self._behavior_net.eval()
        with torch.no_grad():
            state = torch.Tensor(state).to(self.device).flatten().unsqueeze(0)
            action = self._behavior_net.forward(state).argmax().item()
    return action
```

The action is selected by ε greedy algorithm. Given a probability ε, the algorithm will select the action with the highest score by ε and select the action randomly by 1 – ε.

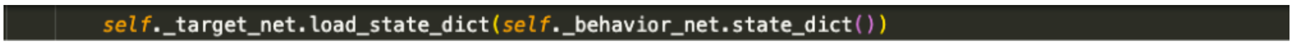
Behavior network



```python
    self._behavior_net.train()
    self._target_net.eval()
    q_value = self._behavior_net.forward(state).gather(1, action.long())
    with torch.no_grad():
        q_next = self._target_net.forward(next_state).max(dim=1)[0].flatten().unsqueeze(1)
        q_target = reward + (1 - done) * gamma * q_next
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

The behavior network is updated according to $y_j = r_j + \gamma \max_a widehat(Q)\left(phi_{j+1}, a'; \theta\right)$ when the episode not terminated.

Target network

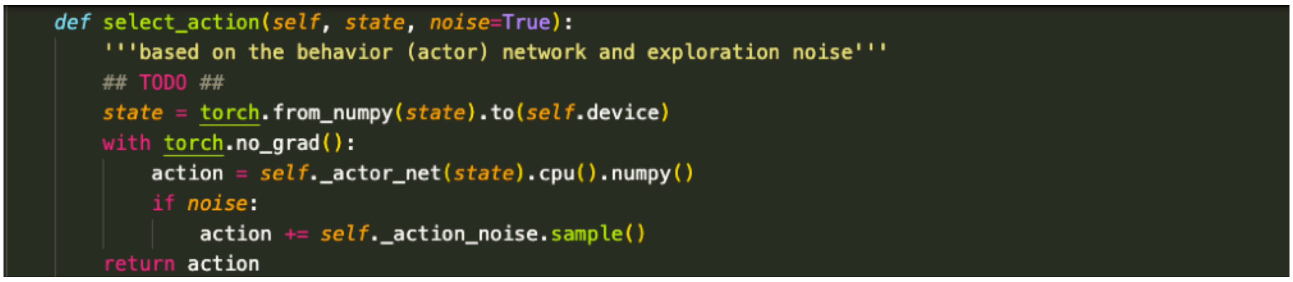


```python
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

The target network is updated via coping behavior network's weight in every several episodes.

DDPG

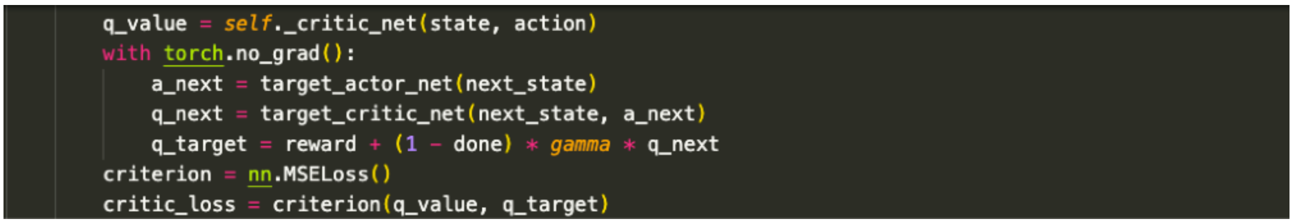
Select action



```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.from_numpy(state).to(self.device)
    with torch.no_grad():
        action = self._actor_net(state).cpu().numpy()
        if noise:
            action += self._action_noise.sample()
    return action
```

The gaussian noise is added to aid the selection of action.

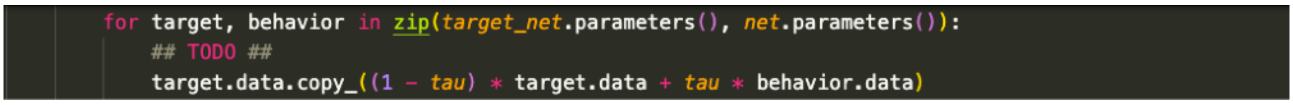
Behavior network



```
q_value = self._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 + (1 - done) * gamma * q_next
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
```

In each step, the behavior network is updated by the Q value of both actor and critic networks. The criterion is MSE Loss.

Target network



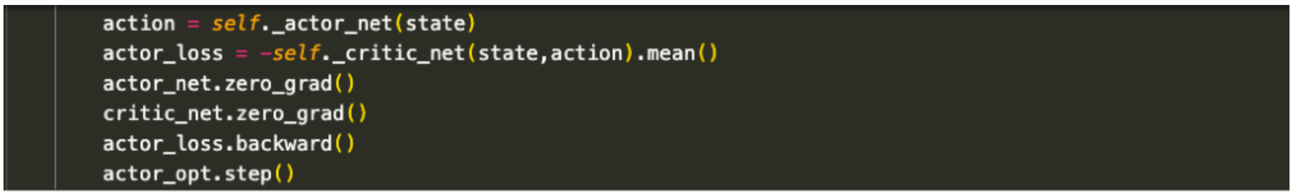
```
for target, behavior in zip(target_net.parameters(), net.parameters()):
    ## TODO ##
    target.data.copy_((1 - tau) * target.data + tau * behavior.data)
```

The target network is updated by by soft copying from the behavior network. The update is composed of $1 - \tau$ os target network and τ of behavior network.

Differences between implementation and algorithms

In the optimization step of DQN's algorithm, the gradient is clipped during the updating procedure of behavior network.

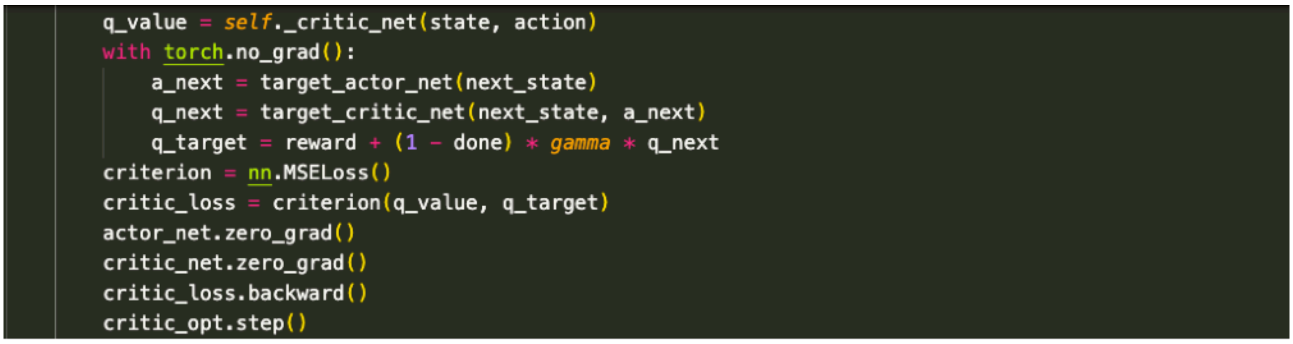
Implementation and the gradient of actor updating



```
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

Since the actor's goal is to maximize the output value of the critic, calculating the gradient of the actor is equivalent to minimizing the loss of the critic.

Implementation and the gradient of critic updating



```
q_value = self._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 + (1 - done) * gamma * q_next
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

The next Q value and the target Q value are calculated via actor and critic network. The both values are used to determine the gradient of the critic network.

Discount factor

The discount factor is a value that is used to describe the importance of a state.

Epsilon-greedy in comparison to greedy action selection

We might miss some great action during the training. By applying epsilon-greedy selection, we force the network to select some action that has never been taken under the current policy. This is to explore hidden good choices.

Target network

The behavior network is updating commonly and is always changing during training. We need a network that is more stable to apply steady policy. As a result, we copy the behavior network in every several episodes to fix the network for a longer time.

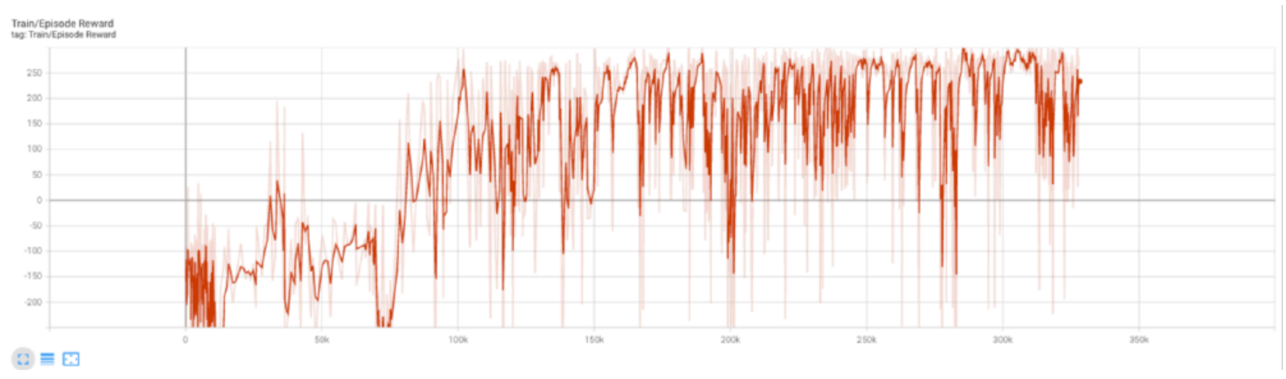
Replay buffer size

The replay buffer size act like the memory of the past game. If the replay buffer size is too small, the network is unable to remember some long term policy. In that case, the model is always playing the game and training with recent steps, which may overfit the model easily. However, if the replay buffer is too big, it will be difficult to sample new steps due to the enormous memory of history and increase the time of training.

Bonus

Double DQN

Results



```
(yutt) ibm@ibm:/media/ibm/D/yutt/DLP/Lab6-Deep-Q-Network-and-Deep-Deterministic-Policy-Gradient/src$ python3 double_dqn.py --device cuda:2 --test_only
/home/ibm/anaconda3/envs/yutt/lib/python3.10/site-packages/torch/utils/tensorboard/__init__.py:4: DeprecationWarning: distutils Version classes are depr
aging.version instead.
  if not hasattr(tensorboard, '__version__') or LooseVersion(tensorboard.__version__) < LooseVersion('1.15'):
Start Testing
Average Reward 282.8401732470084
```

Implementation

The main difference between DQN and Double DQN is the way they determine target Q value. Since DQN may sometimes overestimate Q value, Double DQN performs the action selection by behavior network.

```
        a_next = self._behavior_net.forward(next_state).max(dim=1)[1].flatten().unsqueeze(1)
        q_next = self._target_net.forward(next_state).gather(1, a_next.long())
        q_target = reward + (1 - done) * gamma * q_next
```

Double DQN

```
        q_next = self._target_net.forward(next_state).max(dim=1)[0].flatten().unsqueeze(1)
        q_target = reward + (1 - done) * gamma * q_next
```

DQN

Extra hyperparameter tuning

During training DQN, I found that with target_freq set to 100, the network learns very slow, and the performance of the model is unsatisfactory and unstable. To make it more stable, I increased the target_freq to 1000 for DQN and Double DQN and found the training result to be satisfied.

Additionally, I set the batch size as large as I can in the beginning, but found that there is no need to do that. Since the performance did not improved (actually dropped) and the time consuming problem did not solved, I left batch size to be set to default afterwards.