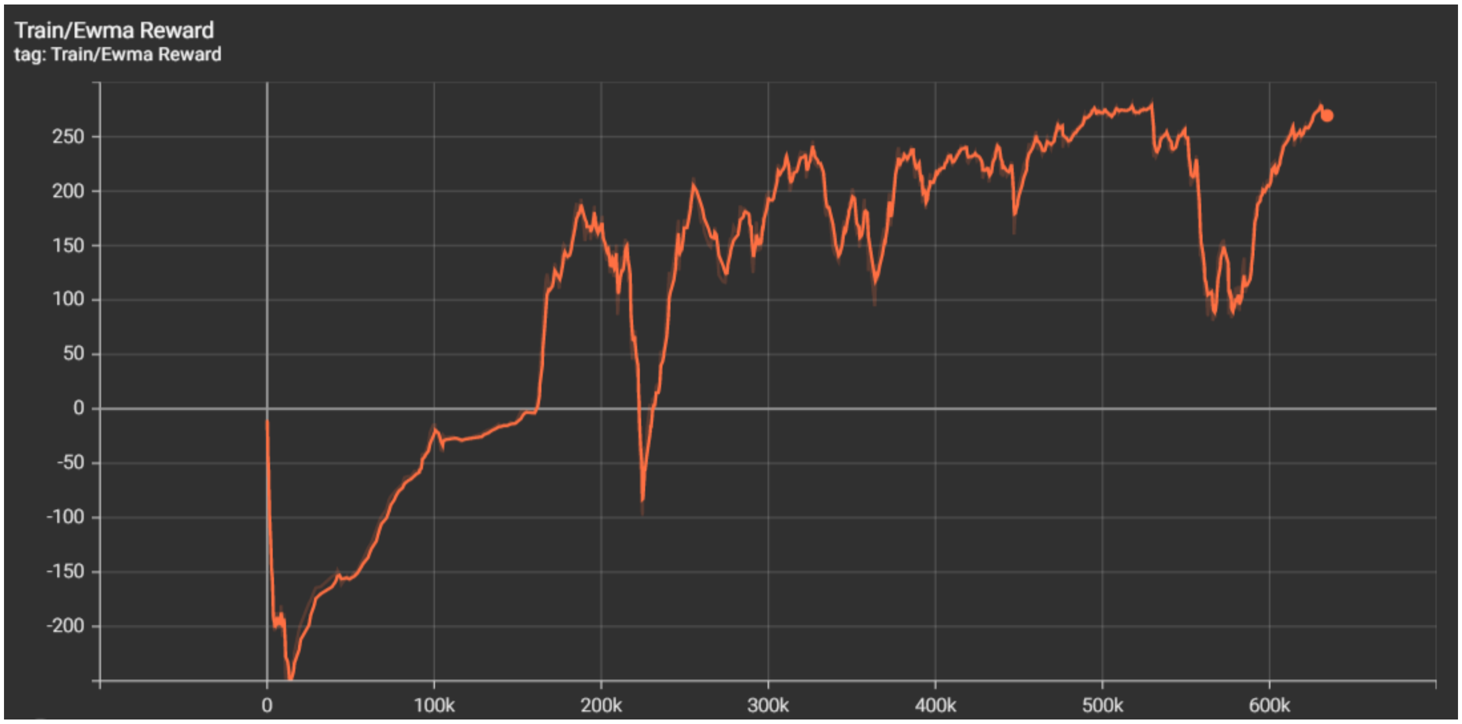
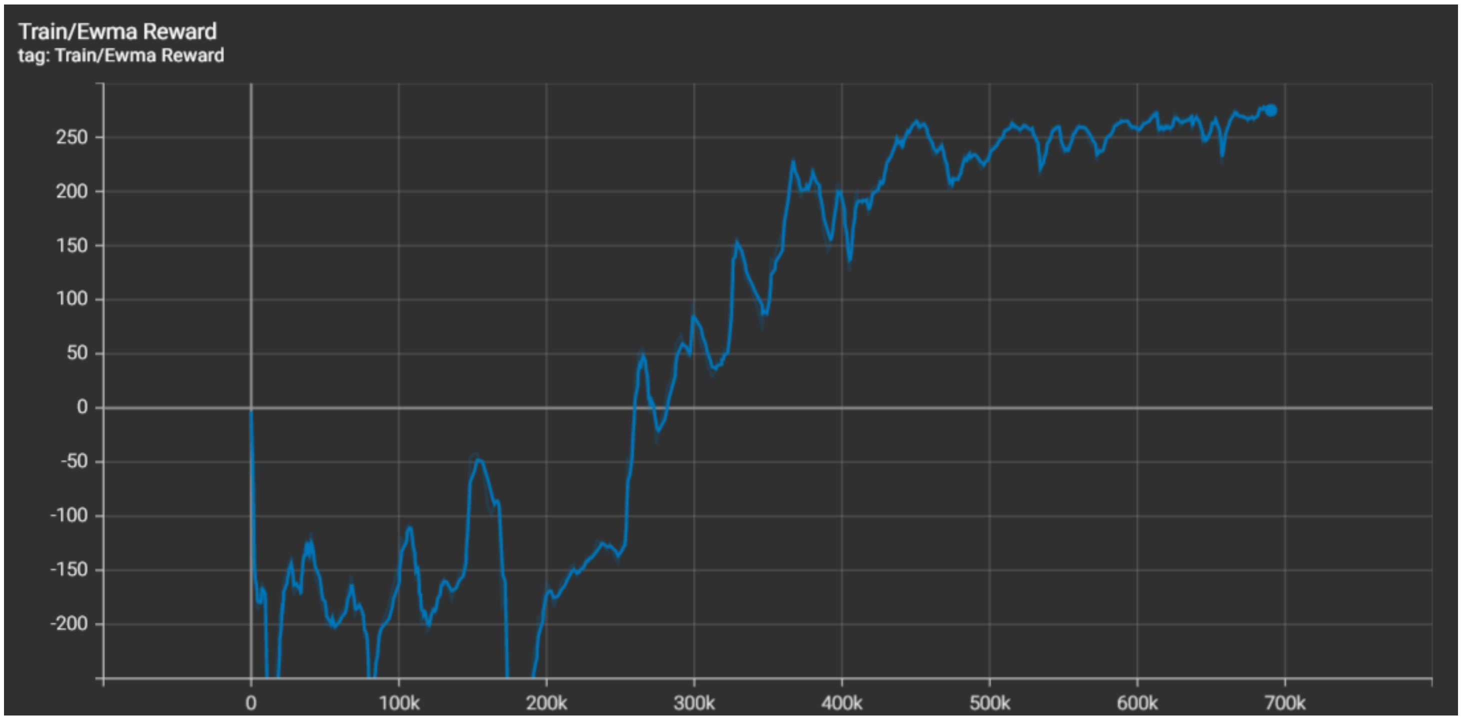
Lab6 Deep Q-Network and Deep Deterministic Policy Gradient

311554046 林愉修

1. LunarLander-v2 episode rewards



2. LunarLanderContinuous-v2 episode rewards

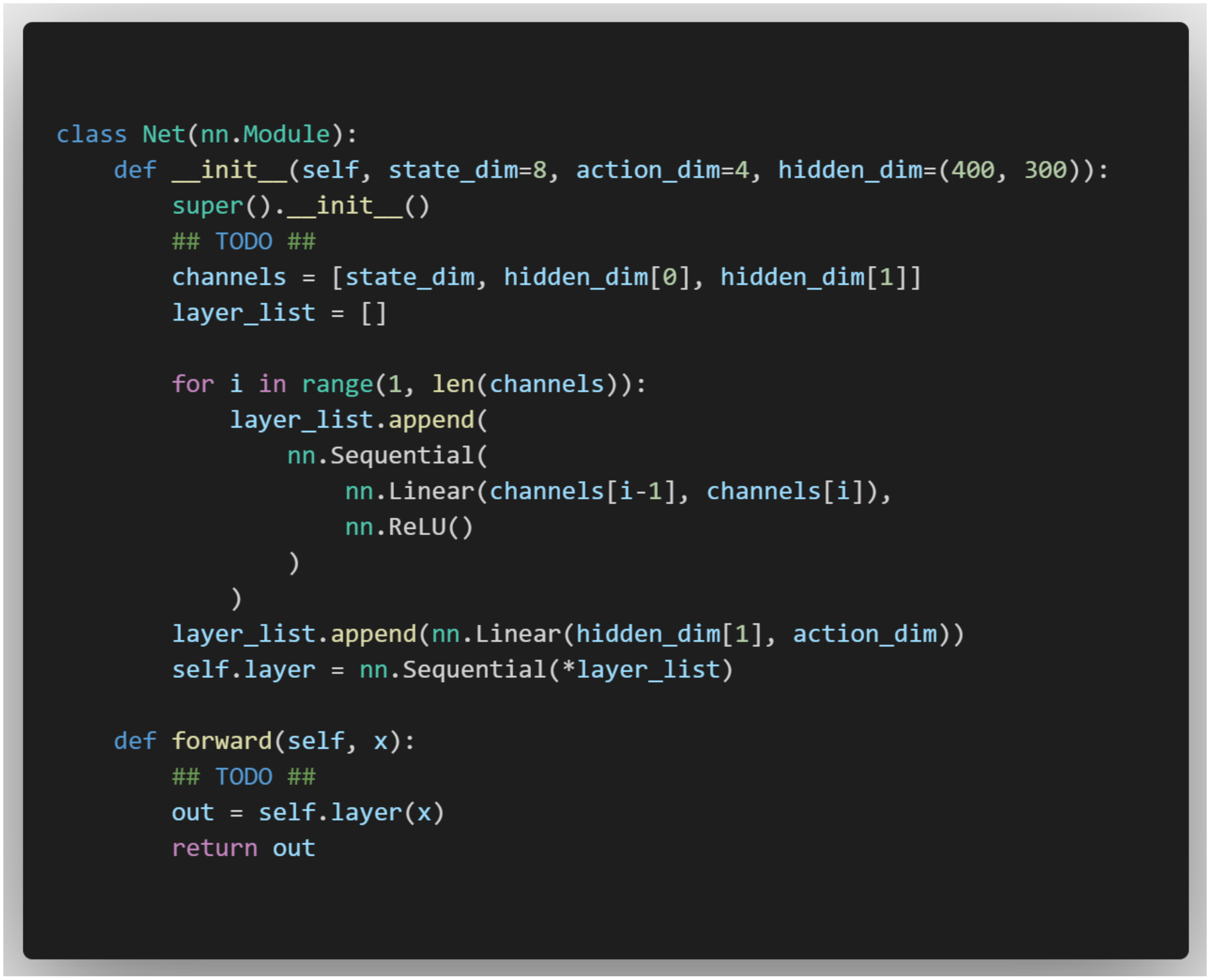


3. Implementation of both algorithms

3.1 DQN

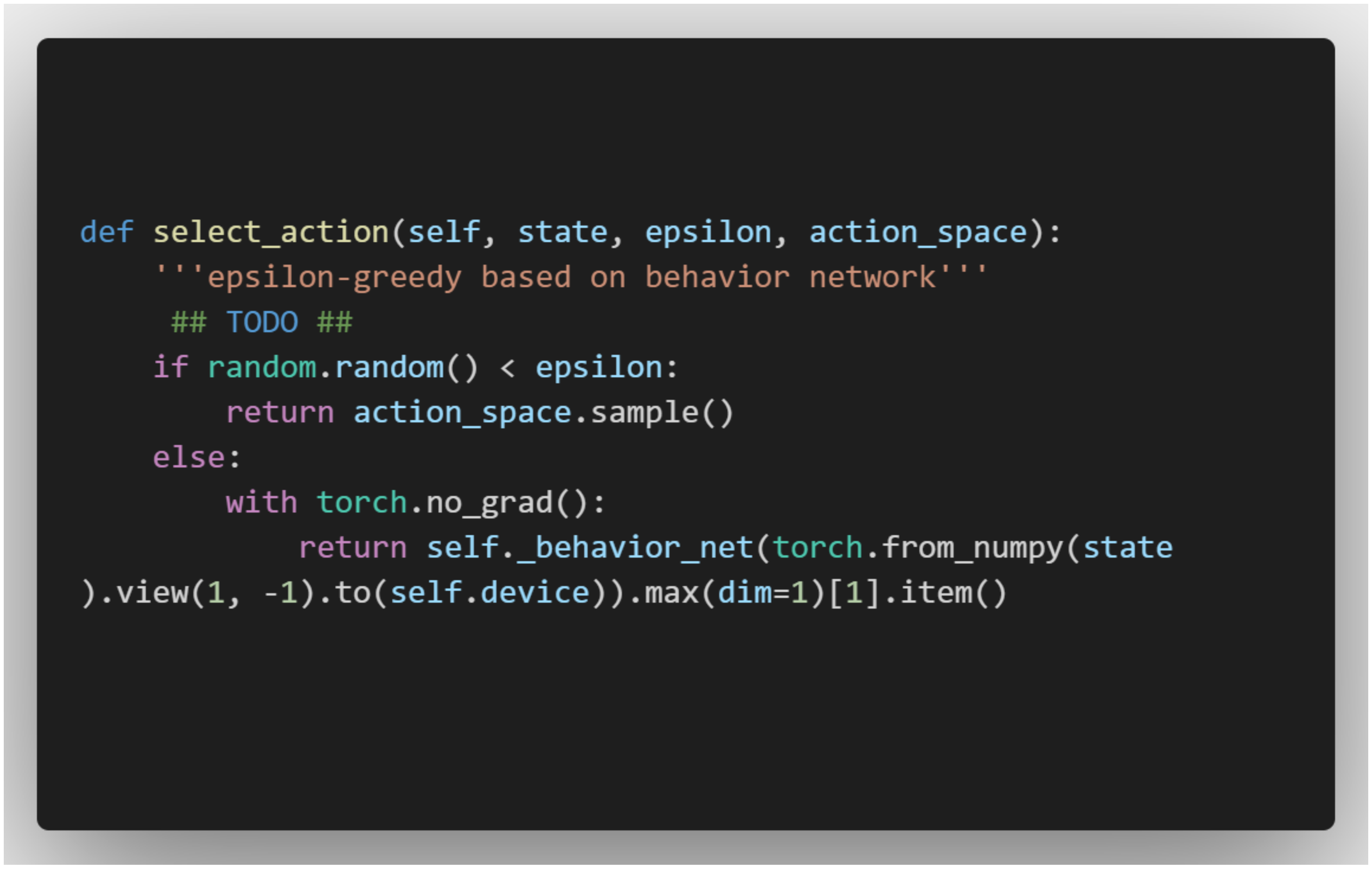
I construct the network which contains three fully-connected layers, with a $ReLU$ function in evey layer excpet the last one.
The network takes the state vector with dimension eight as input, and output an action vector with dimesion four, since there are four actions (no-op, fire left engine, fire main engine, fire right engine) in LunarLander-v2.

> Note that the hidden dimension is 32 in the sample code, but soon I found that the rewards during training is terrible, and I assumed the main reason of it might be the low hidden dimension, so I change the hidden_dim to (400, 300).



```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        channels = [state_dim, hidden_dim[0], hidden_dim[1]]
        layer_list = []

        for i in range(1, len(channels)):
            layer_list.append(
                nn.Sequential(
                    nn.Linear(channels[i-1], channels[i]),
                    nn.ReLU()
                )
            )
        layer_list.append(nn.Linear(hidden_dim[1], action_dim))
        self.layer = nn.Sequential(*layer_list)

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

The implementation of ϵ -greedy action selection. With probability ϵ, the agent randomly choose an action. the agent would choose a action based on the maximum $Q(s, a_i)$ with probability $1 - \epsilon$.

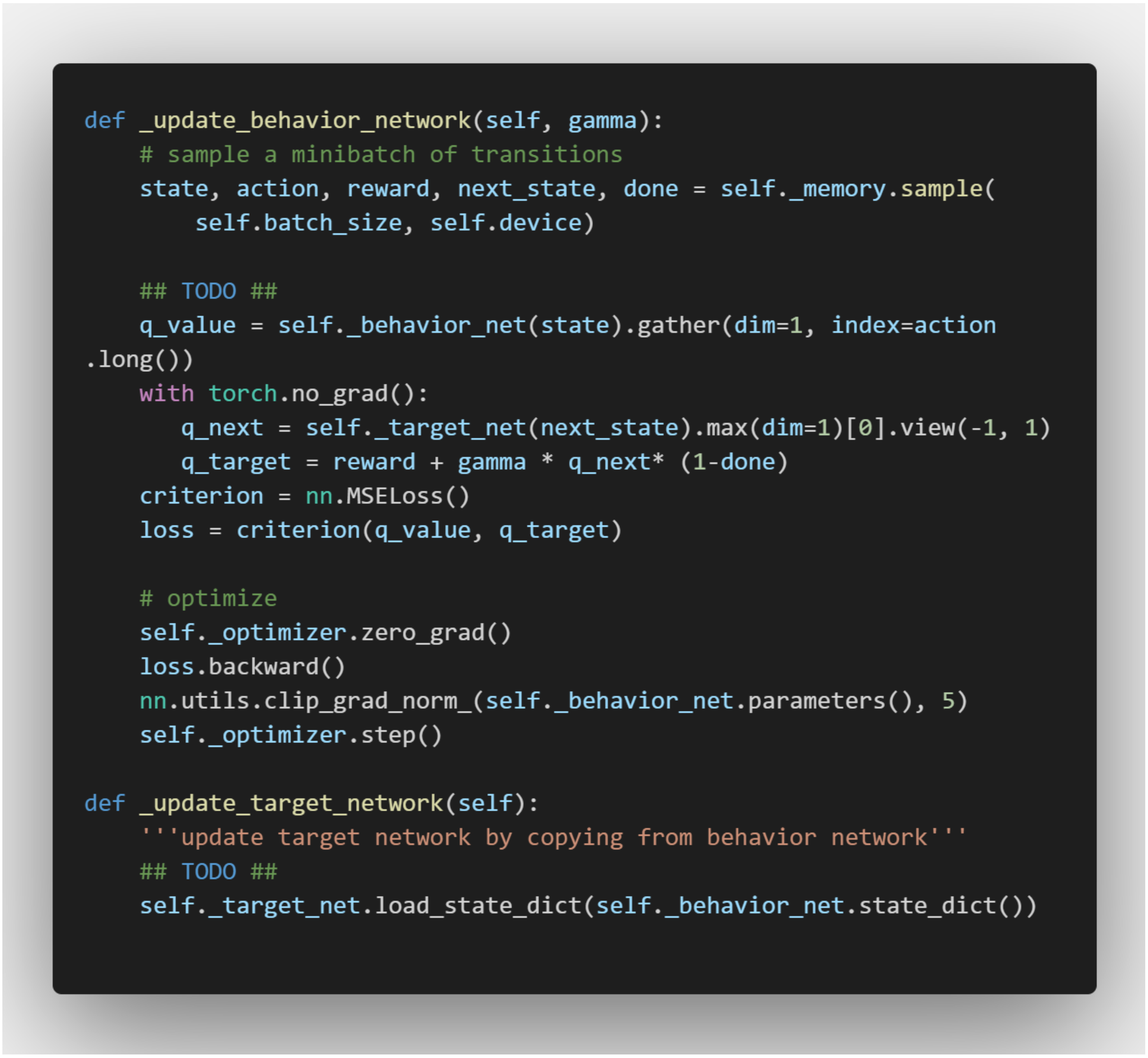


```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            return self._behavior_net(torch.from_numpy(state
).view(1, -1).to(self.device)).max(dim=1)[1].item()
```

For the update of the networks, it starts with sampling a batch of state transition observations (s_i, a_i, r_i, s_{i+1}) from the replay memory. The target value t_i is computed as follow,

$$t_i = \begin{cases} r_i, & \text{if episode terminates at step } i+1 \\ r_i + \gamma \underset{a'}{max} Q_t(s_{i+1}, a'; \theta_t), & \text{otherwise} \end{cases}$$

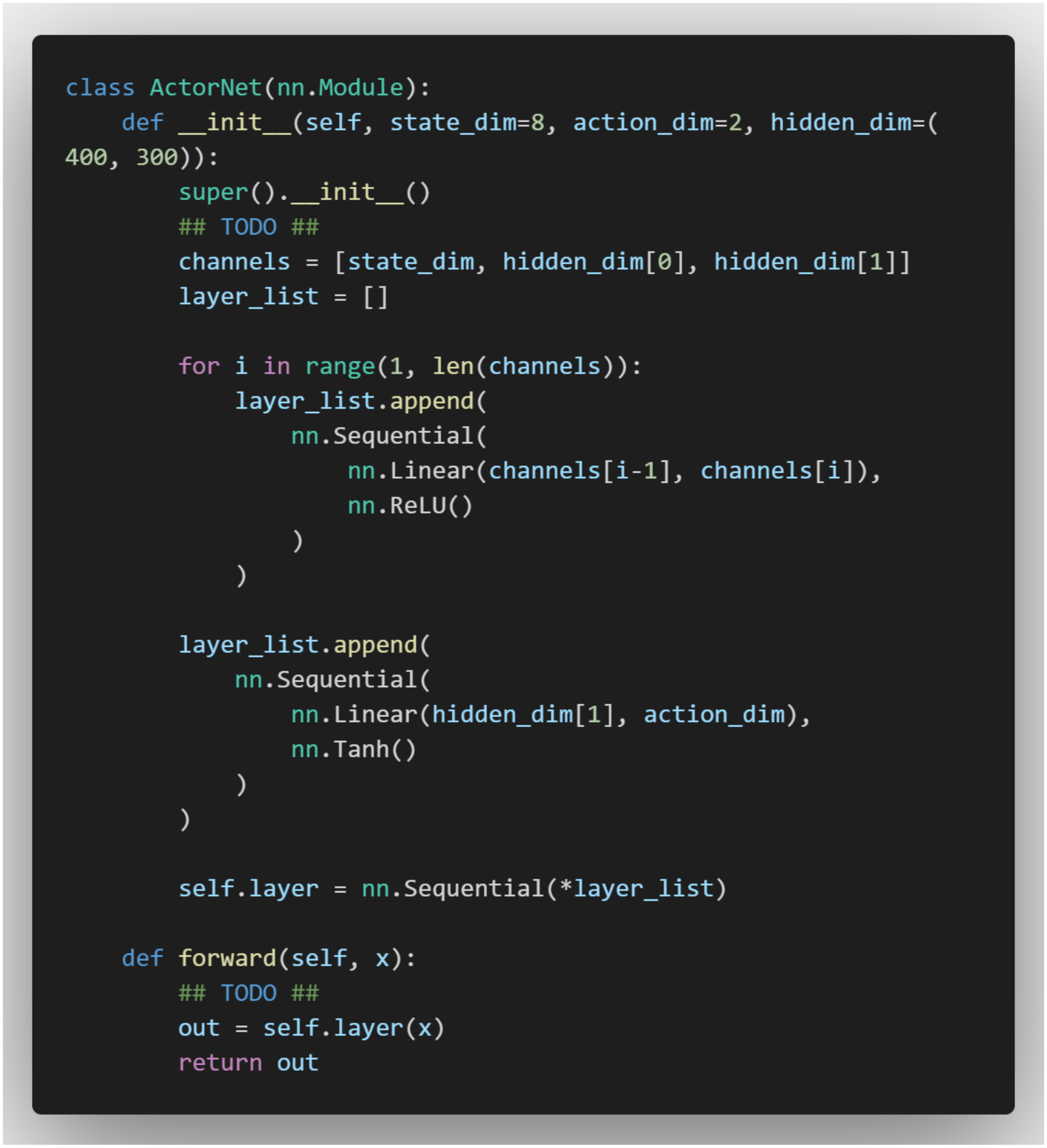
Perform a gradient descent step on MSE $(t_i - Q(s_i, a_i; \theta))^2$ w.r.t network parameters θ



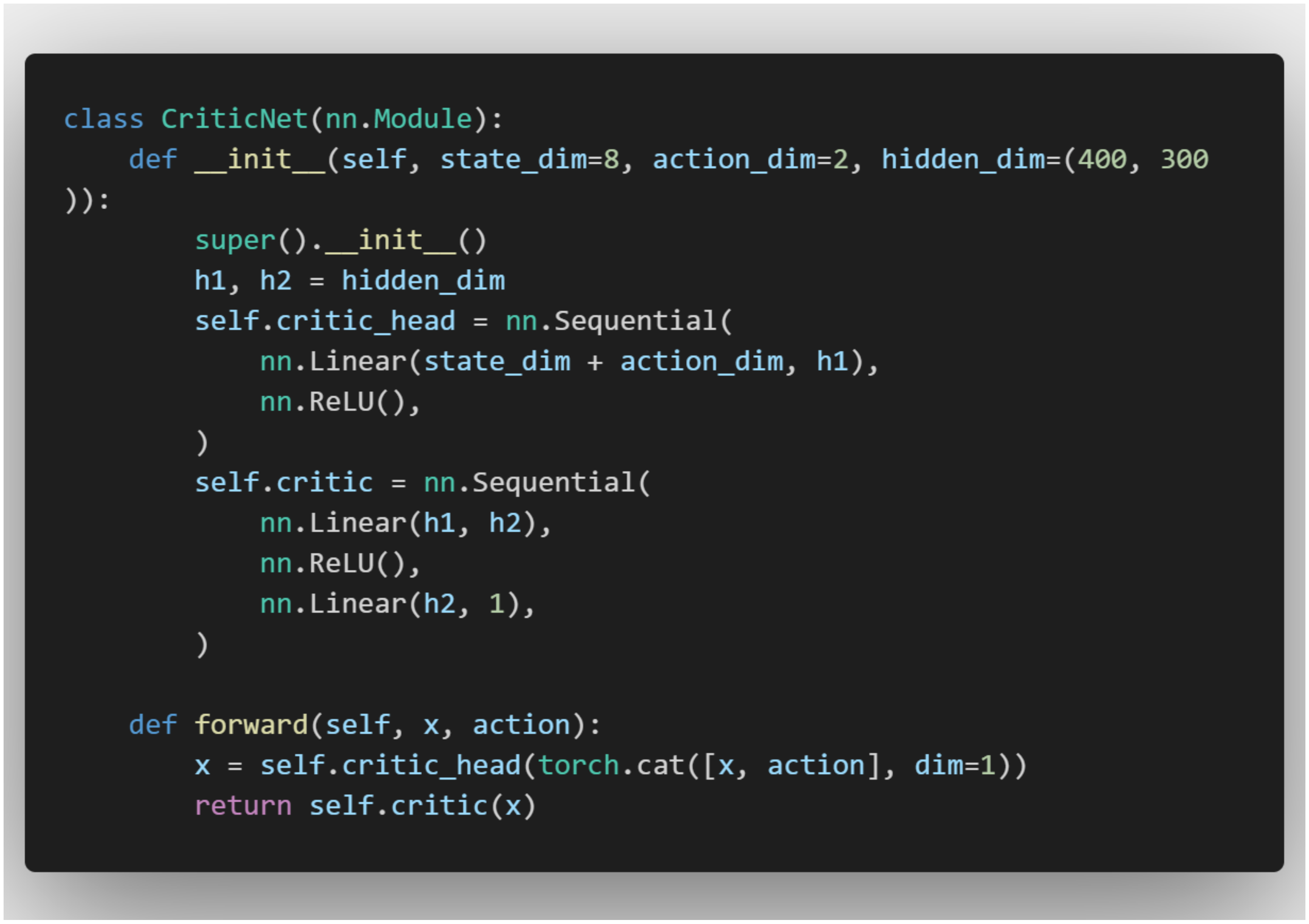
```
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## 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(dim=1)[0].view(-1, 1)
       q_target = reward + gamma * 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()

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())
```

3.2 DDPG

Both the actor network and critic network composed of three fully-connected layers, and the first two layers of them have a $ReLU$ function. The activation function in the last layer of actor network is $tanh$.
Given the current state, the action network is responsible for predicting an action, so it takes the state vector with dimension eight as inputs and outputs the action vector with dimension two, since there are two actions (main engine: -1~+1, left-right engine: -1~+1).
The critic network estimates the Q-value based on the current state and the predicted action, so it takes the state vector and the selected action as inputs and outpus a scalar value.

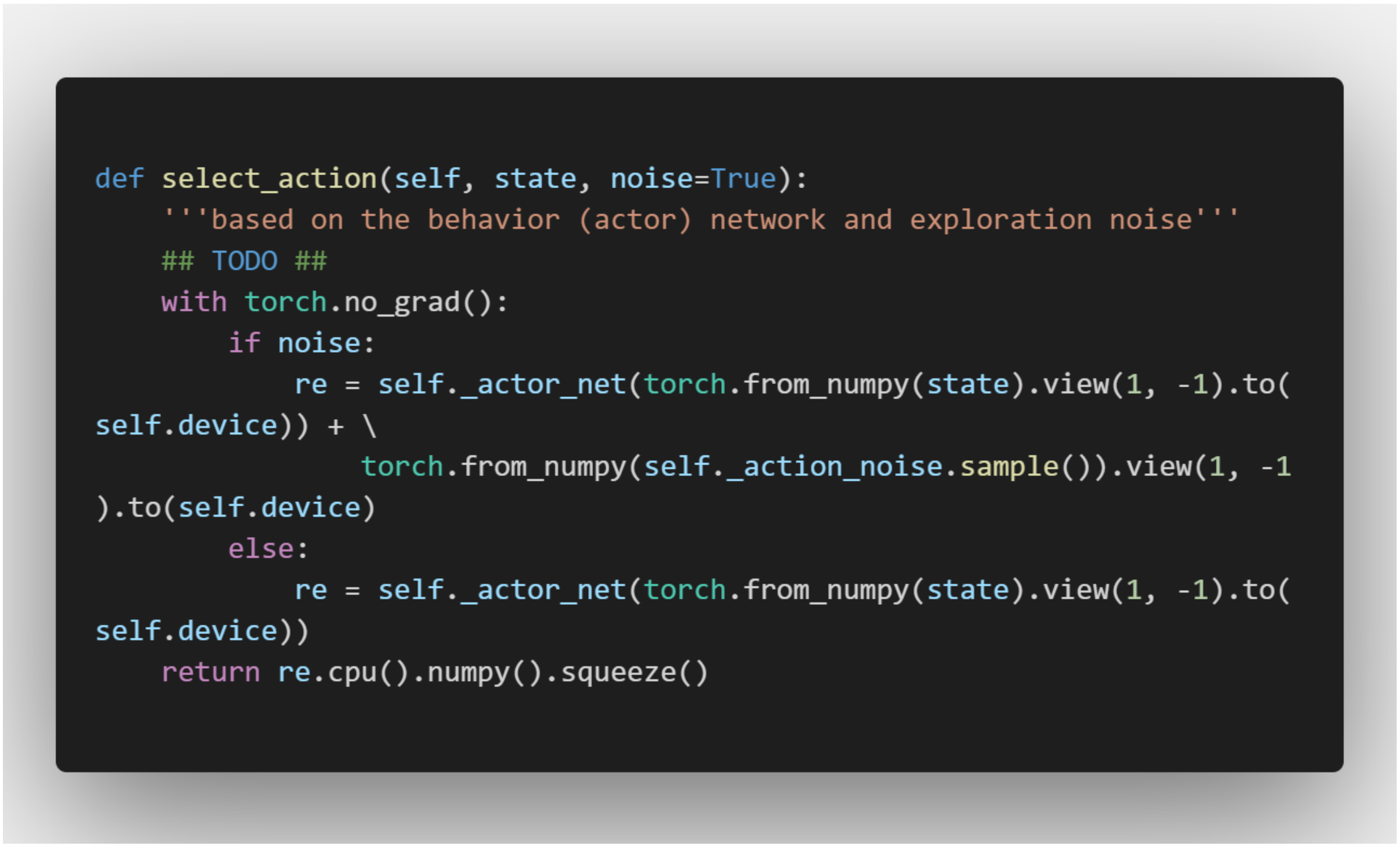


```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(
400, 300)):
        super().__init__()
        ## TODO ##
        channels = [state_dim, hidden_dim[0], hidden_dim[1]]
        layer_list = []

        for i in range(1, len(channels)):
            layer_list.append(
                nn.Sequential(
                    nn.Linear(channels[i-1], channels[i]),
                    nn.ReLU()
                )
            )

        layer_list.append(
            nn.Sequential(
                nn.Linear(hidden_dim[1], action_dim),
                nn.Tanh()
            )
        )

        self.layer = nn.Sequential(*layer_list)

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



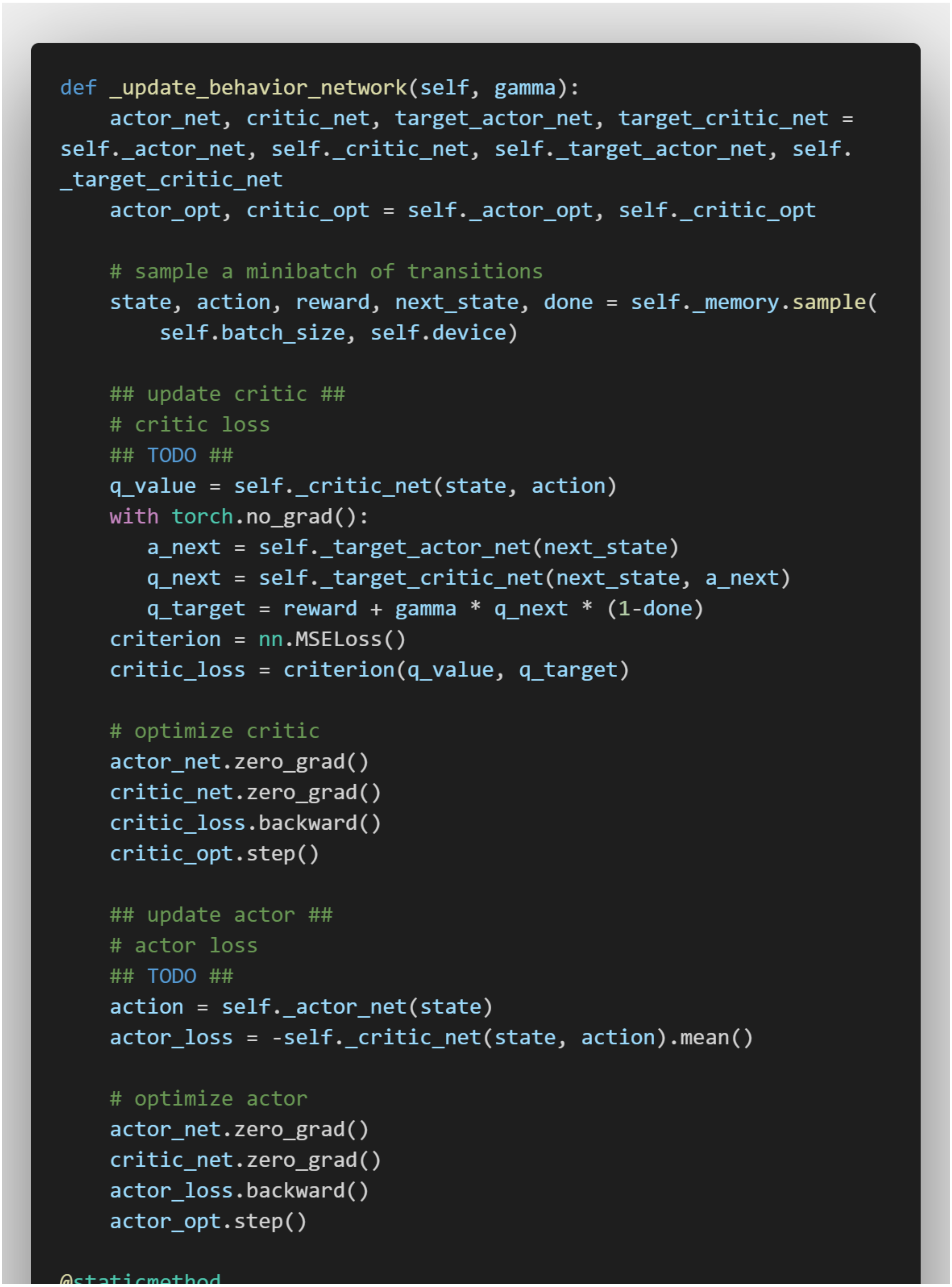
```
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300
)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = 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)
```

We select an action by the actor network mentioned above. In order to induce the agent to explore more choices, we add a noise sampled from the Gaussian distrubution.

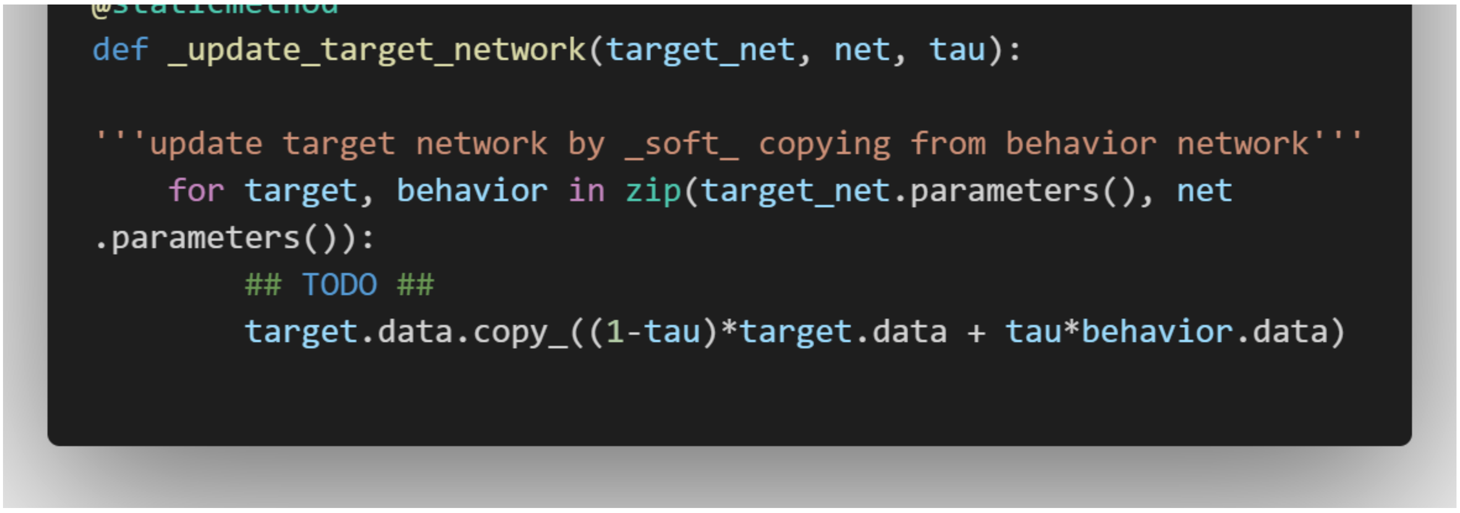


```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            re = self._actor_net(torch.from_numpy(state).view(1, -1).to(
self.device)) + \
                torch.from_numpy(self._action_noise.sample()).view(1, -1
).to(self.device)
        else:
            re = self._actor_net(torch.from_numpy(state).view(1, -1).to(
self.device))
    return re.cpu().numpy().squeeze()
```

We use the Q_{target} output from the target network and the $Q(s,a)$ output from the behavior to compute the MSE loss, and then update Q.
We can obtain $Q(s,a)$ by the actor network μ and the critic network Q from behavior network. Since we want to maximize $Q(s,a)$ by updating actor network, we define $L_\mu = \mathbb{E}[-Q\left(s,\mu\left(s\right)\right)]$, and update through backpropagation.



```python
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 = 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 ##
    # actor loss
    ## 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()

@staticmethod
```



```
@staticmethod
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 ##
        target.data.copy_((1-tau)*target.data + tau*behavior.data)
```

4. Differences between implementation and algorithms.

For both DQN and DDPG, the training first start with the warm up step, which I set to 10000 iterations, during the warm up step, the parameters of the networks are not updated, and the action is randomly selected from the action space and for the state transitions, we store into the replay memory.
For DQN, We don't update the behavior network every iteration, but every four iterations.

5. Implementation and the gradient of actor updating.

Action network: μ with parameters θ_μ
Critic network: Q with parameters θ_Q
The actor network is reponsible for updating its parameters in a way suggested by the critic network.
First, we sample a random mini-batch of transitions (s_i, a_i, r_i, s_{i+1}) from the replay memory buffer.
We can obtain the $\mu(s_i|\theta_\mu)$ via the actor network by the current state s_i, and using it to caluculate $Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q)$ by the critic network. Since we want to maximize the Q-value by updating actor network, we define the loss function as follow,

$$L_\mu = -\frac{1}{M}\sum_i Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q)$$

where M is the batch size.
The derivation of the actor network's gradient is as follow,

$$\nabla_{\theta_\mu} L_\mu = -\frac{1}{M}\sum_i \nabla_{\theta_\mu} Q(s_i, \mu(s_i|\theta_\mu)|\theta_Q)$$

$$= -\frac{1}{M}\sum_i \nabla_a Q(s,a|\theta_Q)|_{s=s_i,a=\mu(s_i|\theta_\mu)} \nabla_{\theta_\mu}\mu(s|\theta_\mu)|_{s=s_i}$$

6. Implementation and the gradient of critic updating.

Target network of critic: Q_t with parameters θ_{Q_t}
Target network of actor: μ_t with parameters θ_{μ_t}
The target Q-value for the critic network is as follow,

$$t_i = r_i + \gamma Q_t(s_{i+1}, \mu_t(s_{i+1}|\theta_{\mu_t})|\theta_{Q_t})$$

where the γ is the discount factor.
Since we wnat to minimize the difference between the $Q(s_i, a_i|\theta_Q)$ and the target value t_i over all M transitions. The loss function is defined as follow,

$$L_Q = \frac{1}{M}\sum_i (t_i - Q(s_i, a|\_Q))^2$$

which is the MSE loss.
The derivation of the critic network's gradient is as follow,

$$\nabla_{\theta_Q} L_Q = -\frac{1}{M}\sum_i \nabla_{\theta_Q}(t_i - Q(s_i, a|\theta_Q))^2$$

$$\approx (r_i + \gamma Q(s_{i+1}, \mu(s_{i+1}|\theta_{\mu_t})|\theta_{Q_t})) - Q(s_i, a_i|\theta_Q)\nabla_{\theta_Q} Q(s_i, a_i|\theta_Q)$$

7. Effects of the discount factor.

An agent's discounted reward is as follow,

$$G_t = R_{t+1} + \lambda R_{t+2} + \lambda^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

where $\lambda < 1$ is the discount factor.

With smaller discount factor, as in the equation above shown, the agent would mostly focus on the current rewards, on the other hand, with the larger discount factor, the long-term rewards have more impact to the agent.

8. Benefits of epsilon-greedy in comparison to greedy action selection.

By using ϵ-greedy, we can easily balance the exploration and exploitation, if we only select the best action at each time step, just like the greedy action selection, there is a still a chance that we missing out the better choices.

9. The necessity of the target network.

Unlike Q-learning, The DQN replace an exact value function with a function approximator.
With Q-learning you are updating exactly one state/action value at each timestep, whereas with DQN you are updating many, which causing the problem that you can affect the action values for the very next state you will be in, thus make the agent's learning process unstable.
By using a stable target network could solve this problem, and a typical solution would be predicting the value of each state by a fixed network, which its parameters are only updated after a period.

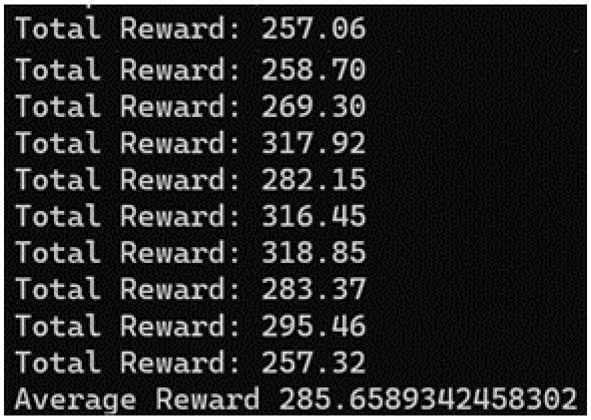
10. Effect of replay buffer size in case of too large or too small.

If the replay buffer is too large, then it might take a long time to become refreshed with good trajectories when they finally start to be discovered. And the larger buffer size cosumes more memory usage.
If the replay buffer is too small, then it might force the network only care about what it saw recently, hence the training process might be unstable.

11. Performance

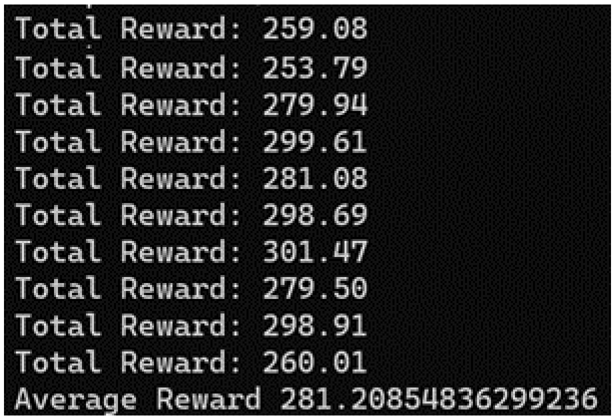
The learning rate of DQN is `1e-5` and the learning rate for both actor and critic networks in DDPG are set to `1e-3` .
Both algorithms warm up for `10000` iterations and are trained for `2000` episodes.
The discount factors are set to `0.99` .
The batch size for DQN is `128` and for DDPG is `64` .

11.1 DQN



```
Total Reward: 257.06
Total Reward: 258.70
Total Reward: 269.30
Total Reward: 317.92
Total Reward: 282.15
Total Reward: 316.45
Total Reward: 318.85
Total Reward: 283.37
Total Reward: 295.46
Total Reward: 257.32
Average Reward 285.6589342458302
```

11.2 DDPG



```
Total Reward: 259.08
Total Reward: 253.79
Total Reward: 279.94
Total Reward: 299.61
Total Reward: 281.08
Total Reward: 298.69
Total Reward: 301.47
Total Reward: 279.50
Total Reward: 298.91
Total Reward: 260.01
Average Reward 281.20854836299236
```