DLP-Lab6

ILE 310553043 You-Pin, Chen

May 22, 2022

1 Introduction

In this assignment, we are asked to implement two deep reinforcement algorithms by solving LunarLander-v2 using deep Q-network (DQN)[2] and LunarLanderContinuous-v2 using deep deterministic policy gradient (DDPG)[1]. DeepMind proposed DQN in 2013 and 2015 which uses a deep network to represent the value function. According to the Q-Learning in reinforcement learning, it provides the target value for the deep network and continuously updates the network until it converges. DQN uses two key technologies, one is a sample pool to break the correlation between samples, and the other is a fixed target network for better training stability and convergence. DQN can cope with high-dimensional input, but is helpless with high-dimensional action output. Subsequently, DDPG, also proposed by DeepMind, can solve situations with high-dimensional or continuous action spaces. It is an Actor-Critic method that combines deep networks. The code is available in github[link].

2 Result

2.1 Plot

2.1.1 LunarLander-v2

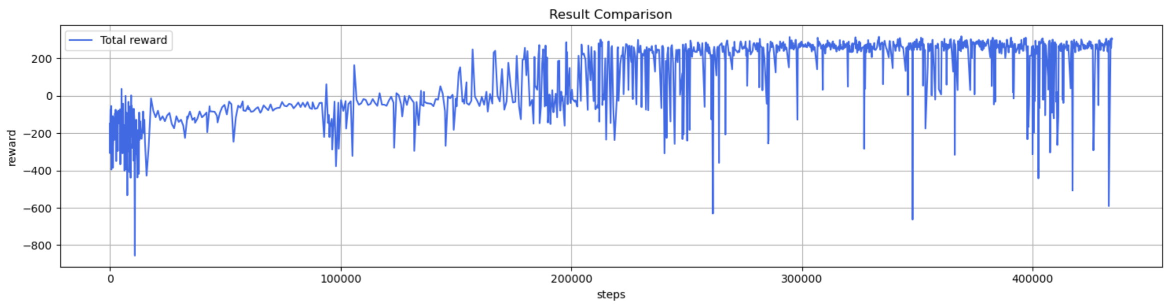

Figure 1: Total reward.

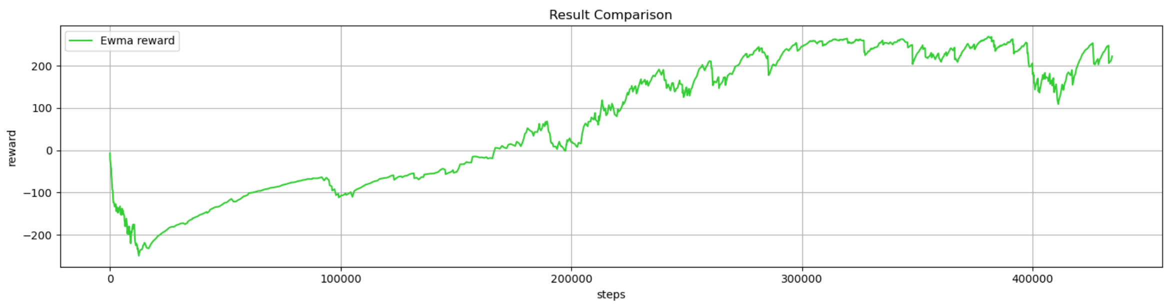

Figure 2: Ewma reward.

2.1.2 LunarLanderContinuous-v2

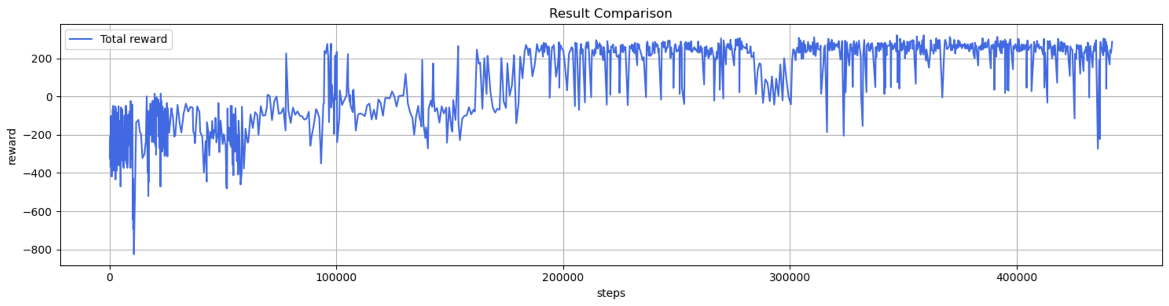

Figure 3: Total reward.

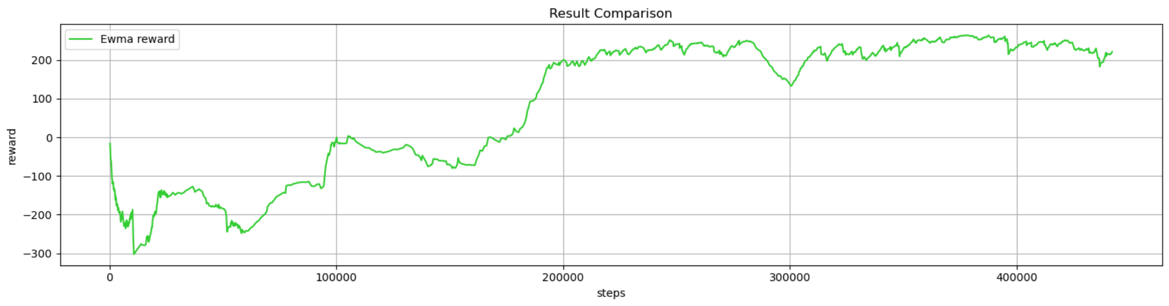

Figure 4: Ewma reward.

2.2 Performance

2.2.1 LunarLander-v2

```
Start Testing
Length: 206   Total reward: 260.26
Length: 228   Total reward: 249.40
Length: 245   Total reward: 274.16
Length: 188   Total reward: 306.07
Length: 239   Total reward: 279.27
Length: 218   Total reward: 308.13
Length: 205   Total reward: 304.34
Length: 193   Total reward: 279.80
Length: 254   Total reward: 290.89
Length: 211   Total reward: 253.31
Average Reward 280.561535729407
```

2.2.2 LunarLanderContinuous-v2

```
Start Testing
Length: 146   Total reward: 259.16
Length: 170   Total reward: 260.97
Length: 387   Total reward: 268.67
Length: 215   Total reward: 314.28
Length: 200   Total reward: 282.14
Length: 163   Total reward: 294.50
Length: 155   Total reward: 303.30
```

```
Length: 162    Total reward: 290.61
Length: 212    Total reward: 285.03
Length: 161    Total reward: 268.01
Average Reward 282.66754364019397
```

3 Implementation

3.1 Major implementation of Both algorithms in detail

3.1.1 Net (DQN)

This neural network is the place where DQN update the status of target value. The input is eight observations while the output is four actions. For our network architecture, we reimplemented the structure which provided by TA in the lab document as baseline. It contains input and output layer with one hidden layer. The dimension of hidden layer is 32 in default which in our experiment it did not perform well, so we changed different dimensions to 256 and got 280.56 reward score.

- Observation
  - Horizontal Coordinate
  - Vertical Coordinate
  - Horizontal Speed
  - Vertical Speed
  - Angle
  - Angle Speed
  - If first leg has contact
  - If second leg has contact
- Action
  - No-op
  - Fire left engine
  - Fire main engine
  - Fire right engine

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=256):
        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
```

3.1.2 Algorithm (DQN)

About DQN algorithm, the optimizer used while training is Adam. The explanation of implemented function shows in below.

- select_action
  - The model will select a unknown action with probability ϵ, otherwise it will select the action by maximum Q-values. For ϵ value, we set 1.0 in default which will decay after warm up steps by another hyperparameter *eps_decay*. The lower bound of ϵ will be 0.01 till the end of training which we said that the model will focus on exploitation.
- _update_behavior_network
  - This part is the place to update the weights. Since we will first sample transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from replay memory. The update equation in Eq.3.1.1 while γ is discount factor and $\hat{Q}$, θ^- is target network and its weights respectively. Then we choose Mean Square Error (MSE) as loss function to perform a gradient descent step on Eq.3.1.2 with respect to the network parameters θ.

$$y_j = \begin{cases} r_j & , if\ episode\ terminates\ at\ step\ j+1 \\ r_j + \gamma\ max_{a'} \hat{Q}(\phi_{j+1}, a', ; \theta') & , otherwise \end{cases} \tag{3.1.1}$$

$$MSE\ Loss = (y_j - Q(\phi_j, a_j; \theta))^2 \tag{3.1.2}$$

- _update_target_network
  - This function update the target network by copying from behavior network in every C steps.

```
class DQN:
    def __init__(self, args):
        self._behavior_net = Net().to(args.device)
        self._target_net = Net().to(args.device)
        # initialize target network
        self._target_net.load_state_dict(self._behavior_net.state_dict())
        ## TODO ##
        self._optimizer = torch.optim.Adam(self._behavior_net.parameters(), lr=args.lr)
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)

        ## config ##
        self.device = args.device
        self.batch_size = args.batch_size
        self.gamma = args.gamma
        self.freq = args.freq
        self.target_freq = args.target_freq

    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_tensor = torch.tensor(state, device=self.device).reshape(1, -1)
                actions = self._behavior_net(state_tensor)
                return torch.max(actions, dim=1)[1].item()
```

```
def append(self, state, action, reward, next_state, done):
    self._memory.append(state, [action], [reward / 10], next_state,
                        [int(done)])

def update(self, total_steps):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma)
    if total_steps % self.target_freq == 0:
        self._update_target_network()

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()

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())

def save(self, model_path, checkpoint=False):
    if checkpoint:
        torch.save(
            {
                'behavior_net': self._behavior_net.state_dict(),
                'target_net': self._target_net.state_dict(),
                'optimizer': self._optimizer.state_dict(),
            }, model_path)
    else:
        torch.save({
            'behavior_net': self._behavior_net.state_dict(),
        }, model_path)

def load(self, model_path, checkpoint=False):
    model = torch.load(model_path)
    self._behavior_net.load_state_dict(model['behavior_net'])
    if checkpoint:
        self._target_net.load_state_dict(model['target_net'])
        self._optimizer.load_state_dict(model['optimizer'])
```

3.1.3 Testing (DQN)

Epsilon-greedy strategy is still used while selecting the action during testing. The value of ϵ is set 0.01 while we hope the model will estimate the action value by itself.

```python
def test(args, env, agent, writer):
    print('Start Testing')
    action_space = env.action_space
    epsilon = args.test_epsilon
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        for t in itertools.count(start=1):
            if args.render:
                env.render()
            action = agent.select_action(state, epsilon, action_space)
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                print(f'Length: {t:3d}\tTotal reward: {total_reward:.2f}')
                rewards.append(total_reward)
                break
    print('Average Reward', np.mean(rewards))
    env.close()
```

3.1.4 Replay Memory (DDPG)

Replay Memory is the place to store transition, and then DDPG will sample randomly while selecting action based on the actor network and exploration noise. The implementation is similar with the same part in DQN.

```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'''
        ## TODO ##
        transitions = random.sample(self.buffer, batch_size)
        return (torch.tensor(x, dtype=torch.float, device=device) for x in zip(*transitions))
```

3.1.5 ActorNet (DDPG)

The Actor output is a deterministic action, and the network that generates this deterministic action is defined as $a = \mu(s|\theta^{\mu})$. In the past, the policy gradient adopted a random strategy, and each acquisition action required sampling the distribution of the current optimal strategy, while DDPG adopted a deterministic strategy, which was directly determined by the function μ. Actor's estimation network is $a = \mu(s|\theta^{\mu})$, θ is the parameter of the neural network, and this estimation network is used to output real-time actions. In addition, Actor has a target network with the same structure but different parameters, which is used to update the value network Critic. Both networks are output actions action. The structure is referenced by the document provided by TA.

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.actor = nn.Sequential(
            nn.Linear(state_dim, hidden_dim[0]),
            nn.ReLU(),
            nn.Linear(hidden_dim[0], hidden_dim[1]),
            nn.ReLU(),
            nn.Linear(hidden_dim[1], action_dim),
            nn.Tanh())

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

3.1.6 CriticNet (DDPG)

Its role is to fit the value function $Q(s, a|\theta^{Q})$. There is also an estimation network and a target network. Both networks output the value Q-values of the current state at the output, and differ at the input. The target network input of Critic has two parameters, which are the observation value of the current state and the action output of the Actor's target network. The input of the Critic estimation network is the action action of the current Actor's estimation network output. The target network is used to calculate Q_{target}.

```
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)
```

3.1.7 Algorithm (DDPG)

The optimizer used in DDPG is same with DQN which is Adam. The explanation of implemented function shows in below.

- select_action
  - Compare to epislon-greedy strategy used in DQN, DDPG algorithm will combine the output of ActorNet with N_t according to the current policy and exploration noise. The noise will only be added in training process while we directly choose the output of ActorNet as next action in testing process.
- _update_behavior_network
  - This part is the place to update the weights. Since we will first sample transitions (s_i, a_i, r_i, s_{i+1}) from replay memory same as DQN. We will calculate the target value in every sample while updating behavior critic network. The update equation is in Eq.3.1.5 while γ is discount factor and Q', $\theta^{Q'}$ is target critic network and its weights respectively. Then we choose Mean Square Error (MSE) as loss function to perform a gradient descent step on Eq.3.1.4 with respect to the network parameters θ. Since our goal is to emphasize the ability of actor network in deciding, we estimated the value with the status between behavior actor network and behavior critic network while updating the behavior actor network.

$$y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}) \tag{3.1.3}$$

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{3.1.4}$$

- _update_target_network
  - Updating the target network by soft copy. Instead of copy the weights from behavior network to target network directly, given a percentage τ to update the weight.

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'} \end{aligned} \tag{3.1.5}$$

```
class DDPG:
    def __init__(self, args):
        # behavior network
        self._actor_net = ActorNet().to(args.device)
        self._critic_net = CriticNet().to(args.device)
        # target network
        self._target_actor_net = ActorNet().to(args.device)
        self._target_critic_net = CriticNet().to(args.device)
        # initialize target network
        self._target_actor_net.load_state_dict(self._actor_net.state_dict())
        self._target_critic_net.load_state_dict(self._critic_net.state_dict())
        ## TODO ##
        self._actor_opt = torch.optim.Adam(self._actor_net.parameters(), lr=args.lra)
        self._critic_opt = torch.optim.Adam(self._critic_net.parameters(), lr=args.lrc)
        # action noise
        self._action_noise = GaussianNoise(dim=2)
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)

        ## config ##
        self.device = args.device
        self.batch_size = args.batch_size
        self.tau = args.tau
        self.gamma = args.gamma

    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)
            if noise:
                noise_tensor = torch.tensor(self._action_noise.sample(), device=self.device)
                action = self._actor_net(state_tensor.reshape(1, -1)) +
                    noise_tensor.reshape(1, -1)
            else:
                action = self._actor_net(state_tensor.reshape(1, -1))
        return action.squeeze().cpu().numpy()

    def append(self, state, action, reward, next_state, done):
        self._memory.append(state, action, [reward / 100], next_state,
                            [int(done)])

    def update(self):
        # update the behavior networks
        self._update_behavior_network(self.gamma)
        # update the target networks
        self._update_target_network(self._target_actor_net, self._actor_net,
                                    self.tau)
        self._update_target_network(self._target_critic_net, self._critic_net,
                                    self.tau)

    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
    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_(tau*behavior.data+(1-tau)*target.data)

    def save(self, model_path, checkpoint=False):
        if checkpoint:
            torch.save(
                {
                    'actor': self._actor_net.state_dict(),
                    'critic': self._critic_net.state_dict(),
                    'target_actor': self._target_actor_net.state_dict(),
                    'target_critic': self._target_critic_net.state_dict(),
                    'actor_opt': self._actor_opt.state_dict(),
                    'critic_opt': self._critic_opt.state_dict(),
                }, model_path)
        else:
            torch.save(
                {
                    'actor': self._actor_net.state_dict(),
                    'critic': self._critic_net.state_dict(),
                }, model_path)

    def load(self, model_path, checkpoint=False):
        model = torch.load(model_path)
        self._actor_net.load_state_dict(model['actor'])
        self._critic_net.load_state_dict(model['critic'])
        if checkpoint:
            self._target_actor_net.load_state_dict(model['target_actor'])
            self._target_critic_net.load_state_dict(model['target_critic'])
            self._actor_opt.load_state_dict(model['actor_opt'])
            self._critic_opt.load_state_dict(model['critic_opt'])
```

3.1.8 Testing (DDPG)

The process is same with DQN testing. While you can see the noise is setting false which we had mentioned in previous section.

```
def test(args, env, agent, writer):
    print('Start Testing')
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        for t in itertools.count(start=1):
            if args.render:
                env.render()
            action = agent.select_action(state, noise=False)
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                print(f'Length: {t:3d}\tTotal reward: {total_reward:.2f}')
                rewards.append(total_reward)
                break
    print('Average Reward', np.mean(rewards))
    env.close()
```

3.2 Differences between your implementation and algorithms

During the implementation, there will be a warmup stage at the beginning of the training, which is not present in the algorithm. During the warmup process, we will not update the weights of the model, but simply let the computer continuously generate random actions and change the game's weights. The process is recorded in the replay buffer. Then, another differences between our implementation and algorithm is that the weights of the behavior network are updated every step of play in DQN, while in the implementation, it is preset that the weights will be updated every four steps of play.

3.3 Implementation and the gradient of actor updating

Our goal is to strengthen the behavior actor network's ability to determine actions, so that the behavior critic network can get a larger estimate, so the loss is defined as Eq.3.3.1. We can also separate the equation into two parts 3.3.2 and utilize the chain rule to calculate the gradient.

$$Loss = -Q(s, \mu(s|\theta^{\mu})|\theta^{Q}) \tag{3.3.1}$$

$$\frac{\delta Loss}{\delta \theta^{\mu}} = -\frac{\delta Q(s, \mu(s|\theta^{\mu})|\theta^{Q})}{\delta a} \cdot \frac{\delta a}{\delta \mu(s|\theta^{\mu})} \cdot \frac{\delta \mu(s|\theta^{\mu})}{\delta \theta^{\mu}} \tag{3.3.2}$$

```
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()
```

3.4 Implementation and the gradient of critic updating

Randomly extract some state transitions from the replay buffer, and then use the y which is calculated by the target critic network Q' and the calculated estimated value of the behavior critic network Q to calculate the MSE and update the behavior critic network. The calculation formula of y and MSE shows in Eq.3.1.5 and Eq.3.1.4. The Gradient is derived as follows.

$$\frac{\delta Loss}{\delta \theta^{\mu}} = \frac{2}{N} * (Q(s, a|\theta^{Q}) - y) \tag{3.4.1}$$

```
## 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()
```

4 Explanation

4.1 Effects of the discount factor

The weight of the current reward is the largest, and the weight of the reward is smaller as the time axis goes back, and the weight decreases exponentially. The gamma in the code is the discount factor.

4.2 Benefits of epsilon-greedy in comparison to greedy action selection

When the model is trained, in addition to selecting the best action from the known situation (exploitation), it is also important to explore the unknown but possibly the best action (exploration). We use a given probability ϵ to decide the next step of the model. Exploitation or exploration, this is epsilon-greedy.

4.3 The necessity of the target network

If we use the same network as the behavior network and the target network, the target network will also be updated every time the behavior network is updated, and it is difficult to train when the target value keeps changing, so we will use the behavior network with The target network is separated, and the target network is updated at regular intervals, which helps to stabilize the training process.

4.4 The effect of replay buffer size in case of too large or too small

If the replay buffer is too small, the buffer will be filled with the latest episodes. In this case, the state of each sample is related to each other at a high level, which may lead to overfitting of the model; if the replay buffer is enlarged, each sample It is more likely to see states with low correlation with each other. Although the training time will be longer, the entire training process will be more stable, and the model will be able to achieve better performance.

5 Bonus

5.1 DDQN

The difference between DQN and DDQN[3] is in the calculation of the target Q-values of the next states. In DQN, we simply take the maximum of all the Q-values over all possible actions. It will over-estimated being in this state although this only happened due to the statistical error, hence DDPG proposed to estimate the value of the chosen action instead. The chosen action is the one selected by our policy model.

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-) \tag{5.1.1}$$

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \theta_t); \theta_t^-) \tag{5.1.2}$$

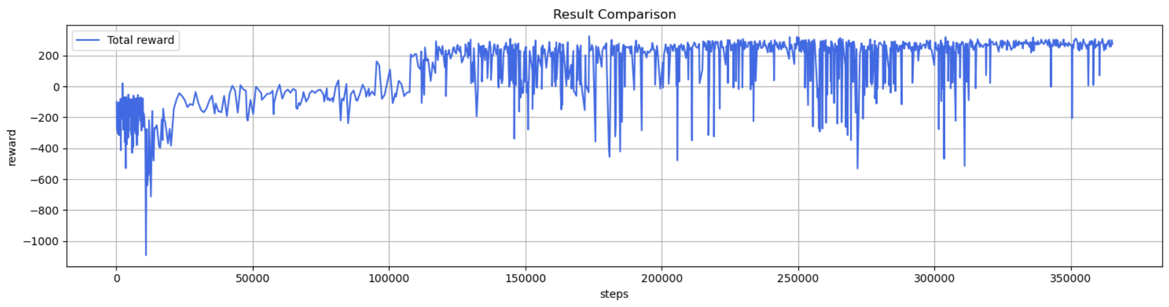

Figure 5: Total reward.

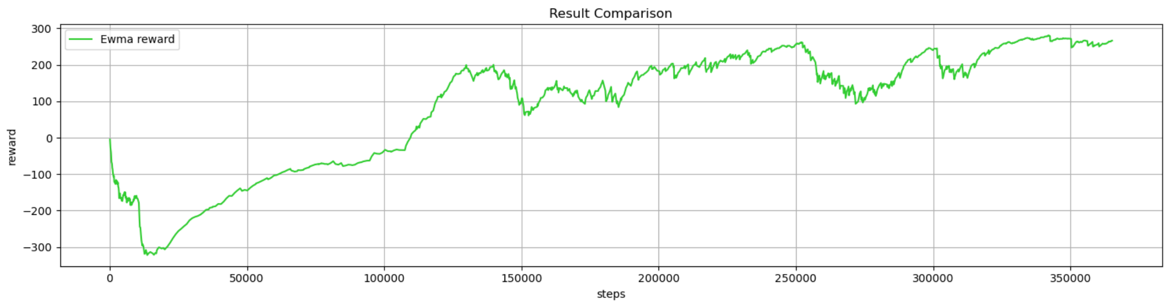

Figure 6: Ewma reward.

```
### DQN ###
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)

### DDQN ###
q_value = self._behavior_net(state).gather(dim=1, index=action.long())
        with torch.no_grad():
            q_next_eval = self._behavior_net(next_state)
            max_q_idx = torch.max(q_next_eval, dim=1)[1].long().reshape(-1, 1)
            q_next = self._target_net(next_state).gather(dim=1, index=max_q_idx)
            q_target = reward + gamma*q_next*(1-done)
        criterion = nn.MSELoss()
        loss = criterion(q_value, q_target)
```

```
Start Testing
Length: 168    Total reward: 267.40
Length: 190    Total reward: 255.91
Length: 203    Total reward: 284.20
Length: 188    Total reward: 311.94
Length: 202    Total reward: 283.44
Length: 191    Total reward: 313.71
Length: 190    Total reward: 303.09
Length: 175    Total reward: 287.90
Length: 210    Total reward: 301.59
Length: 171    Total reward: 263.53
Average Reward 287.27240881450365
```

References

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.