DLP Lab6: Deep Q-Network and Deep Deterministic Policy Gradient

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

August 28, 2021

1 LunarLander-v2 tensorboard plot

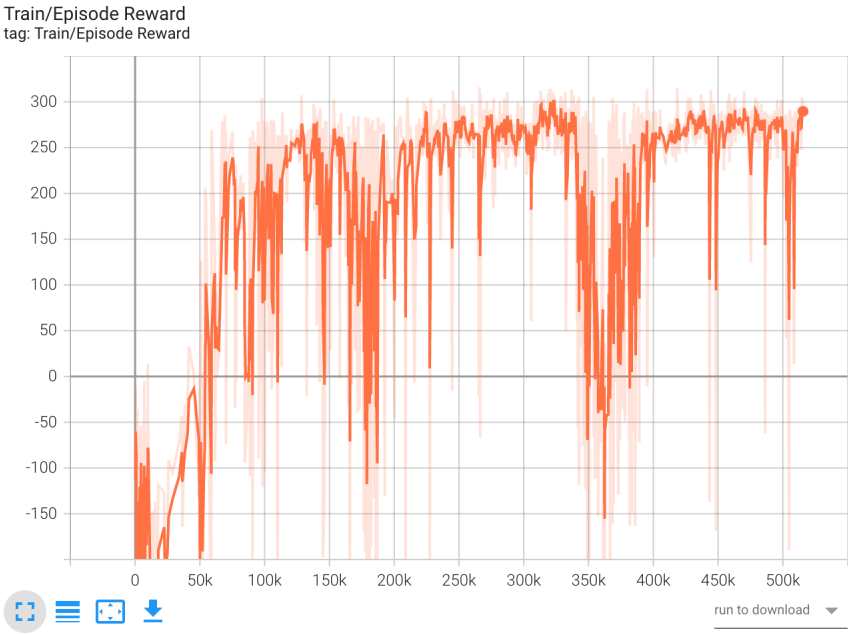

Figure 1: DQN episode reward with 2000 episodes.

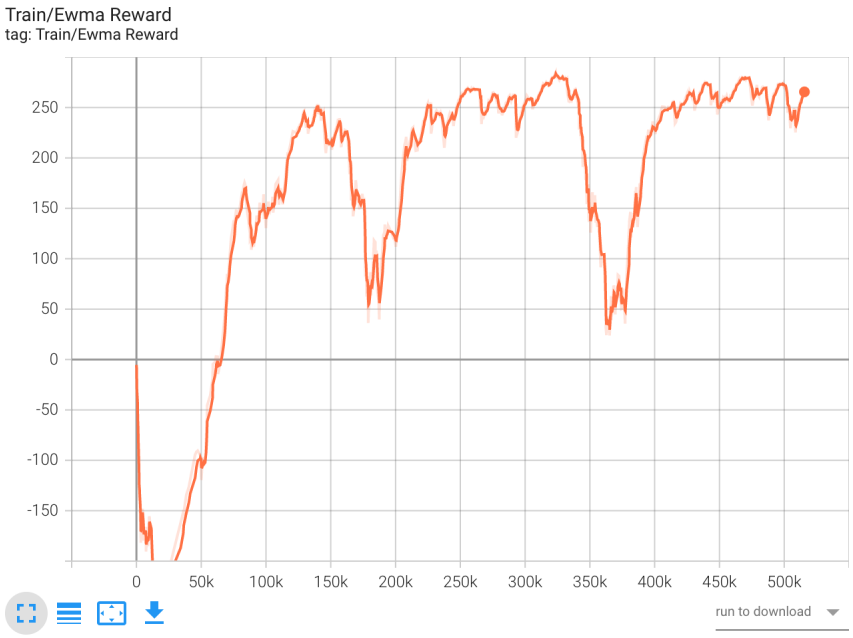

Figure 2: DQN ewma reward with 2000 episodes.

2 LunarLanderContinuous-v2 tensorboard plot

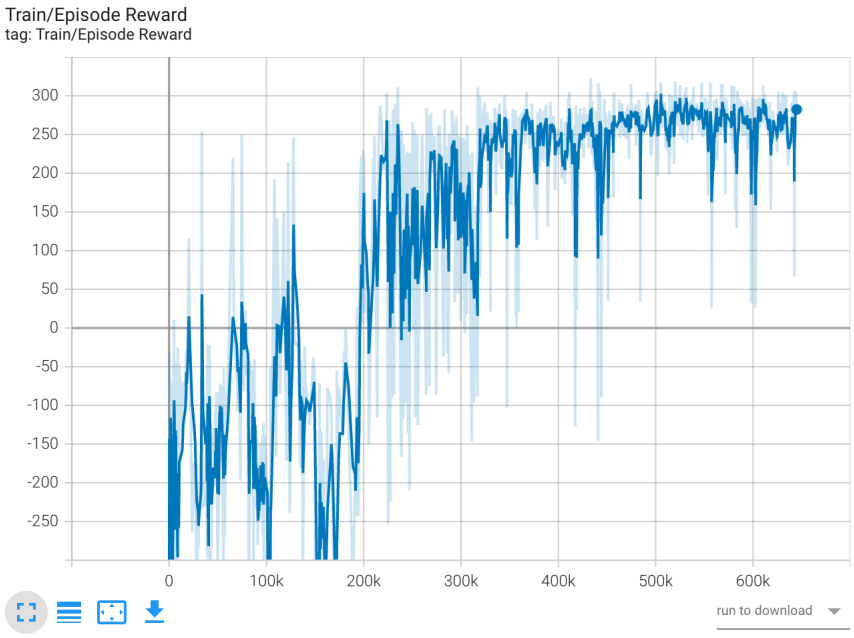

Figure 3: DDPG episode reward with 2000 episodes.

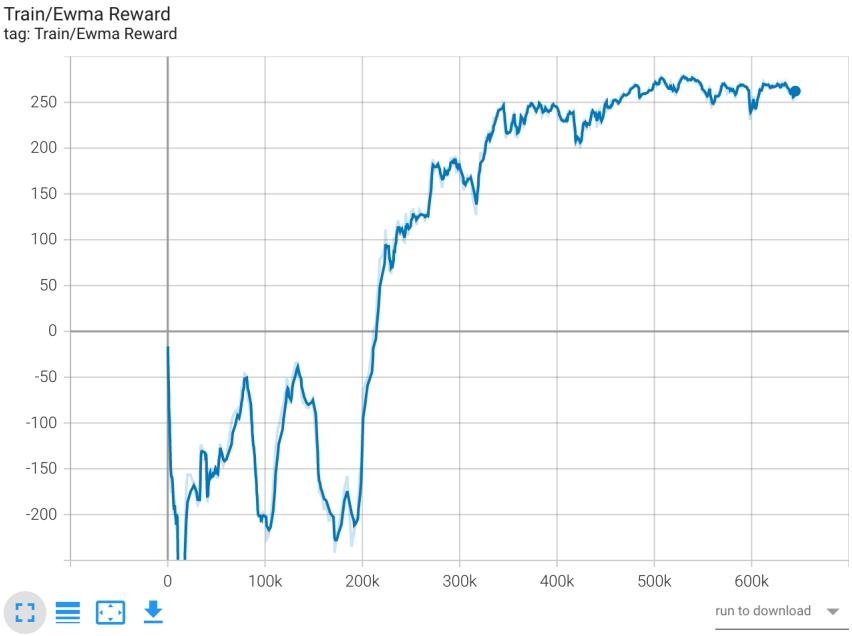

Figure 4: DDPG ewma reward with 2000 episodes.

3 Describe major implementation

3.1 DQN

First, we build a network that can predict $Q(s, a)$. This network is a deep neural network (DNN); it takes an input state and outputs the Q-values of all possible actions for that state. We understand that the input layer of the DNN has the same size as a state size and that the output layer has the size of the number of actions that the agent can take. In our case, the number of actions is 4 (no-op, fire left agent, fire main engine, fire right engine). Also, we set the first hidden layer size to 384 and the second hidden layer size to 256. To summarize, when the agent is at a particular state, it forwards that state through the DNN and chooses the action with the highest Q-value.

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(384,256)):
        super().__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim,hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()

    def forward(self, x):
        ## TODO ##
        x=self.fc1(x)
        x=self.relu(x)
        x=self.fc2(x)
        x=self.relu(x)
        out=self.fc3(x)
        return out
```

Listing 1: DNN network to predict $Q(s, a)$.

We will do the update on the behavior network using gradient descent. We choose Adam as the optimizer with a learning rate of 0.0005.

```python
self._optimizer = Adam(self._behavior_net.parameters(),lr=args.lr)
```

Listing 2: Optimizer for the behavior network.

In order for the agent to choose to explore or exploit, we use ϵ-greedy. We randomly generate a number if the number is larger than ϵ (initial to 1 and gradually decrease to 0.01), then we choose the action a_i with the largest $Q(s, a_i)$ from the behavior network. Otherwise, we let the agent explore.

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    sample=random.random()
    if sample>epsilon:
        with torch.no_grad():
            # t.max(1) will return larget column value of each row
            # second column on max result is index where max element was
            # found, so we pick action with the larger expected reward
            state=torch.from_numpy(state).view(1,-1).to(self.device)
            return self._behavior_net(state).max(1)[1].item()
    else:
        return action_space.sample()
```

Listing 3: Select action based on ϵ-greedy.

To update the behavior network, first, we sample a minibatch of transitions (state, action, reward, next state, done) from the replay memory. Then we get the current $Q(s, a)$ output from the behavior network. And use target network to get $Q(s_{t+1}, a')$ which finds action a' that yields the maximum Q in the next state. Then we get the target Q, y_i by TD learning.

$$y_i = \begin{cases} R_T \text{ for terminal state } s_T \\ R_{t+1} + \gamma \max_{a'} Q\left(s_{t+1}, a'\right) \quad \text{for non-terminal state } s_t \end{cases} \tag{1}$$

Next, we minimize the mean square error between our target Q value (according to the Bellman optimality equation) and our current Q output:

$$L(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}[(y_i - Q(s, a; \theta_i))^2] \tag{2}$$

Optimally, we want the error to decrease, meaning that our current policy's outputs are becoming more similar to the true Q values. Therefore, with the loss function defined as above, we perform a gradient step on the loss function according to:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot),s'\sim\epsilon}[(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)] \tag{3}$$

```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 = self._target_net(next_state).max(dim=1)[0].view(-1,1)
        q_target = reward + gamma*q_next*(1-done) # 1-done: end of trajectory

    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()
```

Listing 4: Updating the behavior network using TD learning and mean square error.

Every 1000 steps, we update the target network by copying from the behavior network. We do this because the behavior network update so frequently that sometimes it may be unstable, so we maintain a stable target network aside from the behavior network.

```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())
```

Listing 5: Update target network by copying the behavior network.

3.2 DDPG

Both DQN and DDPG need a replay memory for exploitation. We sample from replay memory randomly each time we update our behavior network. Each time we sample a minibatch of transitions from the replay memory.

```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))
```

Listing 6: Replay memory.

DDPG used four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

- θ^Q: Q network.
- θ^μ: deterministic policy function.
- $\theta^{Q'}$: target Q network.
- $\theta^{\mu'}$: target policy network.

The Q network and policy network are like simple Advantage Actor-Critic. However, in DDPG, the actor directly maps states to actions (the network's output directly to the output) instead of outputting the probability distribution across a discrete action space. The target networks are the time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improves stability in learning. This is because, in methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to divergence. For example, in Eq.4, $Q(s', a')$ depends Q function itself (at the moment it is being optimized).

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, a) + \gamma \max Q(s', a') - Q(s, a)] \tag{4}$$

So, we have the standard Actor and Critic architecture for the deterministic policy network and the Q network.

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim,hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()
        self.tanh=nn.Tanh()

    def forward(self, x):
        ## TODO ##
        x=self.fc1(x)
        x=self.relu(x)
        x=self.fc2(x)
        x=self.relu(x)
        out=self.fc3(x)
        out=self.tanh(out)
        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)
```

Listing 7: Actor and Critic network of DDPG.

Since we have two networks, so we will need two optimizers to update the network. We use Adam as the optimizer and set the learning rate to 0.001 for both networks.

```python
self._actor_opt = Adam(self._actor_net.parameters(),lr=args.lra)
self._critic_opt = Adam(self._critic_net.parameters(),lr=args.lrc)
```

Listing 8: Optimizer for the two network.

In reinforcement learning, for discrete action spaces, exploration is done via probabilistically selecting a random action (such as ϵ-greedy or Boltzmann exploration), we have done in DQN as well. For continuous action spaces, exploration is done via adding noise to the action itself (there is also the parameter space noise). We use Gaussian noise in our implementation.

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        state=torch.from_numpy(state).view(1,-1).to(self.device)
        action=self._actor_net(state)

        if noise:
            N=torch.from_numpy(self._action_noise.sample())
            N=N.view(1,-1).to(self.device)
            action+=N
        action=action.squeeze().cpu().numpy()

    return action
```

Listing 9: Select action based on the behavior (actor) network and exploration noise.

The behavior network update splits into two parts. (1) Actor (policy) network update. (2) Critic (value) network update. The value network is updated as is done in Q-learning. The Bellman equation obtains the updated Q value:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{5}$$

However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. In the implementation, we used the target actor-network to choose the next action and the target critic network to choose the next-state Q value. Then, we minimize the mean-squared loss between the updated Q value and the original Q value:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{6}$$

Note that the original Q value is calculated with the value network, not the target value network.

For the policy function, our objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}[Q(s, a)|_{s=s_t, a_t=\mu(s_t)}] \tag{7}$$

To calculate the policy loss, we take the derivative of the objective function concerning the policy parameter. Since the actor (policy) function is differentiable, so we have to apply the chain

rule.

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_{\theta^\mu} \mu(s|\theta^\mu) \tag{8}$$

Nevertheless, since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch.

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i [\nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}] \tag{9}$$

We add a negative sign to the policy loss in the implementation since our objective is to maximize the expected return.

```python
def _update_behavior_network(self, gamma):
    actor_net, critic_net = = self._actor_net, self._critic_net
    target_actor_net, target_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()
```

Listing 10: Update the actor and the critic network.

We make a copy of the target network parameters for the target network updates and have them slowly track those of the learned networks via "soft updates". We control the speed of the update by τ, which we set to 0.005.

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \\ \text{where} \quad & \tau \ll 1 \end{aligned} \tag{10}$$

```
@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_((1.0-tau)*target.data+tau*behavior.data)
```

Listing 11: Update the target network by soft copying from the behavior network.

For testing, we run ten episodes, which is equal to 10 trajectories. In each episode, we are given an initial state from the environment. We repeat selecting an action from our well-trained actor-network without exploitation (noise) term and execute the action until the end of the trajectory. We print out the total rewards we get in each episode and the average return of 10 episodes.

```python
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):
            # env.render()

            # select action
            action = agent.select_action(state,noise=False)

            # execute action
            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"Total reward: {total_reward:.4f}")
                rewards.append(total_reward)
                break

    print('Average Reward', np.mean(rewards))
    env.close()
```

4 Difference between your implementation and algorithms

Both DQN and DDPG have a warmup time in our implementation. This happens at the beginning of training. In this short time, we will not update the parameters of the network; we let the agent explore and save the transitions in the replay memory.

In DQN, we do not update the behavior network in every iteration; instead, we update every four iterations.

The original paper of DDPG uses the Ornstein-Uhlenbeck Process to generate the exploration noise, which is correlated with the previous noise, to prevent the noise from canceling out or freezing the overall dynamics. However, in our implementation, we use a simple Gaussian noise instead, which yields good results as well.

5 Implementation and the gradient of actor updating

In policy (actor) learning, We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q(s, a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to the action, we can perform gradient ascent (with respect to policy parameters only) to solve:

$$\max_\theta \mathbb{E}_{s\sim D}[Q(s, \mu_\theta(s))] \tag{11}$$

Note that the Q-function parameters are treated as constants here, which means that the critic is fixed when we update the actor. The calculation of policy (actor) loss and the update of the actor are mentioned in Section 3.2. We copy the context again below.

To calculate the policy loss, we take the derivative of the objective function concerning the policy parameter. Since the actor (policy) function is differentiable, so we have to apply the chain rule.

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s, a) \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

However, since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch.

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i [\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}]$$

We add a negative sign to the loss in the implementation since our objective is to maximize the expected return.

```
## 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()
```

Listing 12: Update actor by maximizing the expected teturn.

6 Implementation and the gradient of critic updating

The Bellman equation describing the optimal action-value function, $Q^*(s,a)$ is given by:

$$Q^*(s,a) = \mathbb{E}_{s' \sim P}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{12}$$

where $s' \sim P$ is shorthand for saying that the next state s' is sampled by the environment from a distribution $P(\cdot|s,a)$. This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q(s,a)$ with parameters θ^Q, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d). We can set a mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q comes to satisfying the Bellman equation:

$$L = \mathbb{E}_{(s,a,r,s'd) \sim \mathcal{D}}[(Q(s,a) - (r + \gamma(1-d) \max_{a'} Q(s',a')))^2] \tag{13}$$

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, primarily minimize this MSBE loss function. All of them employ two main tricks. (1) Replay buffer, in which we will draw a batch of transitions to update the network. (2) Target network, which is the updated (next-state) Q value.

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

In the implementation, we use the target actor-network to get the next action and the target critic network to get the next Q value and applied the equation above to get the target Q value.

We can rewrite Eq.13 as follow:

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

where the original Q value is calculated with the value network, not the target value network.

Also the gradient of the critic loss is

$$\nabla_{\theta^Q} L = \frac{2}{N}(Q(s,a|\theta^Q) - y) \tag{14}$$

```
## 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()
```

Listing 13: Update the critic with mean-squared Bellman loss

7 Effects of the discount factor

The discount factor essentially determines how much the reinforcement learning agents care about rewards in the distant future relative to those in the immediate future. If $\gamma = 0$, the agent will be wholly myopic and only learn about actions that produce an immediate reward. If $\gamma = 1$, the agent will evaluate each of its actions based on the total of all its future rewards. Picking a particular value of γ rewrite an agent's discounted reward G as

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots \tag{15}$$

We concern more about the future near the current state and concern less about the far future. Also, if a trajectory has no end, then without discount factor might get an infinite value on the expected return.

8 Benefits of ϵ-greedy in comparison to greedy action

When the agent explores, it can improve its current knowledge and gain better rewards in the long run. However, it gets more reward immediately when it exploits, even if it is a sub-optimal behavior. As the agent cannot do both at the same time, there is a trade-off.

A greedy agent can get stuck in a sub-optimal state since it never lets itself explore new possibilities, or there might be changes in the environment as time passes. In ϵ-greedy action selection, the agent uses both exploitations to take advantage of prior knowledge and exploration to look for new options.

9 The necessity of the target network

Take DQN as example, if we didn't use the target network, the update target was $(r_t + \max_a Q(s_{t+1}, a; \theta^Q) - Q(s_t, a_t; \theta^Q))^2$, then learning would become unstable because the target $r_t + \max_a Q(s_{t+1}, a; \theta^Q)$ and the prediction $Q(s_t, a_t; \theta^Q)$ are not independent, as they both rely on θ^Q.

We use the target network that does not update as frequently as the behavior network to stabilize the training process.

10 Explain the effect of replay buffer size in case of too large or too small

If the replay buffer is too small, the replay buffer serves little to no purpose. A small buffer might force the network to only care about what it saw recently, and it might overfit and break.

Usually, we want our experience replay buffer to be as large as possible. The larger the experience replay, the less likely we will sample correlated elements; hence the more stable the network's training will be. However, a large experience replay also requires much memory, and it might slow down the training. So, there is a trade-off between training stability and memory requirements.

11 Double-DQN

Double DQN handles the problem of the overestimation of Q-values. From Eq.1, we can see that, by calculating the TD target, we face a simple problem: how are we sure that the best action for the next state is the action with the highest Q-values? We know that the accuracy of Q-values depends on what action we tried and what neighboring states we explored. Consequently, we do not have enough information about the best action to take at the beginning of the training. Therefore, taking the maximum Q-value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q-value than the optimal best action, the learning will be complicated. Double DQN provides the solution. When computing the Q target, we use two networks to decouple the action selected from the target Q-value generation. We use our DQN network to select the best action to take for the next state (the action with the highest Q-value). Then we use our target network to calculate the target Q-value of taking that action at the next state.

$$Q(s, a) \leftarrow r_t + \gamma Q'(s_{t+1}, \arg\max_a Q(s_t, a_t)) \tag{16}$$

where $Q(s, a)$ is the behavior network and $Q'(s, a)$ is the target network. Therefore, double DQN helps us reduce the overestimation of Q values and, as a consequence, helps us train faster and have more stable training.

The only difference between DQN and double DQN is when updating 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)

    ## TODO ##
    q_value = self._behavior_net(state).gather(dim=1,index=action.long())

    with torch.no_grad():
        action=self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
        q_next = self._target_net(next_state).gather(dim=1,index=action.long())
        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()
```

Listing 14: Update the behavior network of DDQN, which takes the action from behavior network and generate the next-state Q value from the target network.

Figure 5 and 6 show the episode and the ewma reward of DDQN respectively.

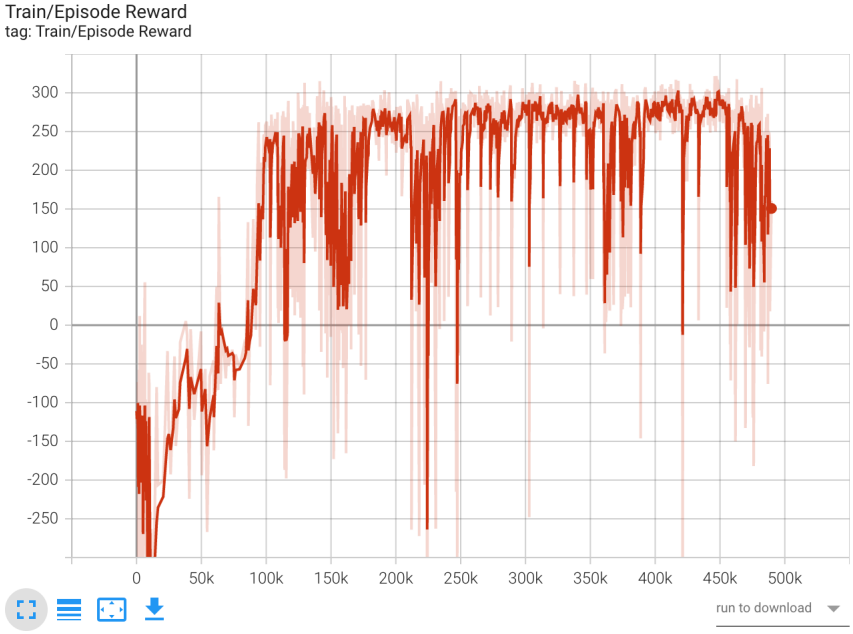

Figure 5: DDQN episode reward with 2000 episodes

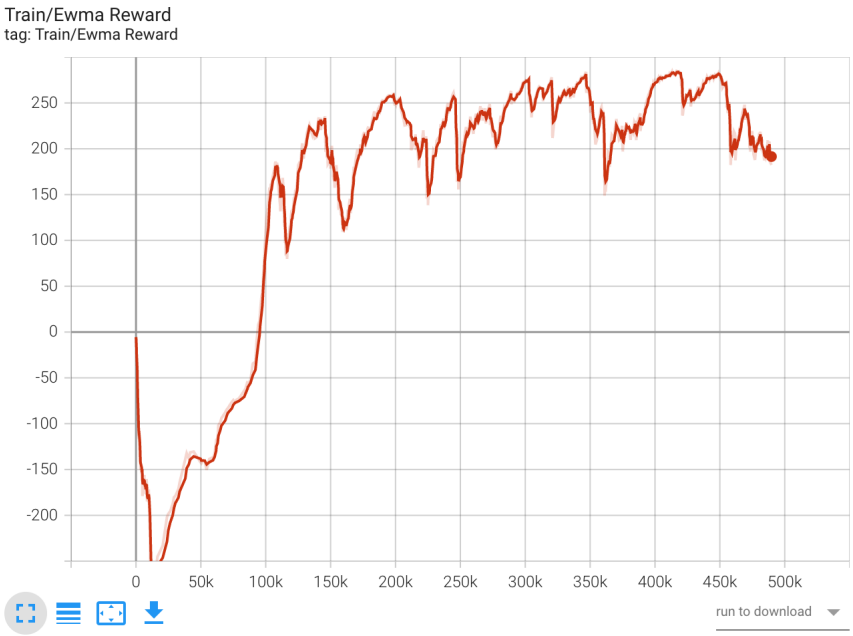

Figure 6: DDQN episode reward with 2000 episodes

12 Results

For all the results, we train for 2000 episodes.

```
Start Testing
Total reward: 247.2968
Total reward: 282.2878
Total reward: 250.9439
Total reward: 262.0189
Total reward: 316.1820
Total reward: 265.7174
Total reward: 278.1898
Total reward: 284.7278
Total reward: 294.1539
Total reward: 297.6763
Average Reward 277.91946829695576
(dqn) zchin@eva-System-Product-Name:~/Deep_Learning_Practice_labs/lab6$
```

Figure 7: DQN testing result. The average reward is 277.

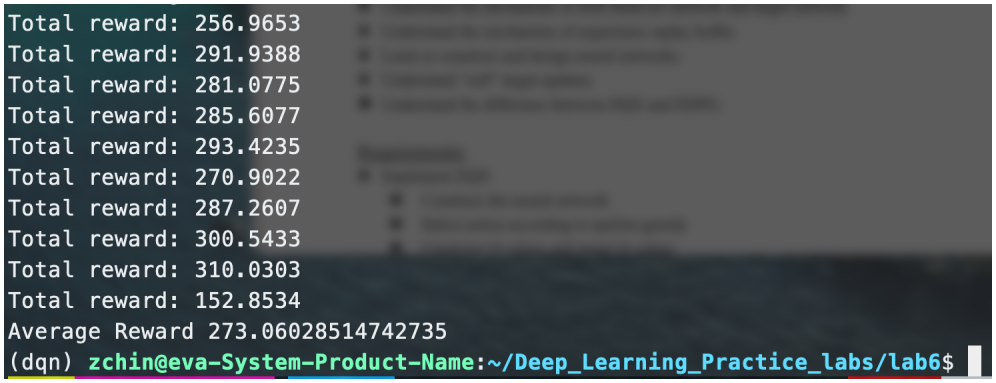

Figure 8: DDPG testing result. The average reward is 273.

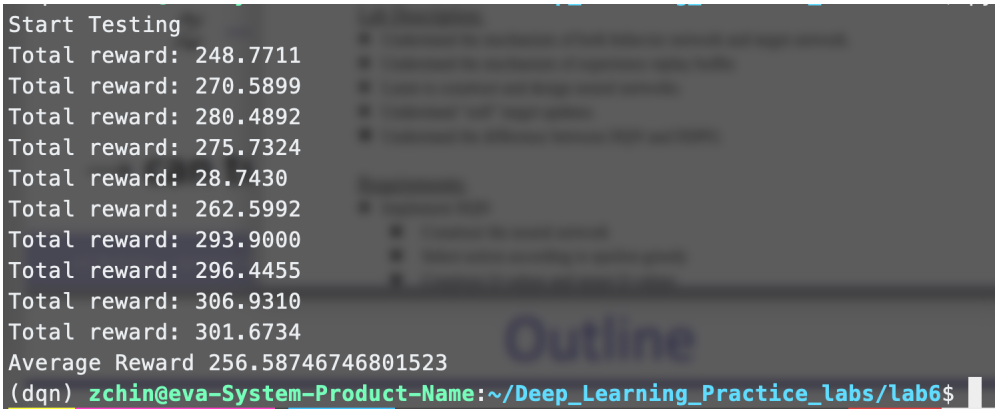

Figure 9: DDQN testing result. The average reward is 256.