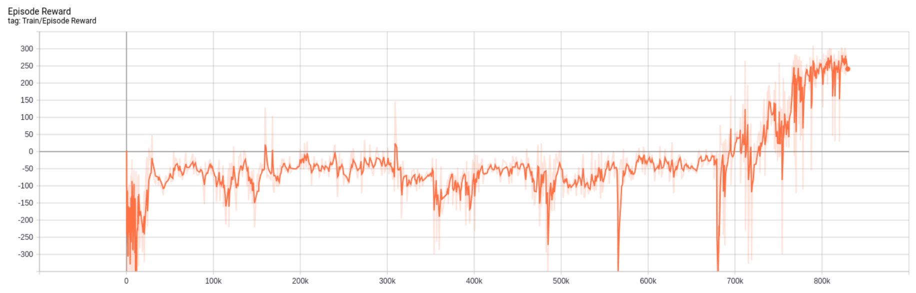
Lab 6: Deep Q-Network and Deep Deterministic Policy Gradient

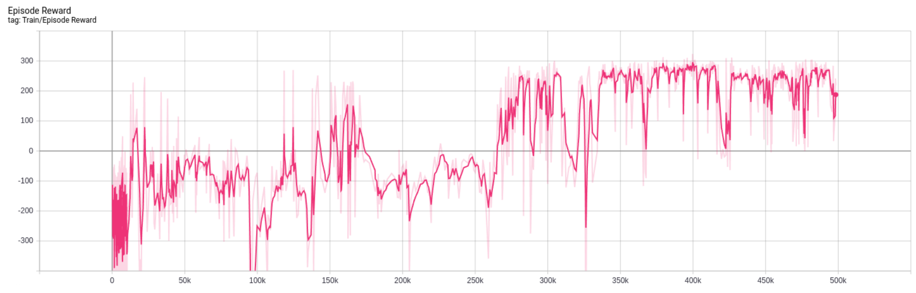
310551083 許嘉倫

1 Episode rewards in LunarLander-v2 and LunarLanderContinuous-v2

Figure 1 shows the tensorboard of LunarLander-v2 and LunarLanderContinuous-v2.



(a) LunarLander-v2 DQN episode reward



(b) LunarLanderContinuous-v2 DDPG episode reward

Figure 1: Training reward in LunarLander-v2 and LunarLanderContinuous-v2

2 Describe your major implementation of both algorithms in detail

2.1 DQN

2.1.1 Net

Implement DQN behavior and target net with 3 fully connected layer, the first two layer followed ReLU activation and the final layer will output q value.

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(inplace=True)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(inplace=True)
        )
        self.fc3 = nn.Linear(hidden_dim, action_dim)

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

2.1.2 optimizer

Use Adam optimizer to train the DQN model.

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

2.1.3 select action

We use epsilon-greedy strategy to select action. If random number less than epsilon, we will random sample an action, otherwise we will use behavior net to choose the action which can get maximum expected q value.

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()

    with torch.no_grad():
        state = torch.tensor(state, device=self.device).view(1, -1)
        outputs = self._behavior_net(state)
        _, best_action = torch.max(outputs, 1)
        return best_action.item()
```

2.1.4 Update behavior network

We will sample random minibatch of transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from replay memory. And compute target value y_j, the formulation is as follows.

- $\hat{Q}$: target net
- $\hat{\theta}$: weights of target net
- γ: discount factor

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

And then use mean square error as loss function to update behavior net.

```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(1, action.long())
    with torch.no_grad():
       q_next = torch.max(self._target_net(next_state), 1)[0].view(-1, 1)
       q_target = reward + q_next * gamma * (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()
```

2.1.5 Update target network

Copy weights of behavior net to update target network 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())
```

2.1.6 Test

When testing the model, we also epsilon-greedy strategy to select action. But we will set ϵ as 0.001, let model to compute expected q value in most cases.

```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()
        for t in itertools.count(start=1):
            if args.render:
                env.render()
            action = agent.select_action(state, epsilon, action_space)
            # 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)
                    rewards.append(total_reward)
                    print(
                    f'Episode: {n_episode}\t\Length: {t:3d}\tTotal Reward: {total_reward:.2f}'
                    )
                    break
        print('Average Reward', np.mean(rewards))
        env.close()
```

2.2 DDPG

2.2.1 Sample of replay memory

It's same as implementation of DQN.

```
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))
```

2.2.2 ActorNet

The implementation of ActorNet is as follows. The first two fully connected layers are followed ReLU acctivation, and the final layer is followed tanh activation. It will output actions of main engine and left-right engine.

```
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.fc1 = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(inplace=True)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(inplace=True)
        )
        self.fc3 = nn.Sequential(
            nn.Linear(h2, action_dim),
            nn.Tanh()
        )

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

2.2.3 CriticNet

The implementation of CriticNet is provided by TA. It will output expected q value according current state and the action which is generated by ActorNet.

```
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)
```

2.2.4 optimizer

Use Adam optimizer to update actor net and critic net.

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

2.2.5 Select action

The output of ActorNet will add noise to generate the next action. Through this way, we can get the effect of exploitation and exploration. However, we only add noise to output of ActorNet in training procedure. And we directly use output of ActorNet as action in testing procedure.

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        state = torch.tensor(state, device=self.device).view(1, -1)
        outputs = self._actor_net(state)
        exploration_noise = torch.tensor(self._action_noise.sample(),
                                         device=self.device).view(1, -1)
        if noise:
            return (outputs + exploration_noise).squeeze(0).cpu().numpy()
        else:
            return outputs.squeeze(0).cpu().numpy()
```

2.2.6 Update behavior network

It's same as DQN. We will sample random minibatch of (s_i, a_i, r_i, s_{i+1}) from replay memory. And compute target value y_i, the formulation is as follows.

- Q: behavior critic network
- u: behavior actor network
- $\hat{Q}$: target critic network
- $\hat{u}$: target actor network
- γ: discount factor
- $y_i = r_i + \gamma * \hat{Q}(s_{i+1}, \hat{u}(s_{i+1}))$

And use mean square error as loss function to update behavior critic network.

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

And update behavior actor network by following formulation.

$$ActorLoss = -\frac{1}{N} \sum_i (Q(s_i, u(s_i))$$

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

## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
```

2.2.7 Update target network

Update target network by soft copy with a ratio τ. The implementation is as follows.

```
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)
```

2.2.8 Test

The implementation is same as DQN. And when we test the model, we should turn off the noise exploration.

```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):
            if args.render:
                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'Episode: {n_episode}\tLength: {t:3d}\tTotal reward: {total_reward:.2f}'
                )
                rewards.append(total_reward)
                break

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

3 Describe differences between your implementation and algorithms

3.1 Warmup of DQN and DDPG

We have a warmup stage in the beginning of training, this warmup stage doesn't appear in algorithms. In warmup stage, we will generate transitions randomly and won't update our models. The transitions which generated randomly will store in replay memory.

3.2 The update frequency of behavior network in DQN

In the implementation, the update frequency of behavior network in DQN is set to 4. It means that we will update behavior network every 4 steps, but it will update every step in the algorithm. Also we can modify our update frequency.

3.3 Save best weight

I will save the best weight of network every 100 episodes by checking if the testing scores of current model weights is better than previous best weights. If so, I will update the best weights.

4 Describe your implementation and the gradient of actor updating.

We want to let behavior actor network predict an action which can get maximum q value by critic network. Use actor network to generate actions, and then get the q value by critic network. Compute the mean q value and back-propagation to update actor network. Therefore, we define actor loss as follows. And it is same as figure 2. We can see the implementation in section 2.2.6

- s: current state.
- Q: behavior critic network.
- u: behavior actor network.

$$Actor\ Loss = -Q(s, u(s))$$

Update the actor policy using the sampled gradient:

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

Figure 2: Actor loss

5 Describe your implementation and the gradient of critic updating

First, sample random minibatch from replay memory, and use actions of batch to calculate corresponding q value by critic network. We also need to calculate target value.
The formulation of target value is as figure 3. First, get action of next state by target actor network, and then generate q value of next state by target critic network. Multiply discount factor and plus reward, we can get the target q value.
Finally, we can calculate MSE loss by q value and target q value. The implementation is as section 2.2.6

Set $y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i\left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$

Figure 3: Critic loss

6 Explain effects of the discount factor

The figure 4 is Q learning algorithm, γ is the discount factor. The smaller discount factor is, the agent will focus on current reward. The larger discount factor is, the agent will focus on future reward.

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

Figure 4: Q learning formulation

7 Explain benefits of epsilon-greedy in comparison to greedy action selection

The model choose the action which is the best for current state, but the result is not necessarily the best for the future. So, to let model explore other action, we use epsilon-greedy to set a probability ϵ which can let model to randomly choose other actions.

8 Explain the necessity of the target network

If we don't have target network, the loss is calculated with behavior q value and behavior target q value. This may cause difficulty for training. So, to avoid this situation, we need target network and copy weights to target network from behavior network every constant steps.

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

If replay buffer size is too large, maybe the model can get better performance because of the number of sample but it will let speed of training too slow. If replay buffer size is too small, it can let speed of training fast but it will let model only focus on recent transitions and result in bad performance.

10 Bonus: Double-DQN

The different between DQN and Double-DQN is target q value like figure 5. Double-DQN can prevent over-optimistic value estimates on DQN. The implementation of DDQN is as follows.

$$Y_t^{Q} = r_{t+1} + \gamma \max_a Q(S_{t+1}, \boxed{a} | \theta^-)$$

$$\Downarrow$$

$$Y_t^{DoubleQ} = r_{t+1} + \gamma Q\left(S_{t+1}, \boxed{\operatorname*{argmax}_a Q(S_{t+1}, a|\theta)} | \theta^-\right)$$

Figure 5: The difference of target q value

```
q_value = self._behavior_net(state).gather(1, action.long())
with torch.no_grad():
   behavior_action = torch.max(self._behavior_net(next_state), 1)[1].view(-1, 1)
   q_next = self._target_net(next_state).gather(1, behavior_action.long())
   q_target = reward + q_next * gamma * (1.0 - done)
```

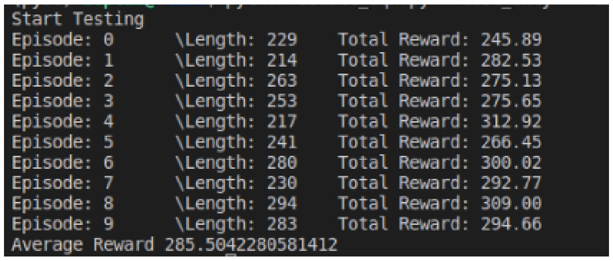
The following table shows the score of DQN and Double-DQN.

	DQN	Double-DQN
Score	266.01	285.50

11 Performance

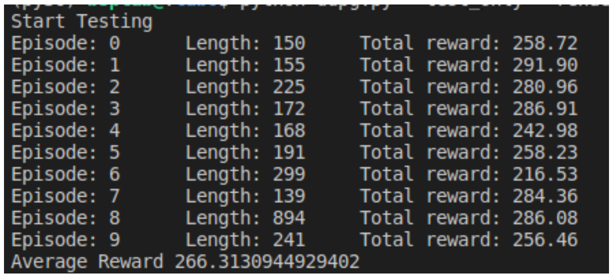
Figure 6 is the average score of DQN and DDPG.

	LunarLander-v2	LunarLanderContinuous-v2
Score	285.50	266.31



```
Start Testing
Episode: 0      \Length: 229    Total Reward: 245.89
Episode: 1      \Length: 214    Total Reward: 282.53
Episode: 2      \Length: 263    Total Reward: 275.13
Episode: 3      \Length: 253    Total Reward: 275.65
Episode: 4      \Length: 217    Total Reward: 312.92
Episode: 5      \Length: 241    Total Reward: 266.45
Episode: 6      \Length: 280    Total Reward: 300.02
Episode: 7      \Length: 230    Total Reward: 292.77
Episode: 8      \Length: 294    Total Reward: 309.00
Episode: 9      \Length: 283    Total Reward: 294.66
Average Reward 285.5042280581412
```

(a) Double-DQN



```
Start Testing
Episode: 0      Length: 150     Total reward: 258.72
Episode: 1      Length: 155     Total reward: 291.90
Episode: 2      Length: 225     Total reward: 280.96
Episode: 3      Length: 172     Total reward: 286.91
Episode: 4      Length: 168     Total reward: 242.98
Episode: 5      Length: 191     Total reward: 258.23
Episode: 6      Length: 299     Total reward: 216.53
Episode: 7      Length: 139     Total reward: 284.36
Episode: 8      Length: 894     Total reward: 286.08
Episode: 9      Length: 241     Total reward: 256.46
Average Reward 266.3130944929402
```

(b) DDPG

Figure 6: Average score of Double-DQN and DDPG