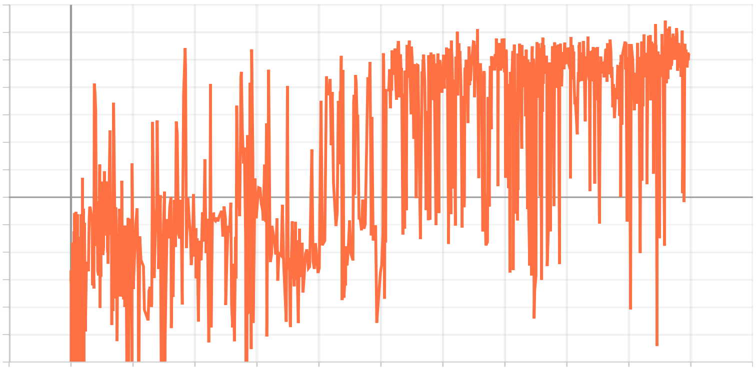
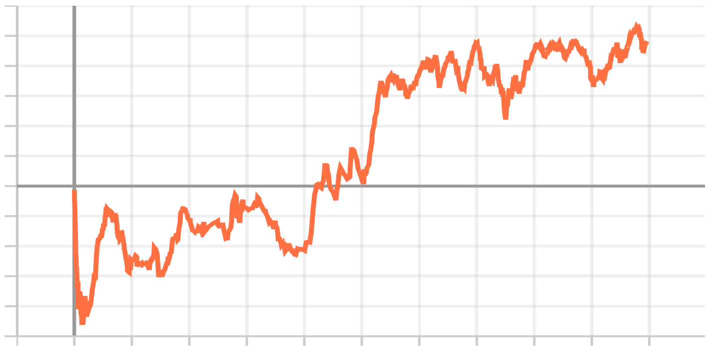
Deep Learning and Practice Lab6 – Deep Q-Network and Deep Deterministic Policy Gradient

310554009 杜葳葳

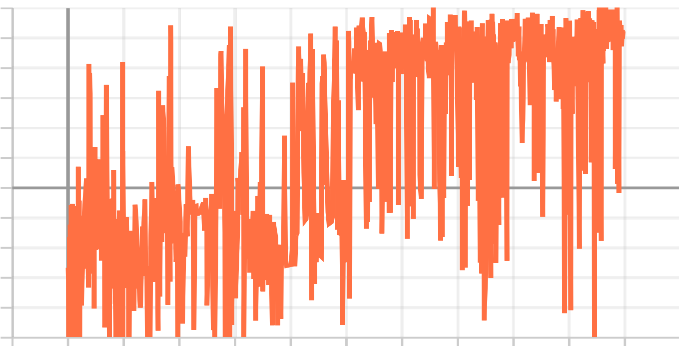
1. **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%)** Smoothing = 0
   - **Train/Episode Reward**



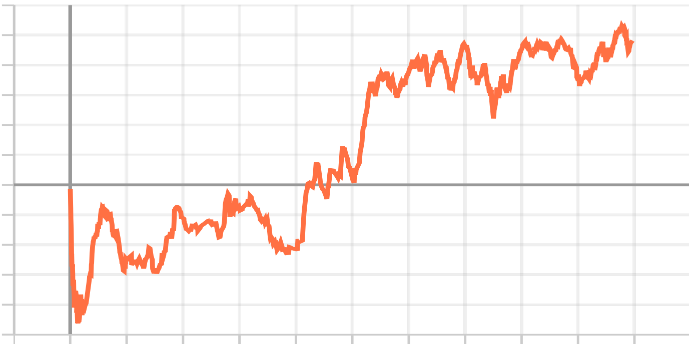
   - **Train/Ewma Reward**



2. **A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2 (5%)**
   - **Train/Episode Reward**



- **Train/Ewma Reward**



3. **Describe your major implementation of both algorithms in detail. (20%)**

   A. DQN

- 依照 spec 的規定，建立三層全連接層、ReLU 的網絡，最後一層輸出 4 維的 expected action

Network Architecture

- Input: an 8-dimension observation (not an image)
- First layer: fully connected layer (ReLU)
  - input: 8, output: 32
- Second layer: fully connected layer (ReLU)
  - input: 32, output: 32
- Third layer: fully connected layer
  - input: 32, output: 4

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4,
        hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.layer3 = nn.Linear(hidden_dim, action_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        ## TODO ##
        output = self.layer1(x)
        output = self.relu(output)
        output = self.layer2(output)
        output = self.relu(output)
        output = self.layer3(output)
        return output
```

- 流程：環境給一個 obs → agent 根據 value function 得到所有 Q (s, a) → 利用 epsilon-greedy 選擇 action 並做出決策 → 環境接收到這個 action 後會給一個 reward 和下一個 obs → 根據 reward 去更新 value function

- DQN 是使用 epsilon-greedy 演算法來輸出 action，為了滿足 exploration 和 exploitation，有 epsilon 的機率隨機抽一個 action (exploration)，剩下 1-epsilon 的機率選擇 value function 最大的 action (exploitation)

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        return
            int(torch.argmax(self._behavior_net(torch
            .Tensor(
            [state]).to(self.device))).detach().cpu()
            .numpy())
```

- 使用 Adam 作為 optimizer

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

- 每 1000 steps 更新一次 target network

```python
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()
```

```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())
```

- 從 replay buffer 中採樣、並更新 network，predicted value 是 behavior network 的輸出、target value 則透過下面的公式計算，接著再使用 gradient descent 在 MSE 來更新參數。

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

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

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

```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)
    q_value_each_action = q_value[0][int(action[0][0])].view(1, 1)
    for i in range(1, action.size(0)):
        q_value_each_action = torch.cat((q_value_each_action,
            q_value[i][int(action[i][0])].view(1, 1)), 0)

    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), 1).values.unsqueeze(1)
        q_target = reward + gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    loss = criterion(q_value_each_action, q_target)
    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

- Testing phase

  與 Training phase 類似，但不用 warming up 和更新 network

```python
for n_episode, seed in enumerate(seeds):
    total_reward = 0
    env.seed(seed)
    state = env.reset()
    ## TODO ##
    for t in itertools.count(start=1):
        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)
            break
    rewards.append(total_reward)
```

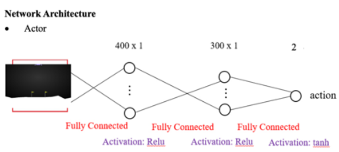
B. DDPG

- DDPG 可以解決連續動作空間的問題，critic 計算 action 的好壞，actor 針對 critic 網路調整參數獲得更好的策略。大致架構與 DQN 相同，也有 replay buffer 和 freezing target network，惟比 DQN 多了 policy network 和 policy target network。

- 從 replay buffer 中隨機抽樣

```python
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))
```

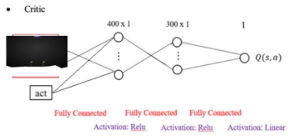
- Actor network 依照 spec，建立三層全連接網路，ReLU 作為全兩層的輸出，tanh 作為最後一層的輸出




```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim[0])
        self.layer2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.layer3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        output = self.layer1(x)
        output = self.relu(output)
        output = self.layer2(output)
        output = self.relu(output)
        output = self.layer3(output)
        output = self.tanh(output)

        return output
```




```python
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)
```

- Adam 作為 optimizer

```python
## 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)
```

- 根據現在的 state 和加上一些 noise (exploration)，來選擇 action

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    return self._actor_net(torch.Tensor([state]).to(self.device)).detach().cpu().numpy()[0] \
        + int(noise) * self._action_noise.sample()
```

- 從 replay buffer 中隨機採樣，計算 predicted value 和 target value 的 MSE 來更新 critic network，其中，predicted value 是 critic network 的 output，target network 透過下面的公式計算

```python
## 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)
```

Sample random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R

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

Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

- 從 actor network 得到 action，Loss 是 critic network 的輸出的 negative mean，因為目標是讓 critic network 的輸出越大越好。

```python
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state, action).mean()
```

- 用 soft copy 來更新 behavior network

```python
@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)
```

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

- Testing phase 和 training 的時候類似，惟增加 warming up 和更新 network。

```python
## TODO ##
for t in itertools.count(start=1):
    action = agent.select_action(state)
    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)
        break
```

4. Describe differences between your implementation and algorithms. (10%)

- warm up 機制 (前 10000 個 steps)，在這個階段不更新，讓訓練更穩定
- network 不會每個 steps 都更新，而是每 4 個 steps 更新一次

5. Describe your implementation and the gradient of actor updating. (10%)

為了讓 critic network 越大越好，先根據 current state 從 action network 獲得 action，Loss 是 critic network 的輸出的 negative mean。

```python
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state, action).mean()
```

6. **Describe your implementation and the gradient of critic updating. (10%)**

從 replay buffer 中隨機抽樣，並對 network 進行更新，用 MSE 來計算 predicted value (critic network 的輸出) 和 target value (reward+gamma*maxQ(s, a)) 的誤差。

```
## 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)
```

7. **Explain effects of the discount factor. (5%)**

Discount factor 是讓當前的 reward 比未來的 reward 更受重視，也就是讓遙遠的 reward 比重比較低，Discount factor 一般小於 1。

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

8. **Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)**

- Greedy action selection: 每次選擇當下 action value 最大的選項，只考慮 exploitation (選擇當下最有利的)、沒有考慮 exploration (隨機選擇，謀取長遠的利益)。
- Epsilon-greedy action selection: 給予一定的機率，讓 agent 可以探索環境 (exploration)，增加當前的知識，以長遠來看可以獲得更多的 reward。

9. **Explain the necessity of the target network. (5%)**

Q function: Q(s, a) = reward + r * max_a' Q(s', a')。

如果只有一個 network，在更新 Q(s, a) 時，target Q(s', a') 也會變化，導致訓練過程不穩定，故需要有兩個 network，behavior network 和 target network，讓 target network 在一定的 episodes 後更新。

10. Explain the effect of replay buffer size in case of too large or too small. (5%)

Training data 之間存在時間上的關聯性，故使用 replay buffer 能夠降低 dependency、讓資料能夠接近 iid，replay buffer 越大、採樣的相關性就越小，訓練也越穩定。

然而，當 replay buffer 很大時，會需要很大的 memory、也需要更長的訓練時間。但當 replay buffer 太小，採樣的相關性越大，也造成訓練時的不穩定。

11. Implement and experiment on Double-DQN (10%)

DDQN 可以解決 DQN 的過估計問題，DDQN 和 DQN 僅在計算 target value 時有差異，DQN 是根據下一個 state 使 target network 的輸出最大、並用公式計算，DDQN 則是根據下一個 state 得到 behavior network 輸出值的 index，然後根據這些 index 作為 target network 每個輸出的 index，並全部連接在一起，用公式得到目標值。

```python
## ddqn
with torch.no_grad():
    max_index = torch.max(self._behavior_net(next_state), 1).indices
    q_next = self._target_net(next_state)
    q_next_each_action = q_next[0][int(max_index[0])].view(1, 1)
    for i in range(1, max_index.size(0)):
        q_next_each_action = torch.cat((q_next_each_action,
            q_next[i][int(max_index[i])].view(1, 1)), 0)
    q_target = reward + gamma * q_next_each_action * (1-done)
```

12. Extra hyperparameter tuning. (10%)

- Capacity 越大越好 (但要考慮 memory 和訓練時間)
- gamma 越高需要越多時間才能收斂
- learning rate 如果太高會 overfitting，但如果太小又會需要花很多時間訓練

13. Performance

A. DQN

```
(dl) wwdu@wwdu-System-Product-Name:~/lab6$ python dqn.py --test_only
Start Testing
Average Reward 233.33363159943983
```

B. DDPG

```
(dl) wwdu@wwdu-System-Product-Name:~/lab6$ python ddpg.py --test_only
Start Testing
Average Reward 260.96550100515225
```