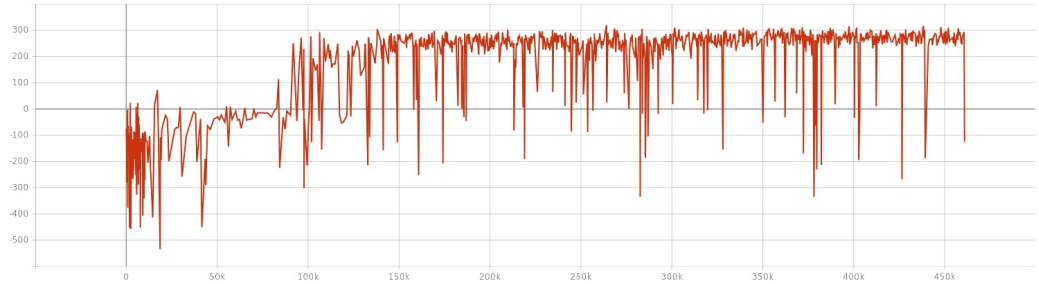
Lab 6: Deep Q-Network and Deep Deterministic Policy Gradient

Hsiang-Chun Yang
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw

1 A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2

Figure 1 是 DQN 的 episode reward 與 EWMA reward，EWMA reward 的 α 設定為 0.05。



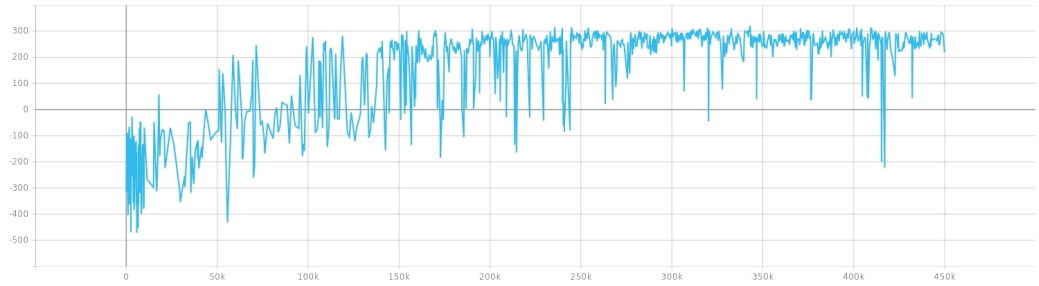
(a) Episode reward

(b) EWMA reward

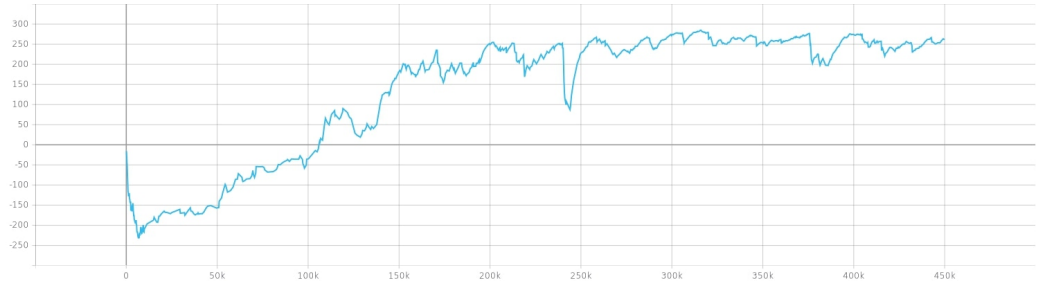
Figure 1: Rewards of DQN

2 A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2

Figure 2 是 DDPG 的 episode reward 與 EWMA reward，EWMA reward 的 α 設定為 0.05。



(a) Episode reward



(b) EWMA reward

Figure 2: Rewards of DDPG

3 Describe your major implementation of both algorithms in detail

3.1 DQN

下面會依照程式碼中 `TODO` 出現的順序來說明。

3.1.1 Net

DQN 利用神經網路模型來近似每個狀態的 target value，我用的模型架構是參考助教提供的 PDF 檔，唯一不一樣的地方只有 hidden layer 的 neuron 數量，我將 `hidden_dim` 從 32 提高到 256，程式碼如下：

```
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 Optimizer

訓練 DQN 時所使用的 optimizer 為 Adam。

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

3.1.3 Action selection

我們使用 epislon-greedy 來選擇下一步的 action，模型會有一定的機率 ε 隨機選擇一個未知的動作，反之則利用 greedy algorithm 去挑選估計值最佳的動作。一開始我們將 ε 設定為 1，使模型著重在 exploration，然後將 ε 指數遞減至 0.01，讓後期的訓練著重在 exploitation，action selection 的程式碼如下：

```
def select_action(self, state, epsilon, action_space):
    ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            state_tensor = torch.tensor(state, device=self.device)
            actions = self._behavior_net(state_tensor.reshape(1, -1))
            return torch.max(actions, dim=1)[1].item()
```

3.1.4 Update behavior network

我們會先從 replay buffer 裡面隨機抽取出一些 $(\phi_i, a_i, r_i, \phi_{i+1})$，對於每個抽取出的樣本計算 target value y_i，計算方式如下，其中 γ 為 discount factor，$\hat{Q}$ 跟 θ^- 是 target network 還有他的 weight。

$$y_i = \begin{cases} r_i & \text{if episode terminated at step i+1} \\ r_i + \gamma \max_{a'} \hat{Q}(\phi_{i+1}, a' | \theta^-) & \text{otherwise} \end{cases}$$

然後用 square error 作為 loss function 並更新 behavior network 的 weights，Q 跟 θ 是 behavior network 還有他的 weight。

$$Loss = (y_i - Q(\phi_i, a_i | \theta))^2$$

程式碼如下：

```
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()
```

3.1.5 Update target network

更新 target network 的方式就相對簡單很多，每隔 C 步之後直接把 behavior network 的 weights 複製到 target network，在實際訓練的時候我將 C 設定為 200 步，程式碼如下：

```
def _update_target_network(self):
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

3.1.6 Testing

測試的時候在 action selection 的地方一樣會用 epsilon-greedy，這時候的 ε 設定為 0.001，主要還是讓模型自己去計算各個 action 的估計值，少數情況才隨機選擇 action，程式碼如下：

```
## 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
```

3.2 DDPG

下面會依照程式碼中 `TODO` 出現的順序來說明。

3.2.1 Replay memory

DDPG 的 replay buffer 的設計基本上跟 DQN 的一樣，所以就直接把 DQN 的程式碼貼過來了。

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

3.2.2 ActorNet

ActorNet 會利用目前的狀態來決定下一步的 action，網路架構是參考助教提供的 PDF 檔，hidden layer 的 neuron 數量直接使用 PDF 上面提供的，程式碼如下：

```
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
```

3.2.3 CriticNet

CriticNet 的程式碼助教已經寫好了，CriticNet 會利用從 ActorNet 得到的 action去計算狀態的估計值，程式碼如下：

```
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)
```

3.2.4 Optimizer

訓練 DDPG 同樣也是使用 Adam optimizer。

```
## 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)
```

3.2.5 Action selection

相較於 DQN 的 epislon-greedy，DDPG 將 `ActorNet` 的輸出與一個隨機的 noise 結合來產生下一步的 action，用以達到 exploitation 與 exploration 的效果，不過這個 noise 只有在訓練階段才使用，測試階段就會直接用 `ActorNet` 的結果當作下一個 action，程式碼如下：

```
def select_action(self, state, noise=True):
    ## TODO ##
    with torch.no_grad():
        state_tensor = torch.tensor(state, device=self.device)
        if noise:
            noise_tensor = torch.tensor(self._action_noise.sample())
            action = (self._actor_net(state_tensor.reshape(1, -1))
                        + noise_tensor.reshape(1, -1).to(self.device))
        else:
            action = self._actor_net(state_tensor.reshape(1, -1))
    return action.squeeze().cpu().numpy()
```

3.2.6 Update behavior network

跟 DQN 一樣，我們會先從 replay buffer 裡面隨機抽取出一些 (s_i, a_i, r_i, s_{i+1})。

在更新 behavior cirtic network 時，我們會先對每個抽取出的樣本計算 target value y_i，計算方式如下，其中 γ 為 discount factor，Q' 跟 $\theta^{Q'}$ 是 target critic network 跟他的 weight，μ' 跟 $\theta^{\mu'}$ 是 target actor network 跟他的 weight。

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

然後用 MSE 作為 loss function 並更新 behavior critic network，Q 跟 θ^{Q} 是 critic network 跟他的 weight。

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

而更新 behavior actor network 時，因為我們的目標是去加強 actor network 決定 action 的能力，所以先利用 behavior actor network 與 behavior critic network 計算狀態的估計值，然後將 loss 定義為下面的公式並更新 behavior actor network。

$$-\frac{1}{N}\sum_i Q(s_i, \mu(s_i))$$

程式碼如下：

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

3.2.7 Update target network

更新 target network 是採用 soft copy 的方式，不會直接把 behavior network 的 weight 複製到 target network，而是給定一個比例 τ，利用下面的公式更新 target network 的 weight，這邊的 τ 設定為 0.05。

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

程式碼如下：

```
@staticmethod
def _update_target_network(target_net, net, tau):
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data.copy_(tau*behavior.data+(1-tau)*target.data)
```

3.2.8 Testing

測試的流程跟 DQN 一樣，程式碼如下，可以發現在 action selection 的時候 `noise` 是 `False`，如同先前提到的，測試的時候直接用 `ActorNet` 的輸出作為 action。

```
## 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
```

4 Describe differences between your implementation and algorithms

4.1 DQN and DDPG

實作的時候在訓練初期會有 warmup 的階段，這是在演算法裡面所沒有的。在 warmup 的過程中，我們不會去更新模型的 weights，而是單純讓電腦不斷地隨機產生動作，並將遊戲的過程紀錄到 replay buffer 中。

4.2 DQN

DQN 的演算法中是每玩一步就會更新一次 behavior network 的 weights，而在實作的時候則是預設每玩四步才會更新一次 weights。

5 Describe your implementation and the gradient of actor updating

我們的目標是去加强 behavior actor network 決定 action 的能力，使 behavior critic network 可以得到較大的估計值，所以將 loss 定義為：

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

我們可以把上面的公式拆成兩個部分，並利用 chain rule 計算 gradient。

$$\begin{cases} a = \mu(s|\theta^{\mu}) \\ Loss = -Q(s, a|\theta^{Q}) \end{cases}$$

$$\frac{\nabla Loss}{\nabla \theta^{\mu}} = -\frac{\nabla Q(s,a|\theta^{Q})}{\nabla a} \frac{\nabla a}{\nabla \mu(s|\theta^{\mu})} \frac{\nabla \mu(s|\theta^{\mu})}{\nabla \theta^{\mu}}$$

程式碼如下：

```
## 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()
```

6 Describe your implementation and the gradient of critic updating

從 replay buffer 裡面隨機抽取一些 state transition，再利用 target critic network Q' 計算的 y 與 behavior critic network Q 的計算估計值去計算 MSE，並更新 behavior critic network，y 跟 MSE 的計算公式如下：

$$\begin{cases} y = r + \gamma Q'(s', \mu'(s'|\theta^{\mu'})|\theta^{Q'}) \\ MSE\ Loss = \frac{1}{N}\sum(y - Q(s,a|\theta^{Q}))^2 \end{cases}$$

Gradient 的推導如下：

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

程式碼如下：

```
# 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()
```

7 Explain effects of the discount factor

當下的 reward 的權重最大，時間軸越往後的 reward 的權重越小，權重成指數型遞減，程式碼中的 gamma就是 discount factor。

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

模型訓練的時候除了從已知的情況中選擇最佳的 action 外（exploitation），探索未知但可能是最佳的 action 也很重要（exploration），我們利用給定一機率 ε 來決定模型下一步要做 exploitation 還是 exploration，這便是 epsilon-greedy。

9 Explain the necessity of the target network

如果我們利用同一個網路當作 behavior network 與 target network，則每次更新 behavior network 的時候 target network 也會一起被更新，在 target value 一直改變的情況下訓練很難穩定下來，所以我們將 behavior network 與 target network 區分開來，並且每隔一段時間才更新 target network，這樣有助於穩定訓練過程。

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

如果 replay buffer 太小，buffer 內會充斥著最近的 episode，在這種情況下，每次 sample 出來的狀態彼此間高段相關，有可能導致模型 overfitting；如果將 replay buffer 擴大，每次 sample 的時候比較有機會看到彼此間關聯性較低的狀態，雖然訓練的時間會變長，但是整個訓練過程會更加穩定，模型也能夠達到比較好的表現。

11 Bonus

11.1 Implement and experiment on Double-DQN

Double-DQN 跟 DQN 主要的差異就是計算 target value 的方式。

$$y^{DQN} = r + \gamma \text{max}_{a'} \hat{Q}(s', a'|\theta^-)$$

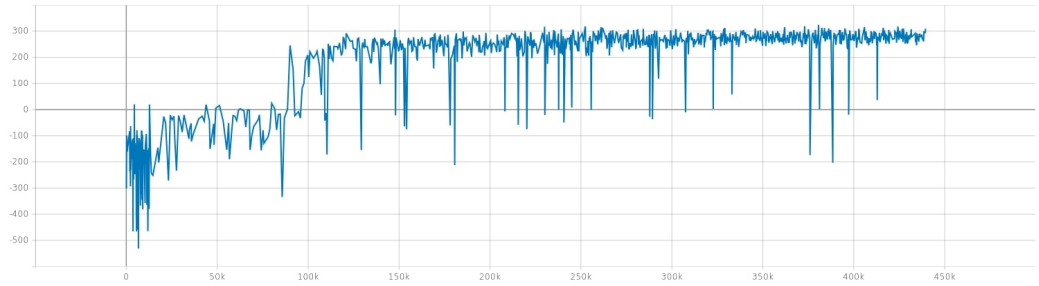
$$y^{Double-DQN} = r + \gamma \text{max}_{a'} \hat{Q}(s', \text{argmax}_a Q(s', a|\theta)|\theta^-)$$

程式碼如下：

```
## 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)
```

```
## Double-DQN ##
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)
```

Figure 3 是 Double-DQN 的 episode reward 與 EWMA reward，EWMA reward 的 α 設定為 0.05。



(a) Episode reward

(b) EWMA reward

Figure 3: Rewards of Double-DQN

12 Performance

12.1 LunarLander-v2

12.1.1 DQN

```
Start Testing
Length: 240     Total reward: 296.21
Length: 182     Total reward: 260.37
Length: 176     Total reward: 259.09
Length: 208     Total reward: 300.25
Length: 193     Total reward: 297.71
Length: 227     Total reward: 258.36
Length: 188     Total reward: 255.64
Length: 227     Total reward: 287.31
Length: 159     Total reward: 246.37
Length: 229     Total reward: 287.46
Average Reward 274.87540081220095
```

12.1.2 Double-DQN

```
Start Testing
Length: 220     Total reward: 283.74
Length: 187     Total reward: 260.40
Length: 165     Total reward: 276.16
Length: 197     Total reward: 302.00
Length: 175     Total reward: 285.23
```

```
Length: 182     Total reward: 275.56
Length: 189     Total reward: 256.07
Length: 217     Total reward: 288.50
Length: 163     Total reward: 258.82
Length: 222     Total reward: 286.85
Average Reward 277.332468580146
```

12.2 LunarLanderContinuous-v2

12.2.1 DDPG

```
Start Testing
Length: 217     Total reward: 262.67
Length: 248     Total reward: 288.17
Length: 210     Total reward: 273.51
Length: 239     Total reward: 253.74
Length: 188     Total reward: 261.43
Length: 179     Total reward: 254.10
Length: 221     Total reward: 258.18
Length: 390     Total reward: 263.51
Length: 242     Total reward: 298.14
Length: 727     Total reward: 283.14
Average Reward 269.65781410770654
```