

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

311581005 智能碩一 吳佳豪

1 A TENSORBOARD PLOT SHOWS EPISODE REWARDS OF AT LEAST 800 TRAINING EPISODES IN LUNARLANDER-V2

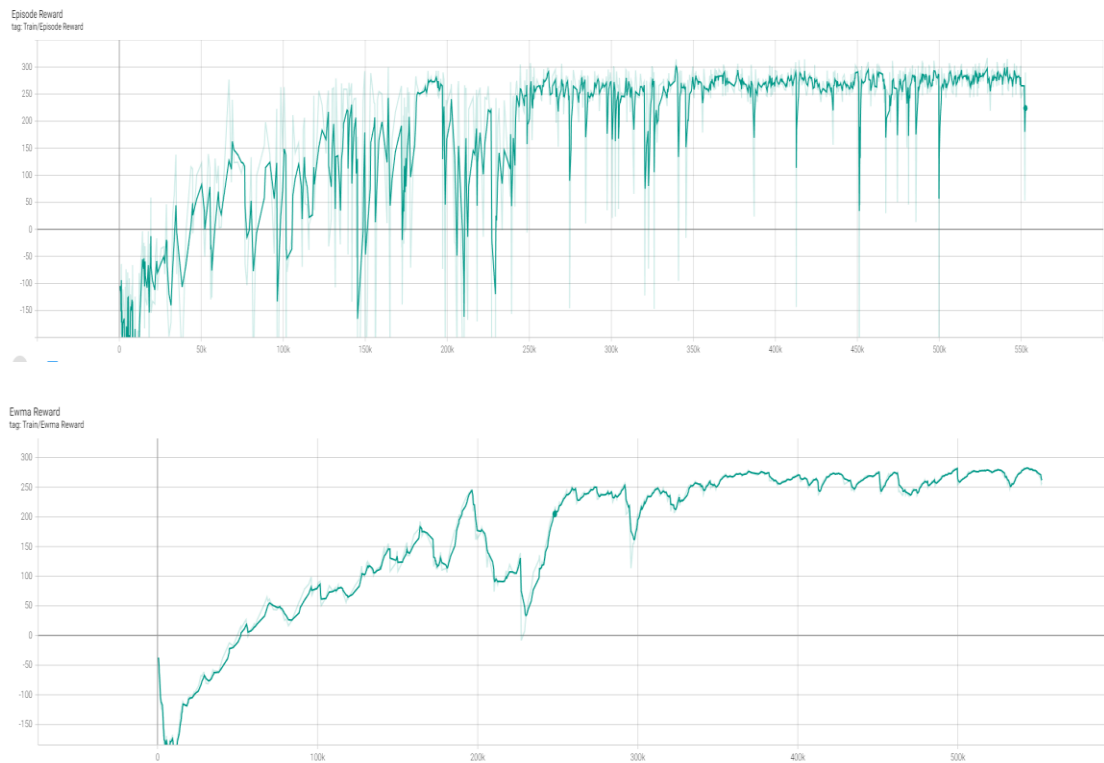


Figure 1. Rewards with DQN

2 A TENSORBOARD PLOT SHOWS EPISODE REWARDS OF AT LEAST 800 TRAINING EPISODES IN LUNARLANDERCONTINUOUS-V2

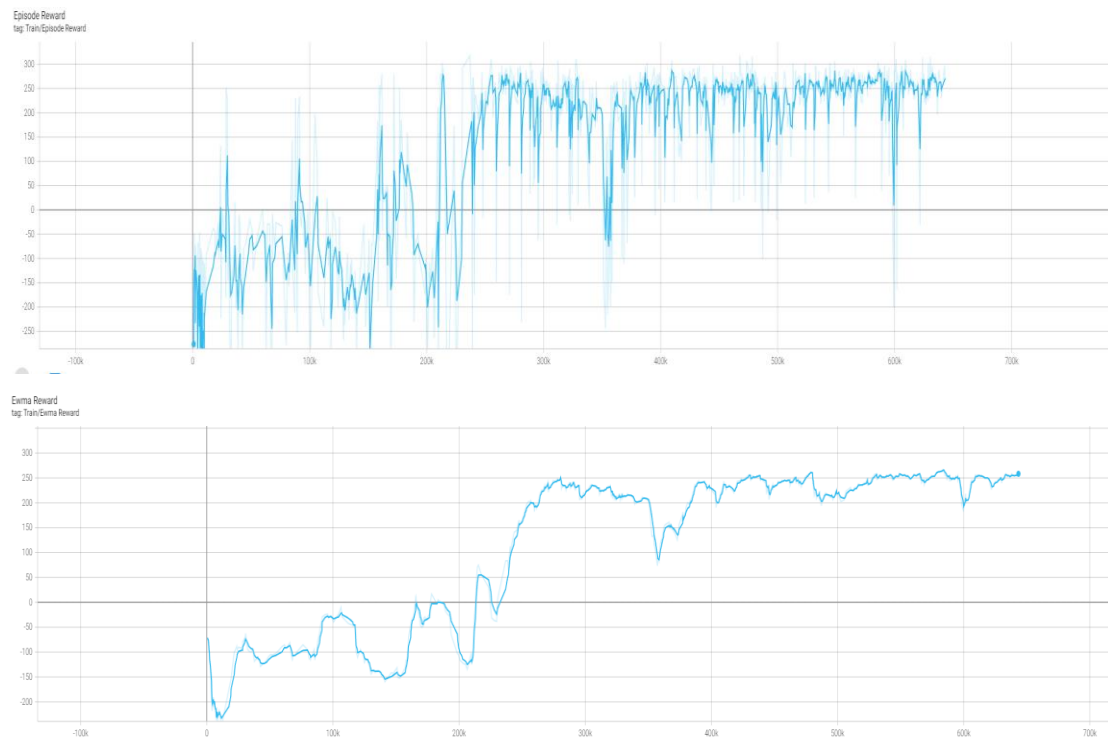


Figure 2. Rewards with DDPG

3 DESCRIBE YOUR MAJOR IMPLEMENTATION OF BOTH ALGORITHMS IN DETAIL.

3.1 DQN

首先下面的程式碼主要建構一個 NN 的 Model 來做到輸入 State 時預測下個 action，因為環境為 LunarLander-v2，所以建構輸入為 State dim=8 而輸出為 action dim=4 的模型，並為了可以方便測試不同 hidden layers，所以建構方式可以自行設定層數與數量大小，經過測試後發現中間 hidden 的層數使用(400,200)的大小就可以得到不錯的成效，因此將此數值設定為預設大小。

```

class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(400,200)):
        super().__init__()
        ## TODO ##
        self.layer_size = len(hidden_dim) + 1
        for i in range(len(hidden_dim) + 1):
            in_feature = hidden_dim[i - 1] if i - 1 >= 0 else state_dim
            out_feature = hidden_dim[i] if i < len(hidden_dim) else action_dim
            setattr(self, f"fc{i}", nn.Linear(in_feature, out_feature))
        self.relu=nn.ReLU()

    def forward(self, x):
        ## TODO ##
        output = x
        for i in range(self.layer_size):
            output = getattr(self, f"fc{i}")(output)
            if i != self.layer_size - 1:
                output = self.relu(output)

        return output

```

Figure 3. Implement action model

下方程式碼實現演算法中的 epsilon-greedy 主要控制 DQN 要選擇哪個 action，如果 random 的數值小於 epsilon 就隨機選擇一種方法，反之則從所有 action 中找到可以獲取到最大的 Q value 的 action。這樣的方法可以增加探索能力，避免只會重複走相同的 action。

With probability ϵ select a random action a_t
otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

Figure 4. Epsilon-greedy algorithm

```

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():
            return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()

```

Figure 5. Implement the method of using epsilon-greedy in DQN

下方程式碼實現如 Figure 6.中的演算法，先隨機從 buffer 中 sample 一個 batch 大小的資料，接著將 state 資料傳給 behavior_network 中來得到選擇特定 action 的 Q value，之後將 next state 輸入到 target_network 中來得到最大的 Q value 並使用 bellman Equation 計算 Q target 值，得到了 Q value 與 Q target 就可以計算 loss 且更新 behavior_network 模型參數，然後經過 c 個 step 後再更新 target_network 的參數，並重複下去進行訓練。

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

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

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

Every C steps reset $\hat{Q} = Q$

Figure 6. Some train part in DQN algorithm

```
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(dim=1)[0].view(-1, 1)
        q_target = reward + (gamma * q_next * (1 - done))
    loss = nn.MSELoss()(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())
```

Figure 7. Implement train part in DQN algorithm

3.2 DDPG

DDPG 是由 actor 和 critic 兩個 network 組合而成的架構，下方為實作兩個 model 的程式碼，經過測試後發現 hidden layers 使用(400,200)有不錯的效果，所以將數值設為 hidden layers 的預設大小。

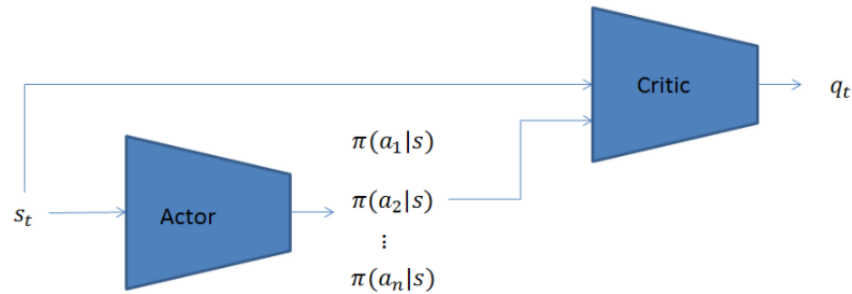


Figure 8. DDPG architecture

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 200)):
        super().__init__()
        ## TODO ##
        self.layer_size = len(hidden_dim) + 1
        for i in range(len(hidden_dim) + 1):
            in_feature = hidden_dim[i - 1] if i - 1 >= 0 else state_dim
            out_feature = hidden_dim[i] if i < len(hidden_dim) else action_dim
            setattr(self, f"fc{i}", nn.Linear(in_feature, out_feature))
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        output = x
        for i in range(self.layer_size):
            activated_func = self.relu if i != self.layer_size - 1 else self.tanh
            output = activated_func(getattr(self, f"fc{i}")(output))

        return output

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)
```

Figure 9. Implement Actor and Critic network

下方程式碼主要實作 DDPG 在回傳 action 時，會選擇 actor_net 傳回的 action 並加入雜訊在其中，來增加探索的能力。

Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise

Figure 10. The method of selecting the action in DDPG

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

Figure 11. Implement the method of selecting the action in DDPG

下方程式碼實作如 Figure 12. 演算法，主要與 DQN 訓練方式差不多，差在會將 actor 的輸出接到 critic 的輸入，並且使用 target_actor 來拿到下個 action 的值後傳入 target_critic 後得到 Q value 並使用 bellman Equation 計算 Q target 值，得到了 Q value 與 Q target 可以計算 loss 並更新 critic 網路，再來會將 state 傳入 actor 中取得當前的 action，並將當前 action 傳到 critic 中得到輸出平均後取負值當成 loss 並更新 actor 網路。

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

Update the actor policy using the sampled gradient:

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

Update the target networks:

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

Figure 12. Some train part in DDPG algorithm

```

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 = target_actor_net(next_state)
        q_next = target_critic_net(next_state, a_next)
        q_target = reward + gamma*q_next*(1-done)
    critic_loss = nn.MSELoss()(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_((1-tau)*target.data + tau*behavior.data)

```

Figure 13. Implement some train part in DDPG algorithm

4 DESCRIBE DIFFERENCES BETWEEN YOUR IMPLEMENTATION AND ALGORITHMS.

多一個 warmup 的機制，在剛開始訓練時會隨機的選擇 action 來增加 replay buffer 的資料，並且在遊玩的過程中不會更新模型參數，來達到隨機探索的功效，豐富資料的多樣性。

5 DESCRIBE YOUR IMPLEMENTATION AND THE GRADIENT OF ACTOR UPDATING.

會將 state 傳入 actor 中取得當前的 action，並將當前 action 傳到 critic 中得到輸出平均後取負值當成 loss 並更新 actor 網路（實作可參考[標題 3.2](#)）。

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 14. The algorithm of the gradient of actor updating

6 DESCRIBE YOUR IMPLEMENTATION AND THE GRADIENT OF CRITIC UPDATING

使用 `target_actor` 來拿到下個 `action` 的值後傳入 `target_critic` 後得到 `Q value` 並使用 `bellman Equation` 計算 `Q target` 值，得到了 `Q value` 與 `Q target` 就可以計算 `loss` 並更新 `critic` 網路（實作可參考[標題 3.2](#)）。

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

Figure 15. The algorithm of the gradient of critic updating

7 EXPLAIN EFFECTS OF THE DISCOUNT FACTOR.

隨著時間越久對給予的 `reward` 影響應該要越來越少，代表越剛發生的 `reward` 應該 `reward` 要影響最大，所以下個值通常會乘上一個介於 $0 < \gamma < 1$ 的值，來達到逐漸遞減。

8 EXPLAIN BENEFITS OF EPSILON-GREEDY IN COMPARISON TO GREEDY ACTION SELECTION

如果只使用 `greedy` 的方法，在一開始訓練時找到一條較高的 `reward` 時，就會使網路每次遇到相同狀況都選擇相同 `action`，而不探索新的 `action`，會限制網路。而使用 `epsilon-greedy` 時，因為會隨機的選擇不同的 `action`，所以可以解決想是遇到剛好此次 `action` 較差但卻是最佳解路徑的情況，使網路有探索(`exploration`)的能力。

9 EXPLAIN THE NECESSITY OF THE TARGET NETWORK

得知 $Q\pi(st, at) = rt + Q\pi(st + 1, \pi(st + 1))$ ，而 $Q\pi(st, at)$ 為輸出，目標為 $rt + Q\pi(st + 1, \pi(st + 1))$ 是會變動的，所以如果都使用相同的 `network` 時，因為模型參數會一直更新，因此會使訓練極為不穩定，為了改善此問題而有了 `target network`，將 `target network` 的 $Q\pi$ 固定住後即 $rt + Q\pi(st + 1, \pi(st + 1))$ 變成固定，那對於 `behavior network` 就只是解單純的 `regression` 的問題，等迭代幾次後再將 `target network` 的 $Q\pi$ 更新。


10 EXPLAIN THE EFFECT OF REPLAY BUFFER SIZE IN CASE OF TOO LARGE OR TOO SMALL

如果 `buffer` 過於大的話，雖然可以使訓練穩定，但是也會使得訓練速度變慢。如果 `buffer` 過小時，因為只會存有最近的 `data` 資料，所以會重於最近遊玩的狀態，而導致 `overfitting` 的狀況出現。

11 IMPLEMENT AND EXPERIMENT ON DOUBLE-DQN

發現因為在 DQN 中會發生高估 Q value 的情況發生，所以修改下方公式，原本是從 target_network 中取得，變成從在 behavior_network 中可以獲得最大 Q value 的 action 後，在將 action 當成 target_network 的參數傳入並經過 Bellman 取得 Q target。

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



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

Figure 16. Between DQN and DDQN different part

```
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_index = self._behavior_net(next_state).max(dim=1)[1].view(-1, 1)
        q_next = self._target_net(next_state).gather(dim=1, index=action_index.long())
        q_target = reward + (gamma * q_next * (1 - done))
    loss = nn.MSELoss()(q_value, q_target)

    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

Figure 17. Implement update_behavior_network function in DDQN

下方為測試的實驗結果：

```
python .\ddqn.py --test_only --seed 1
Start Testing
total reward: 298.27
total reward: 261.48
total reward: 293.68
total reward: 294.55
total reward: 296.20
total reward: 305.21
total reward: 253.42
total reward: 310.99
total reward: 295.05
total reward: 295.46
Average Reward 290.43137977015004
```

Figure 18. The result of DDQN

12 [LUNARLANDER-V2] AVERAGE REWARD OF 10 TESTING EPISODES

```
python dqn.py --test_only --logdir ./result/dqn --seed 1
Start Testing
total reward: 309.21
total reward: 280.46
total reward: 300.77
total reward: 300.17
total reward: 288.62
total reward: 304.85
total reward: 283.40
total reward: 290.50
total reward: 285.37
total reward: 294.77
Average Reward 293.811974394545
```

Figure 18. The result of DQN

13 [LUNARLANDERCONTINUOUS-V2] AVERAGE REWARD OF 10 TESTING EPISODES:

```
python ddpg.py --test_only --seed 1
Start Testing
total reward: 303.30
total reward: 279.97
total reward: 304.22
total reward: 305.29
total reward: 250.26
total reward: 297.48
total reward: 281.13
total reward: 279.65
total reward: 288.31
total reward: 303.02
Average Reward 289.2637501056921
```

Figure 18. The result of DDPG