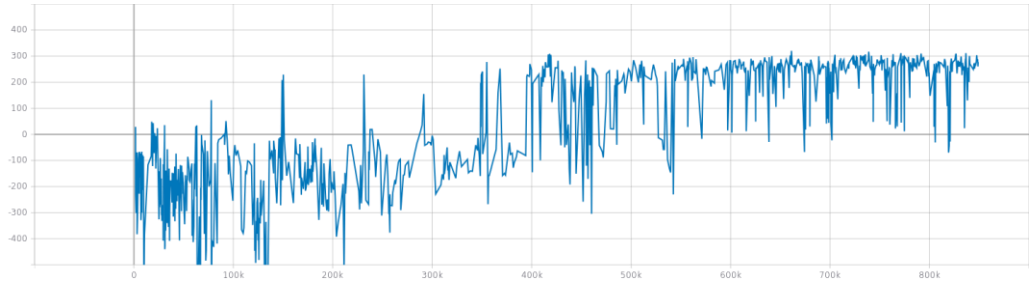
Lab6 : Deep Q-Network and Deep Deterministic Policy Gradient

310605005 王映勻

Report

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

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



3. **Describe your major implementation of both algorithms in detail.**

   ✓ **DQN**

   在一般 Q-learning 的方法中，會建立一個表格儲存所有 action 和 state 所對應的 Q value，而 DQN 就是引入神經網絡代替原本的 Q 值表，以避免記憶體爆炸的情況。

   這次使用 DQN 的環境為 LunarLander-v2，其有 8 種 state、4 種 action，因此設計的神經網路架構，其輸入為 8-dimension，輸出為 4-

dimension，中間有一層 32-dimension 的 hidden layer。

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        # raise NotImplementedError
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, action_dim)
        self.relu = nn.ReLU()

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

在 select action 的部份，這邊採用 epsilon-greedy 的方式。設定一個 0-1 的 epsilon 值，每次隨機選擇一個 0-1 的值，大於 epsilon 就選擇分數最高的 action，反之則隨機選擇一個 action，讓模型能夠有隨機探索的機會。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    # raise NotImplementedError
    if random.random() < epsilon:  # Exploration
        return action_space.sample()
    else: # Exploitation: pick action with the larger expected reward
        with torch.no_grad():
            return self._behavior_net(torch.from_numpy(state).
                view(1,-1).to(self.device)).max(dim=1)[1].item()
```

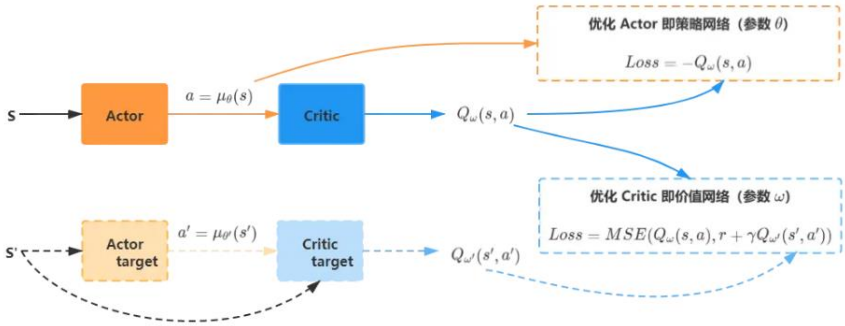
DQN 在訓練過程中將計算 target Q value 的神經網路跟訓練用的神經網路分開，這邊分為 target_network 和 behavior_network 。

在 update_behavior_network 的部分，每次會從 replay memory 所建立的 transition buffer 隨機取樣一個 mini batch，以確保訓練資料不會有連續 step 間的相關性。接著將 state 傳入 behavior_network，得到選擇 action 時用的 q_value，再將 next_state 傳入 target_network，利用其得到的 q_next 計算出 q_target 之後，就可將兩者的 mean square error 作為 loss，利用 back-propagation 更新 behavior_network。

而 target_network 的部分，在訓練一開始，其參數設定會和 behavior_network 一樣，依據我們的設計，behavior_network 是每 4 個 iteration 更新一次，而 target_network 則是每 1000 個 iteration 才會與 behavior_network 的參數同步，以維持訓練的穩定性。

```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()

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 ##
    # raise NotImplementedError
    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)
    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()

def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    # raise NotImplementedError
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

✓ **DDPG**



DDPG 的架構是基於 Actor-Critic 並結合 DQN 的思想，使其不僅像 DQN 可以處理離散型動作問題，也可以處理本次 LunarLander Continuous-v2 這種連續型動作問題。

如上圖所示，actor 和 critic 分別由 target network (_target_actor_net & _target_critic_net)和 behavior network (_actor_net & _critic_net)構成，所以在 DDPG 中相當於總共有 4 個 network 架構。

Actor network 的架構如下，參考助教給的 Network Architecture，其輸入為 8-dimension state，輸出為 2-dimension action，中間有 400*1 和 300*1 共兩層 hidden layer。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        # raise NotImplementedError
        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 ##
        # raise NotImplementedError
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))
        out = self.tanh(self.fc3(out))
        return out
```

Critic network 的部分，其將兩個獨立的輸入 observation 和 action concatenate 在一起，使最後的輸出為 1-dimension 的純量(Q value)。

```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)
```

在 select action 的部份，在 actor network 回傳 action 給環境之前，會加上隨機的 noise，增加探索 action space 的機會。

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    # raise NotImplementedError
    with torch.no_grad():
        if noise:
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))+\
                 torch.from_numpy(self._action_noise.sample().astype('float32')).view(1,-1).to(self.device)
        else:
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
    return re.cpu().numpy().squeeze()
```

在 update_behavior_network 的部分，critic network 的更新方式與 DQN 類似，差別只在這邊必須先經過_target_actor_net 得到 a_next，再輸入到_target_critic_net 才能得到 q_next。而 actor network 的更新方法，是將_actor_net 的輸出傳給_critic_net，然後將其負輸出當成 loss，利用 back-propagation 完成網路的更新。

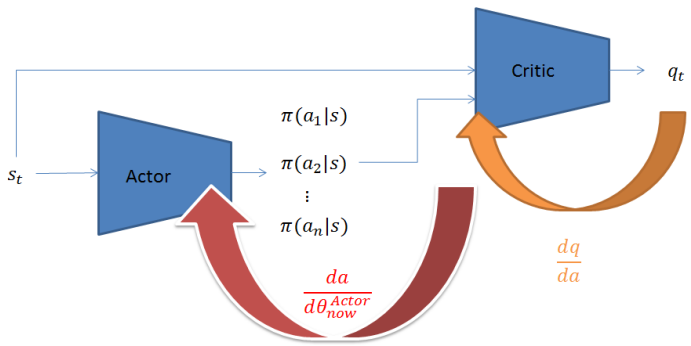
```python
def _update_behavior_network(self, gamma):
    actor_net, critic_net, target_actor_net, target_critic_net = self._actor_net, self._critic_net,
                                                                 self._target_actor_net, self._target_critic_net
    actor_opt, critic_opt = self._actor_opt, 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
    # raise NotImplementedError
    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 ##
    # raise NotImplementedError
    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()
```

4. **Describe differences between your implementation and algorithms.**

在 training 的時候，依據我們設定的 warmup 值，每個 episode 在剛開始的一段時間，會採取隨機探索的策略，也就是隨機從 action space 中選擇 action，將 transition 存進 buffer，直到 warmup 結束才開始 update network 的參數。

另外在 DQN 的部份，並不是每個 iteration 都會更新 behavior network，而是依據我們設定的 frequency，每隔一段時間才會更新一次。

5. **Describe your implementation and the gradient of actor updating.**



如上圖所示，actor network 的目標是要找到一個 action 能使得輸出的 Q value (q_t)最大，因此要優化 actor network 的梯度就是要最大化 critic network 輸出的 Q value，再利用 back-propagation 更新網路，而將輸出取負號當成 Loss 是為了方便做最小化。公式及實際做法如下：

Update the actor policy using the sampled gradient:

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

```
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.**

利用 target network 得到的 y_i 和 behavior network (_critic_net)得到的 Q(s, a)，計算 mean square error 作為 loss，利用 back-propagation 來更新網路。公式及實際做法如下：

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

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

```
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.**

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

上一題所提到的 y_i 以及上面公式的 y_j 分別為 DDPG 和 DQN 計算 total reward 的方式，其中的 γ 就是 discount factor ($\gamma \in [0,1]$)。從公式可看出，discount factor 會使未來所給的 reward 影響越來越小，目的就是要讓當前 reward 的影響力比未來的 reward 更大，但若是未來的 reward 本身影響巨大，還是能左右現在的選擇。

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

一般 greedy 的策略是每次都選分數最高的 action，這樣做雖然可以快速收斂但容易卡在 local optimum。而 epsilon-greedy 的作法是在選擇行動時，依據設定的 epsilon 值，有一定的機率隨機探索，也就是隨機從 action space 中選擇 action，反之就是選擇 Q value 最佳的 action，使模型在大部分時間裡採用現階段最優策略，在少部分時間裡實現探索，達到平衡的關係。

9. Explain the necessity of the target network.

由於數據本身存在不穩定性，每一輪迭代都可能產生一些波動，如果只有使用單一網路訓練，模型通過當前時刻的回報和下一時刻的價值估計隨時進行更新，這些波動會立刻反映到下一個迭代的計算中，這樣就很難得到一個穩定的模型，因此引入 target network 就是為了解決這個問題。

Target network 和 behavior network 不同，不會每一輪迭代都更新，而是要經過一定次數的迭代，才會同步 behavior network 的參數，這樣做就可以讓 Q-Learning 的 target value 不會一直波動，模型也能夠有一定的穩定性。

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

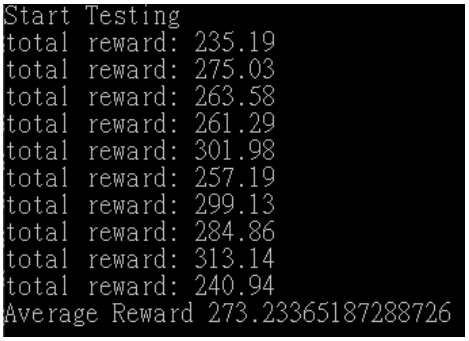
Replay buffer size 越大，訓練會越穩定，但資料量多就會需要更長時間才能收斂。而若 buffer size 太小，裡面的資料就會很相近，容易造成 overfitting 的問題 。

Performance

(Average reward of 10 testing episodes)

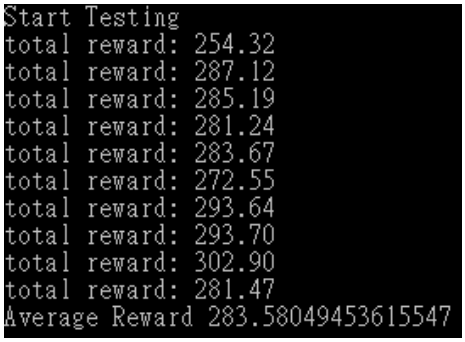
1. **[LunarLander-v2]**

   Average ÷ 30 = 9.108



2. **[LunarLanderContinuous-v2]**

   Average ÷ 30 = 9.453



Bonus

1. **Implement and experiment on Double-DQN (DDQN)**

   DQN 容易出現高估 Q 值的問題，而 DDQN 就是為了解決這個問題誕生的優化版本。參考”Deep Reinforcement Learning with Double Q-

Learning” 這篇論文，發現 DDQN 的架構大致上與 DQN 差不多，關鍵是差在 update behavior betwork 時 q_target 如何取得。

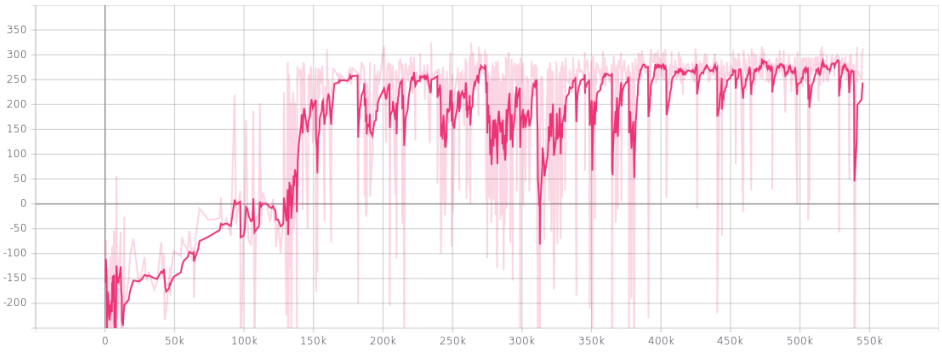
$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t')$$

DQN 是直接從 target_network 取得 $max\ Q(S,a)$ 作為 q_next，而 DDQN 是先用 behavior_network 找到 $max\ Q(S,a)$ 對應的 index，再根據這個 index 從 target_network 取得 q_next。

程式實作及結果如下，原本和 dqn 訓練時遇到一樣的問題，分數結果變化很大，且 ddqn 的上限更遇到瓶頸，但調整一些參數及增加 hidden 的層數後問題就解決了，且表現應該是有比原本 dqn 再好一點：

```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 ##
    # raise NotImplementedError

    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)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```




```
Start Testing
total reward: 255.77
total reward: 286.04
total reward: 284.41
total reward: 279.42
total reward: 292.46
total reward: 271.49
total reward: 289.97
total reward: 294.35
total reward: 314.42
total reward: 297.44
Average Reward 286.5763725871952
```

2. Implement and experiment on TD3 (Twin-Delayed DDPG)

由於 DDPG 是起源於 DQN，因此同樣會有高估 Q 值的問題，而 TD3 就是為了解決這個問題的 DDPG 優化版本，其主要引入了下面三種方法：

(1) Clipped Double-Q Learning：

跟 Double DQN 解決 Q 值過估計的做法一樣，使用了兩套 critic network 來估算 Q 值，並使用較小的值作為更新的目標

(2) Delayed" Policy Updates：

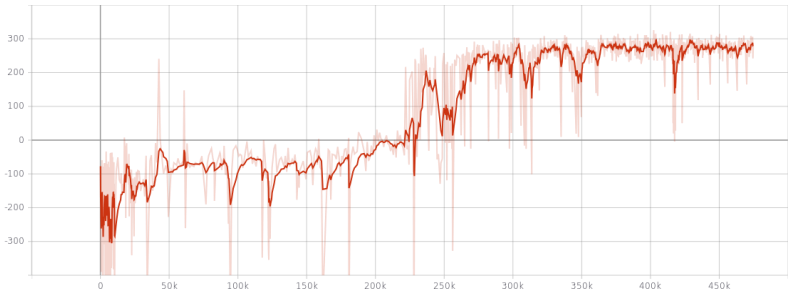
降低 actor network 更新的頻率，使訓練較為穩定

(3) Target Policy Smoothing：

在 target_actor_net 中加入了 noise，使 Q 函數變得平滑避免 overfitting 的問題

程式實作及結果如下，從 tensorboard 的訓練圖可以看出來，雖然收斂時間看起來比 ddqn 慢，但其實整個訓練過程模型是穩定很多的，不像 ddqn 分數跳動很大：

```python
def _update_behavior_network(self, gamma, args, epoch):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(self.batch_size, self.device)

    ## update critic ##
    # critic loss
    ## TODO ##
    q1_value = self._critic_net1(state, action)
    q2_value = self._critic_net2(state, action)
    with torch.no_grad():
    ### Target Policy Smoothing ###
       noise = torch.ones_like(action).data.normal_(0, args.policy_noise).to(self.device)
       a_next = self._target_actor_net(next_state) + noise
       ### Clipped Double-Q Learning ###
       q1_next = self._target_critic_net1(next_state, a_next)
       q2_next = self._target_critic_net2(next_state, a_next)
       q_next = torch.min(q1_next, q2_next)
       q_target = reward + gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    critic_loss1 = criterion(q1_value, q_target)
    critic_loss2 = criterion(q2_value, q_target)

    # optimize critic
    self._actor_net.zero_grad()
    self._critic_net1.zero_grad()
    self._critic_net2.zero_grad()
    critic_loss1.backward()
    critic_loss2.backward()
    self._critic_opt1.step()
    self._critic_opt2.step()

    ### "Delayed" Policy Updates ###
    if epoch % args.policy_delay == 0:
        ## update actor ##
        # actor loss
        ## TODO ##
        action = self._actor_net(state)
        actor_loss = -self._critic_net1(state, action).mean()

        # optimize actor
        self._actor_net.zero_grad()
        self._critic_net1.zero_grad()
        self._critic_net2.zero_grad()
        actor_loss.backward()
        self._actor_opt.step()
```




```
Start Testing
total reward : 250.54
total reward : 289.52
total reward : 274.44
total reward : 277.10
total reward : 317.59
total reward : 267.47
total reward : 296.25
total reward : 303.38
total reward : 297.34
total reward : 295.32
Average Reward 286.89550901683316
```