Racerer car

M11252004 謝奇容

Outline

Q1 sensors (observation) & actuators

Q2 RL model-DDPG

Q3 Reward Function

Q4 Learning curve

Q5 Run Code

Q1

- Sensors
  - Lidar
  - Velocity
  - Acceleration
  - Pose
- Actuators
  - Motor
  - Steering
  - Speed

Sensors - Lidar

- 雷達是所有感測器中最重要的一項
- 透過雷達，知道車子與牆壁的距離，判斷是否撞牆，學習避免撞牆，還有該在哪裡轉彎
- 透過雷達的數值，調整獎勵，讓車子學習不要撞牆或靠著牆壁行駛
- 於訓練過程中發現，即使車子直直撞到牆壁，無法前進，state中的wall_collision參數也大多是回傳False，雷達的值最小也大約有0.36，跟實際車子與牆壁的距離有誤差

Sensors - Velocity

- 期望透過速度的感測，調整action中的速度，讓車子在直線時能全速前進，轉彎處能減速轉彎，避免撞牆
- 於訓練過程中發現，靠牆行駛會讓車子減速
- 車子有從訓練中學會於直線賽道時不要靠著牆壁，盡量全速往前衝

Sensors - Acceleration

- 期望透過加速度的感測，讓車子計算轉彎時機與轉彎角度
- 於訓練過程中發現，加速度對車子的學習沒太多影響，反而會影響學習效果

Sensors - Pose

- 期望車子能藉由姿勢的感測，計算方向盤轉向角度，於轉彎處轉彎，並回正於賽道中間
- 於訓練過程中發現，車子的確會在轉彎處轉彎，但轉彎角度過小，不足以通過彎道，需靠著牆壁才能轉彎。但於訓練後期，能看到車子在碰到牆壁前就會調整姿勢，以正確的姿勢靠牆，才不會被牆壁卡住

Actuators - Motor

- 控制引擎方向，才能控制車子要前進還是後退
- 訓練過程中發現，要使車子倒車較困難，但車子還是有在撞牆後嘗試倒車
- 引擎數值會影響加速度

Actuators - Steering

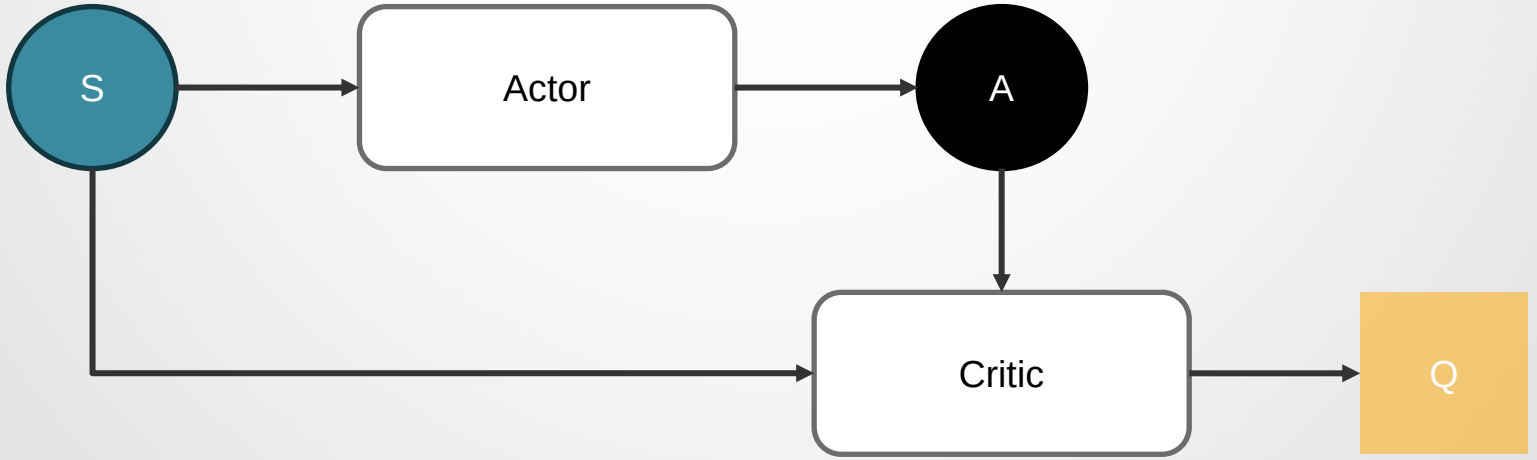
- 控制方向盤角度才能控制車子行進方向
- 訓練過程中發現，較難讓車子急轉彎，要轉角度小的彎很困難
- 即使steering的值是1或-1，也需要一些空間才能讓車子轉彎

Actuators - Speed

- 目標速度決定車子行進的最快(最慢)速度
- 引擎數值引響加速度，進而達到目標速度。但若碰到牆壁，無法達到目標速度

Q2

- DDPG（ deep deterministic policy gradient ）
  - 與DQN想法相似，但能解決DQN不能處理連續性動作的問題
  - 訓練過程是將state放入Actor，尋找哪個動作能得到最大的Q值，輸出動作後，再將動作放入Critic網路預測Q值



DDPG - Actor

- Actor類別會根據輸入的狀態，輸出動作
- __init__內，透過三個全連接層，建立策略網路，以便映射出動作
- 將狀態放入forward中(x)，接著透過兩個激活函數引入非線性性，將x放入三個全連接層，接收傳回的連續動作的值，最後透過tanh將動作值限定在[-1,1]間，回傳x

```
21  class Actor(nn.Module):  # 根據輸入的狀態，輸出動作的策略網路
22      def __init__(self, state_dim, action_dim, max_action):
23          super(Actor, self).__init__()
24          self.l1 = nn.Linear(state_dim, 400)
25          self.l2 = nn.Linear(400, 300)
26          self.l3 = nn.Linear(300, action_dim)
27
28          self.max_action = max_action
29
30
31      def forward(self, x):
32          x = F.relu(self.l1(x))
33          x = F.relu(self.l2(x))
34          x = self.l3(x)
35          x = torch.tanh(x)
36
37          return x
```

DDPG - Critic

- Crictic會根據輸入的狀態與動作，預測Q值，評估動作的好壞，並學習逐漸減小預測與實際得到的Q值的誤差
- 此網路一樣由三個全連接層組成
- 將狀態(x)、動作(u)放入forward中，接著透過兩個激活函數引入非線性性，將x放入三個全連接層，接收傳回的預測q值

```
40  class Critic(nn.Module):  # 根據輸入的狀態與動作，評估Q值
41      def __init__(self, state_dim, action_dim):
42          super(Critic, self).__init__()
43
44          self.l1 = nn.Linear(state_dim + action_dim, 400)
45          self.l2 = nn.Linear(400, 300)
46          self.l3 = nn.Linear(300, 1)
47
48      def forward(self, x, u):
49          x = F.relu(self.l1(torch.cat([x, u], 1)))
50          x = F.relu(self.l2(x))
51          x = self.l3(x)
52          return x
```

DDPG

- DDPG這個類別即為代理人
- 初始化時，首先定義各參數，分別建立Actor、Actor_target、Critic、Critic_target四個模型，並使用Adam優化器優化模型
- 於下頁介紹init_memory()與資料儲存

```
55  class DDPG(object):
56      def __init__(self, state_dim, action_dim, max_action, device, directory):
57          self.device = device
58          self.directory = directory
59          self.memory_size = 50000
60          self.state_dim, self.action_dim, self.max_action = state_dim, action_dim, max_action
61
62          self.actor = Actor(1098, action_dim, max_action).to(self.device)
63          self.actor_target = Actor(1098, action_dim, max_action).to(self.device)
64          self.actor_target.load_state_dict(self.actor.state_dict())
65          self.actor_optimizer = optim.Adam(self.actor.parameters(), lr = 1e-4)
66
67          self.critic = Critic(1098, action_dim).to(self.device)
68          self.critic_target = Critic(1098, action_dim).to(self.device)
69          self.critic_target.load_state_dict(self.critic.state_dict())
70          self.critic_optimizer = optim.Adam(self.critic.parameters(), lr=1e-3)
71
72          self.init_memory()
```

DDPG

- init_memory()是代理人初始化時執行，定義出資料儲存的格式與範圍，共儲存observation內的四個參數、new_observation、action的三個數字、自己定義的reward以及done
- Store_transition()是訓練時，每個step都會執行的函數，若是第一次執行，會先建立儲存次數的計數變數memory_counter，之後用memory_counter判斷儲存空間是否已滿，若滿了會照儲存順序淘汰舊的資料，存入新的資料。
- 儲存時使用np.array格式，因reward和done都是單一變數，不是list，所以要轉型成1,1的np.array
- 儲存完，將memory_counter + 1

```
75      def init_memory(self):
76          self.memory = {
77              "s" : np.zeros((50000, 1098),dtype=np.float64),
78
79              "a" : np.zeros((50000, 3),dtype=np.float64),
80              "r" : np.zeros((50000, 1),dtype=np.float64),
81              "s_" : np.zeros((50000, 1098),dtype=np.float64),
82              "d" : np.zeros((50000, 1),dtype=bool),
83          }
84
85      def store_transition(self, s, a, r, s_, d):
86          if not hasattr(self, 'memory_counter'):
87              self.memory_counter = 0
88          if self.memory_counter <= self.memory_size:
89              index = self.memory_counter % self.memory_size
90          else:
91              index = np.random.randint(self.memory_size)
92          self.memory["s"][index] = s
93          self.memory["a"][index] = a
94          self.memory["r"][index] = np.array(r).reshape(-1,1)
95          self.memory["s_"][index] = s_
96          self.memory["d"][index] = np.array(d).reshape(-1,1)
97          self.memory_counter += 1
```

DDPG

- sample()於DDPG更新時會呼叫，從記憶體中隨機抽取100個資料，讓DDPG學習與更新
- select_action()於每個step呼叫，透過呼叫actor網路，得到網路提供的動作
- reward_func()於每個step呼叫，取代原本環境提供的reward，在我期望的目標給予更多獎勵，我不希望出現的狀態給予懲罰。原本環境的reward沒有懲罰，且即使每次都走到差不多的地方，但獎勵值常常差距不小，才決定直接另外計算獎勵替代原本的reward。此獎勵函數於Q3詳細說明

```
 99      def sample(self):
100          ind = np.random.randint(0, self.memory_counter, size=100)
101          b_s, b_a, b_r, b_s_, b_d = [], [], [], [], []
102          for i in ind:
103              b_s.append(np.array(self.memory['s'][i], copy=False))
104              b_a.append(np.array(self.memory['a'][i], copy=False))
105              b_r.append(np.array(self.memory['r'][i], copy=False))
106              b_s_.append(np.array(self.memory['s_'][i], copy=False))
107              b_d.append(np.array(self.memory['d'][i], copy=False))
108
109          return np.array(b_s), np.array(b_a), np.array(b_r), np.array(b_s_), np.array(b_d)
110
111      def select_action(self, state):
112          state = torch.FloatTensor(state).to(self.device)
113          return self.actor(state)[0].cpu().detach()
114
115      def reward_func(self, observation, state, observation_, state_, best_progress):
116          if(state_['progress'] == 1):  # 若完成一圈
117              return 100                # 給予高額獎勵
118          elif (state_['progress'] > best_progress):   # 若探索到新的地方(好奇心獎勵)，新的進度有5倍獎勵
119              return (best_progress - state['progress']) * 100 + (state_['progress'] - best_progress) * 500
120          elif(state_['progress'] > state['progress']):  # 若車子有前進
121              return (state_['progress'] - state['progress']) * 100 # 獎勵為前進的距離
122          elif(state_['progress'] == state['progress']): # 若車子停在原地
123              return -0.0001                             # 給予懲罰
124          elif(state_['progress'] < state['progress']):  # 若車子往回走
125              return -0.00005                            # 給予小懲罰
```

DDPG

- 每一次更新時，會執行200次
- 先呼叫sample，得到100筆資料，並更改成torch.Tensor的型態
- 接著使用self.actor_target預測動作，critic_target預測Q值
- 使用反向傳播方法，更新模型
- 最後，使用soft update的方法，更新self.actor_target、self.critic_target

```
128     def update(self):
129 
130         for it in range(200):
131             s, a, r, s_, d = self.sample()
132             state = torch.FloatTensor(s).to(self.device)
133             action = torch.FloatTensor(a).to(self.device)
134             reward = torch.FloatTensor(r).to(self.device)
135             state_ = torch.FloatTensor(s_).to(self.device)
136             done = torch.FloatTensor(d).to(self.device)
137 
138             target_Q = self.critic_target(state_, self.actor_target(state_))
139             target_Q = reward + (done * 0.99 * target_Q).detach()
140 
141 
142             self.critic_optimizer.zero_grad()
143             self.critic_optimizer.step()
144 
145 
146 
147             self.actor_optimizer.zero_grad()
148             self.actor_optimizer.step()
149 
150             for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
151                 target_param.data.copy_(0.005 * param.data + (1 - 0.005) * target_param.data)
152 
153             for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
154                 target_param.data.copy_(0.005 * param.data + (1 - 0.005) * target_param.data)
```

DDPG

- save()於訓練時的每個Episode都會執行，將此次訓練結果儲存下來
- load()於再次訓練，或測試時執行，讀取已存在電腦中的模型

```
158     def save(self):
159         torch.save(self.actor.state_dict(), self.directory + 'actor.pth')
160         torch.save(self.critic.state_dict(), self.directory + 'critic.pth')
161 
162     def load(self):
163         self.actor.load_state_dict(torch.load(self.directory + 'actor.pth'))
164         self.critic.load_state_dict(torch.load(self.directory + 'critic.pth'))
165         print("model has been loaded")
```

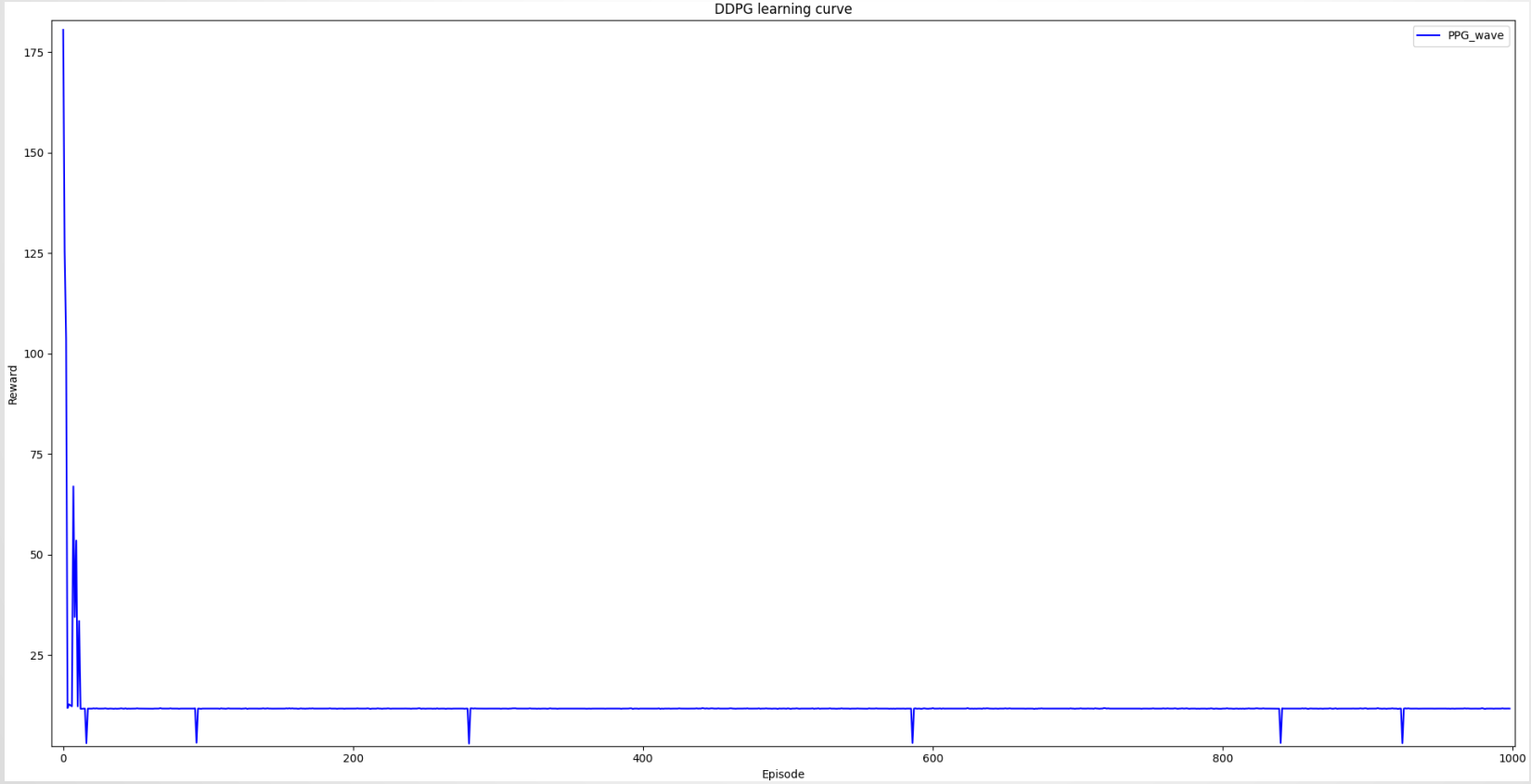
Q3

- 此次訓練的目標，希望車子能越走越遠，因此於抵達新進度時給予好奇心獎勵，前進時給予普通獎勵，若車子停在原地，給予懲罰，若車子倒退，可能在嘗試新的轉彎姿勢，因此只給予小懲罰。若完成一圈賽道，完成目標，給予高額獎勵。
- 訓練過程發現，這樣的設計，在沒有到新進度時，獎勵值相差不多，即使行進距離差了一個check point，獎勵值可能也只差了0.01 ~ 0.1而已。因此，關於普通獎勵的給予，還須調整

```
131     def reward_func(self, observation, state, observation_, state_, best_progress):
132         if(state_['progress'] == 1):  # 若完成一圈
133             return 100                # 給予高額獎勵
134         elif (state_['progress'] > best_progress):    # 若探索到新的地方(好奇心獎勵)，新的進度有5倍獎勵
135             return (best_progress - state['progress']) * 100 + (state_['progress'] - best_progress) * 500
136         elif(state_['progress'] > state['progress']):  # 若車子有前進
137             return (state_['progress'] - state['progress']) * 100 # 獎勵為前進的距離
138         elif(state_['progress'] == state['progress']): # 若車子停在原地
139             return -0.0001                             # 給予懲罰
140         elif(state_['progress'] < state['progress']):  # 若車子往回走
141             return -0.00005                            # 給予小懲罰
```

Q4

- Learning curve
  - 獎勵區間(3 ~ 180)
  - 於下頁說明



Learning Curve

- 因為設有好奇心獎勵，當車子行駛超過目前的最遠距離後，獎勵會大幅提升，所以於訓練初期，獎勵會非常高
- 訓練前期，獎勵值起起伏伏，因為不是每次都能更往前進。約在第10 Episode時，走到此次訓練的最遠距離，因此往後的訓練無法取得好奇心獎勵。後續獎勵值大多相似，因為車子大約能行駛到一定的距離，因此能領到的獎勵值相差不多。但因為模型還不穩定，也有少數次數無法行駛那麼遠，很早就結束回合，所以獎勵值會突然降很低

Q5

- 執行檔案夾中的play.py
- 此python檔案中，包含Actor、Critic、DDPG、Play等需要用到的類別與函數，並透過讀取qnet_ddpg資料夾中的模型，進行遊玩
- 於下頁說明主程式的程式碼

play.py

- 主程式執行時，會先讀取環境設定檔，建立環境，定義遊玩過程中需要的各種參數。之後建立代理人，讀入模型檔，就可以遊玩遊戲了
- 下頁說明def play()

```
267  scenario = './scenarios/custom.yml'     # 設定環境檔路徑
268  frame_size = dict(width=640, height=480) # default frame size
269
270  env = gymnasium.make(                    # 建立環境
271      id='SingleAgentRaceEnv-v0',
272      scenario=scenario,
273      render_mode='rgb_array_birds_eye',
274      render_options=frame_size
275  )
276
277  s_dim = flatten_space(env.observation_space).shape[0]    # 定義狀態的維度
278  a_dim = flatten_space(env.action_space).shape[0]         # 定義動作空間的維度
279  a_bound = flatten_space(env.action_space).high           # 定義動作的最大值
280  a_low_bound =flatten_space(env.action_space).low         # 定義動作的最小值
281  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")    # 決定模型訓練的地方
282  directory = './qnet_ddpg/'                               # 定義模型存放的路徑
283  agent = DDPG(s_dim, a_dim, a_bound, device, directory)   # 建立代理人
284  agent.load()                                             # 代理人讀取訓練完的模型檔
285  output_path = f'./videos_ddpg/racecar-episode-evaluate.mp4'   # 定義影片輸出路徑
286  play(env, agent, output_path)                            # 遊玩一次遊戲
287  env.close()                                              # 關閉環境
```

def play()

- play函數一開始先初始化動作探索率、遊戲是否結束、執行步數、總獎勵、撞牆次數等變數，並重置環境
- 建立envState、state兩個變數，儲存上一個狀態的資訊
- 於每一個step中，取得動作，與環境互動，將observation的內容串成一維陣列

```python
def play(env, agent, output_path = None, i: int | None=0, best_progress = 0):
    var = 3   # 動作探索率
    end_episode = False  # 是否結束此episode
    with torch.no_grad():
        #Reset env
        state = env.reset()

        if output_path is None:             # 若沒有指定影片輸出路徑
            output_path = f'./videos_ddpg/racecar-episode-{i}.mp4'  # 存成訓練episod的影片

        if not os.path.exists(os.path.dirname(output_path)):  # 若影片輸出路徑不存在
            os.mkdir(os.path.dirname(output_path))            # 建立路徑

        step = 0                            # 代理人執行步數
        total_reward = 0                    # 得到的總獎勵

        output_video_writer = cv2.VideoWriter(filename=output_path,           # 建立影片輸出的格式
                                              fourcc=cv2.VideoWriter_fourcc(*'mp4v'),
                                              fps=10,
                                              frameSize=(640,480))


        envState = state[0]   # 儲存observation的內容
        state = state[1]      # 儲存state的內容


        # 將observation的內容串聯成一個一維陣列
        envState_convert = np.concatenate((envState['pose'],envState['acceleration'],envState['velocity'],envState['lidar']))
        wall = 0  # 儲存撞牆次數
        while True:  # 當episode未停止
            action = agent.select_action(envState_convert)  # 代理人選擇動作
            action = np.clip(np.random.normal(action, var), a_low_bound, a_bound)  # 對動作做正態擬合
            action = dict(zip(env.action_space.keys(), action)) # 將動作轉成字典型態

            observation, rewards, done, _, states = env.step(action)  # 與環境互動，得到回傳參數
            envState_ = observation
            # 將envState_的內容串聯成一個一維陣列
            envState__convert = np.concatenate((envState_['pose'],envState_['acceleration'],envState_['velocity'],envState_['lidar']))
```

def play()

- 取得獎勵
- 將執行過程的畫面記錄下來
- 判斷是否撞牆，撞牆次數累積達600次，即停止遊戲
- 更新步數、總獎勵、狀態資訊
- 遊戲結束後，釋放影片編輯，儲存影片

```
234            reward = agent.reward_func(envState, state, envState_, states, best_progress)  # 從獎勵函數取得獎勵值
235
236            if step % 10 == 0:          # 若執行10個step，儲存一幀影片
237                image = env.render().astype(np.uint8)
238                frame = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
239                output_video_writer.write(frame)
240
241            # 若車子沒有前進，且雷達最小值小於0.4，代表車子撞牆了，撞牆次數加1
242            if(states['progress'] == state['progress'] and np.array(envState_['lidar']).min() < 0.4):
243                wall += 1
244            # 若車子沒有前進，有可能是環境尚未更新，也有可能是撞牆了，所以撞牆次數+ 0.5
245            elif(states['progress'] == state['progress']):
246                wall += 0.5
247            # 若車子向後退，有可能是在倒車調整方向，但大多都是走錯路，因此+ 0.8
248            elif(states['progress'] < state['progress']):
249                wall += 0.8
250            # 若車子向前走，撞牆次數歸零，避免下次一停下，就結束遊戲
251            elif(states['progress'] > state['progress'] + 0.0001):
252                wall = 0
253
254            step += 1    # 執行步數增加
255            total_reward += rewards    # 加上此step所得獎勵
256
257            envState = envState_.copy()  # 將new state放入原state
258            envState_convert = envState__convert.copy()  # 將new state放入原state
259            state = states.copy()        # 將new state放入原state
260
261
262
263            if wall > 600:       # 若撞牆次數累積600次，大略無法再前進，結束遊戲
264                end_episode = True
265
266            if done or end_episode:  # 若車子抵達終點、時間到，或自行設定的結束遊戲
267                break                # 跳出迴圈，結束此episode
268        output_video_writer.release()    # 釋放影片更改的權限
```