應用強化學習與生成網路於自動駕駛訓練

摘要

本研究探討如何利用深度強化學習技術來實現自動化賽車操控。以OpenAI Gymnasium 的 Car Racing 環境作為實驗平台，使用不同的強化學習演算法及訓練方式進行模型訓練比較，以期能找出最佳的演算法和訓練方式，讓賽車在複雜賽道上保持高速且穩定的行駛。

在實驗過程中，我們將自動駕駛的實現分為感知AI和決策AI兩部分，感知AI用於了解車輛所處的環境和本身狀態，這部分本研究實驗了圖像輸入、感知融合、配合生成式AI等不同訓練感知的方式；決策AI用於依據感知AI對環境的理解，判斷並做出序列決策，本研究使用DQN模型，及其衍生Double DQN和Dueling DQN實驗並比較其在自動駕駛表現的優劣。

本研究展示了深度強化學習在遊戲與自動駕駛領域的應用潛力，並為未來相關研究提供了參考基礎。

壹、前言

2025年美國消費性電子展(CES)上，自動駕駛和人形機器人為全場最受注目的焦點，電動化與智慧化已經成為未來汽車工業發展的主要方向，其中尤以智慧化中自動駕駛技術將改變整個汽車工業供應鏈，而自動駕駛的核心為人工智慧，隨著新聞不停傳播CES畫面，激起我們想要研究如何實現自動駕駛的技術。

強化學習reinforcement learning（RL）基本上就是通過不斷實驗和探索來，讓訓練的動作者(Agent)學會如何因應環境改變做出正確的決策，是目前在自動駕駛、機器人、無人機等領域實現自動操作的主流演化法。

強化學習主要由下列元素所組成：

一、智能體（Agent）：經過訓練能提供良好策略，依據觀察到的環境狀態(State)採取適合動作(Action)解決問題的主要角色。

二、環境（Environment）：能提供不同狀態感知給智能體，並依據智能體的動作給予獎勵(Reward)。

智能體依據當下(t時刻)觀察到的狀態(S_t)，採取動作(a_t)，獲得獎勵(R_t)，並使環境狀態轉移到(S_{t+1})，當下獎勵固然重要，強化學習是以完成最終任務的回報(G)為目標，理論回報可以用總獎勵來評估，如下式：

$$理論回報：G_t = R_t + R_{t+1} + R_{t+2} + \cdots = \sum_{k=0}^{\infty} R_{t+k}$$

然而在強化學習環境中由於不確定性，對於未來的獎勵通常是不準確的，所以折扣回報比理論回報更符合常理，每往後推一個時刻會多乘一個折扣因子。

$$折扣回報：G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$$

智能體和環境的不斷交互過程，可以透過馬可夫決策過程來建立數學模型，而最佳決策則可以透過貝爾曼方程來求解，求解過程可以透過估計狀態價值$v(s_t)$和動作價值$Q(s_t, a_t)$來形成策略：

一、狀態價值：進入狀態後，獲得回報的期望值。

$$v(s_t) = E[s_t] = E[R_t|s_t] + \gamma E[G_{t+1}|s_t] = R(s_t) + \gamma v(s_{t+1})$$

二、動作價值：進入狀態後，採取動作獲得回報的期望值。

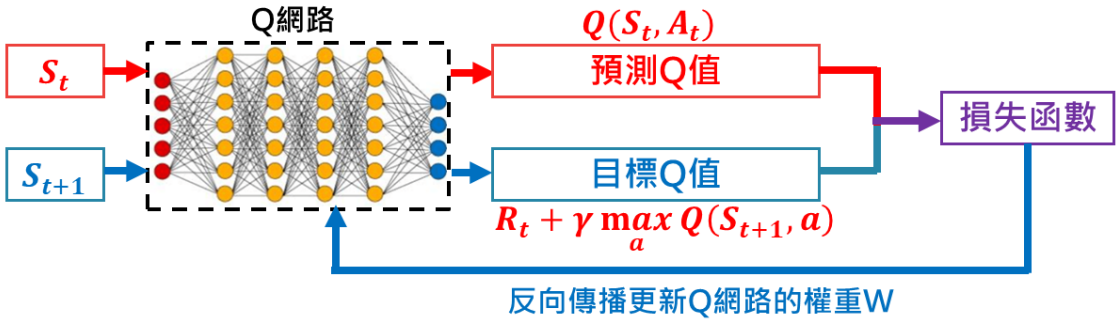
$$Q(s_t, a_t) = E[s_t, a_t] = E[s_t, a_t] + \gamma E[s_t, a_t] = R(s_t, a_t) + \gamma Q\left(s_{t+1}, a_{t+1}\right)$$

上面貝耳曼方程主要是提供由未來狀態的價值遞推目前狀態價值來求解最佳決策，未來狀態價值的估計若改用採取動作價值最大來取代期望值，即為Q-Learning：

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \max_a Q\left(s_{t+1}, a\right)$$

在環境狀態是離散且數量不多時，可以用矩陣(Q-Table)來儲存狀態動作價值，Agent從中選擇價值高的動作形成策略；但狀態如果是離散或數量非常多，Q-Table將變得太大且完全無法運行，解決方法是利用神經網路的方法來估計Q值，即為DQN算法(Deep Q-Network)。

DQN的原理如下圖：輸入s_t狀態經過Q網路得到s_t狀態採取動作a_t的價值預測值，另輸入下一狀態s_{t+1}經過Q網路得到s_{t+1}狀態採取動作的最高值$\max_a Q\left(s_{t+1}, a\right)$，加上獲得的獎勵$R(s_t, a_t)$作為目標值，將預測Q值和目標Q值以均方誤差計算損失函數，再透過梯度下降反向傳播更新Q網路的參數。



為了讓DQN的訓練更加穩定，還加入了兩個創新點：

一、經驗回放：將每一步交互緩存進(Replay Buffer)，累積到一定程度後可以每一次取出Batch Size個“經驗”進行批量學習，除了可以增加資料的使用效率，亦可(一)打亂資料順序，增加網路的泛化能力。

二、固定Q目標：預測Q網路和目標Q網路是同一網路時，隨著預測Q網路每次更新。則意味著目標Q網路每次均會變化，訓練較難穩定，故將預測Q網路和目標Q網路

分為兩個網路，**每隔一段時間用預測Q網路去更新目標Q網路，使目標Q網路維持一段時間穩定。**

DQN結合了深度學習神經網路和強化學習的優勢，能有效應對高維度、連續控制等複雜問題，現今已廣泛應用於各領域，本研究以OpenAI提供的Car_racing 強化學習環境來實作並研究使用DQN訓練自動駕駛的關鍵因素。

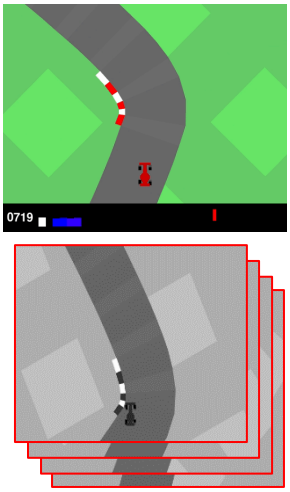
貳、研究設備及器材

名稱	功能	數量
Nvidia Jetson Orin 16GB	訓練模型使用的遠端平台及GPU	18
Python3.10	程式的主要撰寫語言	
Pytorch	搭建及訓練網路的框架	

參、研究過程或方法

一、實作DQN網路，解決Car_racing模擬環境：

(一)Car_racing學習環境會生成自上而下的 96x96 RGB 圖像，以捕獲汽車的位置和賽道配置，以描述狀態。此外圖像底部顯示包括車速、每個車輪的單個 ABS 感測器讀數、方向盤位置和陀螺儀數據。

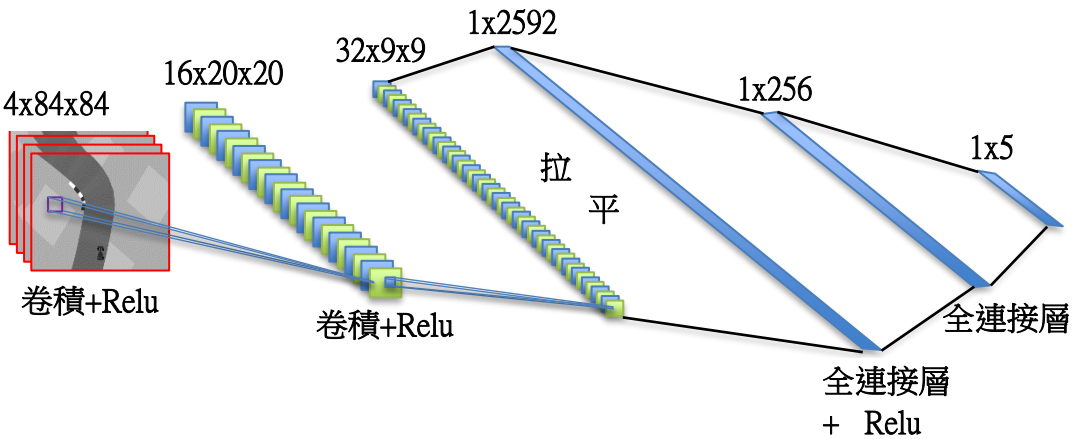


(二)圖片預處理：

(1)裁掉底部訊息條(12像素高)，裁掉左右各6像素，成為解析度 84x84，並轉換為灰階圖片 。

(2)為了使神經網路能夠正確跟蹤汽車運動狀態，每個狀態都由一系列 4 個 84x84 解析度的連續灰度幀重疊。上圖演示了一個狀態示例。通過在堆疊頂部添加新幀並丟棄最遠的幀來執行從一種狀態到另一種狀態的過渡。

(三)DQN神經網路架構如下圖：使用兩個卷積層和兩個全連接層。



(1)卷積層：第一層卷積使用16個8x8的卷積核，步長為4，第二層卷積使用32個4x4的卷積核，步長為2。

(2)全連接層：將卷積層輸出拉平後通過2592到256、256到5兩個全連接層，輸出每個動作的Q值。

```
#原來 DQN 的神經網路
class DQN(torch.nn.Module):
    def __init__(self,n_act):
        super(DQN,self).__init__()
        self.conv1 = torch.nn.Conv2d(4, 16, kernel_size=4, stride=4)    #[N,4,72,72]->[N,16,18,18]
        self.conv2 = torch.nn.Conv2d(16, 32, kernel_size=4, stride=2)    #[N,16,18,18]->[N,32,8,8]
        self.fc1 = torch.nn.Linear(32 * 8 * 8, 256)
        self.fc2 = torch.nn.Linear(256, n_act)
    def forward(self,x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view((-1, 32 * 8 * 8))
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

(四)實作DQN算法，訓練2500回合，探索策略為前1250次， ε 由1衰減到0(如下段內容所述)，觀察訓練情況，並測試模型表現。

二、探索(exploration)與利用exploitation的難題：

(一)探索：讓Agent嘗試一些新的動作，可能獲得更多的回報，避免落入局部最優，但也可能表現變差。

(二)利用：指採取已知較優動作，重複執行這個動作，因為我們知道這樣做可以獲得一定的獎勵。

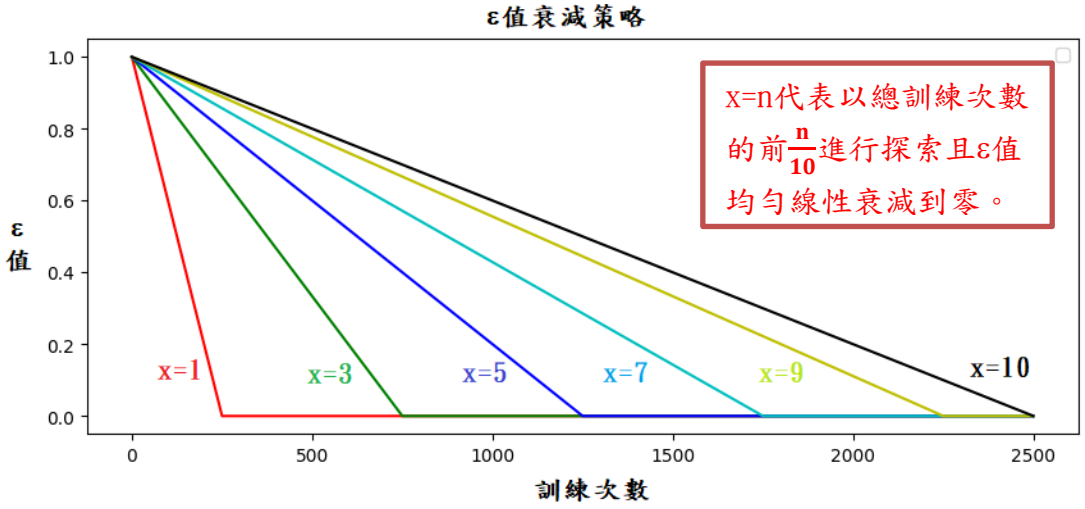
(三)ε-greedy算法：設定一個參數 ε ，選擇一個動作時，先隨機選擇一個0到1之間的數，如果這個數小於等於 ε ，則進行探索；否則利用。

(四)ε值衰減策略：

(1)智能體開始訓練時能力較弱，必須有一個較大的ε值，可以進行較多隨機動作的探索。

(2)而訓練的越久，智能體也會隨之增強，必須有一個較小的ε值，進行較多的最優策略的利用。

(五) 我們以衰減係數x為1、3、5、7、9、10的線性衰減策略訓練模型，並討論分析其訓練結果，期望找出最適合此環境的衰減策略以進行後續的研究。

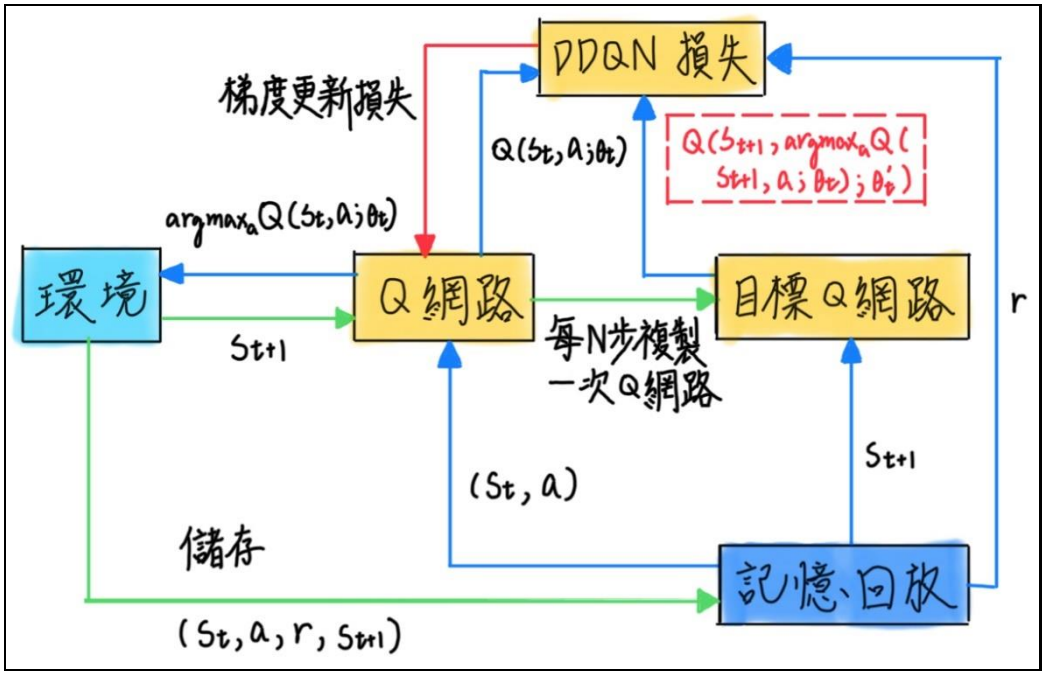


```
#線性衰減策略
self.EPS=max(  1-i*(1-self.eps_low)/(x*N_EPISODES/10)  ,  self.eps_low)
```

三、其他DQN演算法：Double DQN和DuelingDQN比較

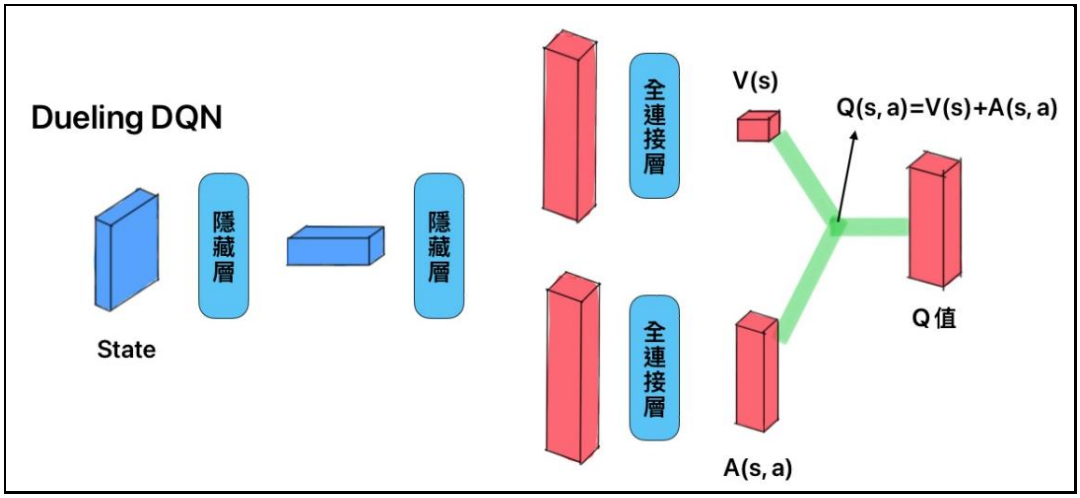
(一) Double DQN (Double Deep Q-Network,DDQN)：

因為DQN使用下一個狀態動作價值最高的來更新現在狀態價值，所以容易產生Q值高估，為解決 DQN 可能產生過度估計 Q 值的問題，所以其使用兩個網路分開進行動作選擇與 Q 值計算，預測網路用以選擇最佳動作，而目標網路則用來提供較穩定的價值評估，此舉可以減少 Q 值的過度估計、提高訓練穩定性、加快收斂速度及提升泛化能力，但可能也因此造成樣本使用效率不高、探索策略依賴 ε -greedy等問題，使其較不適合處理連續控制問題。其具體改進方法如下：



(二)Dueling DQN（Dueling Deep Q-Network）

為提高對決策的判斷能力，Dueling DQN將Q值拆分為兩個部分，第一個是狀態價值函數V(s)，用來評估該狀態本身的價值，與行動無關，第二個則是行動優勢函數A(s,a)，用以評估在該狀態下，選擇某一行動相較於其他行動的好壞程度。可以提升學習效率、提升泛化能力，使其能更好的適應稀疏獎勵的環境，但同時也因為增加了網路分支，所以導致其計算成本較高。其與一般DQN的示意圖如下：



讓Agent嘗試一些新的動作，可能獲得更多的回報，避免落入局部最優，但也可能表現變差。

(二)利用：指採取已知較優動作，重複執行這個動作，因為我們知道這樣做可以獲得一定的獎勵。

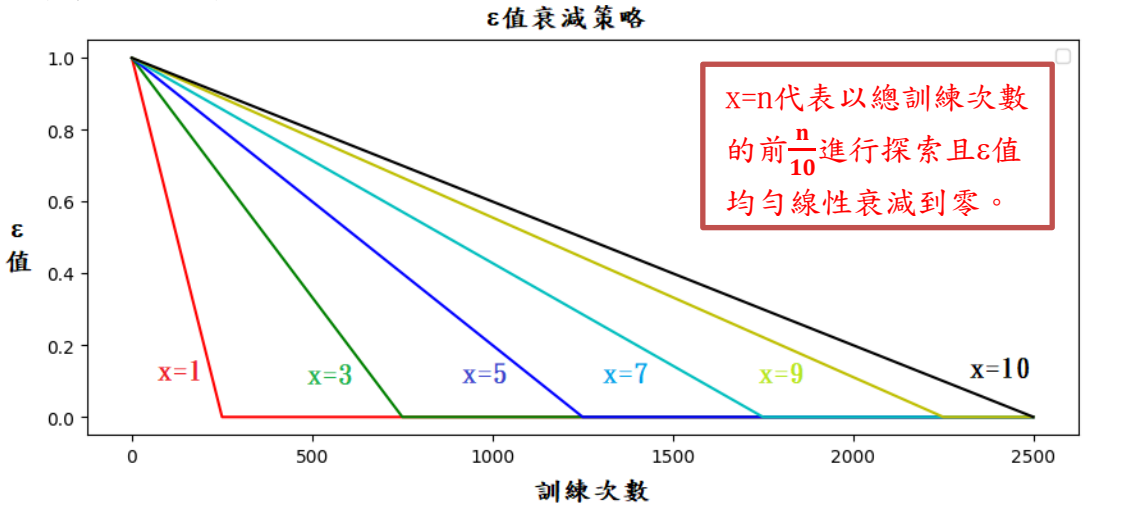
(三)ε-greedy算法：設定一個參數 ε ，選擇一個動作時，先隨機選擇一個0到1之間的數，如果這個數小於等於 ε ，則進行探索；否則利用。

(四)ε值衰減策略：

(1)智能體開始訓練時能力較弱，必須有一個較大的ε值，可以進行較多隨機動作的探索。

(2)而訓練的越久，智能體也會隨之增強，必須有一個較小的ε值，進行較多的最優策略的利用。

(五)我們將以線性衰減策略，探討不同衰減係數x，對於DQN演算法的影響，找出較佳的衰減策略。




```
#線性衰減策略
self.EPS=max(  1-i*(1-self.eps_low)/(x*N_EPISODES/10)  ,  self.eps_low)
```

四、改寫環境的獎勵機制

原獎勵機制為任務導向，依照駛過賽道畫面獲得正向獎勵，車子只要在賽道內，不論是在道路中央或邊緣， 智能體都會認為其行動適當，並沒有對車輛操控提供獎勵，我們想要從車輛操控的優劣來提供獎勵，因為好的操控才能讓車輛行駛在車道上，以鼓勵車輛盡量開在賽道中央。

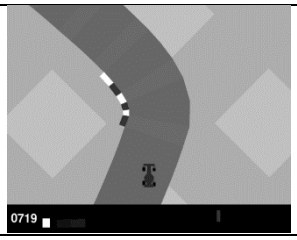
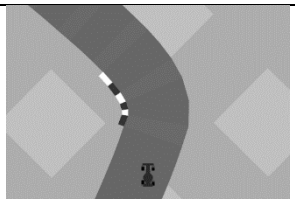
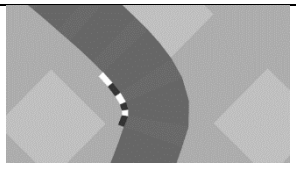
修改獎勵機制，使車子盡量在賽道中央，safe 為車子與賽道中點的歸一化距離，計算賽道區段頂點座標的平均值作為賽道中點，並用歐幾里得公式計算車子與中點距離且規一化，若離中點距離小於安全距離則加分，反之則扣分，其中車輛愈靠近賽道中央加愈多分，愈靠近邊緣扣愈多分。此外，為了讓Agent能做出更好的判斷，將分數同乘倍數，將分數差距拉大，鼓勵車子盡量靠近賽道中心。

```
#  計算額外獎勵分數(越靠近賽道中央，額外獎勵分數越高)
safe  =  0.7          #  安全距離
car_pos  =  np.array(self.car.hull.position)    #  取得車子座標
center  =  None      #  紀錄車子與賽道中央距離

#  遍歷所有賽道區段
for  poly,  _  in  self.road_poly:
        if  is_in_road(car_pos,  poly):
                #  取得賽道中央座標
                poly_array  =  np.array(poly)
                center_point  =  poly_array.mean(axis=0)
                #  計算車輛與賽道中點距離
                center  =  np.linalg.norm(car_pos  -  center_point)
                break
if  center  is  None:
        center  =  0.0

track_half_width  =  TRACK_WIDTH    #  取得  賽道一半寬度
normalized  =  center  /  track_half_width    #  規一化
multiple  =  100.0        #  獎勵倍數(讓分數差別更明顯，Agent更好做判斷)

#  將車子離賽道中央距離轉換成分數
if  normalized  <=  safe:    #  當車子離賽道中央距離小於等於安全距離
        score  =  (safe  -  normalized)  *  multiple    #  加分
else:      #  當車子離賽道中央距離大於安全距離
        score  =  -(normalized  -  safe)  *  multiple  #  扣分
```

五、不同觀測空間的訓練與感知融合

(一)由於原環境所提供的空照圖像畫面包含了車道和車身本身的圖像訊息，我們認為這與現實中的自動駕駛只能取得車前影像有所不符，於是我們將資訊框及車頭一併裁掉進行訓練，透過兩者的訓練及測試結果，了解圖像改變對智能體決策的影響。

其具體的畫面差異如下：

原圖 96x96	裁掉資訊框 84x84	裁掉車頭之後 72x72
		

(二)真實自動駕駛車輛本身也會有許多感測器，我們研究OpenAI Car_Racing.py原始碼，我們改寫 car racing環境，先在 step() 中利用歐幾里得距離公式計算車子真

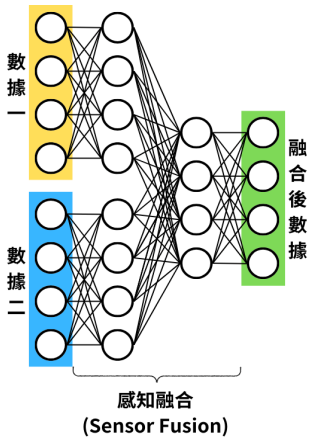
實速度，並取得輪子角速度、轉向角度，以及車子自身角速度，用來模擬車子的ABS感測器、方向盤轉向角度及陀螺儀數值，並將資料回傳 info，並在經驗回放緩衝區(Replay Buffer)中新增儲存info的區域。

```
#  計算車子速度、ABS感測器、方向盤位置、陀螺儀
true_speed  =  np.sqrt(
        #  car.hull.linearVelocity  為車子線性速度，index[0]為x座標，index[1]為y座標
        self.car.hull.linearVelocity[0]**2  +  self.car.hull.linearVelocity[1]**2
)
#  omega  為輪子的角速度
four_ABS_sensors  =  [wheel.omega  for  wheel  in  self.car.wheels]
#  angle  為方向盤及輪子轉向角度
steering_wheel_position  =  self.car.wheels[0].joint.angle
#  car.hull.angularVelocity  為車子相對自身中心的角速度
gyroscop  =  self.car.hull.angularVelocity

#  將數值加入info
self.info  =  {
        "true_speed":  true_speed,
        "four_ABS_sensors":  four_ABS_sensors,
        "steering_wheel_position":  steering_wheel_position,
        "gyroscop":  gyroscop,
}
```

(三)感知融合(Sensor Fusion)常用於自動駕駛系統中，主要功用為利用不同類型的感測器所蒐集到的資訊，找出資料關連並進行運算處理，以用於預測未來的狀態。

即使單一感測器接收到的測量值不可靠，或是面對不同環境條件，感知融合演算法都能處理所有輸入並產生高精度、高可靠性的輸出，保持穩定和高效的運行。

由於要把車前圖像和車子運動資訊合併，我們使用感知融合技術，具體搭建網路架構如下：

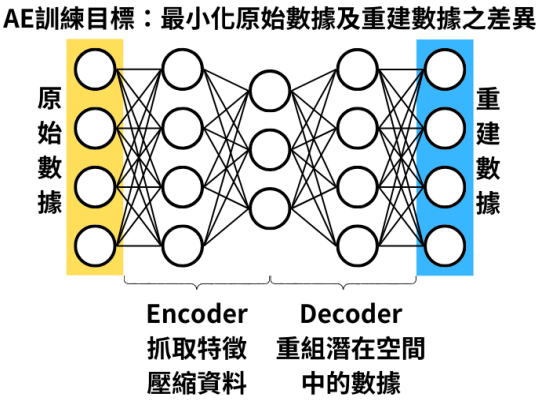



```
class  DQN(torch.nn.Module):
    def  __init__(self,n_act):
        super(DQN,self).__init__()
        self.conv1  =  torch.nn.Conv2d(4,  16,  kernel_size=8,  stride=4)    #[N,4,72,72]->[N,16,17,17]
        self.conv2  =  torch.nn.Conv2d(16,  32,  kernel_size=4,  stride=2)    #[N,16,17,17]->[N,32,6,6]
        self.vision_fc1  =  torch.nn.Linear(1568,  256)  #  圖片的感知
        self.vision_fc2  =  torch.nn.Linear(256,  16)
        self.sensor_fc1  =  torch.nn.Linear(7,  16)  #  info的感知
        self.fusion  =  torch.nn.Linear(32,  n_act)  #  將圖片與  info  融合
    def  forward(self,x1,x2):
        x1  =  F.relu(self.conv1(x1))
        x1  =  F.relu(self.conv2(x1))
        x1  =  x1.view((-1,  1568))
        x1  =  F.relu(self.vision_fc1(x1))
        x1  =  F.relu(self.vision_fc2(x1))
        x2  =  F.relu(self.sensor_fc1(x2))
        x2  =  x2.view((-1,  16))
        x=torch.cat((x1,  x2),  dim=1)
        x  =  self.fusion(x)
        return  x
```

六、使用生成式AI協助強化學習環境感知的訓練

自動駕駛技術通常會先訓練模型環境判斷的能力，再訓練遇到各環境需進行的動作反應，因此我們在選擇動作的DQN前加入自動編碼器(Auto Encoder, AE)與變分自編碼器(Variational Autoencoder, VAE)以實現這點。

(一)自動編碼器(Auto Encoder, AE)是一種無監督式學習，能將高維數據壓縮到低維潛在空間，並盡量完整重建原始輸入，常被用於降噪、特徵抽取及資料壓縮等等。因此我們想將其融入DQN，期望智能體能用更短的時間以及更高的效率做出判斷，減少車子因判斷時間誤差而造成的意外。自編碼器由兩個主要組件組成：用於抓取訓練資料的重點特徵、去除雜訊與降維的編碼器(Encoder)，解碼器(Decoder)則將壓縮至低維度的數據還原回原始數據，最終訓練目標是最小化原始資料與解碼器重建的數據之間的差異。



(1)編碼器與解碼器皆由四層全連接層搭建，並且為對稱性結構。

```
class  VAEEncoder(nn.Module):
        def  __init__(self):
                super().__init__()
                self.fc1  =  nn.Linear(84*84,  512)
                self.fc2  =  nn.Linear(512,  128)
                self.fc3  =  nn.Linear(128,  32)
                self.fc4  =  nn.Linear(32,  16)

        def  forward(self,  x):
                x  =  self.fc4(torch.relu(self.fc3(torch.relu(self.fc2(  torch.relu(  self.fc1(x)  )  )))))
                return  x
```

```
class  VAEDecoder(nn.Module):
       def  __init__(self):
               super().__init__()
               self.fc1  =  nn.Linear(16,  32)
               self.fc2  =  nn.Linear(32,  128)
               self.fc3  =  nn.Linear(128,  512)
               self.fc4  =  nn.Linear(512,84*84)
       def  forward(self,  x):
               x  =  torch.tanh(self.fc4(torch.relu(self.fc3(torch.relu(self.fc2(  torch.relu(  self.fc1(x)  )  ))))))*0.5
               return  x
```

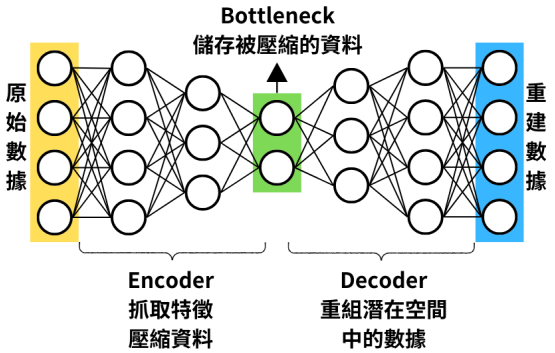
(2)將兩者組合，就形成完整的AE結構。

```
class  VAE(nn.Module):
        def  __init__(self):
                super().__init__()
                self.encoder  =  VAEEncoder()
                self.decoder  =  VAEDecoder()

        def  forward(self,  x):
                x  =  self.encoder(x)
                x  =  self.decoder(x)
                return  x
```

(3)為了最小化輸入資料及重構資料間的差異，在損失函數中使用均方誤差。

```
def VAE_Loss(reconstruct, x):
    reconstruct_loss = nn.MSELoss()(reconstruct, x)
    return reconstruct_loss
```

(二)變分自編碼器(Variational Autoencoder, VAE) 以AE為基礎的生成式模型，通常用於壓縮資料、去除圖像雜訊、異常檢測及人臉辨識，並且能學習數據的潛在分布及生成新的數據樣本。VAE主要由三項要件組成：與AE相似的編碼器(Encoder)、解碼器(Decoder)，和具有潛在空間與被壓縮、降維的輸入資料的瓶頸(bottleneck)。模型主要訓練目的是學習如何從特定輸入中抓取特徵並將其放入潛在空間中，並縮小原始資料與解碼器輸出資料之差異。



VAE訓練目標：最小化原始數據及重建數據之差異

(1)編碼器由兩層全連接層(fc1、fc2)組成，將輸入資料壓縮成隱變數(mean、log_var)。目的是將原始輸入轉換成潛在空間(latent space)，並學習其均值和對數變異數。

```
class VAEEncoder(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, latent_size*2)
        self.latent_size = latent_size

    def forward(self, x):
        x = self.fc2( torch.relu(self.fc1(x)) )
        mean, log_var = x.split(self.latent_size, dim=1)
        return mean, log_var
```

(2)解碼器由兩層全連接層組成，反向傳播使用 sigmoid 激活函數將輸出壓縮到 [0, 1]。目的是將潛在空間中抽樣得到的隱變數盡量還原成原始輸入。

```
class VAEDecoder(nn.Module):
    def __init__(self, latent_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(latent_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.sigmoid( self.fc2( torch.relu(self.fc1(x)) ) )
        return x
```

(3)將編碼器和解碼器組合成VAE模型，並在反向傳播處利用重參數化使模型變得可微分，以讓模型能夠有效收斂。

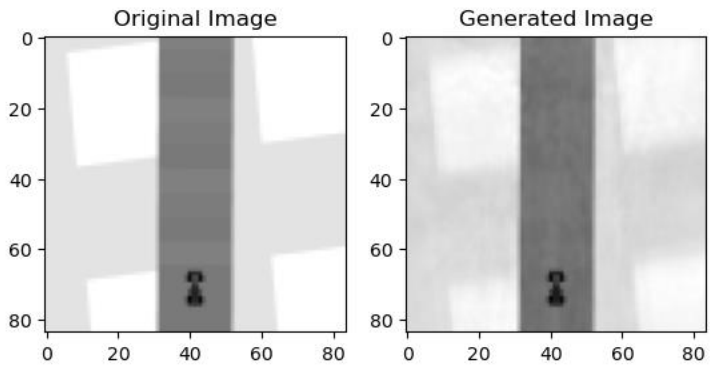
```python
class VAE(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super().__init__()
        self.encoder = VAEEncoder(input_size, hidden_size, latent_size)
        self.decoder = VAEDecoder(latent_size, hidden_size, input_size)

    def forward(self, x):
        mean, log_var = self.encoder(x)          # 均值、對數變異數
        std = torch.exp(0.5 * log_var)           # 標準差
        eps = torch.randn_like(std).to(device)   # 隨機降噪
        z = mean + std * eps                     # 隨機抽樣
        reconstruct = self.decoder(z)            # 重建影像
        return reconstruct, mean, log_var
```

(4)用均方誤差計算重建誤差，套用KL散度公式計算潛在分布與標準正態分布差異，以確保 VAE 產生有意義的隱變數，最後將兩者相加作為VAE模型最後的損失函數。

```python
def VAE_Loss(reconstruct, x, mean, log_var):
    # 重建損失
    reconstruct_loss = nn.MSELoss(reduction='mean')(reconstruct, x)
    # KL散度損失
    kl_loss = -0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp())
    # VAE 損失 = 重建損失 + KL散度損失
    return reconstruct_loss + kl_loss
```

(5)VAE生成圖片效果

輸入大小：
(input_size) = 84*84
隱藏層大小：
(hidden_size) = 800
淺在輸出大小：
(latent_size) = 64

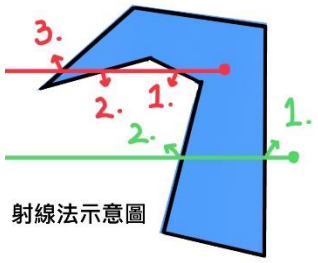


七、調整Replay Buffer採樣機制

(一)改寫環境以更貼近現實生活

在現實生活中，車子必須開在賽道內，不能超出賽道。因此我們將程式碼修改為遊戲中的車輛跑出賽道時，立即終止遊戲，讓遊戲更貼近真實駕駛環境。

定義全域變數 is_in_road() 並將賽道當成一封閉多邊形，利用射線法判斷車子否在賽道內。射線法能夠判斷一點是否在多邊形內部，方法是從該點做射線，並計算它和多邊形邊界交點數，若交點數為奇數，則該點位於多邊形內，反之在多邊形外。



```python
# 利用射線法判斷車子是否在賽道內
def is_in_road(point, race_road):
    n = len(race_road)              # 計算賽道有幾個頂點
    inside = False                  # 紀錄車子是否在賽道內(初始值為否)
    road_x1, road_y1 = race_road[0] # 紀錄頂點1

    # 遍歷賽道頂點
    for i in range(n + 1):
        road_x2, road_y2 = race_road[i % n] # 紀錄頂點2

        # 檢查車子是否超出兩頂點所圍的線段(賽道)
        if point[1] > min(road_y1, road_y2):
            if point[1] <= max(road_y1, road_y2):
                if point[0] <= max(road_x1, road_x2):
                    if road_y1 != road_y2:
                        xinters = (point[1] - road_y1) * (road_x2 - road_x1)
                        / (road_y2 - road_y1) + road_x1
                    if road_x1 == road_x2 or point[0] <= xinters:
                        inside = not inside    # 車輛在賽道內
        road_x1, road_y1 = road_x2, road_y2 # 將頂點1更新為頂點2
    return inside
```

在 step() 中利用自訂義 is_in_road() 函式判斷車子是否超出賽道，若超出賽道就終止遊戲。

```python
# 取得車子座標
car_position = (self.car.hull.position[0], self.car.hull.position[1])
on_road = False      # 紀錄車子是否在賽道內(初始值為否)

# 遍歷賽道頂點座標
for poly, _ in self.road_poly:
        # 判斷車子是否於賽道內
        if is_in_road(car_position, poly):
                on_road = True
                break

# 車子超出賽道就重置遊戲
if not on_road:
        self.reset()
        return self.state, step_reward, False, True, {}
```

(二)嘗試改善因改寫環境所產生的問題

為模擬真實駕駛環境，車輛超出賽道立即終止遊戲。然而，其模型分數上升到一定程度後，表現便無法再突破。推測原因為車輛只要碰到邊界就立即結束，使其較難訓練到遊戲後期的環境，而是重複學習前期場景，導致智能體在後段賽道表現不佳。為解決此問題，我們修改了 Replay Buffer 的採樣方式，從均勻採樣調整為依車輛行駛步數進行機率採樣，使高步數的經驗有較大機率被訓練，讓智能體能更熟悉後期環境，進而突破分數停滯的瓶頸。

(1)為讓智能體能優先訓練後期環境，我們以車輛每回合行駛的步數多寡，作為回放緩衝區的採樣機率。先於 DQNAgent()中，記錄每回合車輛行駛步數，並存入回放緩衝區。

```
def Train(self, N_EPISODES):
    for i in tqdm(range(Load_File, N_EPISODES)):
        self.EPS = max(1-(i*(1-self.eps_low)/(1*N_EPISODES/10)), self.eps_low)
        total_reward = 0
        step = 0                        # 紀錄回合步數
        s, _ = self.env.reset()
        while True:
            a = self.SelectA(self.PredictA(s))
            s_, r, done, stop, _ = self.env.step(a)
            total_reward += r
            step += 1                   # 每走一步，回合步數+1
            self.rb.append(s, a, r, s_, done, step) # 將資訊加入 Replay Buffer
            if self.rb.size > 200 and i%self.rb.num_steps==0: self.Learn()
            if i % 20==0: self.TargetDQN.load_state_dict(self.PredictDQN.state_dict())
            s = s_
            if done or stop: break
        Log["TrainReward"].append(total_reward)
        if i % 10 == 9:
            torch.save(self.PredictDQN.state_dict(), f"Model-{i+1}.pt")
        if i % 50 == 49:
            test_reward = self.Test()
            print(f"\n訓練次數{i+1}，總回報{test_reward}")
            Log["TestReward"].append(test_reward)
            np.save(f"Log-{i+1}.npy", Log)
        if i % 100 == 99:
            display.clear_output(wait=True)
```

(2)在回放緩衝區中，將步數轉換為機率分布，再根據其機率分布進行抽樣。

```
class ReplayBuffer:
    def __init__(self,max_size=int(1e5), num_steps=1):
        self.s = np.zeros((max_size,4,84,84), dtype=np.float32)
        self.a = np.zeros((max_size,), dtype=np.int64)
        self.r = np.zeros((max_size, 1), dtype=np.float32)
        self.tr = np.zeros((max_size, ), dtype=np.float32)
        self.s_ = np.zeros((max_size,4,84,84), dtype=np.float32)
        self.done = np.zeros((max_size, 1), dtype=np.float32)
        self.ptr = 0
        self.size = 0
        self.max_size = max_size
        self.num_steps = num_steps

    def append(self,s,a,r,s_,done,tr):
        self.s[self.ptr] = s
        self.a[self.ptr] = a
        self.r[self.ptr] = r
        self.tr[self.ptr] = tr    # 儲存每回合所走步數
        self.s_[self.ptr] = s_
        self.done[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size+1,self.max_size)

    # 依車輛行駛步數多寡，進行樣本抽樣(步數愈高，愈容易被採樣)
    def sample(self, batch_size):
        # 依該回合所走步數，計算該筆資料被抽樣的機率分布
        probs = self.tr[:self.size] / self.tr[:self.size].sum()
        # 根據機率分布進行抽樣
        ind = np.random.choice(np.arange(self.size), size=batch_size, p=probs)
        return torch.FloatTensor(self.s[ind]), \
               torch.LongTensor(self.a[ind]), \
               torch.FloatTensor(self.r[ind]), \
               torch.FloatTensor(self.s_[ind]), \
               torch.FloatTensor(self.done[ind])
```

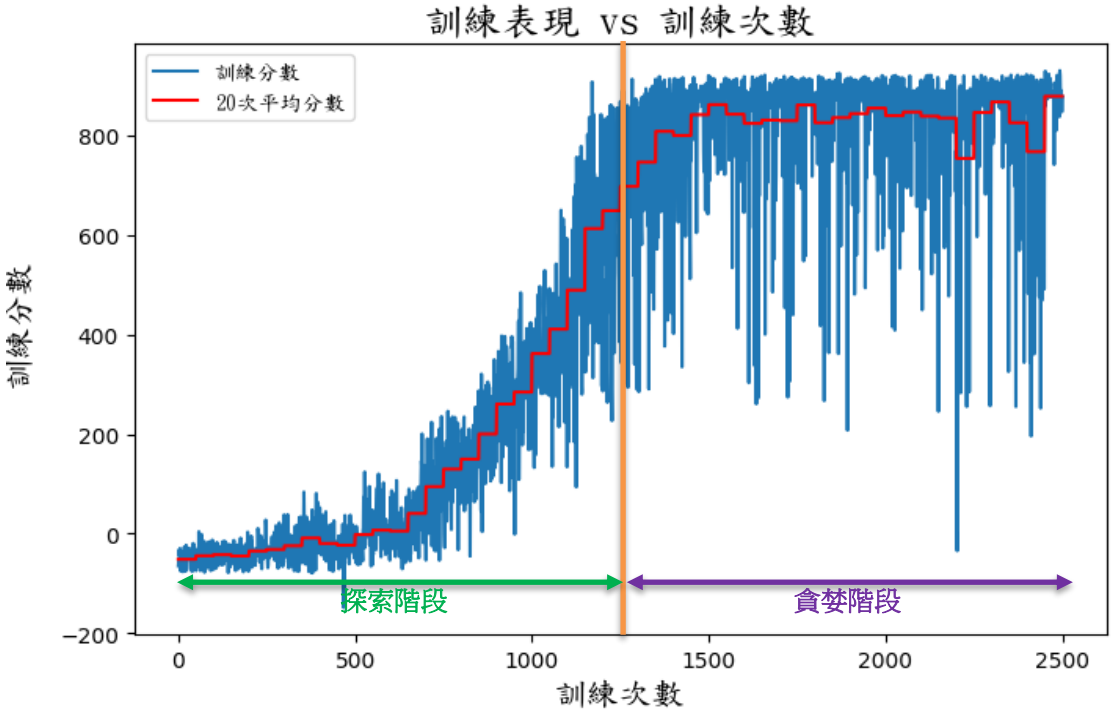
肆、研究結果

一、實作DQN網路，解決Car_racing模擬環境

(一) 訓練情況：

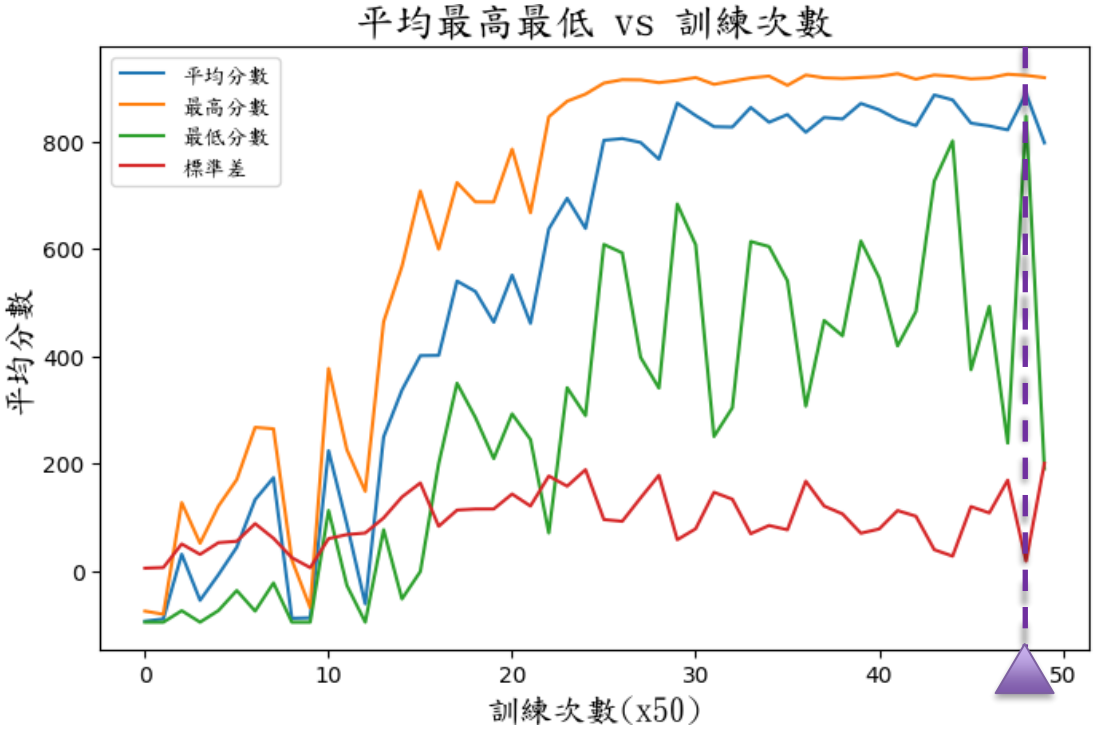
(1)如下圖 ，藍色線條部分為每一回合訓練的分數，紅色部分為每20回合訓練的平均表現。

(2)在探索階段，分數逐漸增加；進入貪婪階段後，算法逐漸收斂，分數漸趨於上限約900分，平均800分。



(二) 評估模型情況：

(1)如下圖 ，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(2)平均分數在訓練1250後達到穩定，在訓練2450次的模型，圖中三角型標注處，得到平均分數最高，且標準差最小，應該是最優模型。

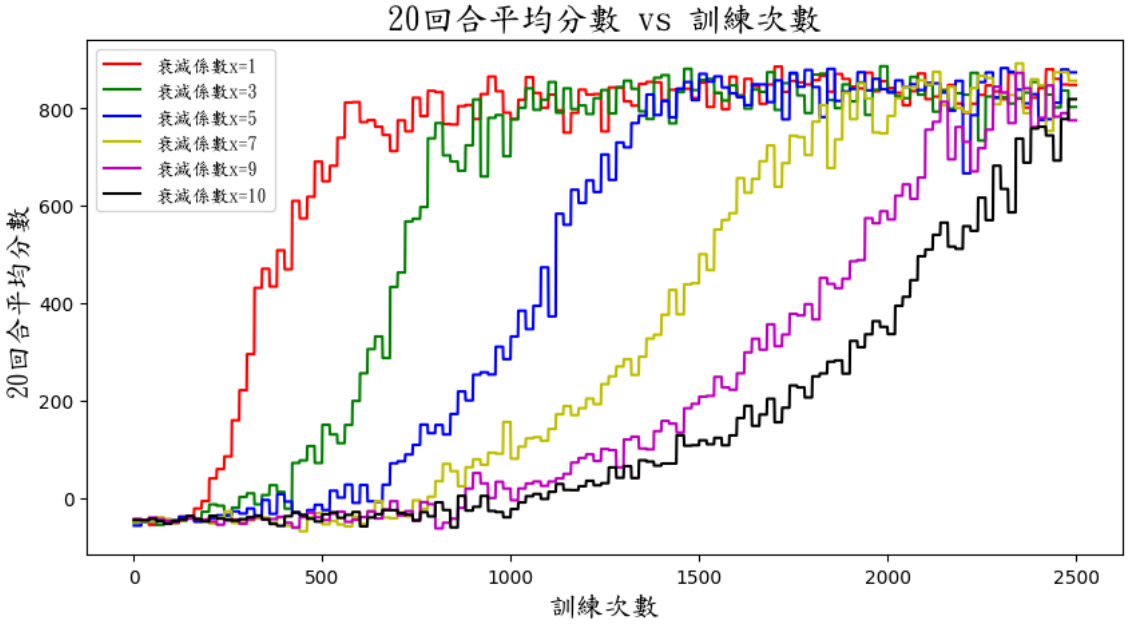
二、探索(exploration)與利用exploitation的難題：

(一) 模型訓練情況比較：

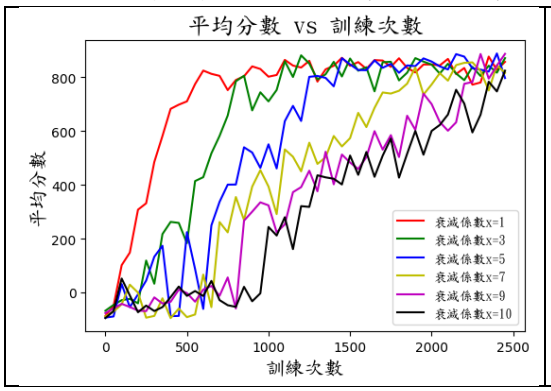
(1)如下圖 ，不論是何種衰減係數，其平均分數值皆有上升，且其最大值皆能高於800分，顯現其皆能有效的訓練，。

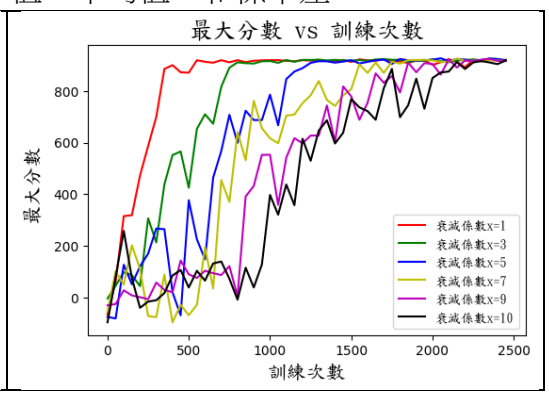
(2)其中衰減係數越小，其學習速度越快(即平均分數較快上升)，在貪婪階段有較穩定的表現。

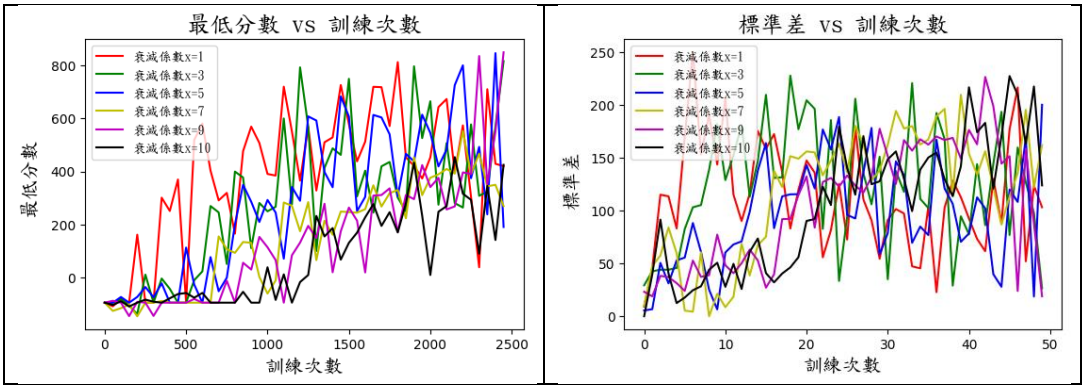
(3)而衰減係數較大的訓練模型，因為較長時間進行探索，算法收斂比較慢，最後分數表現較不佳。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。







(1)衰減係數1、3、5屬於算法收斂較快：平均分數和最高分數比較快達到穩定，但其後期模型測試的平均獎勵有微幅下降。

(2)衰減係數7、9、10屬於算法收斂較慢：在訓練初期大量探索，平均分數未見上升。

(3)取訓練2000次後的評估來做比較：

衰減係數	x=1	x=3	x=5	x=7	x=9	x=10
平均分數	832	829	846	816	754	695
最高分	913	915	920	916	908	894
最低分	464	458	511	394	461	259
標準差	110	113	97	138	135	180

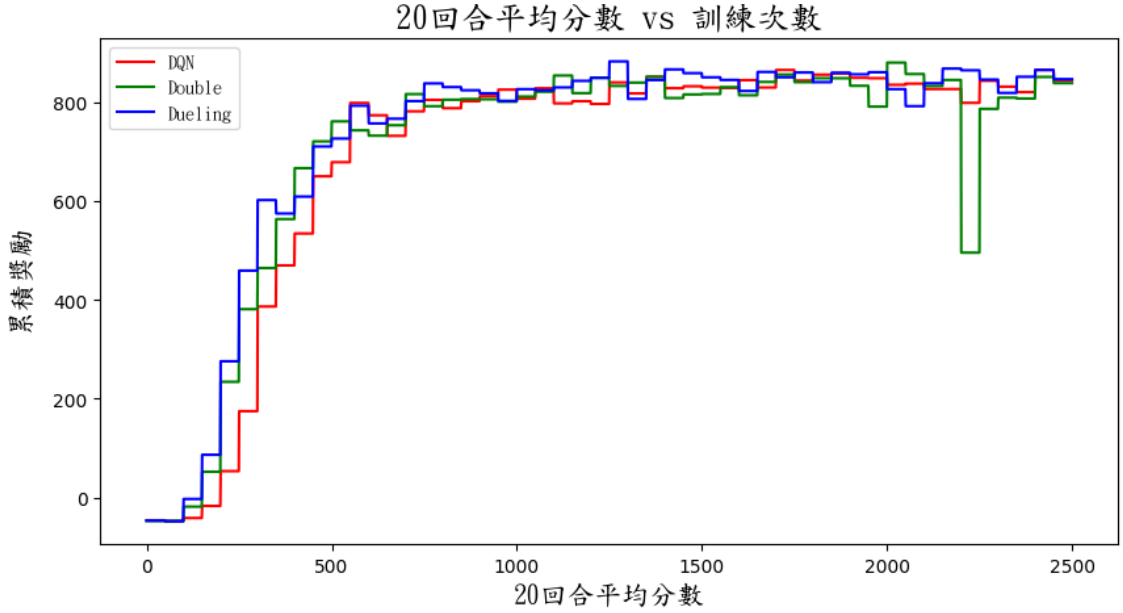
由上表統計可以看出，衰減係數x=5在所有項目的評估中表現都是最優的，其探索策略為前一半訓練進行探索，後一半訓練進行貪婪收斂算法，應該是這各策略能有足夠的探索和收斂。

二、其他DQN演算法：Double DQN和DuelingDQN比較

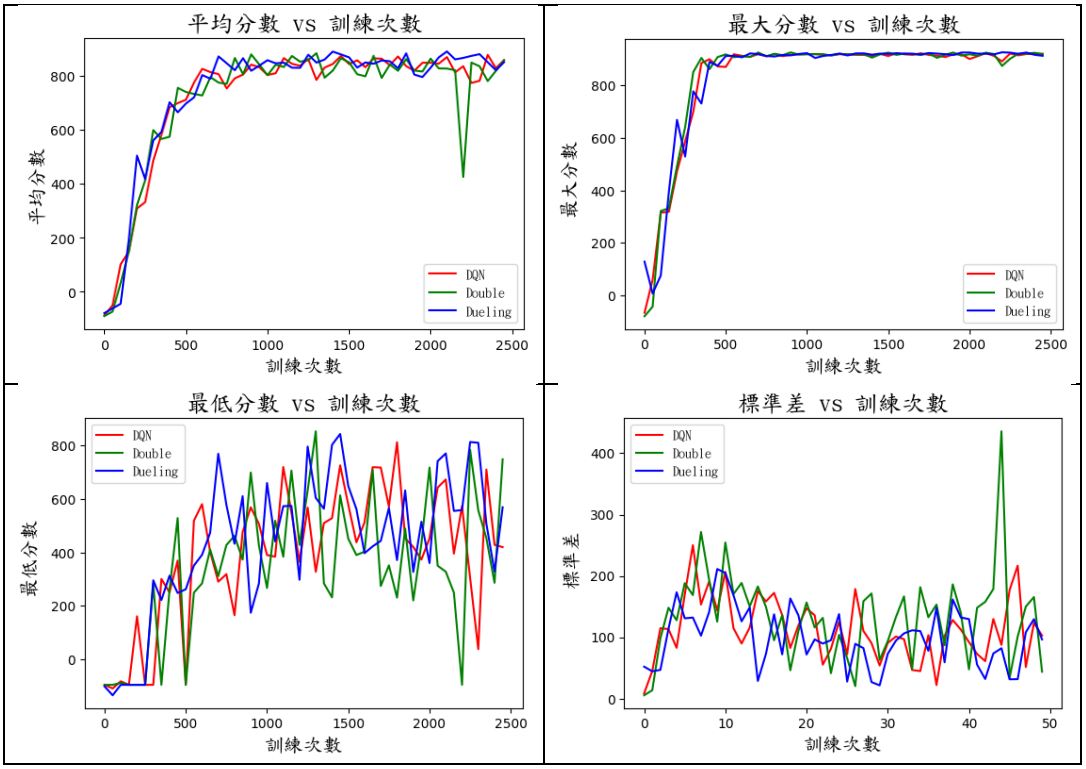
(一) 模型訓練情況比較：我們使用三種演算法分別訓練2500回合後，每50回合取一次平均，繪製成平均分數與訓練次數關係圖如下：

(1)在探索階段：Dueling DQN的平均分數上升最快，DQN則是最慢。

(2)在貪婪階段：Double DQN平均分數有突然下降，顯示其不穩定，其他趨勢則大約相同。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(1) DQN和Dueling DQN的整體表現差異不大，其平均分數及最高分數皆有上升的趨勢，其中Dueling DQN的平均獎勵又略高於DQN，兩者的最低分數也是為上升的趨勢。

(2)其中DQN的最小值略低於Dueling DQN且在後期模型的測試中有出現一較低的最小值，表示其較不穩定，此點也可從其測試結果的標準差略高於Dueling DQN得知，而Double DQN則是在後期模型的測試表現上明顯不佳，有較低的平均獎勵、最低的最小值及較大的標準差。

(3)取訓練2000次後的評估來做比較：

算法	DQN	Double DQN	Dueling DQN
平均分數	832	789	858
最高分	913	913	920
最低分	464	438	602
標準差	110	146	77

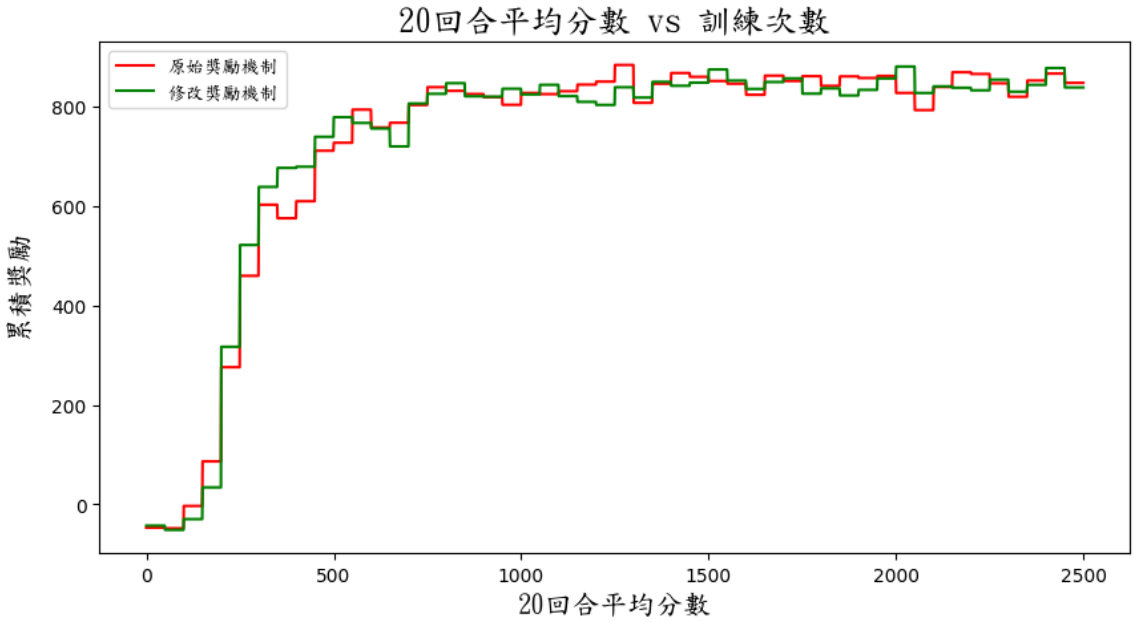
由上表統計可以看出，Dueling DQN在所有項目的評估中表現都是最優的。

四、改寫環境的獎勵機制

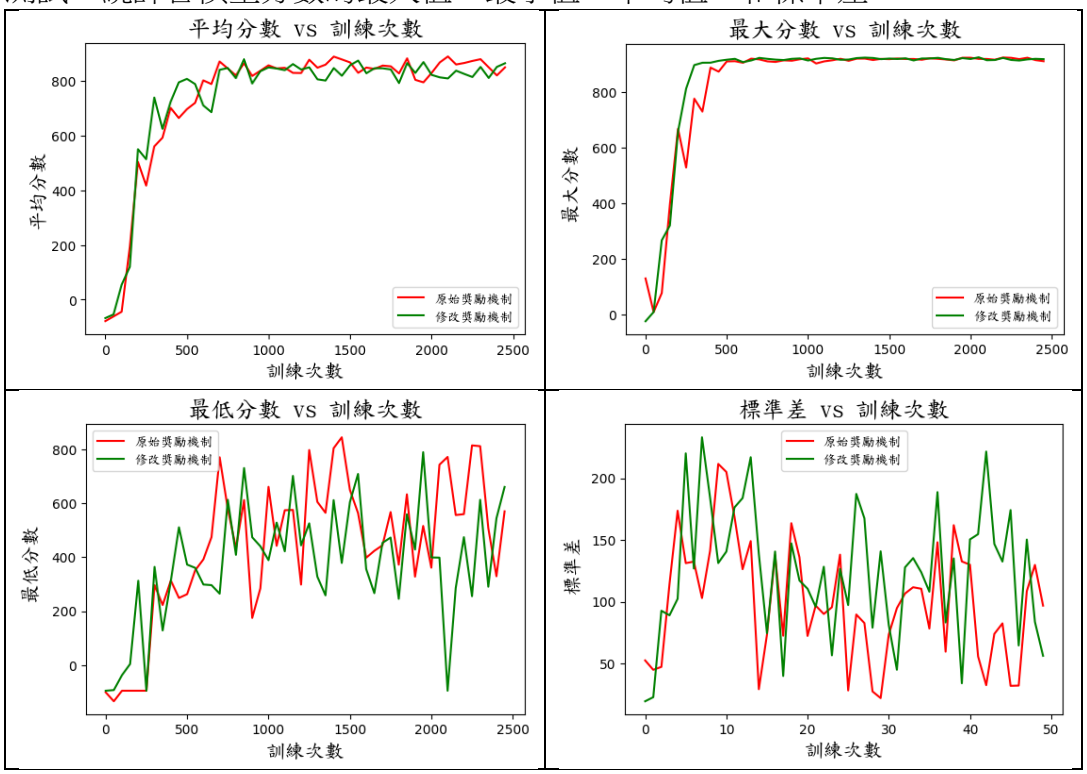
(一) 模型訓練情況比較：我們使用兩種獎勵機制分別訓練2500回合後，每50回合取一次平均，繪製成平均分數與訓練次數關係圖如下：

(1)在探索階段：有修改獎勵機制的平均分數比原始獎勵機制上升較快。

(2)在貪婪階段：兩種獎勵機制相差不大。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(1)修改獎勵機制在探索期間的測試表現平均分數和最高分數都比較高，可見在探索期間，用車道居中來規範車輛的操控，可以加快學習速度。

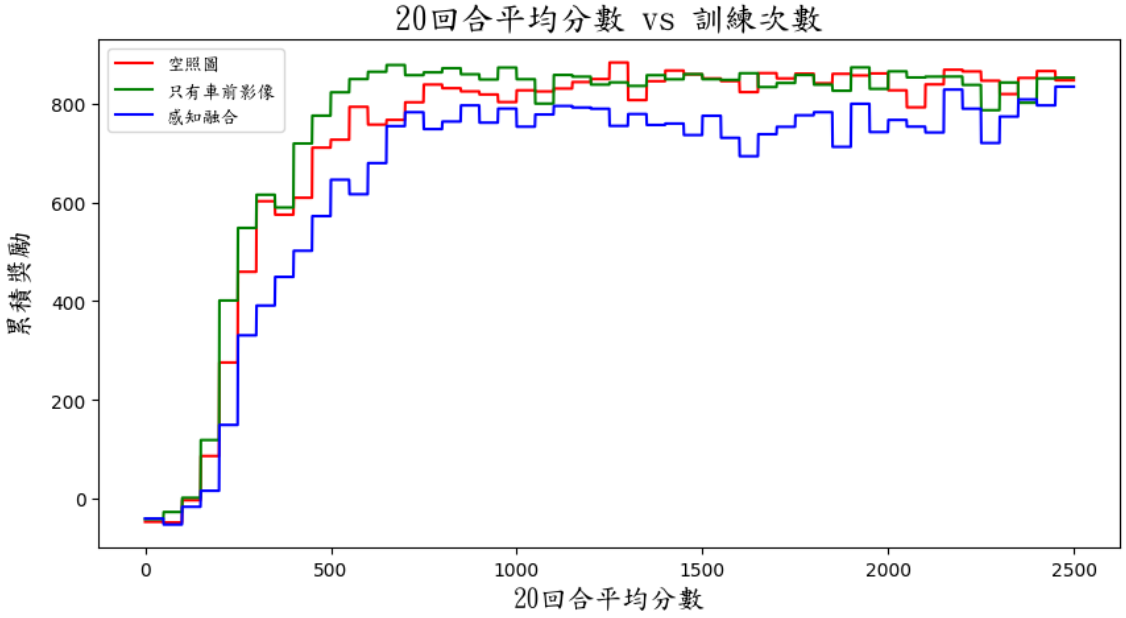
(2)而在貪婪階段修改獎勵機制的表現平均分數和最高分相差不大，但表現波動性較大。

五、不同觀測空間的訓練與感知融合

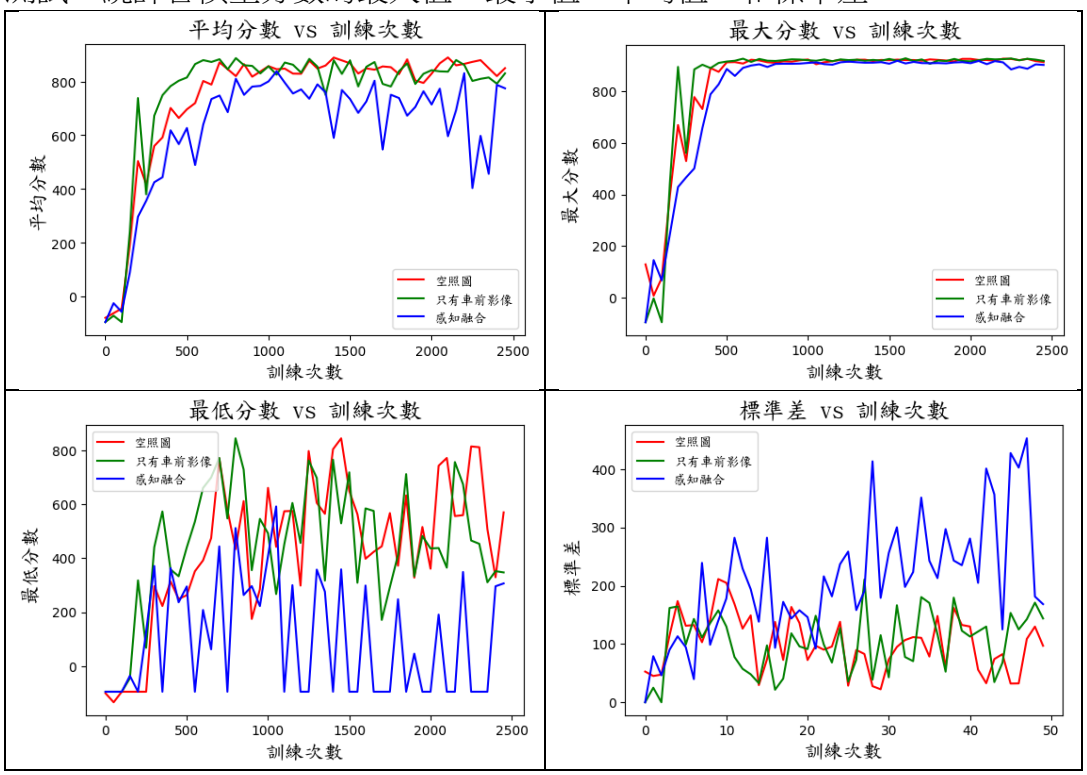
(一) 模型訓練情況比較：我們使用三種演算法分別訓練2500回合後，每50回合取一次平均，繪製成平均分數與訓練次數關係圖如下：

(1)在探索階段：只有車前影像的平均分數上升最快，空照圖次之，感知融合則是最慢。

(2)在貪婪階段：只有車前影像和空照圖表現差不多，感知融合略差。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(1)三種輸入觀測空間在最高分數的表現差不多，但感知融合的標準差較大，較不穩定。

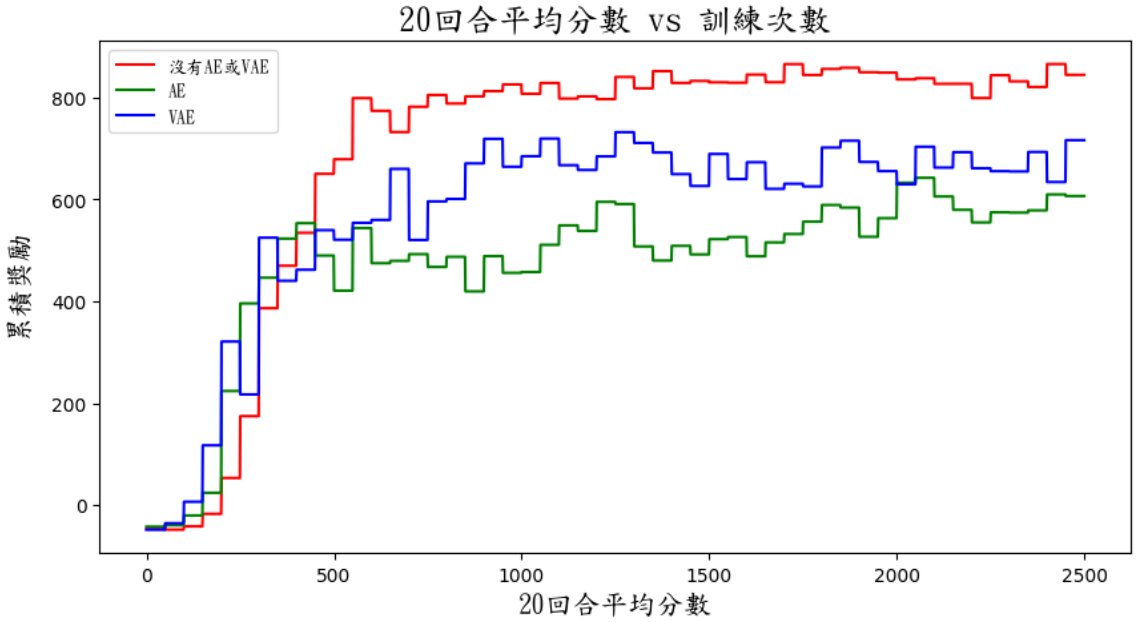
(2)只有車前影像在算法收斂前(約訓練400~500回合時)，平均分數和最高分數明顯優於空照圖。

六、使用生成式AI協助強化學習環境感知的訓練

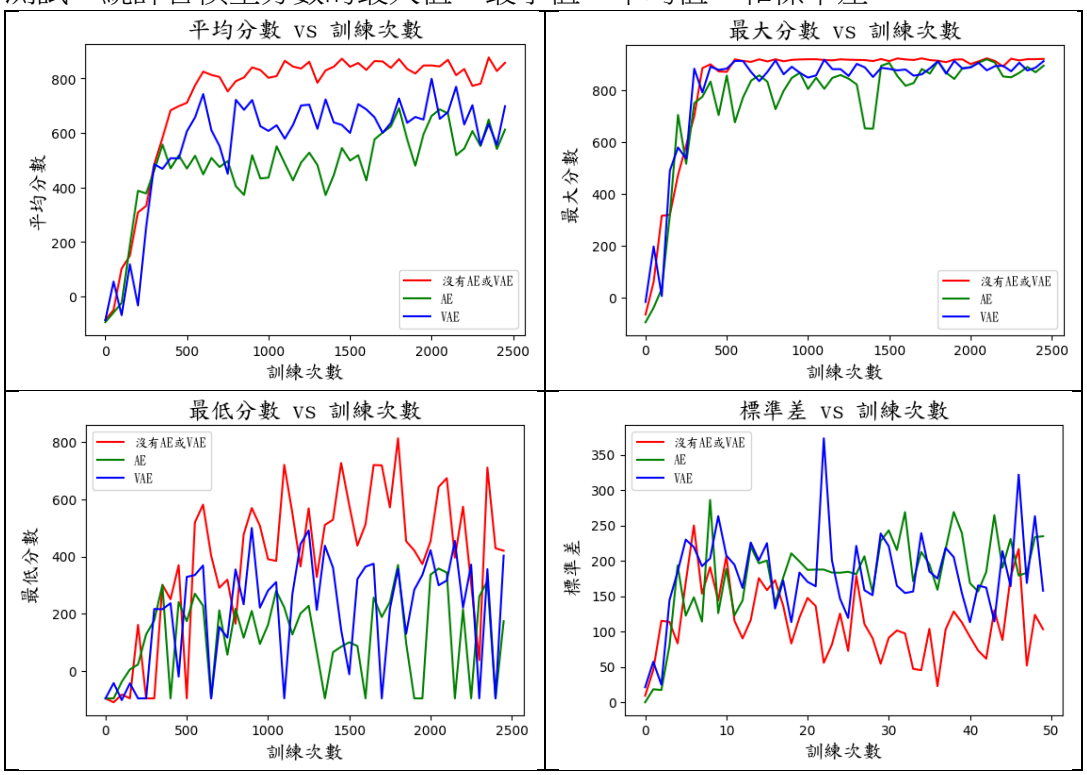
(一) 模型訓練情況比較：我們使用三種演算法分別訓練2500回合後，每50回合取一次平均，繪製成平均分數與訓練次數關係圖如下：

(1)在探索階段：VAE的平均分數上升最快，AE次之，沒有使用AE或VAE則是最慢。

(2)在貪婪階段：沒有使用AE或VAE平均分數最高，VAE次之，AE最低。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(1)由上面的最大值、最小值、平均值、和標準差各項指標觀察，沒有使用AE或VAE的表現優於有使用VAE和AE；而VAE又優於AE。

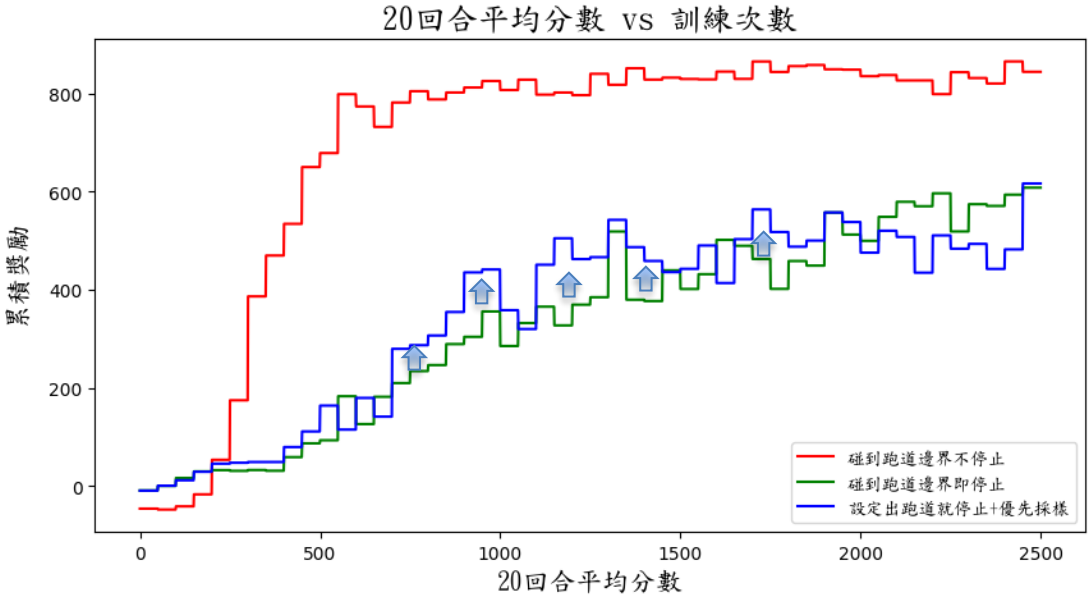
(2)VAE和AE在訓練過程中，表現也是逐步提升，其最高分也能和沒有使用AE或VAE相當，為穩定性較差，仍代表其可行性。

七、**調整Replay Buffer採樣機制**

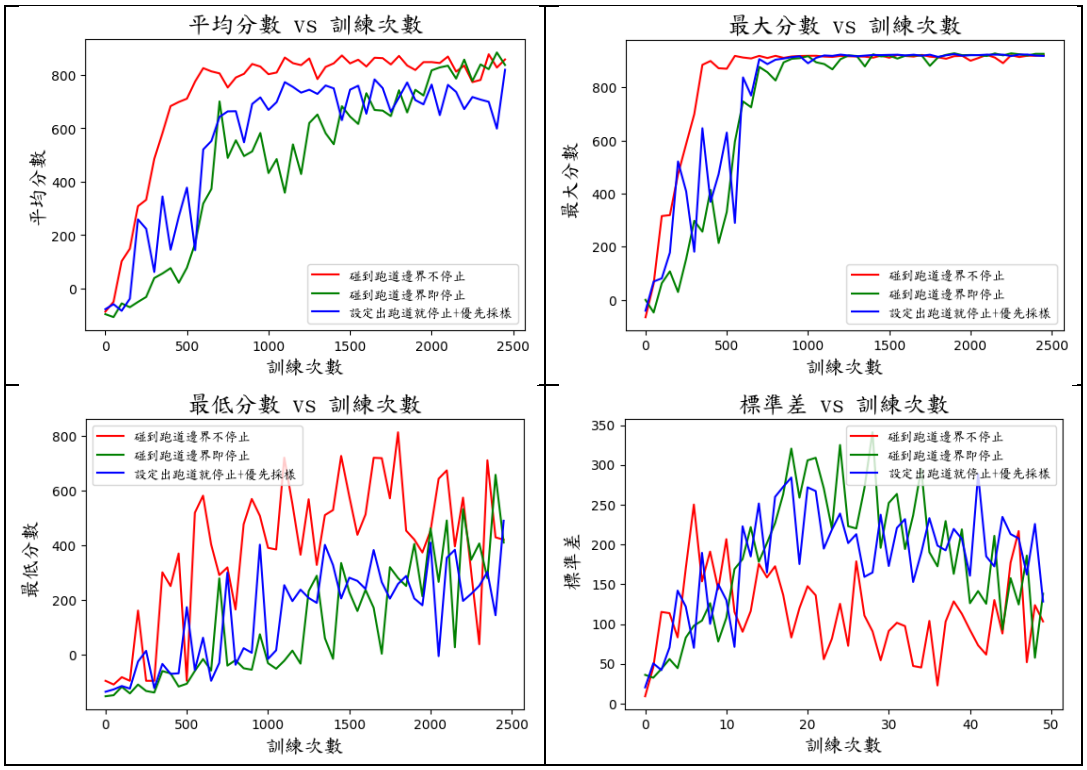
(一) 模型訓練情況比較：我們使用『碰到跑道邊界不停止』、『碰到跑道邊界不停止』、『碰到跑道邊界停止+優先採樣』分別訓練2500回合後，每50回合取一次平均，繪製成平均分數與訓練次數關係圖如下：

(1)可以看出若是出界即終止遊戲的條件會導致累積獎勵低於未設置此條件的模型，並且獎勵有緩慢的上升趨勢。我們推測因車輛一出界便終止遊戲，導致車輛較不易訓練到車道後期影像，進而導致獎勵上升速度慢。

(2)在加入優先採樣後，累計分數較高的車道環境被採樣訓練的機率較大，從下圖中也可以看出效果。



(二) 評估模型情況比較：如下表各圖，以每50回合訓練的完成的模型，進行20次測試，統計各模型分數的最大值、最小值、平均值、和標準差。



(1)可以看出有新增終止條件的模型於中後期的最大值已趨於穩定並與未有此條件的模型分數接近，但最小值、平均值皆明顯小於無終止條件的模型，並且標準差於後期大於無終止條件之模型。
我們推測因為有「出界即終止」條件，一旦失誤便會導致獎勵很低，進而拉低平均，也導致獎勵分數與最大值差異大，進而影響標準差的上升。

(2)「出界即終止」與現實生活情境更加符合，也更能確保車子盡量開在車道中，但在「出界不終止」可以提供Agent學習補救措施的能力，讓車子回到車道中繼續前進。

(3)使用任務採樣方式相較沒有使用任務採樣，在算法收斂前的表現較優，代表其有發揮加速學習的效能，但在算法收斂後期卻被超越，有可能是任務採養太集中於後面的訓練，導致發生過度擬合。

(4)雖然模型於平均值、最小值與標準差三個方面表現較無終止條件的模型差，但從後期訓練累積獎勵來看，模型有逐步成長的趨勢，獎勵最大值與未新增終止條件的模型相差無幾，因此我們推測模型增加穩定性，仍有持續進步的空間。

伍、討論

一、我們在使用OpenAI的Car_racing環境的離散動作模式產生錯誤，經過研究其原始碼，發現544行有Bug，這個debug過程使我們更了解程式碼，為後續環境的改寫奠定基礎。

二、在實作DQN的Replay Buffer時，一開始是使用python collections套件的deque資料型態，其本身就有儲列的功能，結果訓練非常緩慢，後來改用numpy.array來實現，雖然不像deque簡單，但其資料寫入和讀取速度較快，才大幅提升訓練的速度。

三、本來我們是在Google Colab上使用免費的GPU訓練，但常常訓練到中途就異常終止，後來發現Replay Buffer是放在主記憶體，被採樣的訓練資料才會移到GPU上面運算，原來是使用Replay Buffer的MaxSize設太大，用完主記憶體空間，故後來減少MaxSize就解決這個問題。

四、由於目前使用的DQN演算法，對於離散動作有良好表現，卻無法處理連續動作。本研究選用此演算法，將車輛動作簡化成五種，使模型較好收斂。然而，選用離散動作而非連續動作，可能導致車輛的動作控制不夠精細，例如車輛在高速過彎時，無法用最優的動作組合，進而影響車子穩定性。

五、本研究目的是透過不同種強化學習模型，提升車輛在環境中的駕駛性能。然而，部分研究結果不如預期。以下為我們對於智能體未達預期效果的原因推測及反思：由於car racing提供的環境僅有草皮與賽道，簡單的環境雖然能幫助模型快速收斂，但也限制了其對複雜模型的適應性。在過於單純的環境中加入感知融合、AE等演算法，不易讓智能體表現出該有的效果，更容易產生過度擬合。因此我們應該使用更接近真實的模擬真實駕駛環境，更能使智能體發揮其複雜模型的最大效果。

六、使用OpenAI Car_Racing賽車模擬環境，因為是較簡單的自動駕駛強化學習訓練環境，但是我們Nvidia Jetson Orin 16GB來訓練，每2500回合大約要24小時才能完成，更複雜的環境，顯然需要更大算力的GPU來支持。

陸、結論

一、以ε-greedy (部分貪婪)算法解決強化學習中探索（exploration）和利用（exploitation）是非常核心的問題，ε-greedy的性能取決於 ε 值：
(1)設太高：導致演算法的性能下降，因為沒有充分利用已知的最佳動作。
(2)設太低：可能會導致陷入局部最優解，沒有探索足夠多的狀態空間。
所以規劃適合的探索衰減策略是非常重要的。

二、利用在比較不同的衰減策略及DQN演算法的研究中，我們推論此環境較簡易單純且動作空間較少，所以只需使用較快收斂的衰減係數(即較小的衰減係數)即可，若是動作空間較多的環境可能需要更多的探索。

三、強化學習DQN演化法有許多改進版，我們比較了原始DQN、Double DQN、Dueling DQN，結果Dueling DQN在解決Car_Racing問題表現效果是三個演算法中最優。

四、修改環境的獎勵機制，藉以規範車輛行進時保持在車道中間，使每一個動作都能得到明確的優劣判斷，避免因為稀疏獎勵難以訓練，在實驗中修改獎勵後在探索階段，訓練確實較為快速。

五、自動駕駛車輛為了安全保障，使用各種感測器，如攝影機、紅外線、超聲波、光達、雷達等，要統整所有資訊，做出正確的決策，我們實作的感知融合模型隨著訓練雖有優化，但最終表現較Dueling DQN差。

六、我們將環境圖像裁切，只留下車前影像進行訓練，此舉會丟失車輛的圖像訊息，但使畫面更符合現實中自動駕駛的視野。
(一)在探索階段車子訓練環境幾乎為較單純的直線或小幅的曲線，推論可能是因其能更集中於判斷賽道環境而使智能體訓練速度加快。
(二)但到貪婪階段因為較多大幅轉彎，可能因少掉賽車本體的位置畫面，而使其表現不穩定。

七、「出界不終止」相較「出界即終止」，Agent更能學習到將車輛導正回到車道中的能力，但現實環境中車輛發生事故應該不能繼續行駛，因此我們認為在車輛出邊界前加一條緩衝線，以緩衝線為邊界來做訓練，碰到緩衝線給很大的負獎勵，但不終止，如此便可以訓練Agent在碰到緩衝線後採取補救措施，對自動駕駛的安全性增加一層保障。

八、透過調整Replay Buffer樣本的機率分佈，使用不同的任務採樣，我們可以調整Agent應該加強學習目標以增加學習的效率，但是否可能會造成部分樣本的過度擬合，也是要考慮的。

九、在VAE和AE的實驗中，VAE是學習大量圖片的機率分佈來生成，相較AE以產生原圖完全相同為目標，更具泛化能力，較不容易產生過度擬合，實驗結果VAE的表現也明顯比AE好。

十、強化學習的訓練資料完全來自於採樣，若模型太過複雜，環境太簡單，容易產生過度擬合，遇到沒採樣訓練到的資料就容易發生不穩定的情況，在我們實驗中部份模型都可以達到最高分數，但是穩定性不夠，必須更深入研究以克服。

柒、參考文獻資料

一、李茹楊, 彭慧民, 李仁剛, 趙坤. 強化學習演算法與應用綜述. 電腦系統應用, 2020, 29 (12): 13-25. http://www.csa.org.cn/1003-3254/7701 .html

二、Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

三、Sutton, R. S., Barto, A. G. (1992, May 31). Reinforcement Learning: An Introduction.

四、Pykes, K. (2024, November 6). Understanding the Bellman Equation in Reinforcement Learning. Datacamp. https://www.datacamp.com/tutorial/bellman-equation-reinforcement-learning

五、Wang, Z.,Schaul, T., Hessel, M., Hasselt, H. v., Lanctot, M., & Freitas, N. d.(2016).Dueling Network Architectures for Deep Reinforcement Learning.https://doi.org/https://doi.org/10.48550/arXiv.1511.06581

六、Watson, D. (2024, January 1). What Is a Double Deep Q Network? THE ENGINEERING PROJECTS. https://www.theengineeringprojects.com/2024/01/what-is-a-double-deep-q-network.html#google_vignette

七、劉智皓. (2023, April 25). 深度學習Paper系列(04)：Variational Autoencoder (VAE). Medium. https://tomohiroliu22.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92paper%E7%B3%BB%E5%88%97-04-variational-autoencoder-vae-a7fbc67f0a2

八、Nick. (n.d.). [Day5] VAE，好久不見. IT邦幫忙. https://ithelp.ithome.com.tw/m/articles/10323542

九、Bergmann, D., & Stryker, C. (2024, June 12). What Is a Variational Autoencoder? IBM. https://www.ibm.com/think/topics/variational-autoencoder

十、Lee, H.-Y. (2016, December 7). ML Lecture 18: Unsupervised Learning - Deep Generative Model (Part II). YouTube. https://www.youtube.com/watch?v=8zomhgKrsmQ&t=1309s

十一、Udacity team. (2020, August 25). Sensor Fusion Algorithms Explained. UDACITY. https://www.udacity.com/blog/2020/08/sensor-fusion-algorithms-explained.html

十二、算法集市. (2019, January 7). 判斷一點是否在多邊形內部：射線法. 每日頭條. https://kknews.cc/zh-tw/code/j59xqkq.html

十三、Tsai, Y.-R. (2019, March 9). What Are Autoencoders? Medium. https://medium.com/ai-academy-taiwan/what-are-autoencoders-175b474d74d1