以強化學習實作俄羅斯方塊遊戲

專題學生：蘇冠人
張博翔

指導教授：葉慶隆 教授

大 同 大 學
資 訊 工 程 學 系

專題報告

中 華 民 國 112 年 12 月

以強學實俄斯塊戲化習作羅方遊

大同資工系專題報告

姓名 蘇冠人

張博翔

112

致謝

本專題的完成，首先要感謝指導教授葉慶隆教授，當初三上時期教導我們什麼是強化學習，三下幫助我們如何選出自己想要的專題題目，透過三年級的雜誌研讀，讓我們發現許多有趣的研究項目以及應用方法，之後在暑假期間的強化週確實給了我們一些明確的方向，自從那時開始就在更努力專研 Q-learning 這方面的知識，我們還要感謝的是大四上專題展的參展老師，感謝評審們那個時候給出的建議，像是我們做出來的東西，評審說只看到遊戲畫面，無法跟強化學習做聯想，告訴我們必須將 RL 以及我們的專題之間的關係用公式，以圖來講解會比較好，還有就是另外一位評審點出我們環境的複雜度可以在更加提升，以便測試我們做出來的模型是否為一個全面且強大的模型，要是它在更複雜的環境下，也能表現得如此優異，跟人類表現差距越拉越大，那就代表這個模型是成功的。

摘要

俄羅斯方塊是一款經典益智遊戲，以簡單的規則和變化多端的遊戲場景風靡全球，在遊戲過程中玩家需要極盡所能的填滿並消除每一行，避免畫面堆疊至最高，並存活下去，考驗著玩家們的反應速度及策略思維能力。

我們使用了由 Viet Naguyen 開發並在 github 上發表的 Tetris-deep-Q-learning-pytorch 環境進行實驗，我們提出使用強化學習訓練俄羅斯方塊的智能體，嘗試了不同獎勵方式，並且發現了不同獎勵的弊端，而過程中也改用 Double-DQN 演算法進行訓練，更在每個方塊生成方式進行修改，使其環境變得更為複雜，讓智能體去適應更為極端的遊戲環境，以便他去應對較為平凡的環境。

關鍵字：俄羅斯方塊、遊戲、強化學習、Double-DQN。

目錄

表目錄

圖目錄

第1章　緒論

1.1　研究背景

隨著電腦硬體逐漸成熟，使得人工智慧在近期間飛速成長，這種成長不僅體現在訓練時間，更甚至是複雜的演算法、應用，這樣的進步使得 AI 在各個領域都有顯著的發展。

強化學習屬於機器學習中的其中一個分支，其核心概念是讓代理在與環境的互動中學習，決定當下最佳的動作，如同模擬生物的學習行為，藉由獎勵機制使得代理能夠逐漸調整出一套決策。

強化學習真正開始發展是由於 Playing Atari with Deep Reinforcement Learning[1]這篇論文，在這篇論文中他們使用新的 Q-learning 架構，結合 Atari 各項遊戲，而且成功獲得跟人類一樣的分數，這篇論文的應用讓大家了解到現今硬體已經能夠負荷神經網路的處理，並且能在短時間內收斂到一個非常好的網路，自此開始，出現各種使用強化學習應用在不同領域的研究以及改良方法。

最經典的案例便是 2015 年的 Alpha GO 擊敗了南韓棋王李世乭，這次在圍棋領域的勝利，不僅僅創造了人工智慧的一個里程碑，並證明了人工智慧在複雜的任務中，有潛力超越人類，以此為契機，爾後幾年人工智慧也在越來越多領域中活躍，例如醫療、金融、遊戲等都有諸多實際案例。

1.2 研究動機

本研究動機在於探討智能體應對即時策略性的遊戲環境中會有何種表現，是否能夠輕易超越人類，另外也希望跳脫傳統 Atari 遊戲框架，藉此可以證明強化學習能夠更廣泛的應用，不再侷限於 Atari 遊戲中，而我們選定了規則較為簡單易懂，並且不是 Atari 遊戲的俄羅斯方塊來當作環境，這樣便不必著墨於複雜的遊戲機制，而是更專注在應用層面上。

1.3 研究目標

本研究主要目標是可以將強化學習應用俄羅斯方塊上，透過傳送環境參數的方式，訓練出一個能夠比擬甚至超越人類表現的智能體。

為達上述目的，我們進行以下步驟：

1. 建立遊戲環境

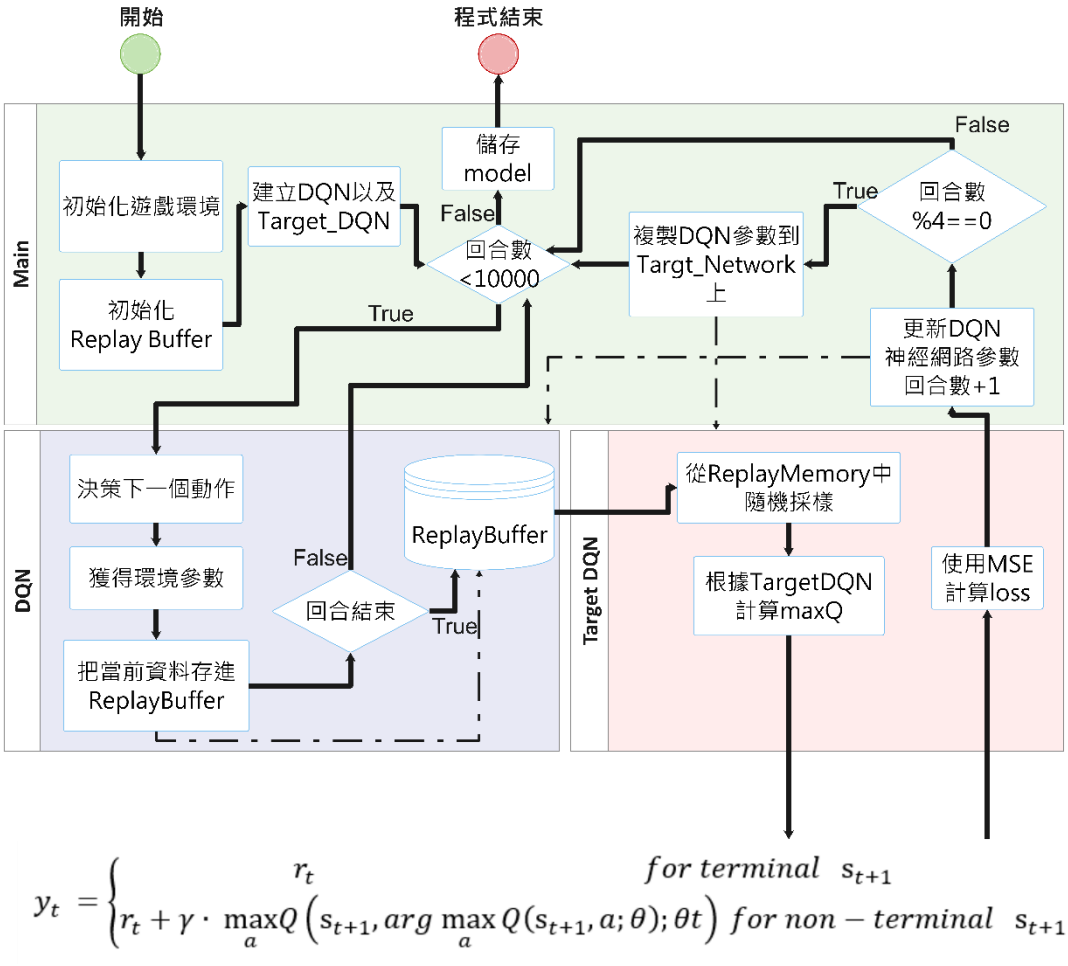
2. 使用 DQN 讓 Agent 能自主學習如何玩俄羅斯方塊

3. 使用 Double DQN 演算法以增加表現的穩定性及更快的達到一般水準。

1.4 系統架構與預期成果

首先建立俄羅斯方塊的遊戲環境，再來照著 DQN 論文的虛擬碼，先建立兩個神經網路，然後開始執行遊戲，一開始把狀態輸入給 DQN，獲得一個動作後，輸入到遊戲當中，獲得我們設定的獎勵以及下一個狀態後，再去把目前的環境以及變數記錄到 ReplayBuffer 當中，最後依照設定的回合數更新 TargetDQN。

我們預測這樣能使得 Agent 達到穩定得分並且不會輕易地死掉。



$$y_t = \begin{cases} r_t & for\ terminal\ \ s_{t+1} \\ r_t + \gamma \cdot \max_a Q\left(s_{t+1}, arg\max_a Q(s_{t+1}, a; \theta); \theta t\right) & for\ non-terminal\ \ s_{t+1} \end{cases}$$

圖 1.1 系統架構圖

1.5 專題專案管理與成本分析

我們在大三下學期初時，選擇要做第一人稱射擊遊戲內的影像辨識，但嘗試了數個月後，發現因技術力不足和依然有法律上的疑慮，而更改題目。

表 1.1 專案進度表

日期	內容

112/02	選擇題目
112/07	更改並確定專題題目
112/07~112/09	嘗試使用不同環境
112/10~112/12	確定環境並進行訓練及各項參數調整
112/12	製作最終報告及測試模型

研究人員方面我們使用最低時薪(176 元)計算，2023 一月至七月每月平均 18 小時在進行其他專案測試，兩人共計 38,880 元，2023 七月後每月平均大約 25 小在進行專案工作，兩人共計 54,000 元，合計 92,880 元

表 1.2 成本分析

內容	金額
硬體設備(主機、螢幕、鍵盤、滑鼠)	55,000
研究人員(2 名)	92,880

第 2 章 文獻探討

2.1 強化學習

強化學習(Reinforcement Learning, RL)是機器學習的一種分支，其中元素如表 2.1 所示，核心概念是讓代理(Agent)與環境(Environment)的互動，不斷嘗試各種動作 a_t 的方式，如圖 2.1，每做完一次動作便會收到執行該動作的獎勵 r_t 和新的環境狀態 s_{t+1}，這個獎勵也有可能為懲罰，並在與環境交互的過程中，藉由回傳的獎勵跟狀態，嘗試找出一套最佳的策略(Policy)，以最大化獲得的獎勵，並達成目標。

如同一個社會新鮮人(Agent)進入了一個職場環境(Environment)，在這環境下他有許多的工作(Action)可以執行，比如接洽客戶、製作報表等，而小組人數、每項工作的進度、每個人的情緒等，這些稱為狀態(State)，並且執行動作都會獲得上司或同事的回饋(Reward)。

表 2.1 強化學習元素

元素	說明	例子
代理(Agent)	負責學習並採取策略的主體	操作正在下落的方塊
環境(Environment)	與代理進行交互並給予反饋	俄羅斯方塊遊戲環境
動作(Action)	代理能夠決定的動作	旋轉方塊
狀態(State)	環境資訊	分數、方塊種類、方塊個數
獎勵(Reward)	評估執行動作的好壞	遊戲結束:-20 分

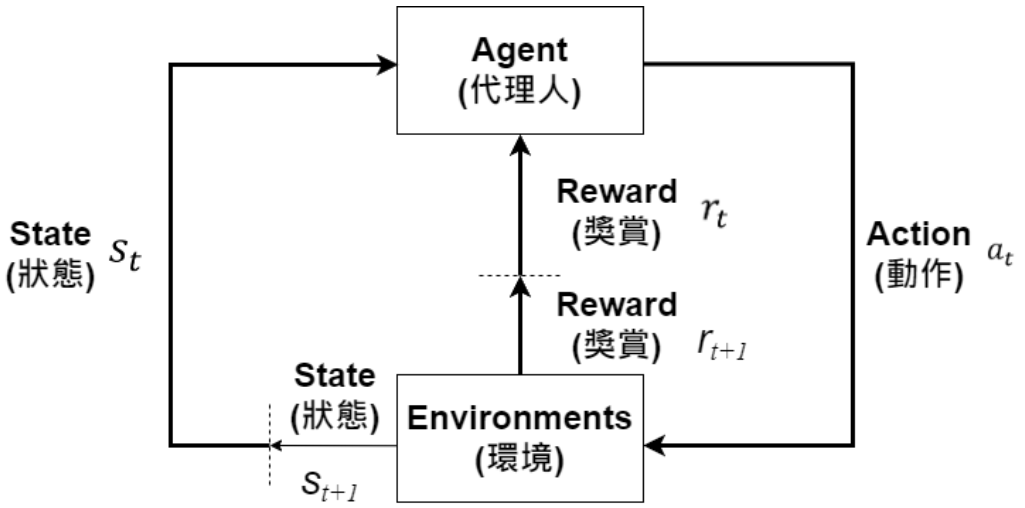


圖 2.1 強化學習流程圖

2.2 Q-Learning

Q-Learning 是一種強化學習的演算法[2]，它追蹤當前步驟的狀態和行動，然後將這些資訊傳遞給 Agent，使其能夠理解哪些行動能夠獲得最大化的獎勵。

Q-Learning 中有一個關鍵概念 Q-Table，當 Agent 在決定如何選出動作時，它會

查找 Q-Table 中對應的狀態，並查看當前狀態中所有可能動作的 Q-Value，即用來儲存每個狀態-動作對應的 Q-Value，這些值反映了在特定狀態下採取特定行動所能獲得的預期獎勵，並從中挑選 Q-value 最大的動作，這就是 Agent 輸出動作的過程。

整個 Q-learning 最關鍵的就是 Q-table 的更新，Agent 輸出給環境一個動作後，得到獎勵 r及下個狀態 s_{t+1}，然後利用貝爾曼方程式作為 Q-table 的更新方法，該方程式使用前一個 Q-value 以及 r 來更新 Q-table。

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a)\right) \quad (2.1)$$

通過這種方式，Q-table 逐漸學習到在給定狀態 s 下，哪些行動最有可能導致最大獎勵，從而指導 Agent 採取最佳行動。隨著時間的推移和足夠的迭代，這個過程將使 Q 表收斂到一個接近最優策略的狀態。[3]

Q-learning pseudo

Algorithm 1: Q-learning

初始化$Q(s, a)$

重複執行回合:

初始化狀態 s

每一回合重複執行步驟:

根據Q的動作策略(e.g., ε-greedy$s$)選擇一個動作 a 給狀態 s

採取行為 a，獲得獎勵 r及下個狀態 s_{t+1}

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a)\right)$$

$s = s_t$

重複執行直到 s 結束

2.3 DQN

對於 Q-table 來說，它只能適用於有限的應用場景，因為進行演算法時，會根據目前的環境建立相對應固定大小的 Q-table，而不是適應連續空間的結構，所以 Q-learning 缺點是無法在連續動作以及狀態中進行好的學習。

2013 年 DeepMind 發表了一篇論文[1]，裡面使用 DQN 架構使得 Q-learning 不再被侷限於固定場景，他們使用 DQN 來玩雅達利（Atari）遊戲，並且獲得超越人類水平的表現。自那時起，DQN 被用於許多應用，包括醫學、工業控制與工程、交通管理、資源分配以及遊戲。此外，DQN 的工作為改進許多理論和為深度強化學習（Deep Reinforcement Learning, DRL）演算法的發展打開了大門，到目前為止，DQN 論文已被引用超過 13738 次。

DQN 的實作方式就是用神經網絡（Neural Network, NN）來替換 Q-table，雖然在簡單的環境中這個方法是可行的，但是在執行複雜任務時，會遇到兩大問題[3]:

- 依賴經驗(The experience correlation): 當 Agent 從連續的經驗中學習時，數據可能高度相關。這是因為特定狀態可能引發某一行動，進而引發特定的狀態和行動，因為這些環境是根據固定轉換函數所產生的。
- 移動目標問題(The moving target problem):

  RL 中 Agent 生成自己的數據，Agent 政策的更改越多，數據分佈的變化就越大，這使得計算 Q 值非常不穩定。因此，模型被認為是在追逐一個不固定的目標。每次 Q 網絡更新，Q 值也會不斷變化。如果更新計算為

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a)\right) \quad (2.2)$$

  ，則學習可能變得不穩定，因為目標$Q(s_t, a_t)$和 $\max_a Q(s_{t+1}, a)$都依賴於 Agent 這個 NN。

為了解決上述的問題，DQN 使用另外一種結構，並且有效緩解上述的問題

- 經驗回放(experience replay):用於打破連續經驗之間的相關性，並確保 *s*

  independent, identical, distributed（IID）-like 的數據。它通過存儲{ state,

  action, reward, new state }的數據，直到擁有這些轉換的批次為止。然後，在

  與環境互動/經驗生成階段分離的階段中，隨機抽取轉換進行學習。[1]

Algorithm 2: Deep Q-learning with Experience Replay

初始化 replay memory D 容量為 N

用隨機權重初始化動作價值函數 Q

Q為 DQN 神經網路

a_t代表當回合動作

s_t代表當回合狀態

for episode = 1, M do:

初始化序列 $s_1 = \{x_1\}$

for episode = 1, T do:

根據機率 ε 選擇隨機動作 a_t

或是選擇動作 $a_t = \max_a Q(s_t, a)$

執行 a_t 給環境，獲得 r_t 圖片 x_{t+1}

設定$s_{t+1} = s_t,\ a_t, x_{t+1}$

儲存$(s_t,\ a_t, r_t, s_{t+1})$到 D 中

從D中隨機抽取小批次的過程(s_t, a_t, r_t, s_{t+1})

設定 $y_t = \begin{cases} r_t & for\ terminal\ \ s_{t+1} \\ r_t + \gamma \cdot \max_a Q(s_{t+1}, a) & for\ non-terminal\ \ s_{t+1} \end{cases}$

根據 MSE 對 $(y_t - Q(s_t, a_t))^2$進行梯度下降

end for

end for

2.4 Double-DQN

為解決剛剛提到的移動目標問題(The moving target problem)問題，我們使用目標網絡(Target Network)，他是透過設置一個單獨的網絡來計算目標 Q 值，使其模型在更新時不會一直有不同的更新目標，而使得模型變得模稜兩可，使用 DouebleDQN 這樣的方法會使得 Agent 訓練變得更加強力和穩定[4]。

實作方法就是在 DQN 原有的結構上，新增一個新的 NN 把它命名為 Q-Target-Network，平常訓練時使用 Q network 來取得最佳動作以及更新神經網路參數，但是在更新的時候使用 Q-Target-Network 來跟原本的 Q network 做平均平方誤差(MSE)，再去更新 Q network 目前的參數。

這樣的更新方法就會使得 Q-network 的最佳化目標會在當前回合是固定的，以俄羅斯方塊來說就是他不會這一步想得分，下一步卻想活得更久，能確保他在一場遊戲內使用的是同一種策略，這樣使模型成長得更穩定，缺點就是因為在訓練當中，Q-target-network 是必須要做更新的，否則他將會不會進步，所以我們會設定，每過幾個回合，就將當前的 Q-network 的參數複製到

Q-target-network，這種方式會導致整個神經網路更新的速度變慢，但是準確跟穩定上升。[3]

θ 定義為 Q-network 神經網路參數

θt定義為 Q-target-network 神經網路參數

更改 Y 值更新公式為[4]

$$y_t = \begin{cases} r_t & for\ terminal \quad s_{t+1} \\ r_t + \gamma \cdot \max_a Q\left(s_{t+1}, arg\max_a Q(s_{t+1}, a; \theta); \theta t\right) & for\ non-terminal \quad s_{t+1} \end{cases}$$

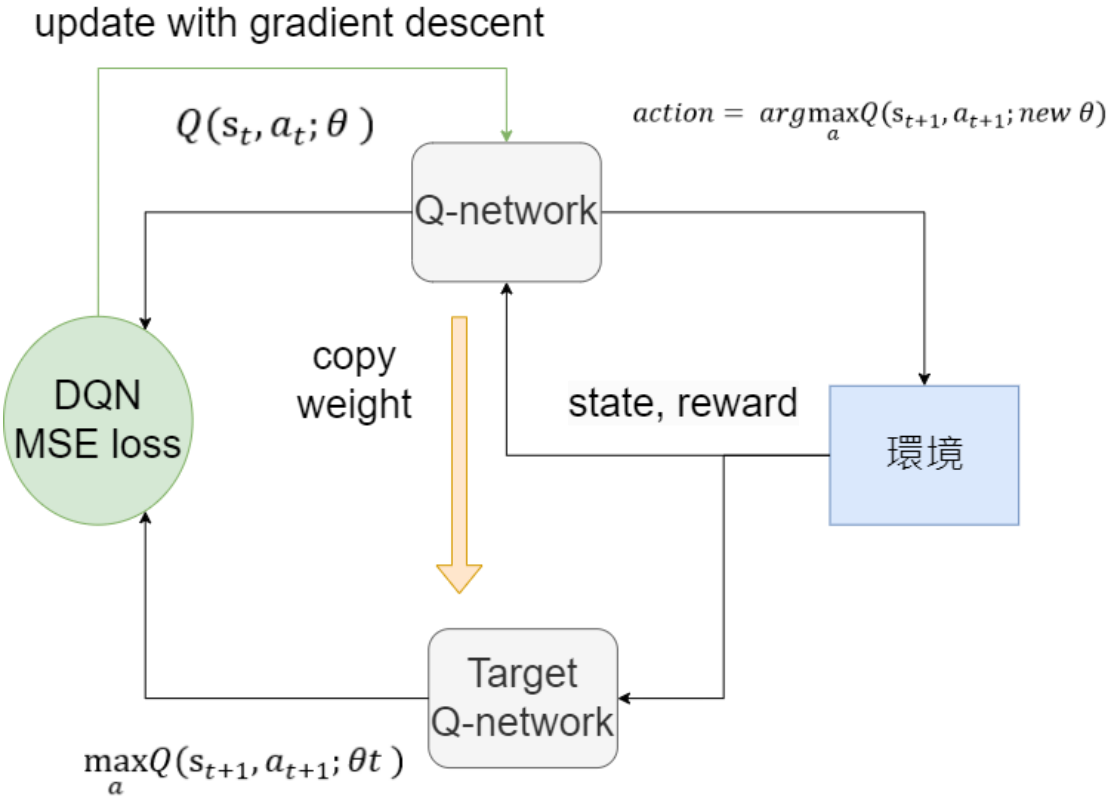


圖 2.2 Double-DQN 流程圖

第 3 章 研究方法

3.1 俄羅斯方塊

俄羅斯方塊是一款經典的益智遊戲，玩家需要操控七種不同形狀的方塊，每一種方塊會隨機出現，透過旋轉方塊改變方塊下落的面向，當每一列被填滿時，便會消除該列並獲得分數，而超出畫面最高格子時，遊戲便會失敗結束。

過程中，玩家需要在不斷變化的情境中做出決定，由一開始的單機遊戲，單純的計算分數，存活時間越長下落速度越快等，之後更延伸出了多人玩法，可能會在消除每一列的同時對敵人造成不同的負面狀態，可能是隨機從最底下產生一列增加高度，或是在對手場地隨機增加方塊讓玩家難以處理，也因為這些遊戲中的不確定因素，能夠增加玩家挑戰性及好勝心，誕生出了俄羅斯方塊世界錦標賽，同時也助長了遊戲的名氣。

即使遊戲畫面單調、遊戲內容簡單易懂，但卻造成了後續遊戲文化深遠的影響，使其依然能夠在現今的遊戲界有著舉足輕重的地位。

3.2 流程圖

我們把強化學習應用在俄羅斯遊戲上，透過第二章的強化學習流程圖來介紹整個遊戲的流程，圖 3.1 中 Environment 代表著遊戲程式本身，一開始透過環境給 Agent 環境參數，然後 Agent 會根據參數輸出一個遊戲動作，順時鐘、逆時鐘轉，輸入到遊戲裡面，然後遊戲會回傳一個 State，也就是遊戲畫面的資料，跟他的 reward，最後再把這些參數傳回 Agent。

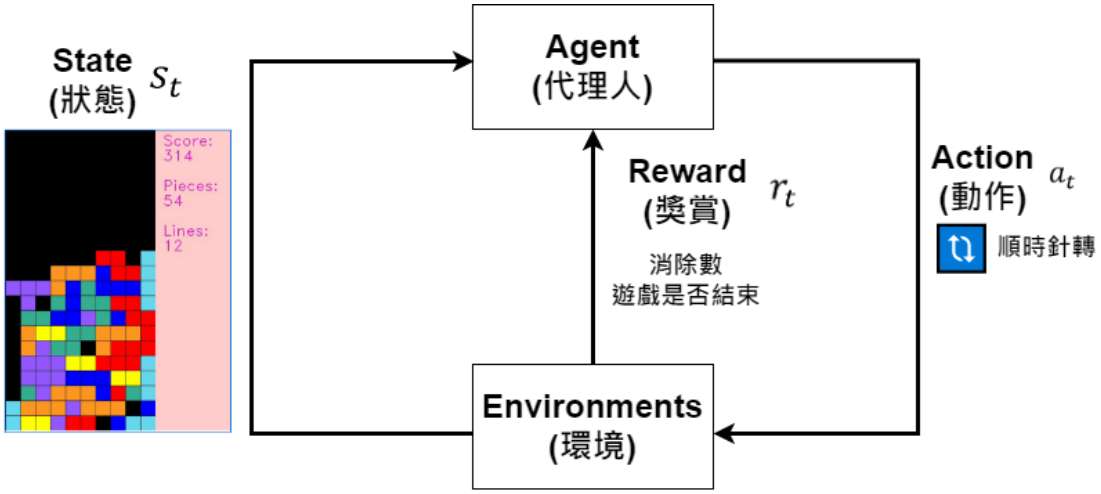


圖 3.1 RL 流程圖

3.3 研究環境介紹

我們所使用的遊戲環境是由 Viet Naguyen (uvipen) 開發並在 github 上發布的 Tetris-deep-Q-learning-pytorch，並修改其獎勵函數，原始獎勵函數參見公式 3.1。

$$R_t = \begin{cases} 1 + \left(cleared\_line_t{}^2\right) * 10 & if\ not\ gameover \\ -20 & if\ gameover \end{cases} \quad (3.1)$$

我們也修改了原始生成方塊機制，原始機制為將七個不同形狀的方塊依照

數字由小到大放入一個陣列後進行打亂再將第一個抽出選為下一個方塊，而我們將生成方式改為隨機抽出兩個可重複數字，並將對應的方塊放入陣列中再進行 POP 選定為下一個方塊增加環境複雜度。

原本所使用的訓練演算法為 DQN，後續我們改為使用 Double DQN 進行訓練。

而在過程中，我們是通過傳遞各項環境資訊參數的方式進行訓練，所用參數如表 3.1，而畫面顯示僅是提供給人類進行參考方塊落點及排列方式，並不包含代理進行旋轉或是移動等動作。

表 3.1 環境參數

參數名稱	功能	初始值
height	遊戲環境高度	20
width	遊戲環境寬度	10
score	當前分數	0
holes	空洞數	0

tetrominoes	放置方塊數	0
cleared_lines	消除條數	0
bag	儲存方塊	0~6 隨機產生 2 個數字
ind	當前方塊	bag.pop()

3.4 設計獎勵函數

獎勵函數是在代理對環境互動的過程中，所得到的一種反饋機制，透過這個機制讓代理認知到當下的動作是否值得去執行，而進行策略的規劃，例如在現實生活中，我們要訓練一隻狗坐下時，我們會利用小餅乾去當作正面獎勵，若牠對主人不理會我們所下的指令，我們可能也會利用表情或是減少互動來給予牠負面獎勵，長久下來牠便能得知道聽到坐下並且坐下後，會得到小餅乾。

而俄羅斯方塊對於人類來說的直觀感受便是是否造成空洞，因為空洞容易造成多一列甚至更多無法消除，如圖 3.2 所示，使得高度越來越高。

圖 3.2 空洞範例

而消除條數、方塊堆疊高度也是人類會去在意的重點，因此也可以將兩項變數納入獎勵函數，而遊戲的目標為消除方塊並且活下去，因此若方塊堆疊到達最高點，即為遊戲結束，因此也要考量遊戲結束時所給予的懲罰。

3.4.1 膽怯反應

我們設計了針對空洞去做懲罰，並且根據消除方塊作為獎勵函數，如公式 3.2.1，R_t為每一步動作所獲得的獎勵，$cleared_lines_t$是每一步所消除的條數，$holes_t$是每一步版面存在的空洞數，而訓練幾次後的結果發現了 AI 產生了膽怯反應，會將左右兩邊疊高後再往中間進行疊高的動作，而不會進行消除動作，推測是因為我們不斷給予懲罰，而無任何獎勵，導致代理害怕失敗，為了避免懲罰的發生，代理選擇採取較為保守且安全的做法，因此不敢做其他嘗試，造成她形成了一個較為保守的策略。

$$R_t = \begin{cases} cleared\_lines_t{}^2 & if\ cleared\_lines_t > 0 \\ -(holes_t * 10) & if\ cleared\_lines_t = 0 \\ -20 & if\ gameover \end{cases}$$ 公式 (3.2.1)

根據 Steve Bakos 及 Heidar Davoudi [6] 提出的為了消除膽怯反應造成的影響，我們要先將代理該獲得的獎勵還回去，並且不要一次給予過多的懲罰，便能減少代理對於懲罰的恐懼，但是保留結束時給予的懲罰。

因此我們提出獎勵機制再進行每一步時無論是否有效都會加分，並且不再去著重判斷是否造成空洞上，如公式 3.2.2 所示，而是讓代理自行找到每一步的最大化獎勵，或許能利用空洞進行人類難以想像的操作。

$$R_t = \begin{cases} 1 + (\text{cleared\_lines}_t * 10)^{\,2} & if\ cleared\_lines_t \geq 0 \\ -20 & if\ gameover \end{cases}$$ (3.2.2)

N為結束前最終步數而每一局的最終獎勵為R_{final}，如公式 3.2.3 所示，

$$R_{final} = \sum_{N}^{t=0} R_t$$ (3.2.3)

3.5 超參數及其他設計

整個過程我們都固定使用學習率為 1e-3，衰退率為 0.99，並訓練 10000 局，若單局放置方塊數超過 20000 個便會結束該局並紀錄一次模型，避免訓練過久，並且每 500 局也會記錄一次模型。

在後續測試中，我們將會使用 10000 次中最穩定的模型進行模擬及測試，以避免過擬合的狀況發生，並以類似環境與人類實際遊玩的狀況進行比較，比較的過程中，只會比較步數、對應分數和最終分數。

第 4 章 研究結果

4.1 研究設備與環境套件

本研究使用 Python 3.10.9 並搭配如表 4-2 所列相關套件。

表 4.1 研究設備

硬體設備	規格
作業系統	Windows 10
CPU	AMD【6 核】Ryzen5 5600X
記憶體	DDR4-3200 32G(16G*2)
GPU	EVGA GeForce RTX 3070

表 4.2 環境套件

套件	版本
Cuda	11.8
opencv-python	4.8.0.76
torch	2.0.1
tensorboard	2.14.1

4.2 程式實作

在本次研究中使用 20*10 方塊大小的遊戲空間，Agent 依照俄羅斯方塊的規則去遊玩遊戲，首先在遊戲中會隨機得到一個方塊，Agent 會對目前環境和方塊，計算所有可能的結局進行 Q 值計算，找出當中 Q 值最大的動作，選擇那個動作當作 Agent 實際的輸出，輸入到環境後，得到下一個環境以及 Reward，Reward 為

$1 + (\text{cleared\_lines}_t * 10)^2$，再來對 Agent 進行更新，更新後的 Agent 再用剛剛得到的環境進行選擇動作，直到遊戲結束。

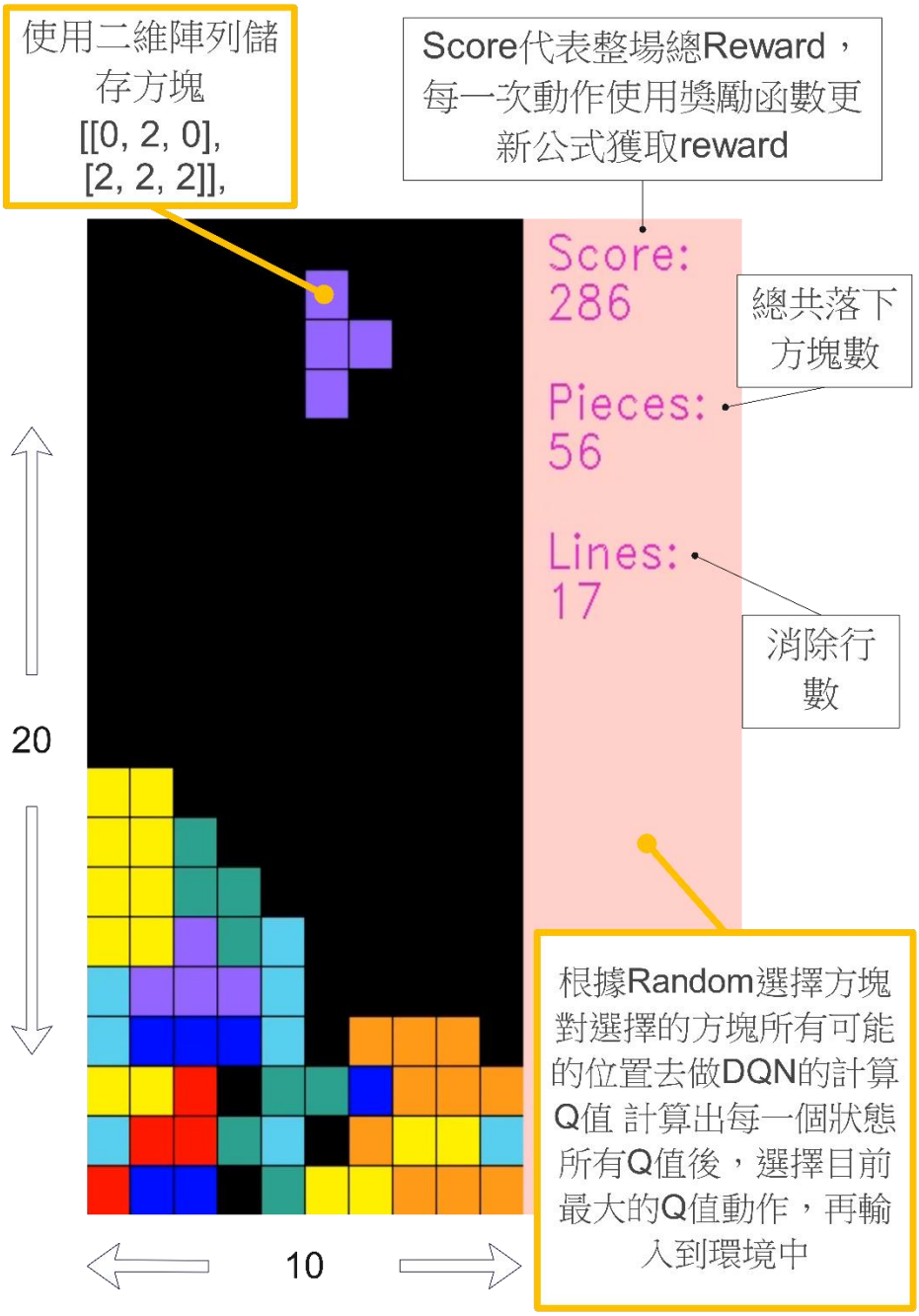


圖 4.1 遊戲架構

作者使用了 DQN 的架構，如圖 4.2 來訓練模型，但是穩定及效率不好，我們更改其架構如圖 4.3，增加一個神經網路，以及更改環境的獎勵函數，嘗試讓模型訓練得更加有效率，我們主要更改的地方有下面四點：

1. 更新獎勵函數：在 3.4.1 節我們的獎勵公式最終決定為

$$R_t = \begin{cases} 1 + (clered\_lines_t * 10)^2 & if\ cleared\_lines_t \geq 0 \\ -20 & if\ gameover \end{cases} \quad (3.2.2)$$

更改的部分為 Reward 的更新公式以及最終遊戲結束時的獎勵。

2. Double DQN：

首先為整個遊戲新增一個神經網路，將其命名為 TargetNetwork。

在更新 Y 值時，根據虛擬碼當中的

$$y_j = \begin{cases} r_j & for\ terminal\ \ \varphi_{j+1} \\ r_j + \gamma \cdot \max\limits_{a'} Q(\varphi_{j+1}, a', \theta') & for\ non-terminal\ \ \varphi_{j+1} \end{cases}$$

使用 TargetNetwork 計算 $\max\limits_{a'} Q(\varphi_{j+1}, a')$ ，將計算出來的 Q 值乘上 γ 加上當前獎勵 r_j

3. 使用剛才計算的 Y 值，根據 MSE 對 $(y_j - Q(\varphi_j, a_j))^2$ 對神經網路進行更新

4. 每經過 4 回合才更新一次 TargetNetwork

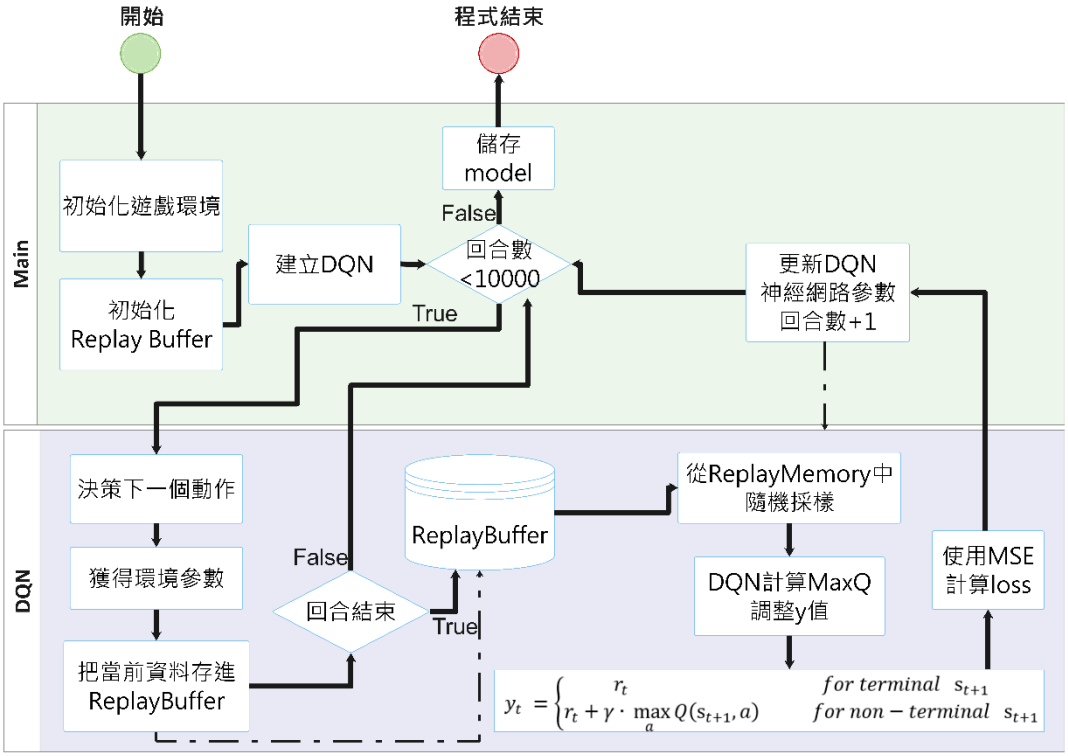
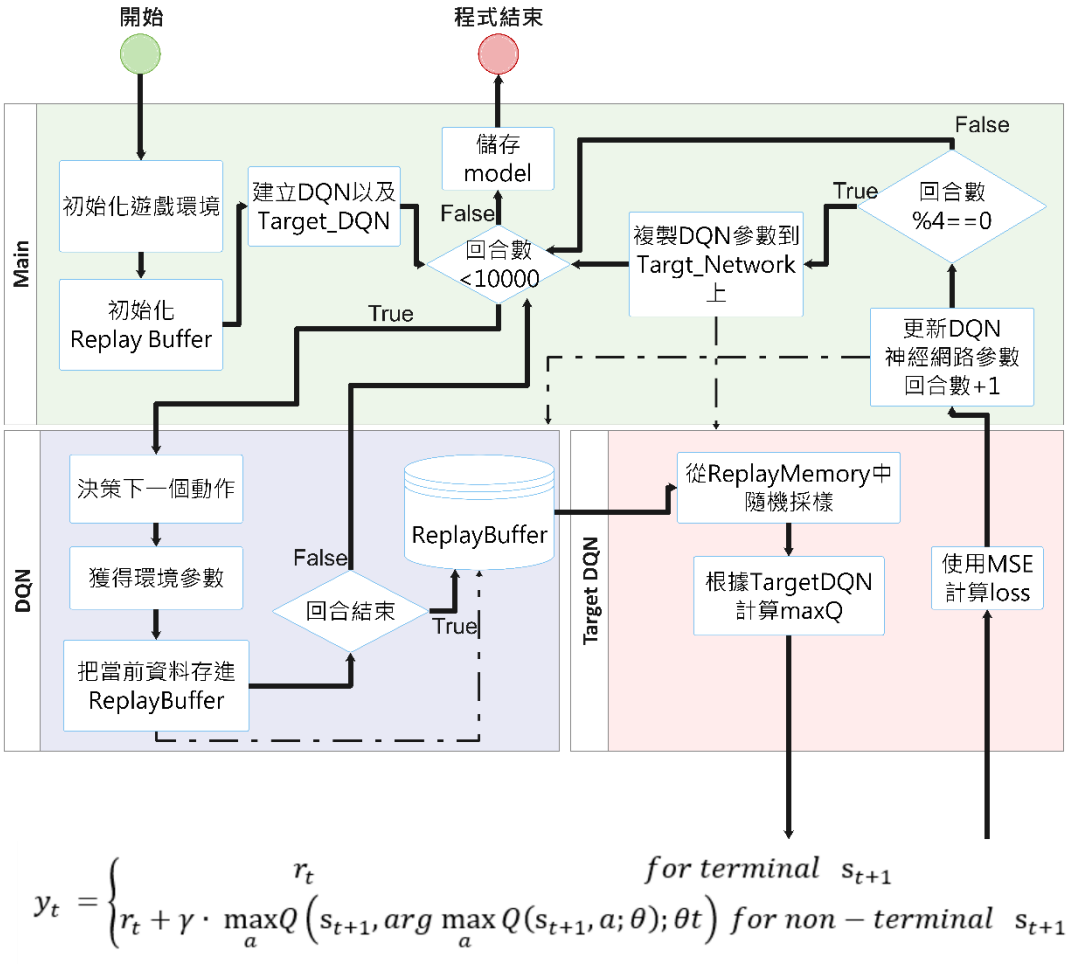

開始
程式結束
Main
初始化遊戲環境
初始化
Replay Buffer
建立DQN
回合數
<10000
False
True
儲存
model
更新DQN
神經網路參數
回合數+1
DQN
決策下一個動作
獲得環境參數
把當前資料存進
ReplayBuffer
回合結束
False
True
ReplayBuffer
從ReplayMemory中
隨機採樣
DQN計算MaxQ
調整y值
使用MSE
計算loss
$y_t = \begin{cases} r_t & for\ terminal\ \ s_{t+1} \\ r_t + \gamma \cdot \max_a Q(s_{t+1}, a) & for\ non-terminal\ \ s_{t+1} \end{cases}$

圖 4.2 DQN 架構



$$y_t = \begin{cases} r_t & for\ terminal\ \ s_{t+1} \\ r_t + \gamma \cdot \max_a Q\left(s_{t+1}, arg\max_a Q(s_{t+1}, a; \theta); \theta t\right) & for\ non-terminal\ \ s_{t+1} \end{cases}$$

圖 4.3 Double DQN 架構

4.3 DQN 模型表現

一開始我們在訓練 Tetris 是用原本 DQN 實作方式，如圖 4.4，當初在試的時候發現到它的每回合的分數落差非常大，雖然有機會得到高分，但是整體表現不太好。

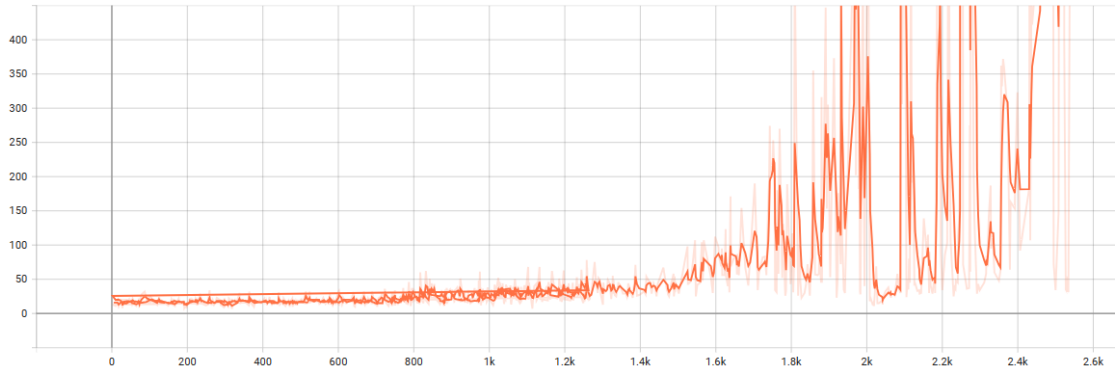


圖 4.4 DQN score

4.4 遊戲過程比較

在 model300 的遊戲畫面，agent 在玩的時候並沒有注意到高度以及坑洞的問題，導致他在第 10、15、20 塊時，並沒有往最右邊那一個大空格放，大致他在第 30 多塊時回合結束，相較於訓練 8013 次的 model 他在第 10 塊到第 15 塊中間，沒有像 model300 一樣，一直堆高最旁邊的高度，而是優先填補空格，降低高度，讓自己可以活得越久，得分越高。

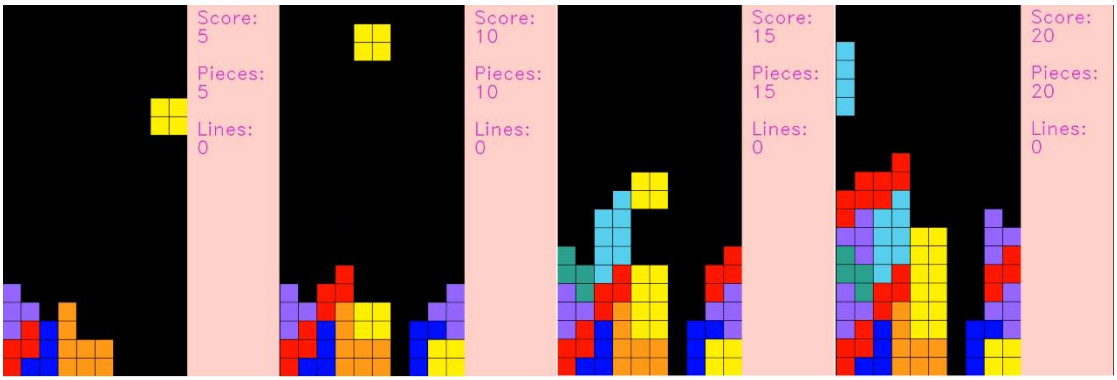


圖 4.5 model 300

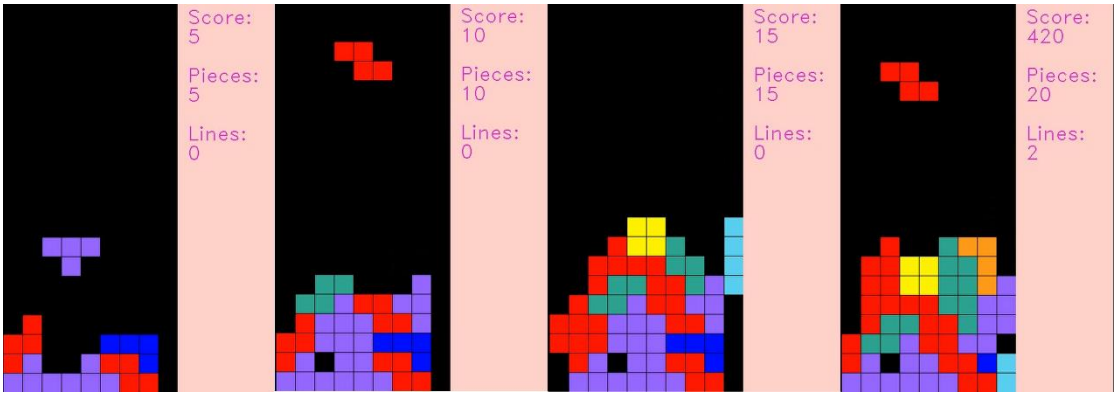


圖 4.6 model 8013

4.5 訓練過程對比

經過 Tensorboard 的圖表分析，我們發現 Double-DQN 比 DQN 消除的最高行數還要多，以及 Double-DQN 消除超過 500 行的次數比 DQN 來的頻繁許多。

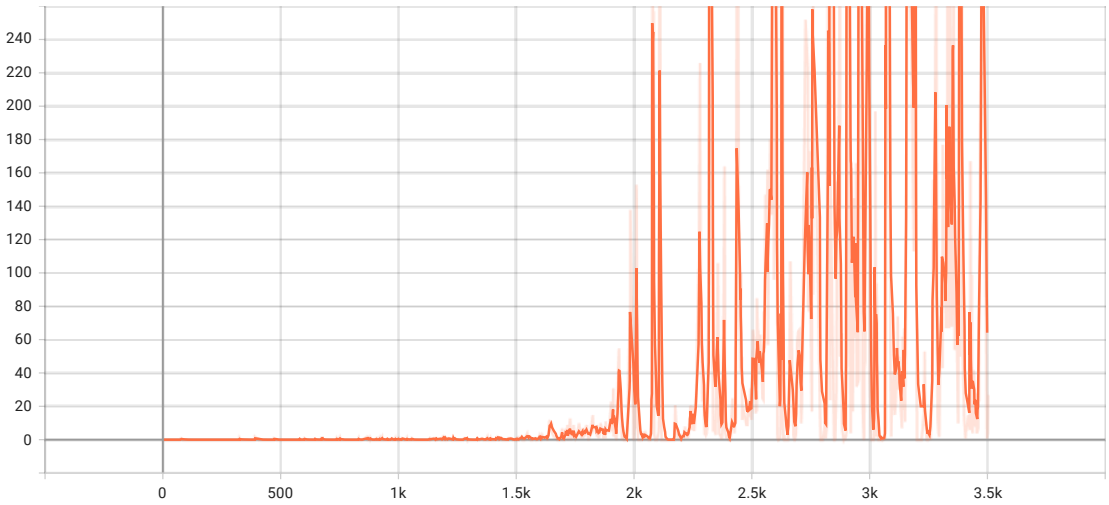


圖 4.7 DQN (20231012) Cleared_lines

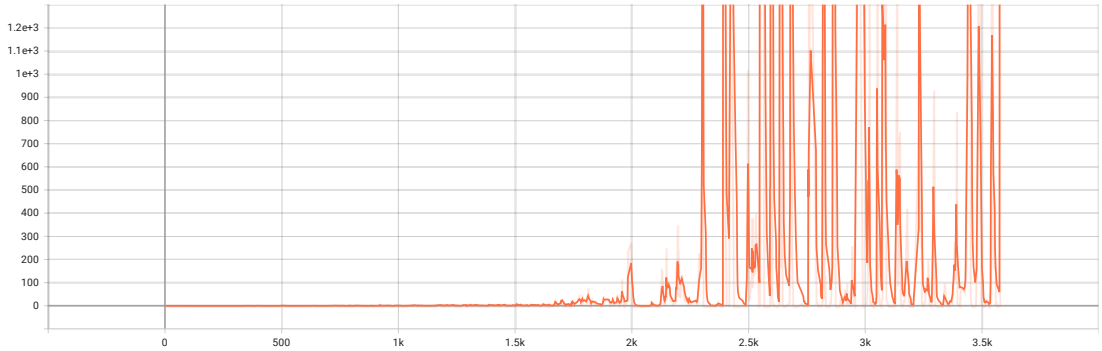


圖 4.8 Double-DQN(20231023) Cleared_lines

第 5 章 結論與未來發展

5.1 結論

根據本次實驗內容，我們成功讓一個 Agent 從剛開始不會填空格，到減少空洞但是無法消除方塊，再訓練成只會疊高其中一邊不注重消除，到最後成功使模型注重在消除行數，能夠以獲取最大分數為目標。

再來是 DQN 以及 Double DQN 的結果討論，在實作 DQN 時我們發現它的 loss value 會在執行 1600 次以後開始飆高，如圖 5.1 所示，隨著分數上升，loss value 也跟著上升，所以後來決定要更改它的結構，想辦法使得 loss value 下降，並且提升穩定度以及最高分數，根據論文[4]中的 DQN 以及 Double DQN 的比較，可以發現到 Double DQN 的 loss value 一定會小於等於 DQN loss value 的一半，這與實驗結果差距非常大，我們猜測因為 loss value 是由 Q-Target-Network 所預估的最大 Q 值，而實際 Q-network 取得的值太小，才導致他的 loss value 過大，這代表著它預估得太大，雖然使用的確有降下一些，但是整體來看還是差不多的。

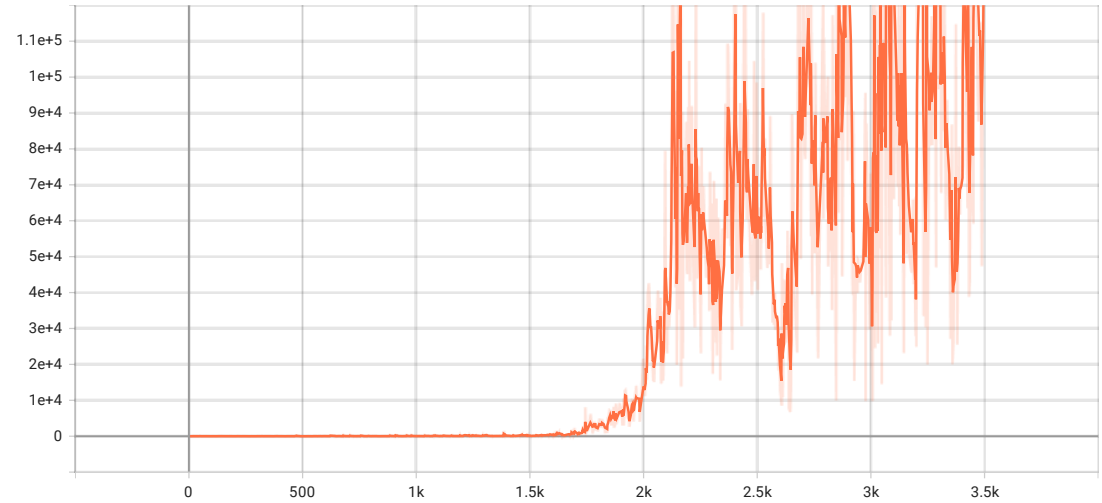


圖 5.1 DQN Loss

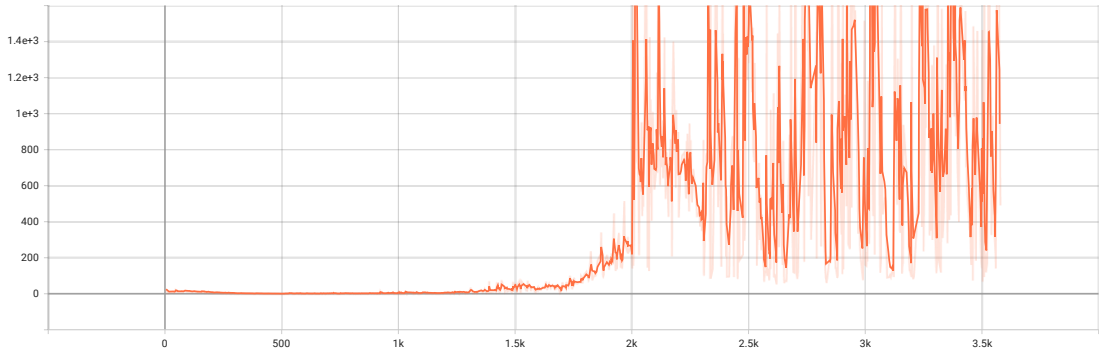


圖 5.2 Double DQN Loss

測試獎勵函數時，在相同的訓練次數下，嘗試更改獎勵公式，把原本放置一塊方塊就加一的獎勵改為零，用來對比原本的訓練方式，比較模型看看是否有差異，首先我們做出 DQN 原本跟更改後的比較圖，如圖 5.3，結果發現 DQN 模型在不計算放置數量時，表現較原本模型差，DQN 原本的平均值為 71580，更新後的平均值為 39406，模型分數差距快兩倍。

再來是 DDQN 原本跟更改後的比較圖，如圖 5.4， DDQN 原本的平均值為

130283，更新後的平均值為 122516，經由這次的對比我們發現到 DDQN 更新前後的差距並沒有 DQN 來的這麼誇張，由此可知獎勵函數的改變影響十分小，可以說是幾乎沒有。

在我們訓練時，因為模型一開始沒有消除行數的話獎勵為 0，所以他會維持一種動作，那就是持續的疊高，持續這個動作大概好幾百回合，直到他非常幸運的選到隨機動作，並且還要一直選到隨機動作，直到消除一行後，模型才會發現，原來消除行數可以獲得獎勵，模型才開始更改他的策略，直到我們看到整場有分數是在大約 2000 次訓練以後，這代表著模型前 2000 次全部都是白做。

最後我們使用這四個模型去做對比，如圖 5.5、圖 5.6，由此可發現到 DDQN 無論在更新前後都會比 DQN 的模型來的強大且穩定。

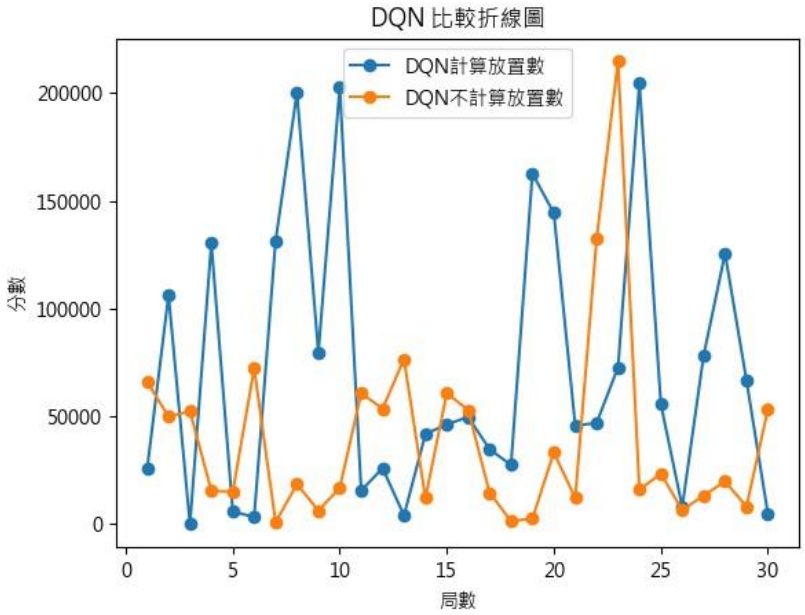


圖 5.3 DQN 獎勵比較折線圖

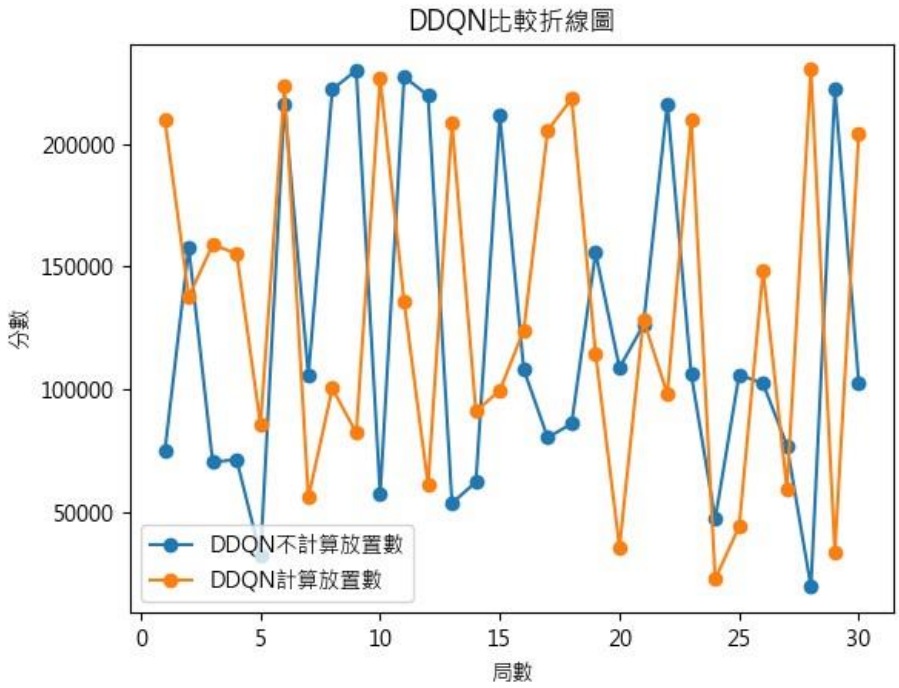


圖 5.4 DDQN 獎勵比較折線圖

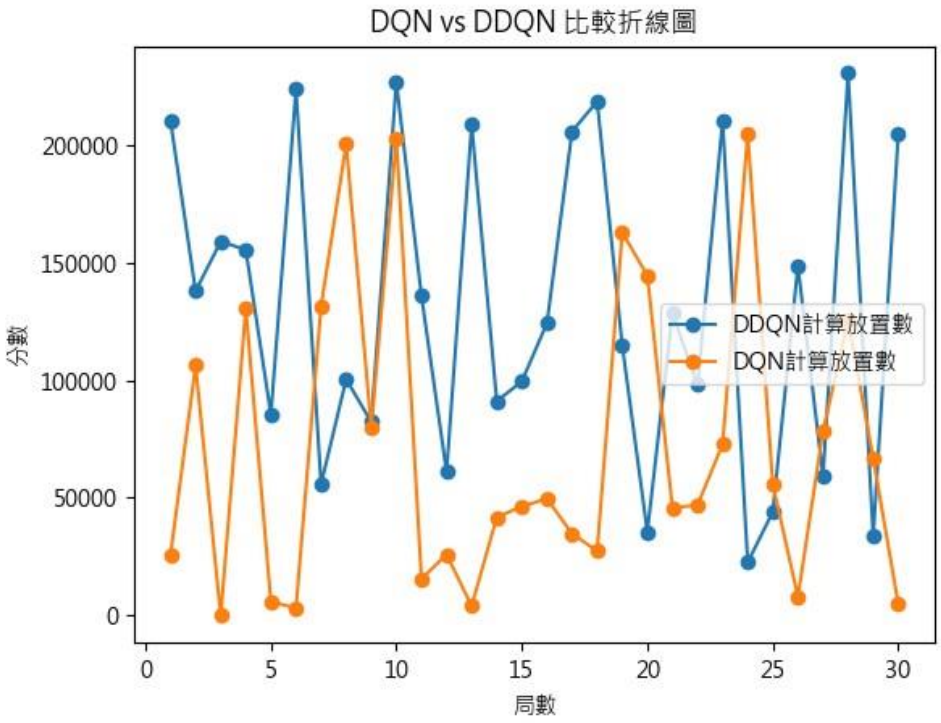


圖 5.5 DQN vs DDQN 計算放置比較折線圖

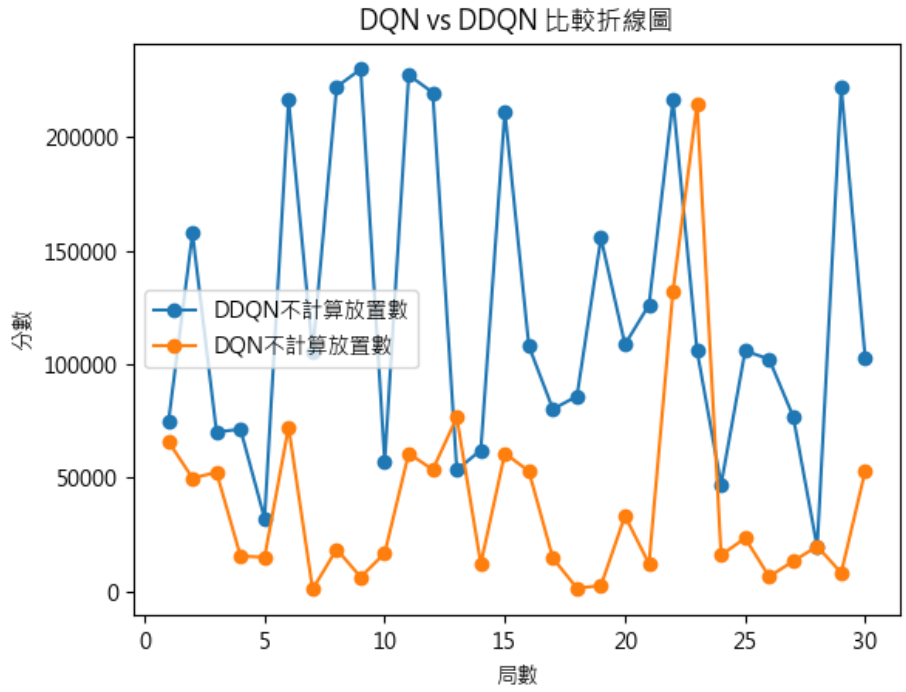


圖 5.6 DQN vs DDQN 不計算放置比較折線圖

5.2 未來發展

由於我們的 Agent 不是使用機器視覺的方法來獲得遊戲狀態，而是直接自己建立遊戲環境，獲得遊戲內部的參數，再把參數丟給 Agent 進行處理，因此無法套用在所有俄羅斯方塊遊戲中，如果未來能夠使用影像辨識的方法，辨識方塊、現有版面和空洞，或許可以套用在大部分的俄羅斯方塊中，增加使用方便性。

參考文獻

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra , Martin Riedmiller , “Playing Atari with Deep Reinforcement Learning”, arXiv preprint arXiv, 2013.

[2]Christopher J. C. H. Watkins, Peter Dayan, “Q-Learning ”,University College Londn,1992.

[3] Mohamed-Amine Chadi, Hajar Mousannif, “Understanding Reinforcement Learning Algorithms: The Progress from Basic Q-learning to Proximal Policy Optimization”, arXiv preprint arXiv,2023.

[4] Hado van Hasselt, Arthur Guez, David Silver, “Deep Reinforcement Learning with Double Q-learning”,arXiv preprint arXiv, 2015.

[5] Mohamed-Amine Chadi, Hajar Mousannif, “Playing Atari with Deep Reinforcement Learning”,arXiv preprint arXiv, 2013.

[6] Steve Bakos, Heidar Davoudi, “Mitigating Cowardice for Reinforcement Learning Agents in Combat Scenarios”, IEEE Conference on Games (CoG),2022.

附錄

DDQN train 程式碼

https://gist.github.com/Felixqaq/84f0b6cc103d11146ab8fdbcb4bb9416

DQN 及環境作者檔案

https://github.com/uvipen/Tetris-deep-Q-learning-pytorch