

# **Value-based Reinforcement Learning**

**劉宇舜 Yu-Shun Liu**

Student ID: 413551030

Department of Computer Science and Information Engineering  
National Yang Ming Chiao Tung University

Course: Deep Learning

April 30, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Implementation</b>	<b>3</b>
2.1	How do you modify DQN to Double DQN? . . . . .	3
2.2	How do you obtain the Bellman error for DQN? . . . . .	3
2.3	How do you implement the memory buffer for PER? . . . . .	4
2.4	How do you modify the 1-step return to multi-step return? . . . . .	4
2.5	Explain how you use Weight & Bias to track the model performance. . . . .	4
<b>3</b>	<b>Analysis and Discussion</b>	<b>6</b>
3.1	Task1 . . . . .	6
3.2	Task2 . . . . .	6
3.3	Task3 . . . . .	8
3.4	Ablation Study . . . . .	8

# 1 Introduction

我覺得最重要的發現就是不要閉門造車，我在先花費大量的時間、算力與資源之後，發現我的 Reward 都在 10 附近徘徊，大約在 400k 左右的 env steps 之後，就再也沒有進展了，即使我認真的去看 video，發現模型可以做出很帥的切球，但是有時候就是這些切球只要剛好讓對方可以打回來，那這球速就快到很難接到，所以可以穩定取得整場遊戲的勝利，但是沒有辦法穩定不失分，也就到不了最終作業要求的 19 分。

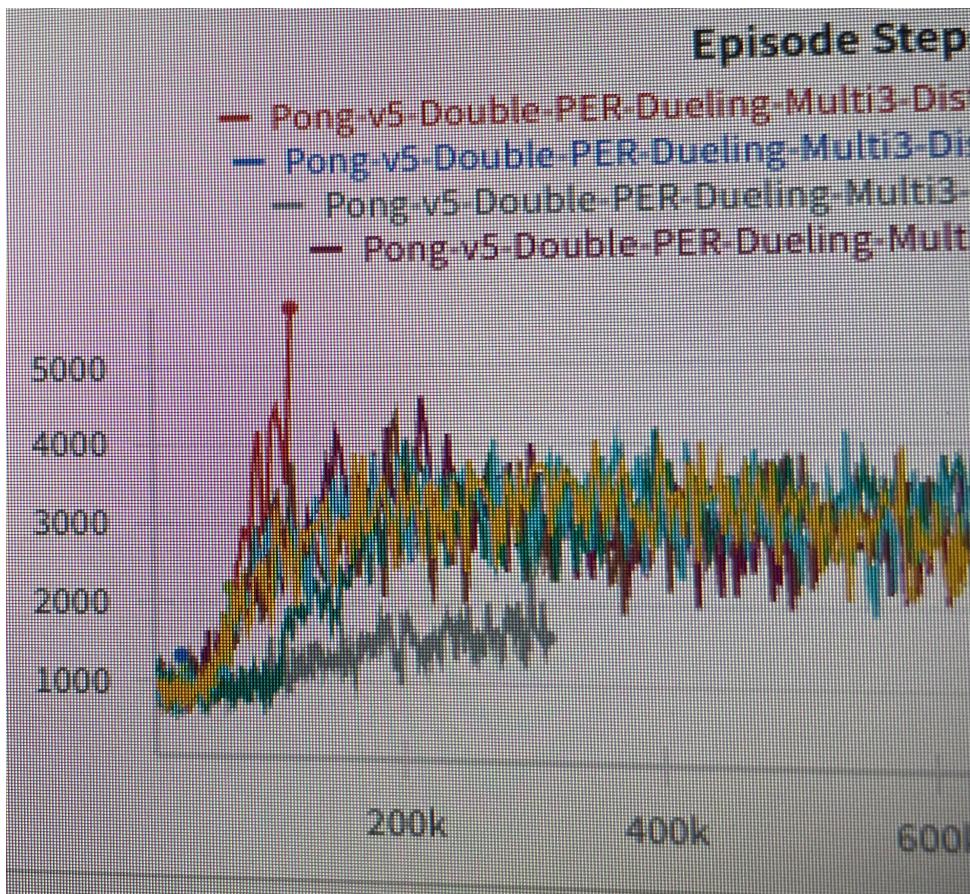


Figure 1: 訓練過程中 Episode Steps 的變化。

在查閱了相關論文之後發現有一種方法 RainbowDQN 可以在 400k 左右的 env steps 之後，達到 19 分，我覺得我當初按照作業要求的實作與 RainbowDQN 的做法主要的差異在於 NoisyLayer 的實作，因為我之前的實作可能由於先學會了切球，然後在快速地獲得 Reward 之後，就卡在這個次優策略中，然而當我實作了 NoisyLayer 之後，發現雖然 Reward 收斂的變慢了，但是 Episode Steps 有顯著的拉高，中的紅色線。

這代表模型學會了穩定的來回對打，我認為這可能會是一個突破次優策略的關鍵。

## 2 Implementation

傳統的 Q Learning 會使用一個 Q table 來記錄每個 state - action pair 的預期回報，但這會遇到一個問題，就是狀態空間過大，如果再乘上可以選擇的 action 數量，這個 table 可能會大到根本無法建立。

所以 DQN 的設計是使用一個神經網路來估計 Q value，這樣一來就可以避免狀態空間過大的問題。

### 2.1 How do you modify DQN to Double DQN?

原始的 DQN 會遇到一個問題就是 Q value 高估的問題，這是因為在 Q 值得估計中，會有模型的 noise 混入，如果每次都選擇最大的 Q value 會有遇到每次都選擇 noise 的最大值，這並不能幫助我們學習，所以 Double DQN 的設計是使用兩個 Q network，一個來選擇 action，另一個來估計 Q value，這樣即使我們在選擇 action 的時候選擇了因為 noise 的存在，而被高估的動作，但是由於計算最終的 Q value 時，我們會使用另一個 Q network 來計算，所以我們有理由相信，同時被高估的機率很小。

考慮到 DQN 的 Loss 設計是希望，當前預計的未來 Reward，與先採取一步驟再估計未來這一步驟採取之後的 Reward 要一致，所以我們把估計下一步的 Q value 使用一個 target network 來估計，這樣一來，就可以透過一直對未來的估計，來專注在提高當前選擇的 Reward。

目標是讓這左右兩式一致：

$$r + \gamma \max_{a'} Q(s', a'; \bar{\theta}) \approx Q(s, a; \theta)$$

如果過於頻繁的更新 network 的參數，會有 Online network 沒辦法好好收斂，建立策略。

### 2.2 How do you obtain the Bellman error for DQN?

下一個問題就是，在每次更新的時候，我們可以直接使用過去的 N 次狀態轉移來更新網路，也可以從記憶的資料庫中，隨機抽取 N 次狀態轉移來更新網路，但是直觀上，我們能夠理解有些記憶會特別有價值，如果是很常見的記憶，則沒有辦法帶給模型更多的知識。

所以 PER 的設計是，我們會使用一個優先權來衡量每個記憶的價值，這樣一來，我們就可以更關注在那些更少見，但是更有價值的記憶上。那們如何衡量記憶的價值呢？就是透過「令人驚訝」的程度來代表「有訊息量」，然後就優先抽取這些令人驚訝的數據進行學習，這裡就導入 TD error 的概念，因為 TD error 越大的話，就代表我們對於未來的估計越不準確，所以這些數據就更有價值。

這裡使用當前的估計，叫做 current estimate，與多做一步之後的估計，叫做 target estimate，這兩個估計的差值，就是 TD error (Temporal Difference error)。

$$\delta = [r + \gamma \max_{a'} Q(s', a'; \bar{\theta})] - [Q(s, a; \theta)]$$

Bellman error 是 TD error 的絕對值，所以可以寫成：

$$p_i = |\delta_i| + \epsilon$$

使用這個 Bellman error 來衡量記憶的價值，我們就可以更關注在那些更少見，但是更有價值的記憶上。

### 2.3 How do you implement the memory buffer for PER?

透過 np 提供的抽樣函數，我們可以給定不同的機率數值，來達到關注那些更少見、更令人驚訝的記憶。透過 Bellman error 我們可以衡量記憶的價值，接下來就要把這個價值傳遞給抽樣函數，這時候會有  $\alpha$  來退火，調整機率的敏感度，之後在用  $\beta$  來平衡探索與利用。

```
indices = np.random.choice(len(self.buffer), batch_size, p=probs, replace=True)
```

Code 1: 使用 np.random.choice 來抽樣記憶。

### 2.4 How do you modify the 1-step return to multi-step return?

考慮 TD error 是代表，當前的估計 Reward 與多做一步之後的估計 Reward 的差值，所以如果我們多確定一些步驟，然後在計算 TD error 的時候，可以讓 loss 更準確的指導 model 更新的方向。這也就是 multi-step return 的概念。

以 4 step 為例，我們可以在現在時間估計未來的 Reward，然後過 4 個 sample 之後，有了 4 個真實的 Reward 之後，這時候我們就可以計算真實的 Reward 加對未來的估計，之後跟 4 個 sample 之前的估計做比較與指導模型更新。

### 2.5 Explain how you use Weight & Bias to track the model performance.

由於這個專案的訓練過程中，我們會需要嘗試各種各樣的參數，所以會使用 Weight and Bias 來管理多台電腦的數據，尤其是比較不同數的曲線變化時特別好用。

一開始由於 Reward 一直趴在 -21，所以我很擔心模型的 loss 並沒有傳遞進去模型中，去優化模型，所以我繪製了 Avg\_Weight\_Norm 的曲線，可以看到在訓練過程中，權重有持續變大，讓我確認到沒有 Loss 傳遞的 bug，令我意外的是，我可以透過比較 Avg\_Weight\_Norm 的曲線，來比較不同模型的訓練效果，這對於我們這種需要嘗試很多不同參數的狀況下，是非常有幫助的。

在開頭提到的 NoisyLayer 的設計，可以從這個圖觀察到他的模型有更快的成長速度，雖然這並不代表 Reward 指標，但透過 Weight & Bias 的協助，我們可以很快的得到這種需要比較的理解。

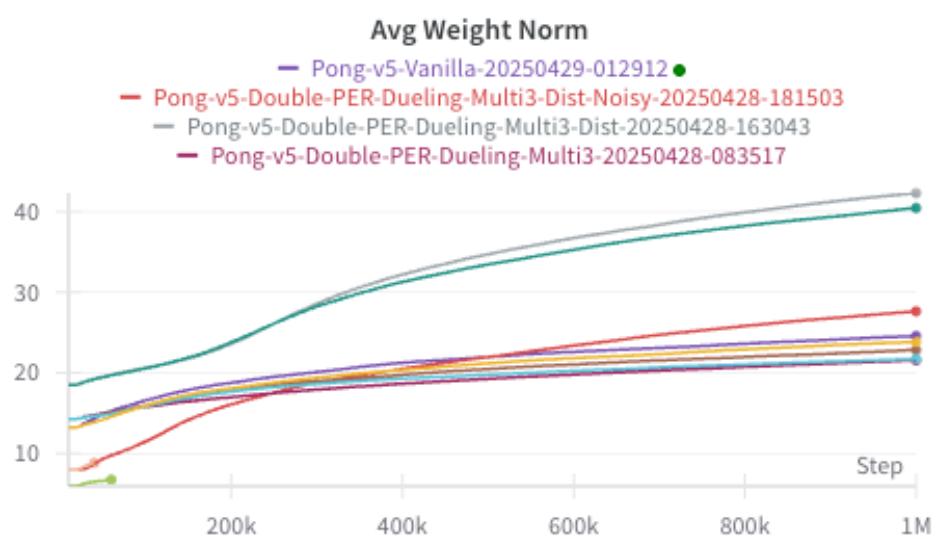


Figure 2: 訓練過程中權重範數的平均值變化。

### 3 Analysis and Discussion

#### 3.1 Task1

首先我在這次的作業中理解到 batch size 不是越大越好。不論是平衡車的任務還是乒乓球的任務，當我的 batch size 開到 1024 的時候，模型都學不動了。



Figure 3: 不同 batch size 下的評估獎勵曲線。

從圖 3 中可以看到，當 batch size 為 1024 時，模型的學習效果明顯幾乎為 0。這可能是因為過大的 batch size 導致梯度更新過於平滑，使得模型難以跳出局部最優解。相比之下，較小的 batch size (如 256 或 512) 能夠提供更好的探索能力，使模型能夠更快地找到更好的策略。

```
put image divisible by the macro_block_size or set the macro_block_size
Saved episode 18 with total reward 500.0 -> ./eval_videos/eval_ep18.mp4
IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size
to (608, 400) to ensure video compatibility with most codecs and players.
put image divisible by the macro_block_size or set the macro_block_size
Saved episode 19 with total reward 500.0 -> ./eval_videos/eval_ep19.mp4
Average reward over 20 episodes: 500.00
```

Figure 4: Task1 中的 Average Reward

最後我使用 256 的 batch size、100k 的 memory size 與 5e-5 的 learning rate 來進行訓練，在 1560 個 episodes 的時候達到了 20 次都連續 500 分的上限，如圖 4。

#### 3.2 Task2

這裡我們要使用 vanilla DQN 來訓練 Pong 的遊戲，作業要我們使用視覺輸入當作網路的輸入，模擬人類真正在玩遊戲的過程，所以這裡我們需要使用 CNN 來處理視覺輸入。

模型的架構如下：

```

self.network = nn.Sequential(
    nn.Conv2d(4, 32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=3, stride=1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 7 * 7, 512),
    nn.ReLU(),
    nn.Linear(512, num_actions)
)

```

Code 2: 用於處理視覺輸入的 CNN 模型架構。

這個模型接收 4 個連續的灰度圖像作為輸入（模擬時間序列），通過三個卷積層提取特徵，然後將特徵展平並通過兩個全連接層輸出每個動作的 Q 值（或分佈），然後輸出是每個 action 的預期 Reward。

由於我在訓練的時候，在每 10k env steps 的時候會執行 5 次的 evaluation。



Figure 5: Task2 中評估獎勵的最大值。

大約在 3.26 M env steps 的時候，評估獎勵達到了 19.2 分，如圖 5。

但是當我嘗試使用 20 次去 evaluate 的時候，發現模型會有幾次特別低分，這是其中一次執行的結果。

```

Average reward over 20 episodes: 14.55 +/- 9.18
Individual rewards: [17.0, -21.0, 19.0, 17.0, 19.0, 17.0, 18.0, 19.0, 18.0, 7.0, 17.0,
                    → 14.0, 17.0, 21.0, 21.0, 3.0, 16.0, 18.0, 19.0, 15.0]

```

Code 3: Task2 中 20 次評估的結果，high evaluation results。

如果細看影片，就會發現，兩邊會來回對打越來越快，但是我方的 agent 並沒有利用自己橫向位移的速度優勢，而是直球對決，再加上我方的 agent 有切球的習慣，所以一個不行，就會讓球漏過去。

所以我在檢索其他次高的 evaluation checkpoint 時，找到在 3.91 M env steps 的時候，平均 20 次可以達到 16.7 分。

```

Average reward over 20 episodes: 16.70 +/- 3.32
Individual rewards: [15.0, 16.0, 20.0, 19.0, 8.0, 17.0, 20.0, 18.0, 16.0, 15.0, 19.0,
→ 19.0, 10.0, 13.0, 20.0, 17.0, 19.0, 19.0, 14.0, 20.0]

```

Code 4: Task2 中 20 次評估的結果，second high evaluation results。

這個結果就比較穩定，不會有特別低的分數。

### 3.3 Task3

接下來我們就採用 Rainbow DQN 的方式來訓練模型，因為這是當前最有效率探索策略的方法。

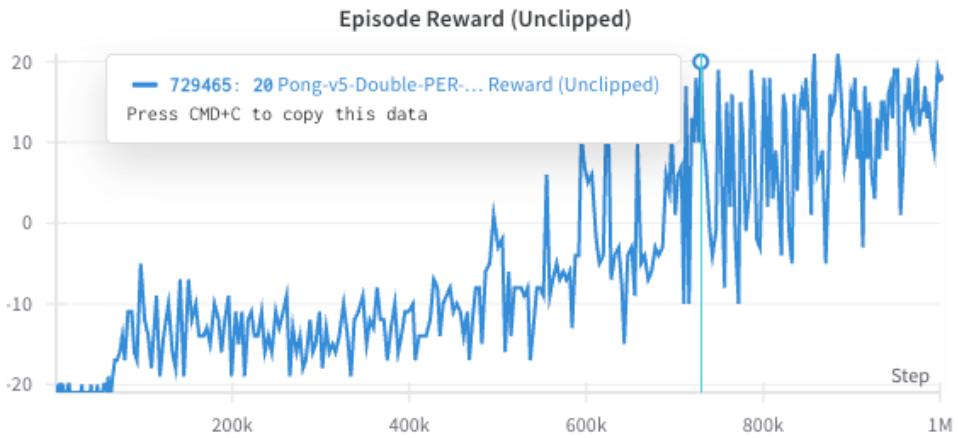


Figure 6: Task3 中最快 over 19 分的時刻。

從圖 6 可以看出，Rainbow DQN 在我實作的版本中，大約需要 730k env steps 就可以達到 19 分。

### 3.4 Ablation Study

原本的要求是使用 Double DQN、PER 以及 Multi-step Learning 來做優化，在 Rainbow DQN 中，多了 Distributional RL 以及 Noisy Networks。

所以我嘗試把這兩個技術拿掉，看看效果如何。

從圖 7 可以看出，首先是紫色線，是兩個技術都沒有使用，可以很快的收斂，但是我猜是次優解，所以還會越訓練越差，當我們加入 Distributional RL 之後，雖然慢，但模型可以穩定的提升，其中的振幅是這三條線中最小的，但很可惜他到了 15 分左右就上不去了，最後當我們加入 Noisy Networks 之後，模型可以穩定提升且灰色線快，這也是完整版本的 Rainbow DQN 的結果。

Note: 令我疑惑的是，在原始論文的實作中，可以在 400 k env steps 就達到 19 分，但是在我實作的版本中，卻需要 730 k env steps 才能達到 19 分，這是為什麼呢？

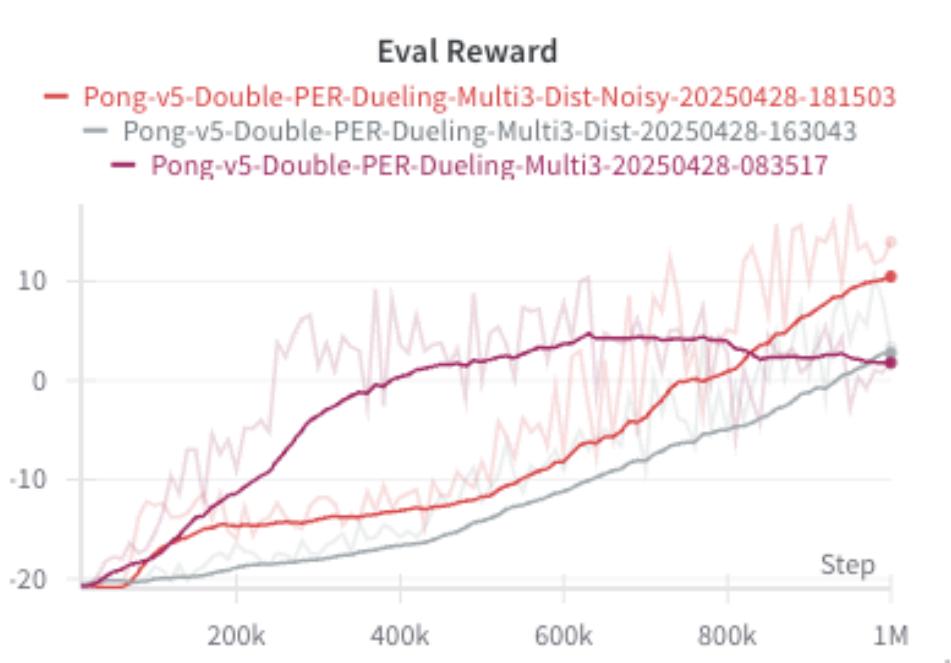


Figure 7: Task3 中不同技術組合的比較。(smooth lines)