國立成功大學
資訊工程學系

深度學習

期末專題報告

Dream a Fighting Game with Attention

組員：P76094169 黃仁鴻

授課老師： 陳奇業 教授

中華民國 110 年 1 月

目錄

表目錄

圖目錄

一、簡介

人類具有優秀的推理及預測能力，即使將過去曾經遊玩過的遊戲於腦海中重建亦非難事，甚至於可以於夢中再現遊玩的情境。

那如果用類神經網路來重現這個能力，就代表著我們可以此方式，在不對程式進行的修改的情況做到跨平台執行，進而降低許多維護成本。

為此，本次實驗將會使用類似 World Model [1]的結構，嘗試將無隨機系統的自製小型格鬥遊戲複製出來，讓其並於網頁上執行。

二、問題描述

本次研究將會利用類神經網路重現出實驗所給定的格鬥遊戲，在完成此目標的研究過程中會遇到以下兩個主要問題：

1. 如何讓模型依據過去的狀態與玩家的輸入生成出下一幀的狀態。

2. 模型將實時生成遊戲畫面，為了保證達成此目的需要控制模型的計算量。

三、研究方法

本次實驗分成由 Auto Encoder 構成的 Renderer Module 與 Transformer [2]構成的 Driver Module 兩個模塊。如圖 1 所示，Driver Module 會利用自回歸的方式不斷生成出下一幀的隱藏狀態，並交由 Renderer Module 將其轉變成遊戲畫面。

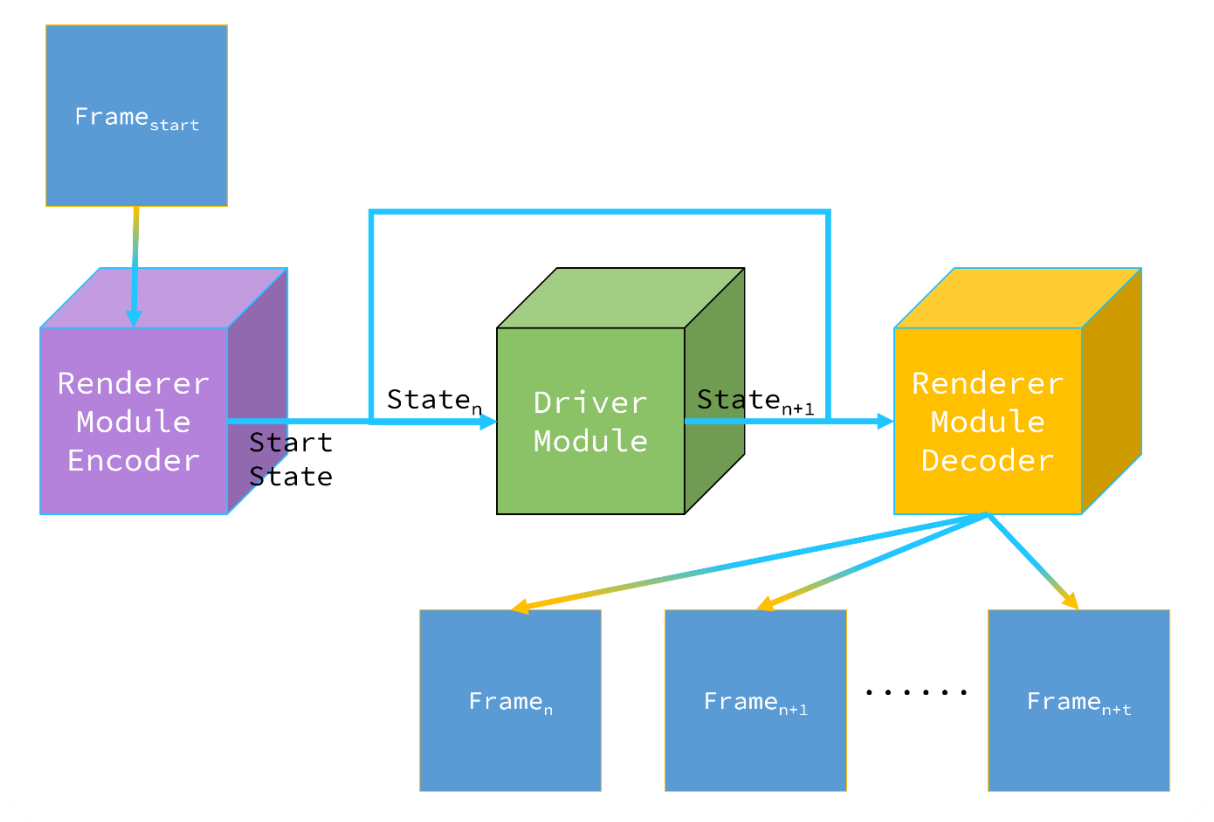

圖 1 各模組間的運作方式

3.1 Renderer Module

如圖 2，Renderer Model 分為把遊戲畫面壓縮成狀態編碼的 Encoder，以及將狀態編碼轉回遊戲畫面的 Decoder。在研究過程中發現如果將狀態編碼量化，在後續 Driver Module 生成時，具有比連續的狀態編碼更加穩定的結果。

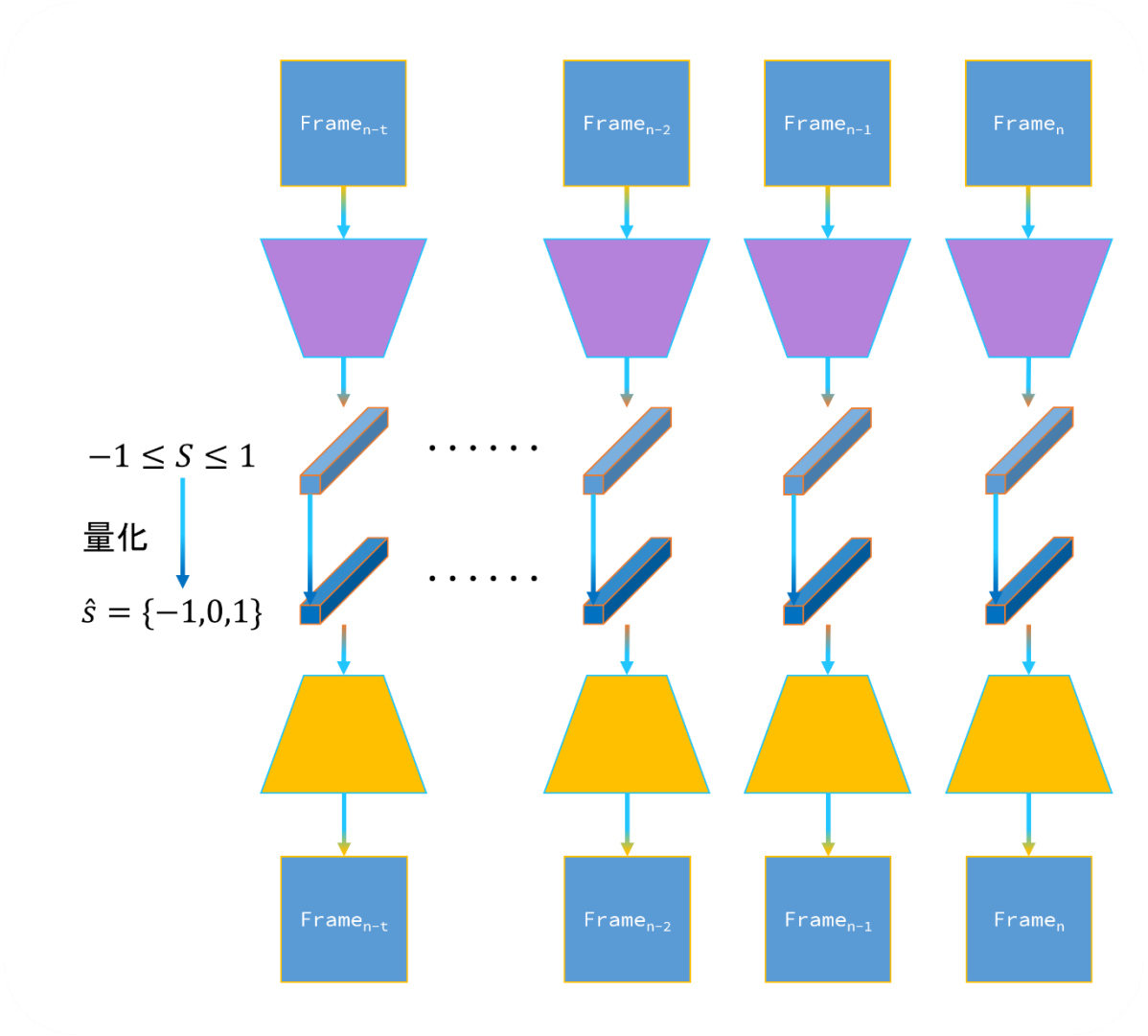

圖 2　Renderer Module

這邊所使用的量化方式是先對 Encoder 的輸出執行 tanh，使其值域介於-1~1 之間，再將其四捨五入得到{-1,0,1}的量化結果，但這種方式無法直接使用反傳遞演算法來優化模型。

為了對應此問題，使用圖 3 的方法將 Encoder 與 Decoder 的梯度分開計算。Encoder 會將連續的 State Embedding 傳給 Decoder 生成輸出畫面，但只會對 Encoder 執行優化。而 Decoder 則是使用量化後的 State Embedding 作為輸入並執行優化。

另外，如果直接使用 MSE 計算連續的 State Embedding 與量化的 State Embedding 的 Loss，或是在 Decoder 計算 Loss 時將連續的 State Embedding 與量化的 State Embedding 一起使用，都會使 Renderer Module 嚴重毀損。

為了增加 Renderer Module 的容錯能力與編碼能力，除了使用到量化的方法外，還會隨機把 State Embedding 部分的值以{-1,0,1}替換掉。

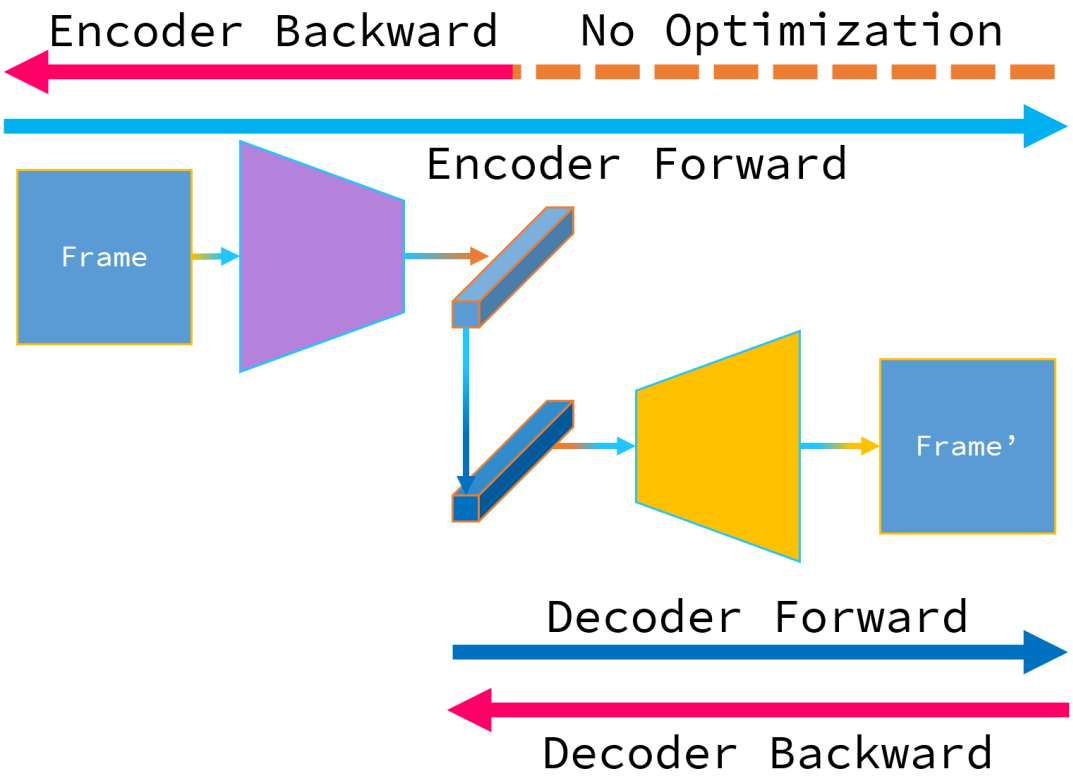

圖 3 因應量化問題的分區優化

3.2 Driver Module

Driver Module 的目標是利用過往的狀態與使用者的操作，生成出下一幀的 State Embedding (如圖 4)，因此在訓練過程中要先依靠 Renderer Module 的 Decoder 將圖片轉為 State Embedding，並由訓練資料中紀錄的使用者操作提取對應的 Act Embedding。而作為主結構的 Transformer Decoder 本身不具備分辨位置的能力，因此還需另外加上式(1)的 Position Encoding。

$$PE(pos, 2i) = sin(pos/10000^{2i/d_{model}}) \quad (1)$$

$$PE(pos, 2i+1) = cos(pos/10000^{2i/d_{model}})$$

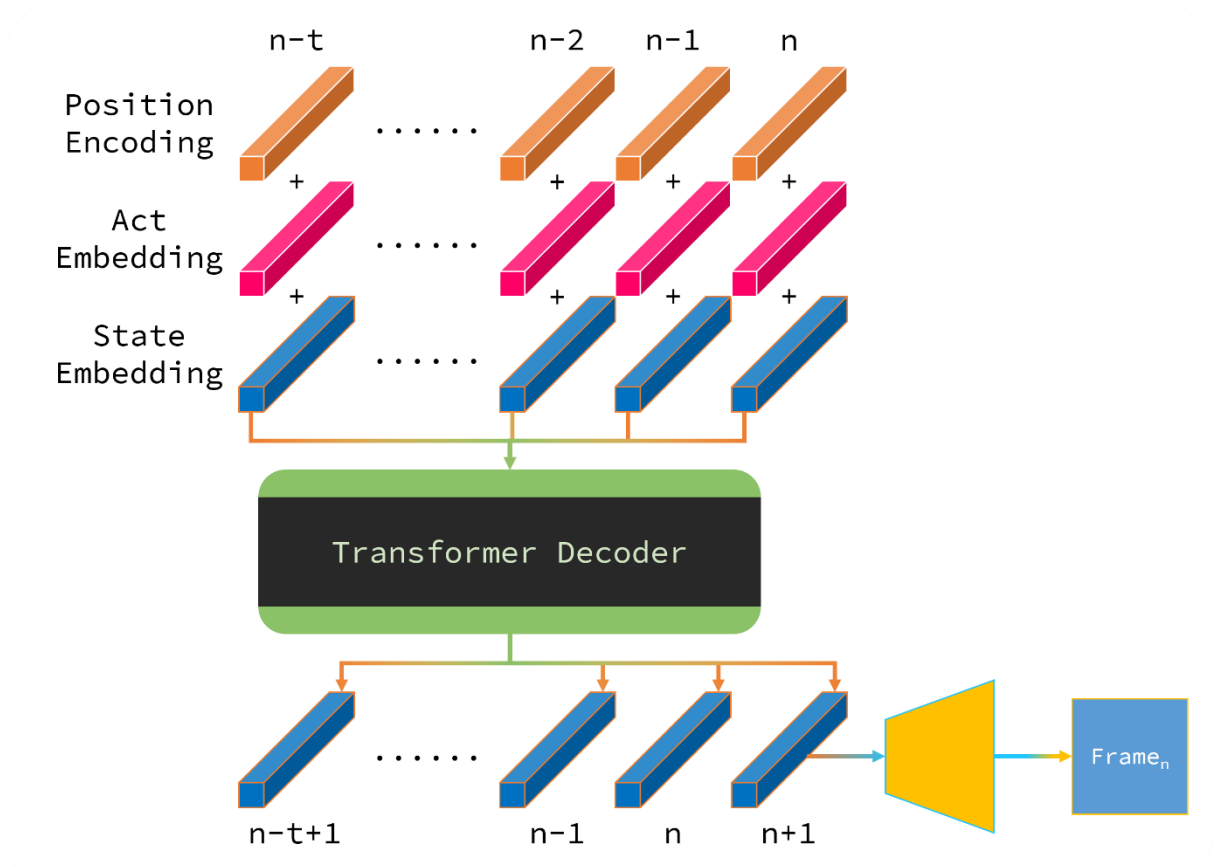

圖 4 使用 Transformer Decoder 作為 Driver Module

四、模型架構

在實做中為了降低模型的計算量，會如圖 5 所示先將 1080 x 1080 的原始畫面所小至 32 x 64，在經過 Encoder 後會壓縮成 512 維的 State Embedding，最後將量化過的 State Embedding 交由 Decoder 還原成影像。

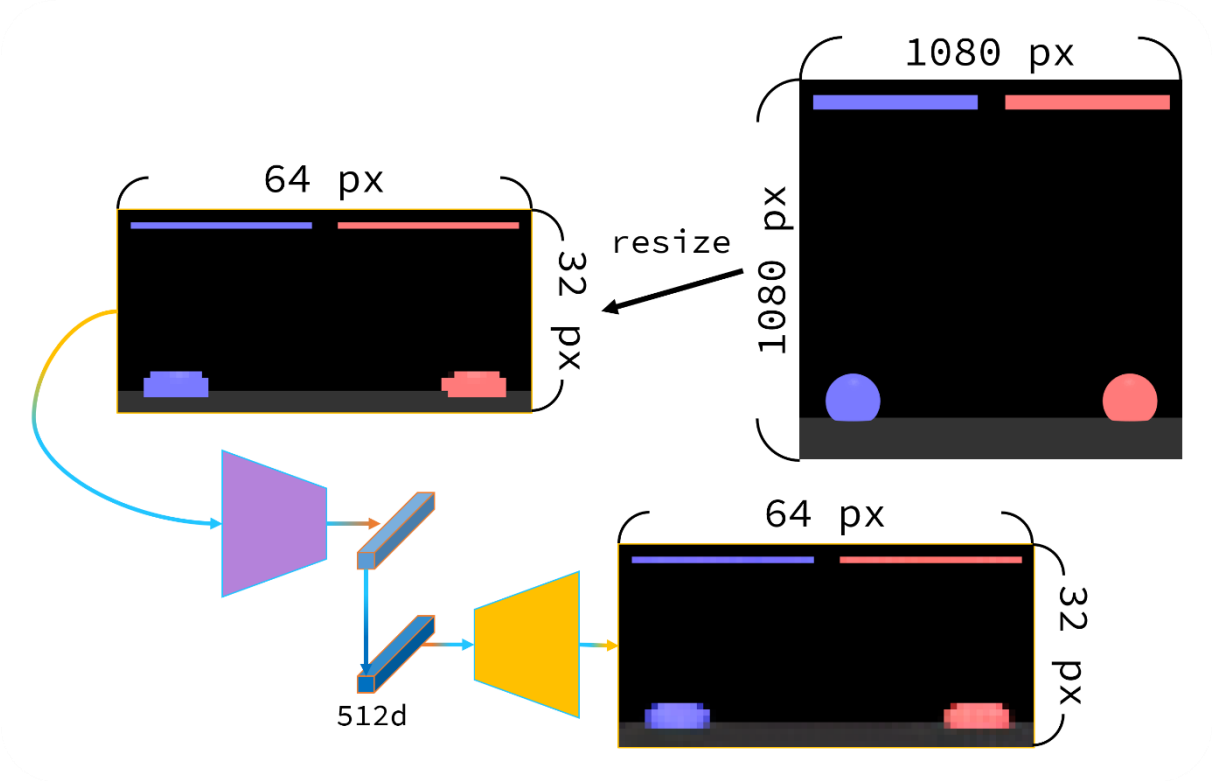

圖 5 先將原圖縮小以降低計算量

如圖 6 與圖 7 中，Encoder 與 Decoder 會分區成 Content 與 Position 兩區塊，這樣的作法比直接編碼成 State Embedding 的做法收斂更快且效果更好。而且在 Decoder 使用 Bilinear Resize 擴展 Content 的大小，可以減少 Decoder 所需要的計算量，以加快實際應用時的運行速度。

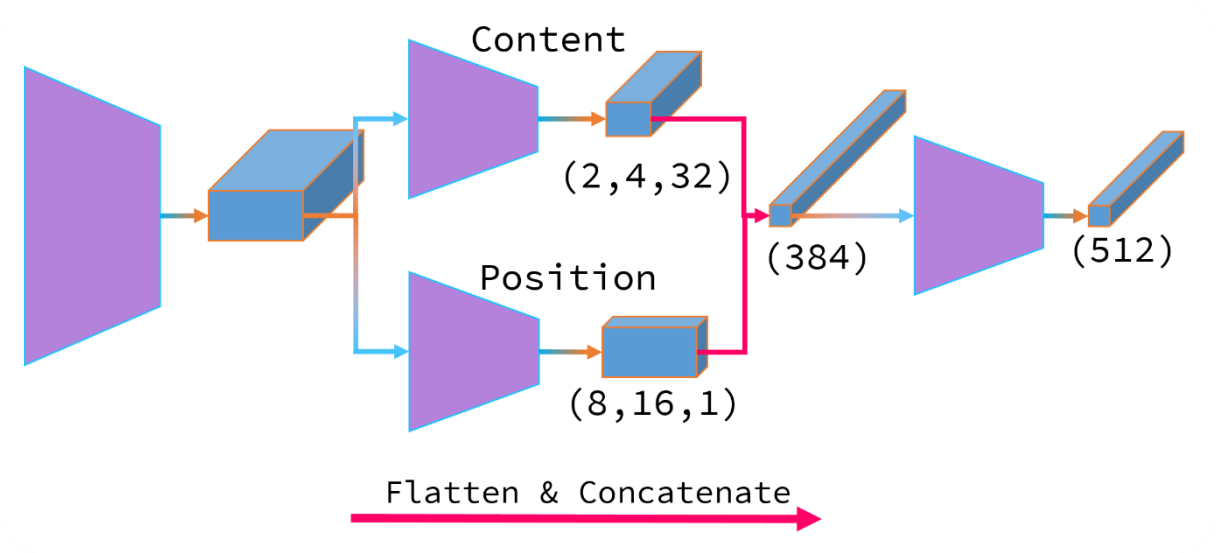

圖 6　Encoder 分為 Content 與 Position

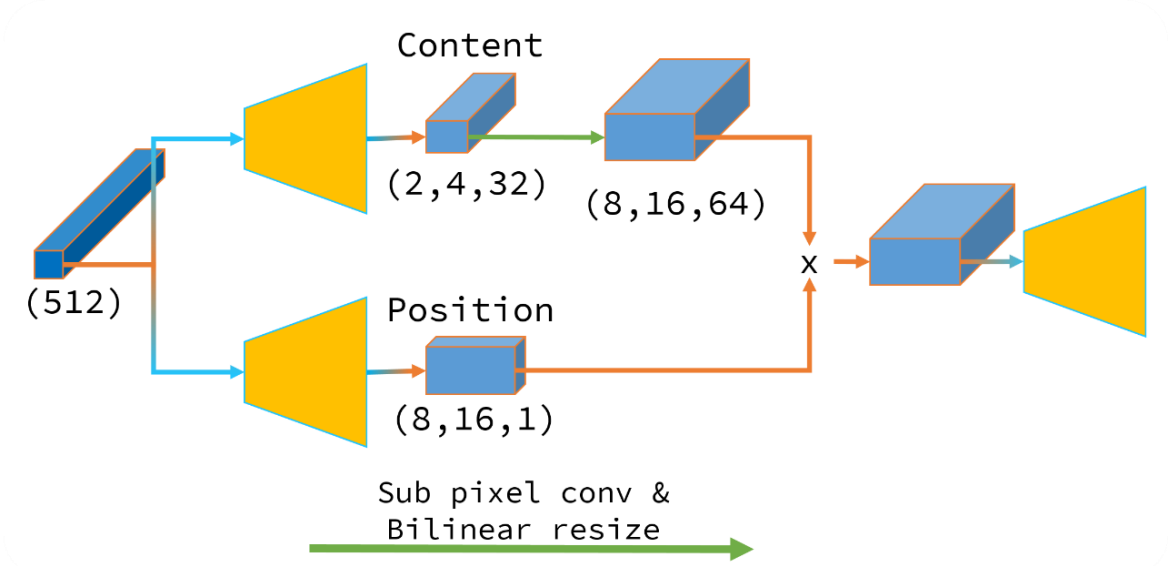

圖 7　Decoder

Driver Modulea 中的 Transformer 由兩層 Transformer Block 所組成，每層 Transformer Block 如圖 8 所示，包含 8 個 head 的 Multi Haead Attention 與 2 層全連接構成的 Feed Forward，之間還使用了 ResNet 的 shortcut 結構。

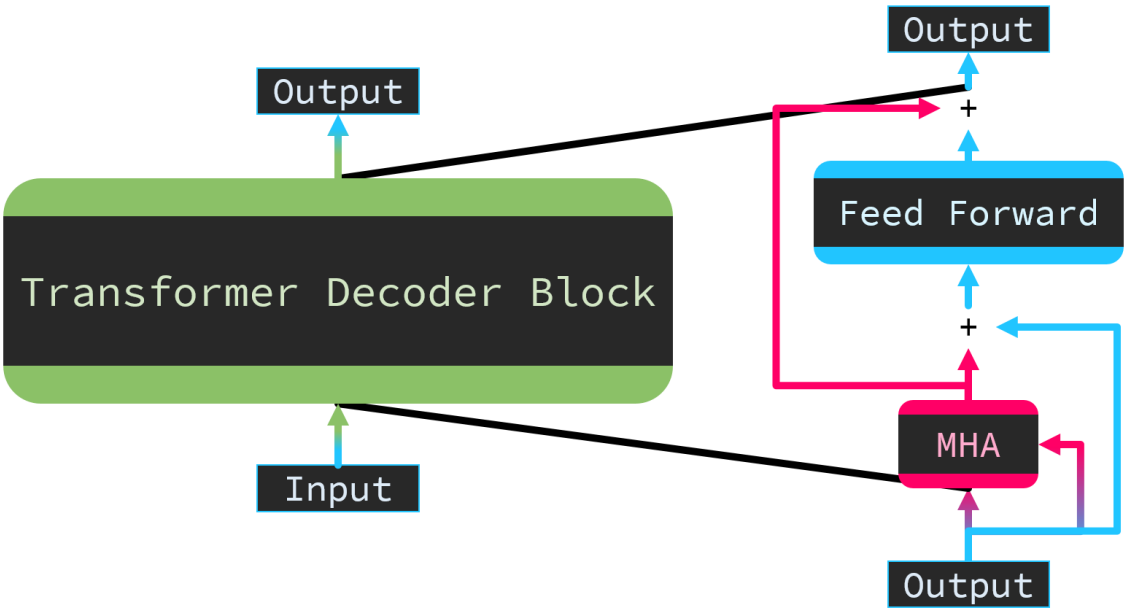

圖 8　Transformer Block

五、實驗成果與討論

5.1 實驗環境

表 1 環境配置

處理器	Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
記憶體	16.00GB
圖形處理器	NVIDIA Geforce GTX 1050 Ti
作業系統	Windows 10 64 位元作業系統、Linux Mint 64 位元作業系統
開發環境	Chrome 87 版以上、Node.js
使用語言	TypeScript
函式庫	Tensorflow.js、Babylon.js

表 1 是本次實驗的環境配置。運用 TypeScript 跨平台的優勢使其能於各作業系統上快速建置開發環境，此外 Tensorflow.js 可利用 webGL API 調用 GPU 的平行運算能力加速計算速，並且不必像 Tensorflow.py 還需要另外安裝 CUDA。

5.2 成果與討論

在實驗過程中發現如果使用一般連續型 Auto Encoder 生成出的狀態編碼，會造成 Driver Module 的生成結果變得混亂且無法控制。改用量化型 Auto Encoder 後則大幅的改善了這個問題。

而整個模型是否能成功學習如何重現目標任務，則與訓練資料的穩定度有很大的關係，最開始實驗中嘗試使用隨機操作的遊戲紀錄做為訓練資料，但是每幀都進行隨機操作大大的干擾了 Driver Module 的預測能力，使其生成結果常常與輸入的指令無關。

六、結論

雖然目前經過大量調整後取得了比一開始較為穩定的結果，但是 Driver Module 的能力依舊有待加強。在最終的實驗裡只學會了跳躍、蹲下與不完整的攻擊。即便是只學習特定動作情況下，對於左右位移還有延遲性觸發的操作，其生成效果都不是很理想。

此外，可能是因為使用了量化 State Embedding 的緣故，雖然穩定卻使得模型缺乏自主泛化不同狀態的能力，可能還需要配合像是 DQN [3]等等強化學習的方法(如圖 9)來廣泛探索各類狀態，並使 State Embedding 更加貼近當下的狀態。

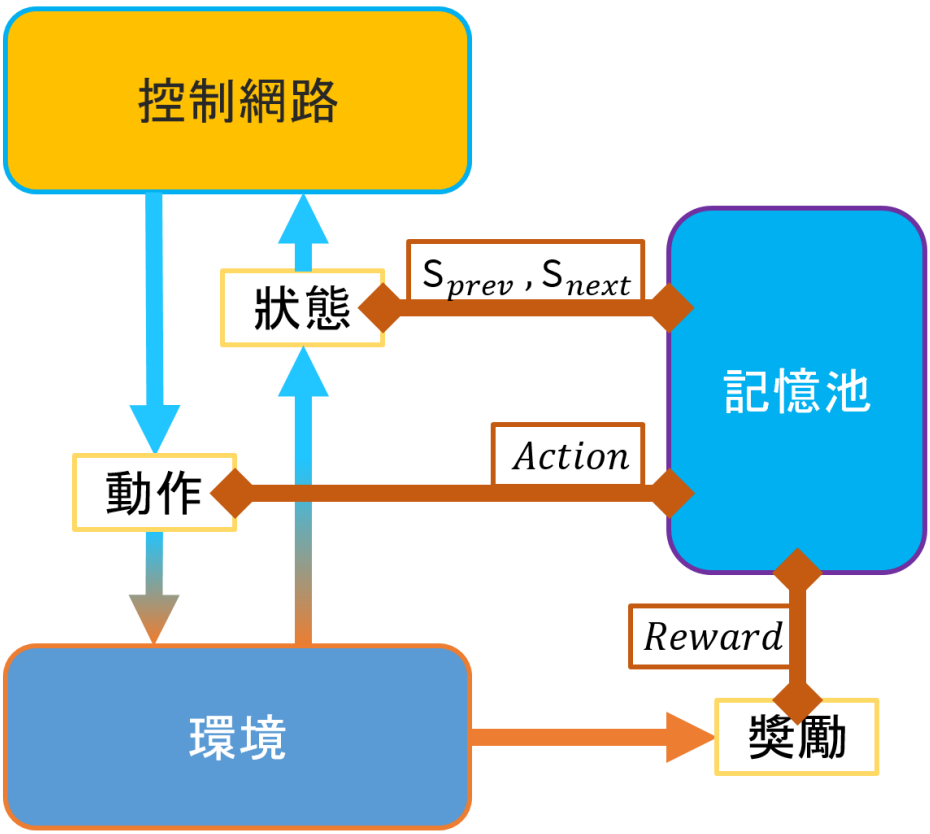

圖 9 強化學習架構

參考文獻

[1] David Ha, Jürgen Schmidhuber, “World Models,”
arXiv preprint arXiv:1803.10122, 2017.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, “Attention Is All You Need,”
arXiv preprint arXiv:1706.03762, 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, “Playing Atari with Deep Reinforcement Learning,”
arXiv preprint arXiv:1312.5602, 2013.