# Dino Run with Different DQN Models

107302002 金融四 張致中


## ABSTRACT

在 Deep Mind 推出基於 Reinforcement Learning 的機器人 Alpha Go 並擊敗韓國第一棋手後，Deep Reinforcement Learning 在學界內掀起一陣波瀾。有別於傳統的 Reinforcement Learning，Deep Reinforcement Learning 藉著使用類神經網路，能夠非常有效的處理更加複雜的環境；原本輸入的維度僅能是低維度，輸出也是低維度，且不能過於複雜，但是在使用了類神經網路後，就能夠做到由高維度映射至低維度的事情。

本次的實驗是要藉由 Reinforcement Learning 中的 DQN Algorithm 來讓電腦學會玩簡單的遊戲，這裡就以 Chrome 瀏覽器中內建的 Dino Run 作為本次實驗的環境。最終目標是要讓 Agent 學會如何去玩 Dino Run。Dino Run 是一款 Chrome 內建的小遊戲，會在無法連線上網時出現在瀏覽器上，主要的玩法為玩家操作小恐龍並使用跳或蹲來躲避障礙物，能夠存活越久就越好。此外，除了使用單純的 DQN，也會使用許多 DQN 的變型去比較不同模型之間的差異。


## AGENT

Reinforcement Learning 的 agent 大致上可分成三種: Actor model ,Critic model, Actor-critic model，其中 Critic model 所估計的是 agent 在狀態 S 下每次採取行動的分數。其估計方法大致上可分成三種:Dynamic Programming(DP)、Monte Carlo 以及 Temporal Difference(TD)。DP 的作法是計算出情況 S 採取動作 A 後的全部可能的下個情況 S'，計算出期望獎勵 R 後才去採取動作，但是這樣的問題在於時間複雜度過高，儘管後來有較為簡化的算法，但是該算法對於需要即時計算結果的環境仍然相當不友善；至於 Monte Carlo 是 Alpha Go 所使用的方法，在此就不贅述；最後，TD 比起 DP 的實時計算，TD 使用了**過去的經驗**來估計 Q 值，在計算上相當的快，只需記錄過去的經驗即可。

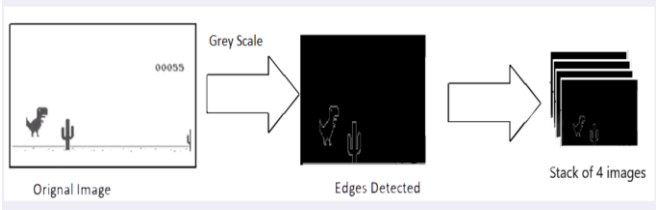
在所有 Critic model 中，最為經典的就屬 Q-Learning。傳統的 Q-Learning 是將狀態 S 和動作 A 編成對應的表，是為 Q 表。其中記載了在狀態 S 下採取動作 A 後所獲得的獎勵 R。除此之外，也會使用所謂的 greedy strategy 來讓 agent 嘗試相同狀態下未試過的動作，藉此讓 agent 有探索的機會，更容易找到更好的行動方式。當 Q 表上的數字沒有任何的改變，我們就可以說該張 Q 表已經收斂。但是 Q-Learning 有個相當明顯的缺點: **維度災難**。他的 state 和 action 是有限制的，無法處理過多以及高維度的 input 以及 output。舉例來說，整體的遊戲畫面就是一個 input，每

當只要有一個 bit 變動，就屬於不同的 input。因此遊戲畫面就是一個高維度的 environment。這讓傳統的 Q-Learning 難以在複雜的情況下有所發揮。

本次實驗所使用的 Agent 是以 Q-Learning 做為基礎而衍生出的 DQN。DQN 是將 Q 表的部分換成了類神經網路，藉此得以輸入高維度的環境，同時由於沒有 Q 表當作過去的經驗，所以還會額外使用 Replay memory 來儲存過去的經驗，在每次訓練時會從裡面隨機抽樣當作訓練資料。由於是隨機抽樣，該方法克服了訓練資料的相關性和非平穩分布的問題，而且過去資料的利用率也提高，有效節省記憶體空間。

## ENVIRONMENT

實驗的環境是利用了 selenium 來當作最基本的環境，並用其內建的函數來和瀏覽器互動並獲得遊戲資訊，再將其利用 Open AI 的 gym 來包裝，以利之後的處理。在環境的處理上也參考了[7]，使用了 opencv 抓取並 resize 成 80*160*3 的遊戲畫面，再將畫面灰化後將連續四幀畫面相疊，所以最終的 input shape 是為 (80*160*4)，讓 agent 能夠知道採取動作後小恐龍會如何移動。整體情況就如圖(一)所示。




圖(一):環境的處理方式

除此之外，我也把整體的遊戲速度設定在固定值，並不讓飛鳥出現，以簡化整體遊玩過程，避免環境過度複雜導致訓練困難。

## MODEL STRUCTURE

**Q table network structure:**

有關模型的基本架構，我使用了三層 Convolution layers，分別給予了 32、64、64 個 neurons，並且 filter size 設為 8、4、3，strides 則是 4、2、1。由於小恐龍本身並不需要特別詳細的特徵辨識，只需知道小恐龍和障礙物的大致位置，所以先用較大的 strides 和 filter size 來模糊 input，之後才用較小的參數來抓取小恐龍和障礙物的位置。同時也使用了 ReLU 當作 activation function，並用了 He 來初始化權重。在這裡我並沒有使用 Pooling layer，由於 Pooling layer 會將特徵給去除，但是對於遊戲來說畫面上的每個地方都相當重要，若隨意添加 Pooling layer，很可能會使重要資訊遺失，導致無法有效訓練模型。

在經過 Convolution layer 後則是會通過一層有 512 個 neurons 的 Fully-connected layer，最後則是 output layer，輸出了三個值，分別可以代表不動、跳，以及蹲等**動作的 Q 值**。

該模型的 loss function 我則是使用了 Huber loss function，這是因為相較於常用的 MSE，Huber 能夠在有效處理離群子的情況下也有快速的收斂速度。最後在 Optimizer 的選擇上使用了 RMSProp，這是因為該模型的 learning rate 我是設定遠小於 0.003，所以並沒有使用常見的 Adam。

這個模型裡我並沒有加入 Dropout layer，這是因為在我的認知裡這個模型代表的是 Q 表，若添加了 Dropout，就相當於在 Q 表上挖洞，這會對模型的訓練穩定程度造成很大的影響。

最後其實我原本有考慮在模型裡加入 RNN 或 LSTM 等能夠有效處理時間序列的 layers。但是在看過[5]之後我認為 Dino Run 這個遊戲的複雜程度並沒有到很高，小恐龍躲避單一障礙物這件事情並不會影響到未來小恐龍躲避其他障礙物的這件事，所以最後並沒有加入其他 layers。

**RL structure:**

模型的設置基本上都是使用 Q-Learning 的基本公式去做設定:

$$Q(s,\ a) = \left(1 - \frac{1}{n}\right) Q(s,a) + \frac{1}{n}[R(s,\ \pi(s), s_{t+1}) + \gamma \max_a Q(s_{t+1}, a)]$$

且在 next state 為停止遊戲時公式會變為:

$$Q(s,\ a) = \left(1 - \frac{1}{n}\right) Q(s,a) + \frac{1}{n}[R(s,\ \pi(s), s_{t+1})]$$

所以最後整個訓練過程就會如圖(二)所示:

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

圖(二):DQN 完整訓練過程，取自[8]

## MODELS

本次實驗使用了共四種不同的模型，分別為:Nature DQN、Dueling DQN、Double DQN 以及 CER DQN。

**Nature DQN:**

在最基本的 DQN 裡，我們使用了單一的 model 去同時做訓練 current-Q 以及預測 expected-Q 這兩件事。由於是一面訓練並一面預測，所要 fit 的 target 是一直變動的，因此模型會難以收斂，有種狗追自己尾巴的意味。所以 Nature DQN 使用了兩個相同的 model 分別去訓練和預測 Q 值。這兩個 model 雖然相同，但是他們更新的速度會不太一樣；相較於負責做訓練的 model，預測 Q 值的 model 會延遲更新權重，讓整個訓練過程較好收斂。其過程如圖(三)所示:

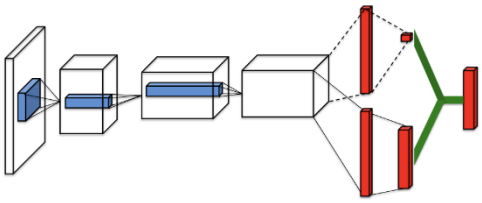
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**

圖(三):Nature DQN

在本實驗中該模型被用來當作 baseline，其他的 model 也會基於此模型，再做更進一步的修改。

**Dueling DQN:**

Dueling DQN 調整了 Q table network 的架構，在 output 的地方分成兩個網路:就我個人的認知，一個是估計當下狀態好壞的 value function，另一個則是估計採取動作好壞的 advantage function。由於無法直接從 Q 值解析出哪部份是 advantage output，哪部份是 value output，所以會將 advantage 減去平均值後加上 value。整體網路狀態如圖(四)所示:

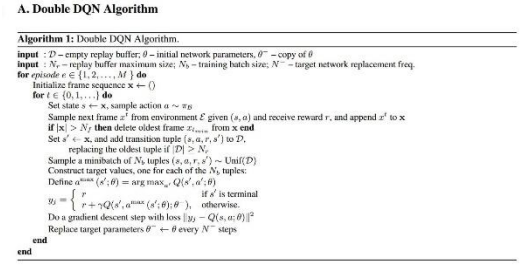


圖(四): Dueling DQN 網路架構

原本的 DQN 會同時評估狀態和動作，但是 Dueling DQN 就有了各自分工的意味，理論上能夠提升整體的結果。

**Double DQN:**

在 DQN 下 expected-Q 的預估是使用最大的 Q 值去做後續處理，但是這也造成了 Q 值高估的問題。由於 DQN 是用類神經網路去做逼近，這使得理論上有相同 Reward 的動作在 Q 值上會有極小的差異，由於公式是直接取最大的 Q 值，這使得 Q 值會被高估。所以在處理上一樣會使用兩個網路，先用主網路選擇動作，再用副網路計算 Q 值。這樣子即使主網路選到高估的動作，只要副網路沒有高估，計算出的一定是正常值；如果今天變成副網路高估 Q 值，那麼該動作就不會被主網路選取。詳情如圖(五)所示:



**A. Double DQN Algorithm**

**Algorithm 1:** Double DQN Algorithm.

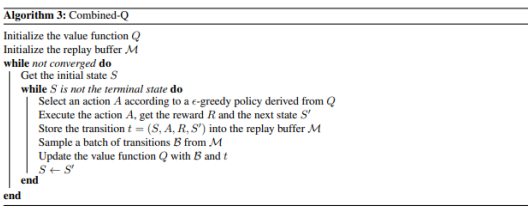
**input** : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
**input** : $N_\mathcal{T}$ – replay buffer maximum size; $N_\mathcal{B}$ – training batch size; $N^-$ – target network replacement freq.
**for** *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  **for** $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_\theta$
    Sample next frame $x'$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x'$ to $\mathbf{x}$
    **if** $|\mathbf{x}| > N_\mathcal{T}$ **then** delete oldest frame $x_{t-N_\mathcal{T}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple $\{s, a, r, s'\}$ to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| > N_\mathcal{T}$
    Sample a minibatch of $N_\mathcal{B}$ tuples $(s, a, r, s') \sim \mathrm{Unif}(\mathcal{D})$
    Construct target values, one for each of the $N_\mathcal{B}$ tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
  **end**
**end**

圖(五): Double DQN

在[3]中主網路和副網路是會交替的，但是在實作中會用一直在更新的網路來選擇動作，延遲更新的網路則是負責 Q 值的估計。

**CER DQN:**

比起網路結構和訓練方法的調整，和其他的 model 不同，CER DQN 是在 Replay memory 上做調整。CER DQN 的作法是將當下的狀態 S、採取的動作 A、獲得的 Reward，以及下個狀態 S'馬上放入 training batch 中訓練。會這樣做是因為當 Replay memory buffer 越大，從裡面抽樣到對於 agent 來說「重要時刻」的樣本數會越少，因此將獲得的觀察值直接拿去訓練，可以助於收斂的加速。詳情如圖(六)所示:



**Algorithm 3:** Combined-Q

Initialize the value function $Q$
Initialize the replay buffer $\mathcal{M}$
**while** *not converged* **do**
  Get the initial state $S$
  **while** $S$ *is not the terminal state* **do**
    Select an action $A$ according to a $\epsilon$-greedy policy derived from $Q$
    Execute the action $A$, get the reward $R$ and the next state $S'$
    Store the transition $t = (S, A, R, S')$ into the replay buffer $\mathcal{M}$
    Sample a batch of transitions $\mathcal{B}$ from $\mathcal{M}$
    Update the value function $Q$ with $\mathcal{B}$ and $t$
    $S \leftarrow S'$
  **end**
**end**

圖(六): CER DQN，取自[4]

很有趣的一點是除了 CER DQN 和[2]以外，DQN 鮮少有其他處理 Replay memory 的方法，但是對於

off-policy model 來說，取樣的品質與方法應該是會對整體表現有非常顯著的影響才對。那這點也在 Deep mind 推出 Ape-X 後得到證實；原本 rainbow DQN 普遍被認為是 DQN 中最好的 model，但是 Deep mind 所推出的 Ape-X 在收斂速度和成績表現上是勝過 rainbow DQN 一大截的，而 Ape-X 僅僅是使用非常大量的環境採取數據並用[2]進行處理，同時也發現整體的表現成績和環境的數量呈現正相關。

## EXPERIMENTS

**Hyperparameters:**

本次實驗的超參數如下:

| | |
|---|---|
| Memory size | 30000 |
| Min memory | 10000 |
| Batch size | 128 |
| Gamma | 0.95 |
| Epsilon | 0.1 |
| Min epsilon | 0.0001 |
| Learning rate | 0.00002 |
| Target update | 5 |
| Episodes | 5000 |

這裡 Epsilon 是表示 Greedy strategy 中隨機動作的機率，他會隨著訓練次數值漸降低至 0.0001。Learning rate 在此指的並不是 RL 中的 Learning rate，而是 Q 表網路中的 Learning rate。至於 Target update 指的是在每五個 episodes 後副網路才會更新權重。

**Algorithm Adjustment:**

在實驗初期，我都是使用 target-Q 去 minimize 與 current-Q 之間的 loss，但是這樣做還多了一個計算 target-Q 的 learning rate 要考慮；除此之外，我也將 terminal state 的處理依照原本公式的作法處理:expected-Q = Reward if state is Terminal；最後我還讓副模型每五個 iteration 就更新一次，這樣做就造成了許多問題:

**過多參數要調整**

由於一開始是 minimize target-Q 和 current-Q 的 loss，所以多了額外的 learning rate 要調整，使得參數調整速度緩慢。後來發現可以只 minimize current-Q 和 expected-Q 即可，就無須再調整 learning rate。

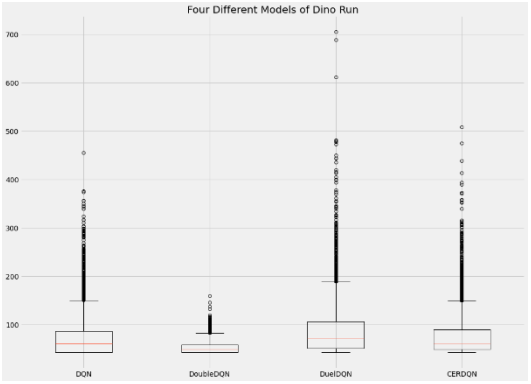
**未考慮遊戲本身狀況**

一開始我是有將 Terminal state 考慮進去。在其他遊戲中，Terminal state 意味著遊戲的結束，但是 Dino run 本身是一個永不結束的遊戲，一旦結束就代表發生了失誤，所以不應該把 Terminal state 放入考慮。於是乎我最後將 Terminal state 處理的部分移除，agent 的得分狀況馬上獲得大幅的提升。

除此之外，每五個 iteration 就更新模型對於 Dino run 來說太快了，這樣會使得模型無法收斂。就 Dino run 來說應該要考慮每幾千次 iteration 或每 n 個 episodes 才更新一次網路會比較好。

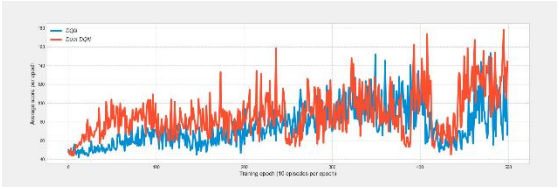
**實驗結果:**

直到我做了以上調整，整體的訓練進行才非常順利，否則分數都沒有有效提高。最後個個模型的表現就如下圖所示:




圖(七): 不同模型之間的訓練結果

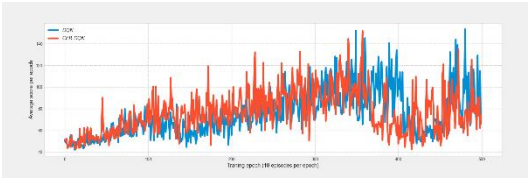
我們可以看到就各個模型而言，Dueling DQN 的表現最好，有高達 700 多分接近 800 分的表現，再來就是 DQN 與 CER DQN，兩者的表現差不多，最後則是 Double DQN，表現非常不好。



圖(八):Dueling DQN 分數取 log

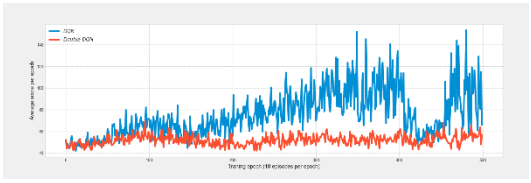
成績為十次平均

由上圖可以看出 Dueling DQN 的收斂速度明顯比 DQN 快上很多，得分也高出許多，這說明了將動作和狀態分開評估對於 Dino run 來說會有很好的結果。



圖(九):CER-DQN

至於 CER-DQN 整體表現和 DQN 差不多，但是可以發現在訓練前期 CER-DQN 學習較為快速，到了後期表現則是都差不多。



圖(十): Double DQN

雖然 Double DQN 能夠消除高估 Q 值的問題，但是在此貌似卻對 Dino run 沒辦法起到很好的效果。但是也許只是訓練得不夠久 Double DQN 還無法起到效果，像是[7]裡的 Double DQN 表現在最後才大幅超過其它的模型。

## CONCLUSIONS & FOUNDS

藉由這些實驗，可以發現對於單純躲避障礙物的遊戲而言，Dueling DQN 會起到很好的效果，因為他能夠有效判斷當下狀態的分數並做出適合的動作。至於其它的模型，並不能說沒有顯著的幫助，他們只是在 Dino run 這個環境裡沒辦法起到太好的效果。

除了單純模型的表現上，在訓練過程中也有發現其它很有趣的事情:

**最近距離起跳:**

第一個觀察到的情況是 agent 偏向在盡可能靠近障礙物時才選擇跳起。

**利用遊戲中的 glitch:**

Dino run 有一個很有趣的 glitch，當恐龍跳起來在空中的時候，若很快速的連續執行跳和蹲的動作，小恐龍就有辦法停留在空中，這件事情難以被人類玩家做到，因為往往會按著蹲下太久而讓小恐龍馬上墜地。

但是訓練出的 agent 卻能夠發現並利用這個 glitch 短暫滯空來躲避障礙物。

**Agent 表現差勁:**

這裡就可以分成兩個部份來講，一個是訓練後期的表現以及整體模型的表現。

由前面的圖可以發現大概在 4000 次左右之後的分數都相當的低。我覺得這可以歸納為副模型的更新是以 episode 作為單位而不是 iteration。玩到越後面一個 episode 的長度會越來越長，這使得模型更新的間隔其實是大幅下降的，所以這很有可能使得模型沒辦法很好的做更新導致表現變差。

除此之外，可以參考其他人做的 Dino run 模型，如這篇[1]和這篇[2]，在第一篇文章裡，他將 terminal state 的處理有考慮進去，但是倘若我一旦將其考慮進去，訓練的結果就變得十分糟糕，但是整體表現情況是差不多的。但是一旦和第二篇相比，我的結果就糟糕許多。他在跑到 2000 episode 時普通 DQN 的成績就已經超過 1000 分，Dueling DQN 則是高達了 5000 分。就連 Double DQN 的分數也有至少 1000 分左右。

我最後處理的方法和第二篇文章的處理方法大相逕庭，唯二的不同是我的 agent 的動作有三種，他的只有兩種；此外最重要的是我不管情況是否為 terminal，都用原本的公式去做更新，但是他的做法是只取 non-terminal state 的資料做訓練。我也有嘗試這樣做，但是整個結果又會因此壞掉。

## FUTURE

關於這份專案，我覺得還有很多可以嘗試的方向，以下是我未來會想繼續做下去的部分:

**Rainbow DQN:**

我原本是想要將 Rainbow DQN 給完整的呈現出來，但由於才剛接觸 Reinforcement learning，在一開始的時候花費了大量時間研究基礎，導致在後來處理其它 DQN 變型的時候就沒有太多的時間讓我好好詳讀每篇論文並研究每個模型，無奈時間有限，因此只呈現了相對簡單處理的模型。

**嘗試不同種類模型:**

現階段我只使用了 DQN 這種古老的 Critic model，我在未來應該會繼續嘗試不同的模型，像是 Policy-gradient，以及較新的 A2C、A3C，以及 PPO 等等的模型，試試看哪種模型在現有的環境上表現會更好。

**超參數調整:**

由於一開始使用了較為麻煩的最小化 loss 的方法，讓 agent 都無法好好運作，於是我也沒將那些參數記下，至於後來解決了 loss function 的問題後，時間所剩無幾，只夠我用來跑其餘模型，所以在此並沒有再去動參數的部分。

所以未來我會慢慢調整模型以及超參數，想辦法讓整體的表現超過第二篇文章裡的表現。

[1] https://blog.paperspace.com/dino-run/

[2] https://github.com/yzheng51/rl-dino-run

**環境的設定:**

在[8]裡，他們分別用了很多不同的方法去處理 environment，但是我就只有用到[7]的方法，將四幀疊在一起。畢竟都已經使用了 gym，之後可以考慮使用不同的設定來處理 environment。除了[8]的方法外，我也會想去嘗試 parallel environments，同時運行多個環境和 agents 來加快訓練速度。

**將 Reinforcement Learning 應用至其它領域:**

Reinforcement Learning 在遊戲這方面已經表現得相當不錯，那是否可以應用至不同的領域上?像是交易機器人之類的。也許未來我可以將市場上的 order book 或交易深度圖，以及其它資料做為 environment，讓機器人依此為據進行交易。

REFERENCES


[1] Rainbow: Combining Improvements in Deep Reinforcement Learning : Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver

[2] Prioritized Experience Replay : Tom Schaul, John Quan, Ioannis Antonoglou, David Silver

[3] Deep Reinforcement Learning with Double Q-learning: Hado van Hasselt, Arthur Guez, David Silver

[4] A Deeper Look at Experience Replay: Shangtong Zhang, Richard S. Sutton

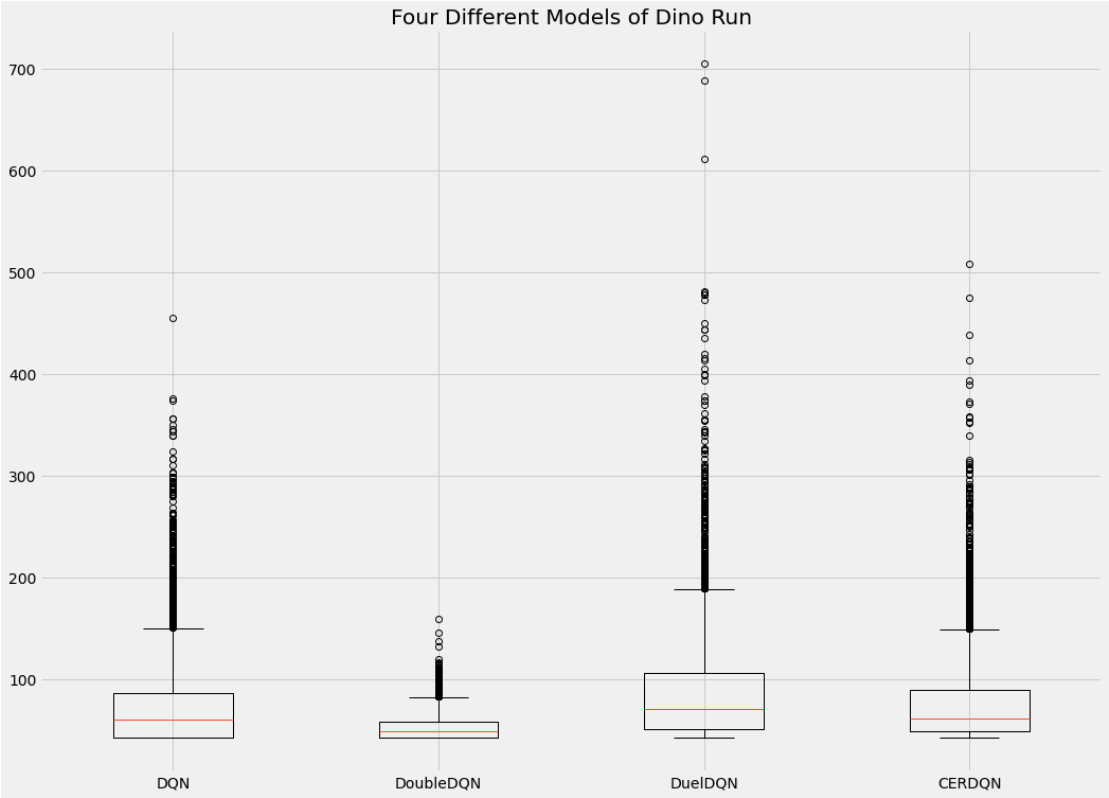
[5] Deep Recurrent Q-Learning for Partially Observable MDPs: Matthew Hausknecht, Peter Stone

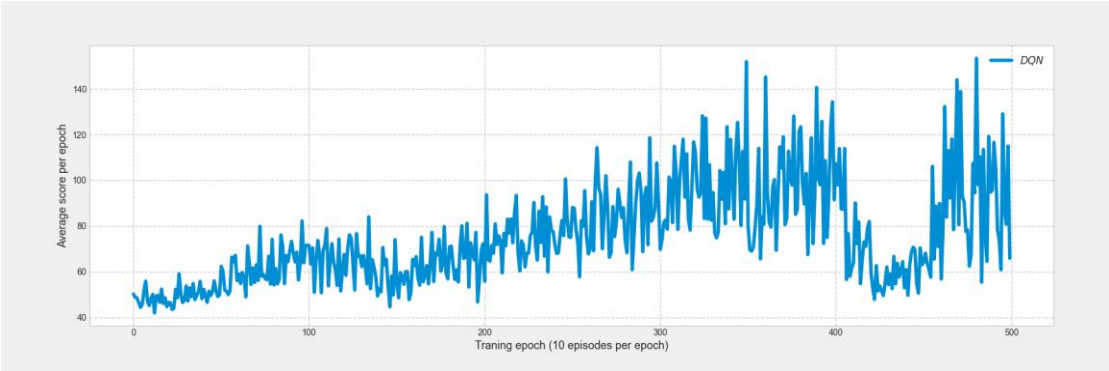
[6] Dueling Network Architectures for Deep Reinforcement Learning: Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

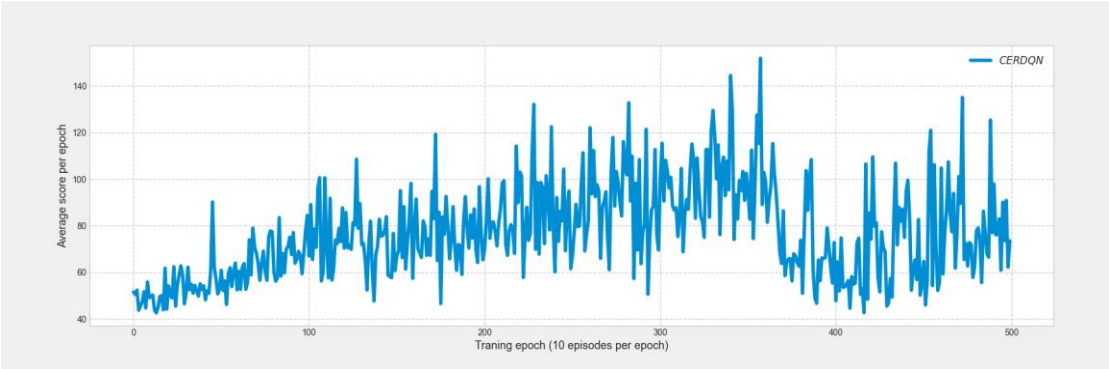
[7] Chrome Dino Run using Reinforcement Learning: Divyanshu Marwah, Sneha Srivastava, Anusha Gupta, Shruti Verma

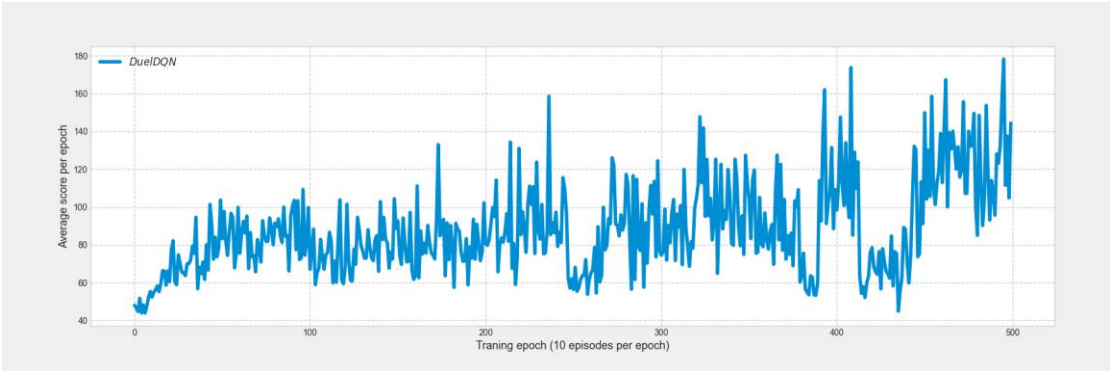
[8] Playing Atari with Deep Reinforcement Learning: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller

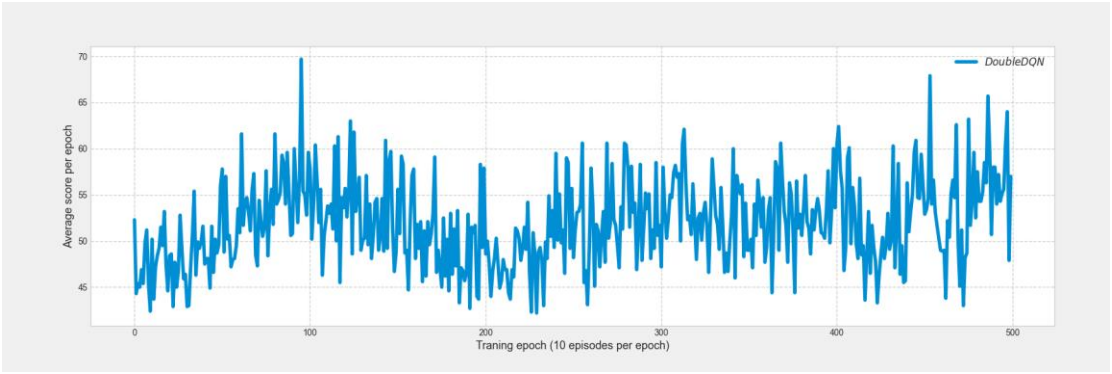
# APPENDIX

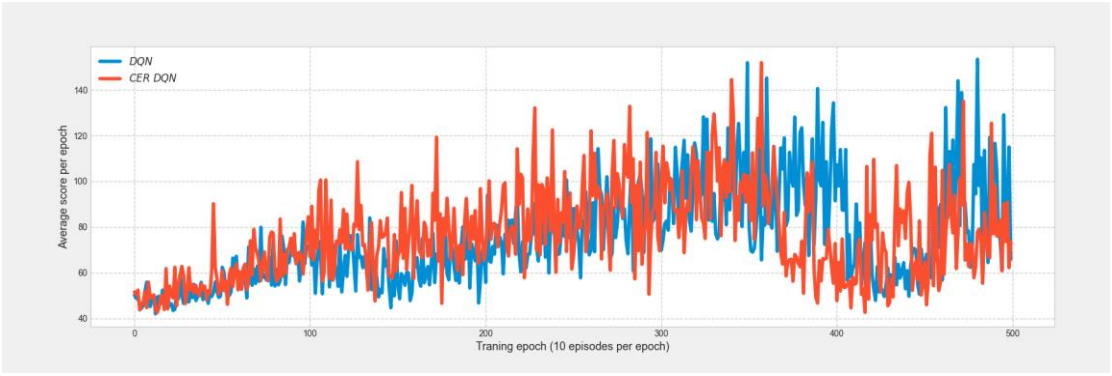

Four Different Models of Dino Run
700
600
500
400
300
200
100
DQN
DoubleDQN
DuelDQN
CERDQN

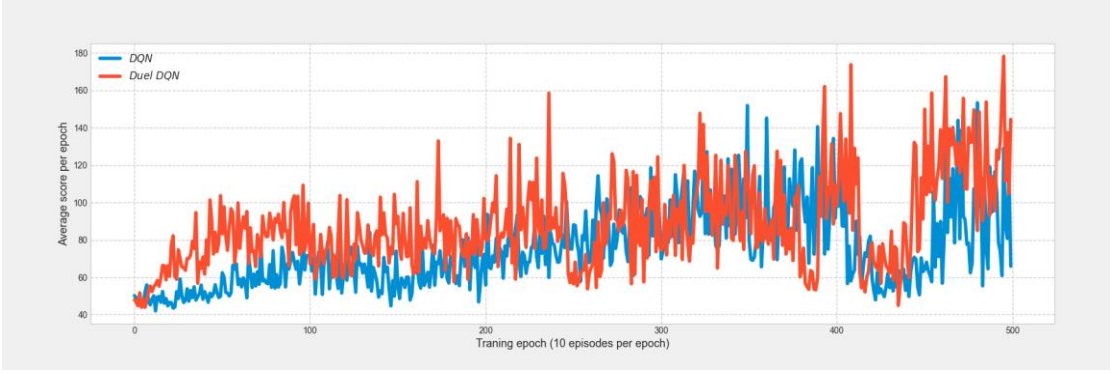

DQN
Average score per epoch
140
120
100
80
60
40
0
100
200
300
400
500
Traning epoch (10 episodes per epoch)

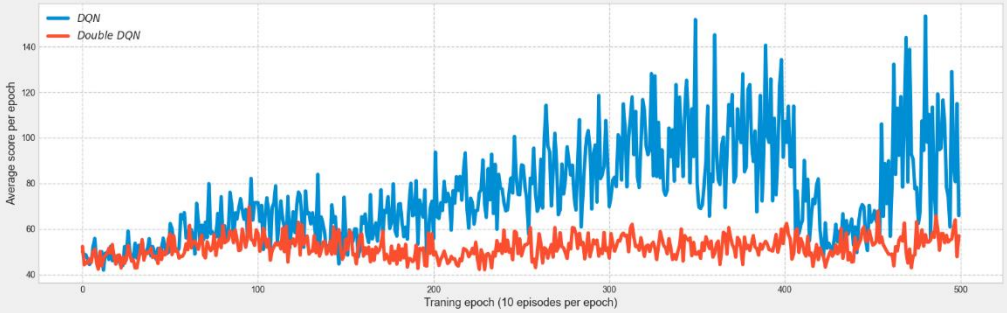

CERDQN
Average score per epoch
140
120
100
80
60
40
0
100
200
300
400
500
Traning epoch (10 episodes per epoch)


DuelDQN
Average score per epoch
Traning epoch (10 episodes per epoch)


DoubleDQN
Average score per epoch
Traning epoch (10 episodes per epoch)


DQN
CER DQN
Average score per epoch
Traning epoch (10 episodes per epoch)


DQN
Duel DQN
Average score per epoch
Traning epoch (10 episodes per epoch)


DQN
Double DQN
Average score per epoch
140
120
100
80
60
40
0
100
200
300
400
500
Traning epoch (10 episodes per epoch)