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

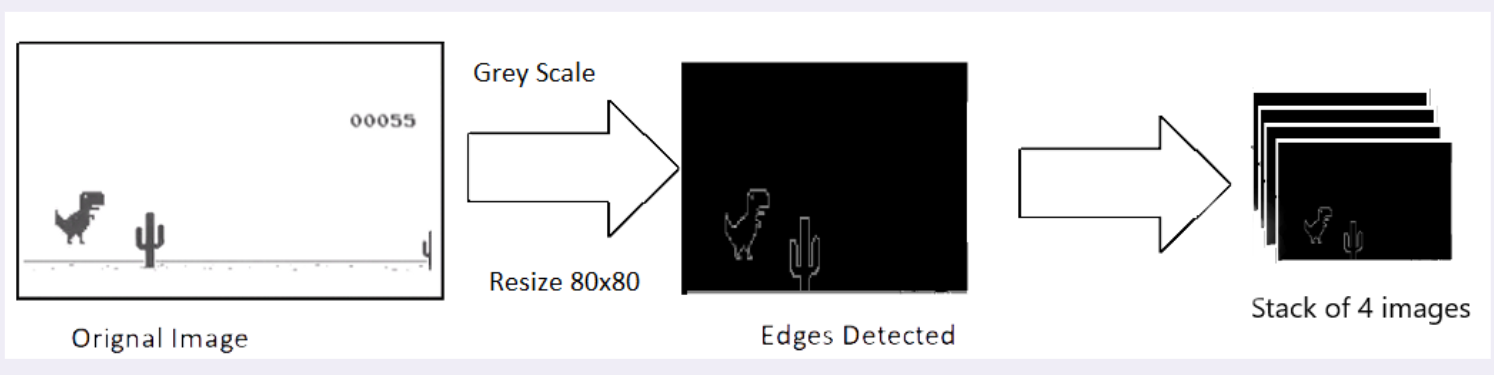
情況S採取動作A後的全部可能的下個情況S’，計算出期望獎勵R後才去採取動作，但是這樣的問題在於時間複雜度過高，儘管後來有較為簡化的算法，但是該算法對於需要即時計算結果的環境仍然相當不友善；至於Monte Carlo是Alpha Go所使用的方法，在此就不贅述；最後，TD比起DP的實時計算，TD使用了**過去的經驗**來估計Q值，在計算上相當的快，只需記錄過去的經驗即可。

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。



Resize 80\*160

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。

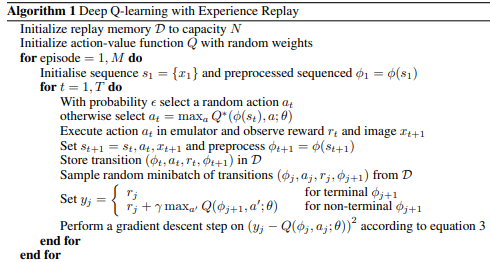
最後其實我原本有考慮在模型裡加入RNN或LSTM等能夠有效處理時間序列的layers。但是在看過[5]之後我認為Dino Run這個遊戲的複雜程度並沒有到很高，小恐龍躲避單一障礙物這件事情並不會影響到未來小恐龍躲避其他障礙物的這件事，所以最後並沒有加入其他layers。

**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:

且在next state為停止遊戲時公式會變為:

所以最後整個訓練過程就會如圖(二)所示:

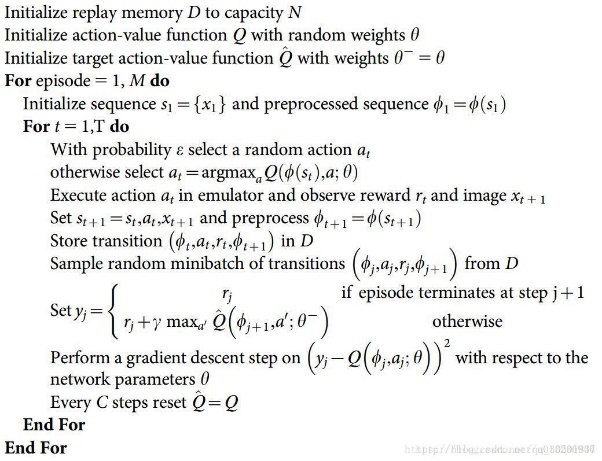


圖(二):DQN完整訓練過程，取自[8]

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。

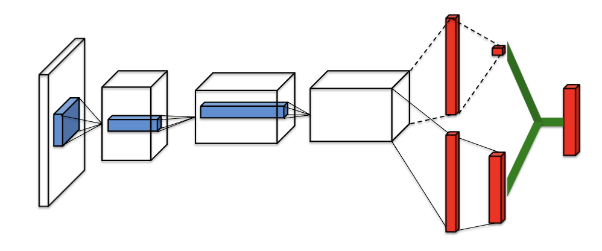
**Nature DQN:**

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: 

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。

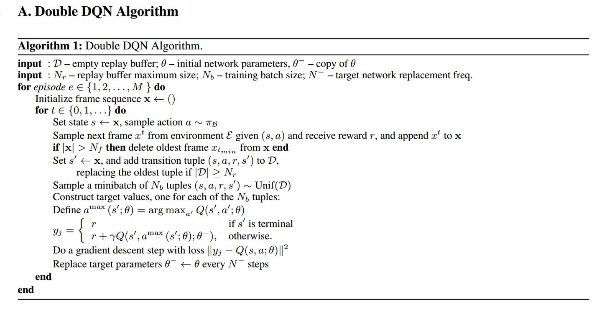
**Dueling DQN:**

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:**

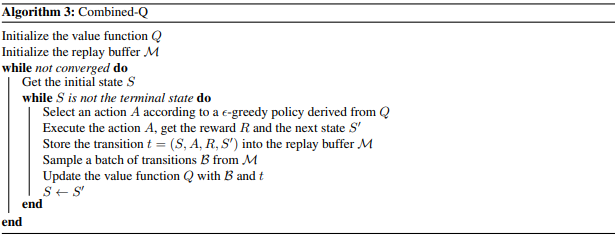
在DQN下expected-Q的預估是使用最大的Q值去做後續處理，但是這也造成了Q值高估的問題。由於DQN是用類神經網路去做逼近，這使得理論上有相同Reward的動作在Q值上會有極小的差異，由於公式是直接取最大的Q值，這使得Q值會被高估。所以在處理上一樣會使用兩個網路，先用主網路選擇動作，再用副網路計算Q值。這樣子即使主網路選到高估的動作，只要副網路沒有高估，計算出的一定是正常值；如果今天變成副網路高估Q值，那麼該動作就不會被主網路選取。詳情如圖(五)所示: 

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:



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**

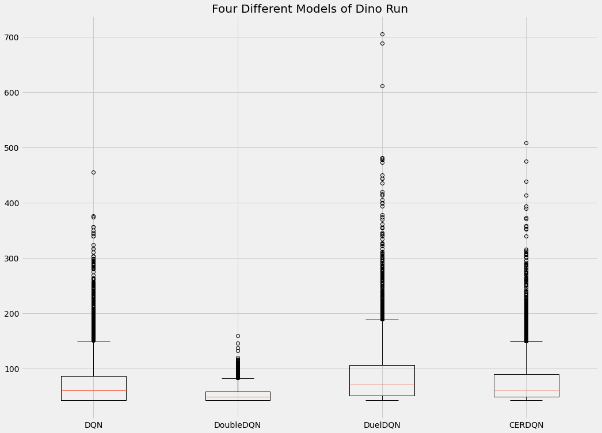
由於一開始是minimize target-Q和current-Q的loss，所以多了額外的learning rate要調整，使得參數調整速度緩慢。後來發現可以只minimize current-Q和expected-Q即可，就無須再調整learning rate。

**未考慮遊戲本身狀況**

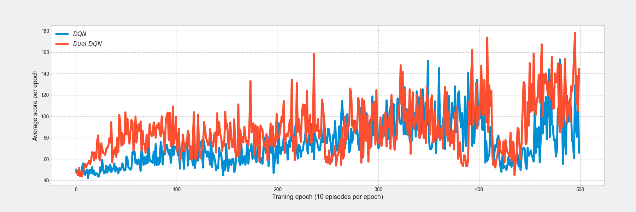
一開始我是有將Terminal state考慮進去。在其他遊戲中，Terminal state意味著遊戲的結束，但是Dino run本身是一個永不結束的遊戲，一旦結束就代表發生了失誤，所以不應該把Terminal state放入考慮。於是乎我最後將Terminal state處理的部分移除，agent的得分狀況馬上獲得大幅的提升。

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:

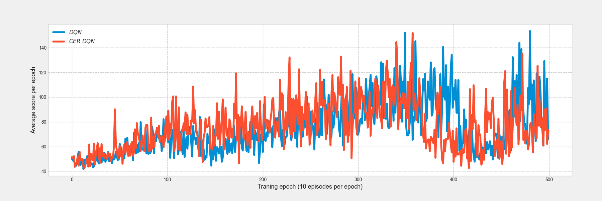
圖(七): 不同模型之間的訓練結果

 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]](#footnote-1)和這篇[[2]](#footnote-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。

**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

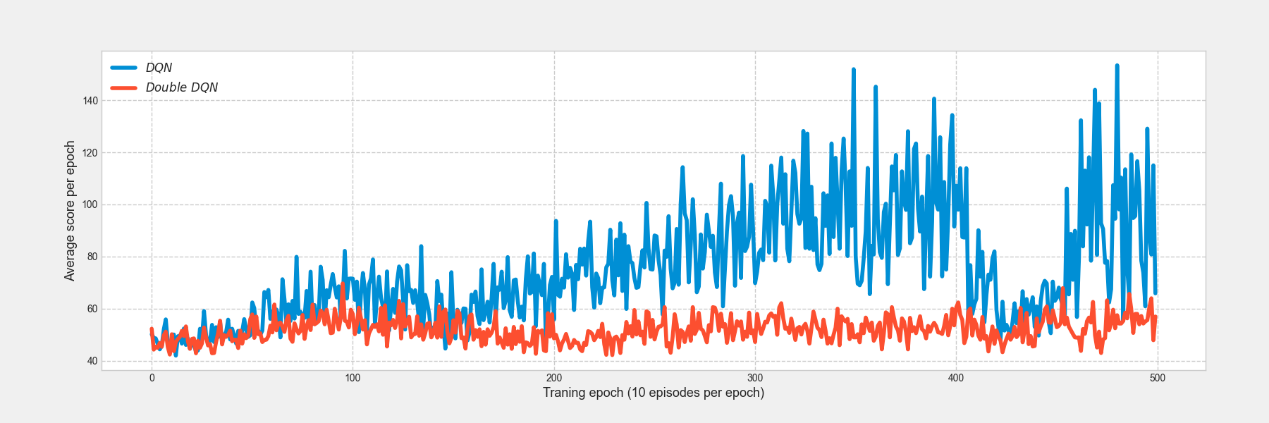
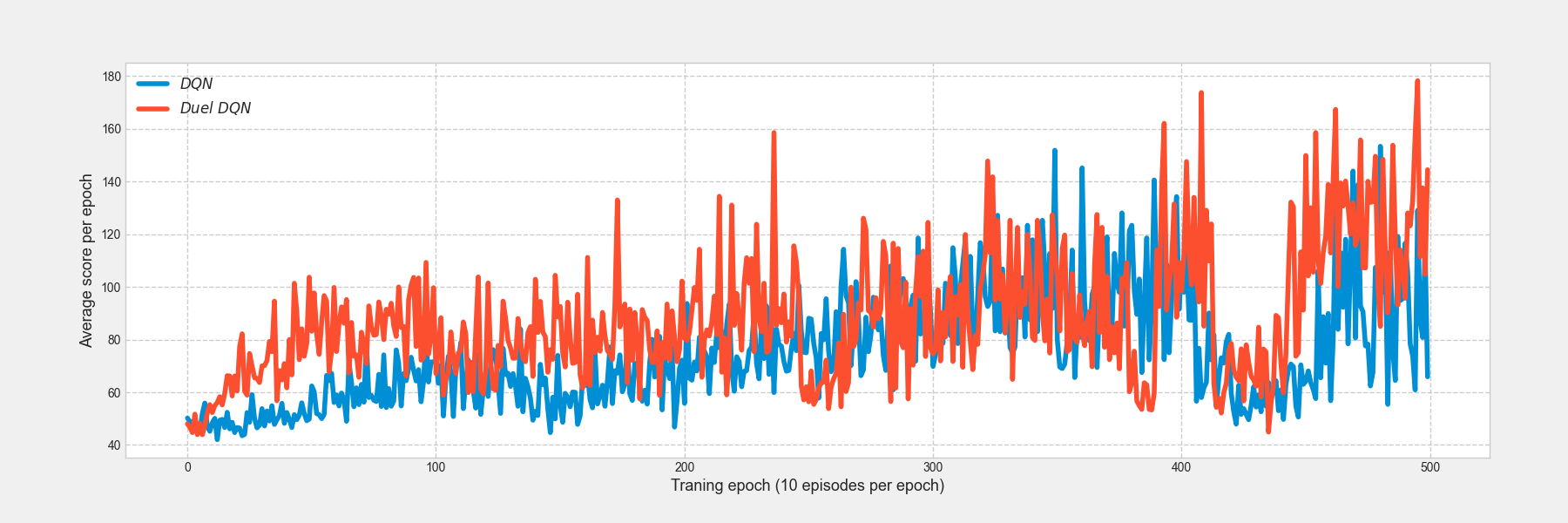
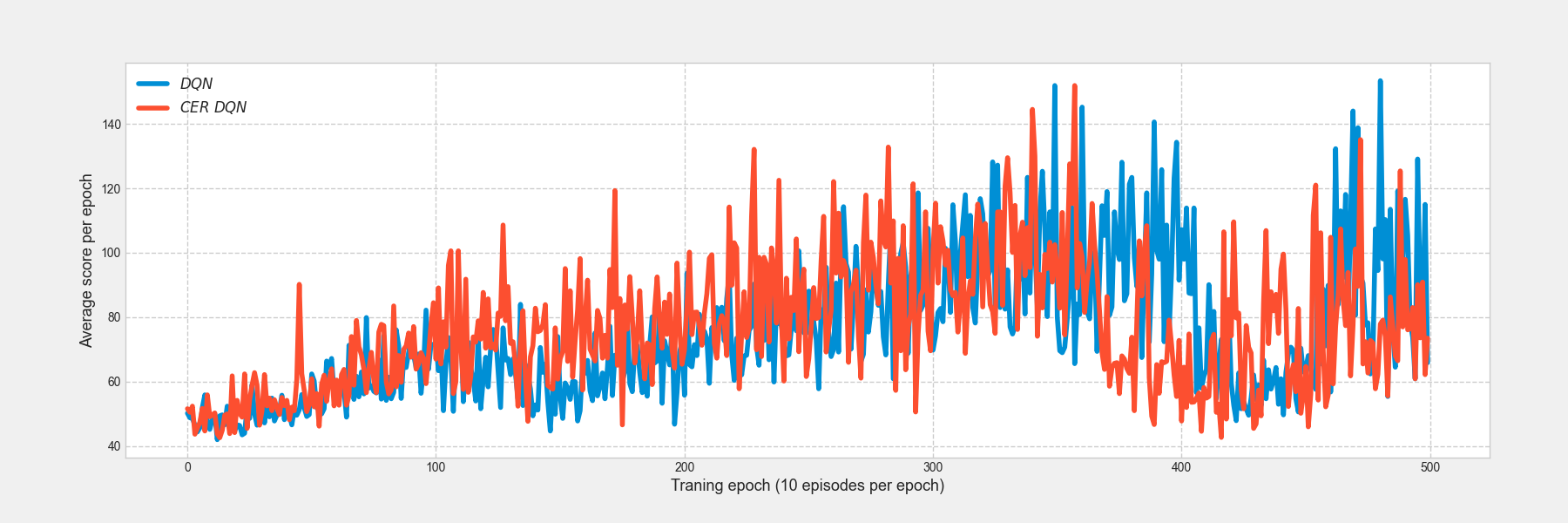
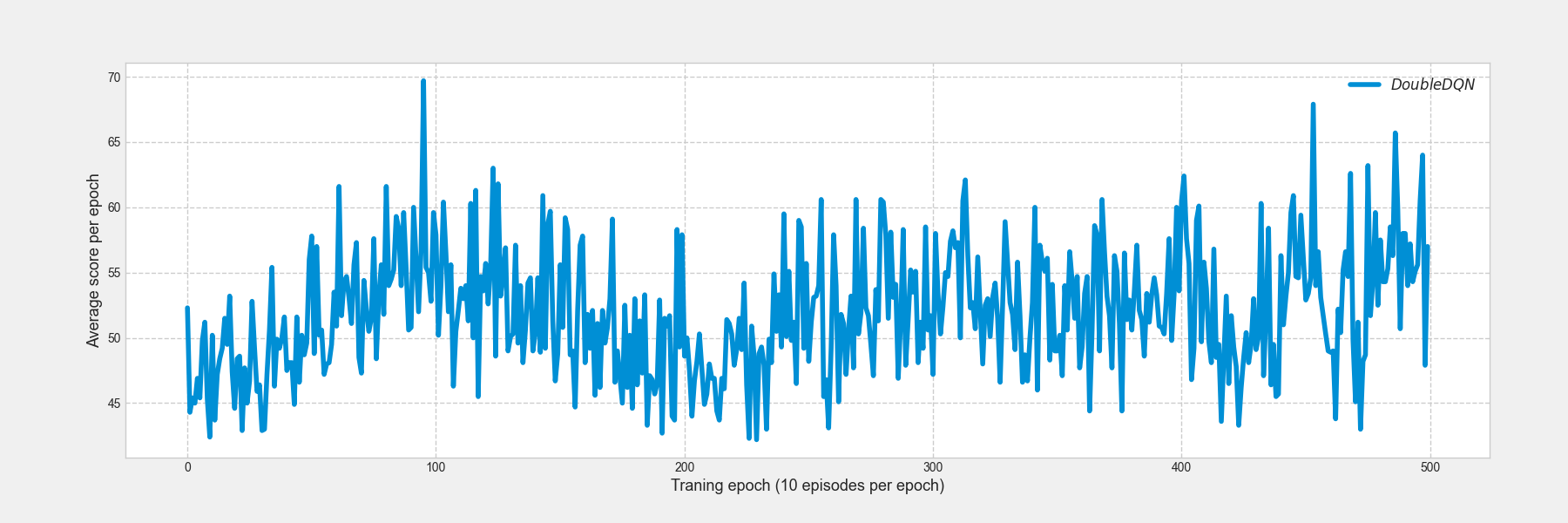
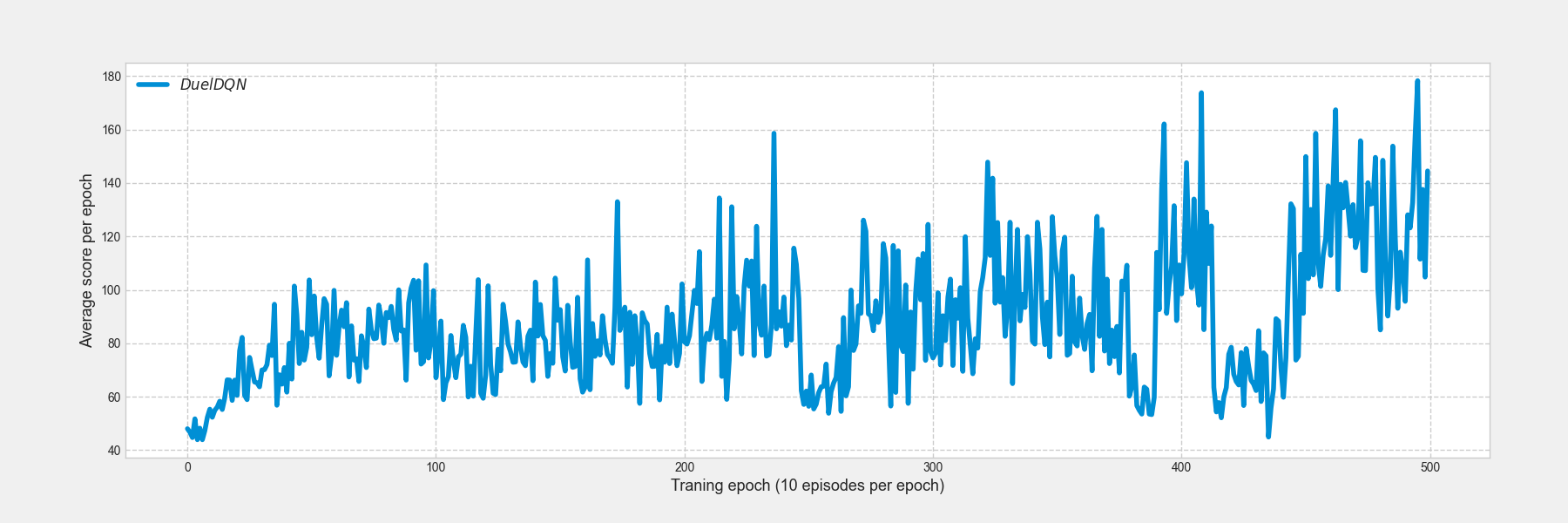
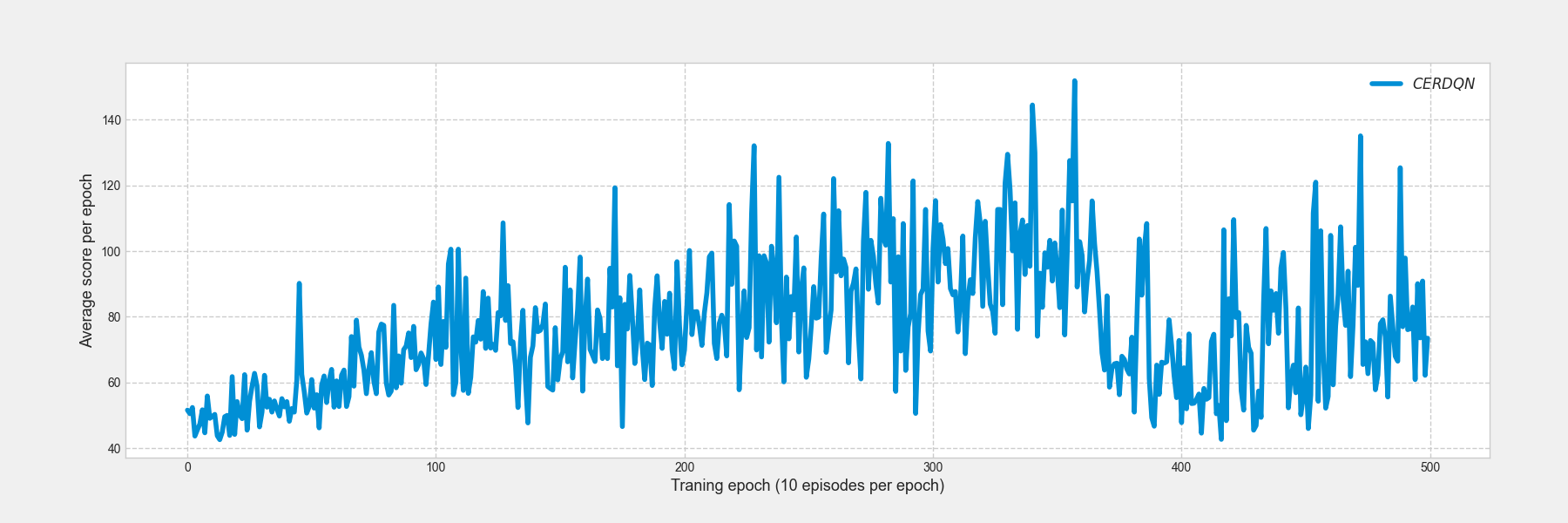
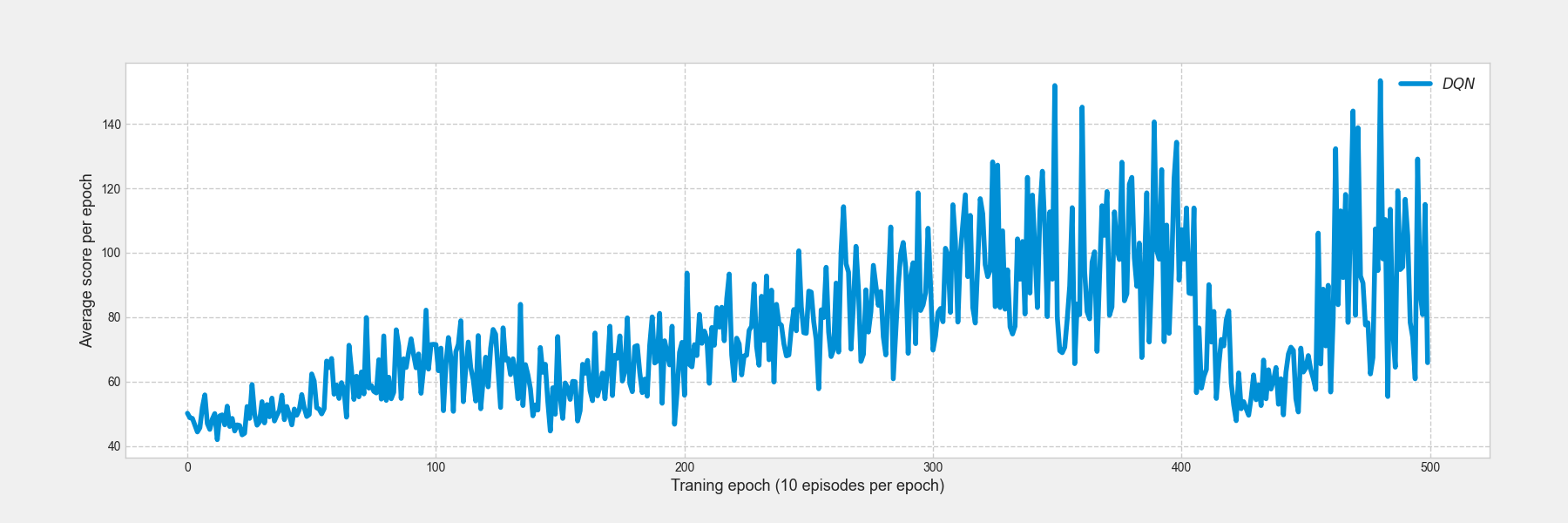
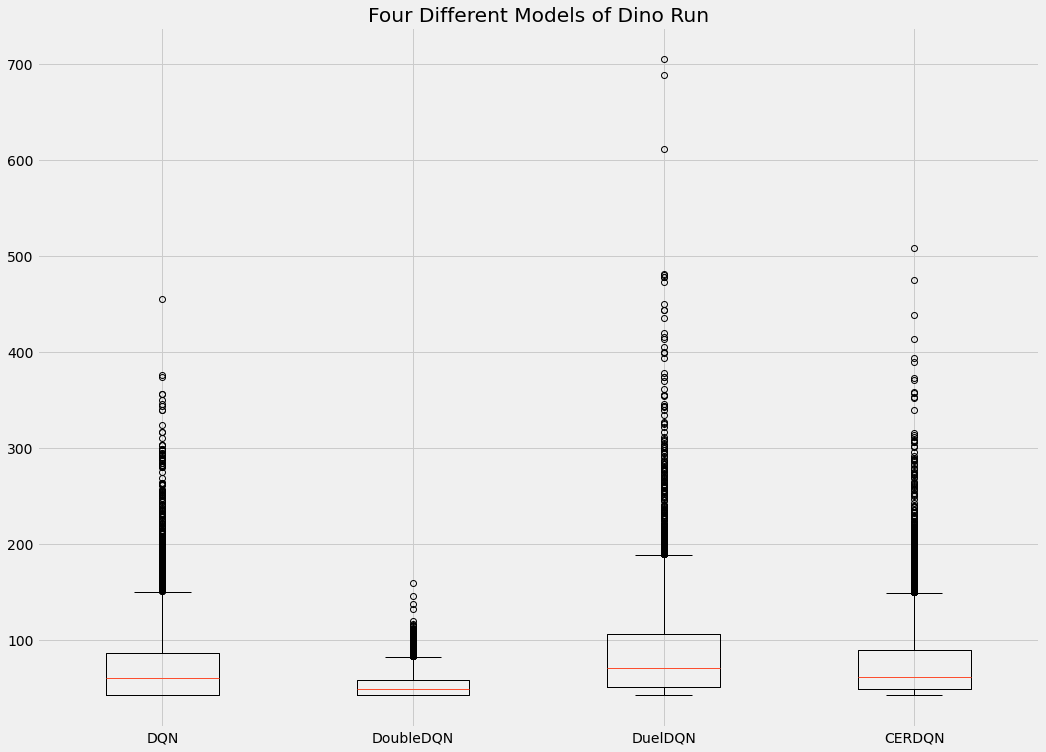
[7] Chrome Dino Run using Reinforcement Learning:

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



1. https://blog.paperspace.com/dino-run/ [↑](#footnote-ref-1)
2. https://github.com/yzheng51/rl-dino-run [↑](#footnote-ref-2)