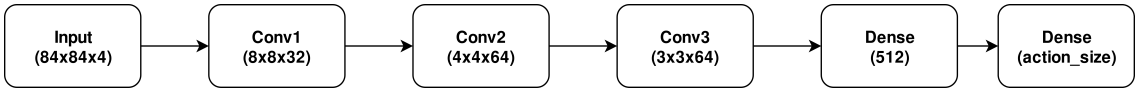
b03901057 詹晉誠

ADLxMLDS HW3 Report

A. Basic Performance

- DQN Model

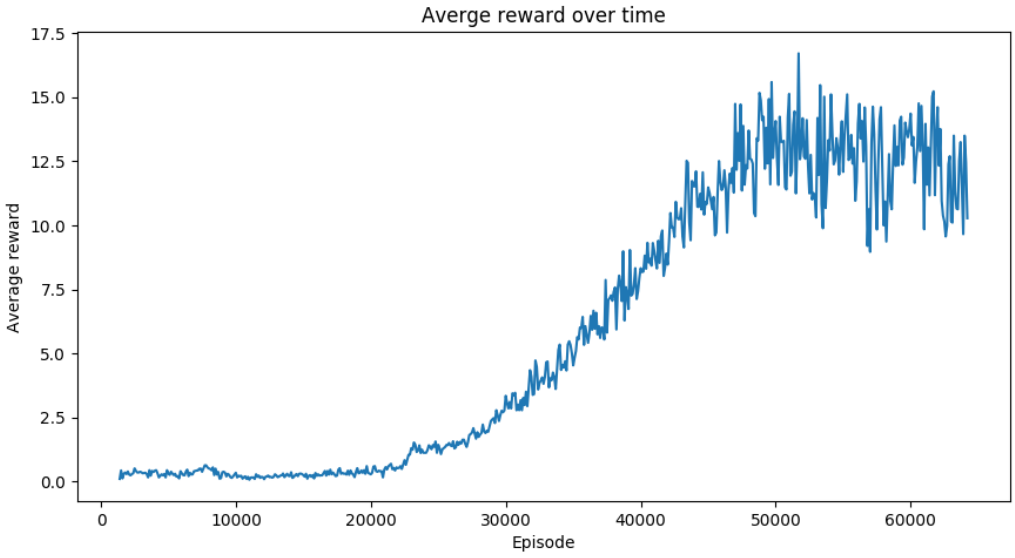
在我的作業中使用的DQN基本上是依照double Q learning所實作出來的。其基本架構是由兩個結構一樣的network所組成的，這兩個network在這份報告中我稱之為value Q network以及action Q network，因為value Q network主要是用來估測一個policy的value (expected future rewards)，而action Q network是用來決定action的。Q network的架構如下圖所示：



Input是一個84x84x4的tensor，然後先會經過一層有32個filters，每個filter的大小是8x8的convolution layer，之後會再經過兩層的convolution layer，在flatten。最後會經過一個大小爲512的dense layer，以及一個size 爲action size的dense layer。
此外，在進行training的時候，我會使用epsilon-greedy的技巧增加model嘗試不同action的機率。epsilon的值會隨著episode慢慢以linear的方式decay，然後當epsilon降到一定的值後就不會再decay了（我設爲0.1)。

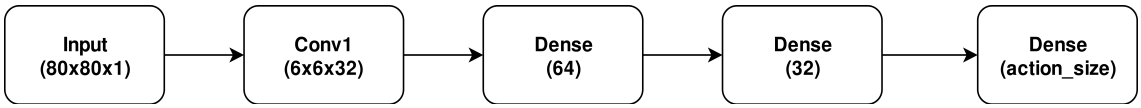
- DQN Performance on Breakout

下圖顯示了我的DQN model training時隨著時間的reward變化（averaged over the last 30 frames)。圖中顯示的是training時在過去30個episodes每個episode的平均reward



- Policy Gradient Model

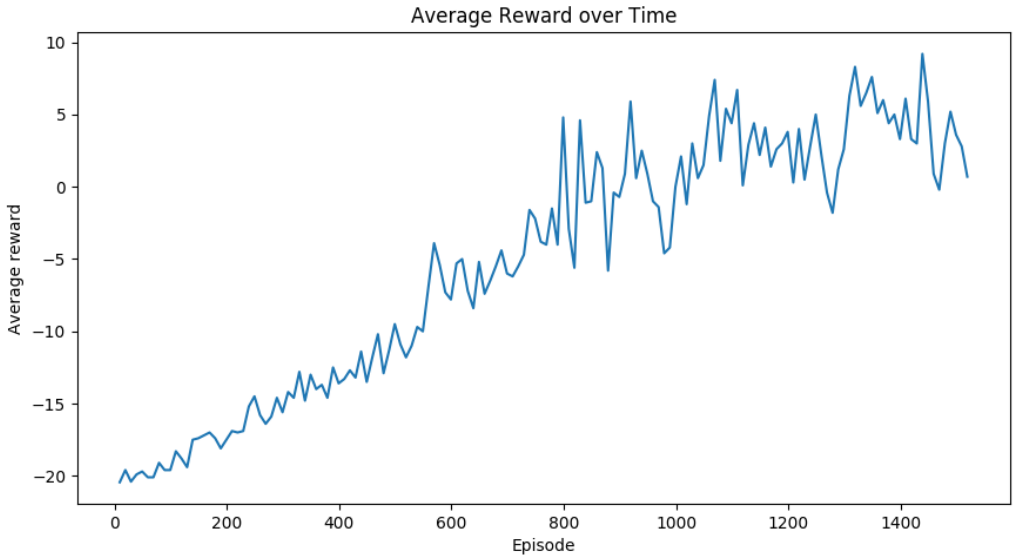
在這次的作業中我實作了最基本的REINFORCE policy gradient演算法。我會先將每個frame作pre-processing，而pre-processing包含擷取畫面中間的視窗，將背景去掉，以及跟上個frame作相減。在經過pre-processing的frame會被餵進value network來預測下個要執行的動作。其中的value network結構如下圖所示：



第一層是一個有32個6x6 filter的convolution layer， 然後會經過大小爲64, 32, 以及action size的dense layer。此外，為了加速我model的訓練速度，我將action限制在1-3之間，也就說action 0, 4, 5是我model無論在training time或是testing time都不會使用的action，因為這些動作對於model是否能訓練好影響不大。另外，在policy gradient中，因為每次reward的scale會不一樣，為了normalize reward的scale，我會在計算完running reward後做normalization (減掉mean再除以std)。

- Policy Gradient Performance on Pong

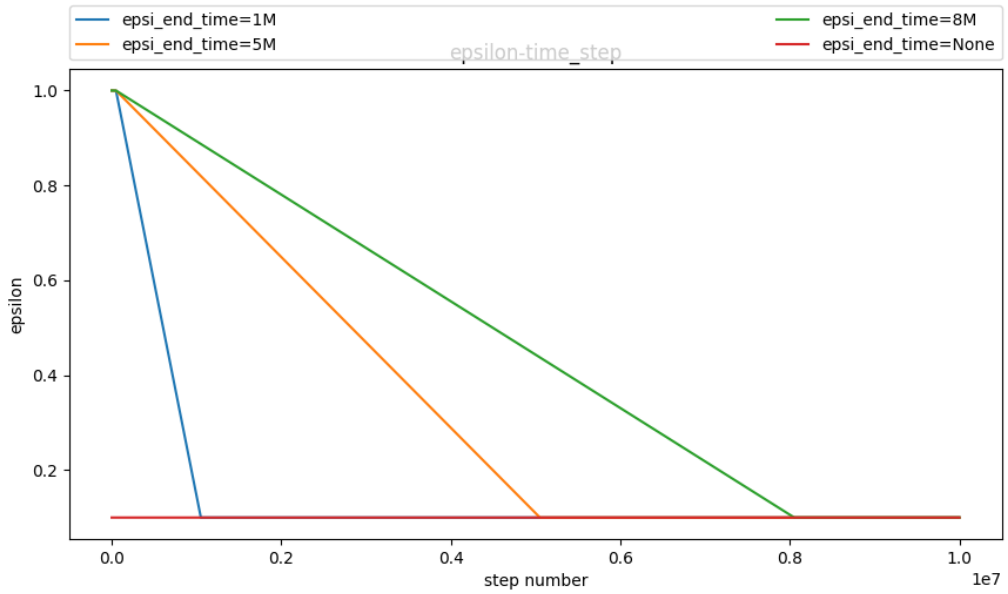
下圖顯示的是我的policy gradient model在training時平均reward隨時間的變化 (過去30的episode的平均），可以發現training一開始的時候平均reward直線上升，不過當reward接近1時，reward就會快速震盪。這樣的現象是在policy gradient中常見的問題，因此我會隨時紀錄平均reward，然後把平均reward較高的幾個model保存起來。此外，我也發現policy gradient對initialization蠻敏感的，有時候如果initialization好，model就會train的比較快，反之，可能平均reward會卡在某個地方一直上不去。

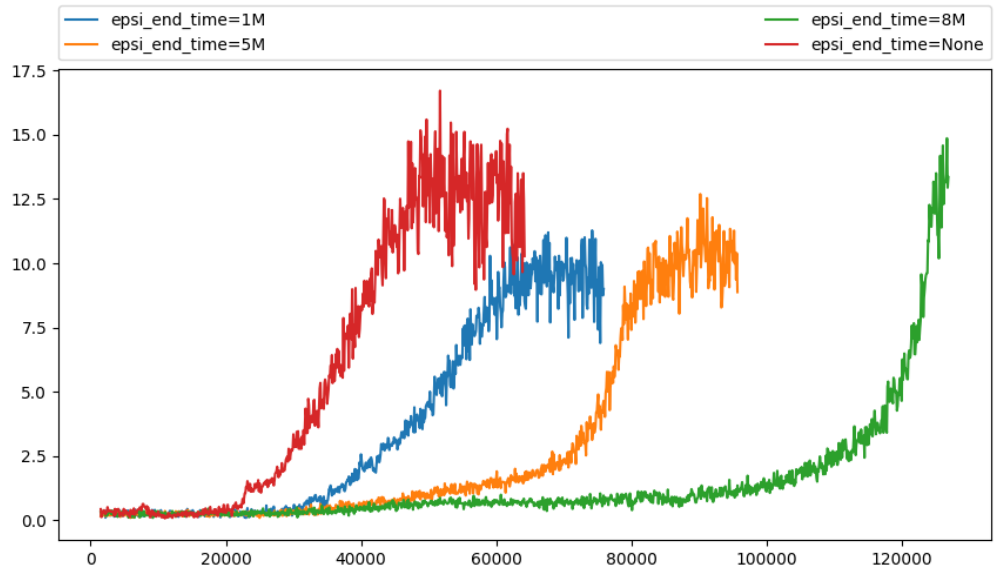


B. Experiment with DQN hyper-parameters

在這次的作業中，我改變了exploration schedule的參數，也就是說我改變epsilon-greedy中的epsilon decay速率。改變這個參數的原因是因為DQN在training時需要透過epsilon-greedy來探索不同的動作，以避免卡在local minimum，例如在training一開始的時候Q network個估測是很不準的，因此從Q network計算出的動作往往也是不準的，這時候如果一直依照著Q network的指示做動作，往往會做出錯誤的動作，因此永遠無法學會更好的動作，這時如果可以讓agent隨機做一些動作，agent就有機會探索到不同的動作，進而學到更好的Q network。不過，如果epsilon 太高（太常做隨機動作）agent自然也無法train好，因此要找出一個最佳的exploration schedule需要透過一些實驗。

以下是我四個實驗中epsilon隨著time step（不是episode)的變化：





epsi_end_time指的是epsilon在第幾個time step時會decay到最終的值（0.1）並且往後就不再decay。下圖是這四個實驗在Breakout中的learning curve (x軸為episode不是step number)

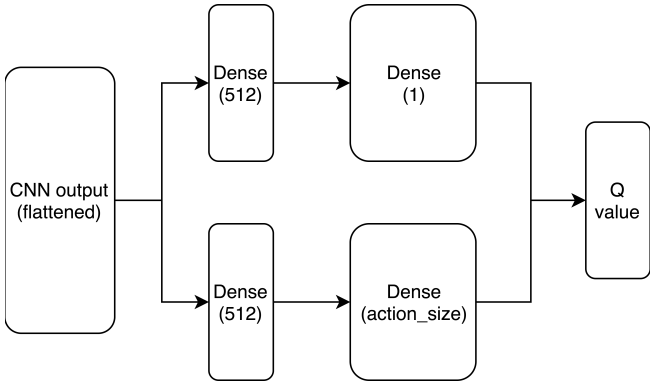
在這四個實驗中我都是固定train 900M個step（每做一個action算是一個step)，而圖中的x軸則是episode number，y軸是average reward (last 30 episodes)。從圖中可以發現當epsilon decay太快時，最後model train出來的performance不會太好（例如藍線與橘線的）。不過有趣的是如果epsilon一開始不decay（紅線），最後model train出來的performance還不錯，而且不需要這麼多episode就可以達到好的performance，相較之下當epsi_end_time設為8M的時候需要幾乎兩倍的episodes才能達到一樣的performance，不過似乎（這邊實驗還沒做的很完整）震盪沒那麼大。但是值得注意的是雖然這四條線training的episodes數量不太一樣，不過他們都是900M個steps（因此training time差不多）。

C. Bonus -1: Double Q and Duel Q learning

Code: network_bonus1.py, agent_dqn_bonus1.py。Double QDN 以及Duel DQN實作在network_bonus1.py, agent_dqn_bonus1.py中，並且透過if-else來控制要用使用哪種結構（例如do_double_q=True，do_duel_q=False，就會只使用double DQN）。

Double Q learning的network架構如Section A中所描述的，而value network update frequency是10000個steps。

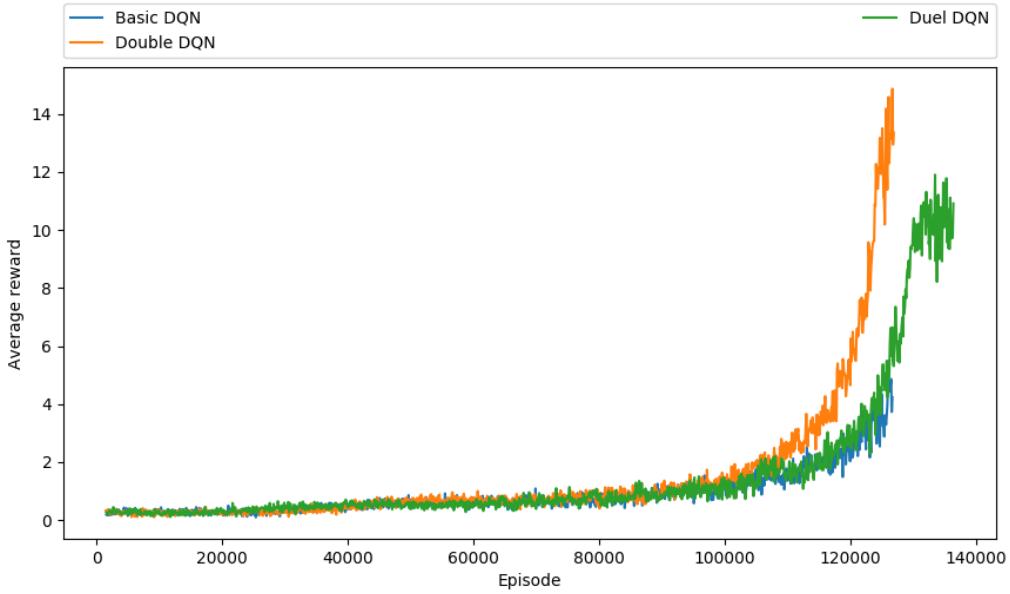
Duel DQN 的network架構如下所示：



CNN output會分別經由兩個大小為512的fully-connected layers，之後上面(value)的fully-connected layer output會在經過一個fully-connected layer變成大小為1的output，下面的(advantage)會經過大小為action size的fully-connected layer，最後兩個output直接相加變成Q value (下面的advantage會先減掉他自己的mean再與value相加)。

Double DQN可以讓network更穩定，因為傳統DQN中同一個network會估測Q value並用argmax決定action，但是這樣很容易over-estimate Q value，讓估測失準，因此double DQN將決定action與Q value的network分開，讓Q value的estimation比較準一些，進而提

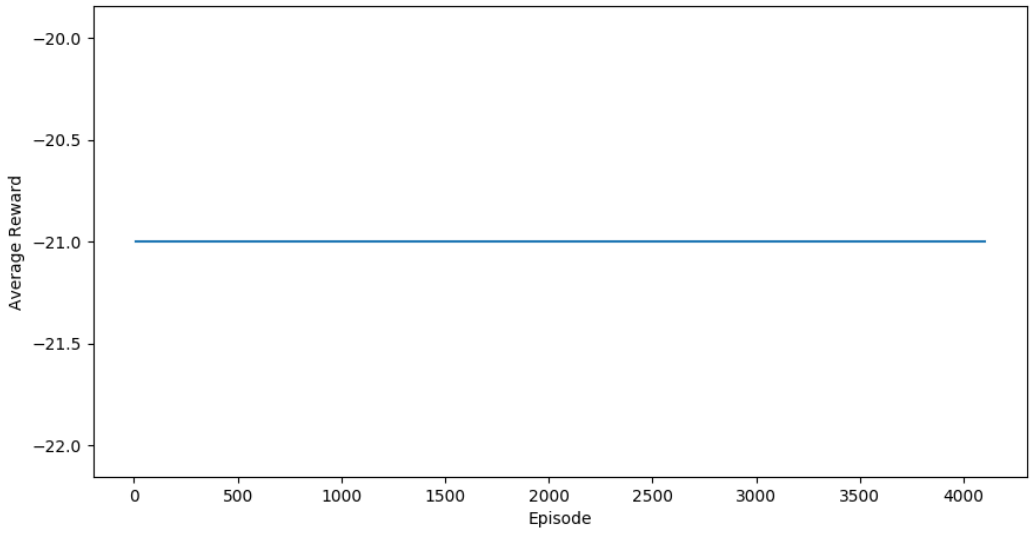
升performance。Duel DQN則是將Q value拆成跟action無關的value function（估計一個state多好）加上一個跟action相關的advantage，就是在現在的state，做這個action可以帶來多少額外的reward。因為將Qvalue拆的更細了，所以network可以learn到比較細節的資訊，因此performance 也會提升。
在實驗中我分別在Breakout中測試basic DQN, double DQN, 以及duel DQN三種方式的表現，並且epsi_end_time設為8M，learning curve如下圖所示。從圖中可以發現最基本的



Q learning表現最差，他的performance上升的最慢，其次是duel Q learning，最佳的是double Q learning，因為他不僅reward上升最快，並且他最後的average reward也是最高的。

D. Bonus -2: Actor-Critic

Code: network_ac.py, agent_ac.py。Actor-Critic因為較為複雜，所以無法與其他code簡單的整合，因此我將其network架構定義在network_ac.py，而make_action, train等function寫在agent_ac.py裡面。
Actor critic是將value-based (ex: DQN)與policy-based (ex: policy-gradient)方法結合，他與policy-gradient最大的差異在於policy gradient中reward（Q value)的估計是用running reward (也就是跑完一次模擬，然後把reward累加起來），這樣的方式屬於Monte-Carlo learning ，因此policy-gradient的問題是variance會很大。而actor-critic則是將Q value的估測換成一個network，因此比較像是TD learning（不是依靠完整sequence的資訊來計算reward)，可以減少variance，因此performance會比較好。
Actor-critic的表現如下所示：



從圖中可以發現，actor-critic的average reward都一直停留在-21，也就是說這個network並沒有學到東西。關於這個問題初步的推斷是因為network參數在update時有寫錯，因為如果network的參數有update到，performance應該多少會有一些變化（不一定會變好，但是至少會震盪），會出現reward都固定的情況比較可能是parameter沒有update到。不過因為礙於時間因素，我無法繼續debug，找出真正的原因。