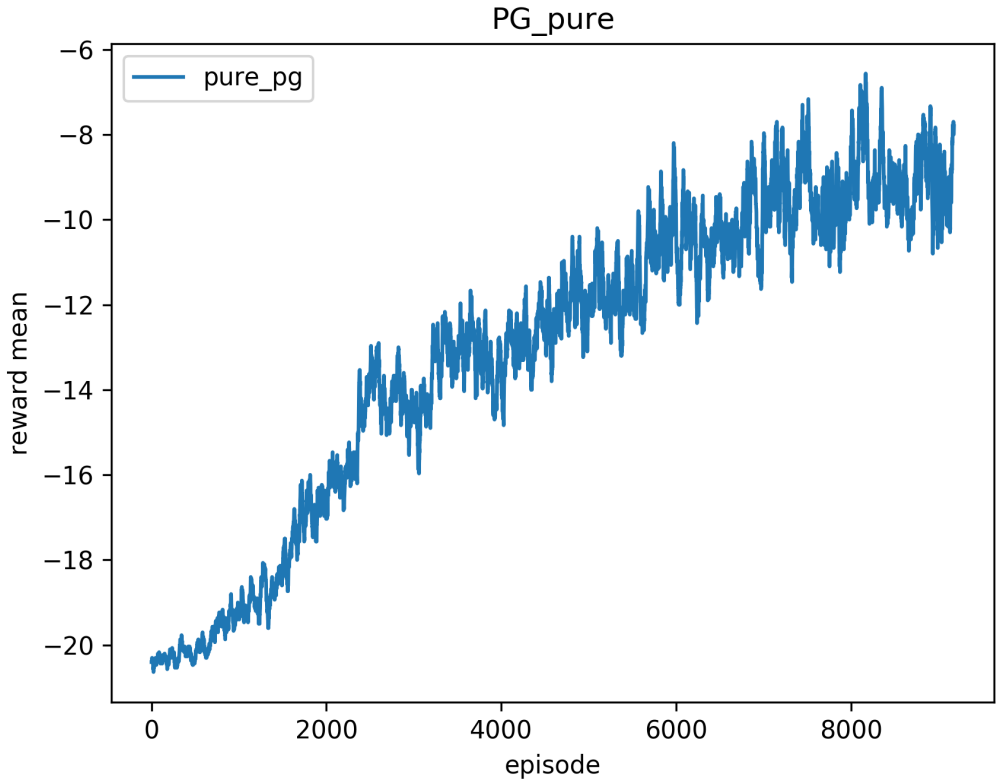
ADL hw3 report

r06922057 梁智泓

Policy Gradient

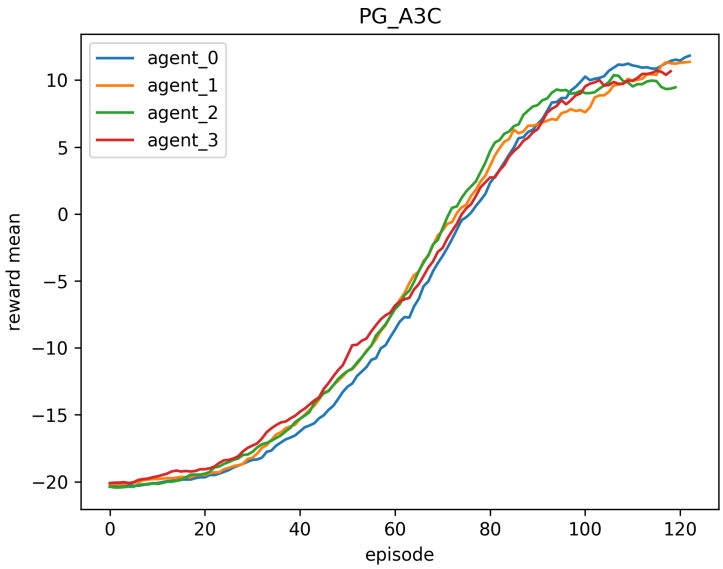
我的 Policy Gradient model 的主要架構是兩層的 DNN，每一個 observation 進來，會先對他做一次 preprocess，把畫面中不要的部分（遊戲畫面外圍）切除，並且把它轉成 80*80 的灰色畫面，再將現在 obs 減掉前一個畫面的 obs，得到 obs_trans，最後將這個 obs_trans 餵進 network 中，得到在目前 obs_trans 下 action 的機率分佈，然後再從這個機率分佈中，照著機率分佈隨機得到新的 action，然後對 env 執行這個 action，以得到下一個 observation，直到遊戲結束為止。當一場遊戲結束之後，才將整場遊戲所得到的 obs_trans 跟 action 以及 reward 餵進 net 中，對 net 做更新，而 reward 在 net 中做更新時，會先對它做一次 discount，透過 gamma 讓 reward 隨著每場遊戲的 step 漸漸衰退，然後在對它做一次的 normalize，使 loss 更容易 gradient descent，而 loss 則是 action 的機率分佈減掉實際 sample 出來的 action 的 l2_loss。透過每場更新 model 參數，讓 model 學習，使其可以慢慢地得到好的分數。下圖是實驗的結果。



從圖中可以看出，雖然看得出來有在學習但學習的速度非常慢，跑了 8000 個 epi 才可以學到平均-8 左右，實際跑過大概可以跑到+1 左右（約兩天），但因為實驗時間不夠，所以只讓他跑到 8000 epi（約半天）。

A3C

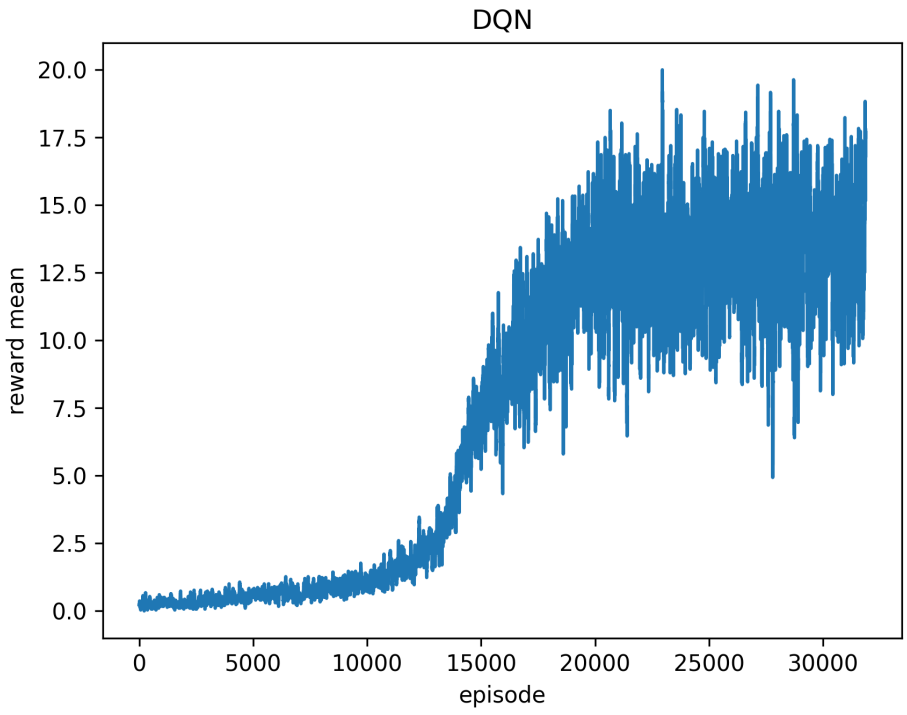
為了讓學習的更有效率且學得更好，我使用了 A3C 的作法，而 A3C 之所以能夠學習的比傳統 PG 來得快是因為 A3C 它結合了以值為基礎的（例如 Q learning）及以 action prob 為基礎的（policy gradients）的強化學習方法，能夠毫不費力的在連續動作中取出合適的動作，更可以進行單步的更新，不像傳統的 policy gradients 需要以局為單位的進行更新，大幅的加速了學習的速度，除此之外，他更加上的平行化的處理，讓多個 sub_agent 同時進行遊戲，然後再不同步對 global model 進行更新，讓學習的效率大幅的提升，也讓每個 sub_agent 學得更好。我的 A3C model 主要的架構是兩層的 cnn，接上一層的 dense，最後再接一層 output 用的 dense，參數則是完全照個助教的提供的參數， actor network 會根據 network 的機率分佈選擇 action，而 value network 會給根據 actor network 所做出的 action 進行評分，然後 actor network 再根據 value network 所給的評分（advantage）進行 loss 的修正，主要跑的流程跟上面傳統的 pg 很接近，唯一比較不同的是 value network 每 5 個 step 或每盤結束時，就會根據過去的 5 個 step 的結果進行一次更新，透過生成的 value 與 reward 更新它的 advantage，讓 actor network 在計算 loss 時，會根據 advantage 影響 actor_loss，而整個 model 的 loss 則是 actor_loss + value_loss*0.5，再透過 RMSPropOptimizer 讓 model 可以透過互相影響的方式同時學習 value network 與 actor network。最後，再做出多個 async 的 sub_agent，每個 sub_agent 在每場遊戲的一開始會先 copy global model 到 local model，然後隨著 value network 更新時，一同更新 global model 的參數，並且將 global network 再一次 copy 到 local network，再繼續 training，我開了 12 個 sub_agent 同時非同步地進行 training，而為了實驗圖片不要過去混亂，我只畫出了前面的四個 sub_agent 的趨勢圖，如下圖所示。



從 A3C 的圖與傳統 PG 的圖，可以看出 A3C 明顯學的比傳統的 PG 還要來得快又好，A3C 可以讓每個 agent 在不到 150 場就能學到平均 10 分左右，而 PG 則需要 8000 多場才能學到平均-8 左右。

DQN

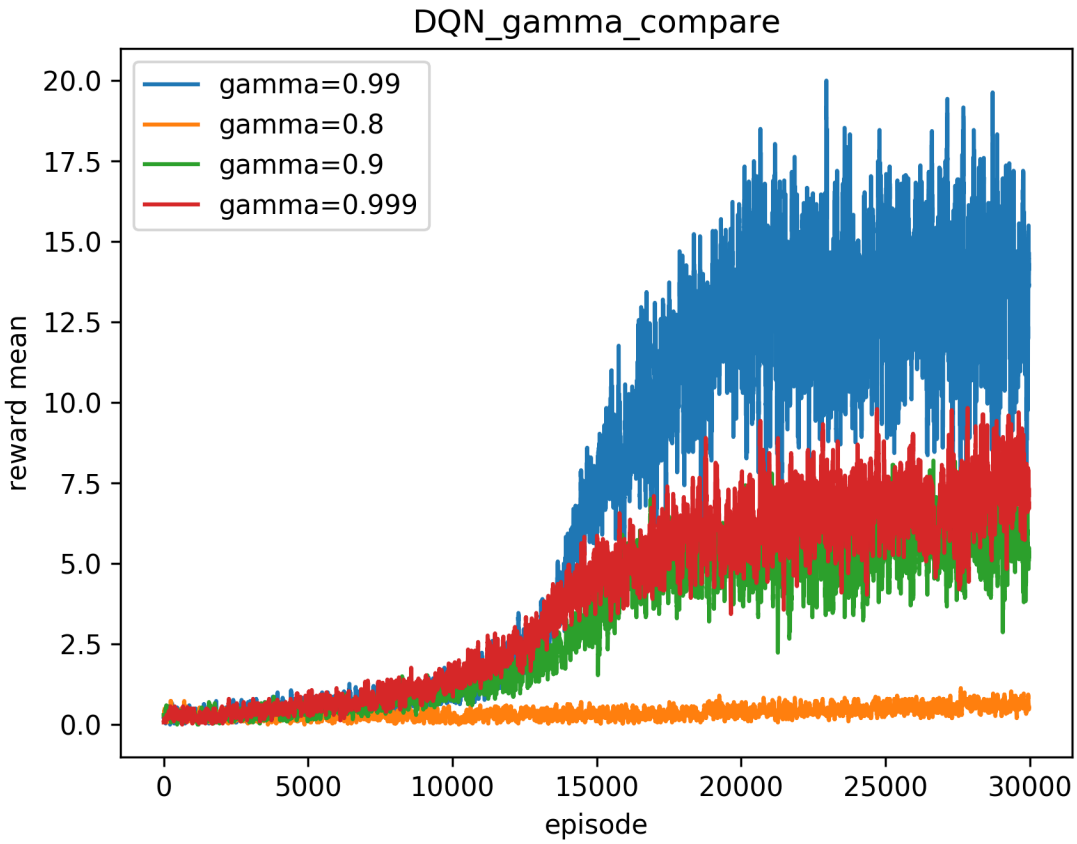
我的 DQN model 的主要架構跟助教所提供的一模一樣，三層的 cnn 加上兩層的 dense，每層之間都有使用 relu，而最後一層則是使用 leaky relu，然後 output 出每個 action 的 Q_value，而 training 的主要流程為，一開始會先透過 epsilon 的大小去機率取樣，若 random 出來的值小於 epsilon，則會 random sample 一個 action，若 random 出來的值大於 epsilon，則會透過 Q_net 選出最好的 action 並且執行，而 epsilon 會隨著 train step 的變大，而漸漸變小，換句話說，training 的一開始 agent 會接近隨機亂動，而隨著 train step 漸漸變多，agent 會漸漸轉而使用 Q_net 所選出的 action，而之所以要做這樣的 random action，是因為在一開始時需要讓每個 agent 做夠多的 exploration，讓它不會只能學到某些特定的 state 與 action 的組合。而在 learning 開始之前，agent 必須記下 memory size 的 state, action, state_next 與 reward。等到 memory 滿了之後，才會進行 model 的 learning。learning 的一開始會先在 memory 中隨機 sample 出 batch size 的記憶，然後用 batch size 的 memory 中的 state 餵進 Q_net 中，然後使用當時 memory 中 action 得到 Q_value。再將 batch memory 中的 state_next 餵到 Q_target_net 中，然後取機率最大的 action，得到 Q_next，再將 Q_next * gamma + reward 作為 Q_value_expect，而需要注意的是，若 state_next 為 done 的情況下，Q_value_expect 就直接是 reward，最後再將 Q_value 與 Q_value_expect 算 loss 做 optimize。而 Q_target_net 與 Q_net 的結構相同，不同的是 Q_target_net 會每 1000 個 step 才會將 Q_net 的參數複製過來，換句話說 Q_target_net 就像是過去的 Q_net，也就是實際的狀況，而 Q_net 就像是預期的狀況，DQN 就是希望能使兩者之間的差異越小越好，讓 Q_net predict 出來的 Q_value 跟實際的狀況一樣。下圖便是我的 DQN model 在 training 時所跑出來的實驗結果



從圖中可以明顯看到，在 training 的前期 reward 相對低，因為 DQN model 在前期需要做 exploration，因為 epsilon 還沒 decay 完，所以選出來的 action 是透過 random sample 出來的，隨著 epsilon 漸漸 decay，model 會漸漸選擇相信自己所選擇的 action，也就是 model 所 predict 的 Q 值越來越接近現實值，所以 training 的中期爬升的很快，直到爬升到平均 12 左右，慢慢趨於穩定，在 10-15 之間震盪。

Hyper parameter experiment

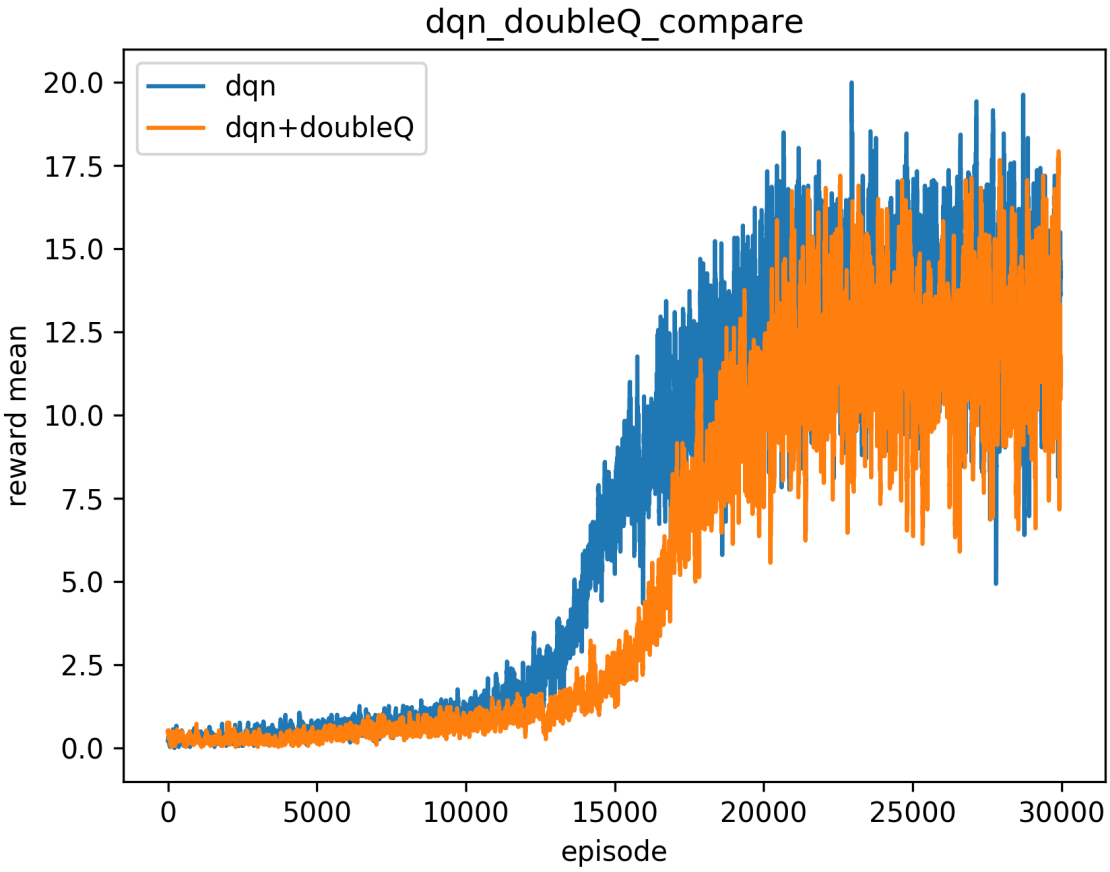
在 Hyper parameter 的實驗中，我選擇 gamma 作為實驗的主要對象，我做了四個實驗，分別是最原始的 0.99 與 0.8, 0.9, 0.999，下圖便是實驗的結果



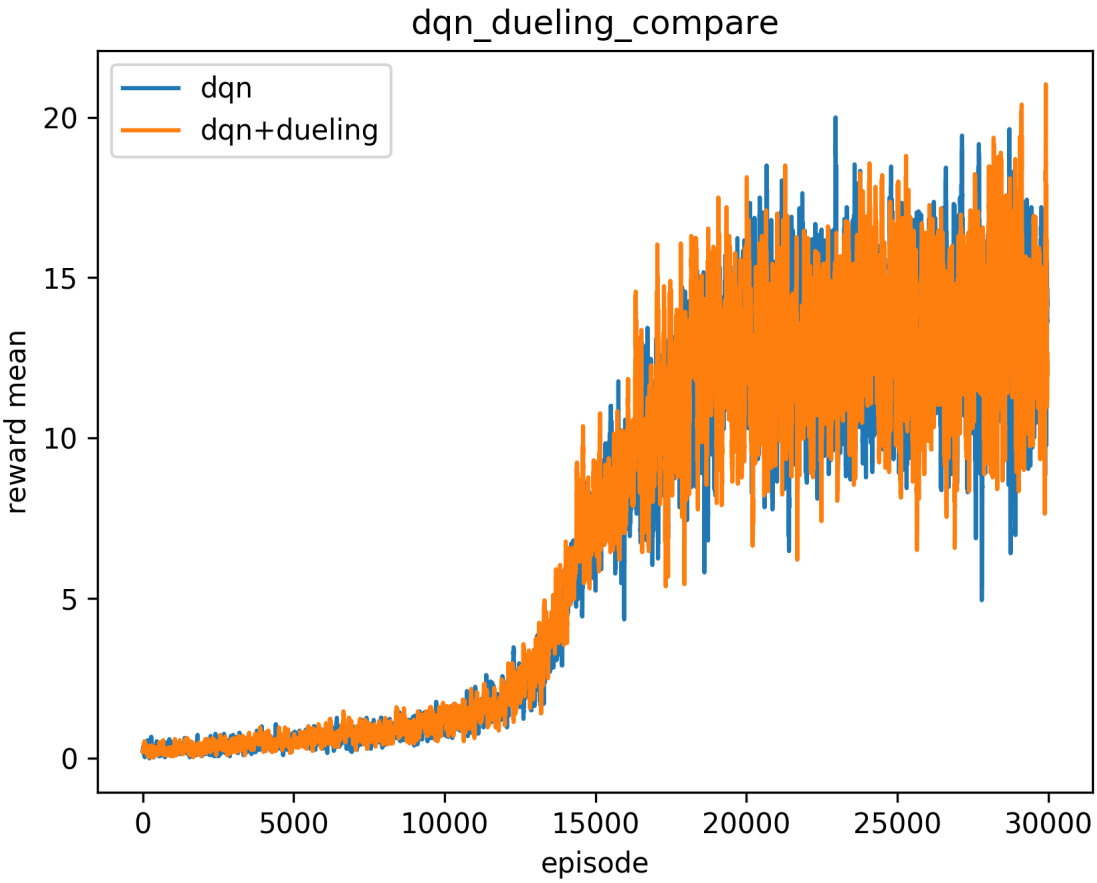
可以直接從圖中看出當 gamma 有微小的變化時，training 的結果也會有劇烈地影響，當 gamma=0.8 時，整個 model 幾乎學不太起來，等於是 model 對 Q_target_net 所 predict 出來的 Q_value 太過不信任，也就是 model 本身不太願意相信過去經驗的事實，預期得到的 Q_value 應該會比過去的經驗的再低一點點，也因如此導致整個 model 無法學起來，因為他太不相信過去的事實。當 gamma=0.9 時，相信的程度沒有 0.99 高，所以最後學出來的結果也比 0.99 低一些，但當 gamma=0.999 時，model 太過相信過去的經驗，給予過去的經驗過高的權重，導致，model 不容易學習到新的 state-action 的組合，導致 model 學出來的 reward 較低。

Double Q and Dueling DQN

實作 double Q 的作法其實就是將原本要餵進 Q_target_net 的 next_state 也同樣的餵給 Q_net，將 predict 出來最大 Q 值的 action 記下來，然後再將原本從 Q_target_net predict 出來的 Q 值中，取該 action 的 Q 值，而非像原本一樣直接取最大，而這樣做的好處是可以避免 over estimate 的問題，透過 Q_net 取 action 來減少直接取最大所造成的誤差，而實作之後的結果如下，可以看出加上 double Q 之後，reward 並沒有變好，反而變差，原因可能是因為 model 本身 over estimate 的問題並不明顯，也就是 Q 值並沒有衝很快導致 over estimate，所以加上 double Q 並不會比較好。



而實作 dueling，則是在原本 model 架構最後的部分，也就是 fc5 那層，將其抽換成 value net（512 -> 1）跟 advantage net（512 -> num_action），然後最後輸出 value +（advantage – mean（advantage））, 而 dueling DQN 的優勢便是，有時候在某些 state 下，不論做任何的 action 都不會對下一個 state 產生多大的影響，而 dueling 中的 value network 就是對這個 state 不受動作影響的 value 值，而 advantage 就是在這個 state 下，某些 action 可能會造成的影響，讓兩個 network 分別表示兩個狀態，可以讓 model 收斂的速度變快。下圖便是加上 dueling 之後的結果，可以看出加上 dueling 之後，model 並沒有明顯變好，收斂的速度差異也不大，可能是因為在 breakout 的遊戲中，任何的 action 都會對下個 state 有很大影響，導致 model 在 training 過程沒有變好。


dqn_dueling_compare
dqn
dqn+dueling
reward mean
episode
0
5
10
15
20
0
5000
10000
15000
20000
25000
30000