AI Final Project Report

1. Github repo link

https://github.com/hedy881028/AI-final-project

2. Introduction

Deep Q-learning 是將強化學習技巧之一的 Q-learning 與深度學習結合的技術，利用遊戲畫面當作 input，並利用 neural network (稱為 Deep Q Network) 取代原本的 Q-table，可以大大增加 Deep Q-learning 的穩定性及實用性。

我們的目的是使用 Deep Q Network (DQN) 來訓練電腦學習推箱子遊戲 (Sokoban)。推箱子遊戲是一款將所有積木推到特定位置的遊戲。我們這次使用的訓練環境是其他人搭建好的 gym 環境[1]。我們此次的目標為應用現有的 Deep Q-learning 技術，並且對一些超參數進行調整，希望能讓電腦學習推箱子遊戲的遊玩方式，並拿到盡量高的分數。

3. Related work

早期將 Deep Q-learning 應用在遊戲上的技術應該是 DeepMind 公司在 2013 年於 "Playing Atari with Deep Reinforcement Learning"[5] 提出的演算法。此技術將強化學習技術之一的 Q-learning 與深度學習結合，並且在許多遊戲上的實驗都獲得了顯著的成效，甚至在某些遊戲上超過了人類的表現。

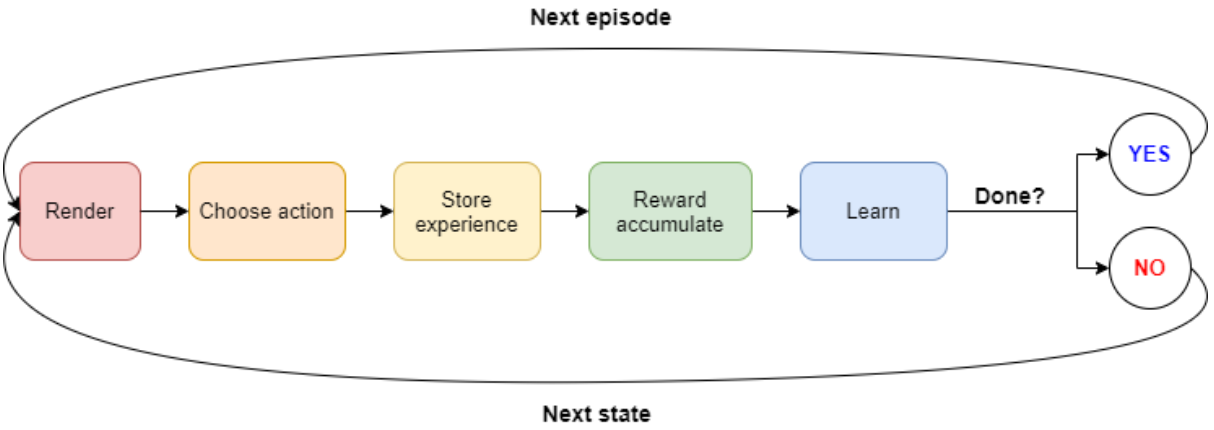
在 2015 年，因為發現了 DQN 的高估問題[6]，有學者提出了第一個解決辦法－Target network [7]。建立兩個完全相同的 network，一個用來控制 agent 及蒐集經驗 (稱為 Evaluation Network)，另一個用來計算 TD target (稱為 Target Network)，如此的算法可以減少原本 DQN 演算法中 bootstrapping [6] 的問題，進而減緩高估問題。

到了 2016 年，有學者對於高估問題提出了更進階的解決方法－Double DQN [8]。演算法跟 Target network 相似，一樣是建立 Evaluation Network 以及 Target Network，但是在計算估計值時，用 Evaluation Network 選擇最佳動作，之後用 Target Network 計算估計值，可以比 Target Network [7] 更有效的減緩高估問題。

4. Methodology

1) 流程圖

以下是sokoban訓練的過程，詳細的解說會在下面呈現。



2) Action

a) Action selecting

在 sokoban 裡面，action selecting 所採用的策略是 epsilon greedy。

epsilon greedy 是 RL 裡面很常用到的一種演算法，常常被用在 action choosing。model 在決定下一個 action 時，可以分為兩種策略。第一種是 exploit，它會讓 model 選取目前已知效果最好的 action。雖然乍看之下這是上上策，但其實可能會有更好的選擇，而因為 model 不知道，所以會錯過；第二種則是 explore，也就是隨機選取下一個action，在這個情況下，有可能model會找到比現在已知最好的 action 還要更好的action，但也有可能會採取一個效果更差的action。

決定 exploit 和 explore 的比例的參數就是 epsilon，當 epsilon 的值越大，model 採取 explore 策略的機率就越高。我們將 epsilon 設為 0.1，也就是 model 有 10% 的機率會採取 explore，90% 的機率則會使用 exploit 策略。

我們利用 np.random.uniform() 來決定 model 接下來應該要 explore 還是 exploit。np.random.uniform() 會 return 一個介於 0-1 之間的小數，當這個數小於 epsilon，也就是 0.1 的時候，model 就會隨機從 action list 裡選一個 action；反之，model會利用現有的 evaluation network 去選擇最佳動作。

```
# epsilon-greedy
if np.random.uniform() < self.epsilon:  # 隨機
    action = np.random.randint(0, self.n_actions)
else:  # 根據現有 policy 做最好的選擇
    # 以現有 eval net 得出各個 action 的分數
    actions_value = self.eval_net(x).cpu()
    action = torch.max(actions_value, 0)[1].numpy()[0]  # 挑選最高分的 action
```

b) Action list

原本的 action 共有九種：原地不動、sokoban 移動 (上下左右)、sokoban 推箱子 (上下左右)。但由於訓練成效不佳，我們便簡化了 action：

i. 拿掉原地不動

在 training 的過程中，我們發現 sokoban 太常選擇待在原地不動，導致遊戲狀態沒有任何改變。因此我們拿掉原地不動這個選項，等於變相強迫 sokoban 必須移動 (但也有可能會因為已經走到死路而被動的無法移動)。

ii. 合併移動和推箱子

在 sokoban 裡，假設右邊有一個箱子，但若選擇往右走而不是推右邊箱子的話，箱子不會被推開，sokoban 本身也不會移動。於是我們合併了推箱子和移動這兩種 action，假設 sokoban 選擇往右，若是右邊剛好有箱子，便執行推箱子的 action，反之則執行移動的 action。

3) Store experience

在進行完 action 之後，會產生一組 (s, a, r, s')，稱為 experience。我們會將每組 experience 存到固定大小的 buffer 裡面，若 buffer 滿了之後就取代掉最早之前放入的 experience。

此處我們利用了 experience replay 的技巧，也就是說每次要訓練 DQN 時，都從 experience buffer 裡面取 batch 去訓練，這個技巧的好處是打破每個 experience 之間的關聯性，讓訓練的樣本更加多樣化，使結果更加穩定。

4) Reward

以下是 Sokoban 的 reward 列表，並針對有修改的 reward 進行說明：

reward名稱	說明	修改前	修改後
penalty_for_step	每走一步的懲罰	-0.1	-0.1
penalty_no_move	Sokoban 移動完但狀態無改變的懲罰		-0.5
penalty_box_off_target	把一個箱子推離 target 所得的懲罰	-1	-3
reward_box_on_target	把一個箱子推到 target 上所得的獎勵	1	5
reward_finished	把所有箱子都推到 target 上，完成遊戲的獎勵	10	10

a. penalty_no_move

這是我們自己新增的 penalty，因為我們發現 sokoban 如果走進死路，還是會一直嘗試往死路的方向移動 (ex 已經走到這條路的最左邊，但還是一直選擇往左邊移動)。在一局最多只能執行 1000 個 action 的情況下，sokoban 會浪費將近一半的 action 額度在做無意義的行動。

b. penalty_box_off_target和 reward_box_on_target

修改這兩個 reward 的原因是因為我們認為 -1 和 1 太小，效果不夠顯著，所以我們讓這兩個 reward 變得更極端一點。

而讓 reward_box_on_target 變得比 penalty_box_off_target 高，或許能增加 sokoban 推箱子的意願。讓sokoban不會因為把箱子推離目標被扣太多分，而不敢推箱子。

5) DQN 網路架構

在將當前狀態 (遊戲畫面) 輸入 DQN 之前，我們會將狀態做一些先行處理。遊戲畫面為 160*160*3 的三維矩陣，先行處理的步驟如下：

i. 轉成灰階

對網路的學習來說，過多的色彩並不是必要的，因為即使遊戲畫面轉成灰階，仍然能夠辨識各項物品的所在位置。因此我們決定將畫面轉成灰階，以降低 input 的複雜度以及減少網路的計算量。

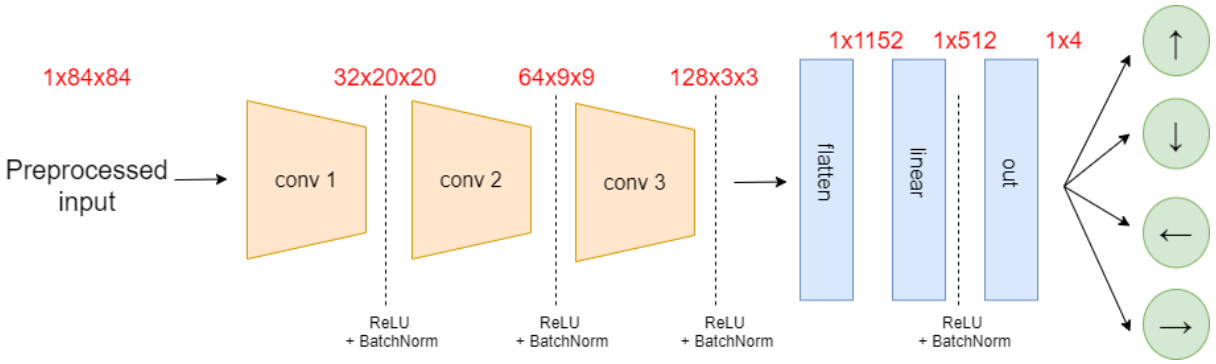
ii. 進行 **normalization**

第一個原因是這個步驟能夠將不同 input 的色彩上的差異去除，讓網路更專注在物體的位置關係上。第二個原因是 normalization 可以一定程度的改善網路的 vanishing gradient problem。

iii. 進行 **resize**

將 160*160 的畫面大小重新調整，變成 84*84 的 input

接下來介紹此次 DQN 的網路架構，以下為架構圖。



網路的輸入為先行處理完的遊戲畫面，首先會經過三層的 Convolution layer，每一層之間都有經過 ReLU 以及 Batch Normalization 的步驟。通過 Convolution layer 之後，會先經過一層 Faltten layer，將資料展開成一維的資料，接下來會經過兩層的 Fully connected layer，最後的輸出為 4 個分數，對應到 4 個可選的 action (向上移動、向下移動、向左移動、向右移動)

網路內各層的實作細節如下：

Layer Name	**Input → Output shape**	**Layer information**

conv 1	(1, 84, 84) → (32, 20, 20)	CONV-(N32, K8, S4, P1), ReLU, BN
conv 2	(32, 20, 20) → (64, 9, 9)	CONV-(N64, K4, S2, P0), ReLU, BN
conv 3	(64, 9, 9) → (128, 3, 3)	CONV-(N128, K4, S2, P0), ReLU, BN
flatten	(128, 3, 3) → (1, 1152)	FLATTEN
linear	(1, 1152) → (1, 512)	MLP-(N512), ReLU
out	(1, 512) → (1, 4)	MLP-(N4)

6) Learn

DQN 要學習的東西是判斷在狀態 s 的情況下, 哪一個動作是最好的 (Q-value最大), 也就是取代 Q-table 的功能。

要訓練 DQN 的最基礎演算法便是 TD 演算法, 簡單來說就是希望 DQN 預測出來的 reward 跟真實的 reward 相差越小越好。

此外, 我們還加上了的是 Target Network 的學習技巧, 也就是建立兩個完全相同的 network, 一個用來控制 agent 及蒐集經驗 (eval net), 另一個用來計算 TD target (稱為 target net), 如此的算法可以減少原本 DQN 演算法中 bootstrapping 的問題, 進而減緩高估問題。

以下解說我們這部分的核心程式碼:

```
# 計算現有 eval net 和 target net 得出 Q value 的落差
# 重新計算這些 experience 當下 eval net 所得出的 Q value
q_eval = self.eval_net(b_state).gather(1, b_action)
# detach 才不會訓練到 target net
q_next = self.target_net(b_next_state).detach()
# 計算這些 experience 當下 target net 所得出的 Q value
q_target = b_reward + self.gamma * \
    q_next.max(1)[0].view(self.batch_size, 1)
loss = self.loss_func(q_eval, q_target)
```

q_eval 是完全由 DQN 所算出的 Q-value, 而q_target 則是有由真實 reward 和 DQN 預測出下一個狀態的 Q-value 所相加, 而 TD 演算法就是希望這兩者的差距越小越好。

另外可以發現在計算 q_eval 的時候所使用的是 eval_net, 而計算 q_next 的時候是使用 target_net, 這裡便是 Target network 的學習技巧。

5. Experiments

原本 Sokoban 的規則是必須要將所有箱子推至目標位置之後才算是遊戲結束, 但我們此次先將「成功」定義為「至少將一個箱子推至目標位置」, 實驗出來的成功率約為 15 - 20%

以下是訓練過程的 reward 趨勢圖 (可忽略顏色相異):

6. Conclusion

a. 結果

雖然我們實作了以上所述的的細節, 但是結果不盡理想。Sokoban 並沒有逐漸進步, 有時候推得到箱子, 有時候什麼事也沒做。以下描述我們猜想的幾個導致無法成功訓練 sokoban 的原因。

i. Sokoban知道的資訊太少

在這個環境裡, sokoban 其實什麼都不知道。它不知道關卡長怎樣, 也不知道自己、箱子和目標的位置。在這樣的前提下, sokoban 能推到箱子其實都是運氣好而已。

此外, 遊戲的設定是多箱子跟多目標, 而且沒有配對箱子跟目標 (也就是任何一個箱子都可以被推到任何一個目標上)。一個環境可能會有很多種解法, 但也有可能只有唯一解。有時候 sokoban 看似有成功把箱子推到目標上, 但其實這樣的推法會影響它把其他箱子推到剩餘的目標上。在不知道任何東西的位置的前提下, sokoban 自然也不會知道它的推法是不是正確的。

ii. 隨機地圖

每次只要 sokoban 使用完它的 action 額度, 或是破解關卡後, 就會進入到下一關, 也就是一個隨機生成的地圖 (但只有不到 1% 的情況是因為破解關卡而進入下一關)。但是從上一點可以了解到, sokoban 其實什麼都不知道, 即使破關也只是偶然。在這樣的情況下, 每一次換關卡時, sokoban 面對的都是一個陌生的環境, 還沒有熟悉這個關卡就會被強制換到下一個關卡。我們認為這樣的設計有太多不確定的因素, 對 sokoban 的學習是不好的。

b. 未來可改進方向

針對以上我們猜想的問題, 我們提出了幾點可能可以改善 sokoban 的想法。

i. 拉長 training 時間

我們在網路上找了一些資料，發現訓練類似這樣的 model 通常需要訓練千萬個以上的 step，才能趨於穩定。但是由於記憶體容量的限制跟 GPU 運算的速度，我們只能訓練大概五百萬個 step。其實說不定 sokoban 是可以被成功訓練的，只是我們的硬體設備不允許。因此，我們認為拉長 training 時間或許可以讓 model 有一定程度的提升。

ii. 降低改變關卡的頻率

原本的 training 方法是，action 額度用完或是破關後，就會換到下一張地圖。但是我們認為這樣子的訓練方法，沒辦法讓 sokoban 在當前地圖裡學到足夠的東西。

我們的想法是，讓 sokoban 不論用了幾個 action，都一定要破解完這一關，才能換到下一張地圖。就跟人玩遊戲一樣，必須要親自成功過才能從遊戲當中獲得經驗，只是因為關卡太簡單而成功是無法學習到足夠的經驗的。而這樣子的方法自然也需要足夠的記憶體跟時間。

iii. 由簡入難

一開始的設定是多個箱子和多個目標，但這樣對於新手sokoban來說，可能太過於困難。或許應該要先從一個箱子和一個目標開始訓練，等穩定之後再慢慢增加箱子和目標的數量。

7. References

[1] https://github.com/mpSchrader/gym-sokoban

[2] https://github.com/uvipen/Tetris-deep-Q-learning-pytorch

[3] https://medium.com/pyladies-taiwan/reinforcement-learning-%E9%80%B2%E9%9A%8E%E7%AF%87-deep-q-learning-26b10935a745

[4] https://skywalker0803r.medium.com/%E8%A8%93%E7%B7%B4dqn%E7%8E%A9atari-space-invaders-9bc0fc264f5b

[5] arXiv:1312.5602v1

[6] https://www.youtube.com/watch?v=X2-56QN79zc&list=PLvOOobtloRntS5U8rQWT9mHFcUdYOUmIC&index=2

[7] Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* Human-level control through deep reinforcement learning. *Nature* 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[8] van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/10295