國立勤益科技大學資訊工程系實務專題進度報告

※※※※※※※※※※※※※※※※※※※※※※
※　　　　　　　　　　　　　　　　　　　　※
※　　　　　　　　　　　　　　　　　　　　※
※　　　　　　人工智慧黑白棋　　　　　　　※
※　　　　　　　　　　　　　　　　　　　　※
※　　　　　　　　　　　　　　　　　　　　※
※※※※※※※※※※※※※※※※※※※※※※

執 行 期 間 ：107學年度上學期至107學年度下學期

專題參與人員：劉彥震

部別：日間部

組別：第58組

班別：四訊三乙　座號：30　姓名：劉彥震

指導老師：　林灶生

中　華　民　國108年　　5月　　31日

目錄

首頁

人工智慧黑白棋
Artificial wisdom reversi

組員:劉彥震 指導老師:林灶生
Liu Yan Zhen, Jzau-Sheng Lin
國立勤益科技大學資訊工程系
Department of Computer Science and Information Engineering
National Chin-Yi University of Technology

摘要

這個專題是我學習alphazero論文的知識後，然後訓練一個黑白棋AI，接著使用影像辨識應用至遊戲上的一個實作，在此報告中會簡單講解alphazero的原理，以及我採取的神經網路架構、參數、MCTS、各種用於強化學習探索的公式，還有我是如何建立黑白棋棋盤的。

This topic is after I learned the knowledge of alphazero papers, then trained an Othello AI, and then used the image recognition application to a game implementation. In this report, I will briefly explain the principle of alphazero and the neural network architecture I adopted. , parameters, MCTS, various formulas for intensive learning and exploration, and how I built the Othello chess board.

前言

約一年前踏入deep learning的坑，一開始覺得AI很酷，但在學習的途中才發現這是一個相當艱深的課程，深度學習牽扯到很多的數學，而且資料的預處理也相當的困難，如何將一批的資料整理並輸入網路就有非常多的方式，好在深度學習已經是出來一段時間的東西了，現在網路的開放式課程相當的多，一天看一個章節也能有最基本的了解，數學公式網路上也有淺顯易懂的解釋，而在寫程式時也有很多東西要學，從基本的numpy套件到深度學習的tensorflow套件、以及影像辨識，我總共看了三本書，最後我才開始碰到深度學習中的強化學習，強化學習通常用在訓練遊戲AI上，而且還不需要預先準備訓練資料，訓練資料由AI自己產生，不但有趣也省了很多麻煩，因此我選擇了棋類的AI當作我的專題。

在一開始其實我是決定要做五子棋的，首先我做了一個15*15棋格的五子棋遊戲，並且配合強化學習中一個最基本的架構DQN，下去訓練15*15棋盤，然而不管訓練了多久，AI絲毫沒有學會任何東西，完全只會亂下，灰心之下在google上搜尋到了大神們是如何訓練的，這才想起來前幾年相當火紅的alphago，然而簡單了解後其訓練所需的時間、設備完全不是普通人所能承擔的，尤其是棋盤越大的規則越複雜的，不過好在2017年alphago團隊釋出了一篇新的論文alphazero，其所需的配備大幅降低，讓我又重新燃起了希望，並且我也將15*15五子棋換成8*8的翻轉棋(俗稱黑白棋)，於是這個專題就是我對於alphazero的一個了解並訓練後應用至遊戲上的一個實驗。

專題簡介

一.製作目的

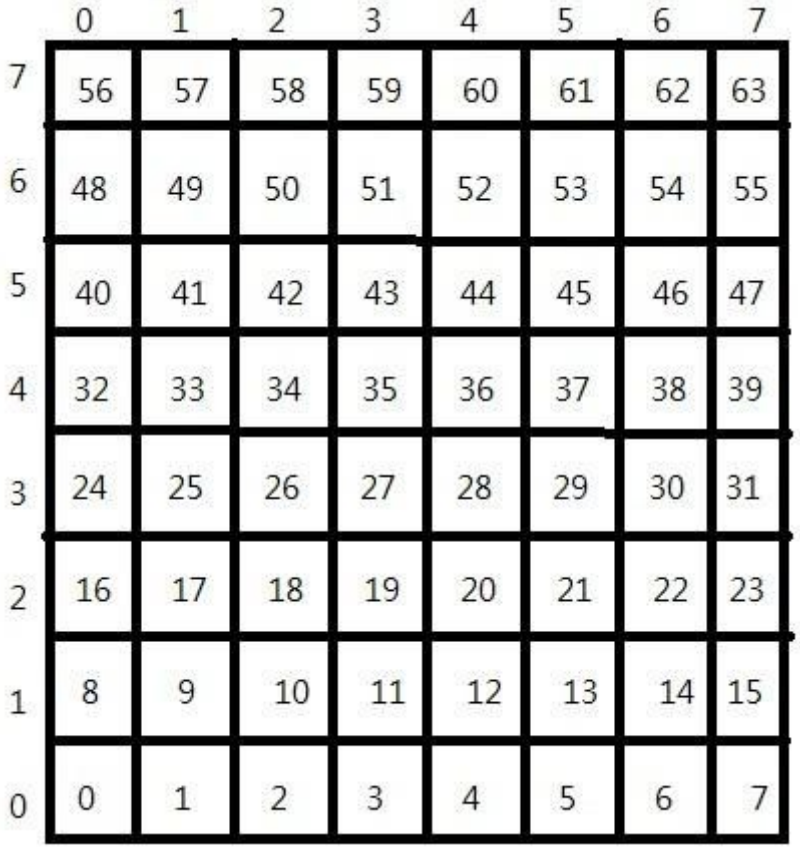
使用alphazero的方法訓練一個黑白棋AI，並且應用至遊戲上，希望至少能打贏電腦。

圖 1

二.方法

(一)棋盤

1.建立棋盤



	0	1	2	3	4	5	6	7
7	56	57	58	59	60	61	62	63
6	48	49	50	51	52	53	54	55
5	40	41	42	43	44	45	46	47
4	32	33	34	35	36	37	38	39
3	24	25	26	27	28	29	30	31
2	16	17	18	19	20	21	22	23
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

圖 2

如圖所示，這是我在程式中所建立的棋盤，8*8並且每個位子以0~63編號，後面皆以動作替代位子來表示。

2.黑白棋規則

棋盤共有8行8列共64格。開局時，棋盤正中央的4格先置放黑白相隔的4枚棋子（亦有求變化相鄰放置）。通常黑子先行。雙方輪流落子。只要落子和棋盤上任一枚己方的棋子在一條線上（橫、直、斜線皆可）夾著對方棋子，就能將對方的這些棋子轉變為我己方（翻面即可）。如果在任一位置落子都不能夾住對手的任一顆棋子，就要讓對手下子。當雙方皆不能下子時，遊戲就結束，子多的一方勝。

3.棋盤掃描

在最一開始以及每次下完棋後都要進行掃描以取得合法落子點，我的方式是每個棋子進行八個方向的掃瞄，掃描到合法落子點時，利用python的字典將沿途會被翻轉的對手棋儲存起來。

然而以我的棋盤表示方式，進行掃描時會碰到一些問題，例如動作30位於第3行第6列，往右上掃描時，可以很自然的使用列尾7-6=1，來進行只掃描一格的任務，然而如果是52往右上掃描，我們無法用列尾7來做判斷，必須改用行尾才行，因此在掃描時要將動作進行分類。

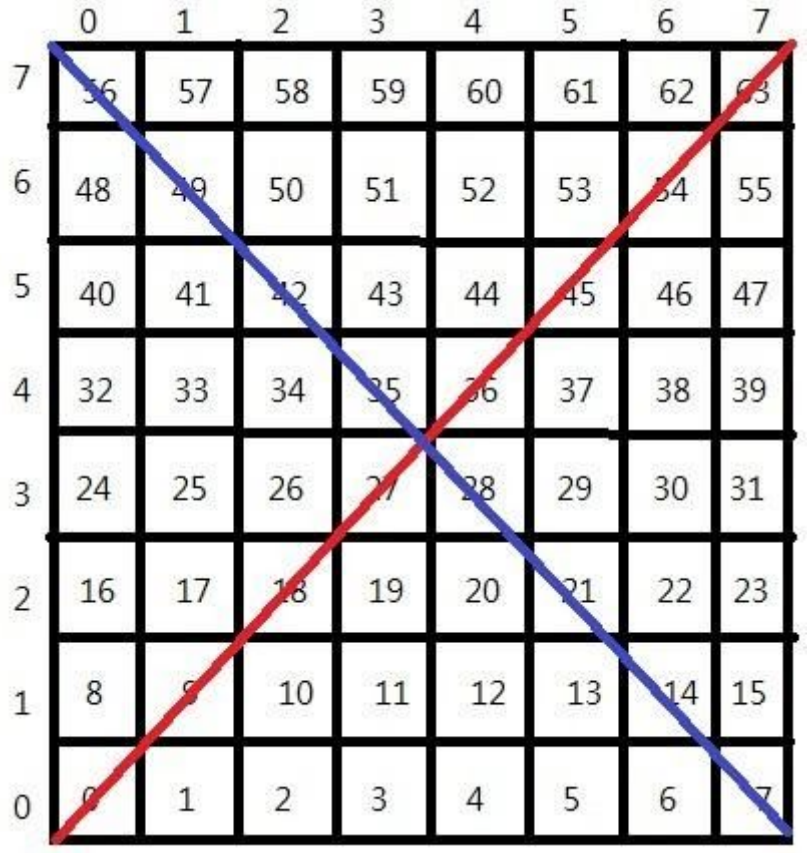


圖 3

如圖所示，經過推算，紅線分類可解決右上、左下掃描問題，藍線分類可解決左上、右下掃描問題。

(二)神經網路

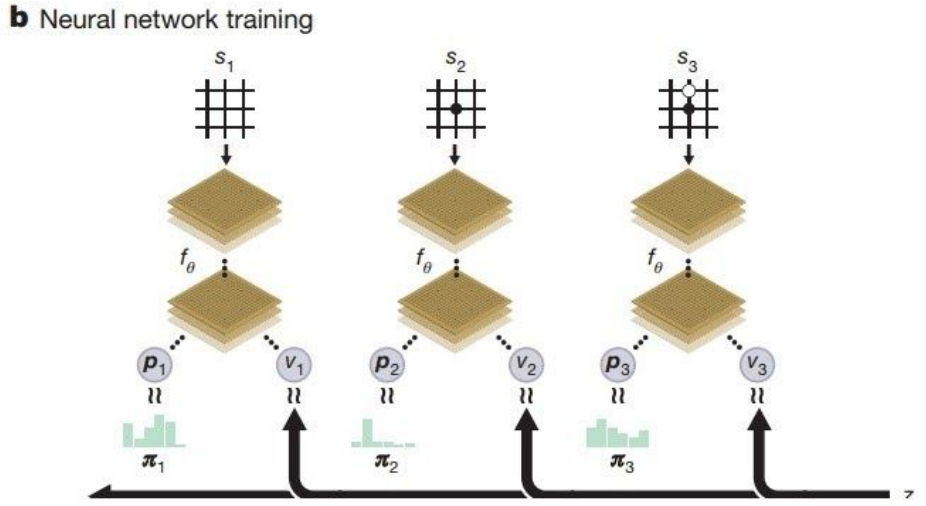


圖 4

Alphazero採用策略價值網路，輸入為當下棋盤的狀態，策略輸出為AI認為當前局面要下在64個動作的概率，價值輸出為AI認為當前局面是好還是壞。

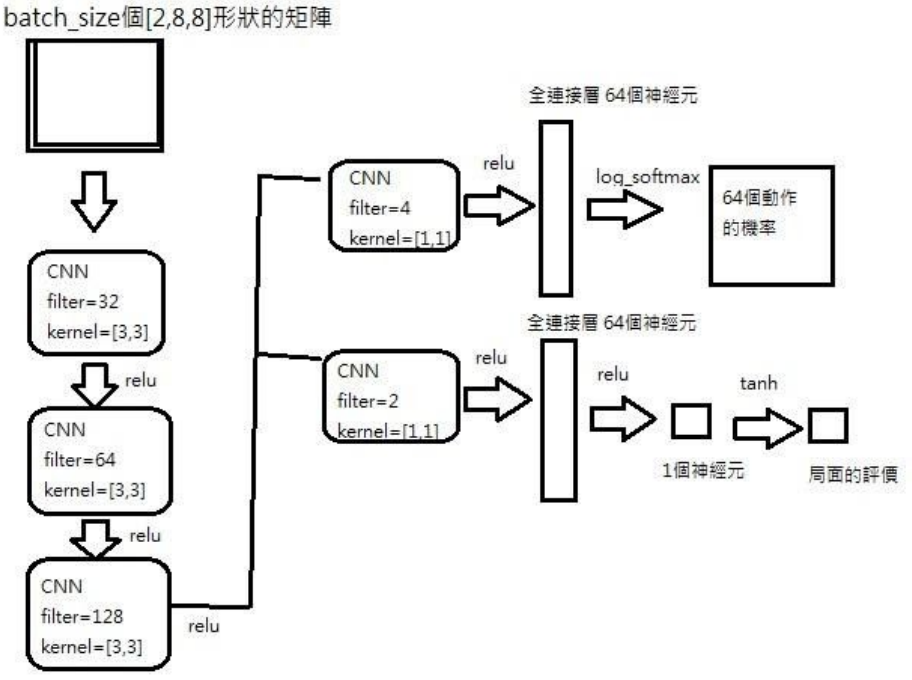


圖 5

然而Alphazero 所使用的神經網路架構還是太難訓練，因此我參考了網路上一個訓練8*8五子棋的架構，圖5為其架構，輸入局面表示為2張8*8的矩陣，分別表示為當前局面我方的棋子以及對方的棋子，而矩陣剛好適合使用CNN來進行輸出，經過3個CNN後分別輸入至策略端以及價值端，策略端經過1層CNN後再經過一層64個神經元的全連接層後輸出，價值端經過1層CNN後經過一層64個神經元的全連接層後再經過一個神經元輸出價值。

(三)MCTS蒙地卡羅搜索樹

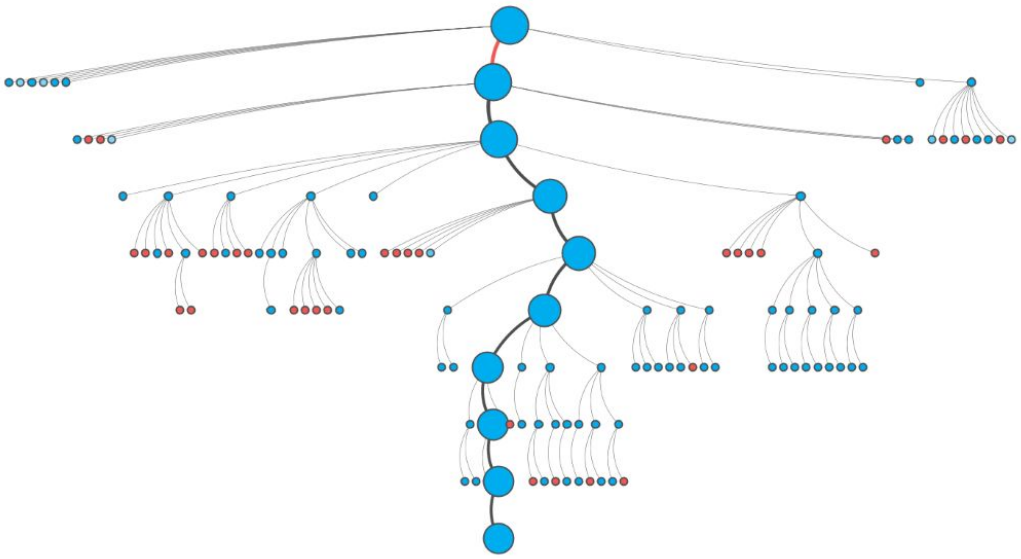


圖 6

MCTS為alphazero的主體，是一種搜索演算法，由於我們不可能計算所有的可能性(黑白棋空間複雜度為10^{28})，故需要盡可能的搜索找出當前局面下合法落子點的價值，以選擇動作。

所以MCTS在拿到當前局面後，便會開始模擬下棋，不只我方會下，還會幫對方下棋，模擬n次後，根據整顆樹的情況，決定合法落子點的輸出機率。

而在模擬下棋中，動作將會以節點的方式來儲存狀態，儲存的有輸出該動作的機率、動作的價值、拜訪次數、父節點、子節點

MCTS有三個步驟:

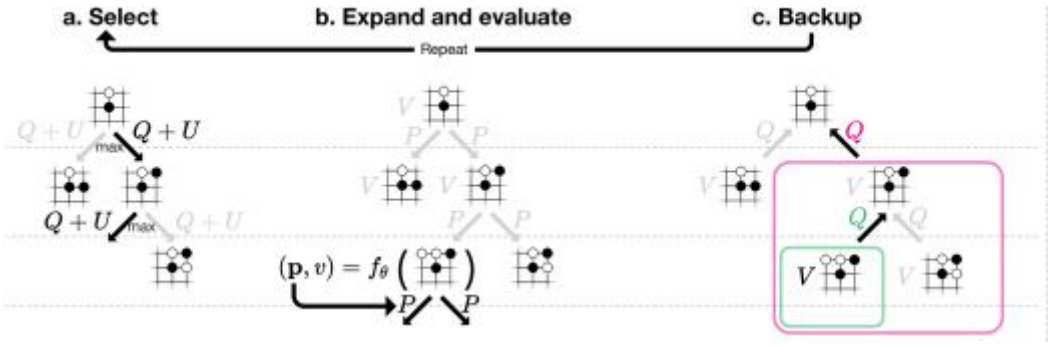


圖 7

1.Select

輸入當前局面，取得合法落子點，接著創建節點，假如此節點已被用過，要把指標指向最下面的根節點，接著在此根節點下選擇，擁有最大動作價值的合法動作，下在模擬的棋盤上。

2.Expand and evaluate

將模擬棋盤送入神經網路中，策略端取得64個動作機率，價值端取得當前局面的價值，接著在Select取得的動作節點下再拜訪所有64個動作節點，並將策略端取得64個動作的機率賦值上去

3.Backup

查看select的模擬下棋是否有造成遊戲結束，我方贏時代表我方節點是好的動作，所以全部更新正面價值，對於對方節點來說則是不好的動作，所以更新負面價值，我方輸時則相反，如果還沒結束，則使用價值端取得的局面價值進行更新。

以上便完成了一次的搜索，每步我將其設定為搜索400次，全部搜索結束後我們會回到整顆樹的最上方，即我們select的那一排的合法動作的節點，而因為是第一次select，代表它也是現實棋盤的合法動作，動作機率則是根據其拜訪次數進行處理後輸出。

(四)Self-play訓練

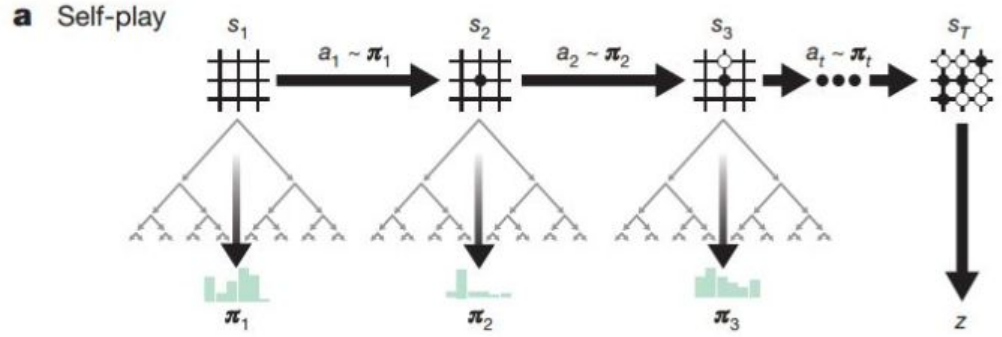


圖 8

有了三個主要的架構後，只要把它們串起來運行即可進行訓練了，訓練時在每次MCTS搜索完時要儲存3個東西S、π 、Z:

每次模擬時當前的局面S:

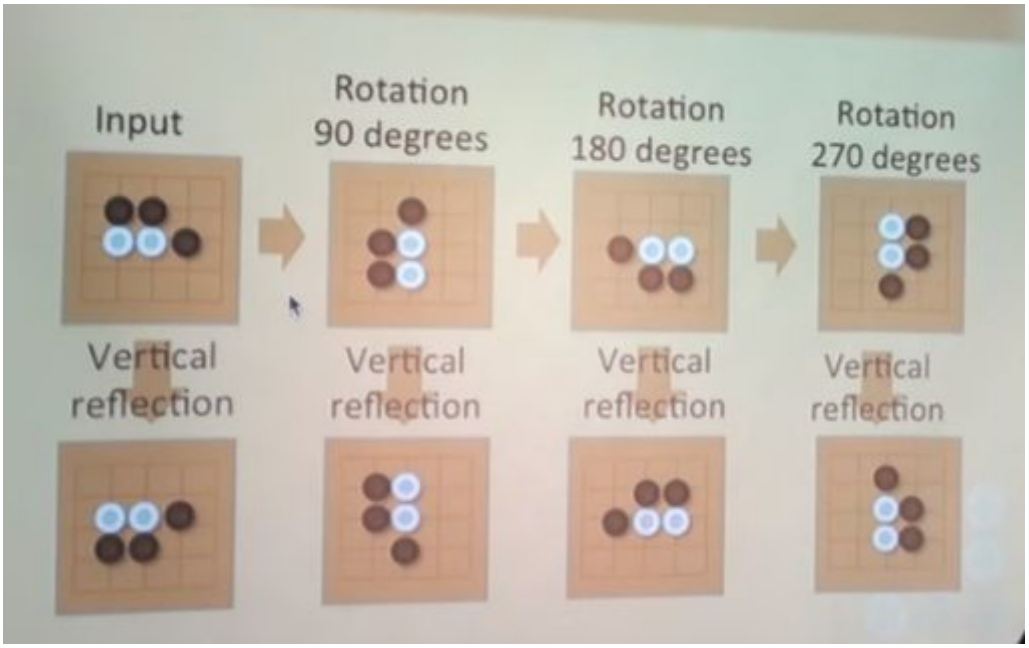


圖 9

由於黑白棋擁有旋轉以及鏡像翻轉的特性如圖所示，以上8種的局面是一模一樣的，這代表我們可以擴充訓練的資料至8張。

2.當前局面每個動作的機率 π

3.一張全1、0、-1分別代表贏、平手、輸的矩陣Z

S用於輸入神經網路，還原當時的情況，以此得到當時的策略、價值輸出接著 π 用於策略的損失函數，Z用於價值的損失函數，最後使用Adam優化器更新網路，根據下圖，由於我的神經網路較小，學習率我選擇一個中間值使用5e-3

Thousands of steps	Reinforcement learning	Supervised learning
0–200	10^{-2}	10^{-1}
200–400	10^{-2}	10^{-2}
400–600	10^{-3}	10^{-3}
600–700	10^{-4}	10^{-4}
700–800	10^{-4}	10^{-5}
>800	10^{-4}	-

Extended Data Table 3: **Learning rate schedule**. Learning rate used during reinforcement learning and supervised learning experiments, measured in thousands of steps (mini-batch updates).

圖 10

(五)影像辨識

(1) 擷取棋盤畫面

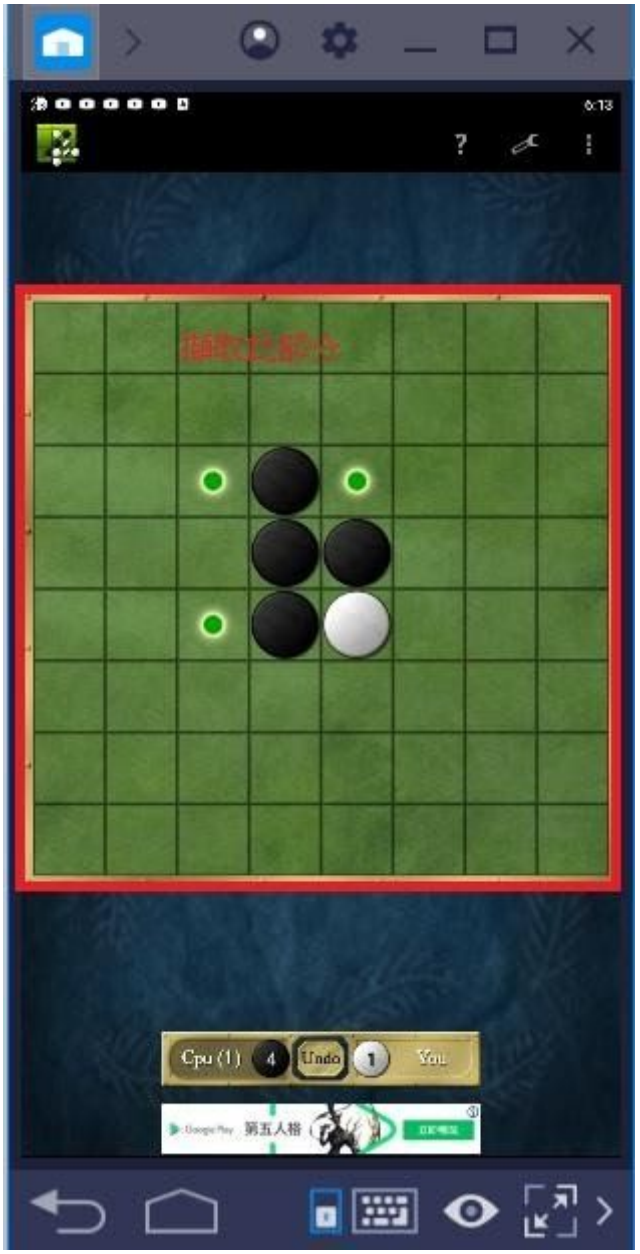


圖 11

使用python的win32gui套件擷取遊戲視窗的棋盤畫面,

(2) 將棋盤畫面二值化

圖 12

二值化是一種最簡單的圖像分割方法，將大於閥值的像素設為255，小於的設為0，從而實現圖12右邊兩個影像的效果，所謂灰度值就是從黑到白的等級，總共是0(黑)~255(白)，所以在二值化前通常要將圖片轉為灰度圖再進行二值化，但經我的實驗，只要調整閥值，直接用RGB圖下去二值化即可。

(3) 分割畫面並計算像素的頻率

依照擷取畫面的大小，將畫面分割成8*8格，接著使用一種分類資料的方法K-means 將像素顏色分為兩種。

接著計算頻率，所謂的頻率即是在這格中一種像素佔了多少比例，我們用程式計算出第二種顏色佔了多少比例。

(4) 將方格分類並取得AI輸出後顯示

從圖可以看到我左下角有設定一個閥值，根據調整頻率門檻來判斷此格是不是空格，接著再根據畫面是二值化後的黑棋還是白棋進行分類，白棋為1，黑棋為2，到此我們便取得了當前的棋盤狀態，接下來只需要將棋盤輸入進MCTS取得AI的動作，最後繪製顯示於畫面，在遊玩時我會看最左邊的圖是否更新到了最新的局面，接著按照AI給出的位子Here手動在遊戲上下棋。

三.結果

圖 13

已可以打敗7級的AI，約有9成的勝率，根據我自身遊玩的經驗，邊緣跟角落是黑白棋的重點，猜想AI最基本應該要很喜歡下邊緣跟角落。

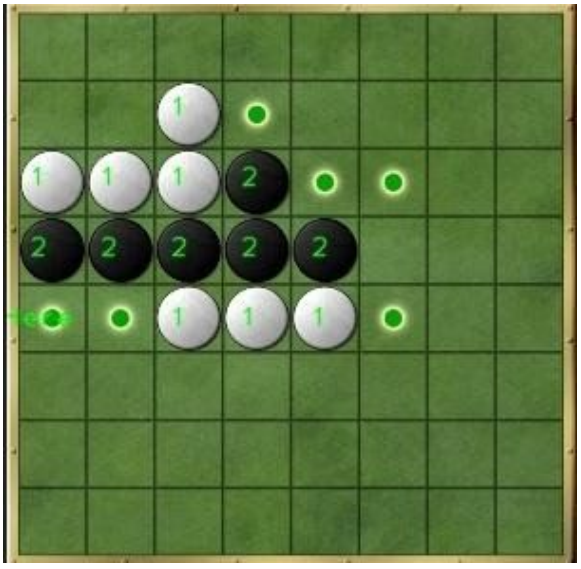


圖 14

圖 15

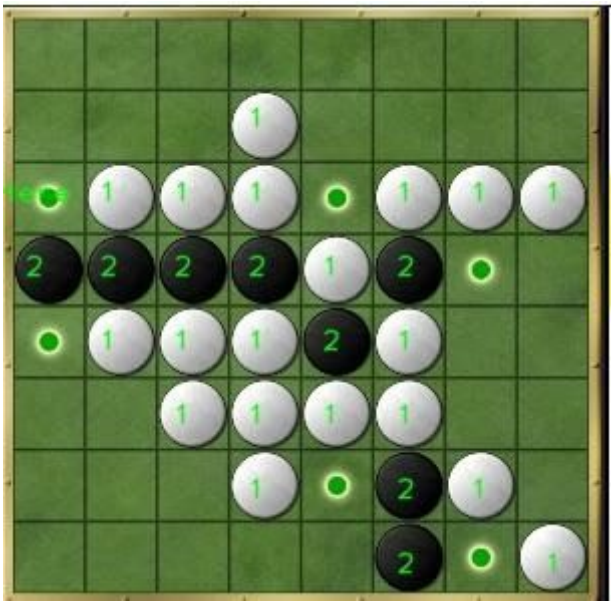


圖 16

根據好幾場的觀察(圖14、15)，AI是有學到的，然而卻導致圖16的狀況很常發生，當下在Here處時，其下方仍然有一個黑棋，很顯然的，只要黑棋方再往Here處上面下，便可奪回邊緣的優勢，這也是為什麼過不了8級電腦的原因，看來AI的棋力需要再進一步的提升。

製作理論探討

一.神經網路

(一)CNN捲積層

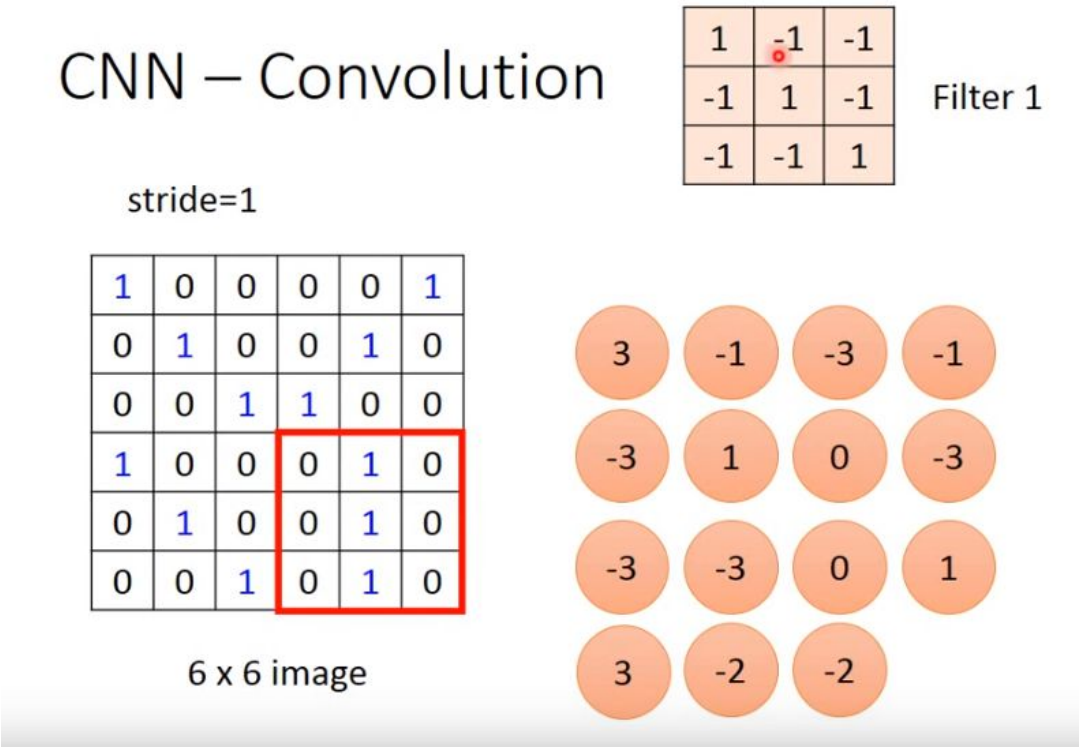


圖 17

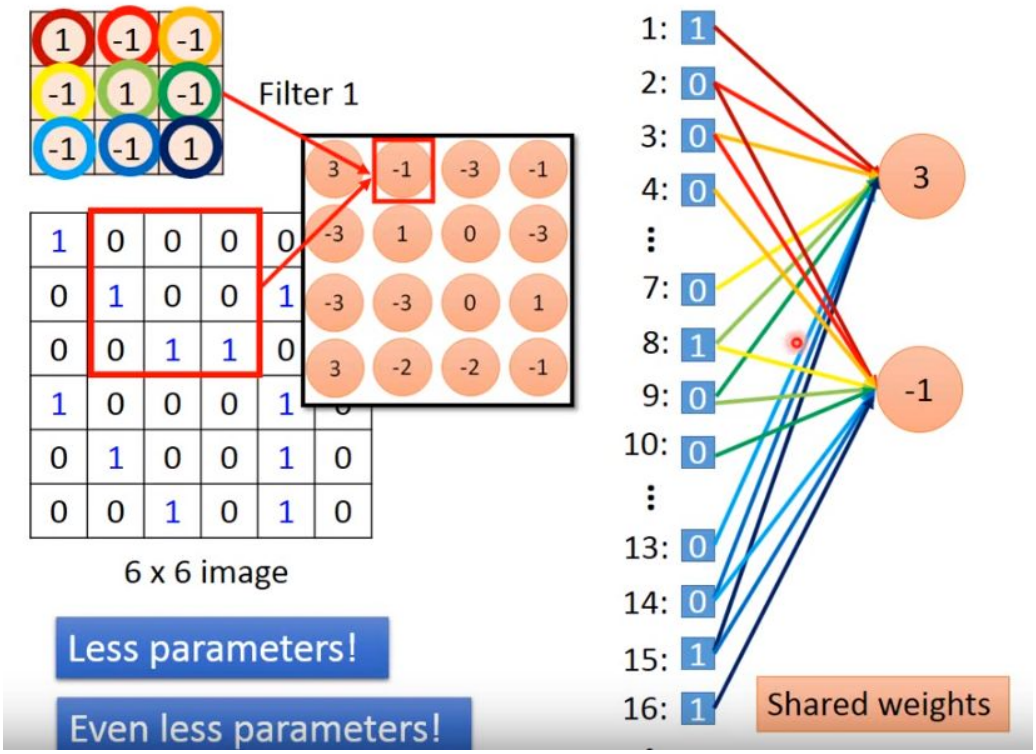


圖 18

簡單介紹一下CNN，在CNN中，有一個東西叫做filter，又叫做掃描器，它也可以看做是一個神經元，所以它自帶有參數，如圖18所示，filter大小為3x3，當一張6x6的圖進來時，filter將會以自身大小選取跟輸入同樣大小的一個區塊(例如圖中紅框)，接著進行矩陣相乘得到一個值，然後filter移動stride格一樣進行計算，如此將整個6x6的圖掃描後就成了右下角4x4的形狀。

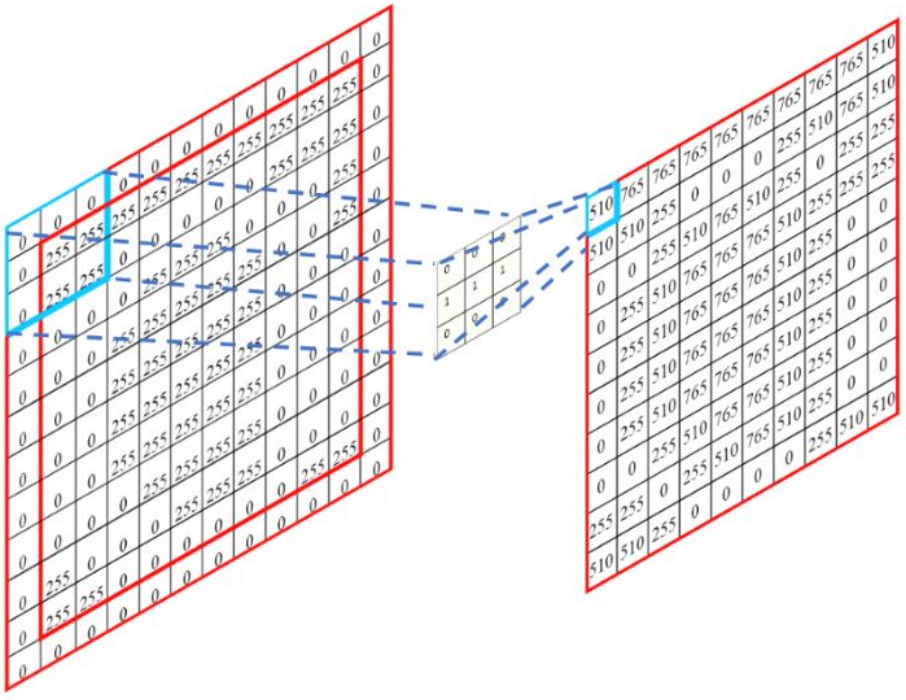


圖 19

CNN我還設定了參數: padding="same"，它的功用如圖所示，一般是為了解決stride步伐過大，導致無法完整掃描的問題，例如一張7*7的圖，如果stride為2，在移動三步後便無法繼續移動導致邊緣掃描不到，因此padding在邊緣上補0解決了此問題，但我的stride為1不會有這個問題，之所以添加是為了更好的獲取邊緣的資訊，因為按照CNN工作原理，不加padding的話邊緣通常只會被掃描到一次。

以上我了解到為何deepmind選擇CNN當作神經網路的主體，相比於直接創建64個神經元的全連接層作為主體，它更能夠獲取棋盤的資訊。

(二)Relu激勵函數,

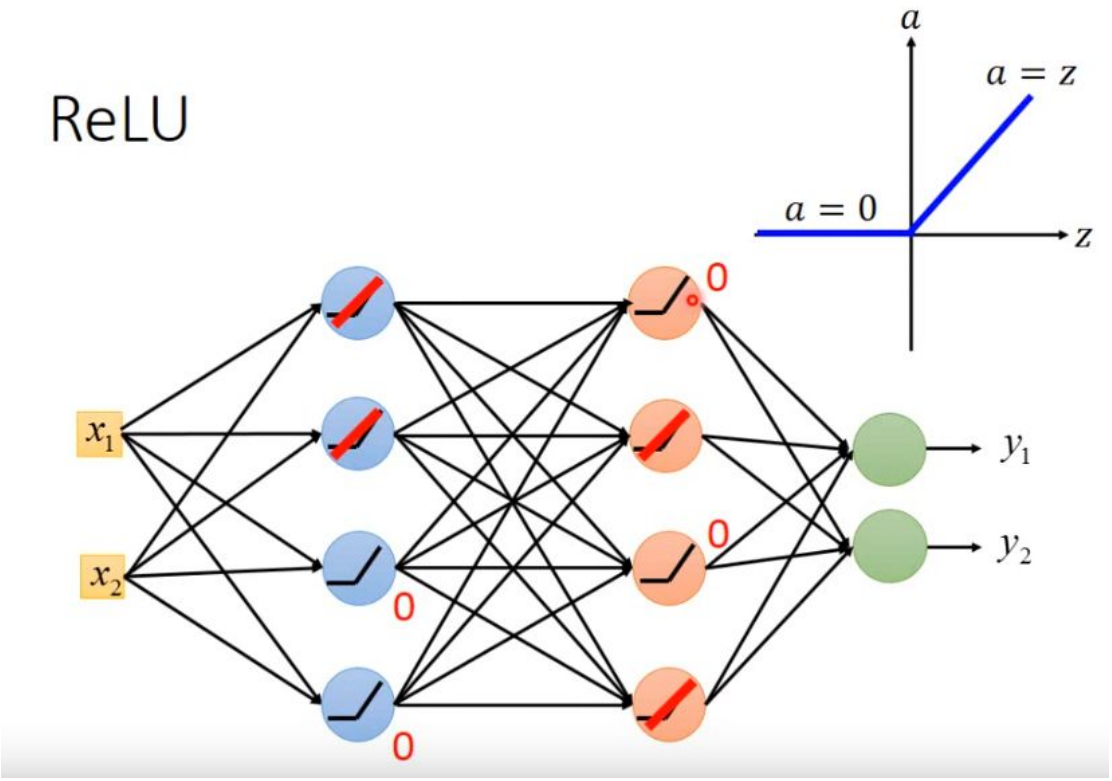


圖 20

Relu是一個在神經網路中很常被使用的一個激勵函數，它解決了其他激勵函數(如tanh、sigmoid)造成的梯度消失的問題，這是因為sigmoid會將大的輸入壓縮成小的輸出，如圖21所示，代表了如果輸入有很大的變化時，經過多層的傳遞之後，傳到損失函數早已消失了。

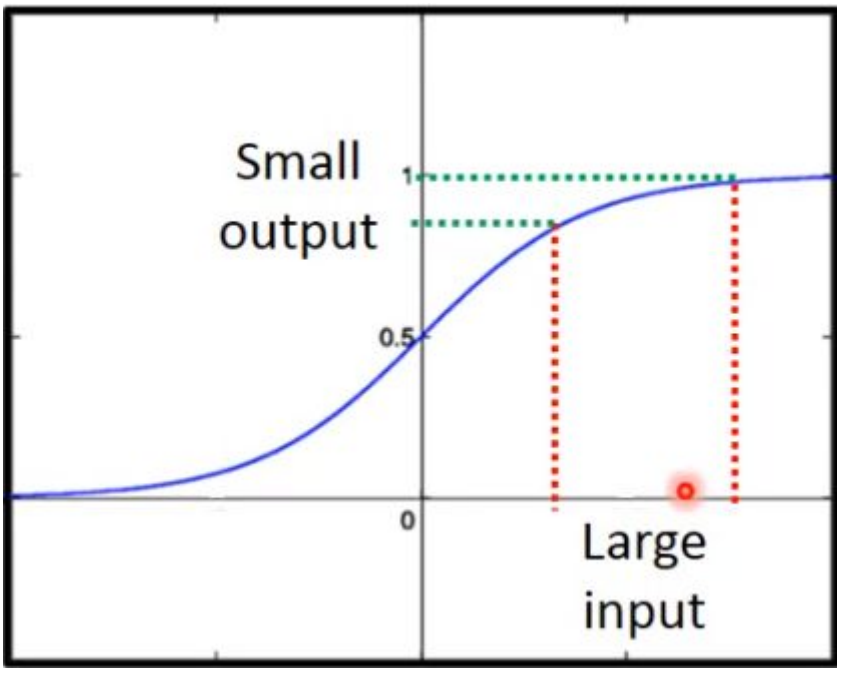


圖 21

因此圖20右上角，Relu激勵函數解決了這個問題，當輸入小於0時會直接輸出0，而大於0則是直接傳遞。

(三)Softmax激勵函數,

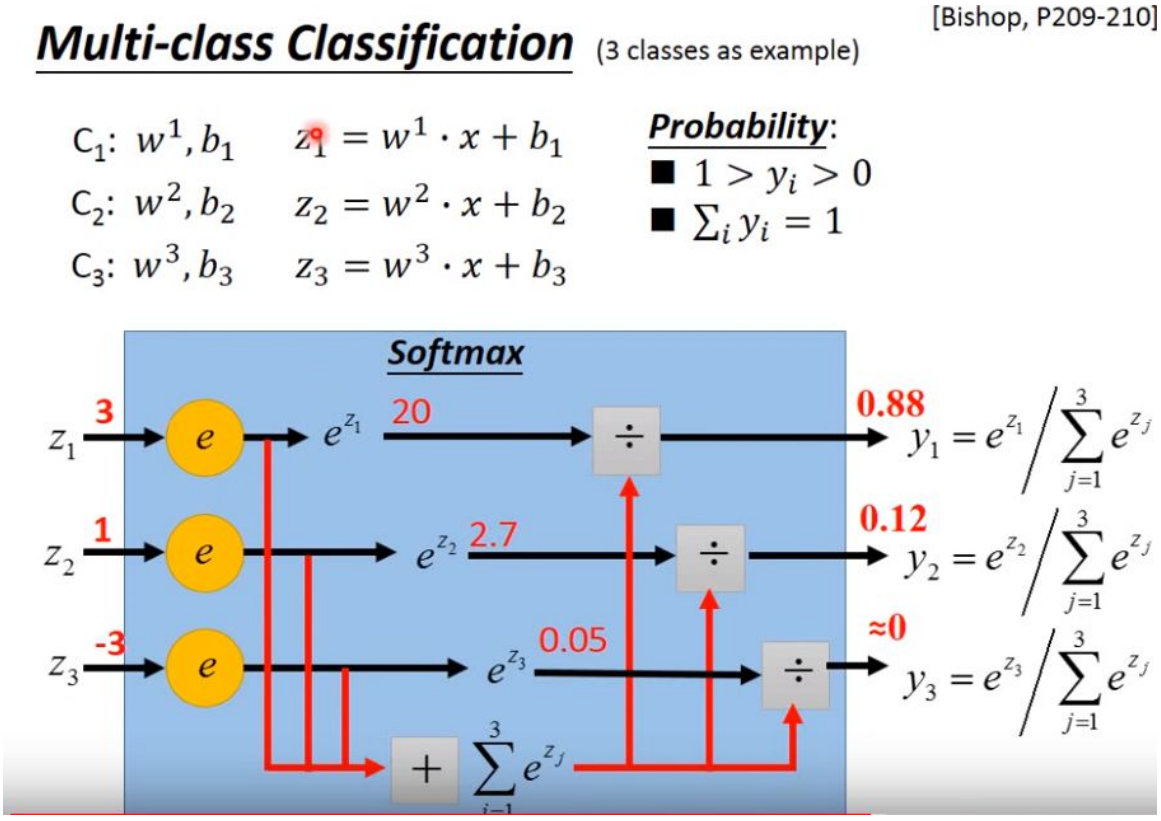


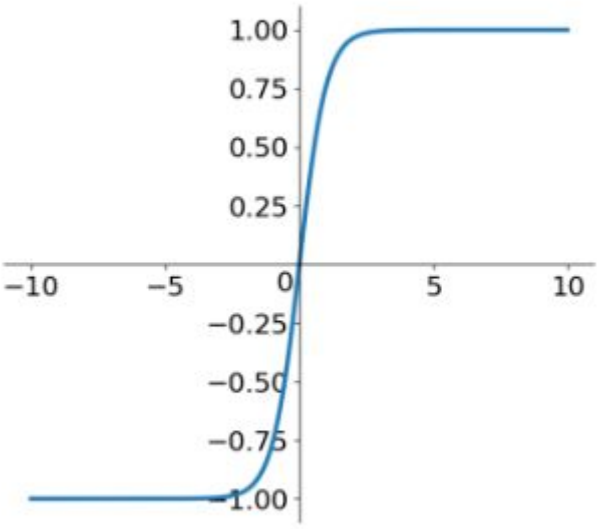
圖 22

在神經網路的策略端我們說到最後輸出是透過一個softmax得到的，如圖22所示，可以把神經

元所有的輸出壓縮在1~0內，並且總和為1，相當適合作為機率的輸出。

(四)tanh激勵函數

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



tan h

圖 23

如圖所示，tanh相當會將輸入壓縮成-1~1之間的值，相當適合拿來輸出局面價值。

(五)損失函數

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c||\theta||^2$$

圖 24

圖24為alphago在策略與價值網路中所定義的損失函數，依序為價值端、策略端的損失函數，最後一項則為L2 regularization。

1. 價值端: Z為下完一盤棋後根據贏、輸、平手所儲存的一個1、-1、0的值，做為價值端的正答案，V為價值端的輸出，損失函數使用均方根（root mean square）來表示，之所以使用均方根是因為直接z-v的話有可能會出現負數，這會導致損失函數一下負一下正，損失難以進行更新。

2. 策略端: π 為MCTS模擬出一步時的動作機率，P則是策略端的輸出，損失函數使用cross-entropy來表示，所謂的cross-entropy主要是表示資料的不確定性，當我們每個動作的機率都很平均時，此值會相當的大，當機率偏向某一個動作時會小，由於我們就是希望AI能夠學會某些特定落子，例如在黑白棋中，AI應該要很看重邊緣及角落的合法動作，理所當然的這些動作的機率應該會大於其他動作很多，因此以cross-entropy來表示。

3. L2 regularization:它是神經網路中才用來解決輸入對輸出敏感的問題，來讓神經網路平滑化，當在loss時加入L2正則化時，當輸入變化了一個 Δ xi，輸出的變化會是wi* Δ xi，而當w小時(通常w都不大)，就代表了輸出的變化很小，如此輸出便會對輸入不敏感，擁有了抗雜訊的能力。

$$y = b + \sum_{\text{all}} w_i x_i$$

$$+w_i \Delta x_i \qquad +\Delta x_i$$

圖 25

圖26為其實際的公式，透過計算梯度，並且代入更新w權重後推導可知，每次在權重更新前，w會先乘上(1-學習率* λ)，是一個小於1的數，如此來減少w更新的量， λ 為L2的自定義參數，在alphago裡使用1e-4。

Regularization

L2 regularization:

$$\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$$

- New loss function to be minimized

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2}\|\theta\|_2$$

Gradient: $\frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda w$

Update: $w^{t+1} \rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta\left(\frac{\partial L}{\partial w} + \lambda w^t\right)$

$= (1-\eta\lambda) w^t - \eta \frac{\partial L}{\partial w}$

Weight Decay

Closer to zero

圖 26

(六)Adam優化方法

由於Adam是由兩個優化方法RMSprop以及Momentum得來的，故要先了解這兩個優化方法。

1.RMSProp

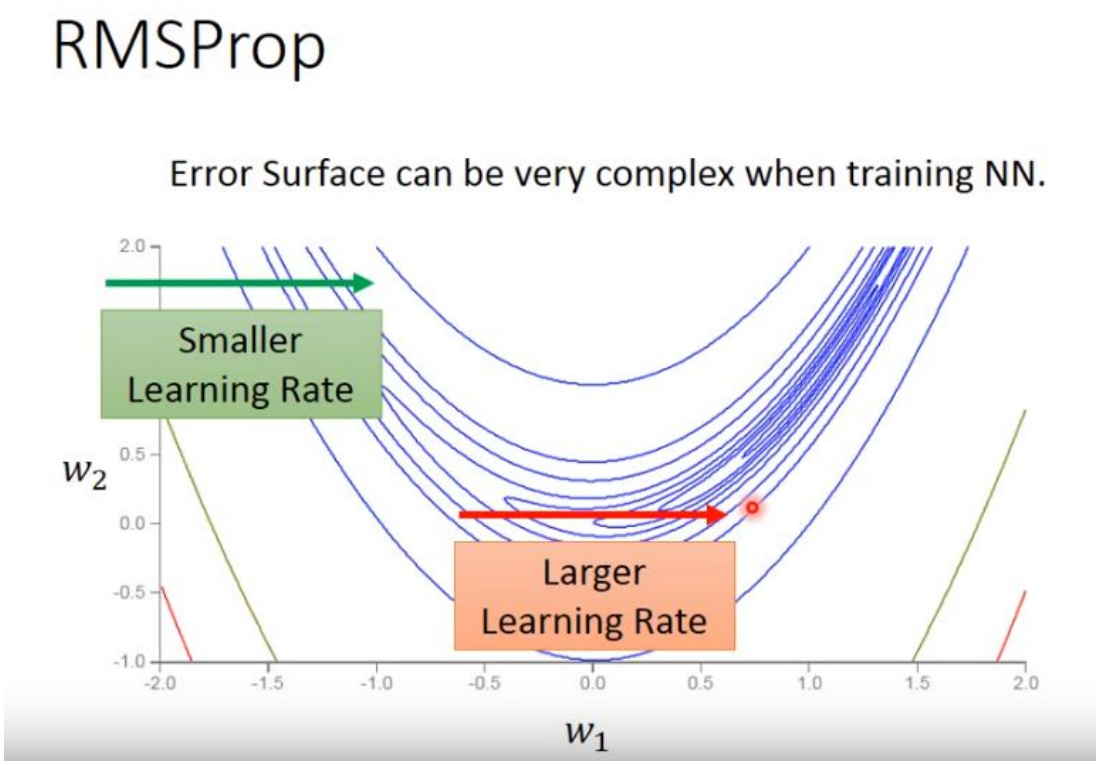


圖 27

RMSProp是為了解決在更新時遇到梯度變化的問題，如圖27所式，以w1進行考慮，綠色的線從梯度圖上來看，圈圈離很遠代`表較為平坦，需要較小的學習率，而在紅色的線卻突然變得非常陡峭，需要大的學習率，同樣的參數變化，梯度卻有了相當大的改變，為了自適應，所以RMSProp就是加上了考慮過往的梯度的算法。

RMSProp

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

圖 28

如圖28所示，在更新參數時，學習率會除以一個 σ 值， σ 值代表了過往的梯度，並且我們可以自定義 π 值來決定要相信新的梯度還是過往的梯度。

2.Momentum

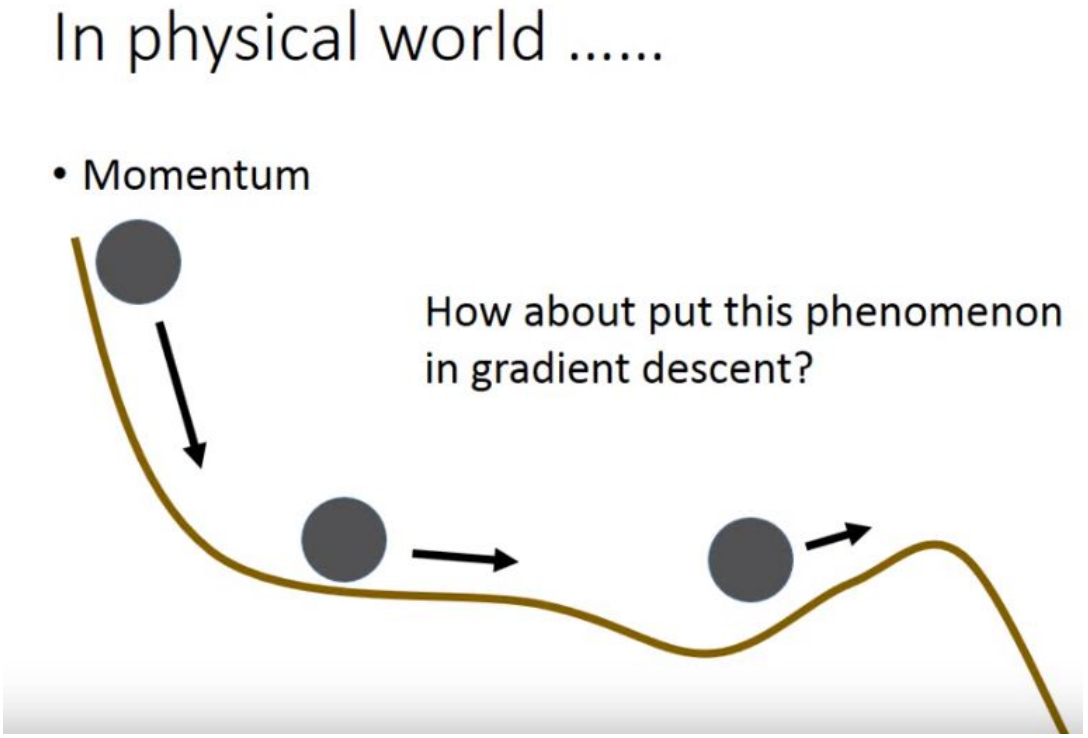


圖 29

顧名思義，Momentum就是加入了動量的算法，記錄過去移動的方向以及大小，來考慮下一步該怎麼更新。

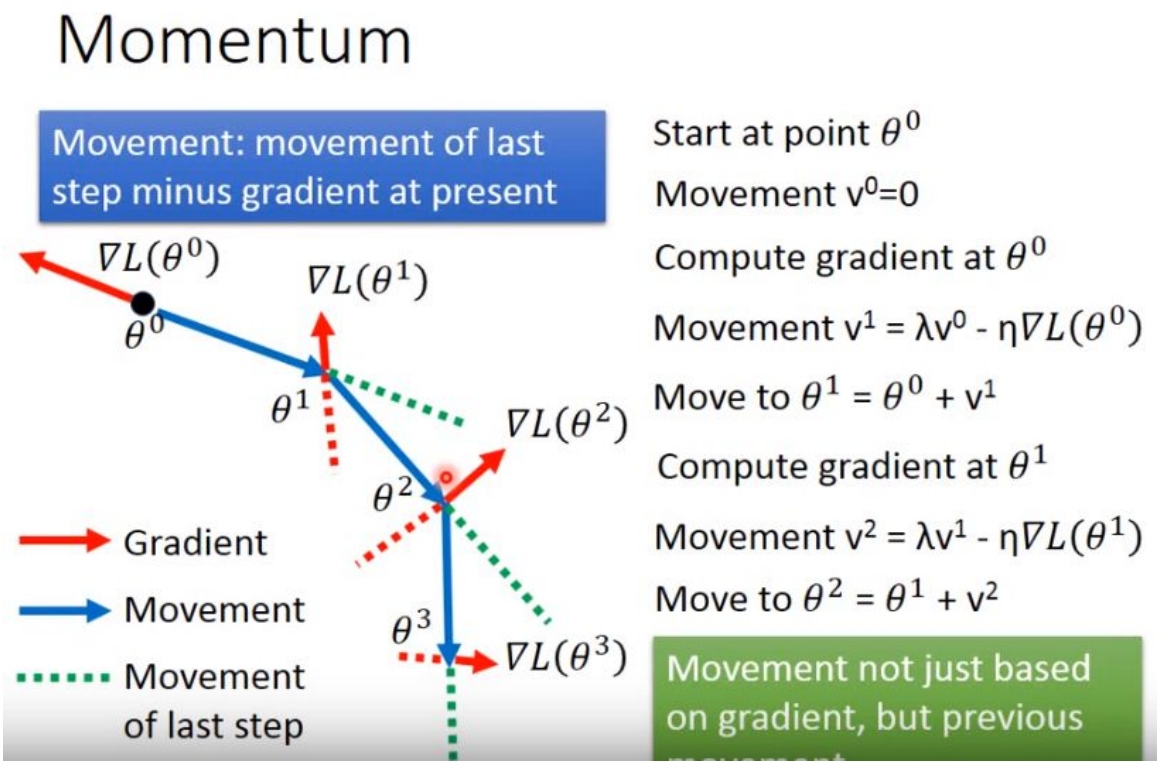


圖 30

因此在更新參數時加入參數v代表動量，原本要更新的是紅線的反方向，然而考慮動量後移動的卻是綠線的方向，在計算v時，會加入上一次的v，並且用自定義的 λ 控制權重。

於是Adam就出來了，同時考慮這兩個因素的優化方法

Adam　RMSProp + Momentum

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum
$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

圖 31

比較特別的是由於動量m以及過往梯度v一開始初始化為0，Adam作者計算出會有問題，因此在倒數2、3行，動量m以及過往梯度v除以了(1-變數 β)，做為偏差修正。

二.MCTS蒙地卡羅樹

在簡介中，我並沒有提到蒙地卡羅樹的一些公式，以及強化學習都有的探索問題，以下在這邊進行補充。

(一)多臂吃角子老虎Multi-arm Bandit問題

吃角子老虎機 (bandit) 是一種賭場常見的機器，玩家將硬幣投入後拉下拉桿，接著會隨機出現不同圖案，如果停止時出現符合相同或特定相同圖案連線，則可以根據賠率得到特定的報酬 (reward)。

因為報酬是隨機的，我們常以「期望報酬」（expected reward，概念上指的是玩吃角子老虎機非常多次後得到的平均報酬）去思考吃角子老虎機問題。

多臂吃角子老虎機 (multi-armed bandit)，指的則是很多台吃角子老虎機給玩家選擇，每一台機器可以得到的期望報酬皆不一樣。站在玩家的立場，目標應該是透過機器的選擇，在遊戲中獲得最大「期望報酬」。

所以想要獲得最大報酬，勢必得去探索其他台吃角子老虎機，而不能一直開發同一台，所以如何兼顧探索與開發便是多拉桿吃角子老虎機策略的核心問題。

(二)PUCT算法

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

圖 32

在蒙地卡羅樹中，每個節點的動作價值主要由Q值以及U值組成，Q值在簡介的backup中已經提過，而U值公式如圖32，P為該節點的動作機率，N(s,b)為父節點的拜訪次數，N(s,a)為子節點的拜訪次數，cpuct為控制此項的權重，透過這公式可知，當父節點拜訪次數越大，子節點的拜訪次數越少，則U值便會越大，透過拜訪次數決定價值，顯然是MCTS中探索的一個方法Cpuct在alphago的論文裡是設為1，而我設為了5。

(三) dirichlet狄利克雷噪點

在select之後，我們會從神經網路得到64個動作的機率，在這alphago將其25%的機率改為從狄利克雷分布中採樣的機率作為替代，此也是一種探索的方法，狄利克雷分布是根據先驗訊息所給出的一個機率分布，例如投擲硬幣，我們都知道正反的機率為50%，然而如果投了三次恰好全都是正面呢?，總不可能直接3/3=100%為正面，故狄利克雷分布便是建立在已投擲過50次正面50次反面的事件上，再加入3次正面所計算出來的機率分布。

Dirichlet noise Dir(α) was added to the prior probabilities in the root node; this was scaled in inverse proportion to the approximate number of legal moves in a typical position, to a value of α = {0.3, 0.15, 0.03}

Alphago中選擇了以0.3、0.15、0.03作為α參數(棋盤越大越複雜則該值越小)，而動作有64個，就代表了我要從一個已經投擲了0.3次的一個64面骰子，得到64個骰子的機率，我透過實驗將其改成投擲1000次，結果每個機率都很接近1/64 = 1.5，而為何是0.3場，由於論文中並沒有提到，故我想應該是他們實驗出來的。

(四) Temperature

act_probs = softmax(1.0/temp * np.log(np.array(visits) + 1e-10))
再MCTS完成n次搜索後，最後我們會回到整棵樹的最上方，取得合法落子點以及其機率，這邊就是對這些機率再動手腳，同樣是探索的手段，在alphago中，self-play實際下棋的前30步temp為1，之後設為0，代表不再探索，使用自己最強的選擇來輸出，而我是一直設為1，乘以log是因為策略端的損失函數同樣輸出也是log，接著透過softmax輸出機率。

三.訓練流程圖

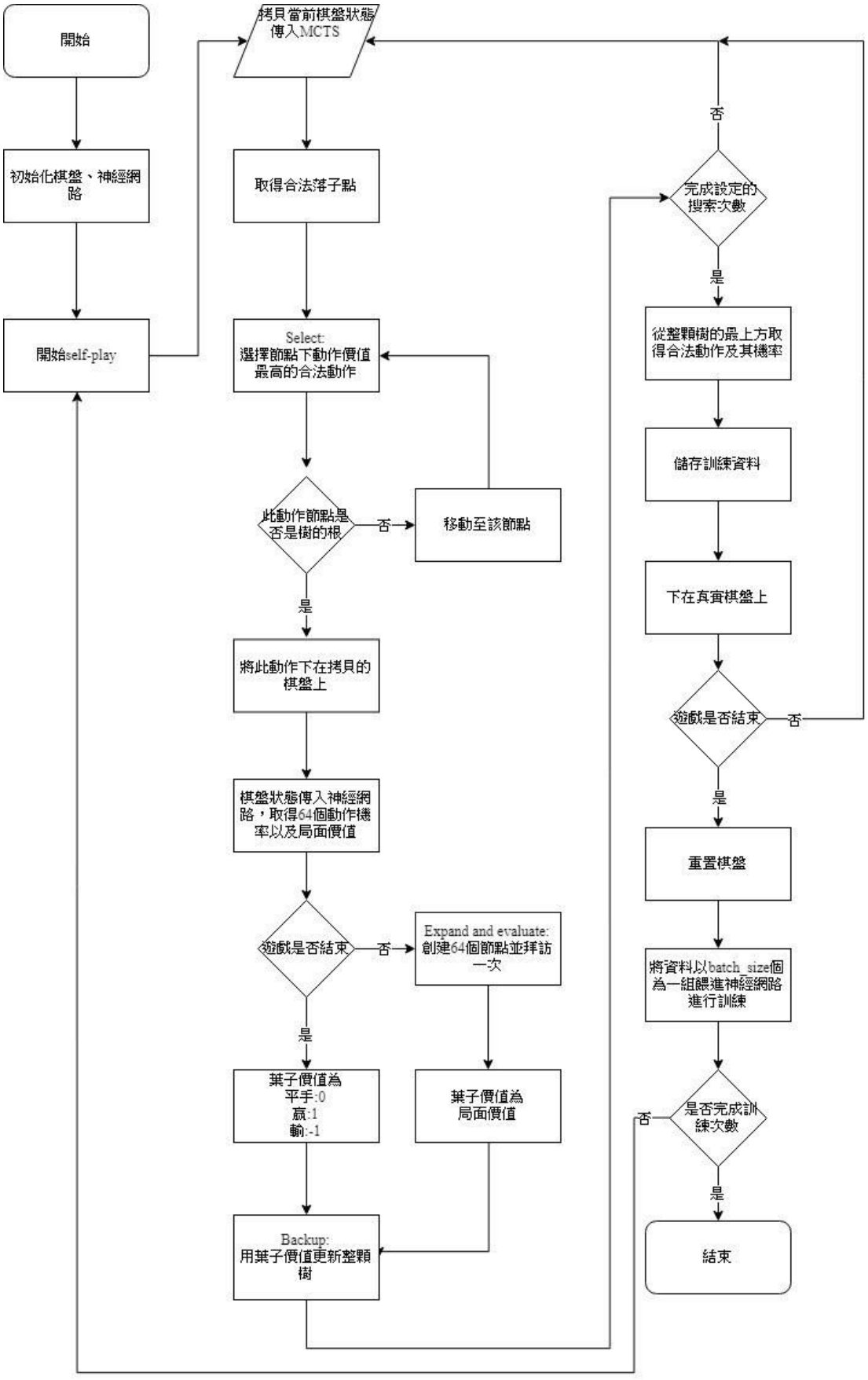

開始
初始化棋盤、神經網路
開始self-play
拷貝當前棋盤狀態傳入MCTS
取得合法落子點
Select:
選擇節點下動作價值最高的合法動作
此動作節點是否是樹的根
否
移動至該節點
是
將此動作下在拷貝的棋盤上
棋盤狀態傳入神經網路，取得64個動作機率以及局面價值
遊戲是否結束
否
Expand and evaluate:
創建64個節點並拜訪一次
是
葉子價值為
平手:0
贏:1
輸:-1
葉子價值為
局面價值
Backup:
用葉子價值更新整顆樹
否
完成設定的搜索次數
是
從整顆樹的最上方取得合法動作及其機率
儲存訓練資料
下在真實棋盤上
遊戲是否結束
否
是
重置棋盤
將資料以batch_size個為一組餵進神經網路進行訓練
是否完成訓練次數
否
是
結束

圖 33

四.遊玩流程圖

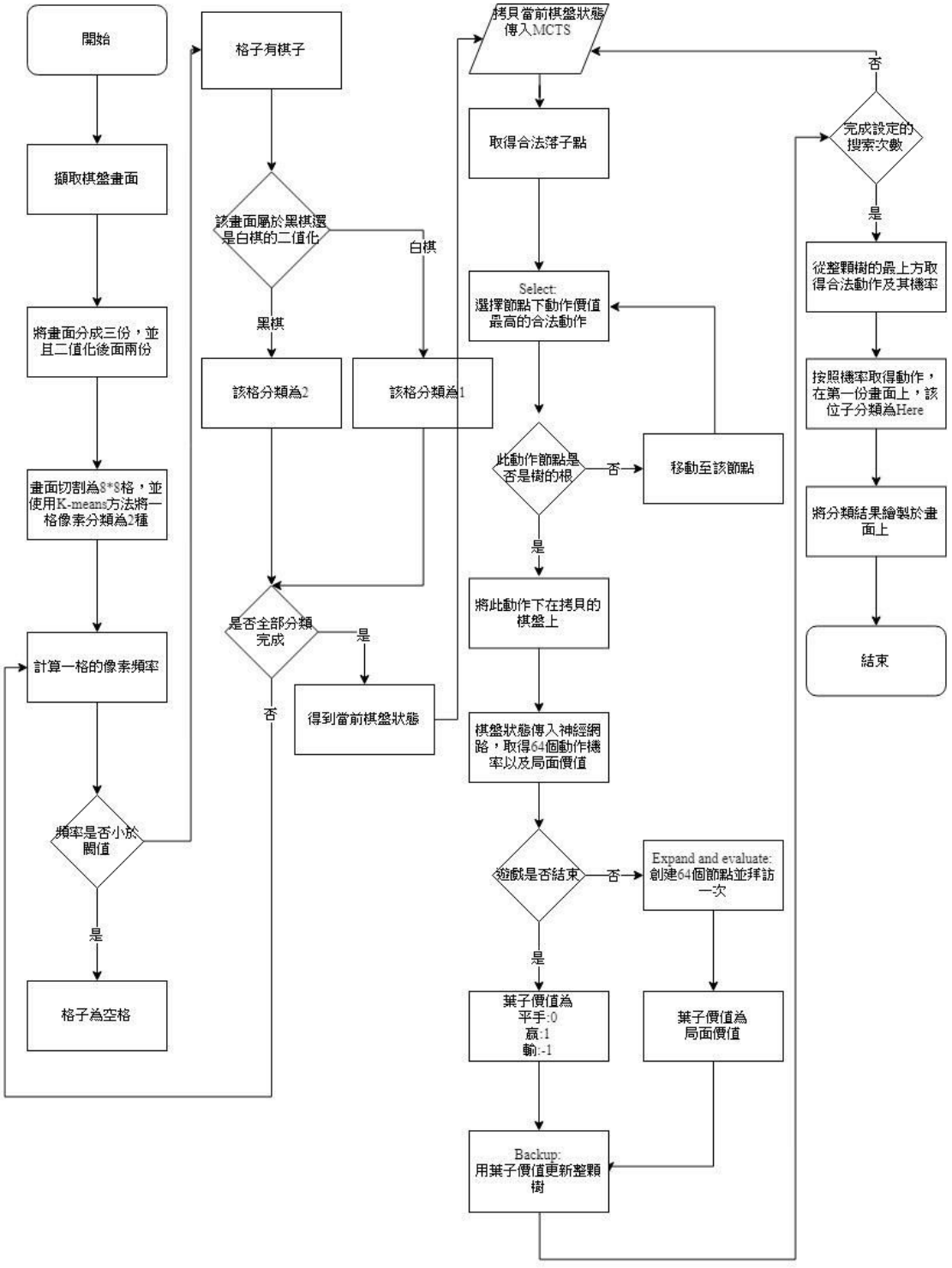

開始
擷取棋盤畫面
將畫面分成三份，並且二值化後面兩份
畫面切割為8*8格，並使用K-means方法將一格像素分類為2種
計算一格的像素頻率
頻率是否小於閾值
是
格子為空格
否
格子有棋子
該畫面屬於黑棋還是白棋的二值化
黑棋
白棋
該格分類為2
該格分類為1
是否全部分類完成
是
否
得到當前棋盤狀態
拷貝當前棋盤狀態傳入MCTS
取得合法落子點
Select:
選擇節點下動作價值最高的合法動作
此動作節點是否是樹的根
否
移動至該節點
是
將此動作下在拷貝的棋盤上
棋盤狀態傳入神經網路，取得64個動作機率以及局面價值
遊戲是否結束
否
Expand and evaluate:
創建64個節點並拜訪一次
是
葉子價值為
平手:0
贏:1
輸:-1
葉子價值為
局面價值
Backup:
用葉子價值更新整顆樹
否
完成設定的搜索次數
是
從整顆樹的最上方取得合法動作及其機率
按照機率取得動作，在第一份畫面上，該位子分類為Here
將分類結果繪製於畫面上
結束

圖 34

軟體分析

由於程式頗大，在這裡我選擇說明一些我認為比較難的程式

一.掃描棋盤

首先我的棋盤主要以字典的方式儲存棋盤內容，key為動作0~64，值為數值，2(黑棋)或1(白棋)。

```
for move in [27,36]:
    self.states[move] = self.players[1] #黑棋
for move in [28,35]:
  self.states[move] = self.players[0] #白旗
```

掃描儲存的方式也是字典，儲存為self.availables，key為合法落子點，值為會被翻轉的黑棋
掃描的步驟如下

1. 取得一顆棋子，查看它是不是當前玩家，是的話才需要掃描
2. 取得其行列
3. 判斷它屬於哪類(圖3)，以決定掃描方式
4. 掃描第一顆判斷是不是我方棋、空格，不是代表這顆是對方棋，將此棋儲存至store_opponent串列裡，接著繼續掃描
5. 直到掃描到空白處，途中如果掃描到我方棋則離開掃描，或是掃描到了終點
6. 掃描到空白處後，查看是否重複掃描(已建立此合法落子點)，是的話將沿途掃描到的黑棋加入至該字典，不是則建立該值於字典當中。
7. store_opponent清空，繼續下一個方向的掃描

```
    def scanning_for_availables(self):
        self.availables={}
        store_opponent = []
        for m in self.moved: #看每個已做過的動作
            w = m // self.width #取整數 看動作在第幾列
            h = m % self.width #取餘數  看動作在第幾行 以設定究竟要檢查幾格 解決超出邊疆的問題
            h_class=0
            w_class=0
            right_upperleft=0

            if h >= w :
                h_class = 1
            elif h < w :
                w_class = 1
            if w > self.upperleft[str(h)][0] :
                right_upperleft = 1
            player= self.states[m] #查看這個動作在state上是上面哪一個玩家的符號
            if player == self.current_player: #如果是當前玩家的旗子 則開始掃描該棋子以找出可下的點
                #往右檢查
                for i in range(m+1,m+(7-h+1)):

                    #首先檢查這顆是自己的棋以及檢查第一顆是不是空白處 都代表不需要繼續掃描 跳出迴圈
                    if (self.states.get(i,-1) == self.current_player) or
(self.states.get(i,-1) == -1 and i == m+1):
                        break
                    elif self.states.get(i,-1) == -1: #接下來繼續掃描 如果碰到空白處 此處即是可下
的點(因有黑棋間隔著)
                        if self.availables.__contains__(i):#此句在確認是否重複掃描 沒有的話
availables[i]會被蓋掉
                            self.availables[i].extend(store_opponent) #以此可下的點為首 與沿途
掃描到的黑棋 存成字典
                        else:
                            self.availables[i]=store_opponent
                        break
                    else:
                        if (self.states.get(i,-1) == self.current_player):
                            break
                        else:
                            #如果是對手棋 就把對手的棋子加入到
                            store_opponent.append(i)
                store_opponent=[]
```

圖 35

二.建立MCTS

節點使用python的class建立，每個都有其方法、資訊

```
class TreeNode(object):
    def __init__(self, parent, prior_p):
        self._parent = parent
        self._children = {}  # a map from action to TreeNode
        self._n_visits = 0
        self._Q = 0
        self._u = 0
        self._P = prior_p #預設是1.0 因為最初初始化時最初的節點"選擇率"肯定是100%
```

圖 36

```
    def select(self, c_puct):
        """Select action among children that gives maximum action value Q
        plus bonus u(P).
        Return: A tuple of (action, next_node)
        """
        #max函數:首先key是一個函數會對self._children.items()進行處理
        #1.一開始_children為空的 呼叫items會返回一個空的字典 而正常_children是表示父節點的"下一排"所有
可能的子節點 因為我們要選擇哪一個為下一個子節
        #2.key是一個function 會套用到_children.items()上 呼叫_children.items的get_value
        #3.就會得到這個_children.items最新的Q值
        #4.運行max選出Q值最大的 選擇他 return

        return max(self._children.items(),
                   key=lambda act_node: act_node[1].get_value(c_puct))
```

圖 37

```
    def update(self, leaf_value):
        self._n_visits += 1
        self._Q += 1.0*(leaf_value - self._Q) / self._n_visits

    def update_recursive(self, leaf_value):
        if self._parent:
            self._parent.update_recursive(-leaf_value)
        self.update(leaf_value)

    def get_value(self, c_puct):
        """
        也就是計算這個節點的value  Q值即被visit的次數
        """
        self._u = (c_puct * self._P *
                   np.sqrt(self._parent._n_visits) / (1 + self._n_visits)) #np.sqrt返回平方
根
        return self._Q + self._u

    def is_leaf(self):
        """Check if leaf node (i.e. no nodes below this have been expanded)."""
        return self._children == {}

    def is_root(self):
        return self._parent is None
```

圖 38

進行搜索時:

```
    def _playout(self, state):
        node = self._root #得到最初的節點
        while(1):

            if node.is_leaf(): #如果到達最後的葉子則break (偵測當前的node是不是下面沒有children了)
              #那為什麼下面還可能會有children呢? 因為我們設定MCTS 2000次 也就是每次都會從"頭"開始搜索
所以有可能選擇到上次選擇的move 直到選到葉尾的棋子
                break
            # Greedily select next move. 運行MCTS的select部分
            action, node = node.select(self._c_puct)
            state.do_move(action)
        action_probs, leaf_value = self._policy(state)
        end,winner = state.game_end() #改成scanning_number來偵測剩多少棋子來判斷是否結束
        if not end:
            node.expand(action_probs)
        else:
            # for end state，return the "true" leaf_value
            if winner == -1:  # tie
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == state.get_current_player() else -1.0
                )
        node.update_recursive(-leaf_value) #這邊是更新select的節點 因為此點是代表我們父點動作的對手
所以價值要更新負的
        #所以也就是從根到select的整根樹枝都會更新
```

圖 39

測試結果

為了提高棋力我調整了很多參數進行實現，在最初，我參考8*8五子棋的方式，棋盤狀態用4個局面表示，前兩個為雙方的棋子分布，第三個為對方最後一步的位子，第四個為表示當前是我方還是對手方的一個全1或全0的矩陣，以五子棋的思維，第三個局面是因為我們通常會下在對手剛下的位子附近，第四個則是因為五子棋對於先後手方有極大的優勢，再經過3天的訓練3000場後，損失函數來到了2附近，然而我將其進行互相對打發現1500場後棋力便沒再提升了。

由於黑白棋不一定下在對方最後一步附近、先後手也沒有特別大的優勢，因此我選擇只以前兩個局面進行實驗:

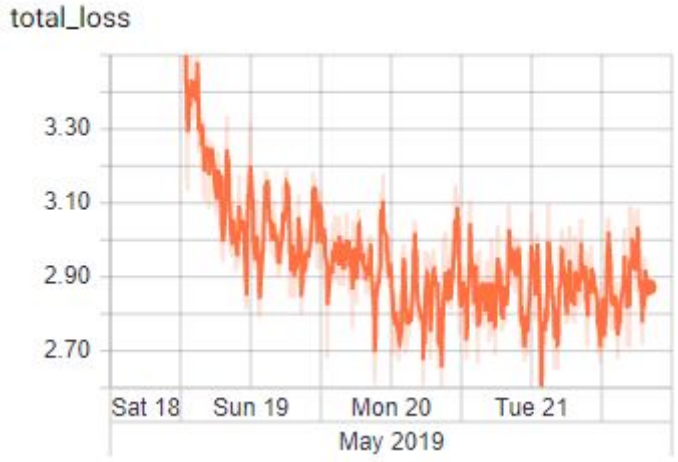


圖 40

可見總loss上升了0.6左右，然而一樣再訓練約1500場之後loss便卡再了2.8左右，同樣約為1500場，實際測試依然1500場後便無法再提升，將其與4個局面的AI對打後，發現旗鼓相當，這代表了2個局面是可行的。

猜想，可能是因為神經網路太小了，目前便已是它的極限，因此我再加了一層128個filter的

CNN。

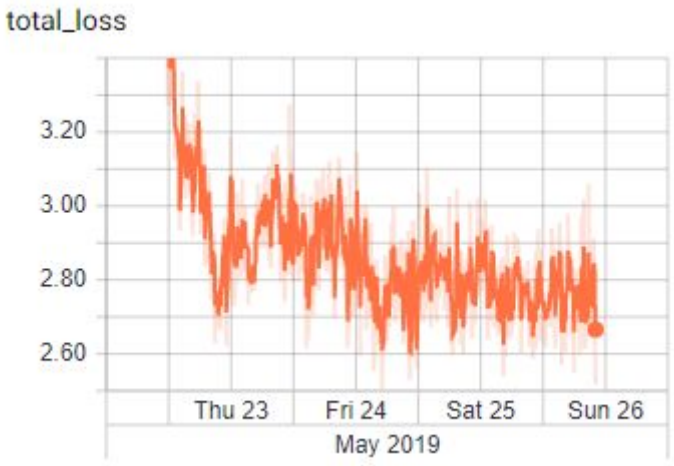


圖 41

然而遺憾的是依然沒有太大的成效，或許是訓練的不多，1500場對決3000場依然是1500場的AI獲勝。

猜想是探索參數的問題，我把探索調得太高了，可能導致在每次都在瞎搜而學不到東西，
因此我將狄利克雷噪點由0.3改為0.015，C_puct從5調成3，temp依舊，
發現並沒有多大的轉變，loss反而變高了，看來在小算力的情況下，探索手段縮小沒什麼用。

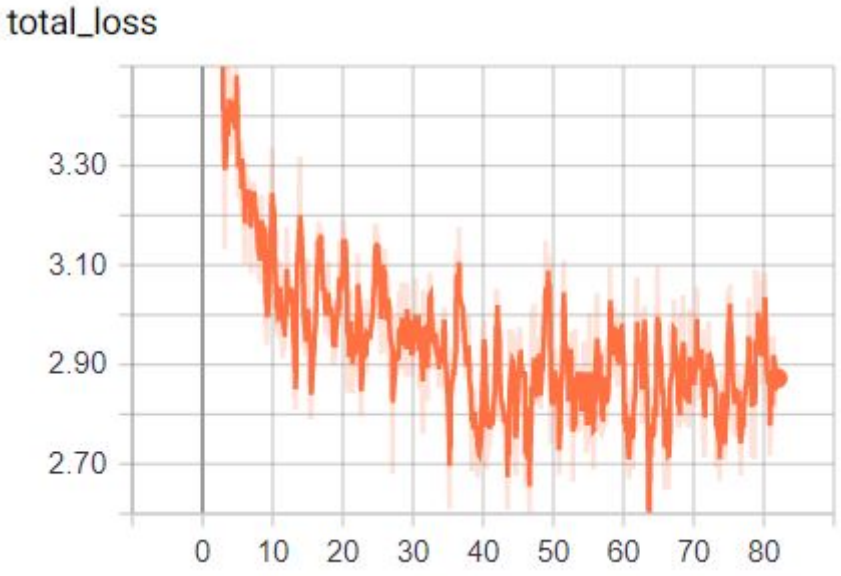


圖 42

結論

雖然很想直接套用alphago的神經網路架構來進行訓練，無奈我的程式實在太慢了，3000場訓練就要3天，GPU加速也只有加速一點點，因為大部分時間耗在了MCTS身上，未來改良可能當務之急就是將建立MCTS的方式改掉，alphazero是使用一種叫做hash table的方式儲存，它有點類似python的字典，而python也有提供該函數為namespace，接著加入multiprocessing，並且將棋盤表示方式改為更加有效的bitboard，即一個64位元的數值。

建議

我建議想要製作AI的盡早學習完畢，把重點放在程式上面，因為實驗所需要的時間相當的長，也建議先多多尋找github上開源的代碼，從他人的開始學起，才會事半功倍。

參考文獻

[1].AlphaZero: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm https://discovery.ucl.ac.uk/id/eprint/10045895/1/agz_unformatted_nature.pdf

[2].AlphaGo Zero: Mastering the game of Go without human knowledge https://arxiv.org/pdf/1712.01815.pdf

[3].程式碼架構主要來自: https://juejin.im/entry/6844903569817239565

[4].機器學習理論學習自李弘毅老師的教程: https://youtu.be/CXgbekl66jc

[5].影像辨識以及GUI(checker程式)學習自此書: https://www.books.com.tw/products/0010768933

[6]. 莫凡python基礎、神經網路程式基礎: https://mofanpy.com/

[7].AlphaGo 圍棋演算法原理解析: https://youtu.be/pgX4JSv4J70