

程式人



用 30 分鐘深入瞭解 AlphaGo 圍棋程式的設計原理

陳鍾誠

2016 年 3 月 18 日

2016 年 3 月 9 日

- Google 的圍棋程式 AlphaGo 第一次挑戰李世石九段超一流高手。

原本

- 大家都認為李世石會贏，問題只是到底贏多少而已！

結果

- 第一盤 AlphaGo 就贏了李世石！

而且、連贏三盤

- 直到第四盤才輸了一次
- 但到第五盤又贏了回來！

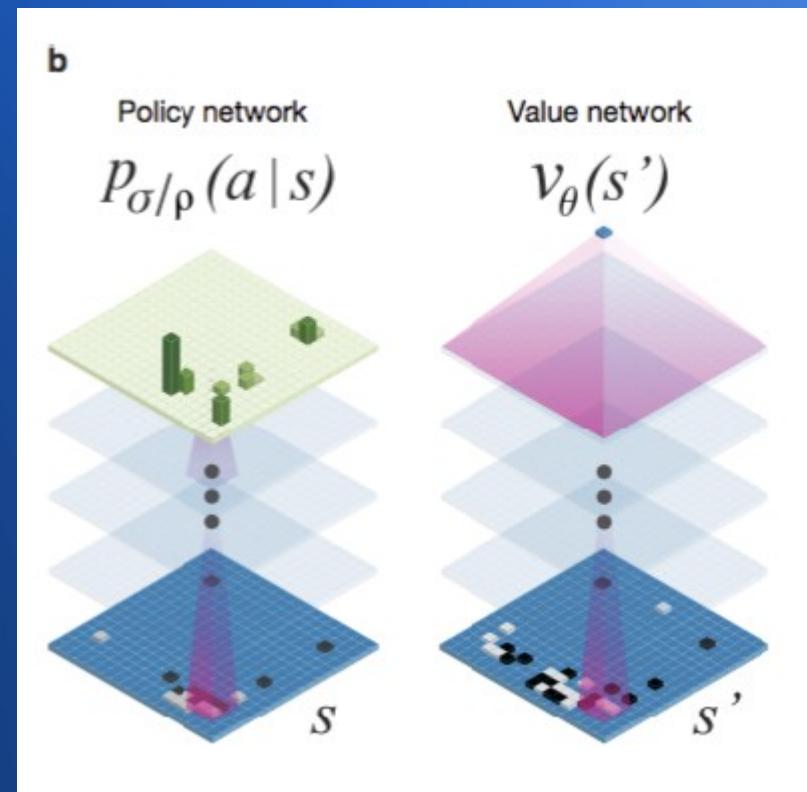
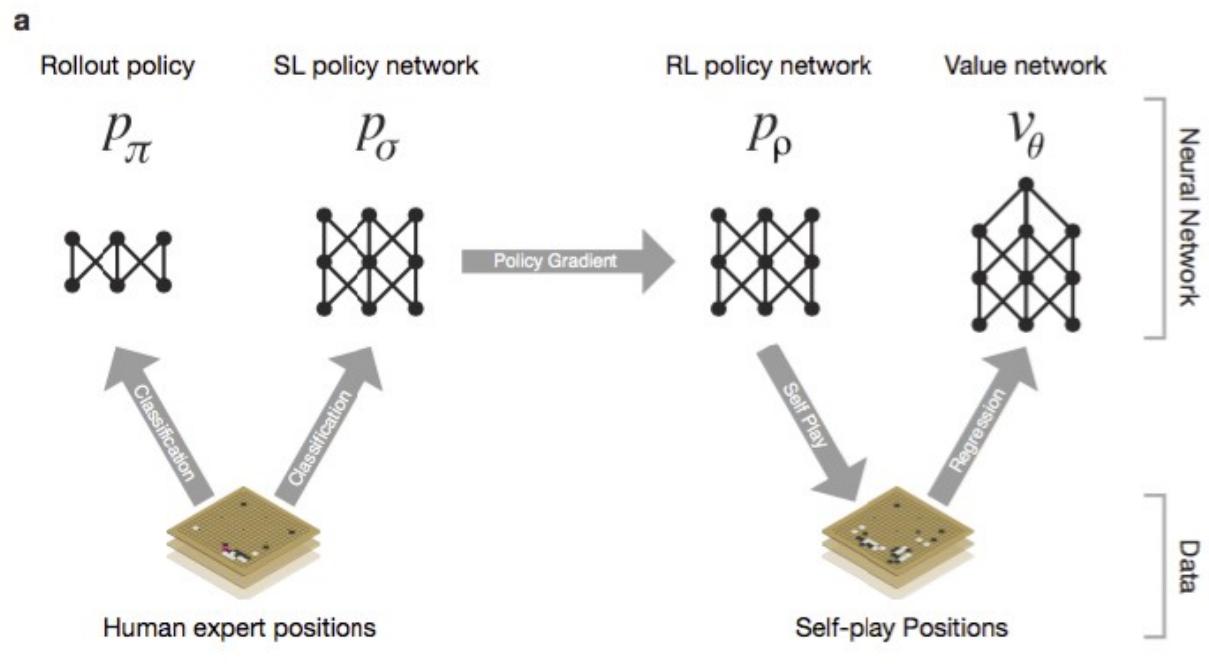
當時、我一邊看棋賽

- 一邊研究 AlphaGo 的設計原理

並且看了 AlphaGo 的論文

The screenshot shows a web browser displaying the article "Mastering the game of Go with deep neural networks and tree search" from the journal *Nature*. The URL in the address bar is www.nature.com/nature/journal/v529/n7587/full/nature16961.html. The page header includes the *nature* logo and the tagline "International weekly journal of science". The navigation menu at the top includes links for Home, News & Comment, Research, Careers & Jobs, Current Issue, Archive, Audio & Video, and For Authors. Below the menu, a breadcrumb trail shows the path: Archive > Volume 529 > Issue 7587 > Articles > Article. A horizontal line labeled "ARTICLE PREVIEW" spans the width of the page below the breadcrumb trail. Below this line, there is a link to "view full access options". The main content area features the title "Mastering the game of Go with deep neural networks and tree search" in large, bold, black font. Below the title, the authors' names are listed: David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis. At the bottom right of the content area, there are sharing and printing icons.

論文裏幾個關鍵的圖形



這些圖形

- 記載著 AlphaGo 的設計方法

後來、我寫了兩篇十分鐘投影片

- 一篇解釋電腦下棋的傳統方法
- 一篇解釋 AlphaGo 的設計並探討可能的弱點。

就在 AlphaGo 以 4 比 1 擊敗李世石的時候

- 我看到了一篇投影片
 - 名稱是 AlphaGo in Depth
- 是 Mark Chang 寫的
- 而且寫得超棒！

這分投影片釐清了我對 AlphaGo 設計的很多疑問！

→ C www.slideshare.net/ckmarkohchang/alphago-in-depth

in SlideShare | Search

Home Technology Education More Topics My Clipboards

Clip slide

The image shows a Go board with black and white stones. The board is 19x19. There are several groups of stones, mostly in the upper half of the board. A cursor arrow is pointing towards the top-left corner of the board area. The background of the slide is yellow.

AlphaGo in Depth

by Mark Chang

1 of 80

在取得了 Mark Chang 的同意之後

- 我決定把上述三分投影片
 - 融合為一份
- 用我的想法，重新詮釋 AlphaGo 的設計原理！

現在、就讓我們開始這趟

- 理解 AlphaGo 電腦圍棋程式的漫長旅程！
- 開始用《程式人》的角度，解說這個令人感到驚訝的智慧型程式之原理。

但是、要理解 AlphaGo 之前

- 我們必須先介紹一下，電腦到底
是如何下棋的！

在此、我不會以圍棋為例

- 因為那會太過困難！

所以我會先用五子棋作範例

- 來說明電腦是如何下棋的！

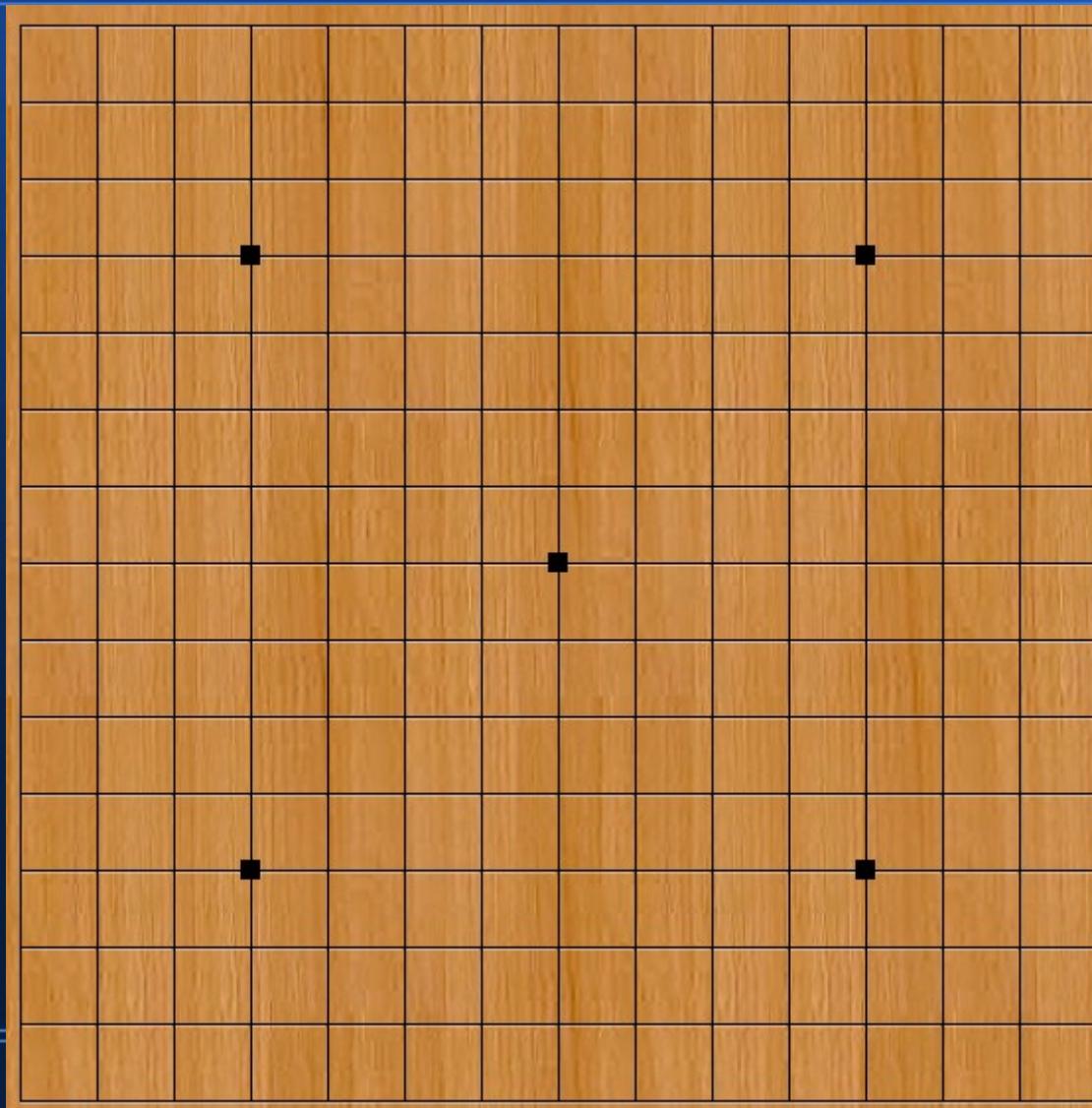
在傳統的電腦下棋方法中

- 通常有兩個主要的關鍵算法
- 第一個是《盤面評估函數》
- 第二個是《搜尋很多層對局》，尋找最不容易被打敗的下法。

首先讓我們來看看盤面評估函數

- 我們會用最簡單的五子棋為例，這樣比較好理解！

請大家先看看這個 15*15 的棋盤



注意：雖然格子只有 14×14 格，但五子棋是落在十字線上的，所以實際上是 15×15 個可以下的點。

如果不考慮最邊邊的話，那就會有 13×13 個可以下的位置。

不過以這個棋盤，邊邊是可以下的，所以應該是 15×15 的情況才對。

如果電腦先下

- 那第一子總共有 $15 \times 15 = 225$ 種下法。
- 電腦下完後換人，此時還剩下 224 個位置
可以下。
- 等到人下完換電腦，電腦又有 223 個位置
可以下！

於是整盤棋的下法

- 最多有

$$-225 * 224 * \dots * 1 = 225!$$

- 種可能的下法

而且、這是 15*15 的棋盤

- 標準圍棋棋盤是 $19*19=361$ 個格線，所以就會有 $361!$ 的可能下法！
- 只要能夠把所有可能性都確認，電腦就絕對不會下錯，基本上也就不會輸了！
- 但是 $361!$ 是個超天文數字，電腦就算再快，算到世界末日宇宙毀滅都還是算不完的！

不過、這件事情先讓我們暫時擱下

- 因為電腦就算算完了也沒有用，重點是要算些甚麼東西出來？
- 這樣才能告訴我們應該下哪一步呢？

這個要算的東西

- 就是盤面評估函數！

以五子棋而言

- 我們可以用很簡單的方法，
計算目前盤面的分數。

以下是一個盤面評估函數的方案

- 連成 5 子： 10000 分
- 連成 4 子： 50 分
- 連成 3 子： 20 分
- 連成 2 子： 5 分
- 連成 1 子： 1 分

等等、這只有考慮自己這方

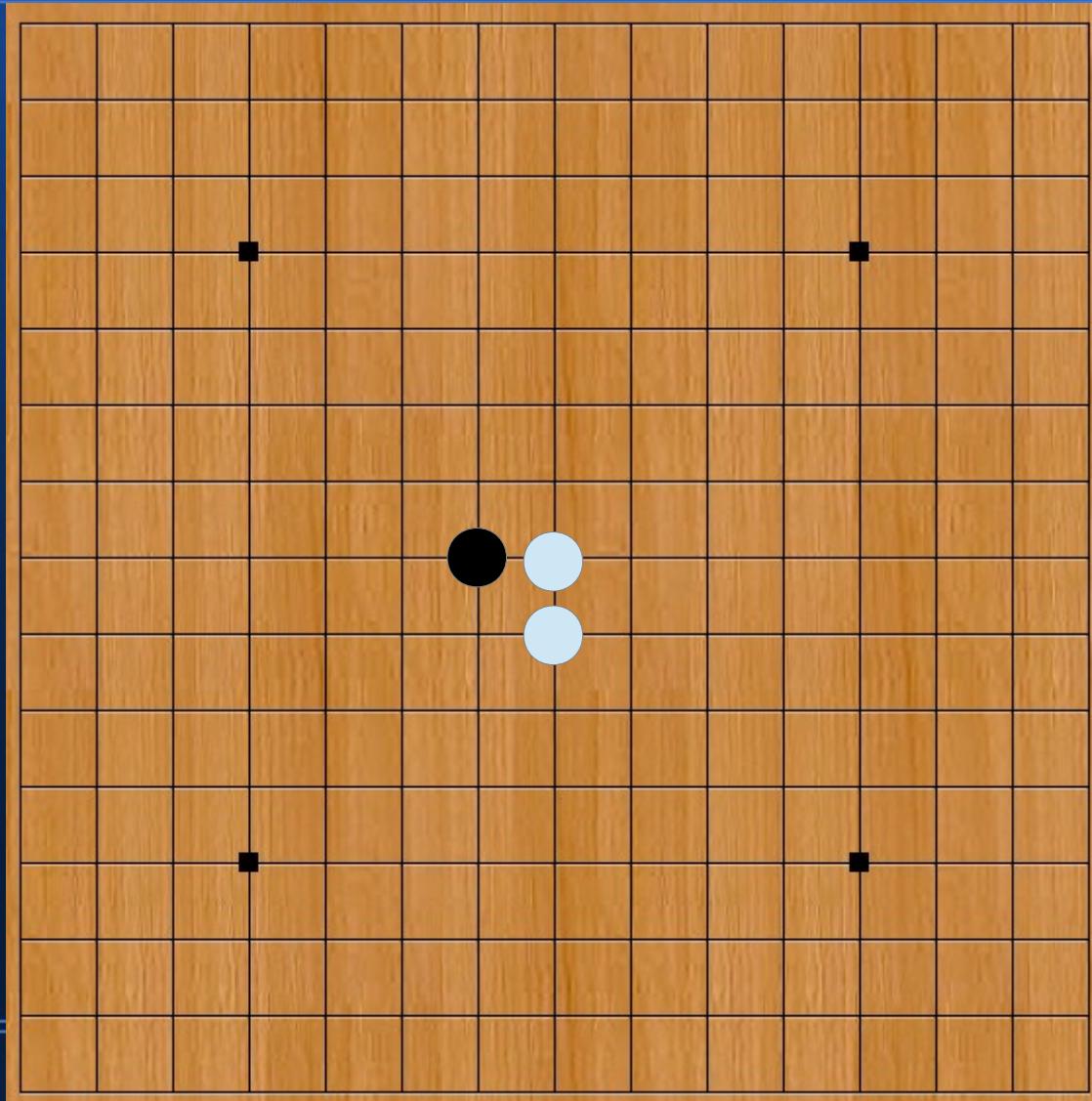
- 沒有考慮對方的得分！

沒錯

- 一個完整的盤面評估函數，應該考慮到雙方！
- 所以可以用
 - 我方得分 – 對方得分
- 做為評估函數

舉例而言

連成 5 子 : 10000 分
連成 4 子 : 50 分
連成 3 子 : 20 分
連成 2 子 : 5 分
連成 1 子 : 1 分



在左邊的盤面中，假設電腦為白子。

白子兩顆連線，黑子只有一顆。

所以白子的得分為

$$5+1*2 = 7$$

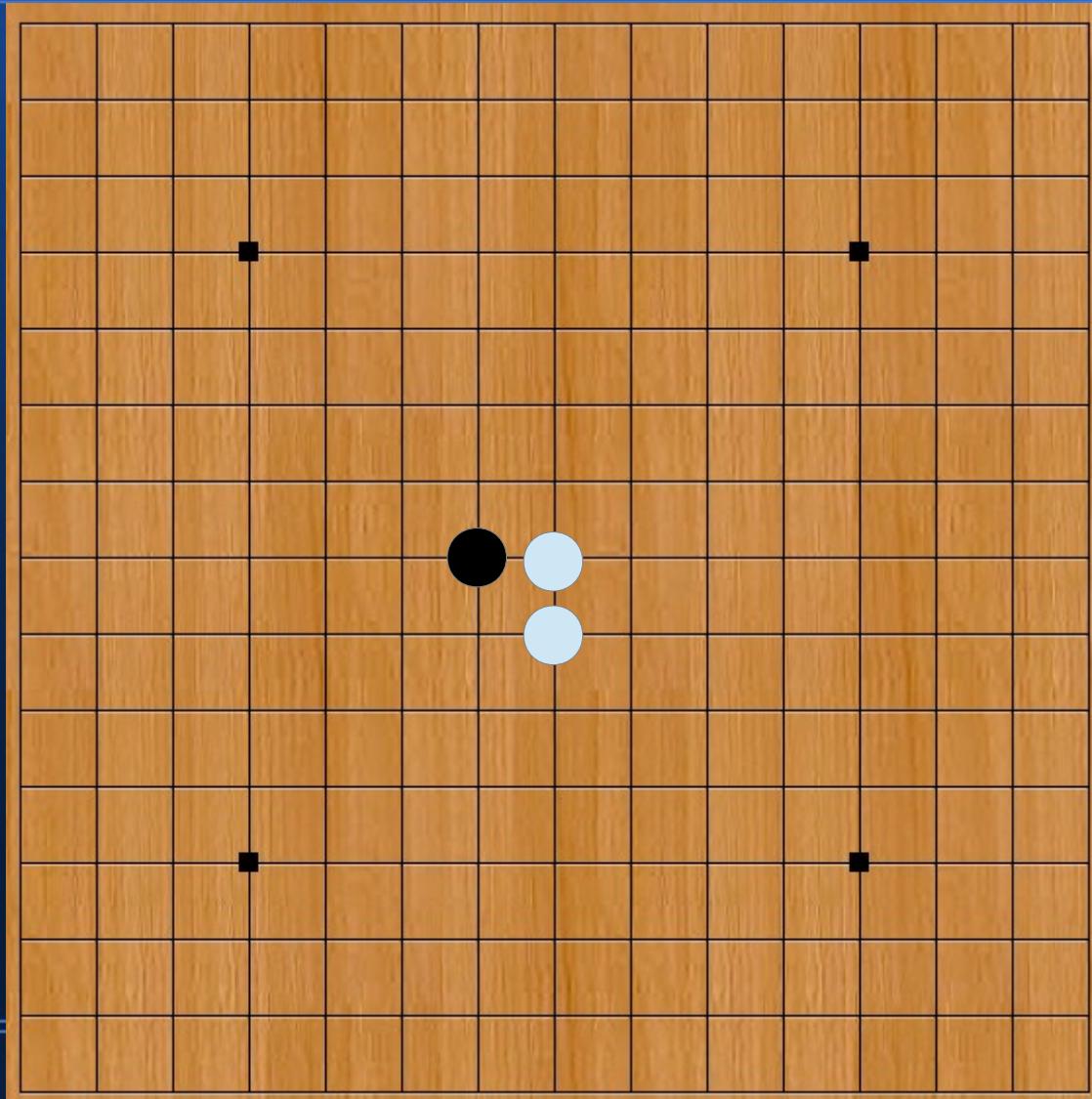
兩顆連線
得 5 分

兩個一顆的情況也計入
各得 1 分

雖然這裡有點重複算，但由於分數的設計
差距夠大，所以沒有關係。

由於黑子只有一個 目前只得一分

連成 5 子 : 1000 分
連成 4 子 : 50 分
連成 3 子 : 20 分
連成 2 子 : 5 分
連成 1 子 : 1 分



所以對電腦而言，
盤面分數為

$$7 - 1 = 6$$

於是我們可以寫一個程式，
計算盤面的分數。

這個程式並不算難，對一個
學過基礎程式設計，會用
二維陣列的人應該是很容易
的。

假設這個程式為 `score(B)`，
其中的 `B` 代表盤面陣列。

有了這個盤面評估函數 score(B)

- 我們其實就可以輕易建構出一個簡單的下棋程式了。
- 因為電腦只要把每個可以下的位置，下子之後的分數算出來，然後下在分數最高的那一格，就可以了！

最簡易的下棋程式

1,1

1,15

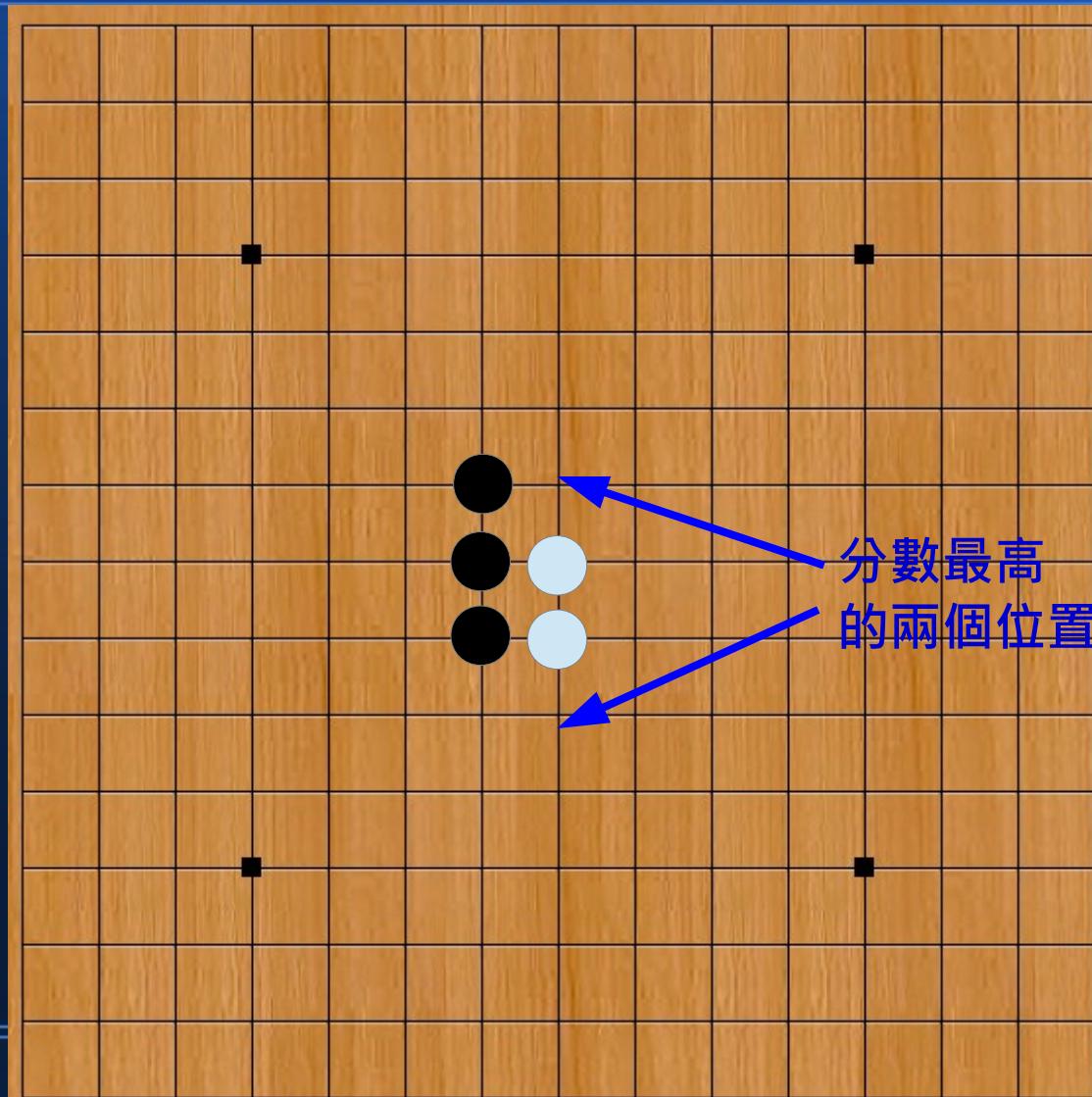
舉例而言，假如電腦為白子，現在換電腦下：

那麼電腦會笨笨的計算

$(1,1), (1,2), \dots, (1,15),$
 $(2,1), (2,2), \dots, (2,15),$
...
 $(15,1), (15,2) \dots (15,15)$

當中還沒被下過的位置，每一格下完後的分數。

然後挑出最高分的位置下子！



15,1

15,15

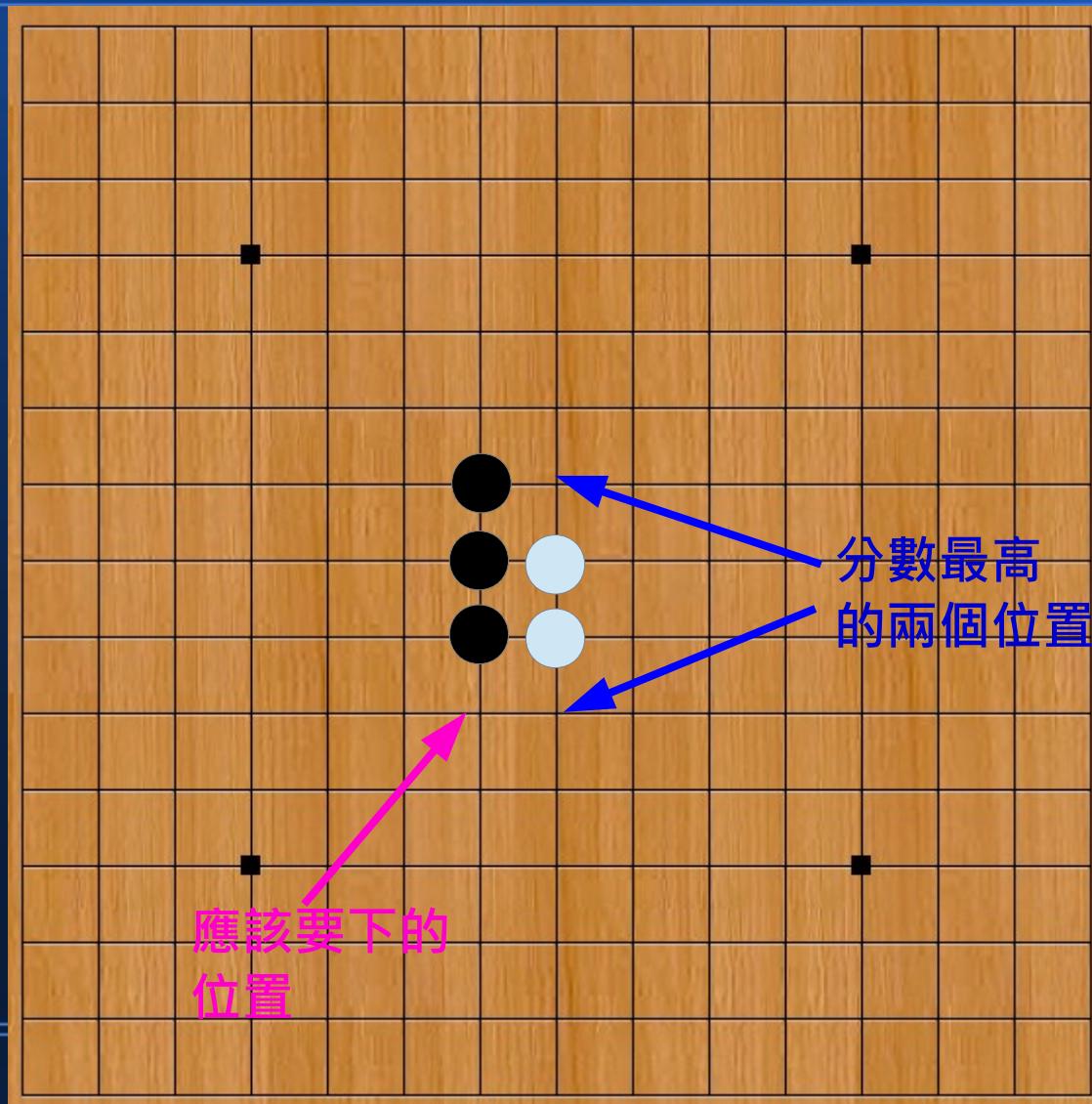
但是

- 這種程式的棋力不強
- 因為只看進攻不看防守！

該程式由於太過貪心

- 只看自己的分數，不看對方下一手的分數。
- 如果你稍微做個洞給他跳，很容易就會贏了！

舉例而言



為了避免這個問題

- 電腦除了考慮攻擊的得分之外
- 還應該考慮防守的得分。

但是即使考慮了防守

- 棋力也不會太強，大概只能
下贏初學的小孩！

要提升電腦的棋力

- 就必須加上《對局搜尋》的功能！

到底

- 《對局搜尋》是甚麼呢？

更明確的說

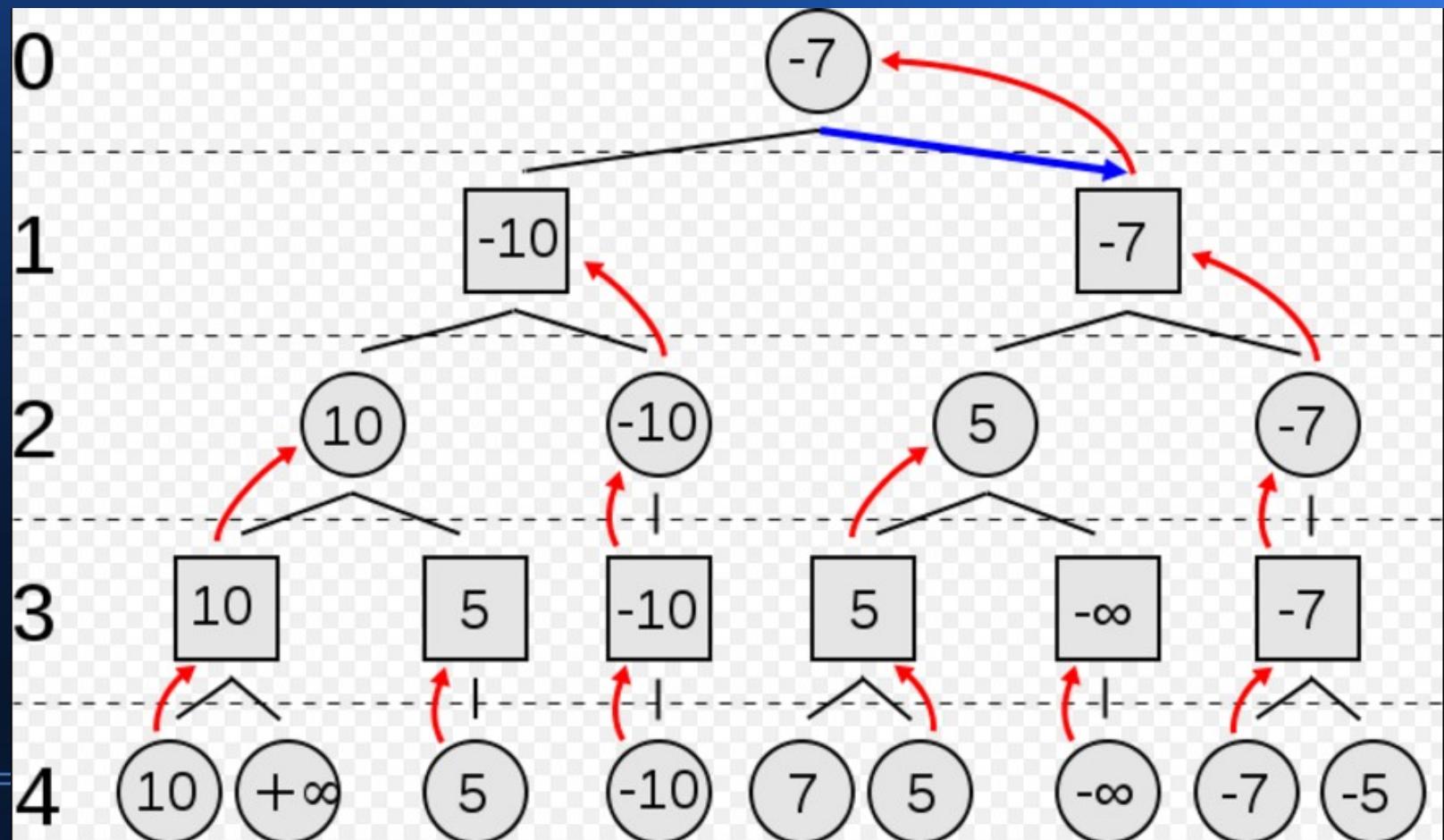
- 就是 MinMax 《極小極大》演算法

以下、讓我們圖解一下

- MinMax 演算法的想法！

下圖中的偶數層，代表我方下子
奇數層代表對方下子

我們必須找一個《最糟情況失分最少的路》，這樣在碰到高手時才不會一下被找到漏洞而打死！



但是、這樣的方式搜尋不了多少層！

- 因為如果每步有 $19 \times 19 = 361$ 種可能，那麼
 - 兩層就有 13 萬種可能
 - 三層就有四千七百萬種可能
 - 四層就有一百六十億種可能
 - 五層就有六兆種可能

電腦再快也無法搜尋超過十層

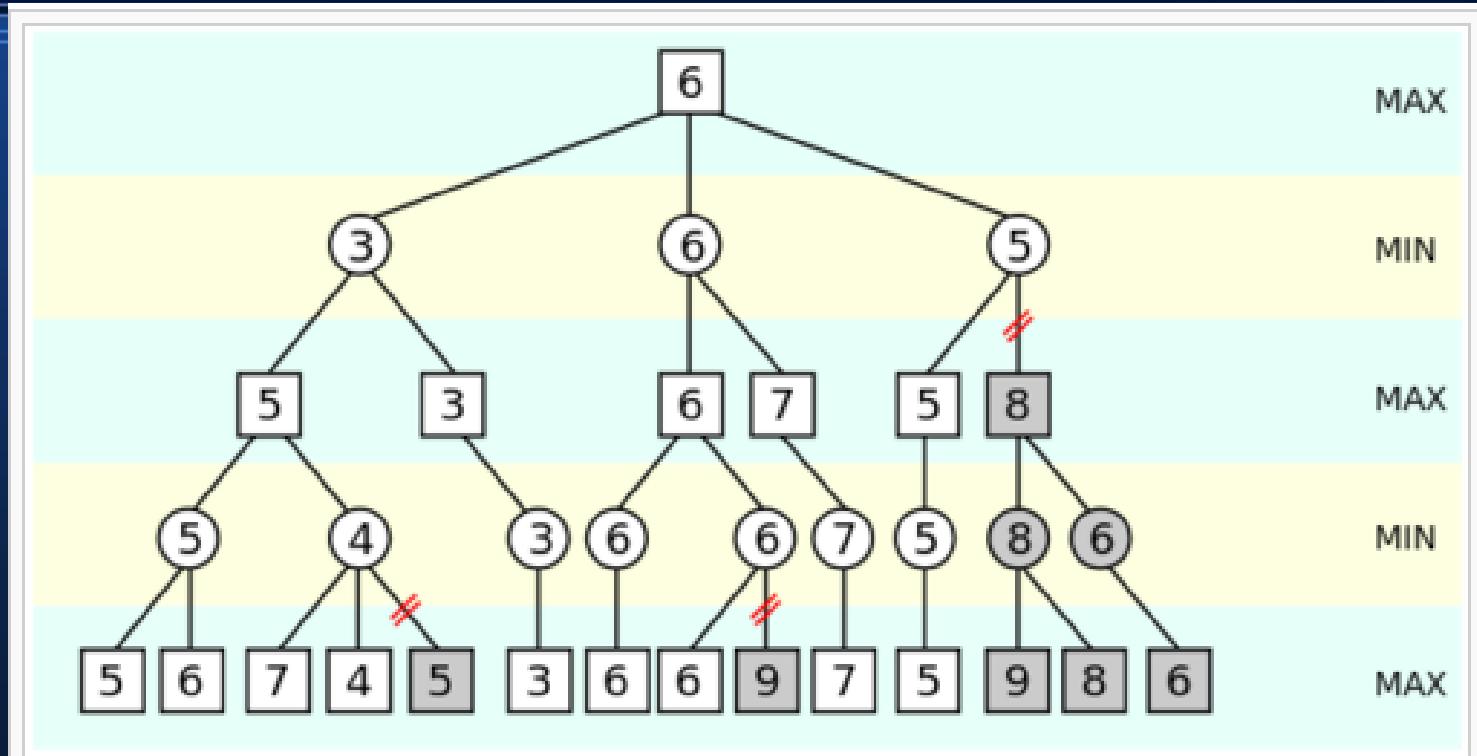
所以

- 還需要一些其他的方法，才能搜尋得更深！

這時候

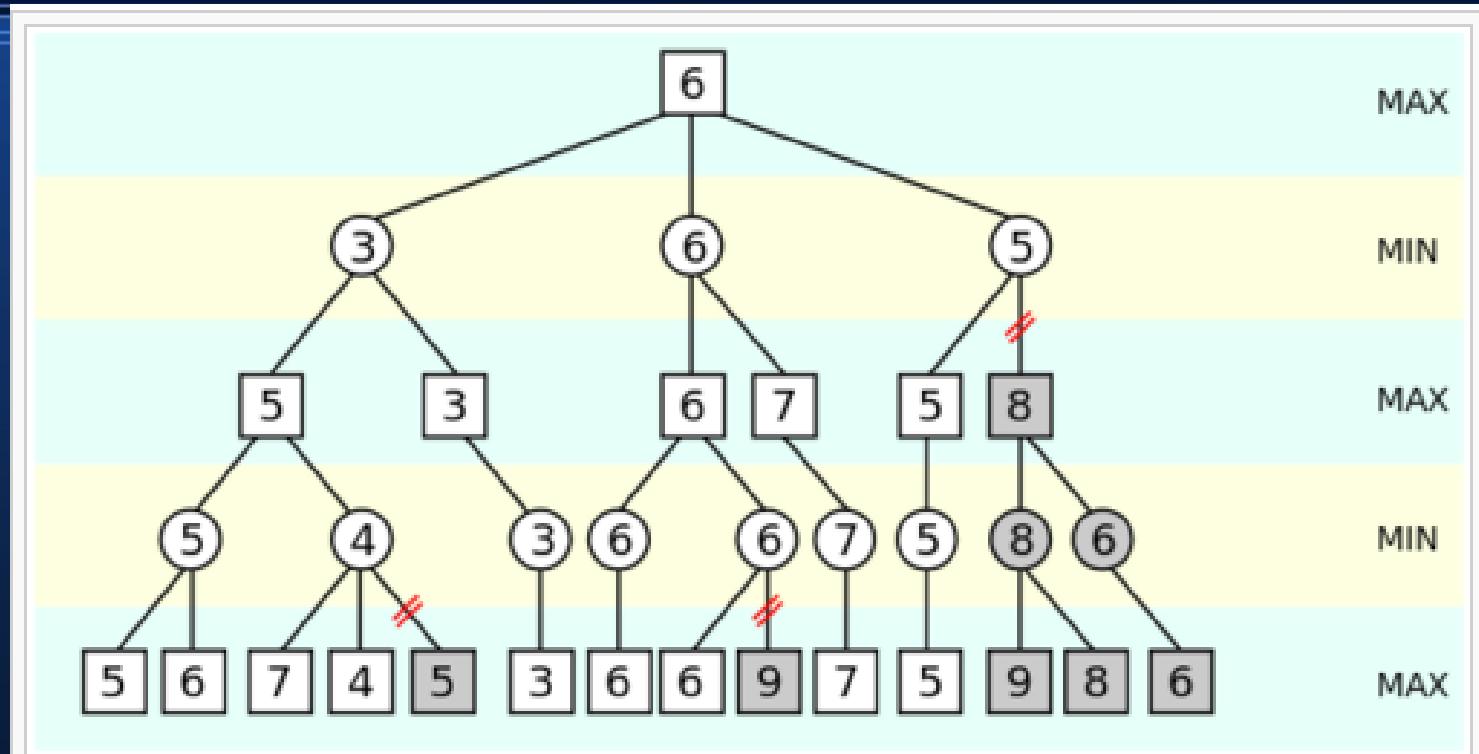
- 可以採用一種稱為 Alpha-Beta 修剪法的演算法
- 把一些已經確定不可能會改變結果的分枝修剪掉。
- 這樣就可以減少分枝數量，降低搜尋空間

以下是 Alpha-Beta 修剪法的範例



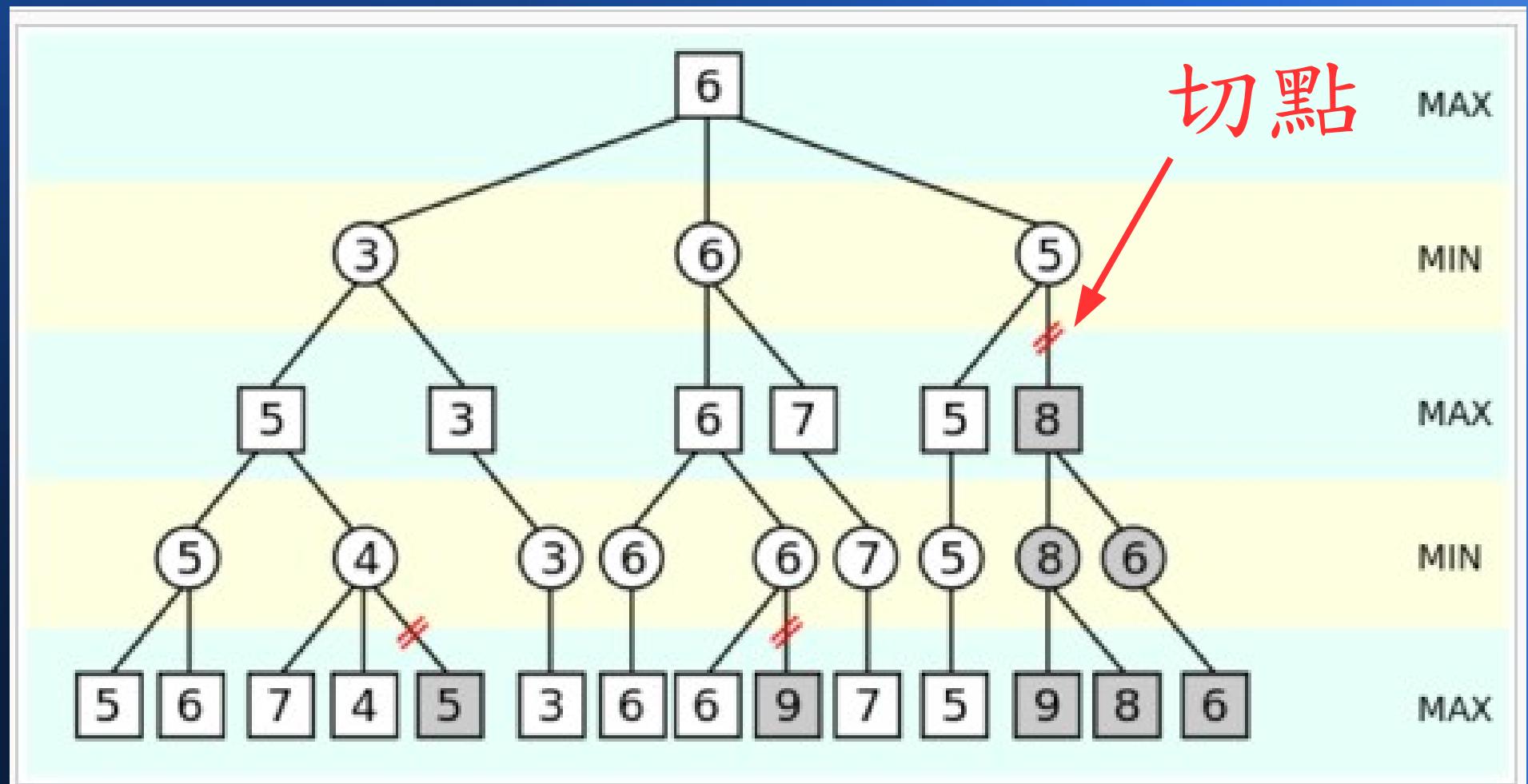
An illustration of alpha–beta pruning. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

您可以看到雙紅線切掉的部分
就是 Alpha-Beta 修剪法的功效



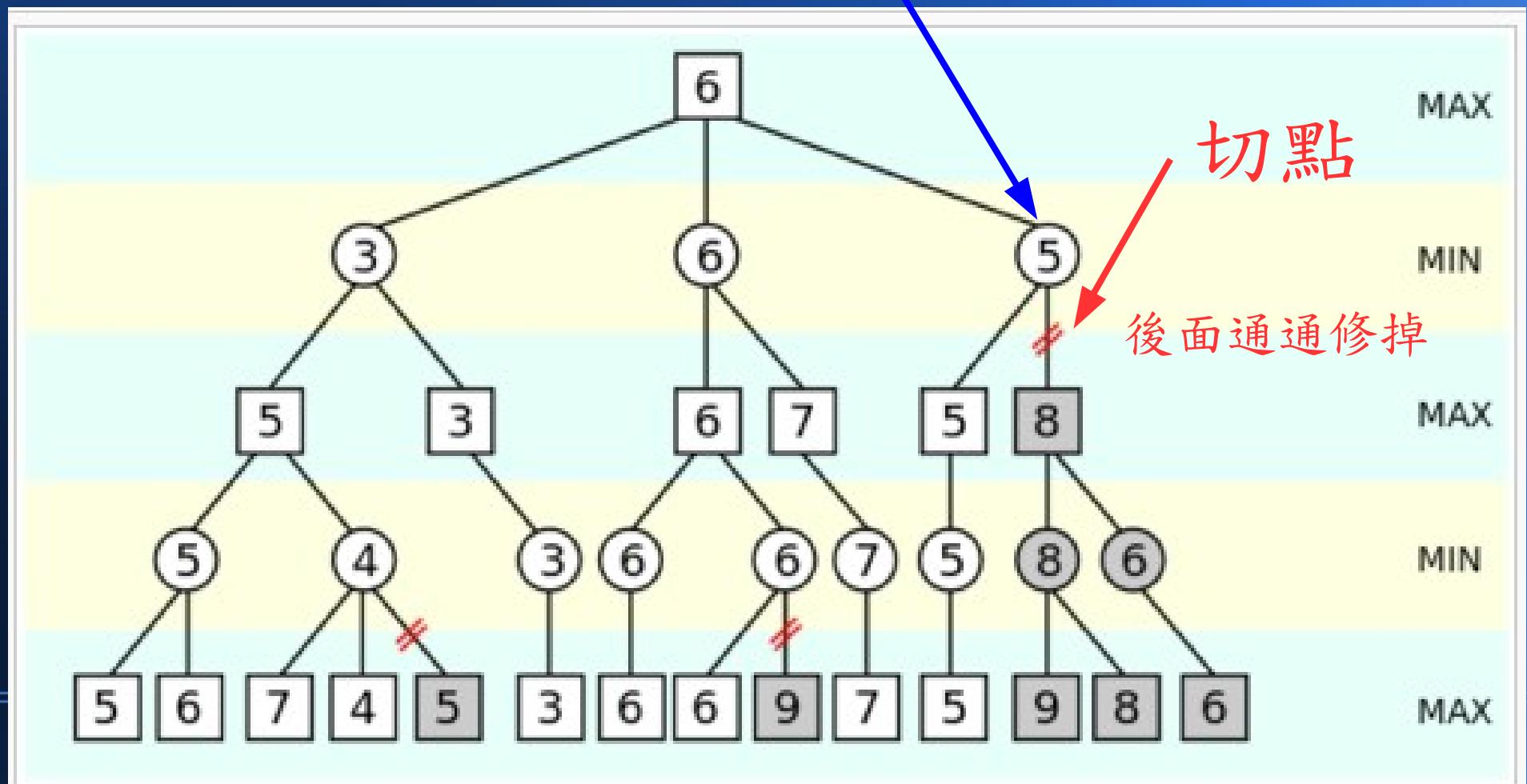
An illustration of alpha–beta pruning. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

讓我們以圖中的切點為例
說明為何該部分可以切掉



因為切點上面 Min 層目前值為 5
比前面的 6 還小

因此後面的值不管多大，都只可能讓此處的數值變得更小，不可能更大了
所以後面的所有分枝都將不需要再算下去，可以修剪掉了！



於是透過 Alpha-Beta 修剪法
就可以大大減少分枝數量

- 讓電腦可以在固定的时间限制
內，搜尋得更深更遠。
- 於是棋力就可以提高了！

這個 Alpha-Beta 修剪法

- 是由 LISP 的發明人 John McCarthy 所提出，後來由 Allen Newell and Herbert A. Simon 兩人實際用在下棋上。
- 這三位後來都曾經得過圖靈獎！

有了 MinMax 的搜尋

- 加上 Alpha-Beta 修剪法，電腦在五子棋上就可以輕易地擊敗人類了！
- 在西洋棋和象棋上，則還需要棋譜來訓練出更強更好的評估函數！

好了

- 現在您已經大致瞭解了電腦下棋的方法，特別是下五子棋的方法了！
- 但是、圍棋雖然和五子棋使用相同的棋盤和棋子，但是兩者的難度卻差很多！

因為、五子棋只要觀察局部

- 連線的子數越多，就會得越多分數！
- 但是圍棋卻要以最後佔地的多寡來計分，而且先吃掉對方的子常常反而是不利的。

換句話說

- 五子棋只要看局部就會有一定的棋力
- 圍棋卻非得要有整體布局才能得到最後的勝利！

而且

- 五子棋很容易設計出盤面評估函數
- 圍棋卻很難在盤中用程式評估盤面的好壞。

這些困難點

- 都是導致圍棋對電腦而言，比五子棋難很多的原因！

所以

- 電腦要能夠打敗九段棋王
- 真的是相當困難的一件事情
- 但是 AlphaGo 做到了！

現在、就讓我們來看看

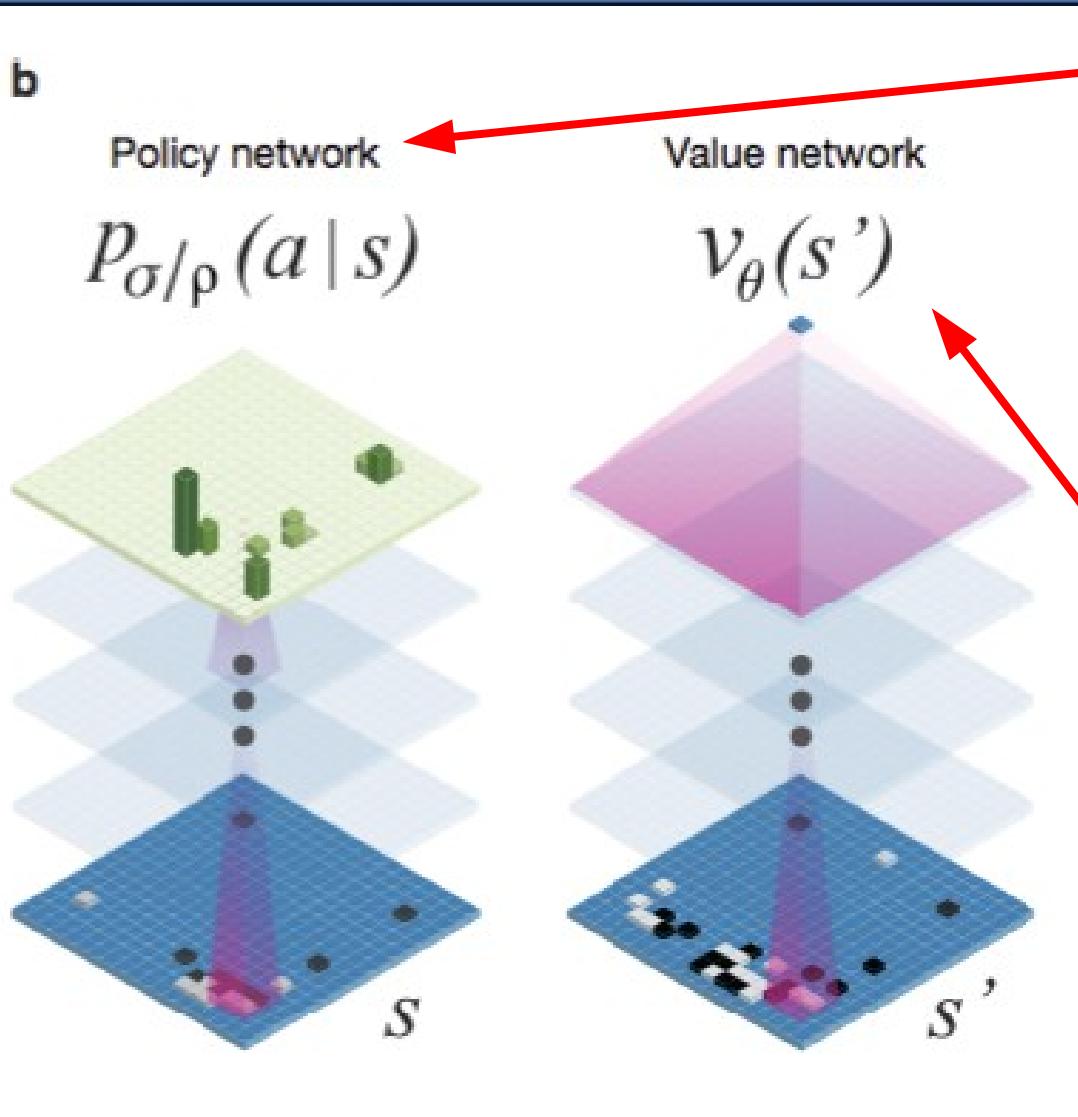
- AlphaGo 的設計原理！

首先、AlphaGo 設計中最重要的 是兩個《神經網路》所形成的函數

- 一個稱為《策略網路》(Policy Network)，該
網路可以預估 AlphaGo 在某盤面時，下某一子的
《機率》。
- 另一個稱為《價值網路》(Value Network)，這
個網路基本上就是《盤面評估函數》。

以下是這兩個網路的示意圖

b



《策略網路》

- $P(a | s)$

在 s 盤面時下 a 那子的《機率》

《價值網路》

- $V(s') =$ 《盤面評估函數》

s' 這個盤面的好壞 (對我方而言)

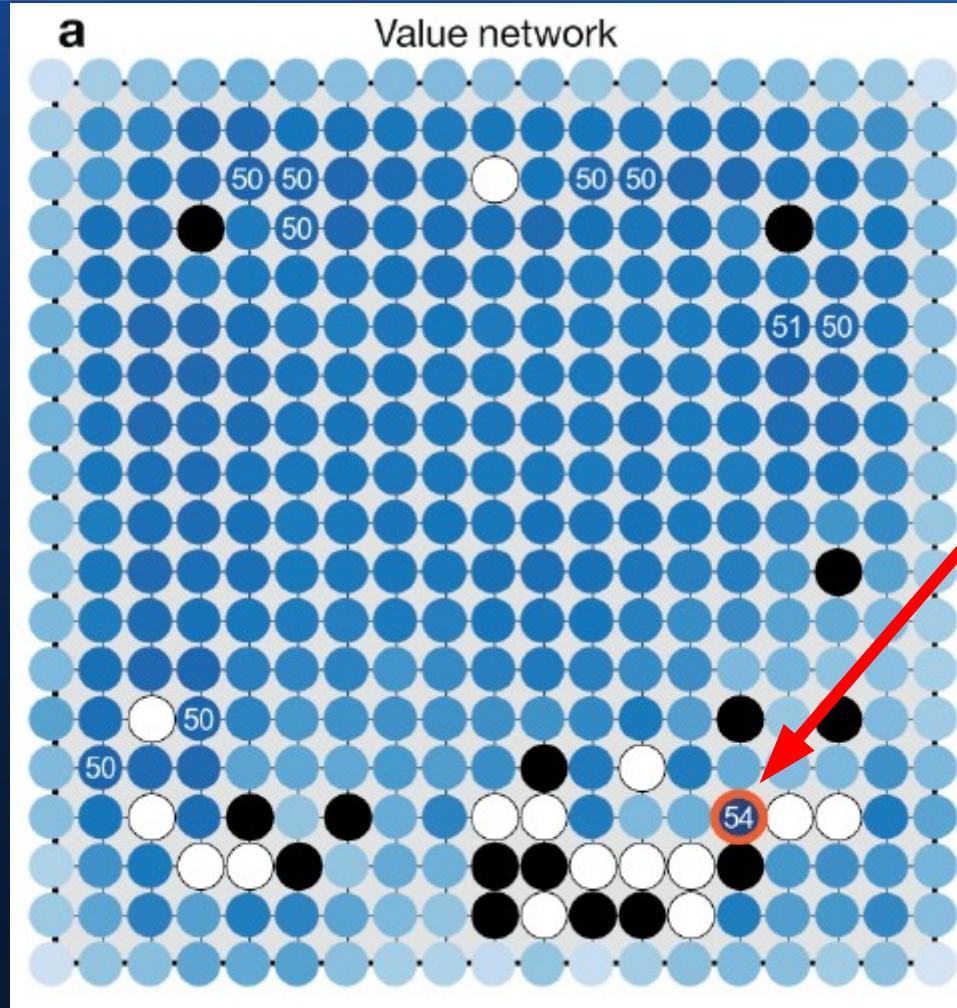
只要、這兩個函數

- 能夠很正確的評估
 - 在《某盤面時應該下哪一子》
 - 以及《盤面的好壞》
- 那就所向無敵了！

事實上、只要其中一個很完美

- 就已經所向無敵了，因為：
 1. 假如策略網路很完美，就能正確評估每一步應該下哪一子。（那每次都選最好的那子下就好了啊）
 2. 假如價值網路很完美，就能正確地知道每個盤面有多好。（那就把我方下一步可能下的位置，下完後的盤面分數都算一遍，選最好的下就好了啊）

舉例而言、在以下盤面當中



AlphaGo 為黑子，樊麾下白子
現在輪到 AlphaGo 下黑子

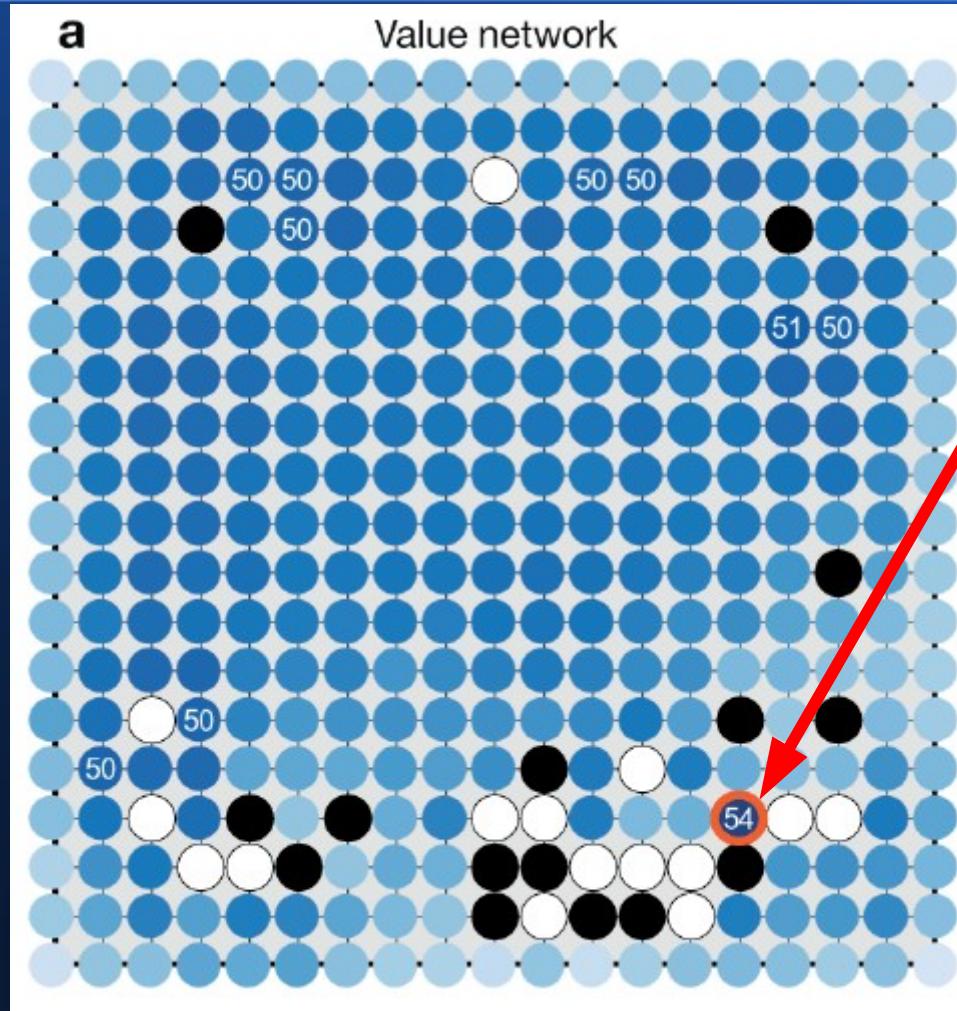
假如策略網路的 $P(a|s)$ 函數夠好，
我們就知道下哪一子機率最大（最好）。

以左圖中而言，我們就應該下在
機率（勝率）54% 的那一點。

所以我們只要有最好的 $P(a|s)$ 函數
就能下出完美的棋局。

（對手差不多是必輸，因為人不可能每一步都下得最好）

同樣在這個盤面中



假如價值網路的 $V(s)$ 函數夠好，
我們就可以計算下哪一子之後，
得到下一個盤面是最好的。

所以我們只要有最好的 $V(s)$ 函數
也同樣能下出完美的棋局。

(對手一樣是輸定了，因為人不可能每一步都下得最好)

所以、策略網路和價值網路

- 其實是一體的兩面，都可以用來《告訴電腦怎麼下好棋》！
- 而且只要有其中一個，就可以推出另外那個！

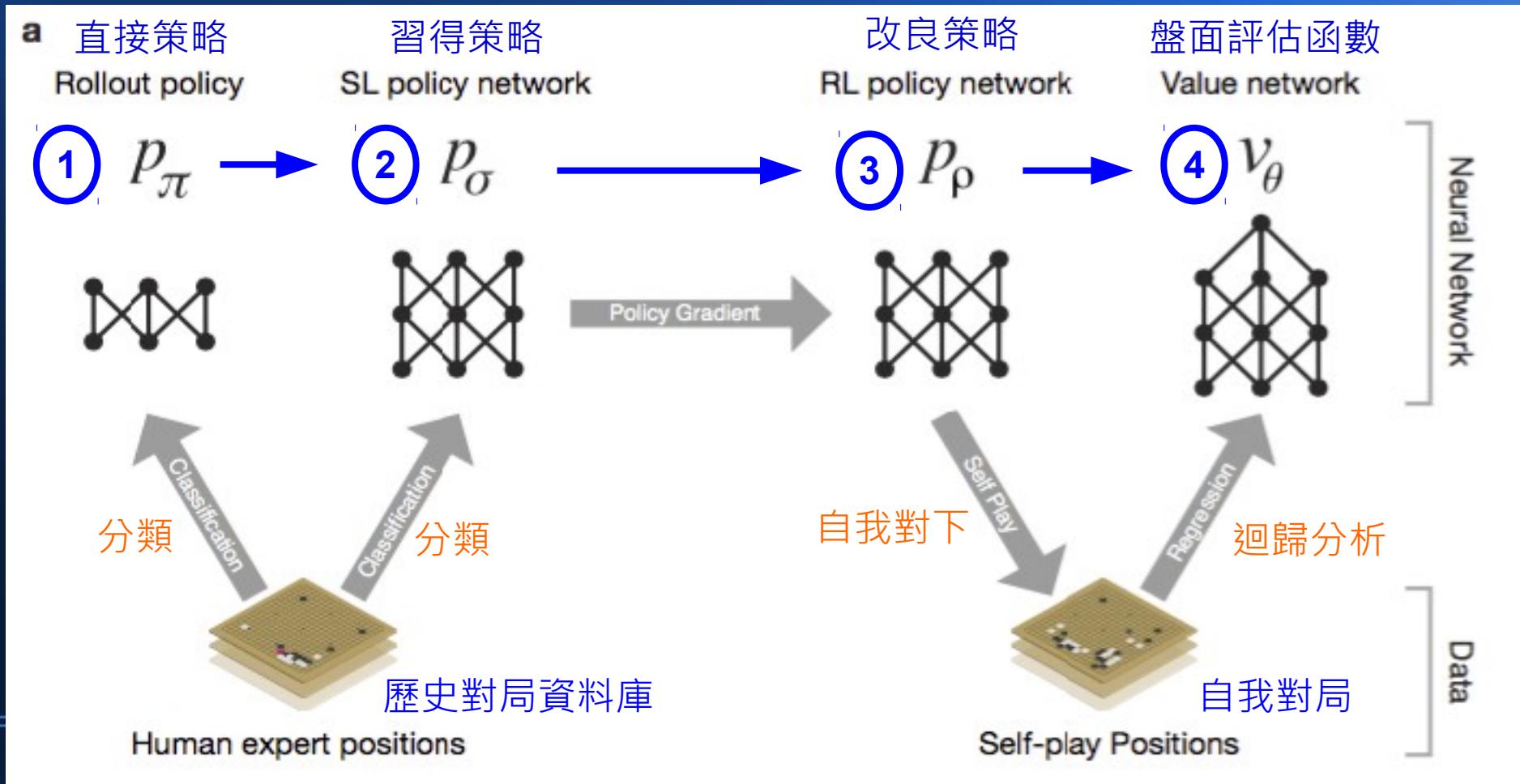
但是、要怎樣得到

- 好的策略網路函數 $P(a|s)$ 或價值
網路函數 $V(s)$ 呢？
- 這就是最困難的地方了！

在 AlphaGo 當中

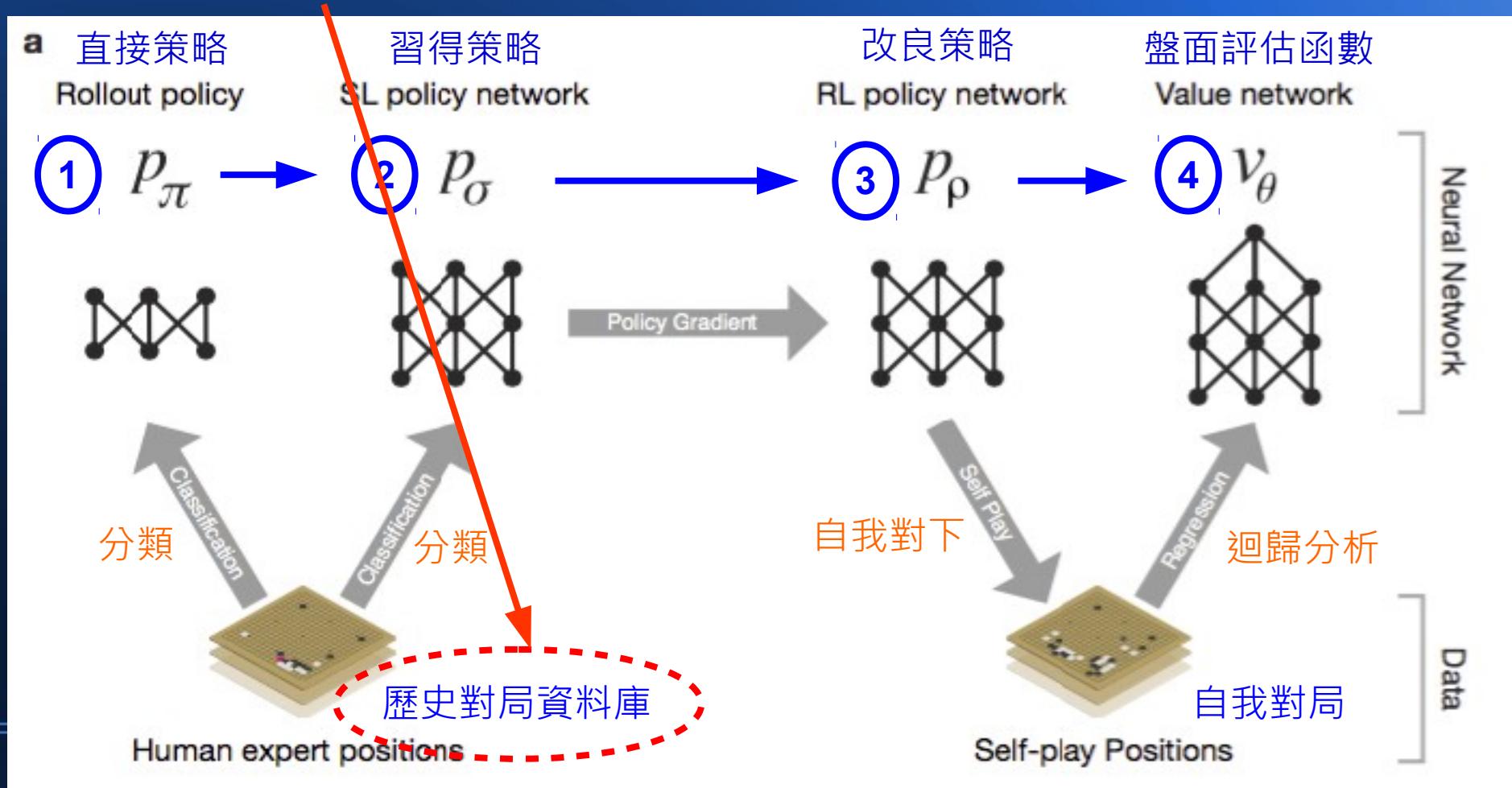
- 首先用《對局資料庫》來訓練出基礎的《策略網路》。
- 然後在利用《自我對下》強化策略網路。
- 最後再利用強化後的策略網路，用迴歸的方式得到《價值網路》。

其訓練過程如下圖所示



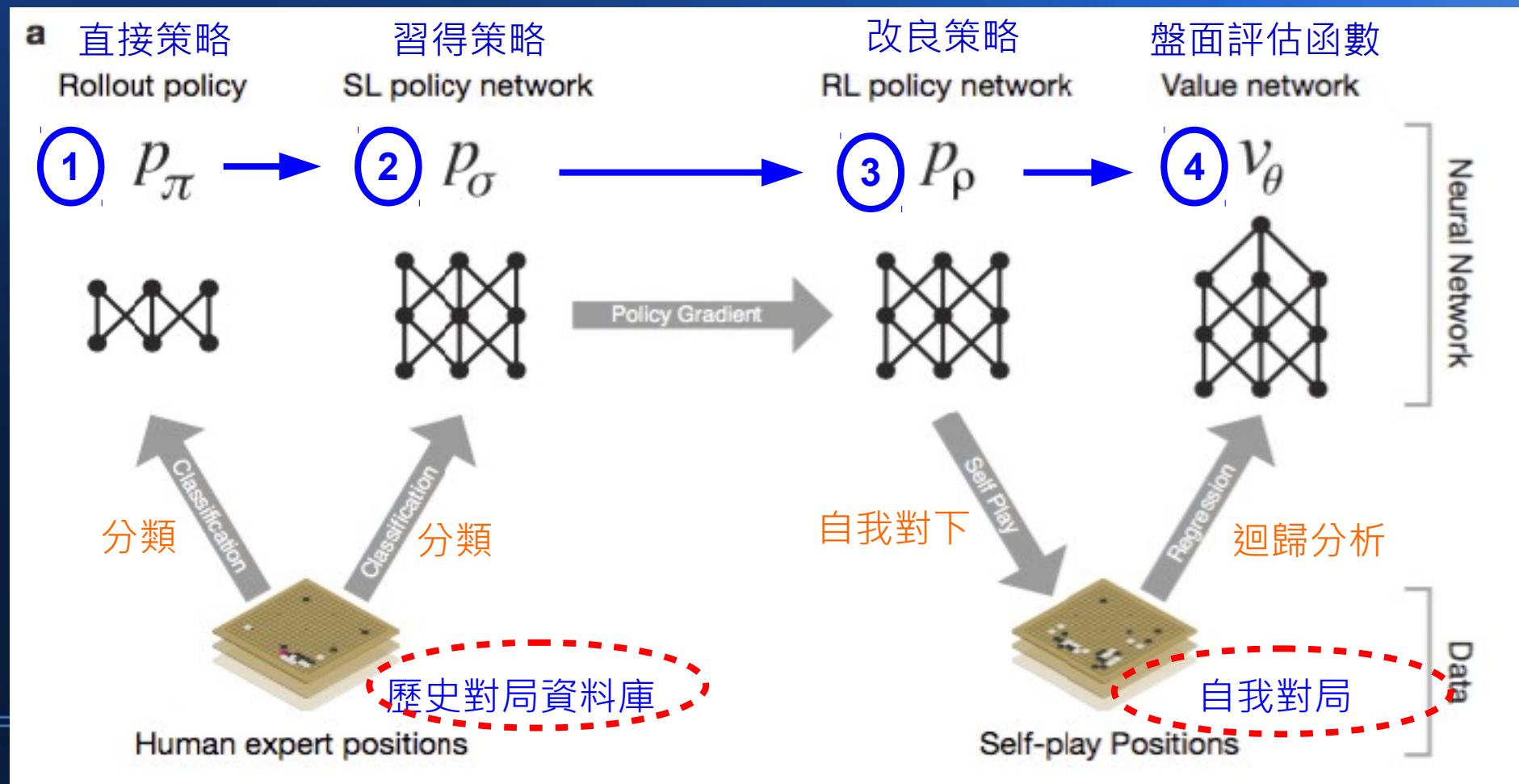
首先讓我們關注 《對局資料庫》的部分

這就是 Google 蔑集的所有歷史對局的完整過程，應該是很大的對局資料庫



AlphaGo 利用這個對局資料庫

1. 進行分類 (Classification) 之後得到《直接策略》
2. 然後再用神經網路一般化之後得到《習得策略》
3. 接著用強化學習 RL 《自我對下》得到《改良策略》
4. 最後利用《迴歸》從中得到價值網路的《盤面評估函數》

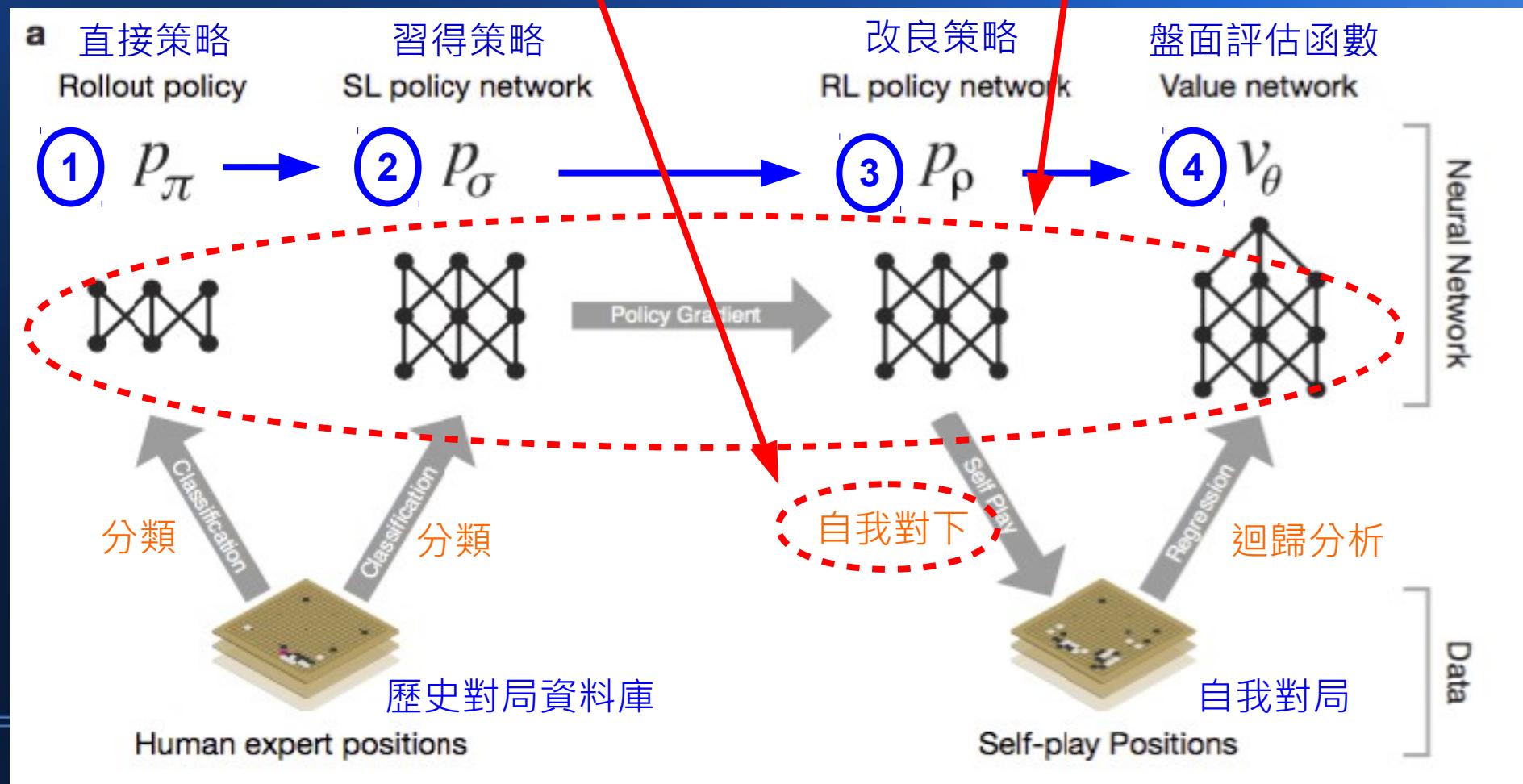


在 AlphaGo 當中

- 網路的表達與訓練，採用的是《深捲積神經網路》
(Deep Convolutional Neural Network, DCNN)
- 然後用《歷史對局資料庫》去訓練《策略網路》
- 再用《蒙地卡羅對局樹搜尋法》(Monte Carlo Tree Search)去找出值得探索的盤面，接著進行《自我對下》以改進這個《策略網路》，以強化 AlphaGo 的棋力。

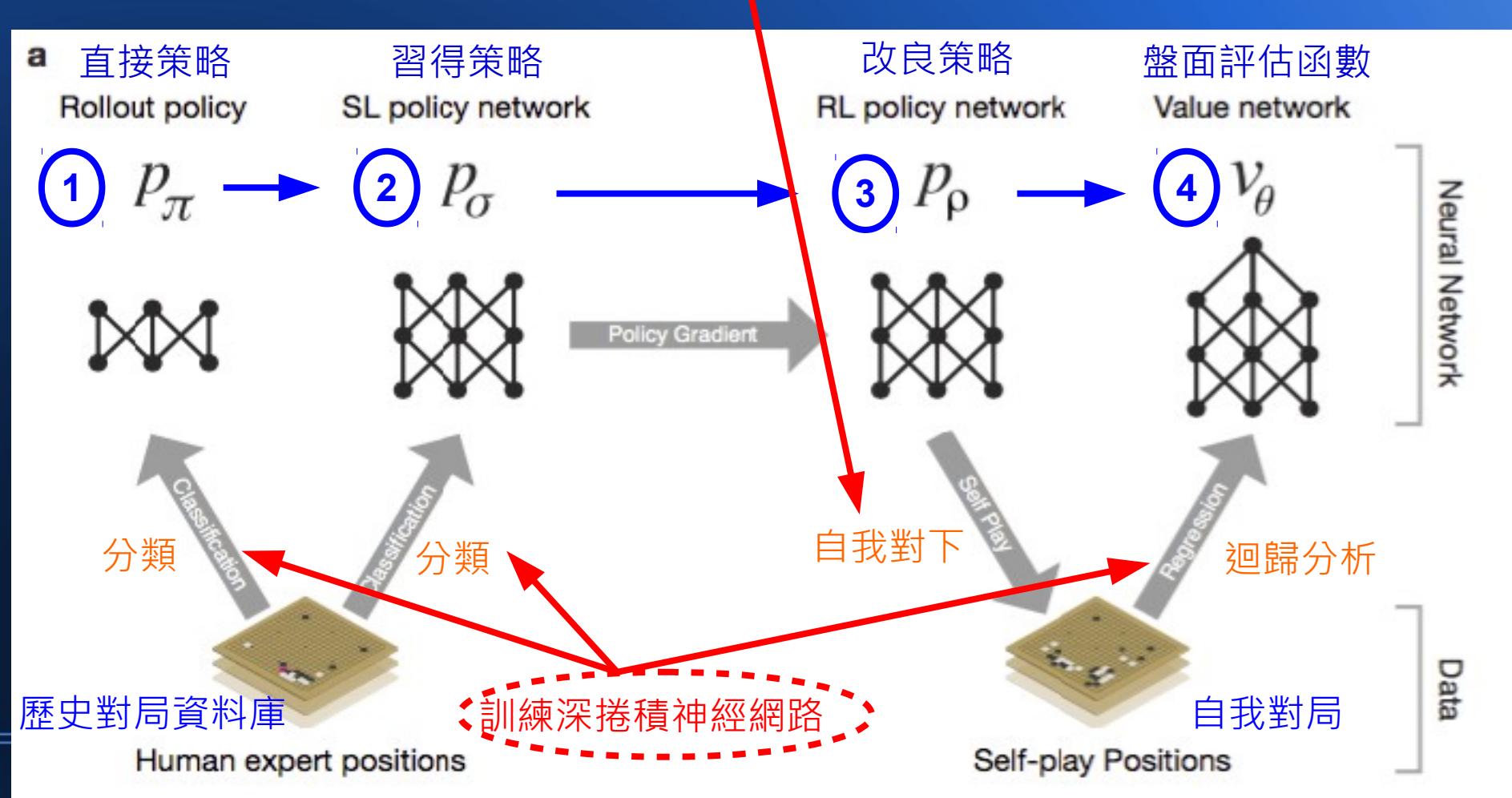
這些方法的使用時機如下圖所示

1. 蒙地卡羅對局搜尋法 (MCTS) 2. 深捲積神經網路 (DCNN)



這些網路都需要一些訓練過程 才能得到適當的網路權重

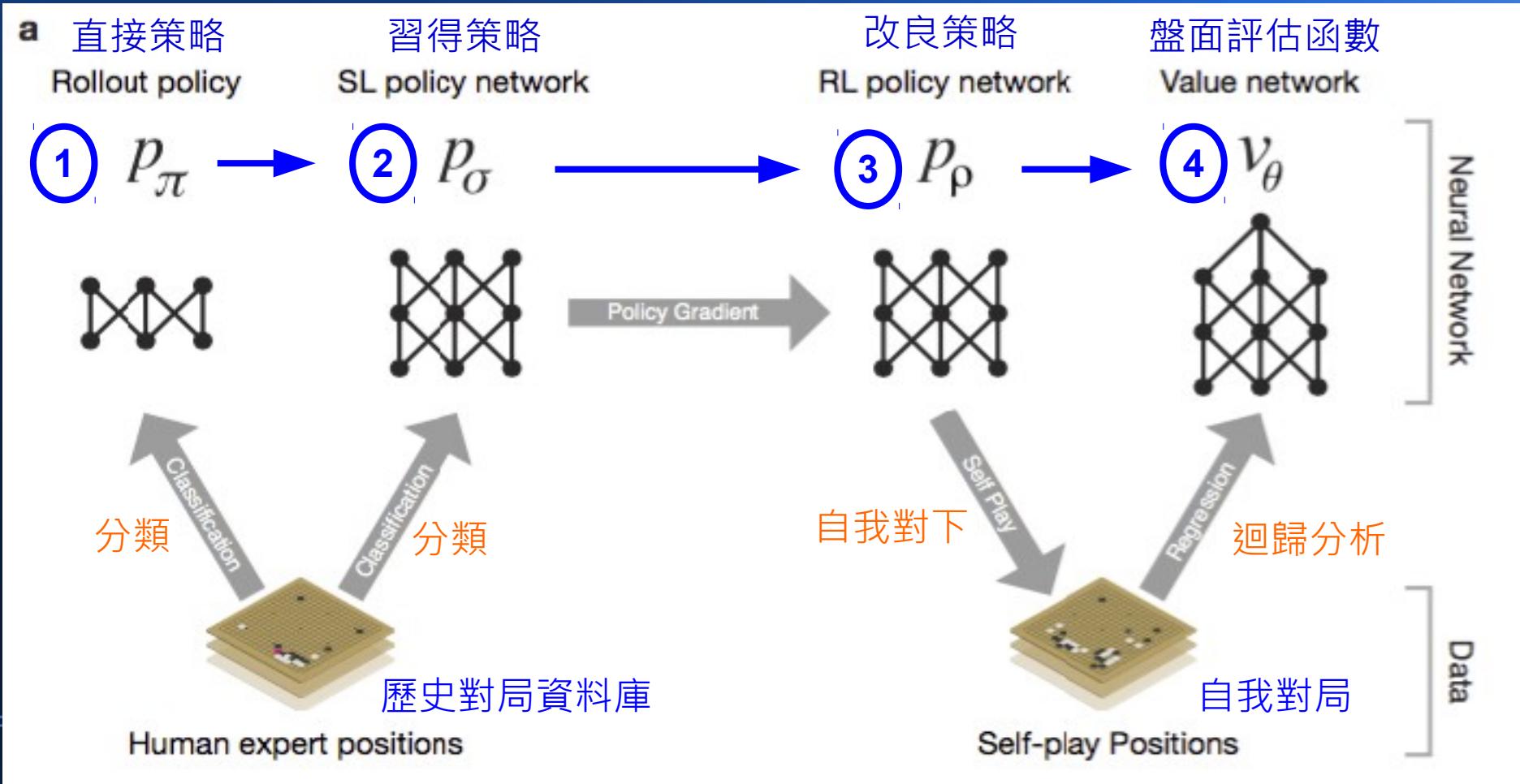
用《蒙地卡羅樹狀搜尋法 + 自我對下》去進一步訓練《深卷積神經網路》



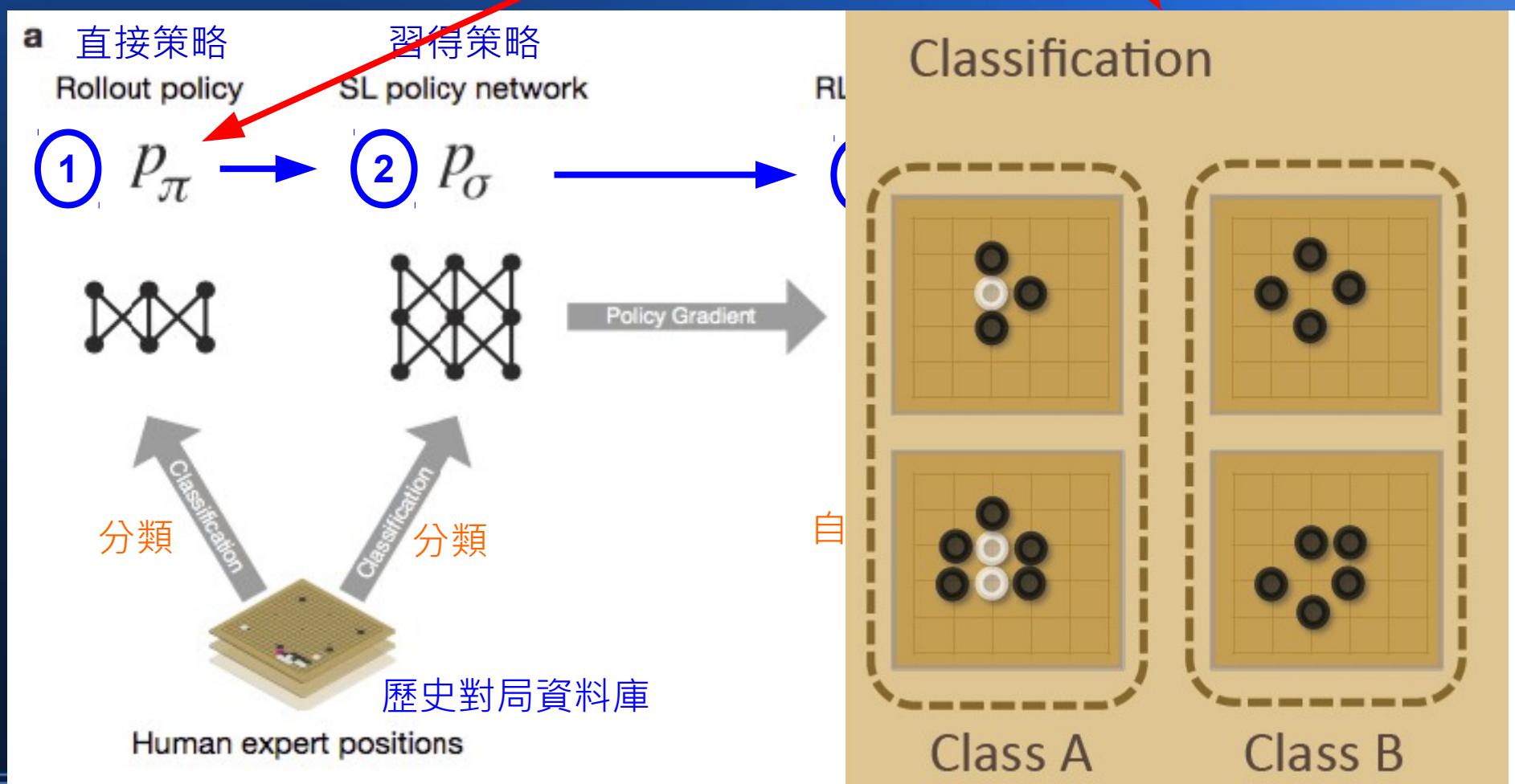
當然、這些訓練

- 得花不少時間，用電腦的程式去訓練調整網路的神經連結權重。
- 這就是 AlphaGo 的《深度學習》。

AlphaGo 當中最簡單的網路 應該是最左邊的《直接策略》

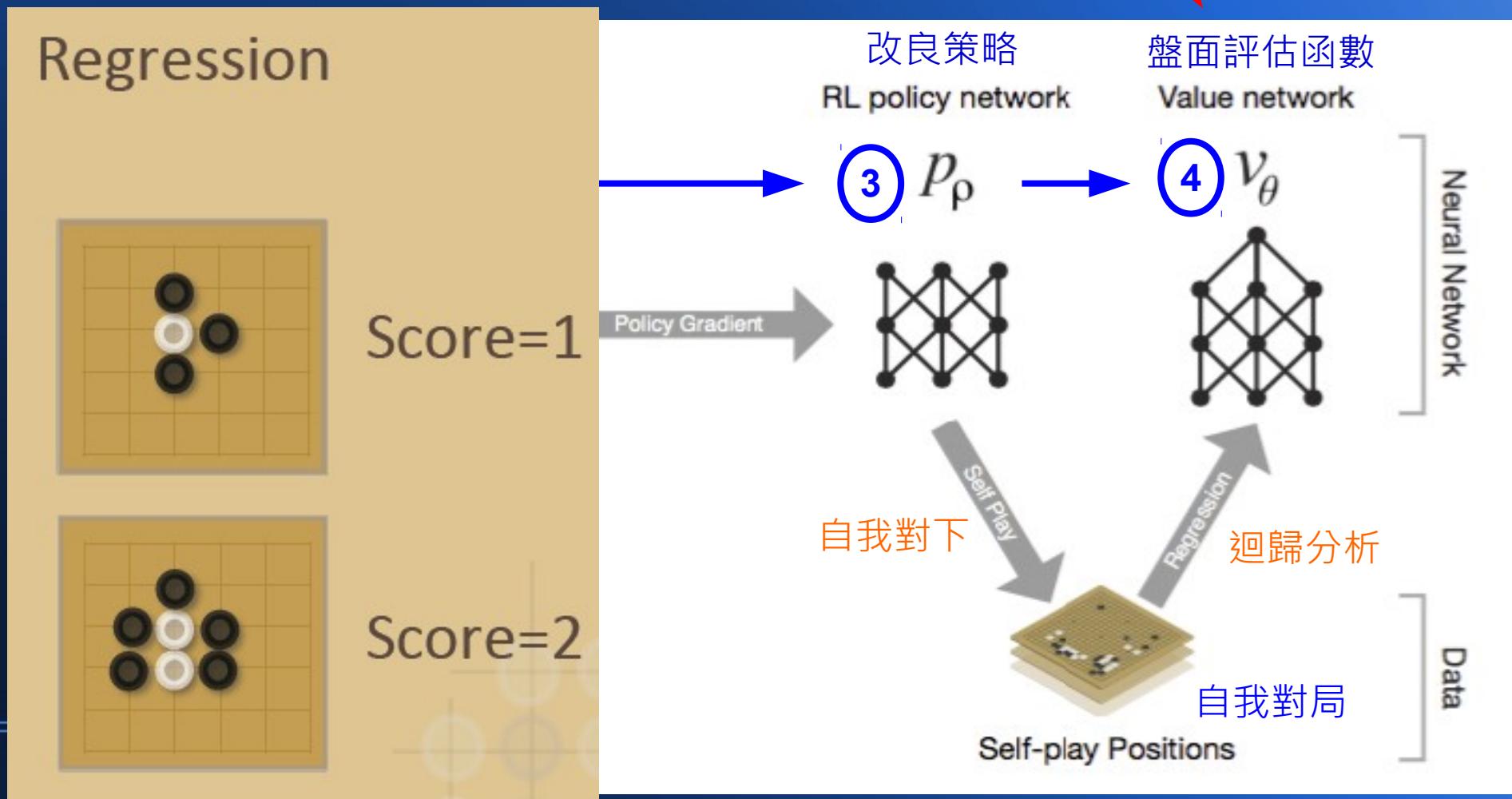


這個直接策略，應該是用簡單的捲積進行分類後所形成的



而最後的價值網路

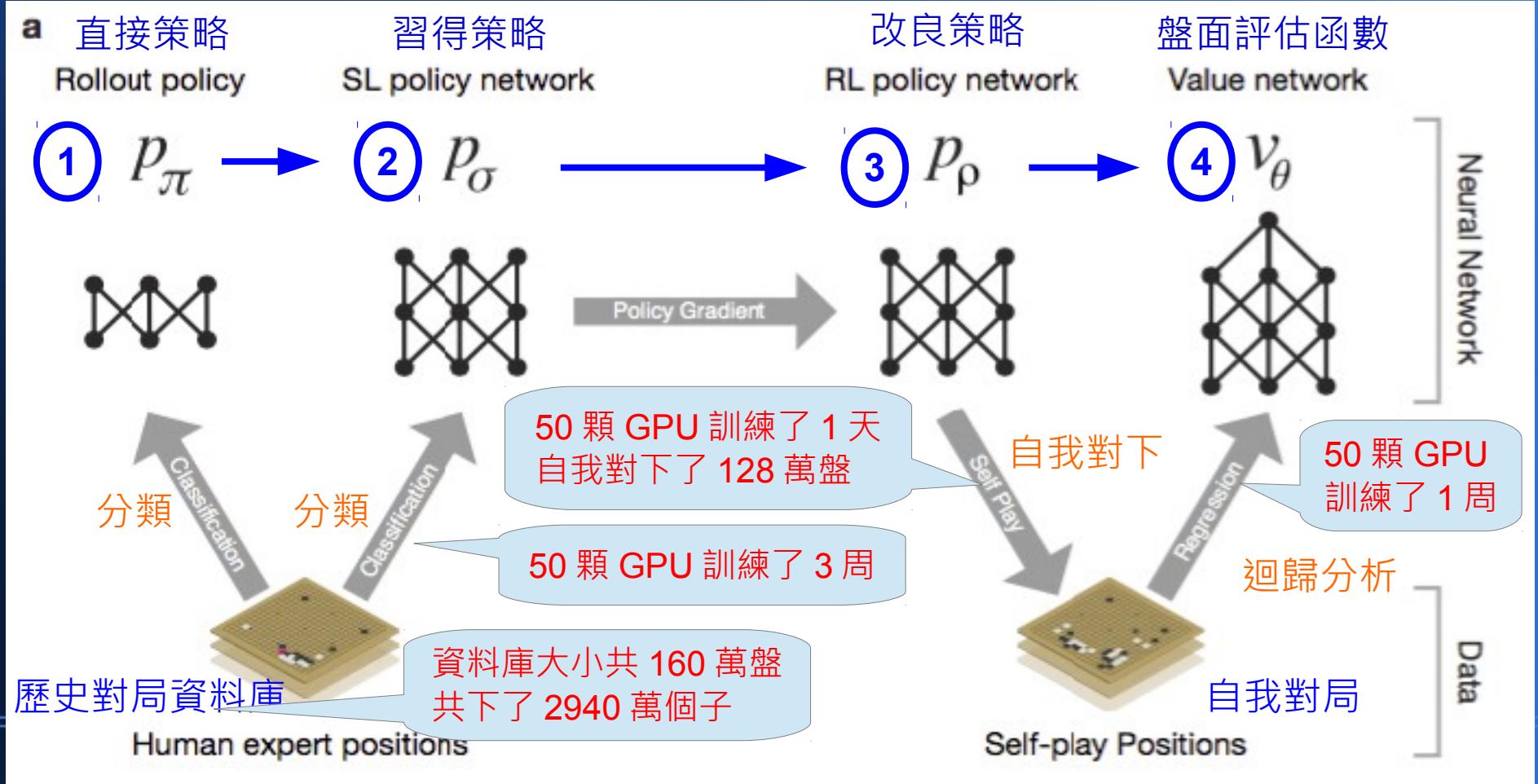
- 則是一個很好的《盤面評估函數》



整個 AlphaGo 系統的關鍵

- 基本上就是從對局資料庫開始
- 學習並改良《策略網路》
最後歸納出《價值網路》的過程

以下是 AlphaGo 訓練 所花的力氣與時間

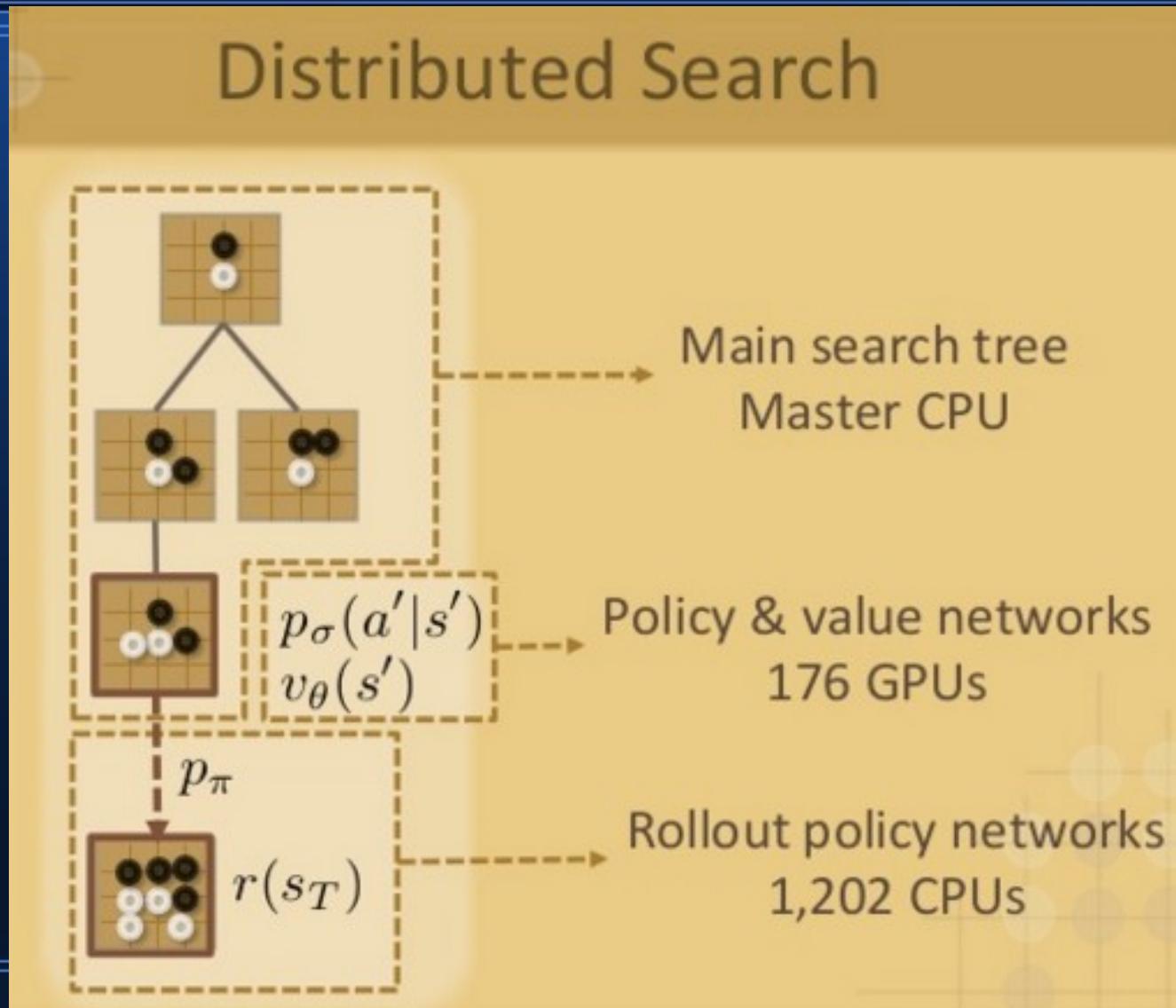


然後、在對下的當場



台灣「土博」黃士傑（左）操作親手研發的AlphaGo，與南韓世界圍棋棋王李世石（右）對弈。
(美聯社)

也有一千多顆處理器進行著計算



因此、這場對戰

- 對電腦可以說是非常費時費力的
- 並不像表面上看來那麼輕鬆！

在 AlphaGo 當中

- 有兩個關鍵的技術
 - 一個是深捲積神經網路
 - Deep Convolutional Neural Network (DCNN)
 - 一個是蒙地卡羅樹狀搜尋
 - Monte Carlo Tree Search (MCTS)

只要理解了這兩個技術

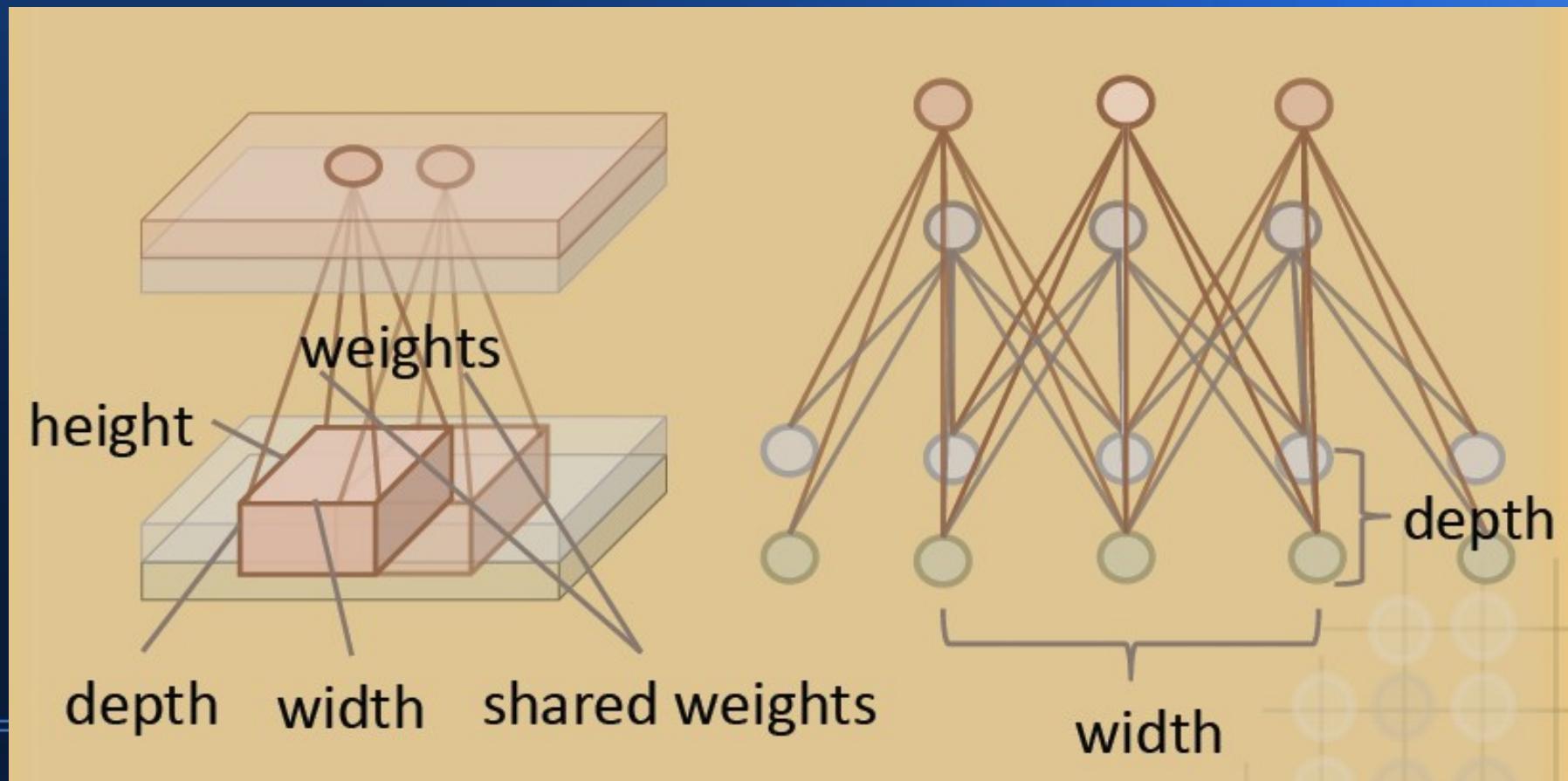
- 應該就能掌握 AlphaGo 的設計原理了！

接下來

- 我們將利用 Mark Chang 的 AlphaGo in Depth 這份投影片，來說明這兩個技術的原理。

首先讓我們看看

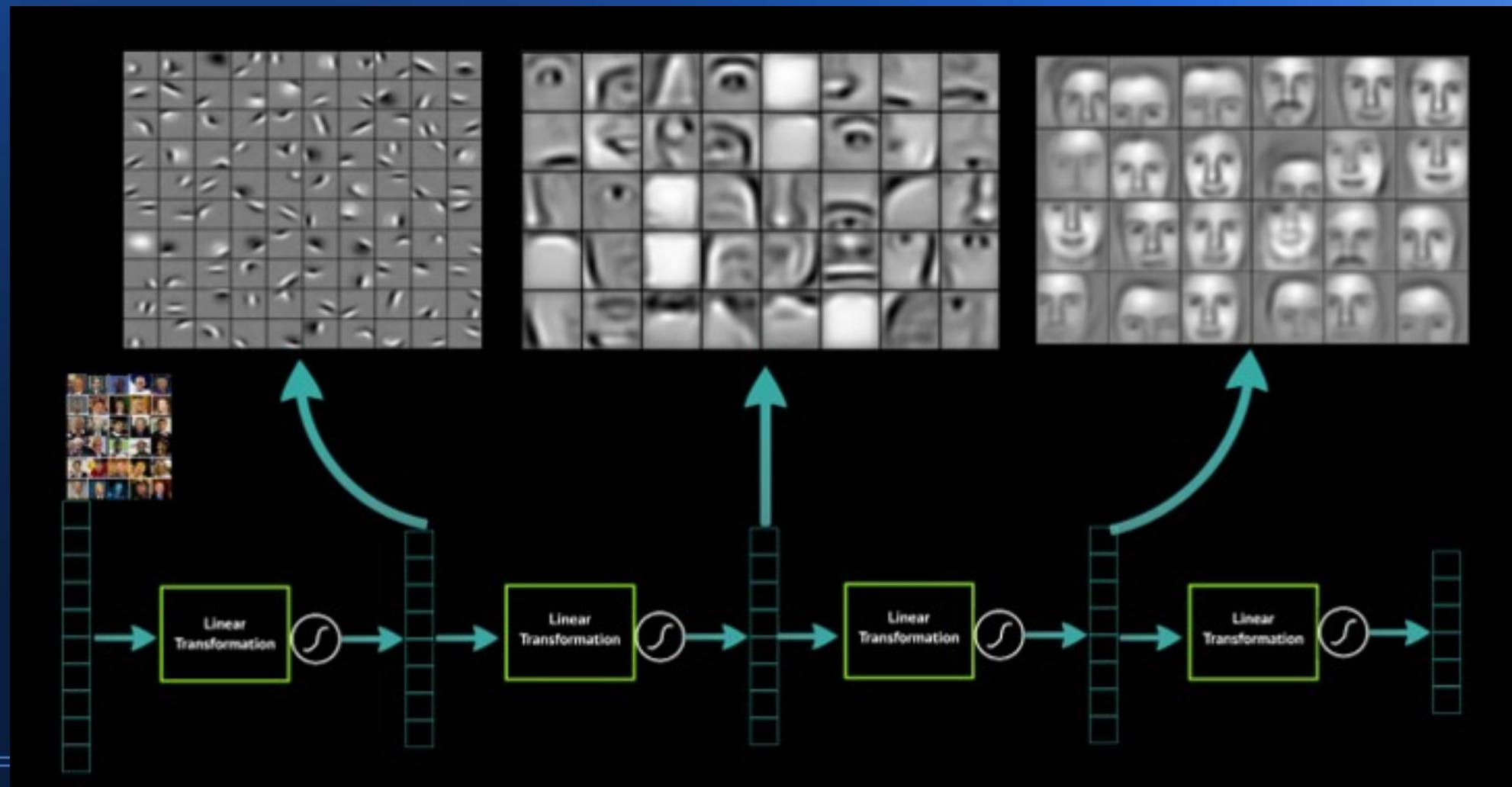
- 深捲積神經網路的運作原理



原本的《捲積神經網路》

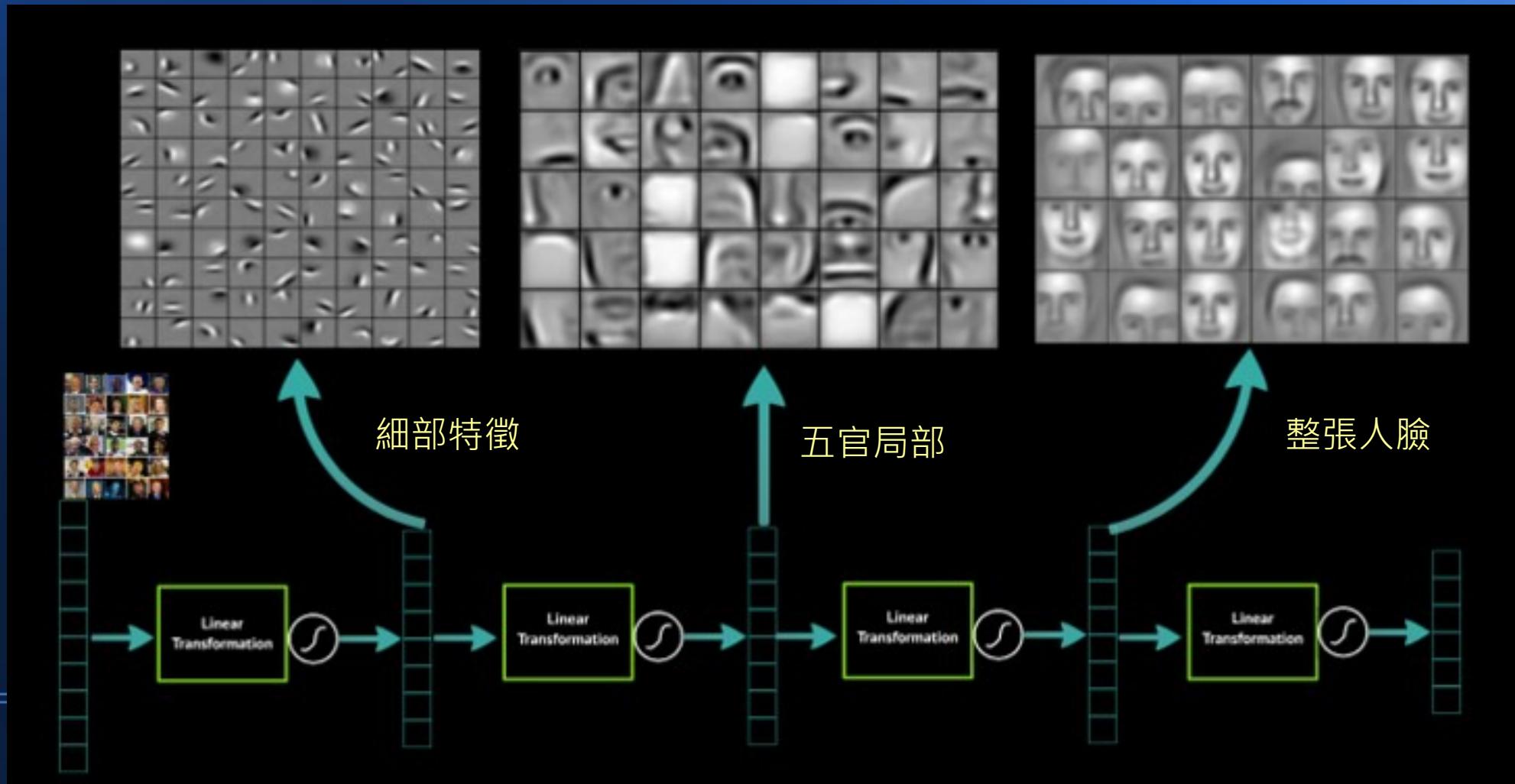
- 通常被用在影像處理上，進行影像的辨識與分類！

透過局部的捲積樣式 逐步組合出整體的圖像



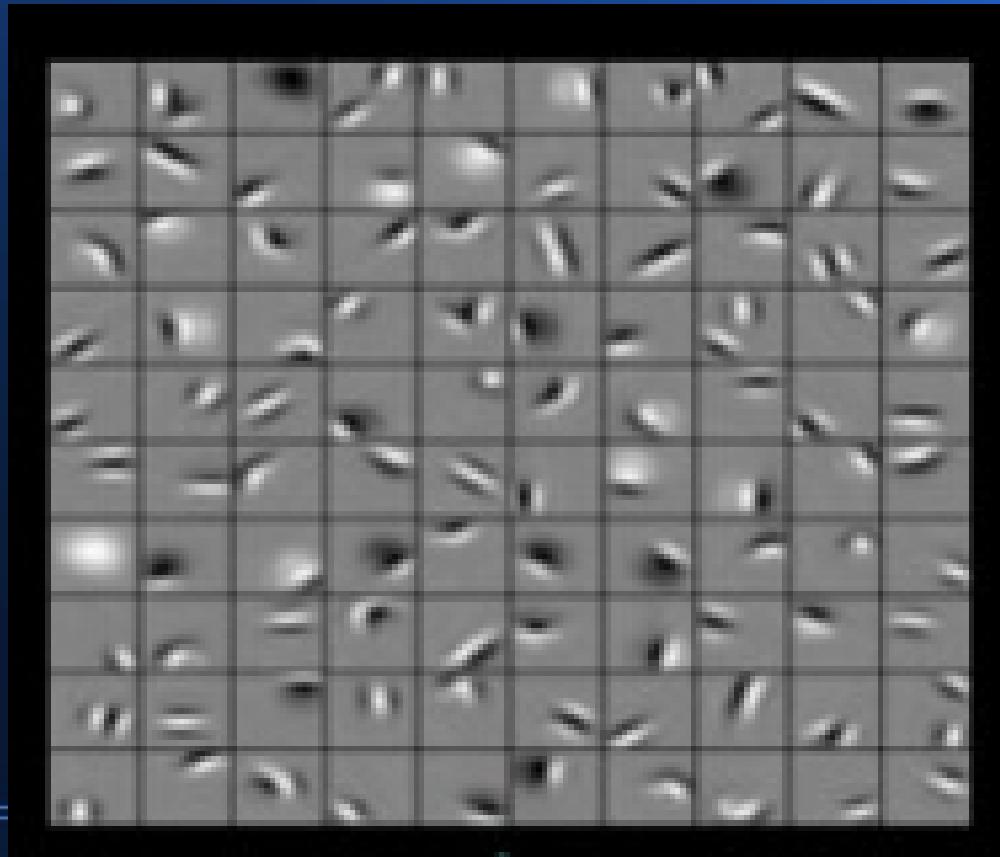
您可以看到下圖中

- 細部的特徵逐步被組合成五官，然後組出人臉



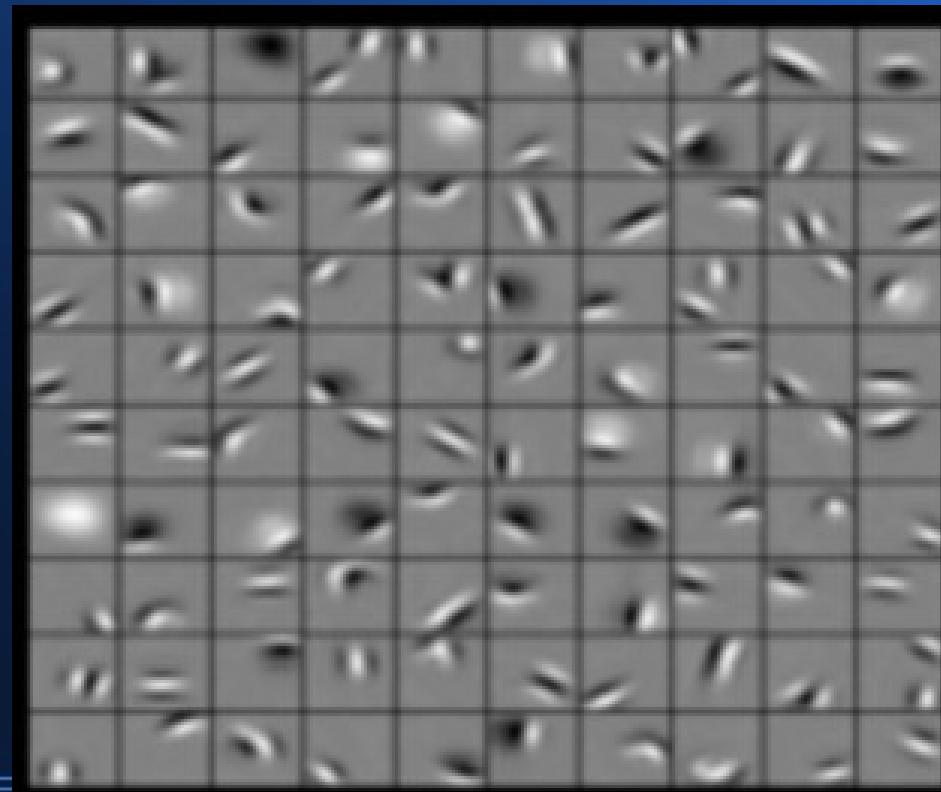
其中的細部特徵

- 就是利用特定樣式開始進行捲積的



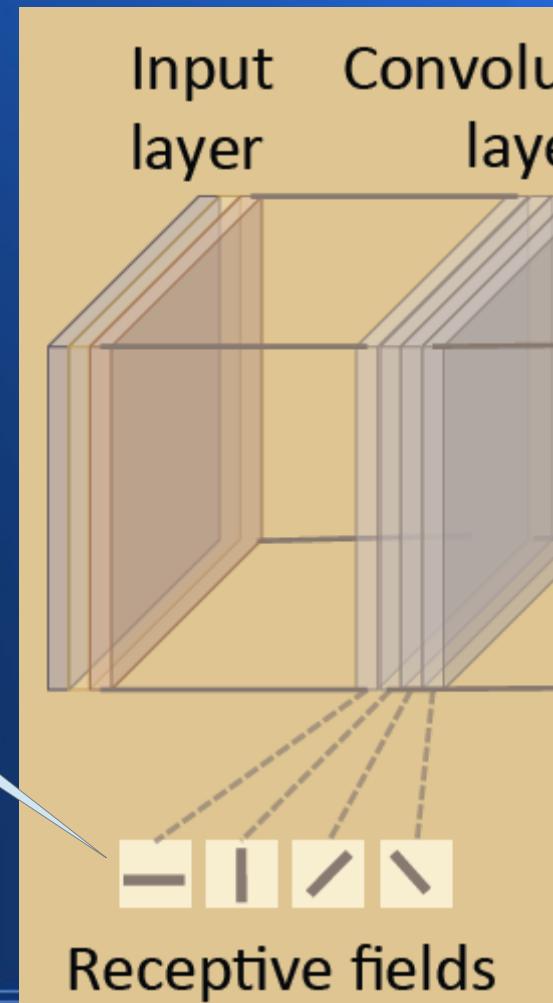
這些特定樣式

- 包含了一些細線，還有區塊



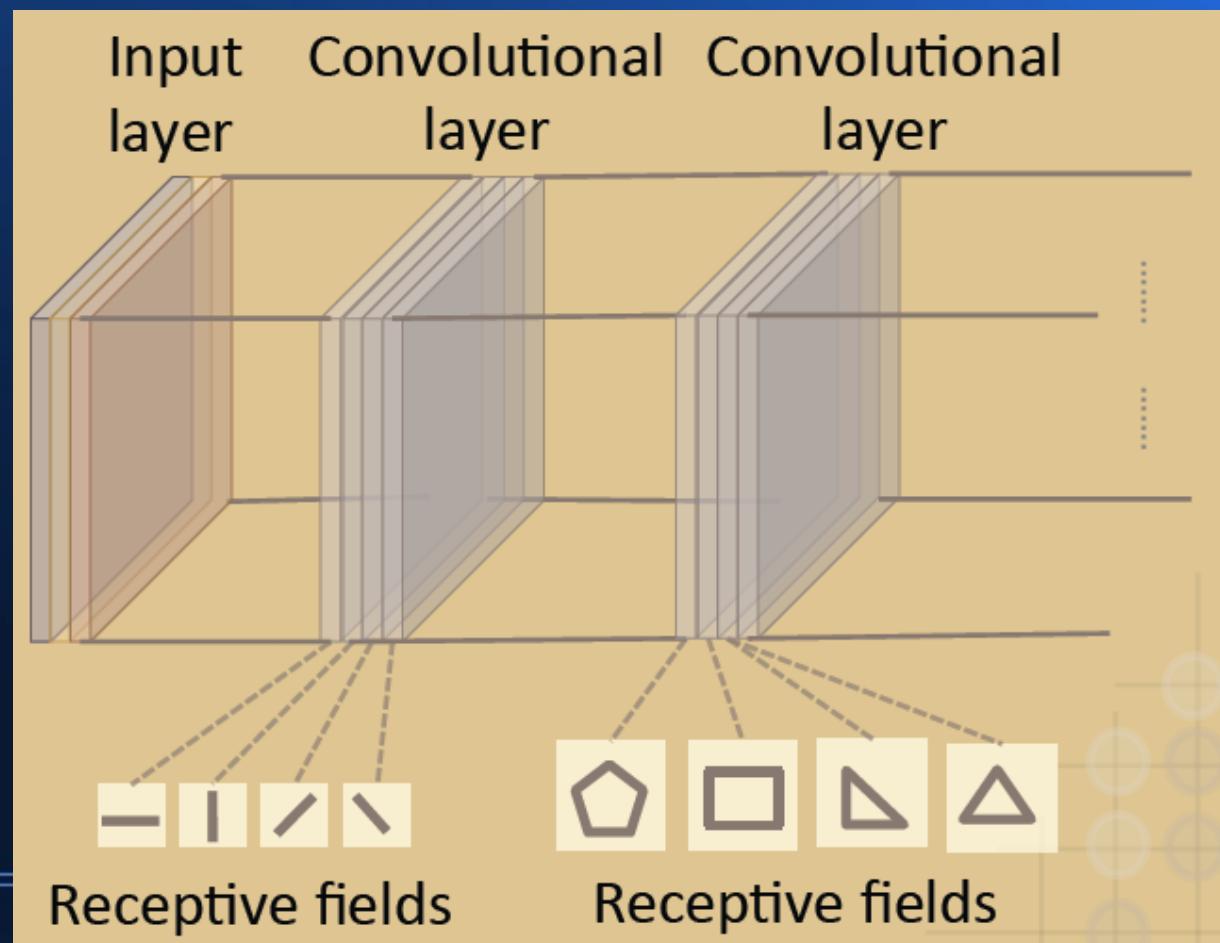
在五子棋中，這些細線本身
就可以用來評判分數

捲積樣式



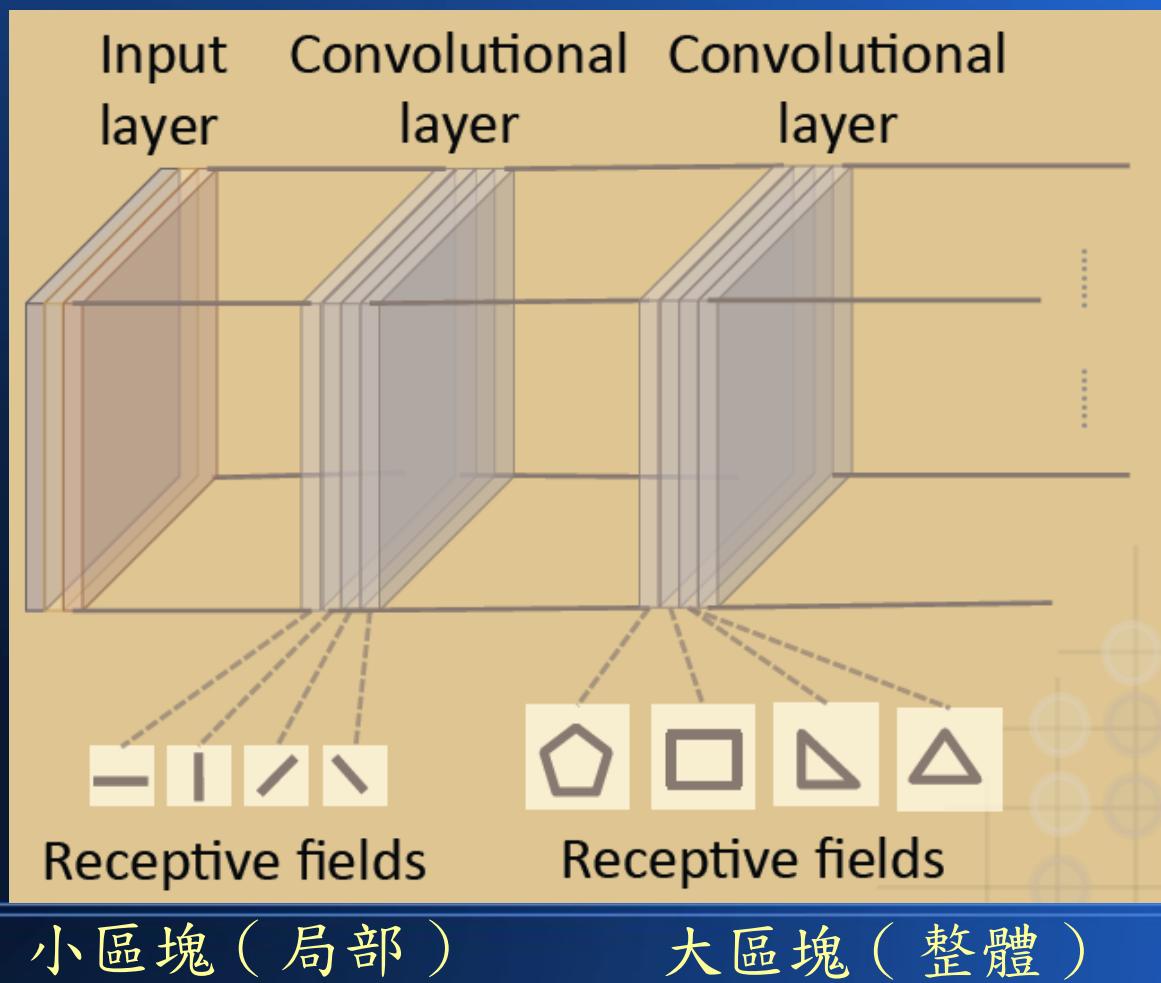
在圍棋中，細線的資訊
會進一步被組合成多邊形

- 這樣才能判斷是否圍住一個區塊！



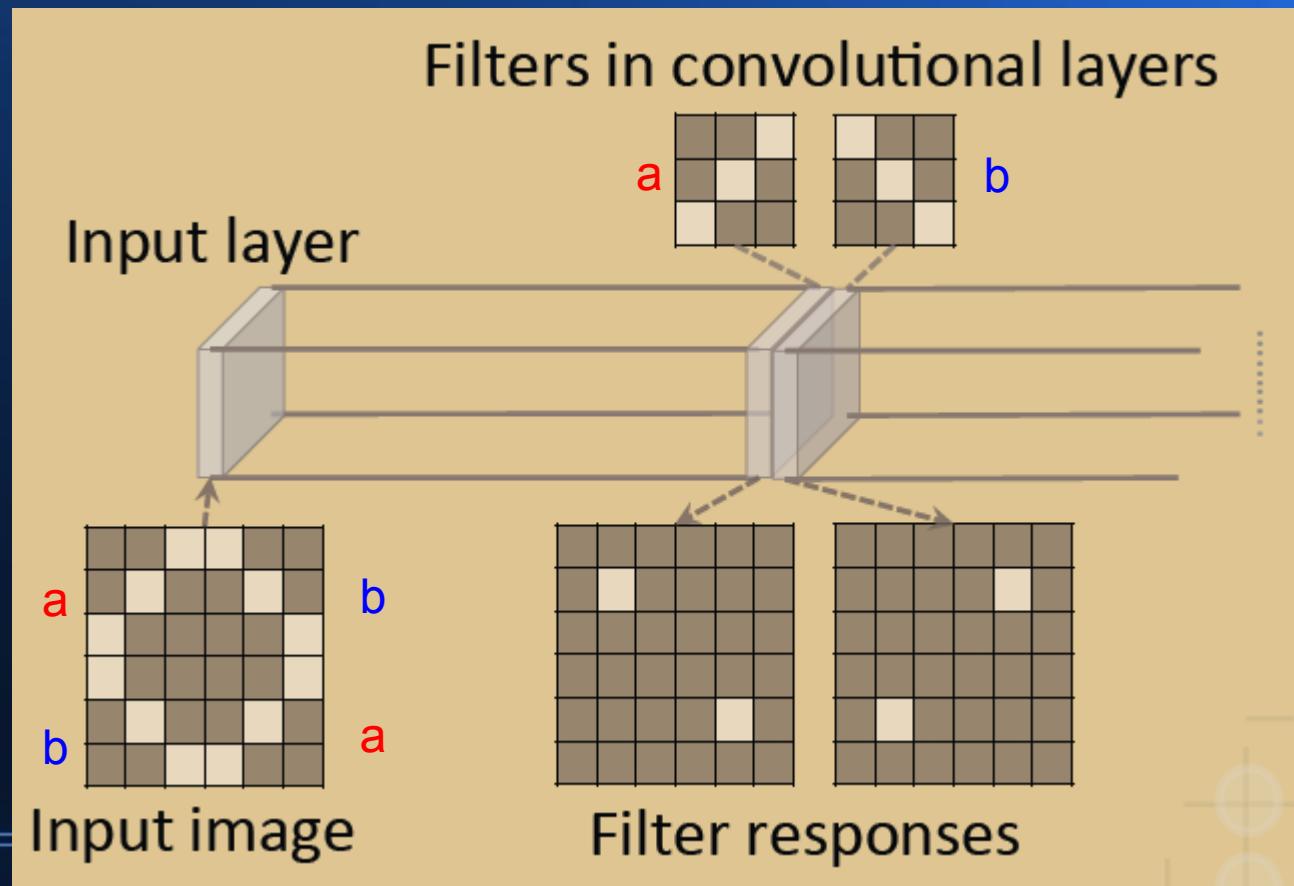
透過捲積神經網路的連結

- 可以將小區塊逐步向上組合成大區塊，進而代表整體



舉例而言

- 我們可以用斜線組合成菱形



所謂的《卷積》是透過《區塊樣式》與《整體》之間逐區塊地進行《相乘後加總》的動作

1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>	0	0
0 <small>$\times 0$</small>	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	1	0
0 <small>$\times 1$</small>	0 <small>$\times 0$</small>	1 <small>$\times 1$</small>	1	1
0	0	1	1	0
0	1	1	0	0

4		

1	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	0 <small>$\times 1$</small>	0
0	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	0
0	0 <small>$\times 1$</small>	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>	1
0	0	1	1	0
0	1	1	0	0

4	3	

1	1	1	0	0
0	1	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	0 <small>$\times 1$</small>
0	0	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>
0	0	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	0 <small>$\times 1$</small>
0	1	1	0	0

4	3	4
2	4	3

1	1	1	0	0
0	1	1	1	0
0	0	1 <small>$\times 1$</small>	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>
0	0	1 <small>$\times 0$</small>	1 <small>$\times 1$</small>	0 <small>$\times 0$</small>
0	1	1 <small>$\times 1$</small>	0 <small>$\times 0$</small>	0 <small>$\times 1$</small>

4	3	4
2	4	3
2	3	4

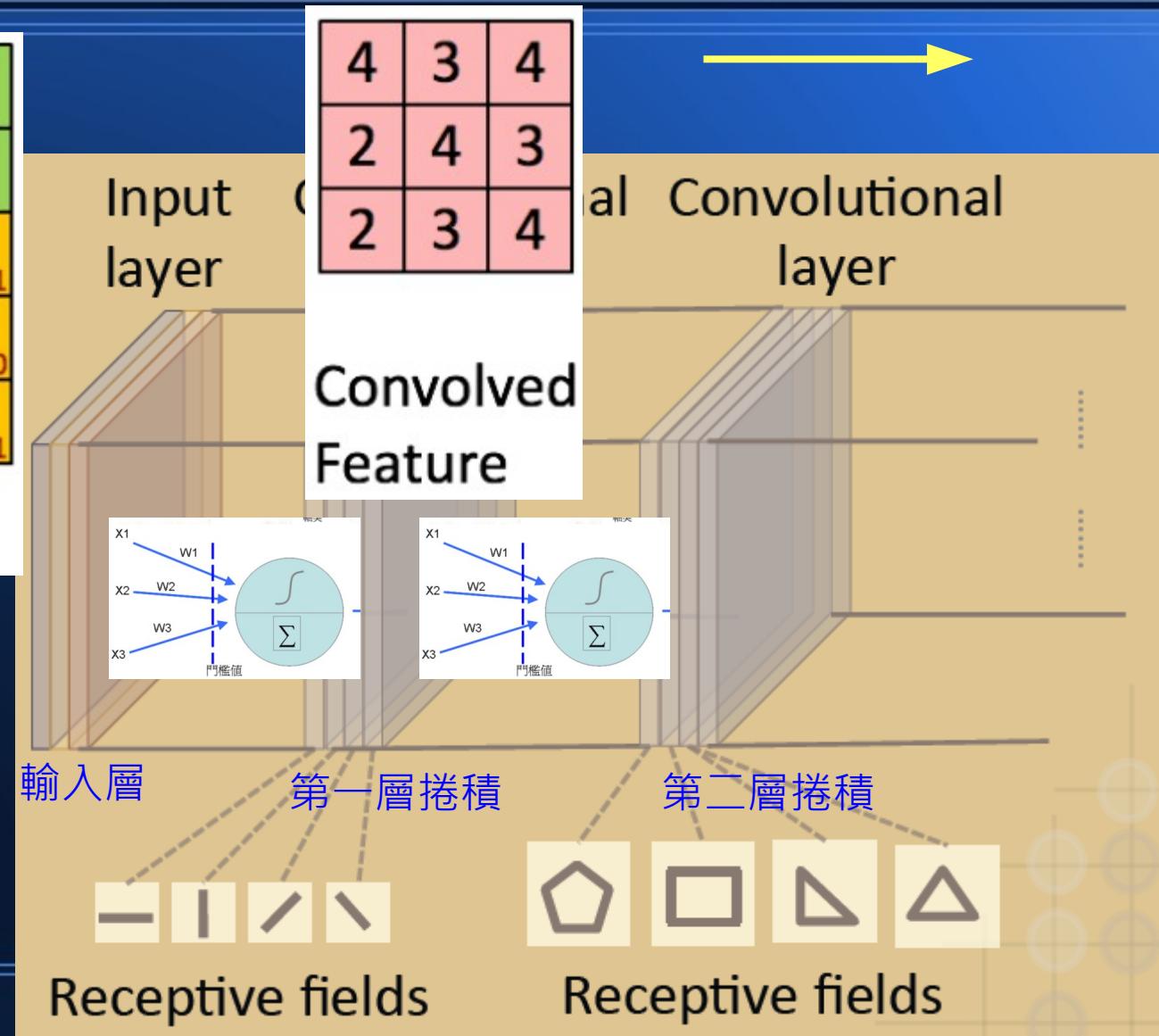
Image

Convolved
Feature

這種捲積所得到的大量統計資訊
可以讓下一層神經網路取用並組合

1	1	1	0	0
0	1	1	1	0
0	0	1 _{x1}	1 _{x0}	1 _{x1}
0	0	1 _{x0}	1 _{x1}	0 _{x0}
0	1	1 _{x1}	0 _{x0}	0 _{x1}

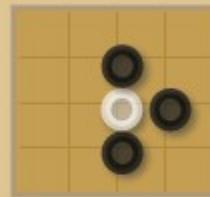
Image



但必須注意的是 AlphaGo 使用的
捲積樣式是《為圍棋所特製》的

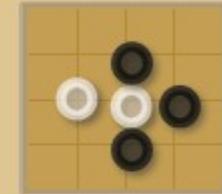
Input/Output Data

Input



Stone color, Liberty, Turns
since, Capture size,
Self-atari size, Ladder capture,
Ladder escape, Sensibleness.
Total: 48 planes

Output



Next
position

Stone color:
3 planes
player, opponent, empty

0	0	0
0	1	0
0	0	0

0	1	0
0	0	1
0	1	0

1	0	1
1	0	0
1	0	1

Liberty:
8 planes
1~8 liberties

0	0	0
0	1	0
0	0	0

0	0	0
0	0	0
0	0	0

0	1	0
0	0	1
0	1	0

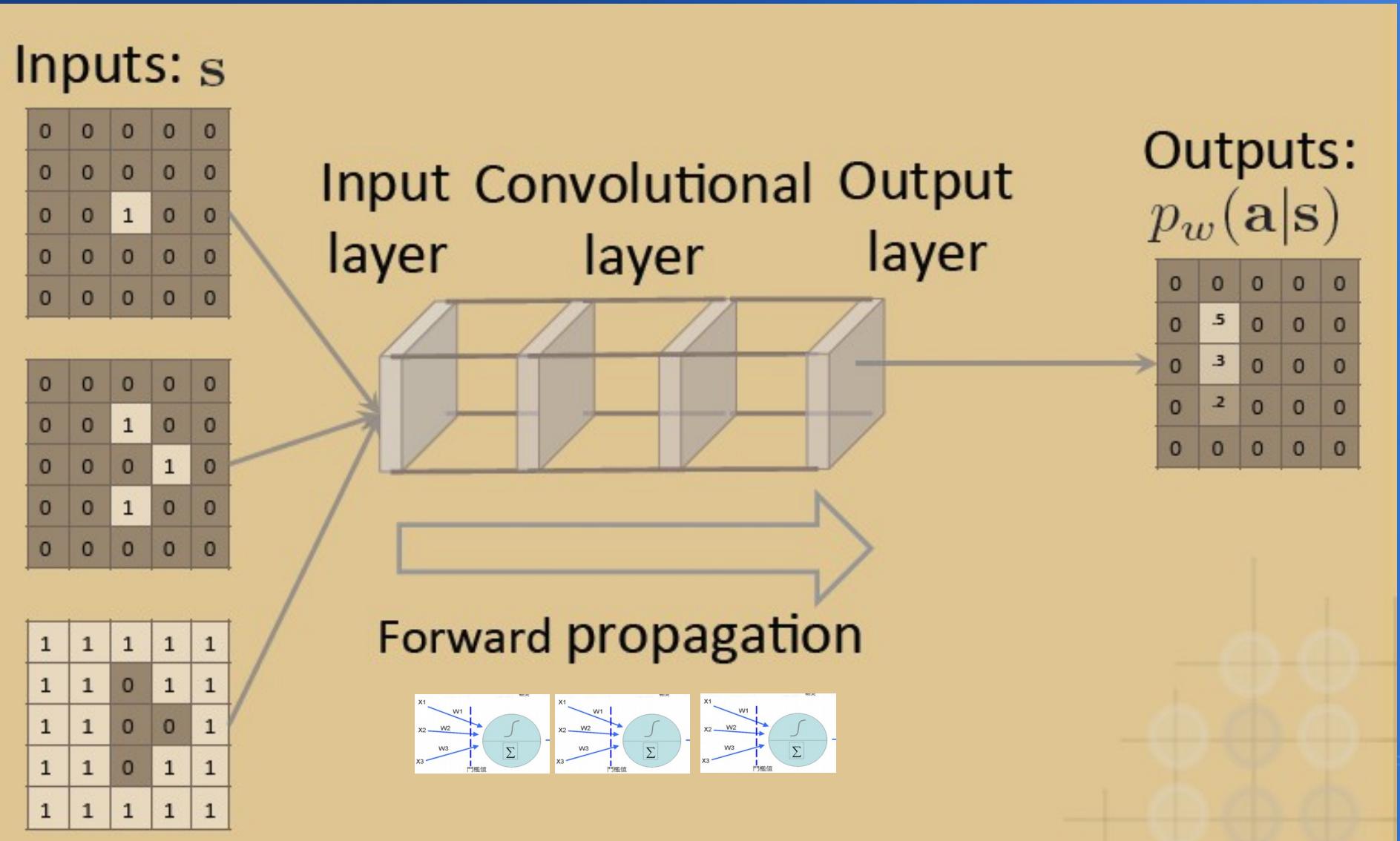
0	0	0
0	0	0
0	0	0

以下是所有第一層的捲積平面

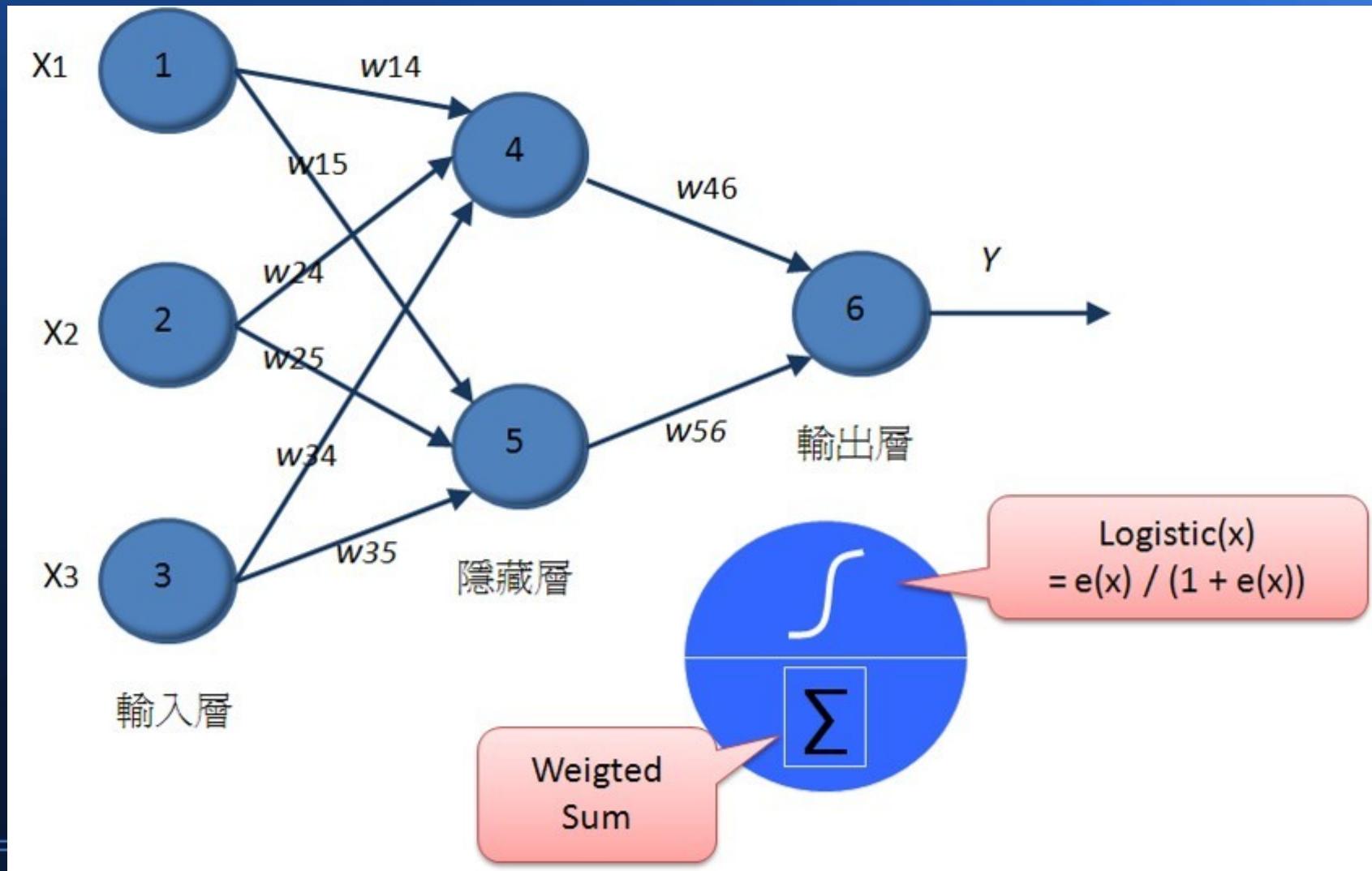
Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Extended Data Table 2: **Input features for neural networks.** Feature planes used by the policy network (all but last feature) and value network (all features).

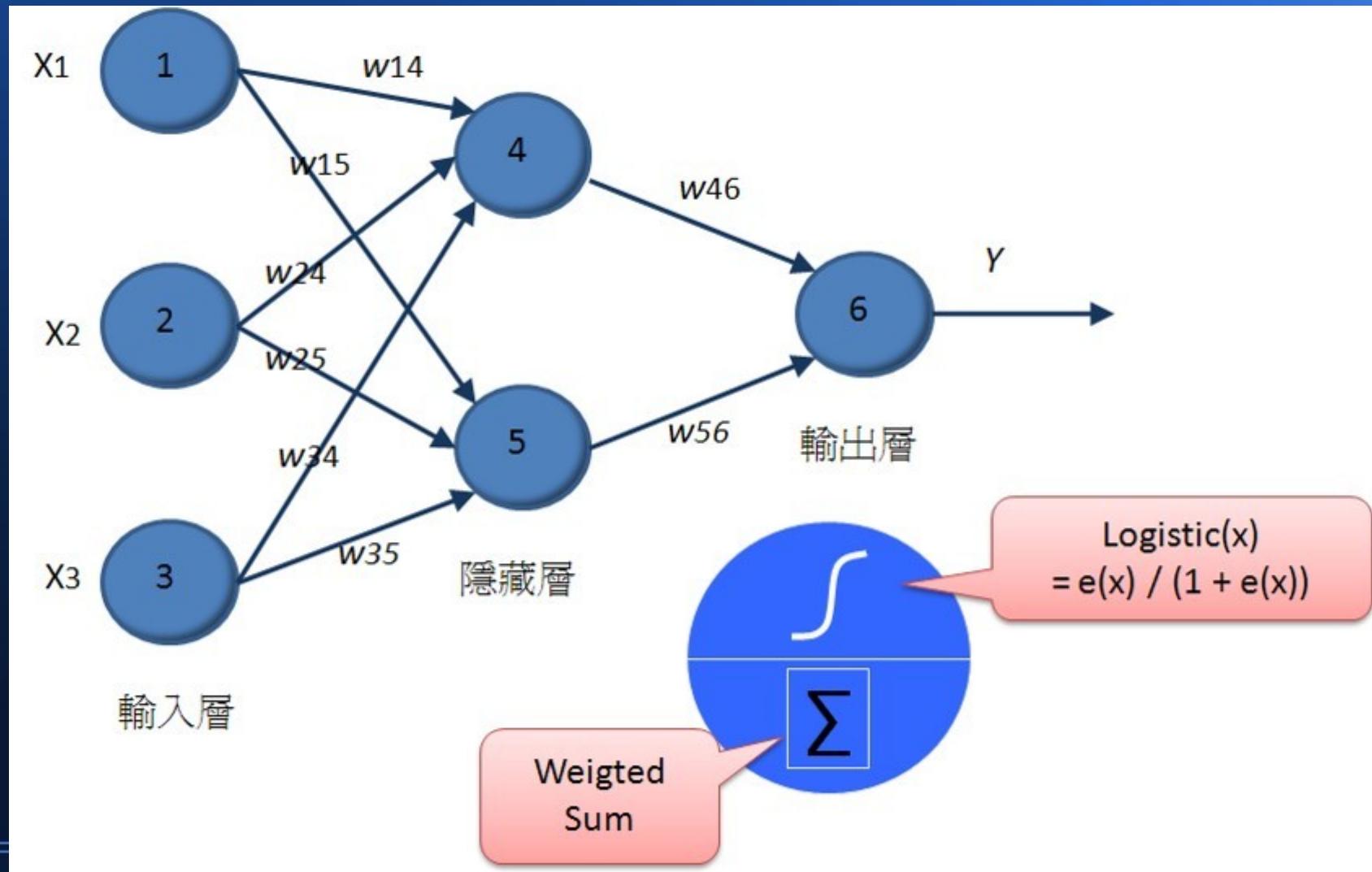
AlphaGo 透過這種方式從小區塊組合出整體的《策略網路》



其中每一層之間都有《神經線》連接著



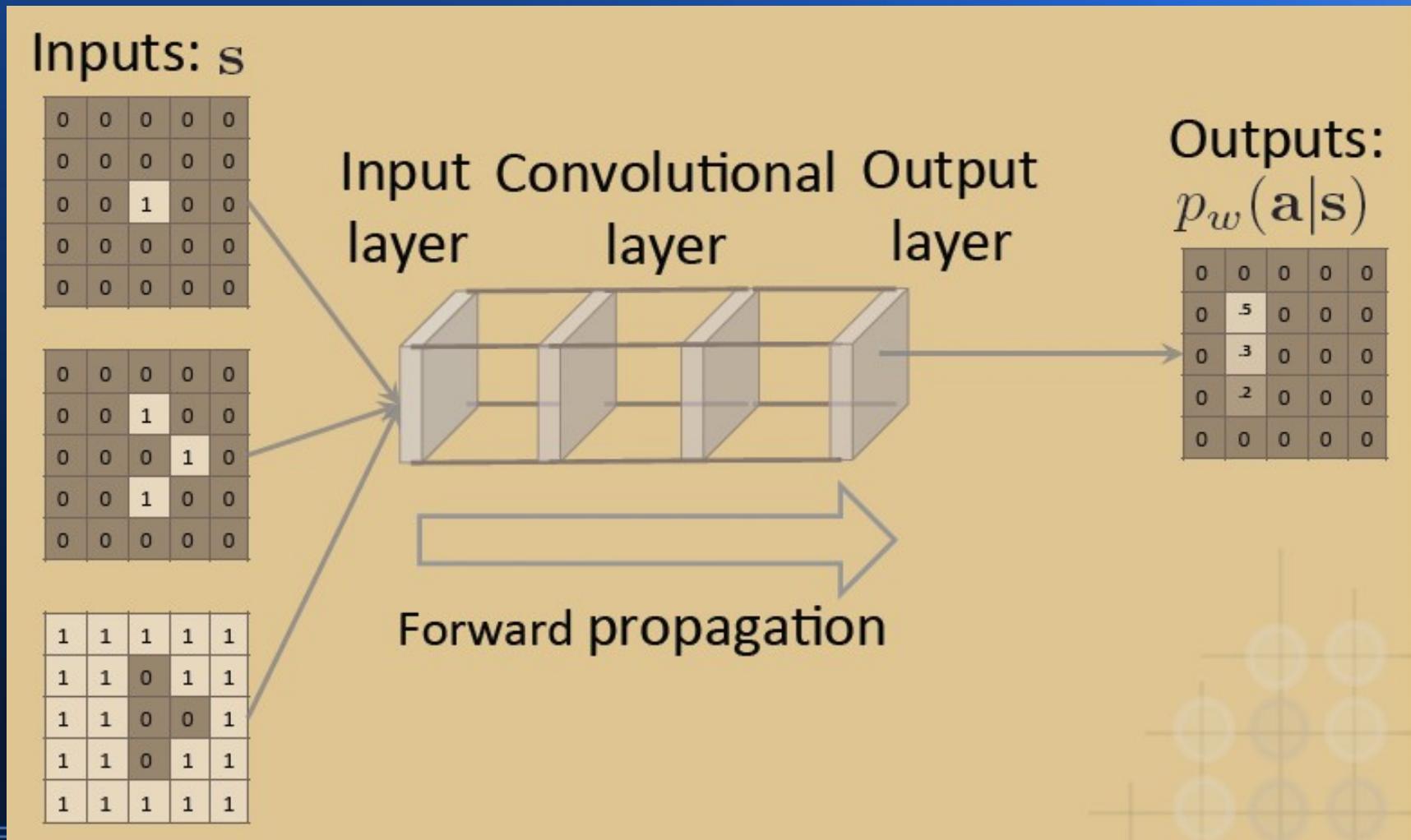
只要用《梯度下降法》調整《神經線的權重》
就能訓練好神經網路



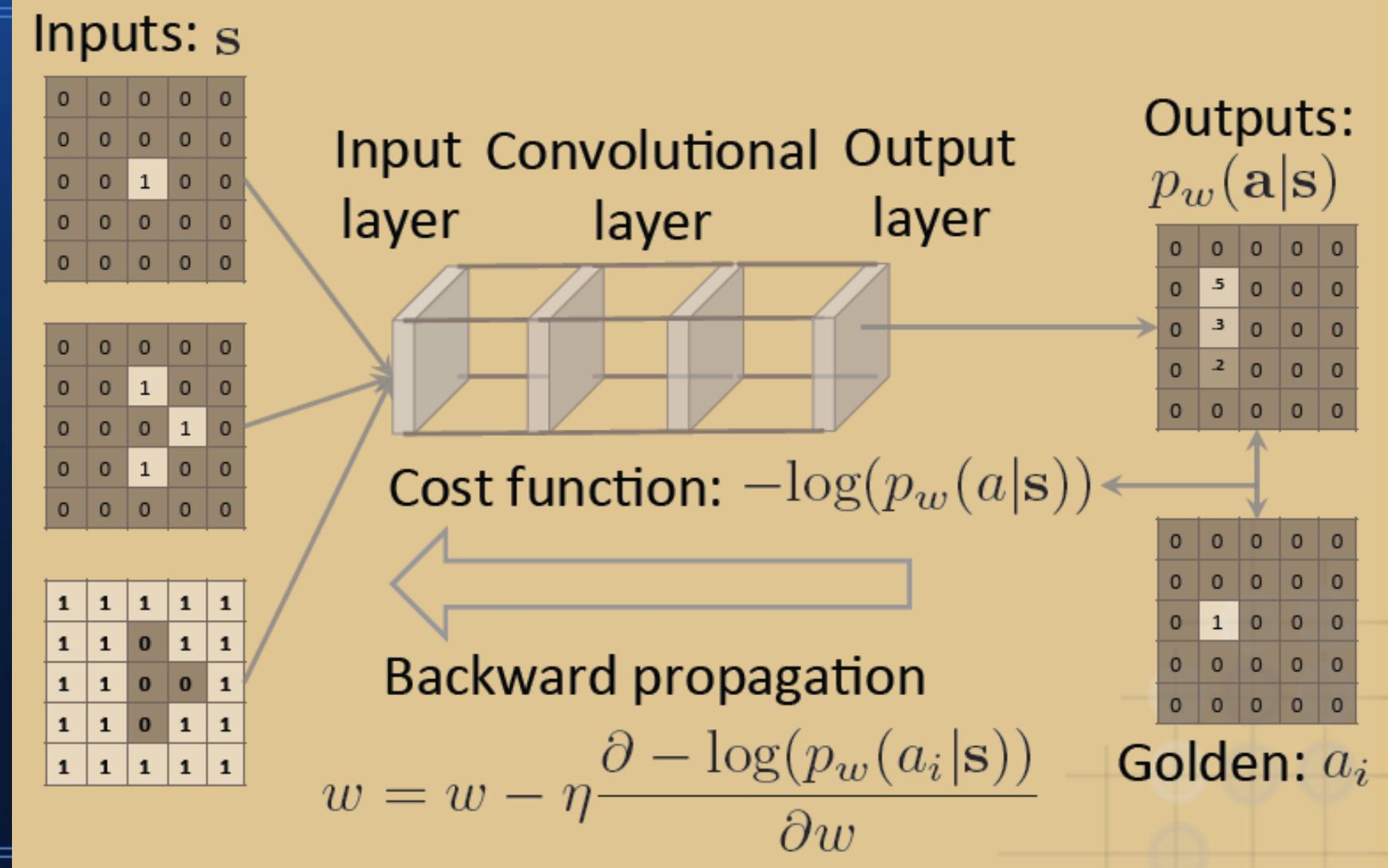
神經網路的訓練分為兩階段

- 第一階段是前向傳遞
 - 計算目前網路的輸出結果
- 第二階段是反向傳遞
 - 用來根據與標準答案間的差距，調整網路權重。

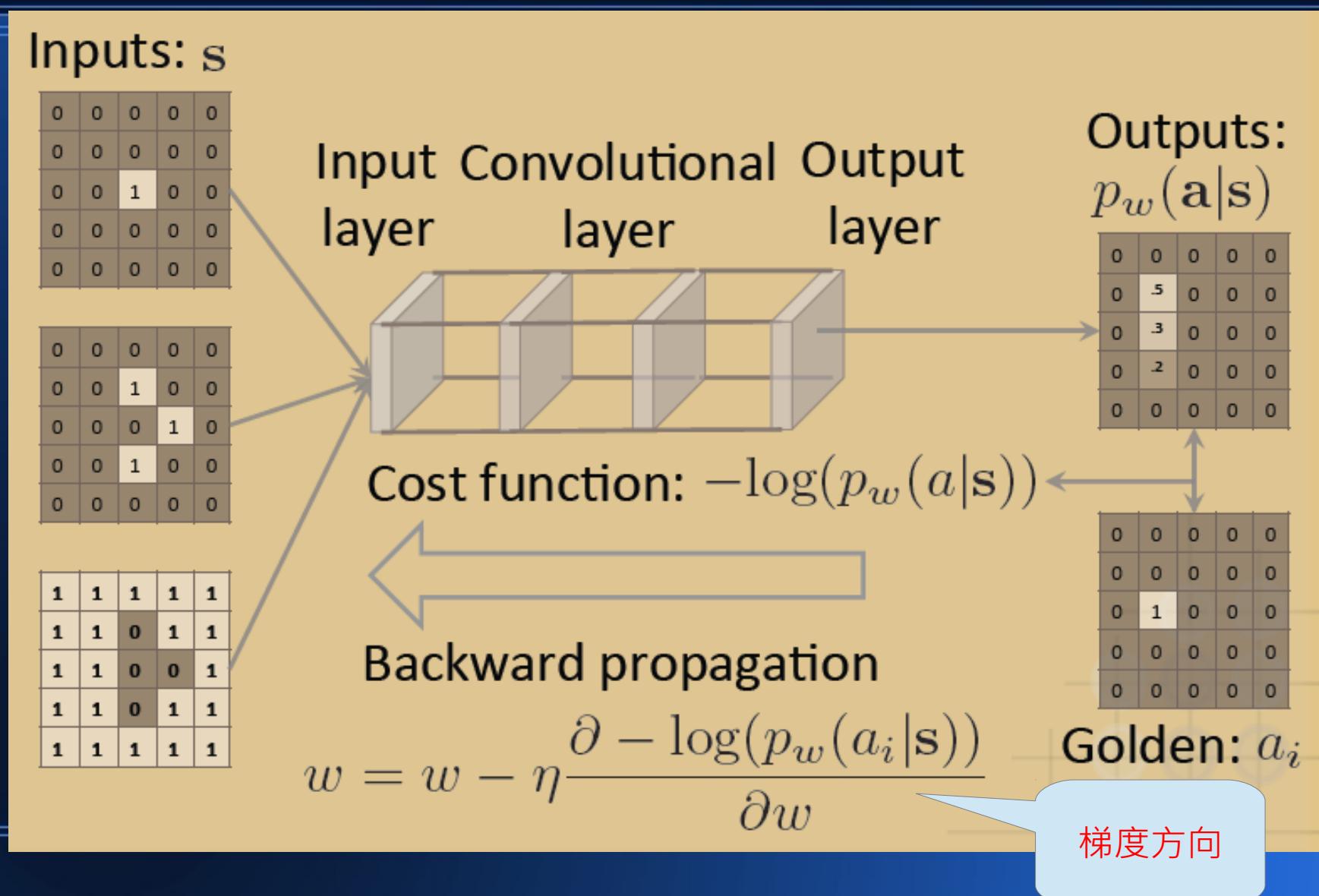
第一階段：前向傳遞



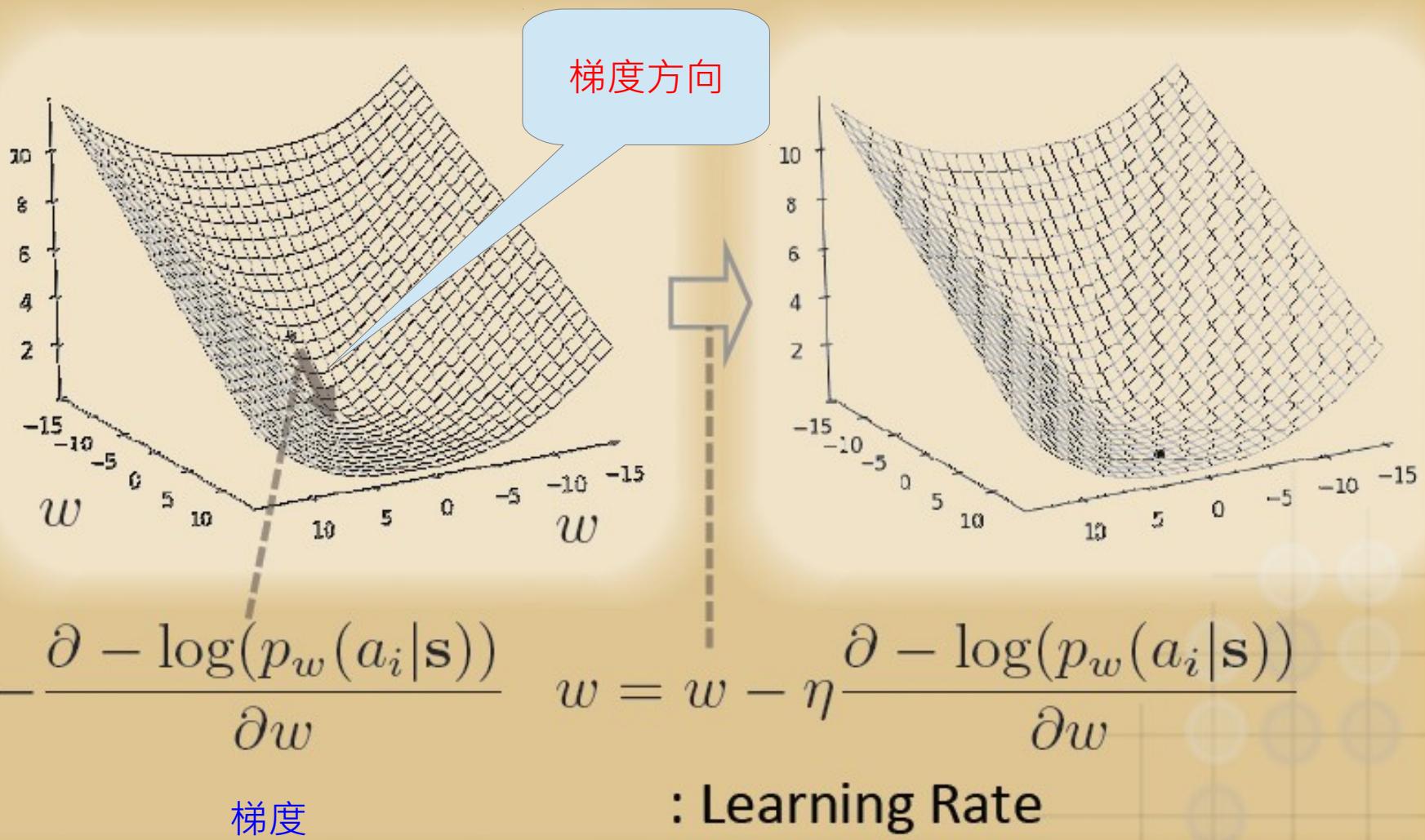
第二階段 反向傳遞



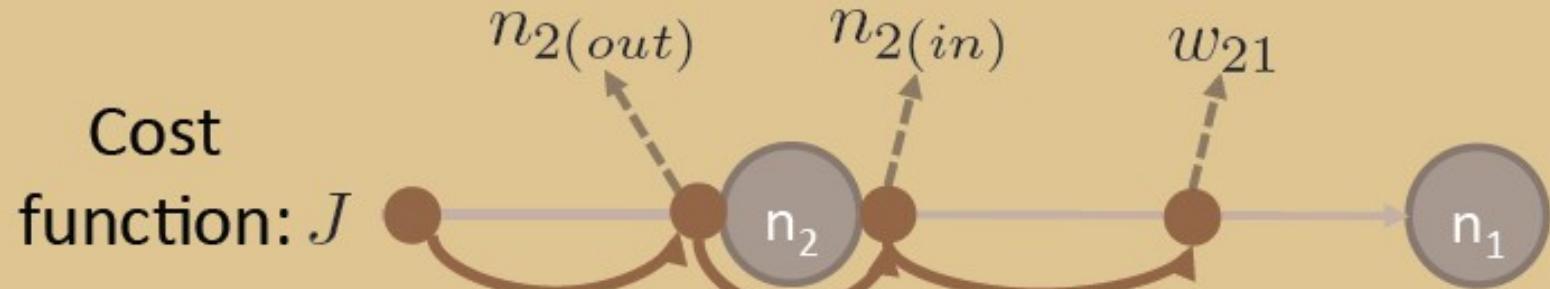
在反向傳遞時，採用偏微分方式
朝《梯度方向》調整，因此稱為《梯度下降法》



梯度下降法



更詳細的梯度調整公式



$$\frac{\partial J}{\partial n_{2(out)}} = \frac{\partial J}{\partial n_{2(out)}} \frac{\partial n_{2(out)}}{\partial n_{2(in)}} \frac{\partial n_{2(in)}}{\partial w_{21}}$$

$$w_{21} \leftarrow w_{21} - \eta \frac{\partial J}{\partial w_{21}}$$

$$w_{21} \leftarrow w_{21} - \eta \frac{\partial J}{\partial n_{2(out)}} \frac{\partial n_{2(out)}}{\partial n_{2(in)}} \frac{\partial n_{2(in)}}{\partial w_{21}}$$

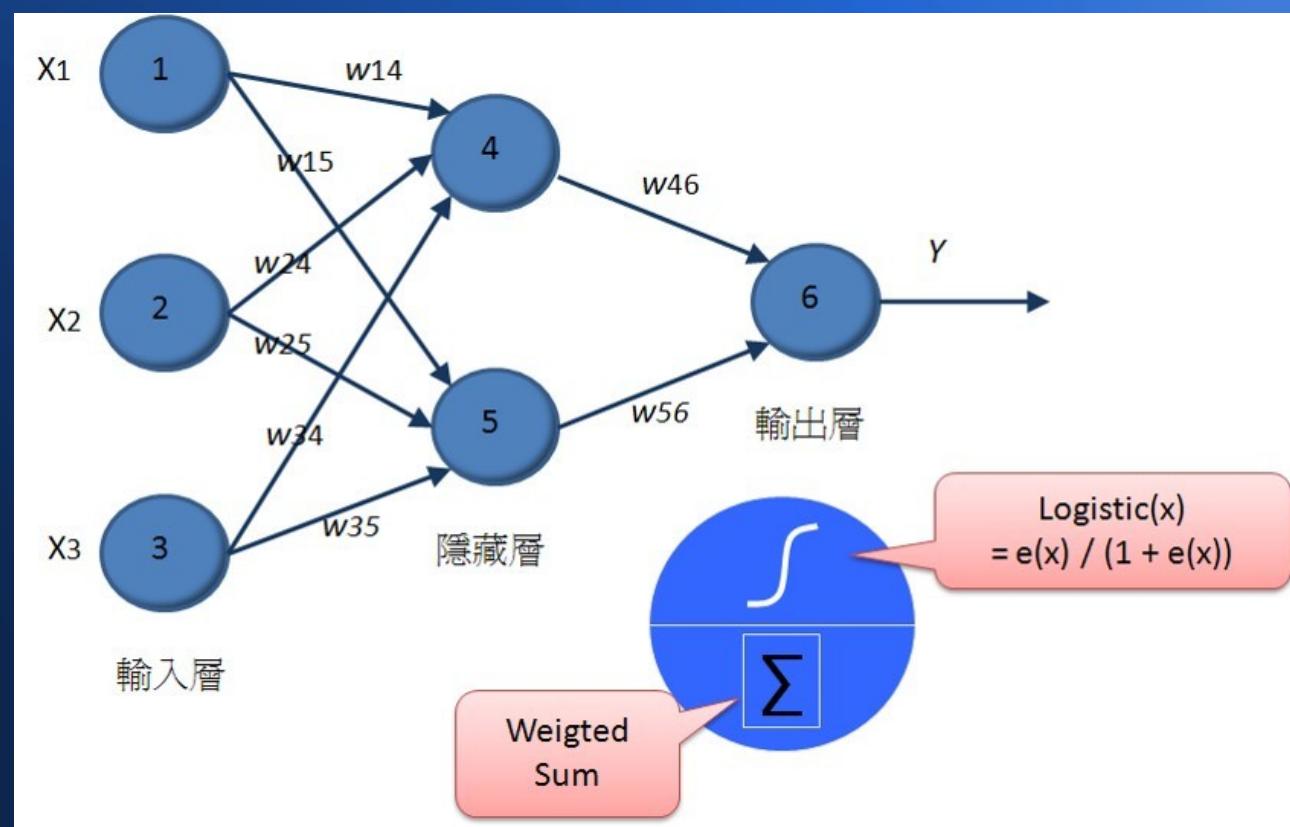
在此、讓我們更詳細的介紹一下

- 最基礎的神經網路運作原理
- 以便讓對神經網路不熟的讀者可以理解《到底神經網路是甚麼》？

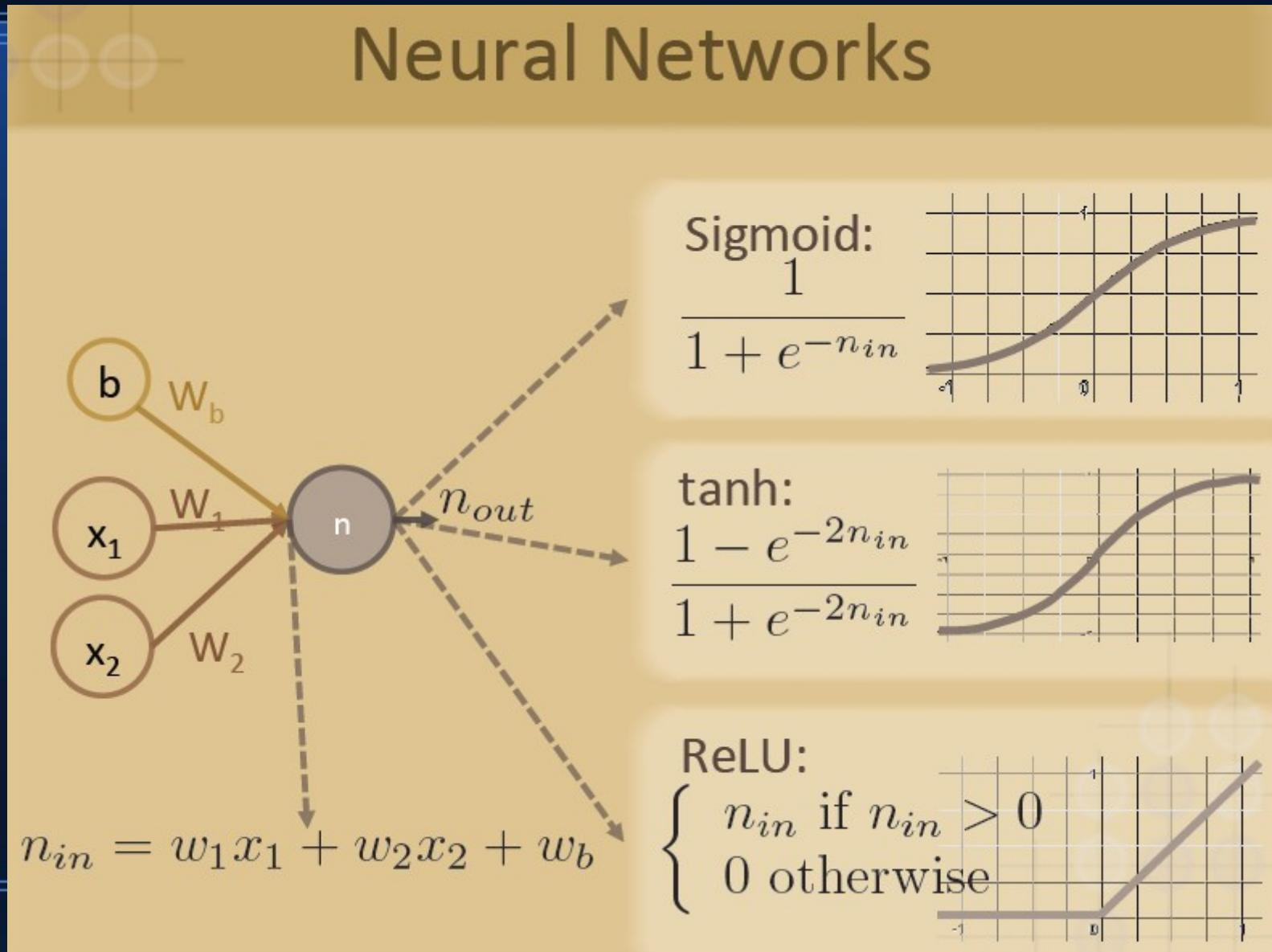
所謂的神經網路

包含一堆《節點》（神經元）

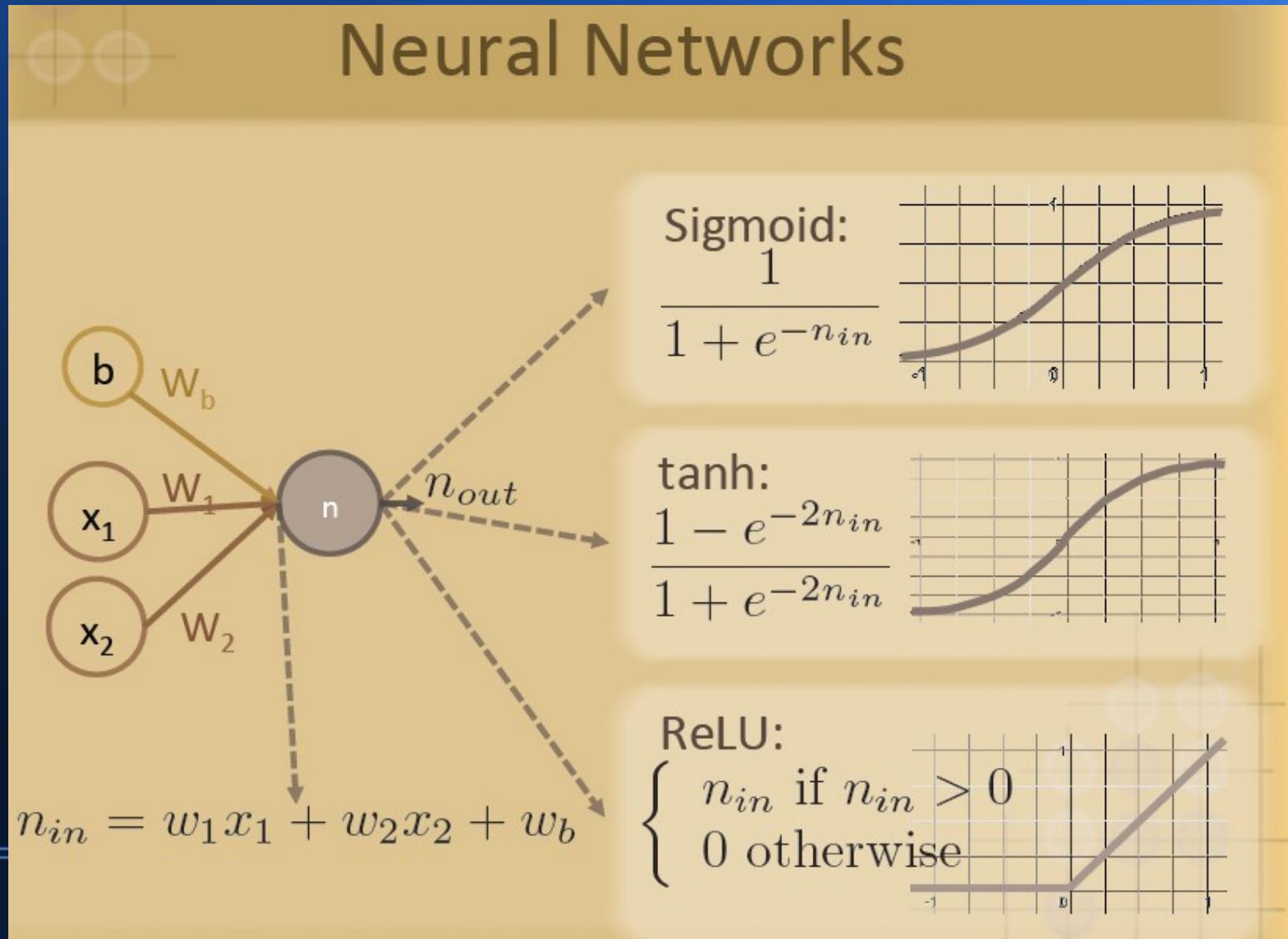
還有《節點之間的連線》
（神經突觸）



單層的神經網路，架構如下圖

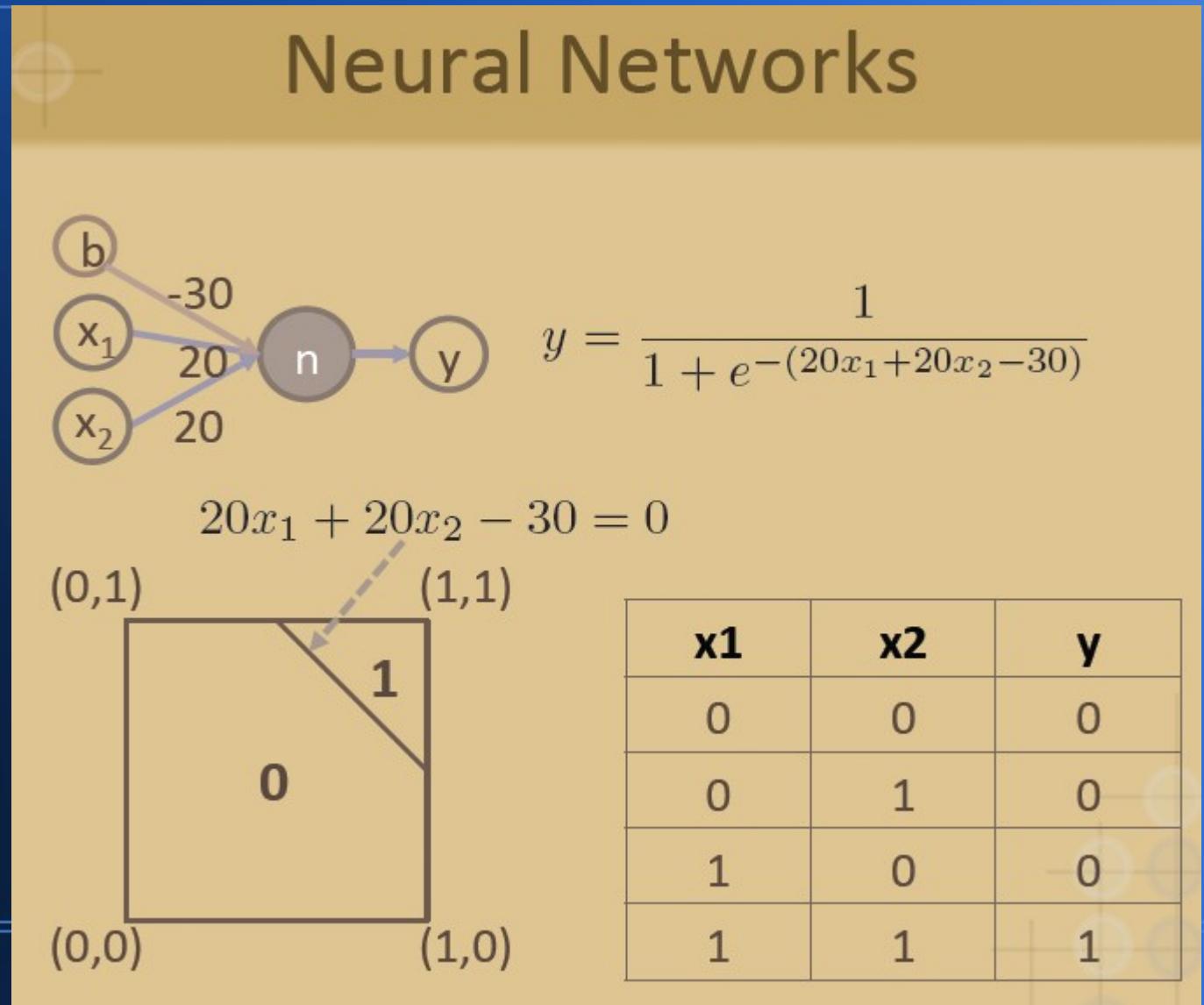


輸出節點彙整輸入後，會用 sigmoid, tanh, ReLU 等函數轉換為接近 $(0, 1)$ 的輸出值



這種網路可以對輸入進行區分，
以便輸出正確的解答

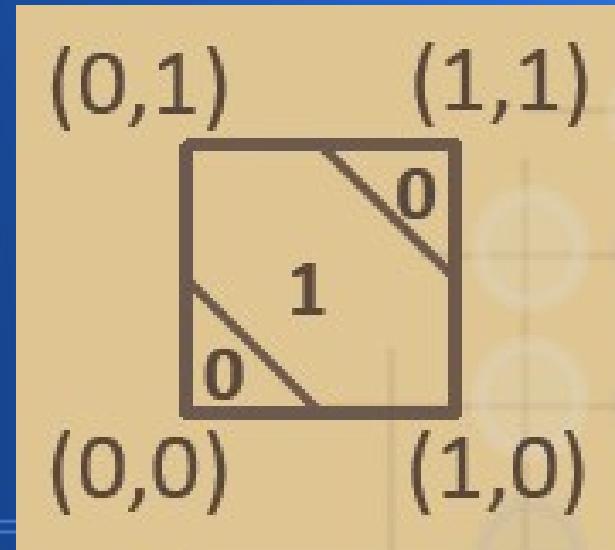
- 右圖為 AND 閘
的《真值表》
與對應的
《神經網路》



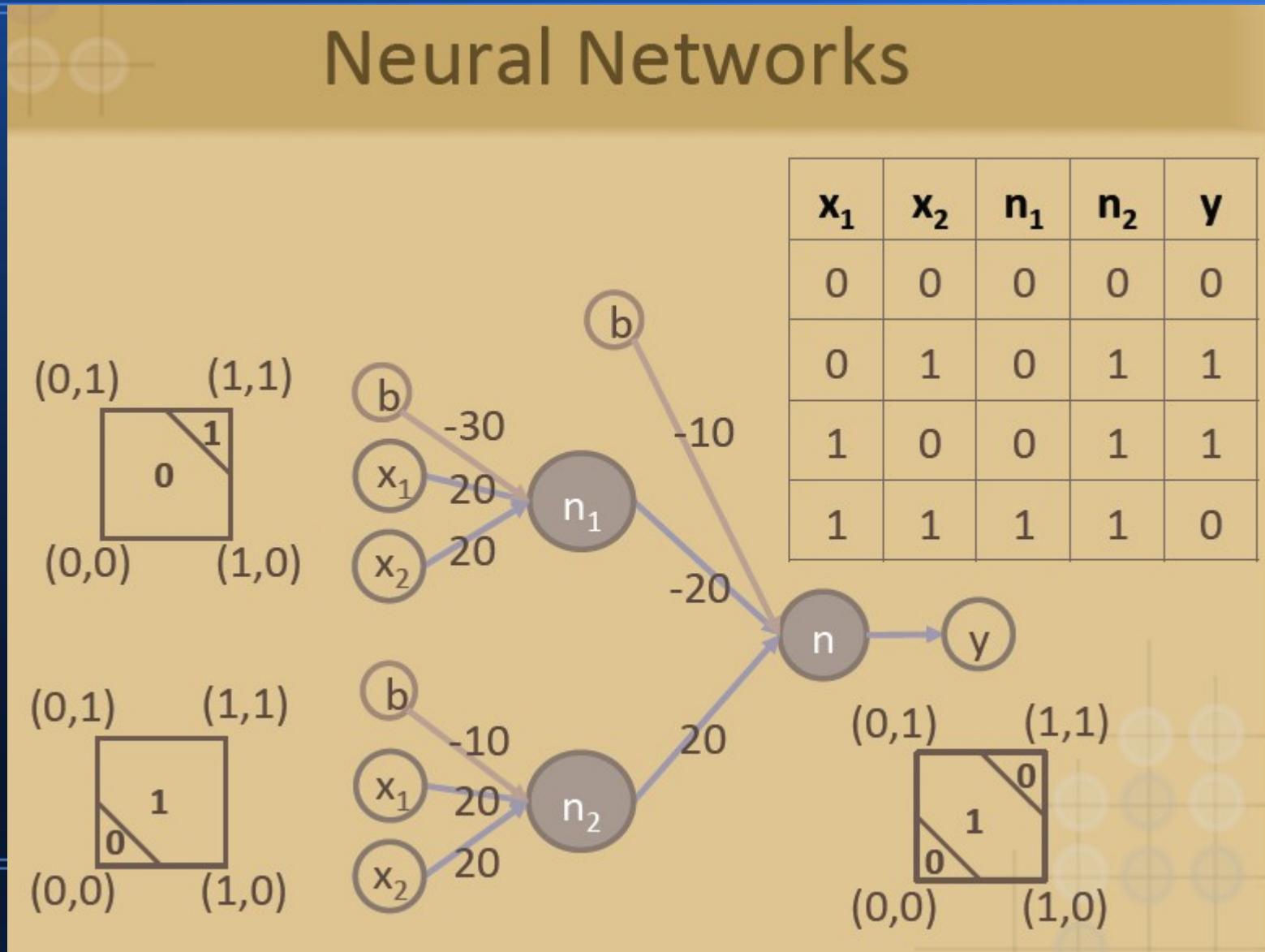
但是這種單層網路

- 對於稍微複雜一點的問題
理論上根本就無法處理。
(像是右圖所代表的 XOR 閘，
就沒辦法用單層神經網路解決)
- 因為它只能對整個平面劃一刀
所以沒有辦法用一刀把右圖
的 0 與 1 兩個區域切開來。

x_1	x_2	n_1	n_2	y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



這時候就需要用多層的神經網路
才能解決較複雜的問題（像是 XOR）



神經網路常被用來

- 解決影像辨識的問題
- 像是手寫數字辨識，就是一個神經網路的經典問題。
- 我們可以在輸出層安排 10 個節點，分別對應到 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 的數字的辨識強度。
- 然後將強度最強者視為辨識結果！

這種問題稱為多類別分類問題

- 我們可以在輸出節點用 SoftMax 函數來進行強度轉換

Multi-Class Classification

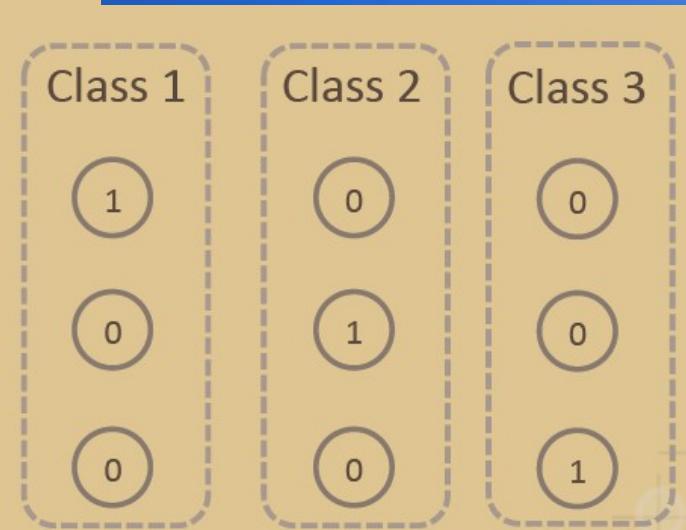
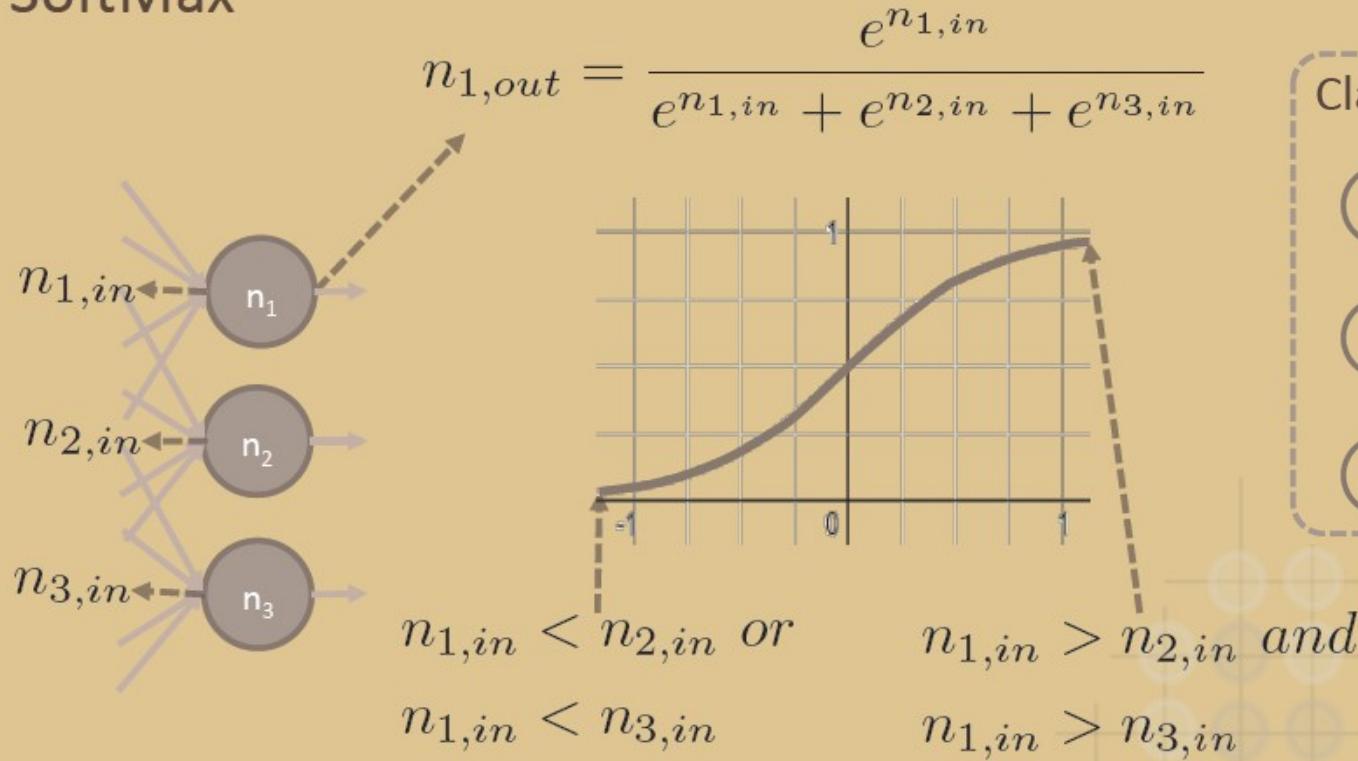
- SoftMax

$$n_{1,in} \xrightarrow{n_1} n_{1,out} = \frac{e^{n_{1,in}}}{e^{n_{1,in}} + e^{n_{2,in}} + e^{n_{3,in}}}$$
$$n_{2,in} \xrightarrow{n_2} n_{2,out} = \frac{e^{n_{2,in}}}{e^{n_{1,in}} + e^{n_{2,in}} + e^{n_{3,in}}}$$
$$n_{3,in} \xrightarrow{n_3} n_{3,out} = \frac{e^{n_{3,in}}}{e^{n_{1,in}} + e^{n_{2,in}} + e^{n_{3,in}}}$$

因為 SoftMax 函數的輸出通常
只會有一個非常接近 1，其餘都幾乎都是 0

Multi-Class Classification

- SoftMax



- 這樣就可以進行《分類》而不會有同時屬於很多類的問題了。

這樣，我們大致已經講解完

- 捲積神經網路的原理了

但是有個問題

這種方法怎麼用在圍棋上呢？

這就牽涉到另一個方法

- 那個方法就是

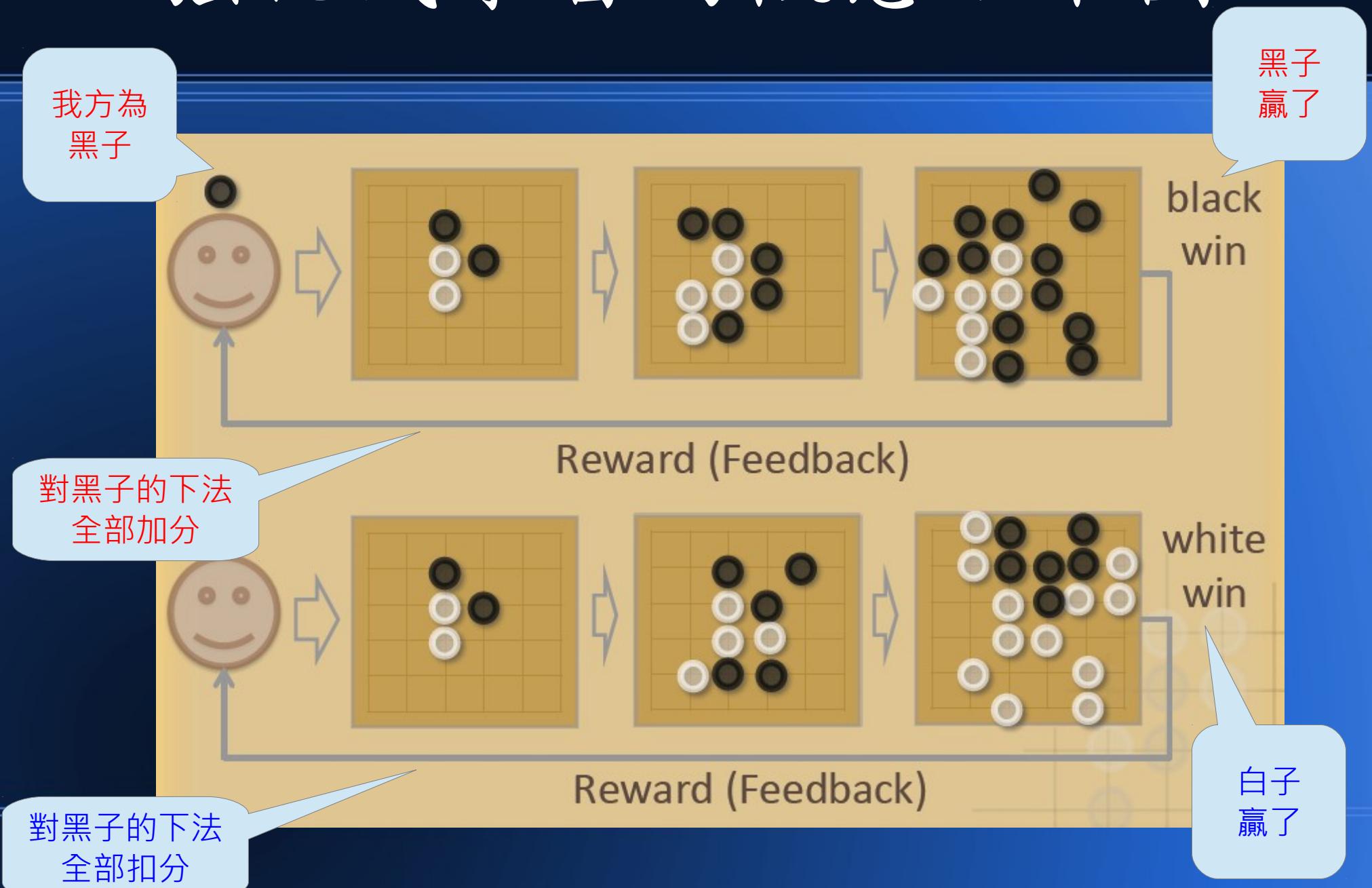
強化式學習

所謂的強化式學習

- 就是對好的結果加分
- 對不好的結果扣分

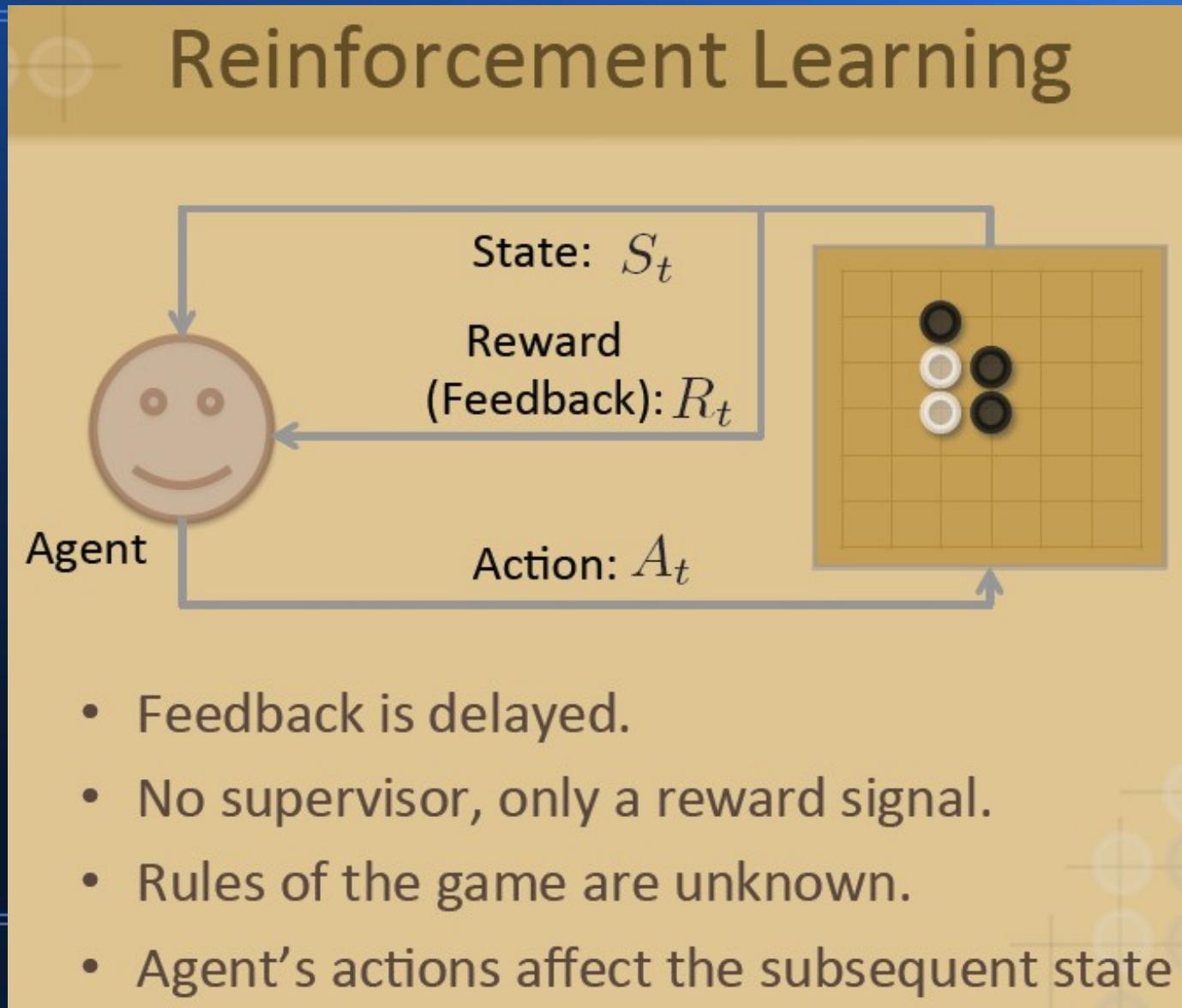
的一種學習方法

強化式學習的概念如下圖



於是、只要依據結果的好壞

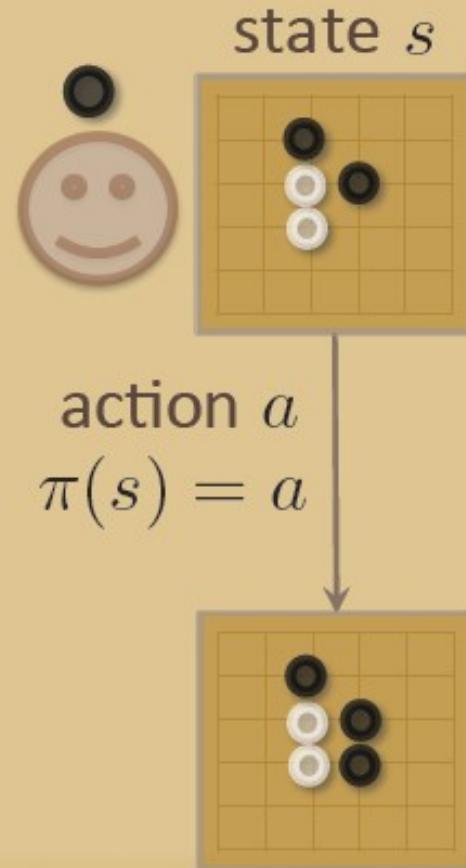
- 就能不斷修正並強化這個程式



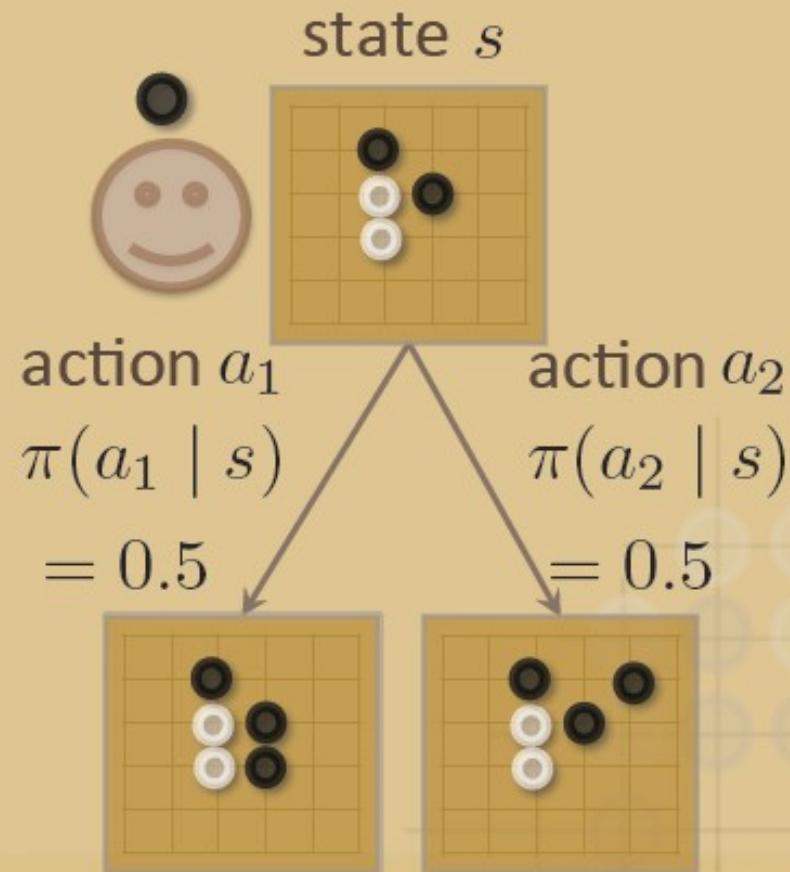
在 AlphaGo 當中，這種方法會
用來調整策略網路的機率

The behavior of an agent

Deterministic Policy



Stochastic Policy



其調整方法是《策略梯度下降法》

Policy Gradient Method

- REINFOCE
 - the REward Increment = Nonnegative Factor Offset ReinforCEment

$$w \leftarrow w + \alpha(r - b) \frac{\partial \pi(a|s)}{\partial w}$$

weights in learning rate reward baseline
policy function rate (usually = 0)

```
graph TD; A[w ← w + α(r - b) ∂π(a|s) / ∂w] --> B[weights in policy function]; A --> C[learning rate]; A --> D[reward]; A --> E[baseline<br/>(usually = 0)]
```

但是、圍棋比較複雜

- 解說起來並不容易
- 讓我們先用一個簡單的機器人走路問題為例，說明這個強化學習的調整原理。

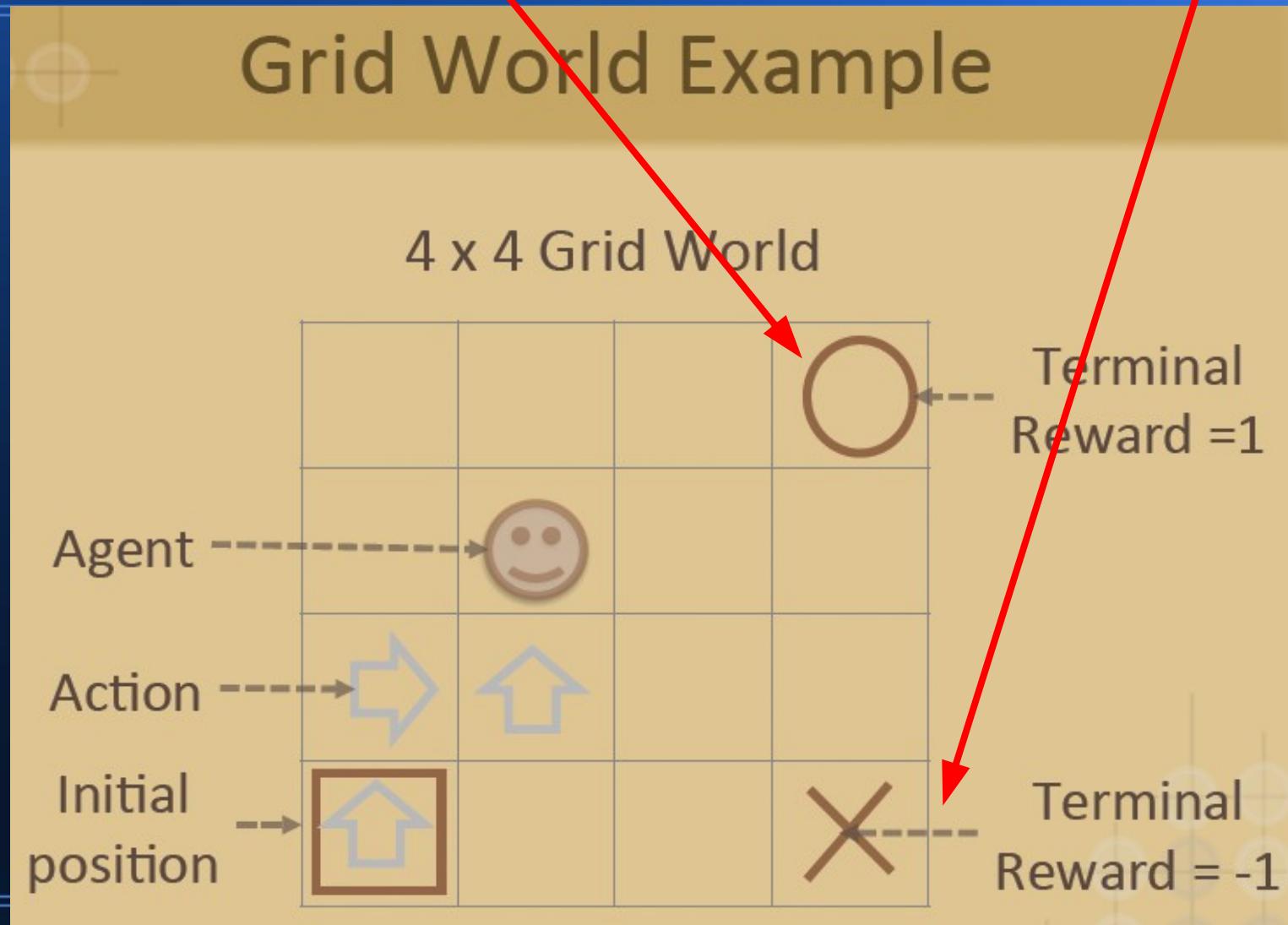
在一個方格世界中 有個機器人

Grid World Example

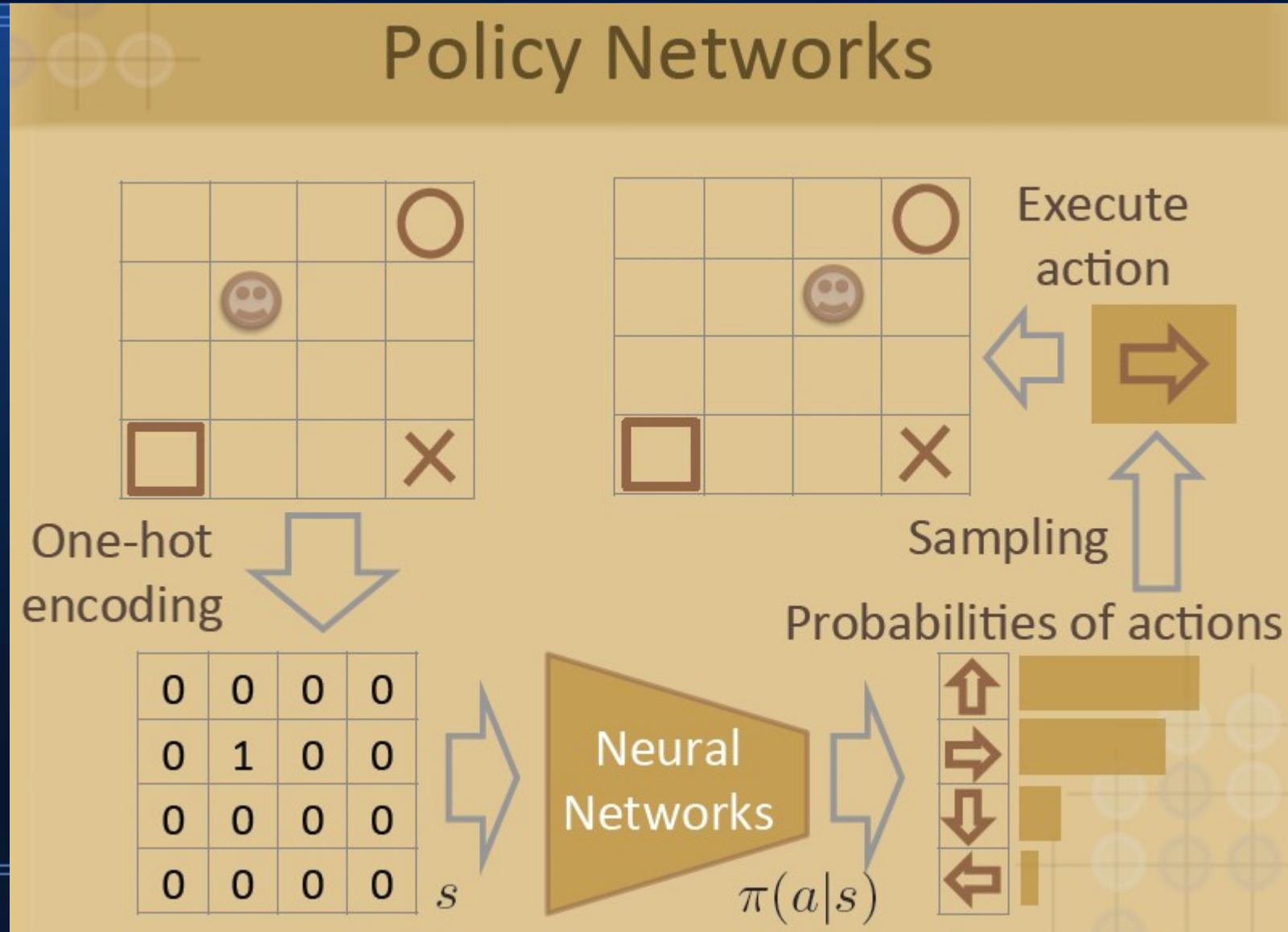
4 x 4 Grid World



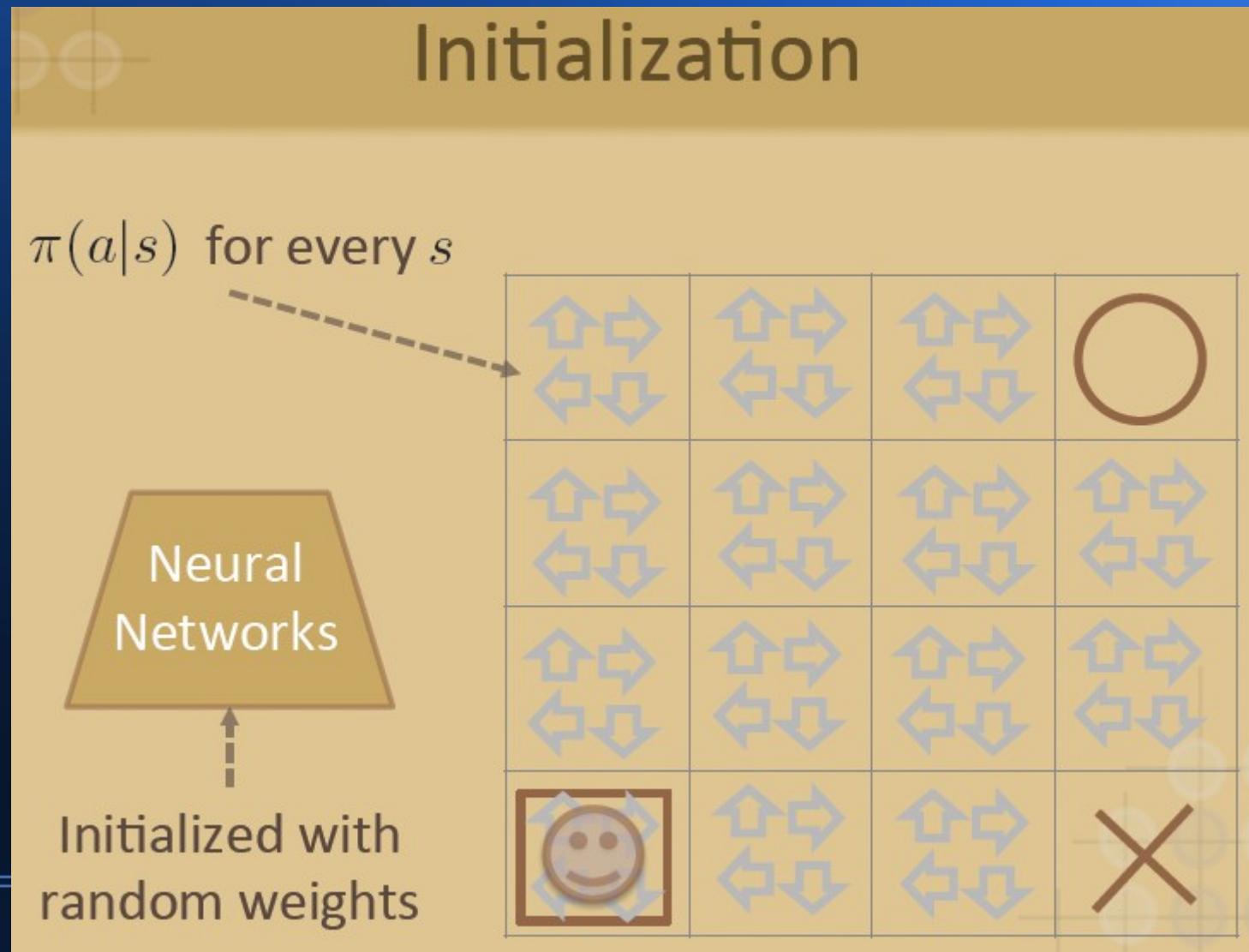
如果他
走到○就得一分，走到X就扣一分



然後我們要訓練它
往上下左右走的機率

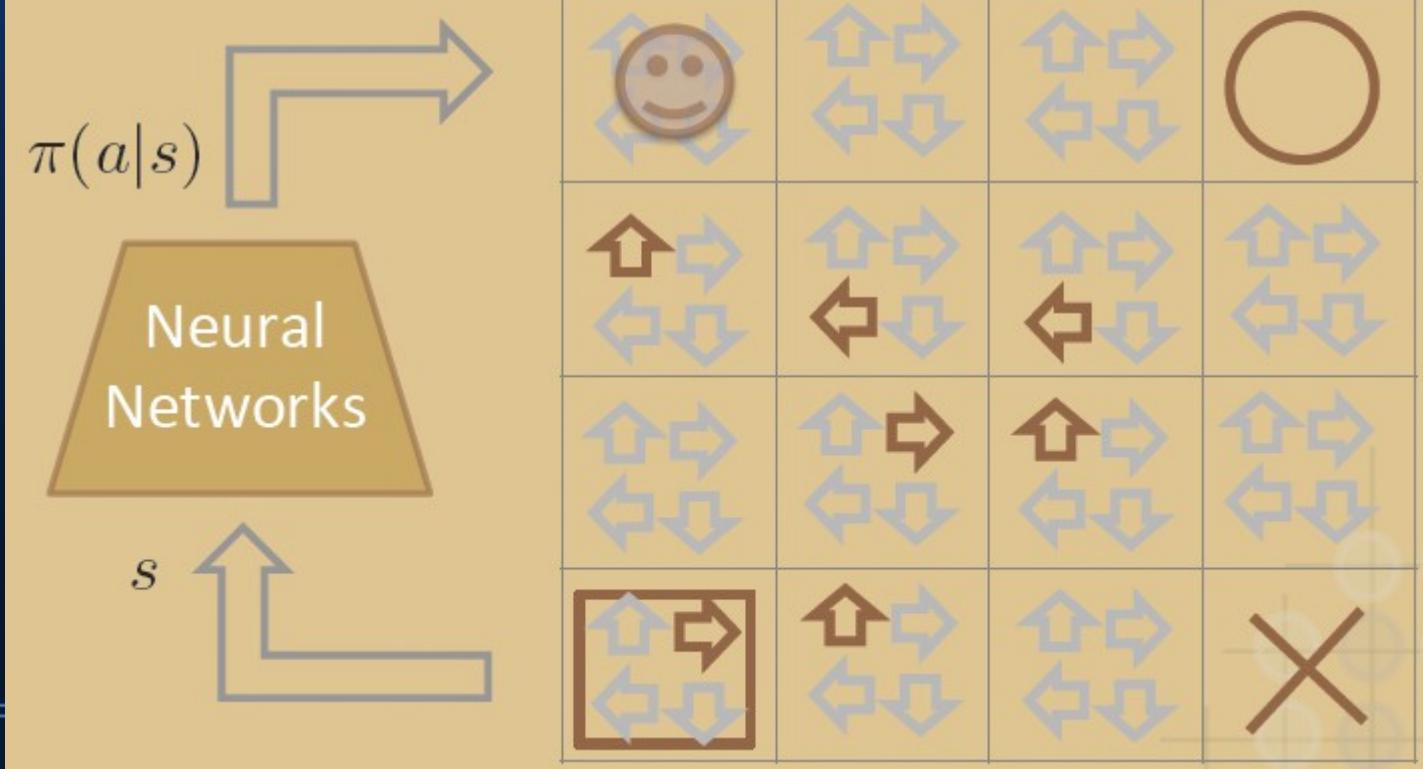


一開始《上下左右》 的機率各為 $1/4$

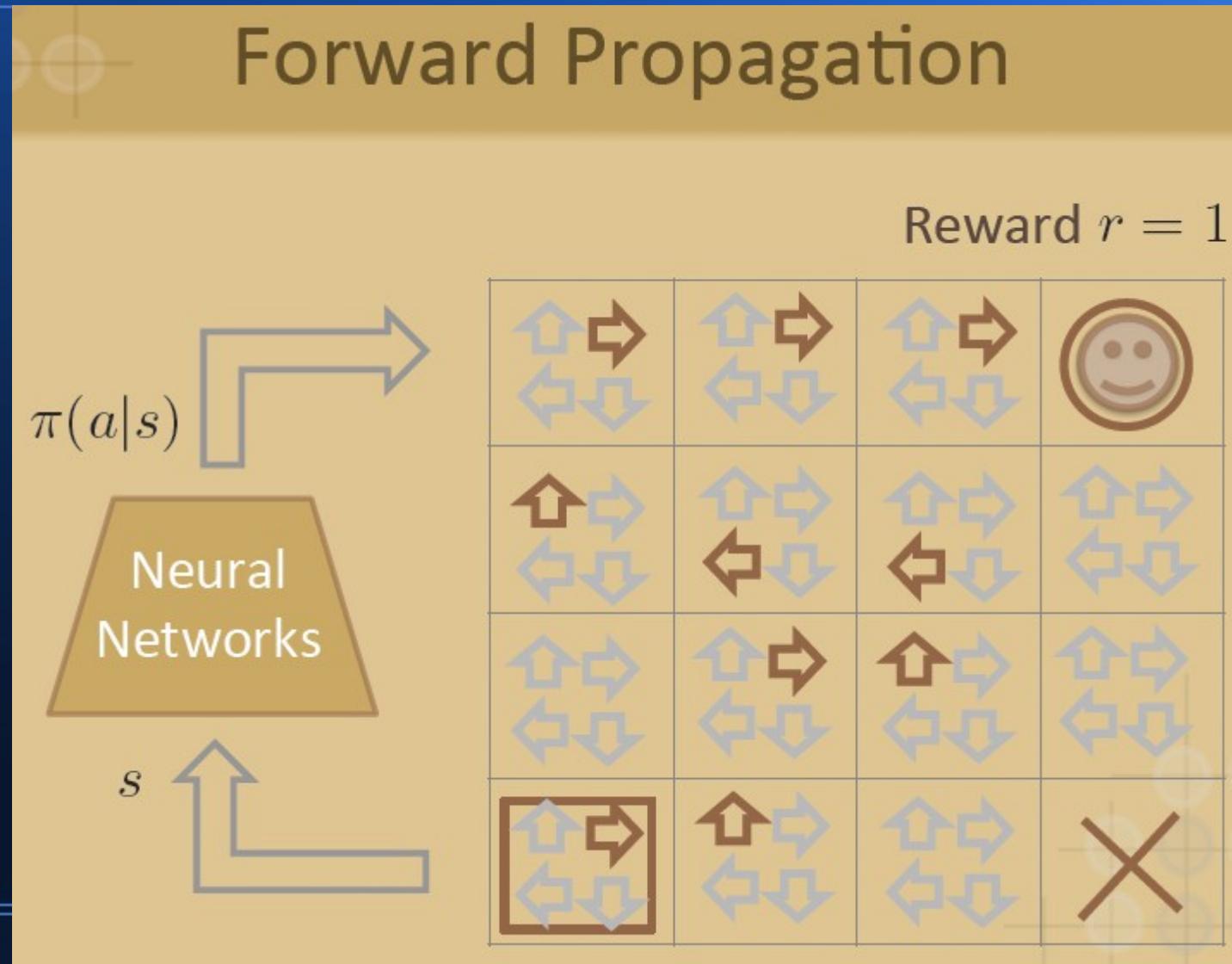


於是它開始亂走

Forward Propagation



如果走到○，就得了一分



於是路上的
所有《決策方向》都會被加分

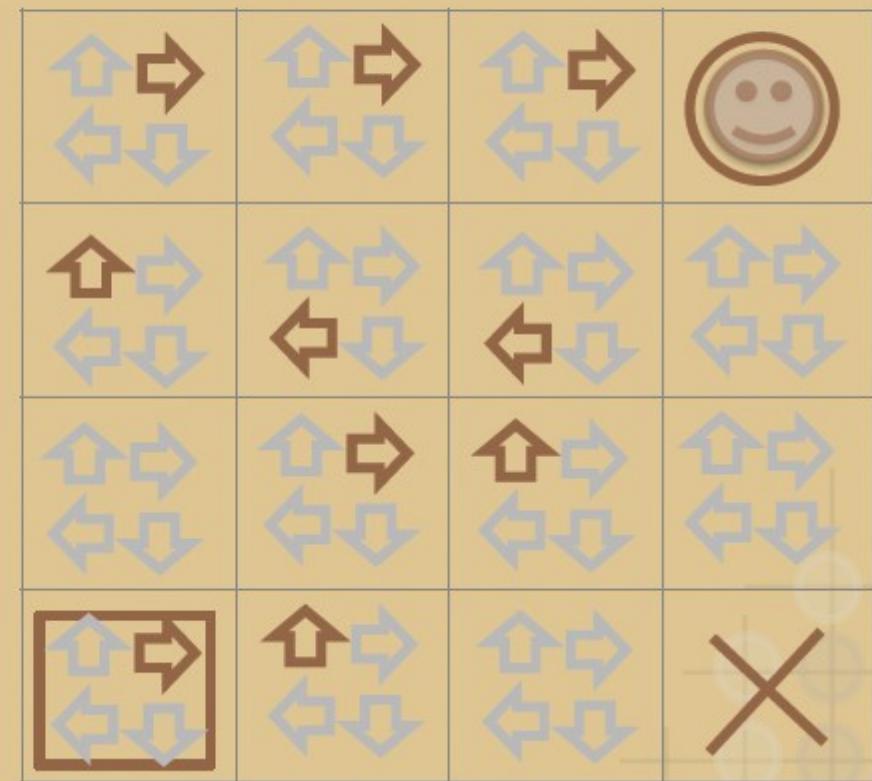
Backward Propagation

Reward $r = 1$

$$r \frac{\partial \pi(a|s)}{\partial w}$$

↓


$$w \leftarrow w + (r - b) \frac{\partial \pi(a|s)}{\partial w}$$



結果箭頭就會長大

Backward Propagation

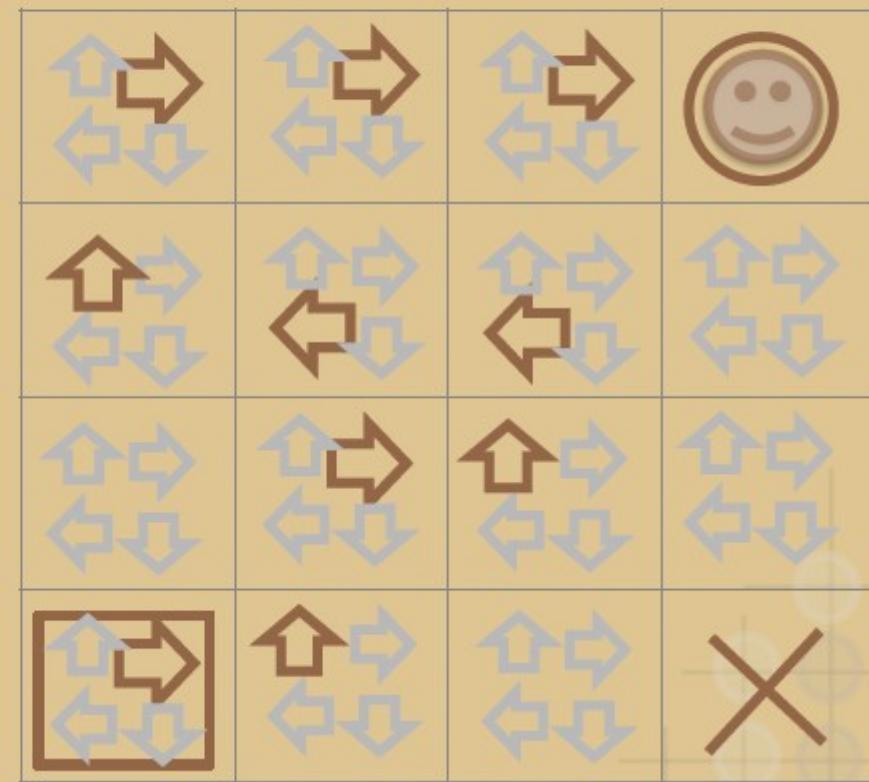
$$r \frac{\partial \pi(a|s)}{\partial w}$$

↓

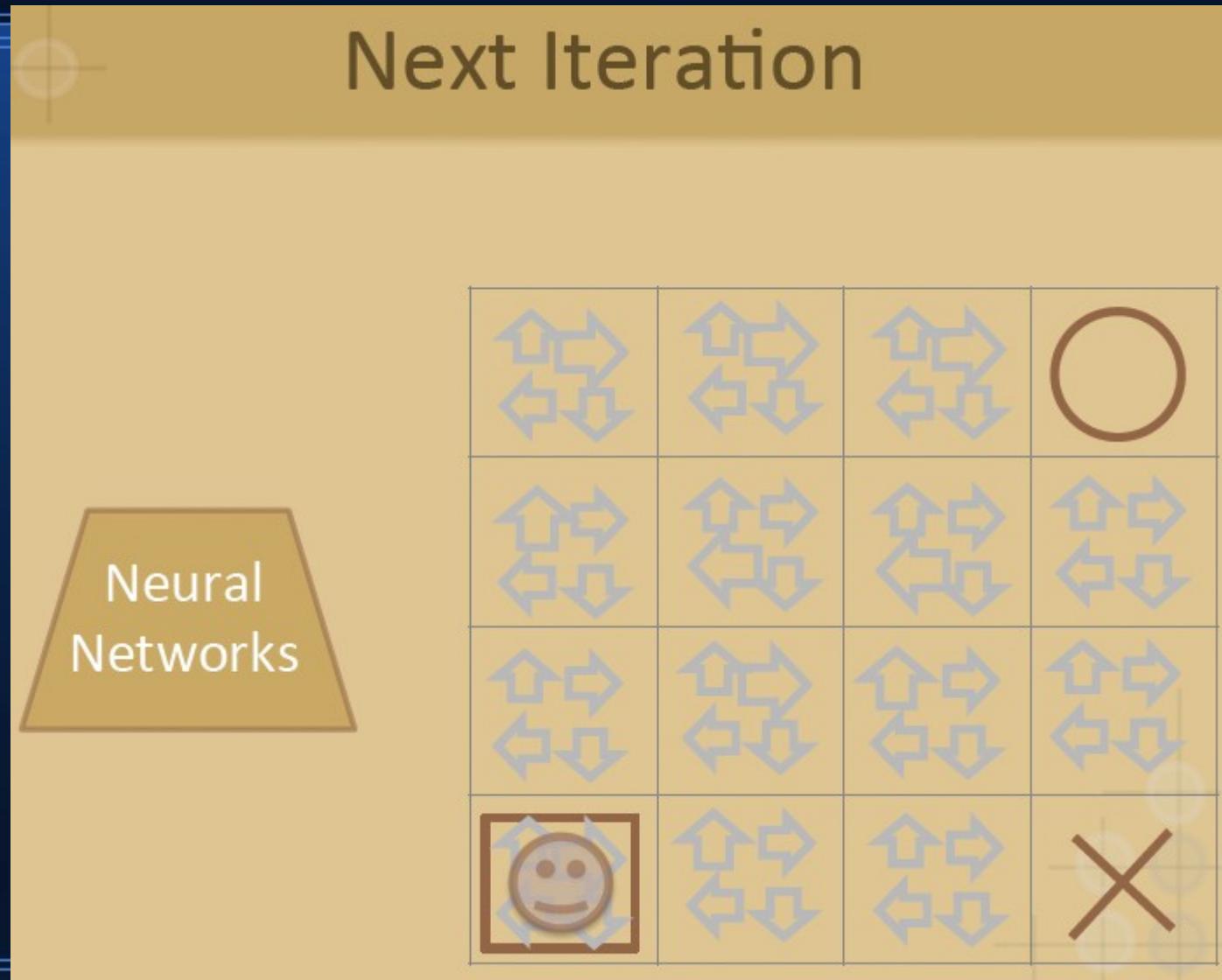
Neural Networks

$$w \leftarrow w + (r - b) \frac{\partial \pi(a|s)}{\partial w}$$

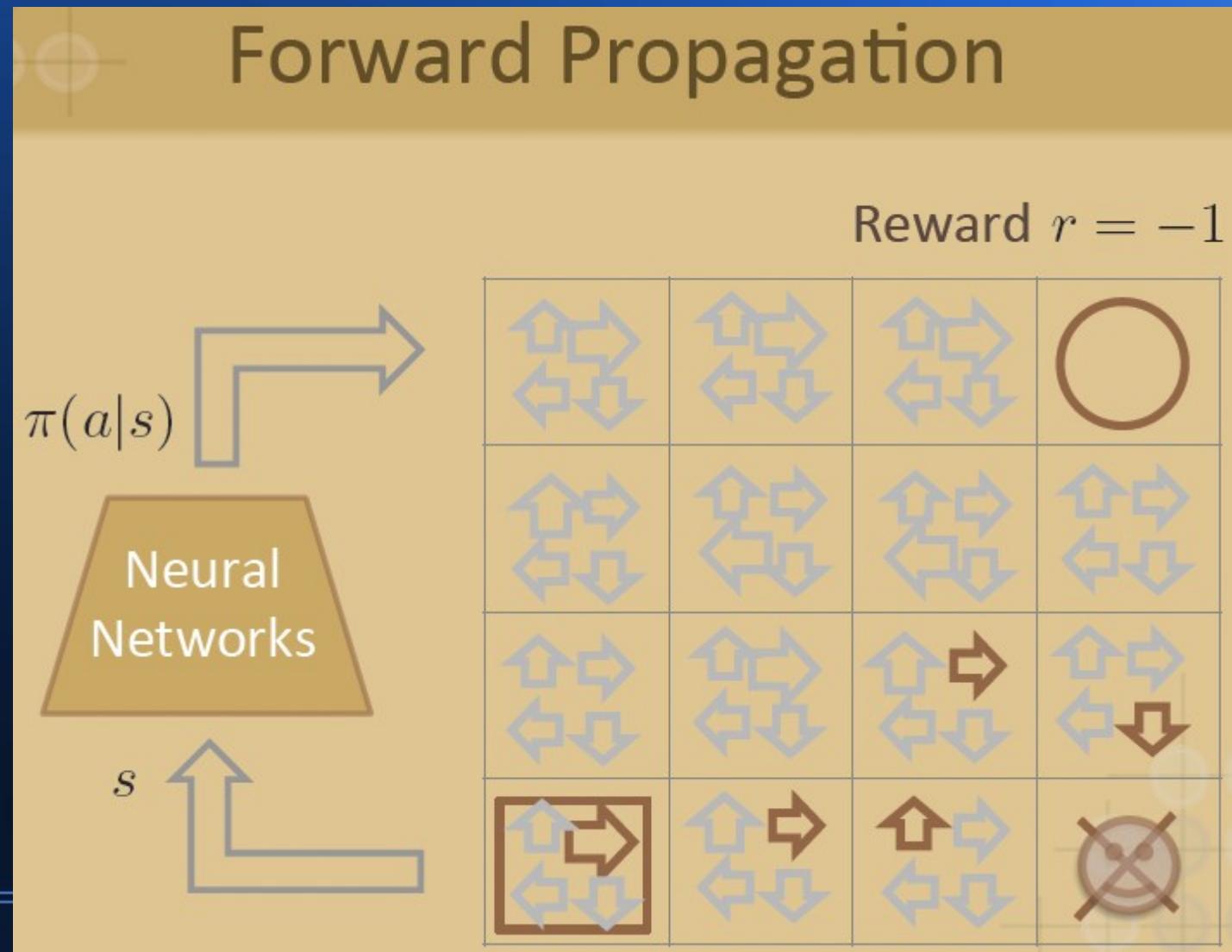
Reward $r = 1$



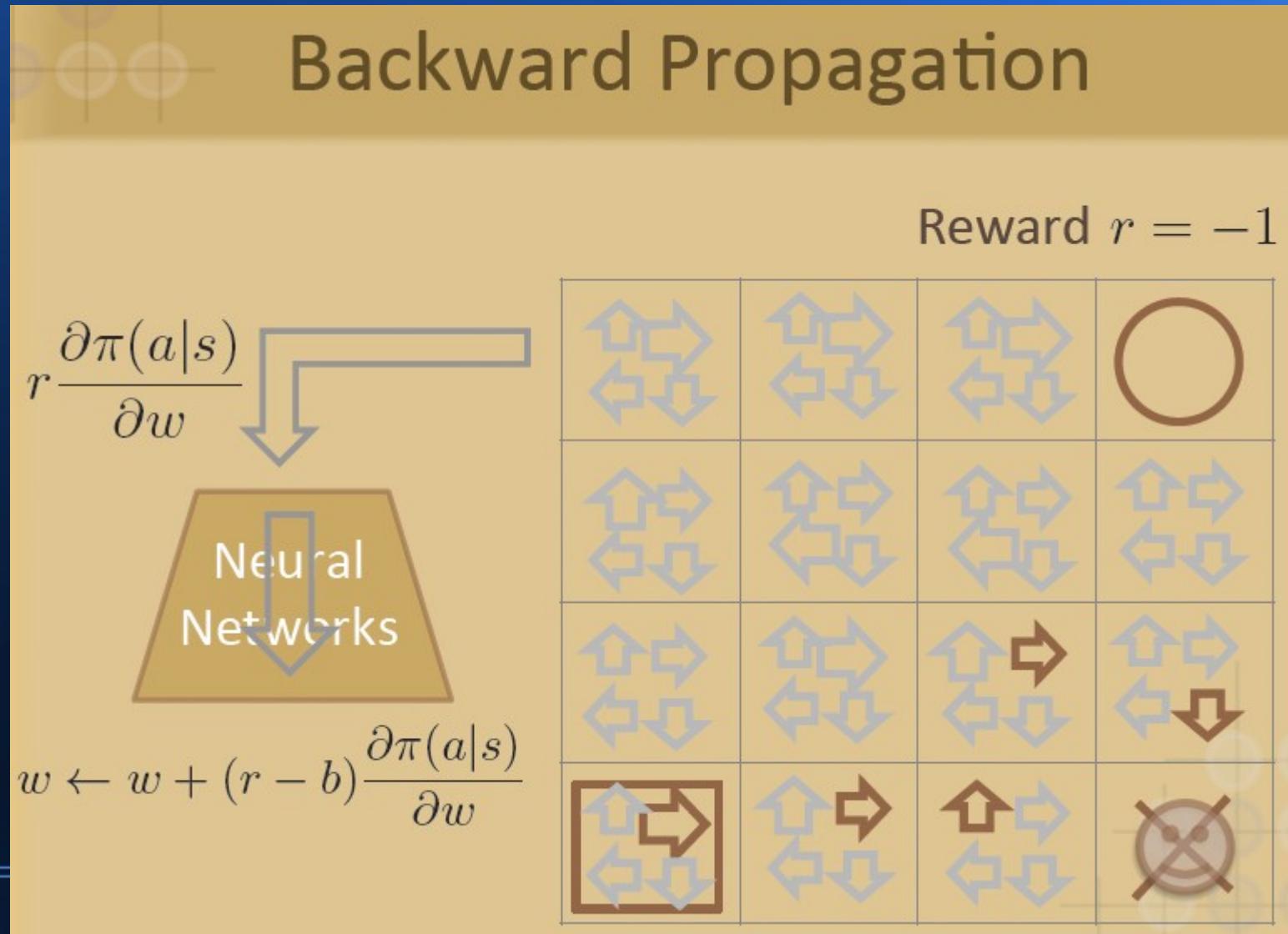
接著繼續下一輪的亂走



假如不小心走到 X，那就會扣一分



然後一樣調整所有路上的決策
進行扣分的動作

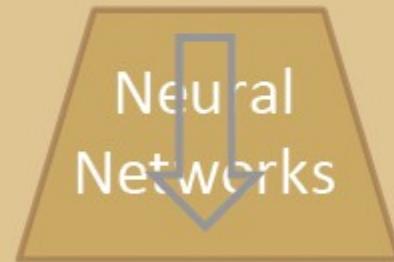


於是箭頭就變小了
(權重調低)

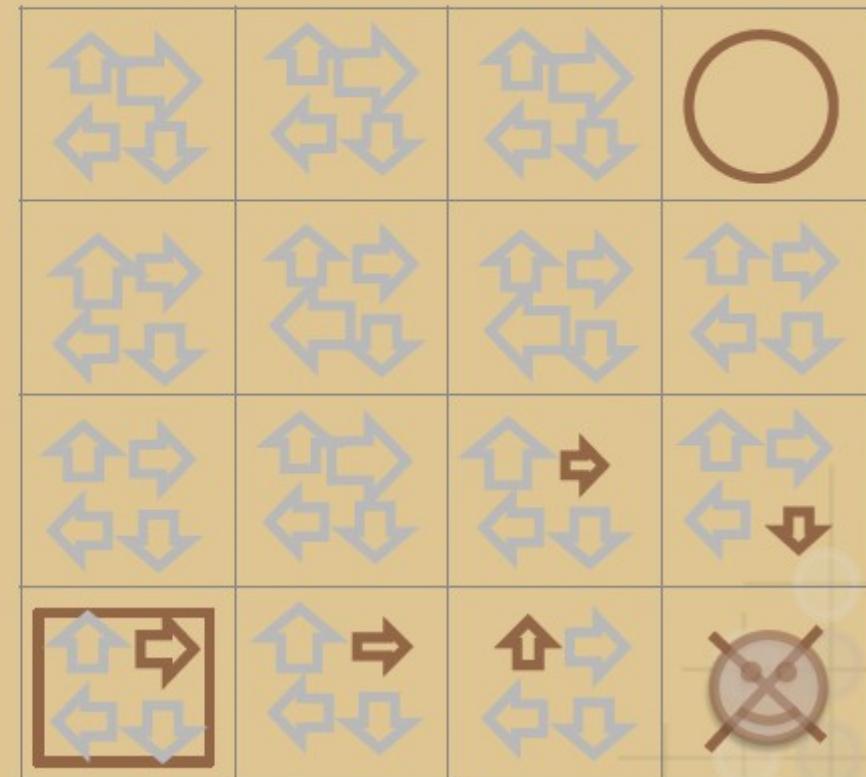
Backward Propagation

Reward $r = -1$

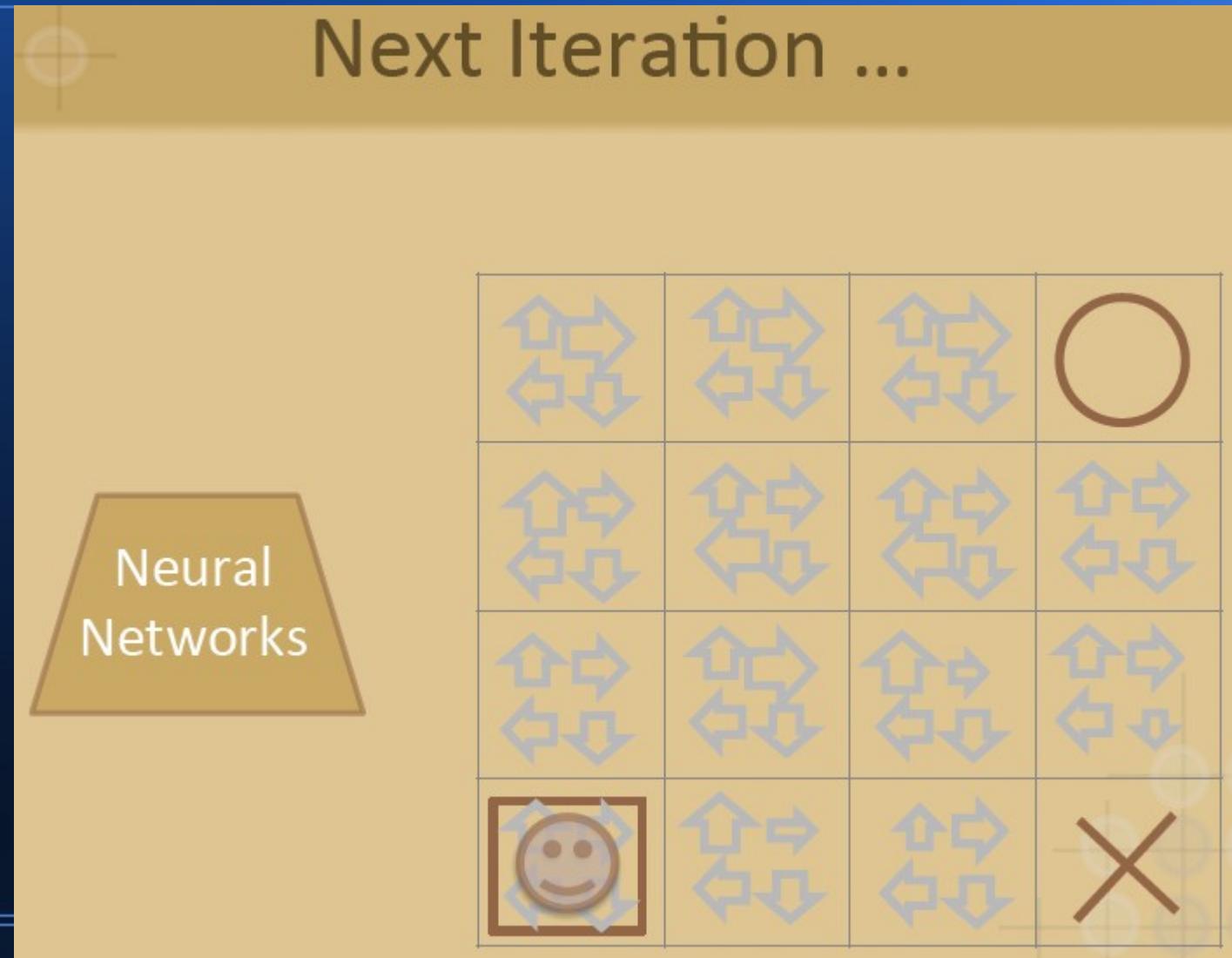
$$r \frac{\partial \pi(a|s)}{\partial w}$$



$$w \leftarrow w + (r - b) \frac{\partial \pi(a|s)}{\partial w}$$



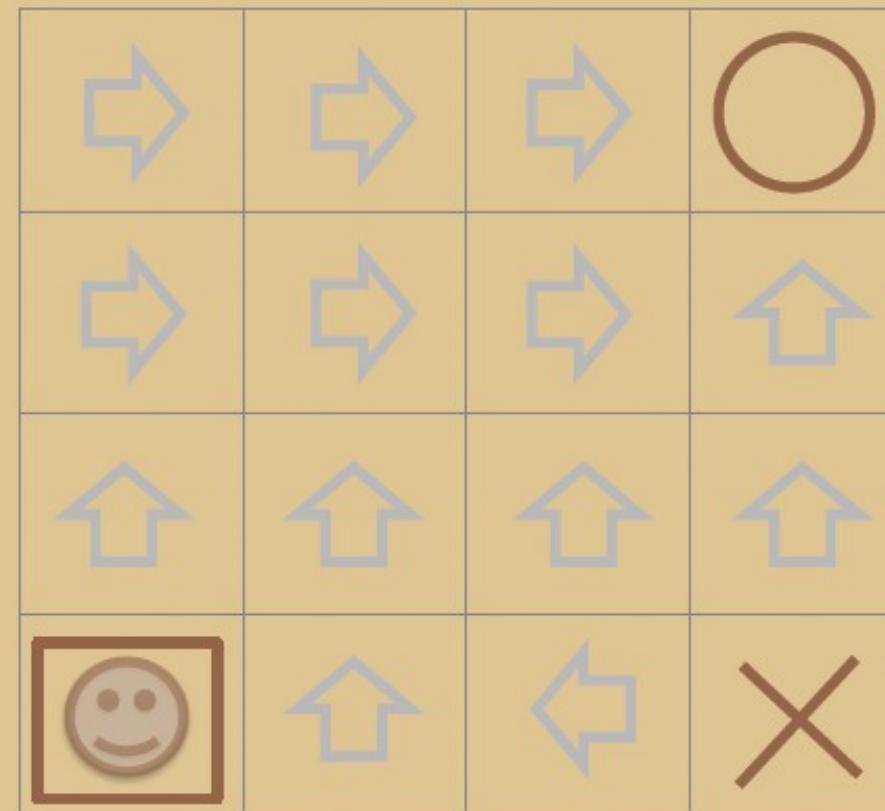
接著就再進行下一輪亂走



很多輪之後 . . .

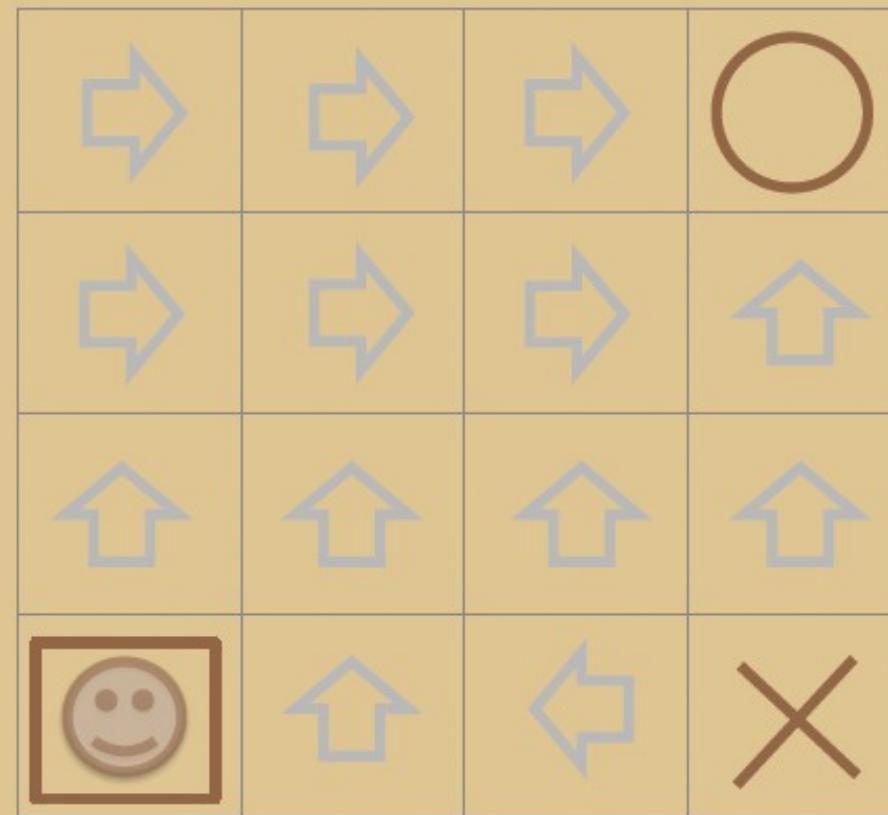
權重都調得很好了
機器人就學到了很好的走法

After Several Iterations ...

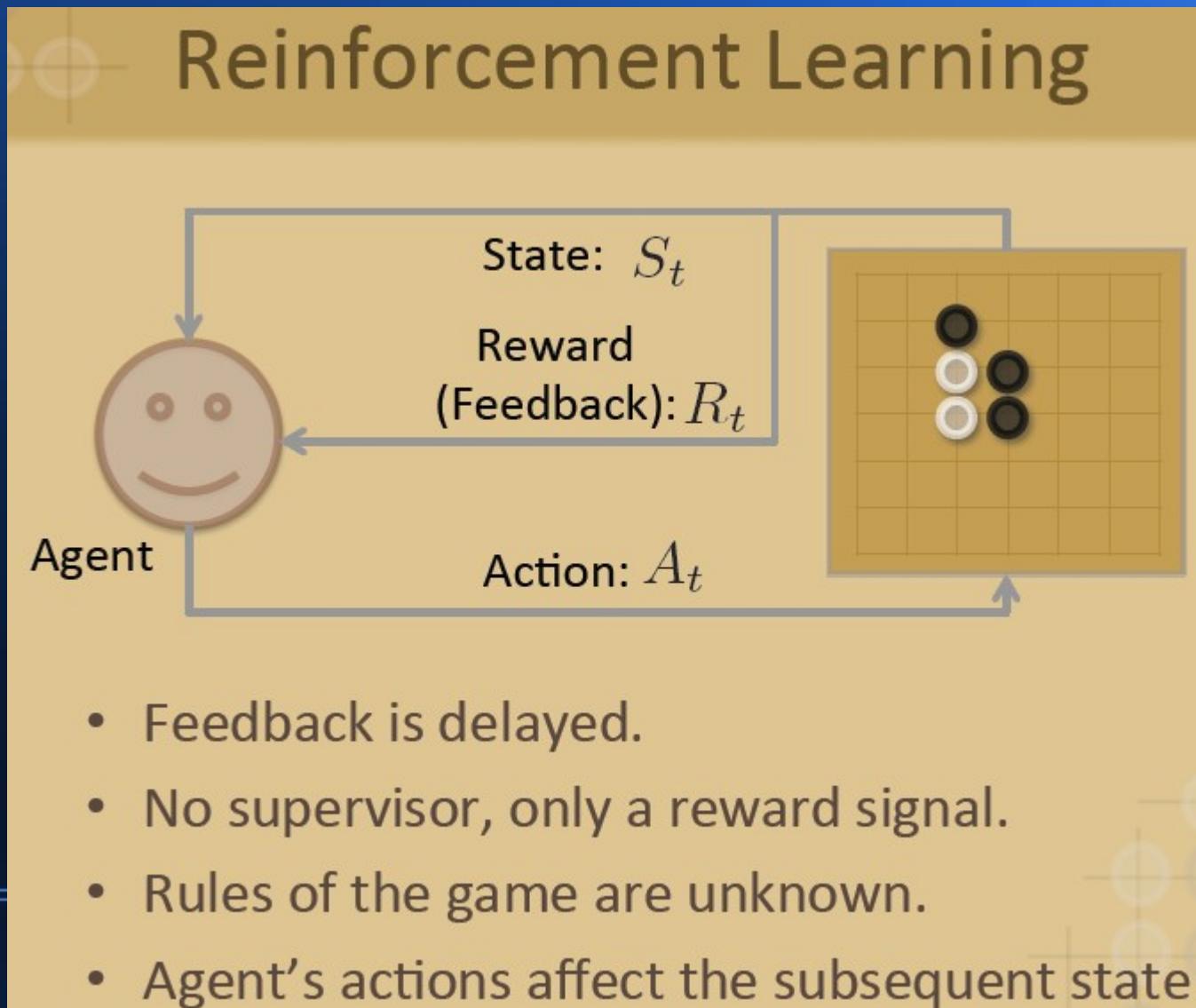


於是每次都會走到○
不會再走到 X 被扣分了

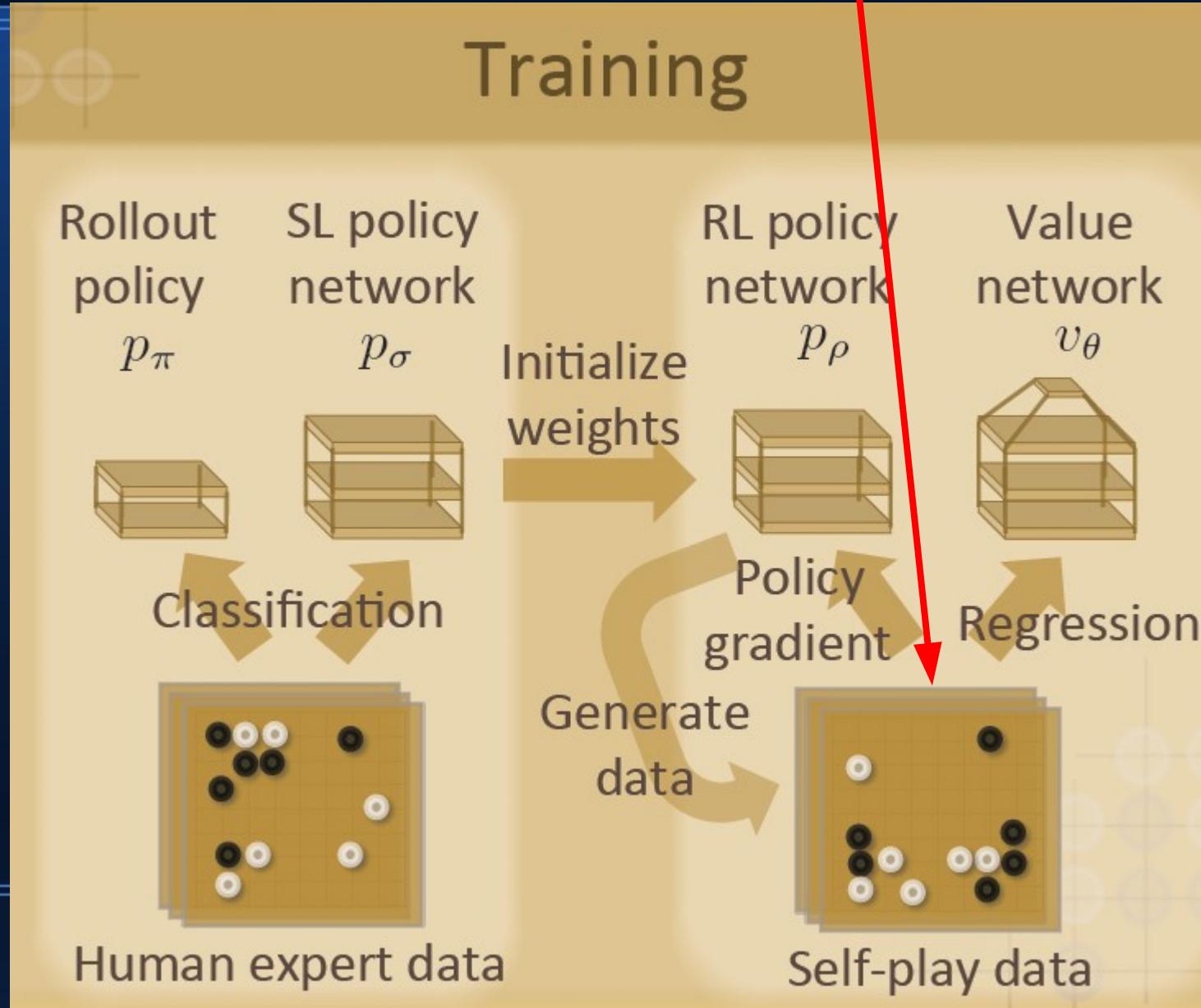
After Several Iterations ...



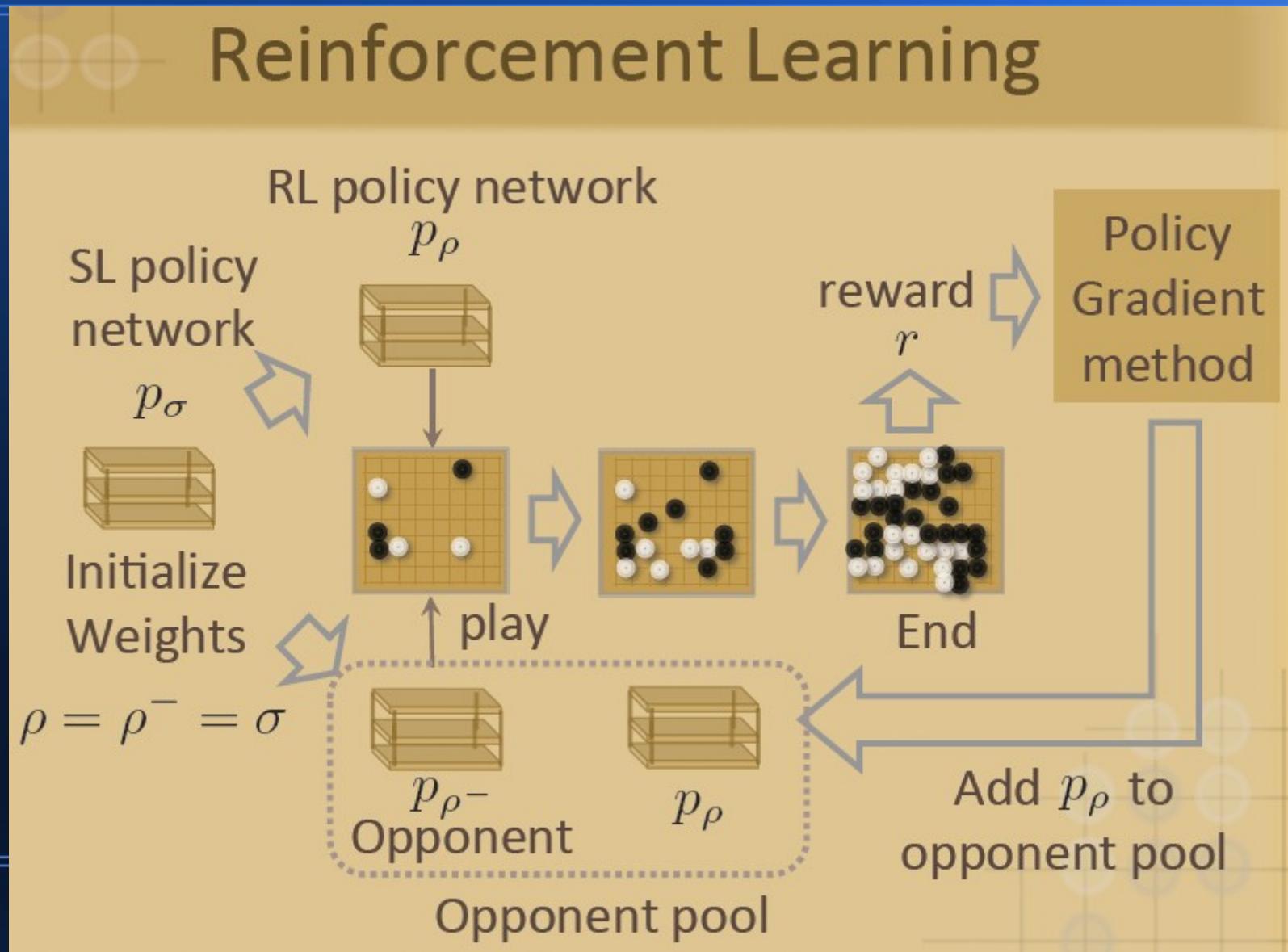
以上這種方法
就是所謂的強化式學習！



AlphaGo 在《自我對下》時 就採用了這種強化式學習

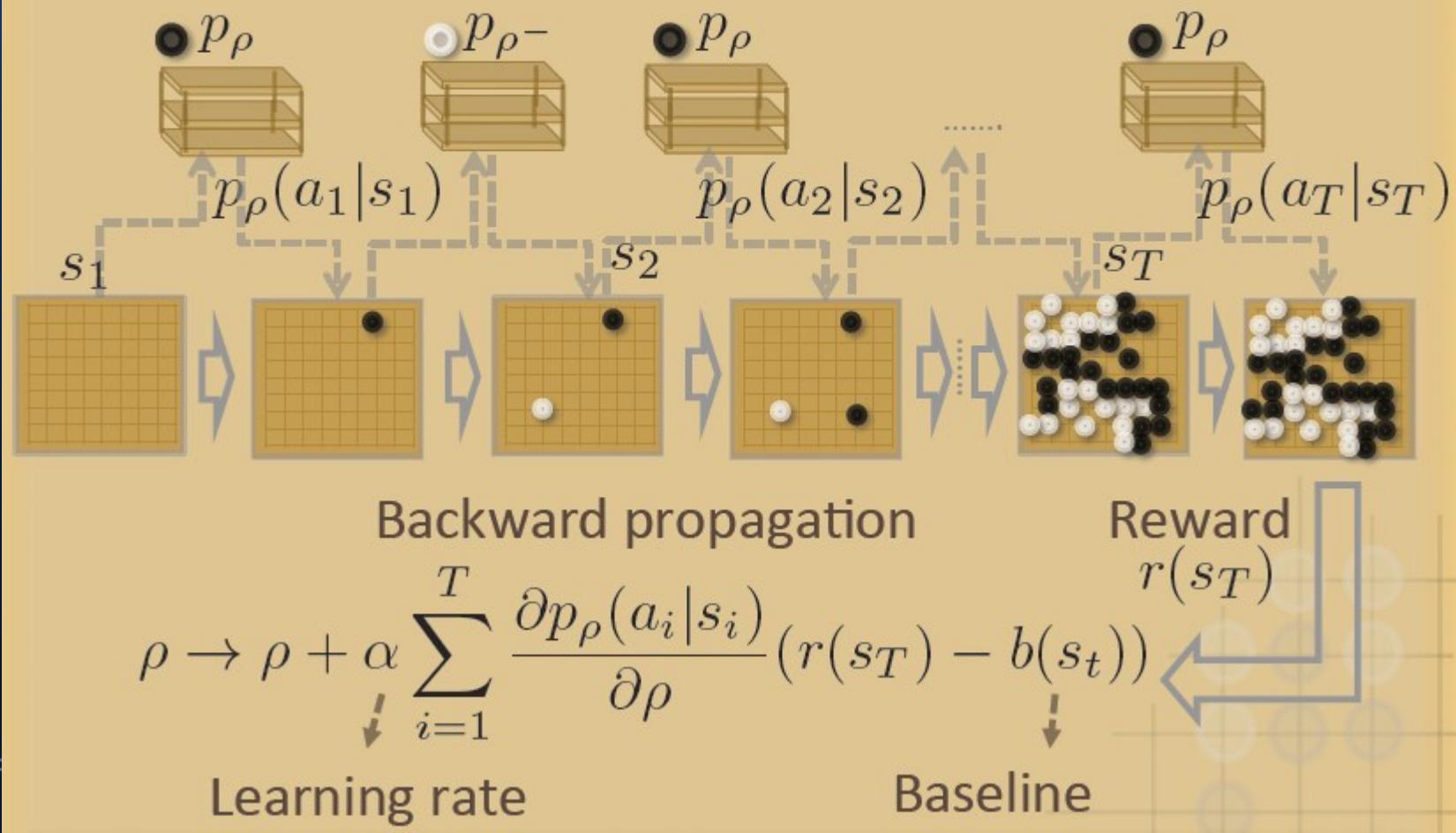


其《自我對下》的 強化學習模型如下圖

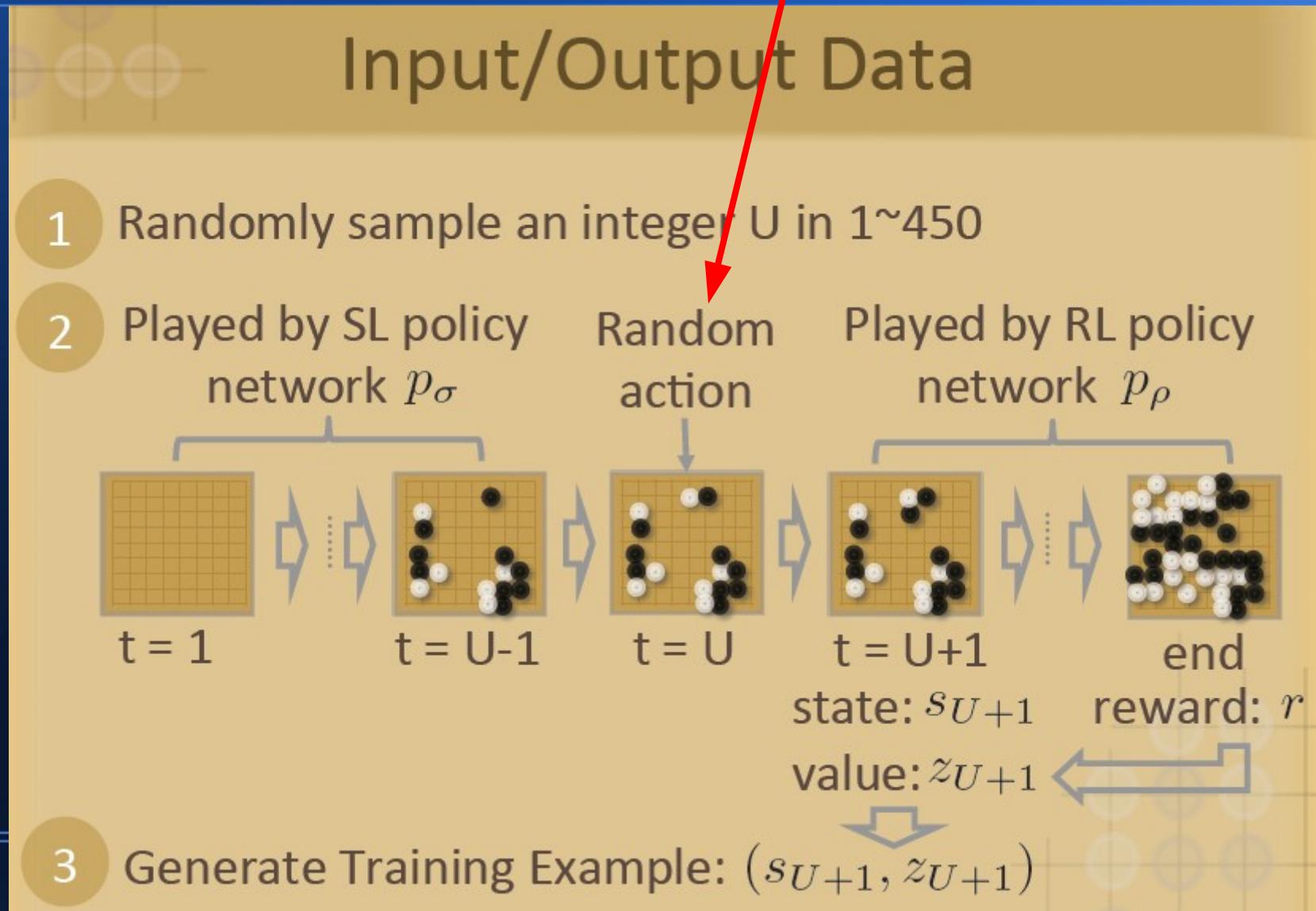


並採用如下的《梯度調整公式》

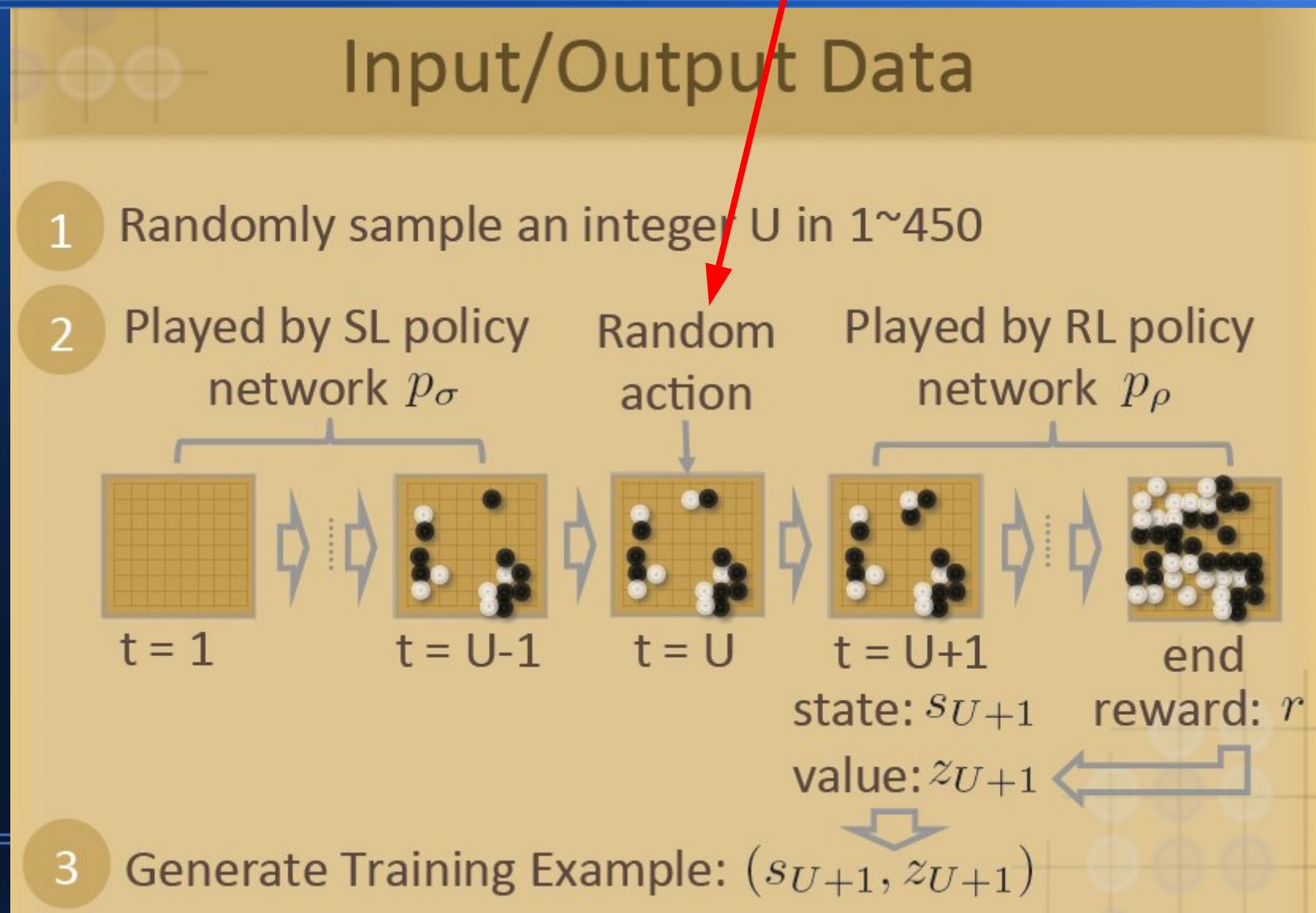
Policy Gradient Method



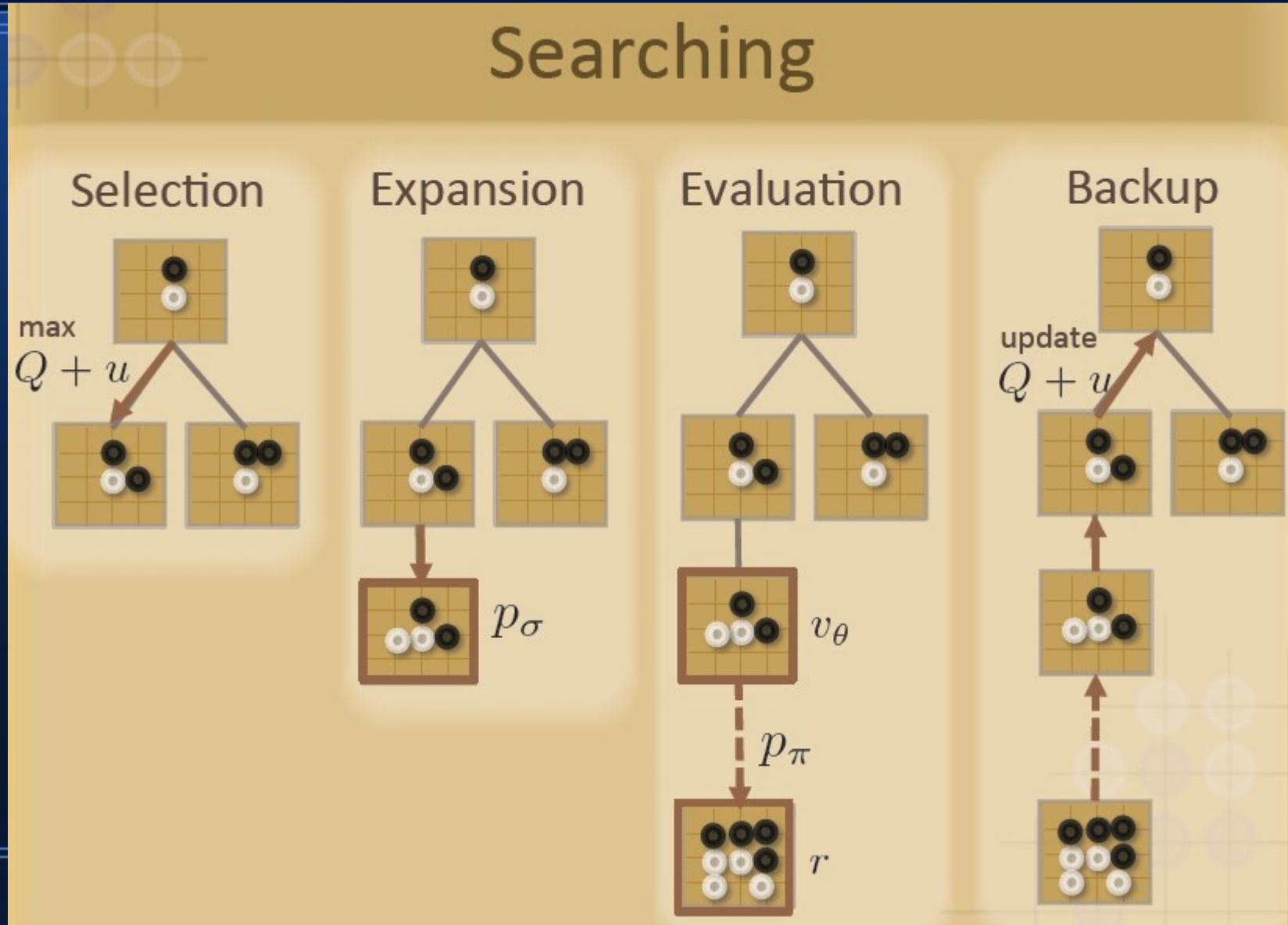
而且為了學習沒見過的棋局
在自我對下的過程中，加入了一步《亂走》的情況



這樣可以擴大探索空間，增強 AlphaGo 的學習範圍



在《自我對下》的過程中 採用了蒙地卡羅樹狀搜尋法



搜尋時必須記錄下列資訊

Searching

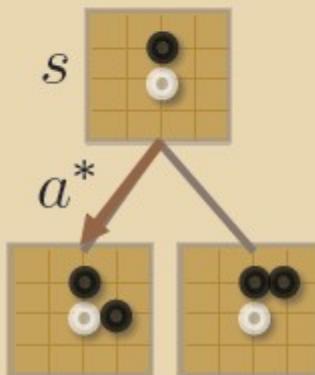
- Each edge stores a set of statistics



- $Q(s, a)$: combined mean action value
- $P(s, a)$: prior probability evaluated by $p_\sigma(a|s)$
- $W_r(s, a)$: estimated action value by $p_\pi(a|s)$
- $W_v(s, a)$: estimated action value by $v_\theta(s)$
- $N_r(s, a)$: counts of evaluations by $p_\pi(a|s)$
- $N_v(s, a)$: counts of evaluations by $v_\theta(s)$

每次都取《信賴區間上界最大的路徑》
做為下一次搜尋的節點

Selection



Choose action

$$a^* = \operatorname{argmax}_a(Q(s, a) + u(s, a))$$

Exploitation

Exploration

PUCT Algorithm:

$$u(s, a) = cP(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)}$$

Level of exploration

Visit counts
of parent node s
Visit counts
of edge (s, a)

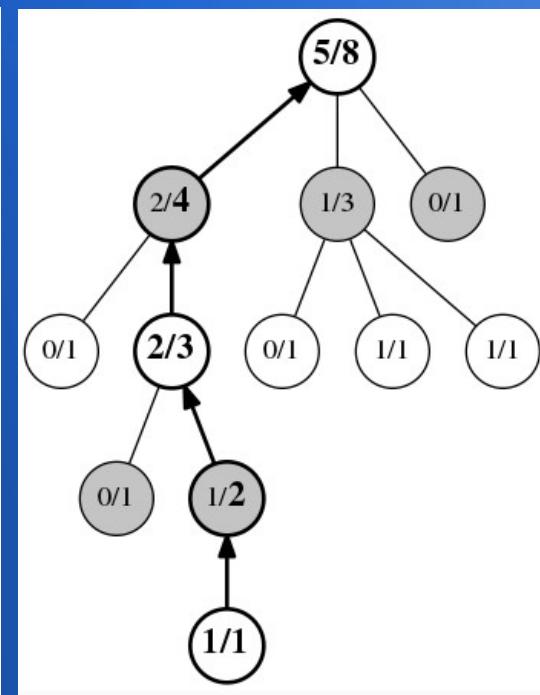
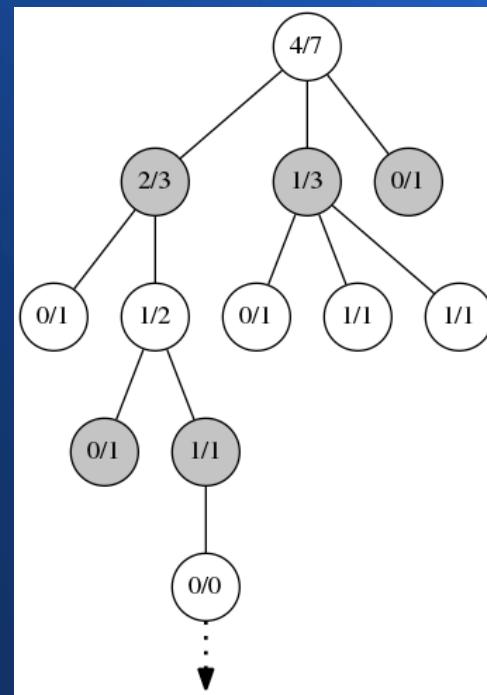
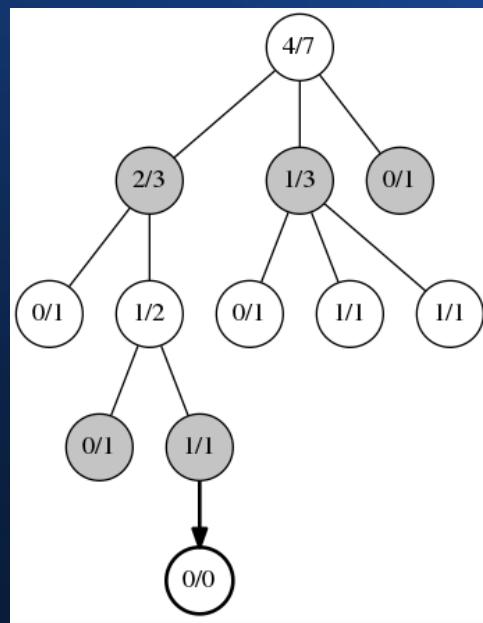
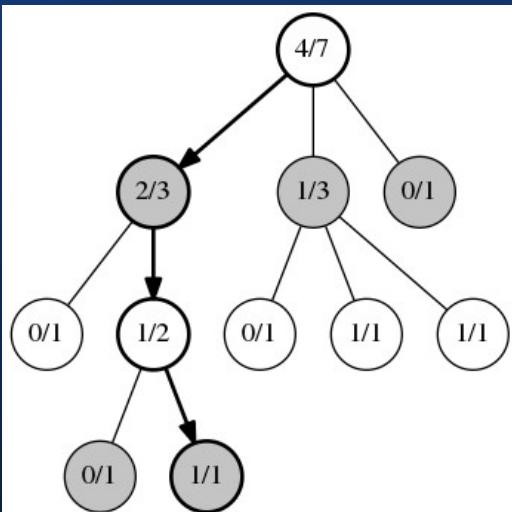
以下是《蒙地卡羅對局搜尋法》(MCTS) 的一個搜尋擴展範例

1. 選擇上界 UCB
最高的一條路
直到末端節點

2. 對該末端節點
進行探索 (隨機
對下，自我對局)

3. 透過自我對局，
直到得出本次對局
的勝負結果

4. 用這次的對局結果，
更新路徑上的勝負統計
次數！

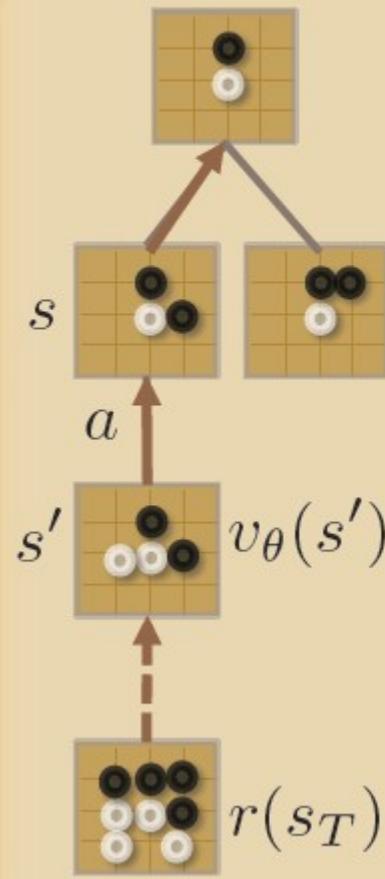


說明：上圖中白色節點為我方下子時的《得勝次數 / 總次數》之統計數據，

灰色的為對方下子的數據，本次自我對局結束後，得勝次數與總次數都會更新！

每次模擬對局完之後 都要更新統計資訊

Backup



Update the statistics of every visited edge (s, a) :

$$N_r(s, a) \leftarrow N_r(s, a) + 1$$

$$W_r(s, a) \leftarrow W_r(s, a) + r(s_T)$$

$$N_v(s, a) \leftarrow N_v(s, a) + 1$$

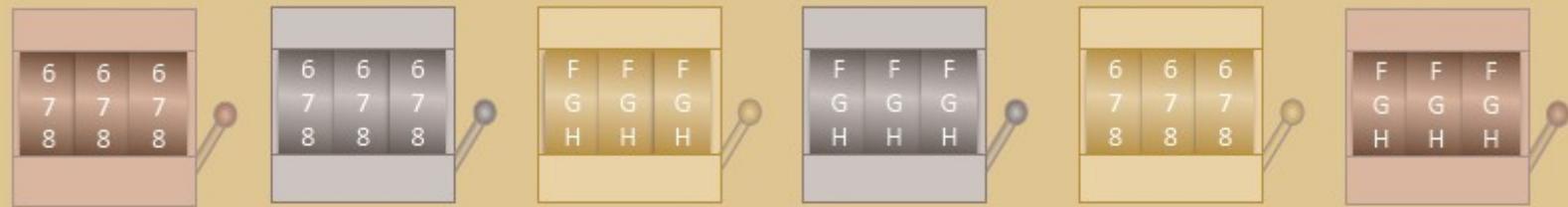
$$W_v(s, a) \leftarrow W_v(s, a) + v_\theta(s')$$

$$Q(s, a) = (1 - \lambda) \frac{W_v(s, a)}{N_v(s, a)} + \lambda \frac{W_r(s, a)}{N_r(s, a)}$$

↓
Interpolation constant

整個蒙地卡羅搜尋過程 都是由信賴區間上界 (UCB) 所指導的

UCB1 algorithm



$$\operatorname{argmax}_i (\bar{x}_i + \sqrt{\frac{2\log n}{n_i}})$$

- \bar{x}_i : The mean payout for machine i
- n_i : The number of plays of machine i
- n : The total number of plays

所以稱為 UCB1 演算法

UCB1 algorithm

$$R(s, a) = Q(s, a) + c \sqrt{\frac{\log N(s)}{N(s, a)}}$$

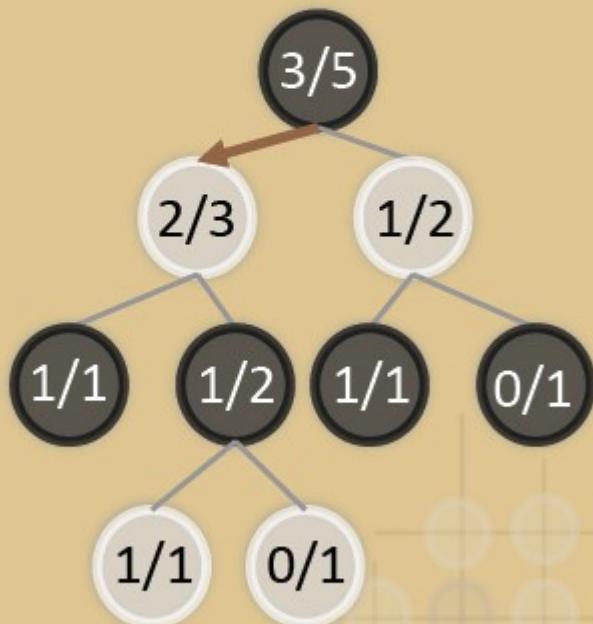
$$a^* = \operatorname{argmax}_a R(s, a)$$

$$N(s) = 5$$

$$Q(s, a) = 2/3$$

$$N(s, a) = 3$$

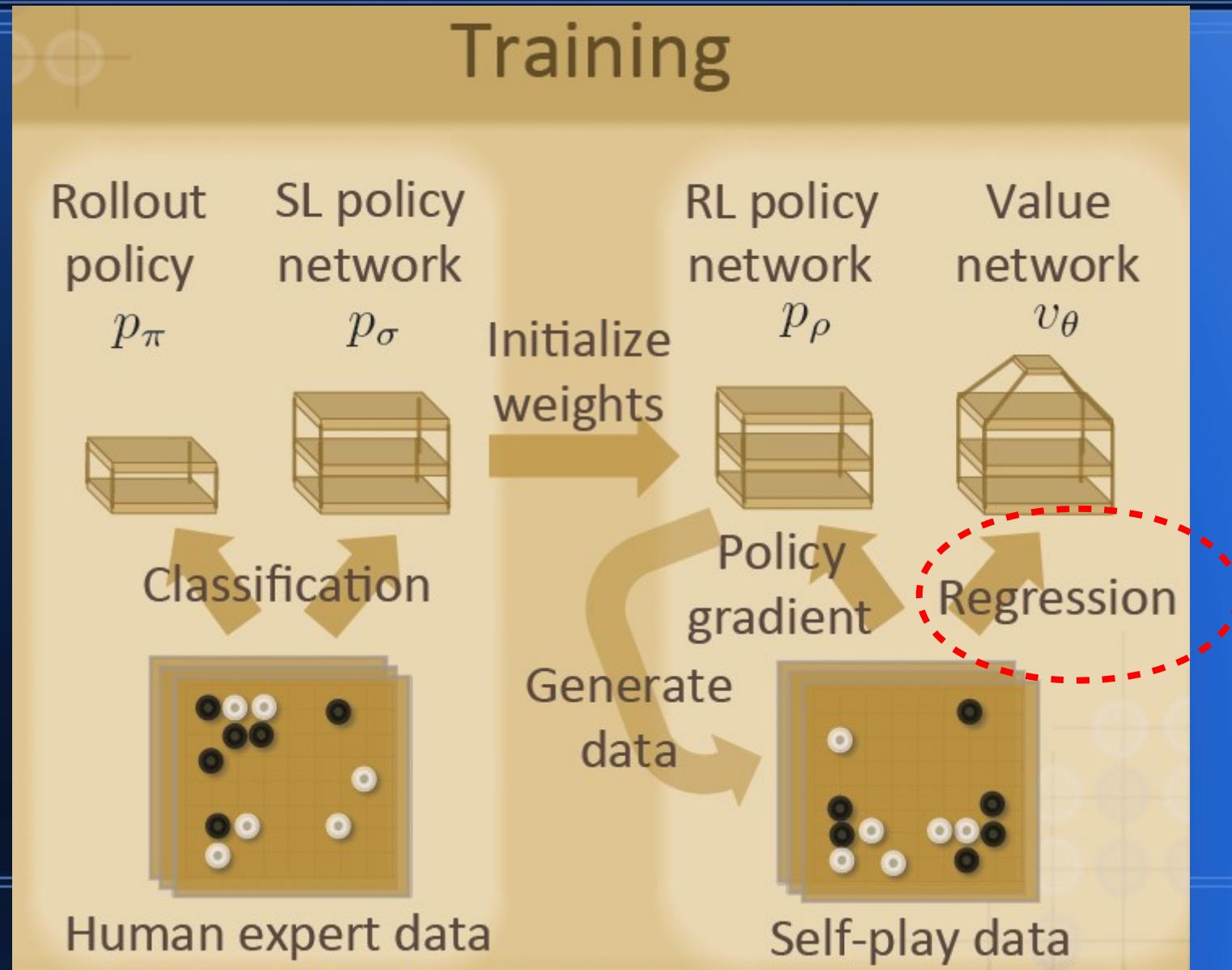
$$c = \text{constant}$$



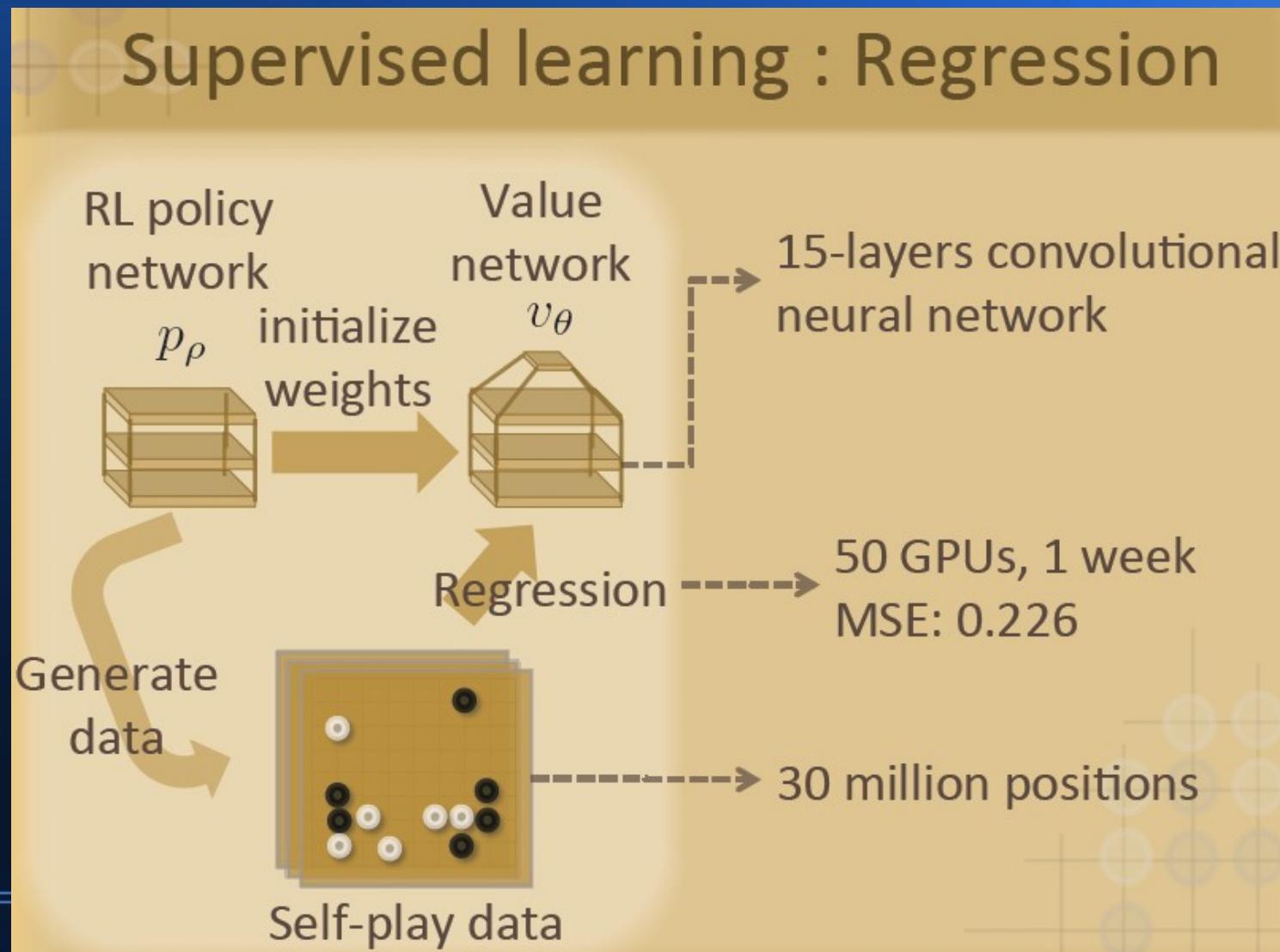
於是、透過

- 捲積神經網路 + 蒙地卡羅搜尋法
- 並利用《強化式學習》反饋並調整《捲積神經網路》的權重
- 就可以得到《策略網路》
(policy network)

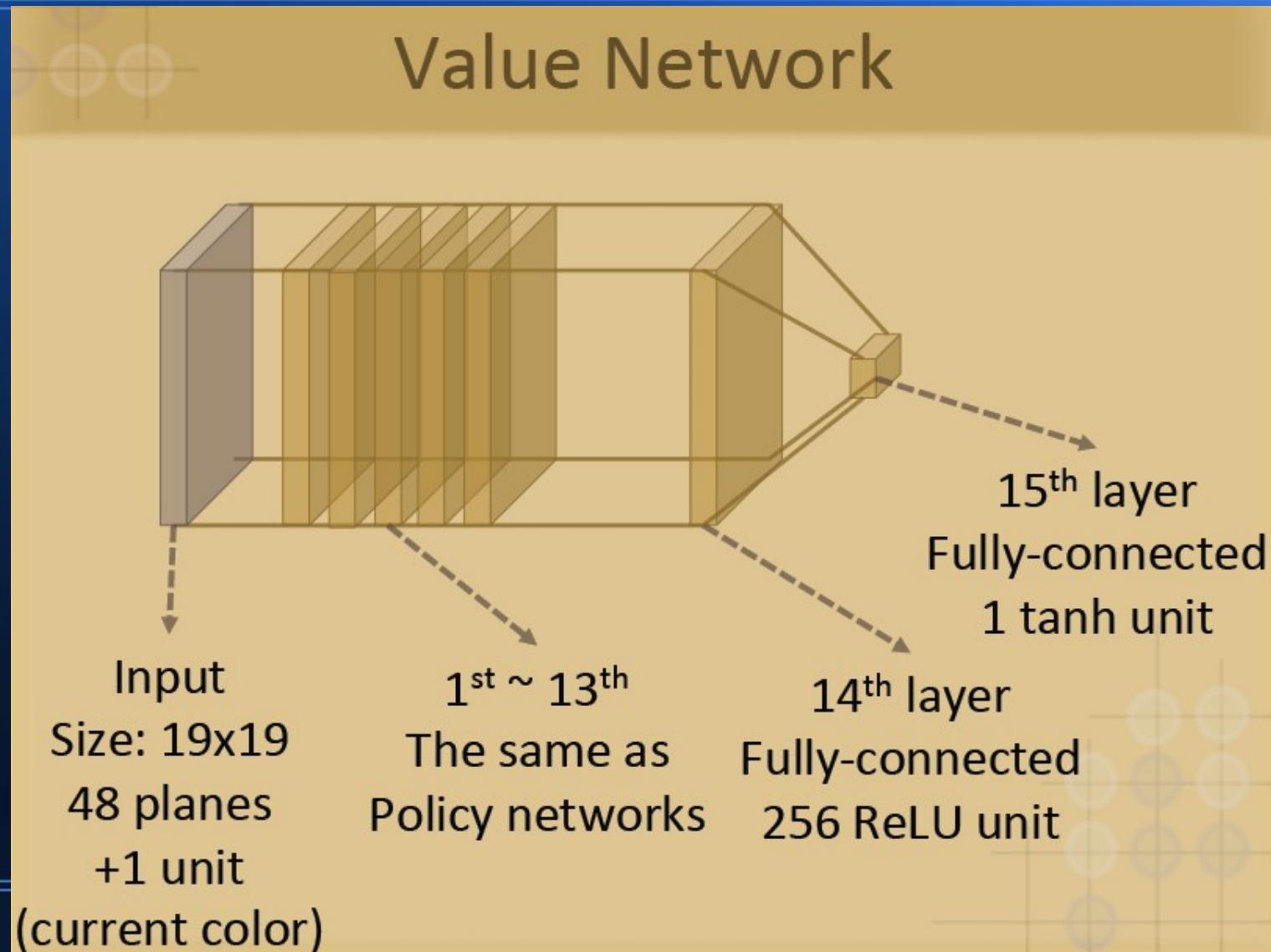
最後再透過策略網路
用迴歸的方式計算出價值網路



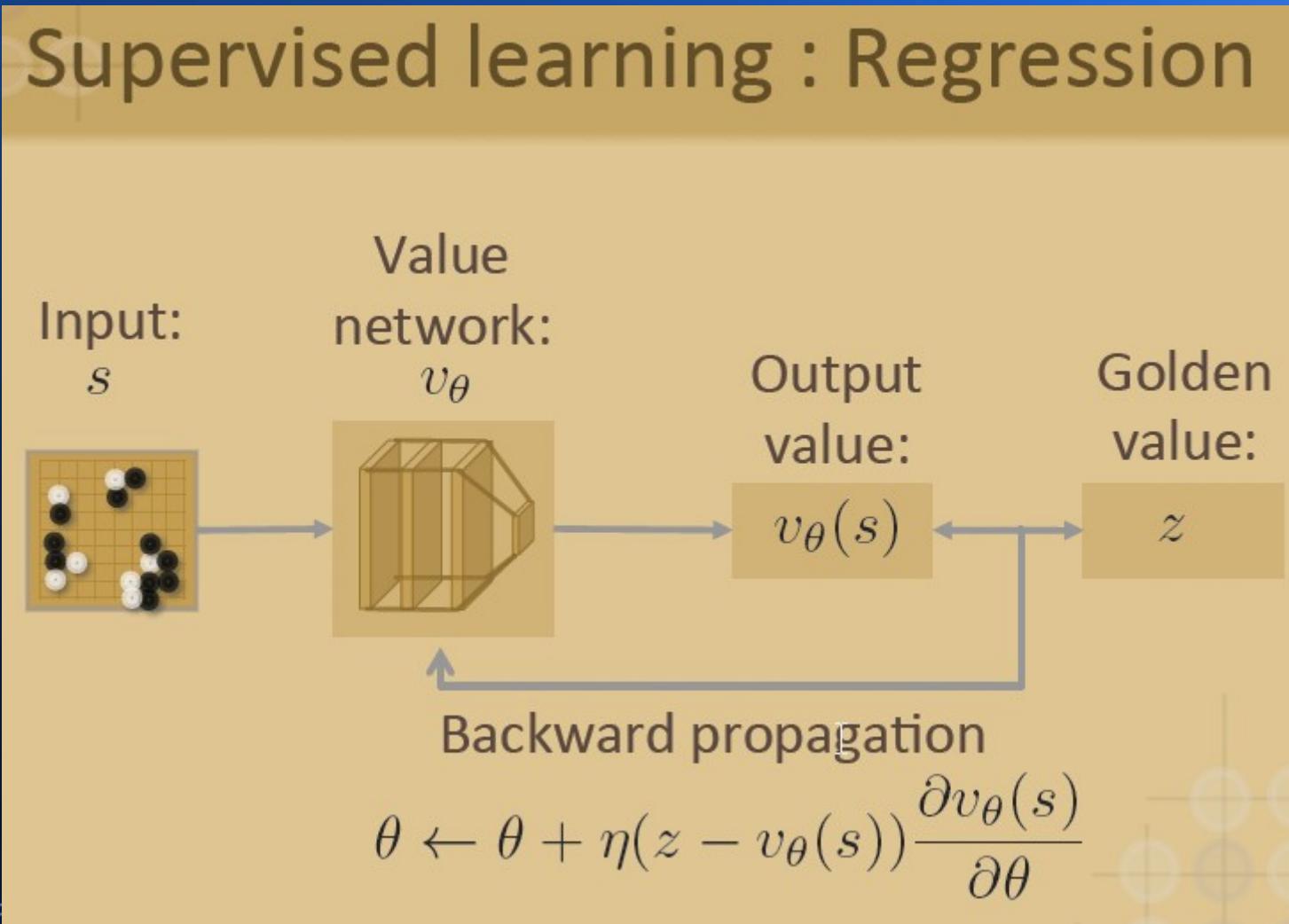
方法是把學到的策略網路
加上兩層之後再訓練



第14層是全連結 256 單元的 ReLU 層
第15層則只是權重加總的單一輸出

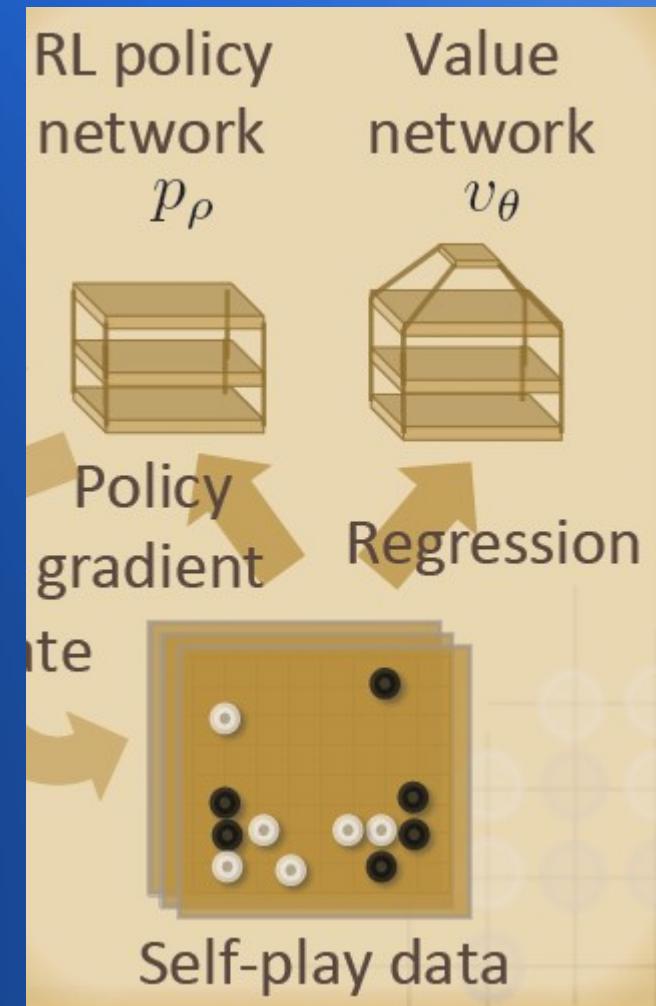


其訓練調整的公式如下



於是 AlphaGo 就學到了

- 策略網路 $P(a|s)$
- 價值網路 $V(s)$



最後、在 AlphaGo 真正下棋的時候

- 除了可以用《策略網路》 $P(s|a)$ + 《蒙地卡羅樹狀搜尋》來限縮搜尋的分支數量（廣度）以外！
- 也可以用《價值網路》 $V(s)$ 直接限縮搜尋的深度《在一層直接用 $V(s)$ 判定盤面分數，就不用一路搜到底浪費太多時間了》

以上就是

- 我目前對 AlphaGo 圍棋程式設計原理的認知！

希望、這份加長三倍的十分鐘系列

- 能讓你對 AlphaGo 的設計原理，有更進一步的認識！

我們下回見囉！

Bye Bye !