國立交通大學

資訊科學與工程研究所

碩士論文

六子棋時序差異學習之研究

Temporal Difference Learning in Connect6

研究生：蔡心迪

指導教授：吳毅成　教授

中華民國　一百　年七月

六子棋時序差異學習之研究
Temporal Difference Learning in Connect6

研 究 生：蔡心迪　　Student：Hsin-Ti Tsai
指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

July 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年七月

六子棋時序差異學習之研究

研究生：蔡心迪　　指導教授：吳毅成

國立交通大學 資訊科學與工程研究所

摘要

六子棋是吳毅成教授 2005 年發表的一個棋類遊戲，近年來已經發展成為世界性的遊戲。而 NCTU6 則是一個六子棋的 AI 程式，曾獲得兩屆電腦奧林匹亞賽局競賽六子棋組金牌，以及在人機賽中擊敗許多棋士。

這篇論文的目的是研究如何應用時序差異學習(Temporal Difference Learning)於六子棋程式上。由於六子棋的一些遊戲特性和其他遊戲不同，因此這篇同時也討論了時序差異學習在六子棋上有哪些值得注意的議題。

根據我們的實作經驗，時序差異學習演算法能讓新的程式對上原本的程式 NCTU6 有 57.95% 的勝率，顯示 NCTU6 確實增加了棋力。新的程式也在 2011 TCGA 六子棋組獲得了金牌的好成績。

Temporal Difference Learning in Connect6

Student: Hsin-Ti, Tsai Advisor: I-Chen, Wu

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

The Connect6 game, first introduced by Professor I-Chen Wu in 2005, now becomes one of the popular games in the world. NCTU6 is a Connect6 AI program developed by our team, has won gold medal in ICGA tournaments twice and defeated many professional players in Man-Machine Connect6 championship.

The main purpose of this thesis is to research temporal difference learning in Connect6. Some characteristics of Connect6 are different from other games, so we also discuss the issues of temporal difference learning in Connect6.

According to our practical experience, the new program with temporal difference learning reaches a 57.95% win percentage against original program. This shows that the new method successfully improves NCTU6. The new program has also won gold medal in 2011 TCGA tournaments.

誌謝

這篇論文之得以完成，首先要感謝我的父母，謝謝你們辛苦培養我長大，在求學過程中不斷給我支持，沒有你們就沒有今天的我。

感謝我的指導教授吳毅成老師，指出我許多要改善的地方，並且指導了我研究該有的態度和許多方法，讓我在研究期間這兩年有許多的收穫和成長。

感謝各位口試委員所提供的建議，讓這篇論文的架構更趨於完善，也指出了論文中的小缺點，使論文的品質更為提升。

感謝實驗室的學長姐們，林秉宏學長指導我解決許多程式上的問題，林宏軒學長細心幫我修稿以及提供許多寫作時的經驗，林沂珊學姐在我口試時幫我詳細記錄了口委們的建議。

感謝實驗室的同學們，林正宏在很多地方都幫了我很多忙，陳昱維和韓尚餘提供和維護實驗的系統和機器，讓我能順利跑完實驗。甘崇緯和楊景元也提供了許多我在實作上的意見。

感謝實驗室的學弟們，康皓華跟鄭吉閎幫我準備口試時的餐點，讓我能夠更專心於口試內容的準備。

最後，僅以此篇論文獻給所有陪伴我的家人、老師和朋友們，謝謝你們。

民國一百年七月　於　新竹交大工程三館 511 實驗室

目錄

摘要 i
Abstract ii
誌謝 iii
目錄 iv
圖目錄 vi
表目錄 vii
第一章、介紹 1
1.1 六子棋 1
1.2 NCTU6 3
1.3 研究目的 4
1.4 貢獻 4
1.5 論文組織 5
第二章、研究背景 6
2.1 迫著空間搜尋 6
2.2 Alpha-Beta Search 9
2.2.1 Mini-Max Search 10
2.2.2 評估函數(Evaluation Function) 10
2.2.3 特徵(Feature) 11
2.2.4 Alpha-Beta Search 12
2.3 NCTU6 的搜尋 12
2.3.1 NCTU6 的迫著空間搜尋 13
2.3.2 NCTU6 的 Alpha-Beta Search 13
2.4 時序差異學習 13
2.4.1 策略(Policy) 14
2.4.2 回饋 15
2.4.3 模型 16
2.4.4 TD(λ) 16
2.4.5 TD(0)和 TD(1) 19
第三章、研究方法 21
3.1 簡單的時序差異學習 21
3.2 虛擬程式碼 25
3.2.1 TD Learning 25
3.2.2 SelfPlay 26
3.2.3 Greedy 27

3.2.4 Eval 28
3.3 實作議題 28
3.3.1 特徵的選擇 28
3.3.2 特徵標準化 30
3.3.3 迫著空間搜尋 31
3.3.4 讀高手棋譜訓練 32
第四章、實驗 33
4.1 實作細節 33
4.1.1 更新比例 33
4.1.2 選點策略 33
4.1.3 更新標準化 34
4.1.4 分數標準化 34
4.1.5 回饋(Reward) 35
4.1.6 兩層更新 35
4.2 實驗環境 36
4.3 區分階段 37
4.4 特徵標準化 39
4.5 迫著空間搜尋 39
4.6 讀高手棋譜訓練 41
第五章、結論與未來展望 43
參考文獻 44

圖目錄

圖 1. 六子棋棋局 1
圖 2. 棋局範例 3
圖 3. 迫著範例 7
圖 4. 雙迫著追殺範例 8
圖 5. 單迫著追殺範例 9
圖 6. Mini-Max Search Tree 10
圖 7. Alpha-Beta Search 12
圖 8. 時序差異學習的模型 14
圖 9. 時序差異學習一輪的狀態序列 16
圖 10. TD(λ)更新示意圖 18
圖 11. TD(0)示意圖 19
圖 12. TD(1)示意圖 19
圖 13. 擲硬幣樹狀圖 21
圖 14. 擲硬幣訓練結果 22
圖 15. 第二個例子樹狀圖 23
圖 16. 第二個例子訓練結果 24
圖 17. 第三個例子訓練結果 25
圖 18. 離中心差別示意圖 29
圖 19. 棋型方向差別示意圖 29
圖 20. 區分階段差別示意圖 30
圖 21. 去掉必勝路徑示意圖 31
圖 22. σ(x)函數圖 35
圖 23. 兩層更新示意圖 36
圖 24. 區分階段實驗結果 37
圖 25. 加入開局使用手工特徵分數實驗結果 38
圖 26. 特徵標準化實驗結果 39
圖 27. 迫著空間搜尋實驗結果 40
圖 28. 讀高手棋譜實驗結果 41

表目錄

表 1. Game Tree Complexity..... 2
表 2. NCTU6 戰績表..... 4
表 3. 特徵分數比較表..... 40
表 4. 時間比較表..... 42

第一章、介紹

一開始的章節中，會來介紹一些背景知識，章節 1.1 講解本篇論文的應用遊戲六子棋，章節 1.2 介紹用來加強的程式，章節 1.3 提出研究的動機和目的，章節 1.4 簡單整理此篇論文的貢獻，章節 1.5 介紹整篇論文的架構。

1.1 六子棋

六子棋是交大吳毅成教授在 2005 年 9 月發表於第十一屆電腦賽局發展學術研討會(ACG 11)的一個遊戲，玩家分成黑白雙方，各持黑子和白子，六子棋有三個重要的特性[1]。

第一個特性是規則簡單，規則如下，首先第一次黑方先下一顆子，之後雙方輪流每次下兩子，先連成六子以上的就算獲勝，沒有任何禁手。規則非常簡單，不像五子棋的日規為了平衡黑白雙方，讓先手限制許多禁手導致規則複雜。而棋盤一般採用圍棋的十九路棋盤，若下到滿仍無法分出勝負則判和局。如圖 1 是一個六子棋棋局的例子，白方連六勝。

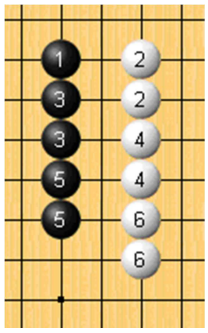


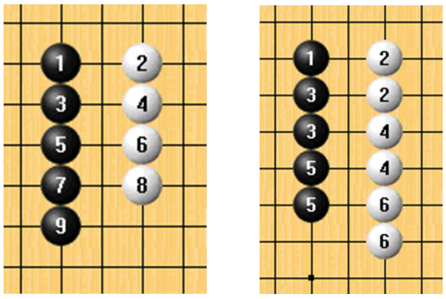
圖 1. 六子棋棋局

第二個特性是變化複雜，由於六子棋一次可以下兩顆，使得變化相當豐富。而如果從學術的角度來看的話，可以來分析六子棋的 Game Tree Complexity[4][5][7]，一局平均下來一顆子可能下的位置約有 300 個，則一手可能的變化就約有 300*300/2 種，平均棋局長度大概是 30~40 手，因此Game Tree Complexity 約為$(300*300/2)^{30}$~$(300*300/2)^{40}$= 10^{140}~10^{188}，表 1 列出了常見的遊戲及其 Game Tree Complexity，可以看到六子棋大約介於象棋和將棋之間。

Games	Game Tree Complexity
Go（圍棋）	10^{360}
Shogi（將棋）	10^{226}
Connect6（六子棋）	$10^{140\text{~}188}$
Chinese Chess（象棋）	10^{150}
Chess（西洋棋）	10^{123}
Go-Moku（五子棋）	10^{70}

表 1. Game Tree Complexity

第三個特性是遊戲公平，首先先從五子棋來看，五子棋普遍是認為對先手黑方比較有利，這點可以從棋子數量來觀察，當黑方下完時，棋盤上的黑棋數一定比白棋數多一顆子，然而當白方下完時，棋盤上的白棋數頂多和黑棋數相同，見圖 2.a，因此這不平衡的現象使得黑棋在五子棋中較具有優勢。然而六子棋則不一樣，不管是黑白任何一方下完時，棋盤上的數量一定比對方多一顆子，不會偏頗任何一方，見圖 2.b，因此從棋子數量的角度來看的話，六子棋是對雙方較為平衡的一個遊戲。



(a) (b)

圖 2. (a) 五子棋棋局 (b)六子棋棋局

六子棋從 2005 年發表至今已經有許多發展,像是維基百科當中也有六子棋項目，也是電腦奧林匹亞賽局競賽的比賽項目之一，在許多知名的線上遊戲如 Little Golem[11]、Pente[13]、Ludoteka[12]等也都有提供六子棋遊戲，並且也舉辦過許多六子棋的比賽像是交通大學盃六子棋公開賽、人腦對電腦六子棋大賽等。因此六子棋可以說是世界性的遊戲之一。

1.2 NCTU6

NCTU6，又稱交大六號，是一支六子棋的電腦 AI 程式，是由交大吳毅成教授的團隊所開發，作者有以下這幾位：吳毅成、林秉宏、蔡心迪、康皓華。

時間	比賽	結果
2006 年	第十一屆國際奧林匹亞電腦賽局競賽六子棋組	「NCTU6」獲得冠軍
2008 年	第十三屆國際奧林匹亞電腦賽局競賽六子棋組	「NCTU6-Lite」獲得冠軍
2008/8/9	世界棋王周俊勳與電腦六子棋對抗賽	「NCTU6」獲得 3 勝 0 負
2008/8/24	第一屆人腦對電腦六子棋大賽	「NCTU6」獲得 11 勝 1 負
2009/10/11	第二屆人腦對電腦六子棋大賽	「NCTU6」獲得 8 勝 0 負
2011/3/20	第三屆人腦對電腦六子棋大賽	「NCTU6」獲得 5 勝 3 負

表 2. NCTU6 戰績表

表 2 列的是 NCTU6 至今所獲得的一些成就，從上表的成績可以看出 NCTU6 的棋力非常強。

1.3 研究目的

此篇論文的動機，是想研究有沒有辦法更進一步增加 NCTU6 的棋力，而此篇決定研究的著眼點在於 NCTU6 的參數。原本的 NCTU6 有許多參數都是用人工的方式去調整決定，雖然當時在決定時也是經過反覆實驗取出比較好的一組參數當作最後的結果，不過是否存在更好的參數呢？因此此篇論文便是想將原本的人為調整，改成自動調整來學習，看是否能得到更好的參數，藉此來加強 NCTU6 的棋力，而參考的學習方法則為時序差異學習(Temporal Difference Learning)[3][14]。

1.4 貢獻

此篇論文的主要貢獻如下：

- 將時序差異學習用淺顯易懂的例子來說明。
- 實作時序差異學習在 NCTU6 上，並成功提高其棋力。
- 分析時序差異學習應用在六子棋上面時，有哪些值得注意的議題。

1.5 論文組織

此篇論文的第二章，會來介紹一些過去的研究，包含迫著空間搜尋(Threat Space Search)[9][10]、Alpha-Beta Search[2]、NCTU6 的搜尋、時序差異學習，第三章是研究方法，首先會舉個例子來簡單說明時序差異學習，再說明實作時的虛擬程式碼，然後介紹實作在六子棋上時有哪些議題要討論，第四章是實驗，會介紹實作的細節，以及列出各種機制的比較，最後第五章作結論和未來展望。

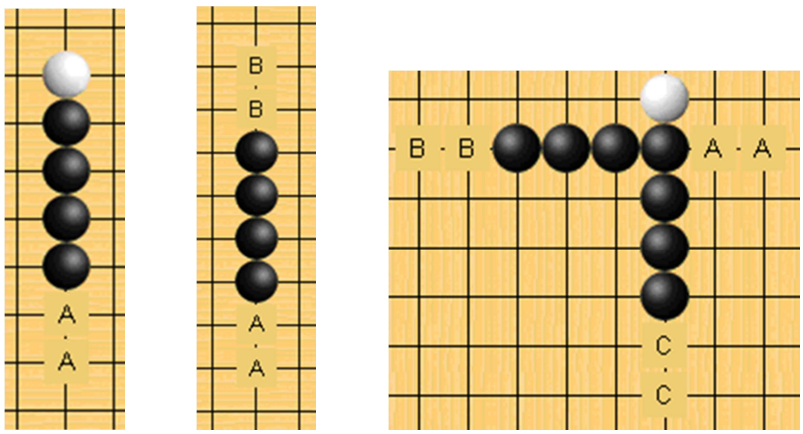
第二章、研究背景

此章節首先先介紹迫著空間搜尋(Threat Space Search)[9][10]和 Alpha-Beta Search[2]，再來會介紹 NCTU6 是如何使用這兩種搜尋，最後是此篇使用的學習方法時序差異學習(Temporal Difference Learning)[3][14]。

2.1 迫著空間搜尋

迫著(Threats)的詳細定義如下:若對方最少要下 t 顆子來避免我方下一步直接獲勝，則代表我方有 t 個迫著。

以六子棋為例，圖 3.a 是黑方有單迫著(Single-Threat)的例子，白方至少要下一顆在 A 的位置，如果沒下的話，下一步黑方就能直接下在兩個 A 連六獲勝。圖 3.b 則是黑方有雙迫著(Double-Threat)的例子，白方至少要在 A 和 B 各下一顆共兩顆才能阻止，當然也不能兩顆都下在外側，否則下一步黑方也可以下在兩個內側連六獲勝。圖 3.c 則是黑方有三迫著(Three-Threat)的例子，白方至少要在 ABC 各下一顆共三顆才能阻止，但是六子棋一次只能下兩顆子，因此這個盤面實際上已經是黑必勝的盤面。



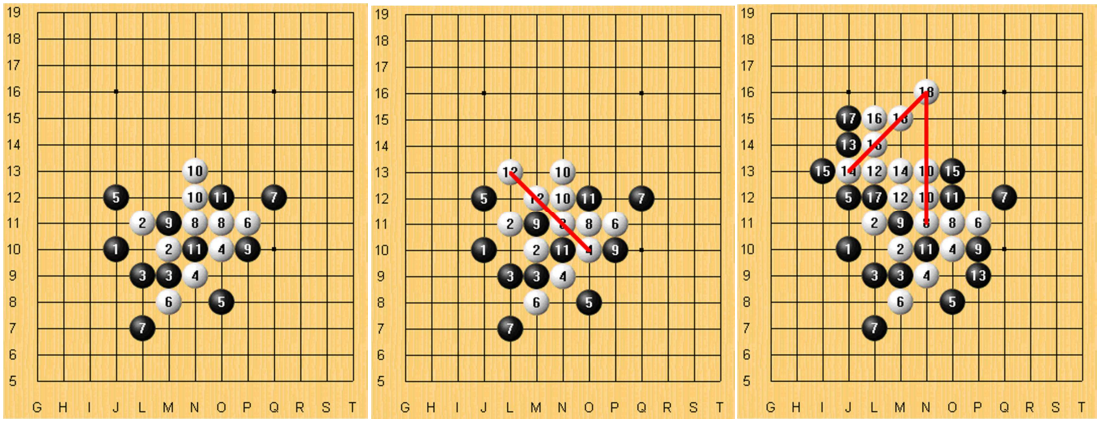
(a) (b) (c)

圖 3. (a)單迫著例子 (b)雙迫著例子 (c)三迫著例子

而迫著空間搜尋就是利用迫著，來搜尋找出是否存在必勝法。我們定義想要找到必勝法的那方為攻擊方，想防止對方找到必勝法的那方為防守方。在六子棋中，迫著空間搜尋可簡單分成以下三種[6][7][15]：

- 雙迫著追殺(Victory by Continuous Double-Threat-or-more moves; VCDT)
- 單迫著追殺(Victory by Continuous Single-Threat-or-more moves; VCST)
- 無迫著追殺(Victory by Continuous Non-Threat-or-more moves; VCNT)

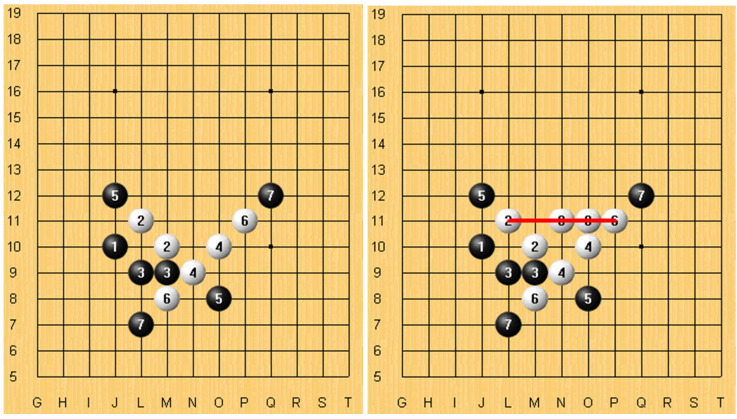
雙迫著追殺，是指攻擊方每次下完，至少要有兩個迫著，在這樣的條件下搜尋找出是否存在必勝法。舉圖 4 為例，攻擊方為白方，白方在圖 4.a 時已經存在必勝法，可以下白 12 這個下法形成兩個迫著，如圖 4.b，這麼一來黑方能擋的下法就非常有限，之後每次都下完產生兩個迫著，一路下到白 18 形成三個迫著必勝，如圖 4.c，由於攻擊方每次下完都會有兩個迫著，防守方的兩顆子一定都要拿來擋這個兩個迫著，使得能下的地方非常少，這可以大大地減少搜尋複雜度，讓整個雙迫著追殺的搜尋很快就結束。



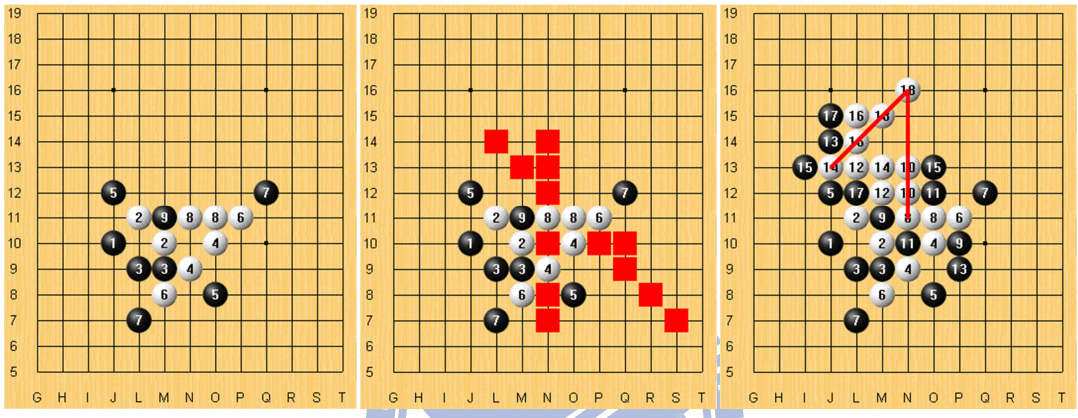
(a)　　(b)　　(c)

圖 4. 雙迫著追殺範例 (a)起始盤面 (b)白 12 形成雙迫著 (c)白 18 形成三迫著必勝

單迫著追殺，是指攻擊方每次下完，至少要有一個迫著，在這樣的條件下搜尋找出是否存在必勝法。舉剛剛的例子為例，實際上在更早，也就是圖 5.a 時，攻擊方白方就已經有必勝法了。白方可以下白 8 形成一個迫著，如圖 5.b，這麼一來防守方至少要花一顆子擋住這個迫著，我們假設黑 9 擋在中間，如圖 5.c。但是這樣防守方黑方還是有一顆自由的子有許多空點可以選擇，這樣子搜尋還是很花時間，所以我們可以建立一個區域 RZone，如圖 5.d 的方型區域，可以證明另一顆子下在 RZone 以外的話都是必敗的。至於 RZone 要如何建立，可以參考[5][6]，這樣我們就只需要將搜尋集中在 RZone 裡面即可。如果搜尋完不管另一顆子下在 RZone 的任何一個位置，攻擊方都還是能一路下下去每次至少有一個迫著，最後形成三個迫著，如圖 5.e，這樣我們就能證明圖 5.c 黑 9 擋這個位置是個必敗的擋法。



(a) (b)



(c) (d) (e)

圖 5. 單迫著追殺範例 (a)起始盤面 (b)白 8 形成單迫著 (c)黑 9 擋 (d)RZone (e)白 18 形成三迫著必勝

無迫著追殺，是指攻擊方每次下完，可以沒有任何迫著，在這樣的條件下搜尋找出是否存在必勝法。由於攻擊方下完可以允許沒有迫著，使得防守方最多有兩顆子可以自由下，雖然還是可以利用 RZone 來限制其搜尋範圍，不過普遍來說搜尋無迫著追殺還是很花時間。

2.2 Alpha-Beta Search

Alpha-Beta Search 是 Mini-Max Search 的一種改良，因此首先先來介紹 Mini-Max Search 以及相關的知識，再回過頭來介紹 Alpha-Beta

Search

2.2.1 Mini-Max Search

下圖是一個 Mini-Max Search Tree 的例子，在 Mini-Max Search Tree 中分成兩種節點(node)，一種是方型的 Max node，一種是圓形的 min node。每個節點上的數字代表分數，分數越高對我方越有利，分數的更新是從 leaf node 往上更新，而 leaf node 的分數可由評估函數得到，這個下一小節會提到。至於更新的方式，Max node 可以想像成是我方，我方一定會選對我方最有利的走法，因此會選分數最大的 children，更新成自己的分數，而 Min node 則可以想像成是對方，對方一定會選對我方最不利的走法，因此會選分數最小的 children，更新成自己的分數。最後更新完，每個節點的分數，就代表若進行到這個節點，雙方都是最強時結果會是多少分數，而圖中的粗線，代表雙方都走最好時形成的路線。

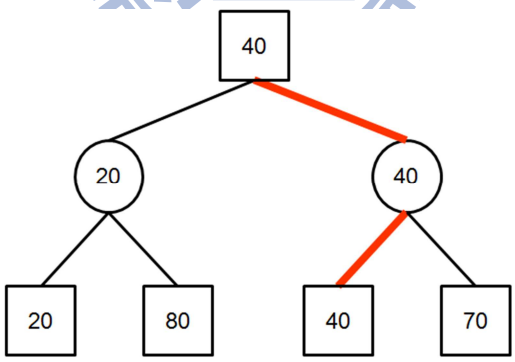


圖 6. Mini-Max Search Tree

2.2.2 評估函數(Evaluation Function)

評估函數，是用來評估某個狀態對我方有多少好處，並且用一個量化的數字來表示，前一小節提到的 leaf node 的分數，便可以用評估函數得

到。

我們定義 V 為評估函數，而 $V(s)$則是估計狀態 s 價值多少分數。

2.2.3 特徵(Feature)

特徵，是指某個狀態具有的一些特別的徵象，舉六子棋為例的話，活一棋型、活二棋型都可以當作是特徵。活 i(死 i)的詳細定義如下，若某個棋型再下 4-i 顆子，可形成雙迫著(單迫著)的話，代表此棋型為活 i(死 i)，例如活二代表再下兩顆子可以形成雙迫著。

我們定義 $\varphi(s)$，代表某個狀態 s 的特徵數量，$\varphi(s)$是一個向量。延續上面的例子，若 $\varphi(s)$ = [3,2]，則代表盤面 s 有 3 個活一和 2 個活二。

再來定義 θ，代表特徵分數，θ 也是一個向量。例如 θ = [10,20]，代表活一價值 10 分，活二價值 20 分。

則前一小節提到評估函數則可以利用特徵數量和特徵分數內積得到公式(1)

$$V(s)=\varphi(s)\cdot\theta \qquad (1)$$

舉剛剛的例子[3,2]· [10,20] = 3*10+2*20 = 70(分)

當然並不是評估函數一定要用這種方式才可以計算出分數，只不過用這種方法有許多好處，像是實作簡單，效率高，也方便除錯等，而且這種線性的組合有許多數學特性適合用在很多演算法，因此有許多評估函數是採用這種方式。

2.2.4 Alpha-Beta Search

Alpha-Beta Search 是 Mini-Max Search 的一種改良，我們的目的是為了得到 root node 的分數，才能知道在 root node 時要選擇哪個 children。原本的 Mini-Max Search，有許多地方可以不需要搜尋就得到 root node 的分數。舉圖 7 為例，節點 A 是一個 min node，已經找完第一個 children 得知其分數是 10 分，這時候可以得知 A 的分數一定是小於等於 10 分。而節點 B 是一個 max node，已經找完第一個 children 得知其分數是 15 分。則在第一個 children 15 分和第二個 children(節點 A)小於等於 10 分之間，A 一定會選擇第一個 children，因此節點 B 確切的分數是多少已經不重要，B 剩下的 children 就可以 cut-off 不做搜尋。

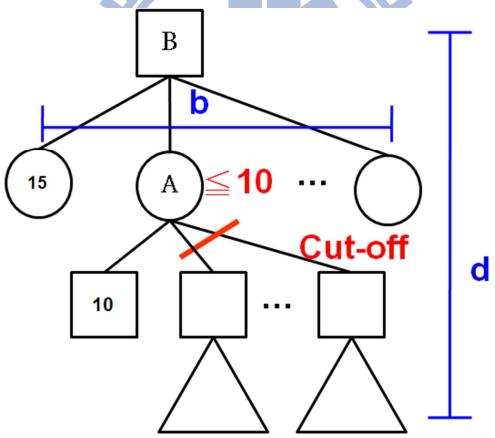


圖 7. Alpha-Beta Search

類似的方式可以砍掉許多 subtree，省下很多搜尋時間。若 b 為搜尋樹寬度，d 為搜尋樹高度，在 best case 時，甚至可以將原本的時間複雜度從 $O(b^d)$ 改良至 $O(b^{d/2})$[8]。因此跟原本的 Mini-Max Search 比，效率可以說是大幅提升。

2.3 NCTU6 的搜尋

NCTU6 的搜尋主要可以分成兩個階段，第一階段是迫著空間搜尋，第二階段是 Alpha-Beta Search，也就是前兩節所介紹的搜尋，而由於 NCTU6 是要拿來實戰用，因此作法稍微有些變化，在接下來的小節作說明。

2.3.1 NCTU6 的迫著空間搜尋

NCTU6 在第一階段迫著空間搜尋時，只有搜尋雙迫著追殺和單迫著追殺，而沒有無迫著追殺。這是因為無迫著追殺普遍搜尋寬度太寬，甚至是不存在必勝法時，要搜尋完整個搜尋樹所花的時間會相當驚人，因此無迫著追殺在目前的 NCTU6 暫時不考慮。

而在搜尋雙迫著追殺和單迫著追殺，也會設定深度限制和時間限制，避免某些情況下搜尋花太多時間。因此就算存在雙迫著追殺或單迫著追殺必勝法，若這個必勝法太深太難，NCTU6 也不一定能找的到。

2.3.2 NCTU6 的 Alpha-Beta Search

若是 NCTU6 在第一階段迫著空間搜尋找不到必勝法，就會進入第二階段 Alpha-Beta Search，NCTU6 會固定搜尋一定的寬度和一定的高度，最後找出一個好步。

而在 Alpha-Beta Search 時，也可以用到迫著空間搜尋，若是搜尋樹的某個節點可以用迫著空間搜尋證明是對方必勝，則我方就不考慮這個節點，以此類推。這麼一來也可以 cut-off 掉許多 subtree 以節省時間。

2.4 時序差異學習

時序差異學習是一種學習的演算法，用來調整狀態分數，若是評估函數是採用特徵數量和特徵分數的線性組合的話，那也可以用來調整特徵分數，進而影響狀態分數。

時序差異學習有兩個重要的特性，一個是試誤法搜尋(trial-and-error search)，一個是延遲回饋(delayed reward)。試誤法搜尋代表會從過去的經驗學習成長，延遲回饋代表會根據行動之後得到的回饋(reward)當作主要依據。舉一個例子，將某支老鼠放在有兩個開關的箱子中，其中一個開關是正確答案，按下會跑出食物，則延遲回饋代表的就是這些食物，而試誤法搜尋代表老鼠學習的過程，會因為選錯沒東西吃而變聰明，之後就會比較容易選到正確答案。

下圖是時序差異學習的模型(model)示意圖，2.4.1~2.4.3 會來定義解釋這個模型，2.4.4~2.4.5 則來介紹時序差異學習是如何學習。

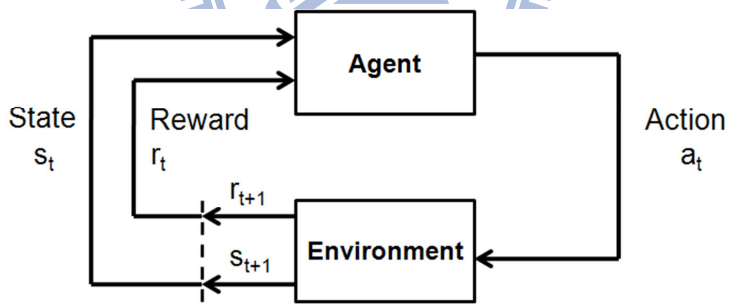


圖 8. 時序差異學習的模型

2.4.1 策略(Policy)

策略，是指 Agent 根據目前的狀態(state)，選擇動作(action)時所採用的計畫。舉例來說，Agent 可能代表一個棋士，則策略代表他腦中的想法，會根據目前的盤面，選擇要下哪一步。我們可以用 π 來代表一個固定

的策略，而 $\pi(s,a)$代表根據策略 π 在狀態 s 會選擇動作 a 的機率。

有一些常用的策略，像是均勻隨機策略(uniform random policy)，我們定義 $Q(s,a)$為估計狀態 s 在執行動作 a 之後價值多少分數，而$|A(s)|$代表狀態 s 下有多少種動作可以選，則均勻隨機策略的選點策略如公式(2)，代表不管是哪個動作都有相同的機率選到。

$$\pi(s,a)=\frac{1}{|A(s)|} \tag{2}$$

還有貪婪策略(greedy policy)的選點策略如公式(3)，ε 是一個介於 0 到 1 的參數，最好的那個動作有較高的機率被選到，比其餘的動作機率高了 $1-\varepsilon$ ，ε 越接近 1 則代表越接近均勻隨機策略。

$$\pi_{\varepsilon}(s,a)=\begin{cases}1-\varepsilon+\dfrac{\varepsilon}{|A(s)|} & if\ a=\arg\max_{a'} Q(s,a') \\ \dfrac{\varepsilon}{|A(s)|} & otherwise\end{cases} \tag{3}$$

2.4.2 回饋

回饋代表作了某個動作後所得到的好處。如示意圖，在狀態 s_t 時選擇一個動作 a_t，影響環境(environment)使狀態變成 s_{t+1}，這中間就會產生一個回饋 r_{t+1}。舉棋類遊戲來說，可能下到最後一步獲勝，獲得回饋+1，最後一步輸則獲得回饋-1，最後一步平手或是下到棋局一半還沒結束，都得到回饋+0，這麼一來若要得到最多回饋，也就會越想要贏。另一個例子舉機器人走迷宮，在迷宮中每走一步都得到一個回饋-1，直到走出迷宮，這麼一來若要得到最多回饋，也就會想要走最短路徑到出口。

定義 $V^{\pi}(s)$代表狀態 s，在根據策略 π 之下，得到的回饋期望值。時序差異學習的目的之一，就是希望 $V(s)$能夠越接近 $V^{\pi}(s)$越好，這麼一來就能使評估函數更具有鑑別度，因此我們也用 $V(s)$來估計 $V^{\pi}(s)$。

2.4.3 模型

那麼再回過頭來看圖的模型，學習的流程從起始狀態 s_0 開始，一直根據策略來選擇 action 直到到達終點狀態 s_T 為止，結束的話重新從起始狀態 s_0 開始下一輪的學習，不斷下去。這中間 Agent 會根據所得到的回饋來做學習，調整策略讓下次選擇時可以選到更好的動作。

2.4.4 TD(λ)

而 Agent 是怎麼做學習的呢？我們首先列出一條路，如下圖：

圖 9. 時序差異學習一輪的狀態序列

從起始狀態 s_0 開始，根據某個策略 π，一路選下去直到終點狀態 s_T。假設目前狀態進行到 s_t，如果我們想要以 s_t 為基準估計 s_t 之後到終點可以得到多少回饋，可以用 $V(s_t)$來估計。如果 π 作了一個動作，進行到 s_{t+1} 時，這時候我們可以重新估計，得到 $R_t^{(1)}$這個值如公式(4)：

$$R_t^{(1)} = r_{t+1} + V(s_{t+1}) \tag{4}$$

因為從 s_t 到 s_{t+1}，中間得到了一個回饋 r_{t+1}，因此可以重新估計得到 $R_t^{(1)}$這個更準確的值。因此若進行到 s_{t+n} 時，我們可以重新估計，得到 $R_t^{(n)}$：

$$R_t^{(n)} = r_{t+1} + r_{t+2} + \cdots + r_{t+n} + V(s_{t+n}) \qquad (5)$$

$R_t^{(n)}$包含了從 s_t 到 s_{t+n} 之間得到的 n 個回饋，再加上 s_{t+n} 之後估計得到的回饋。而最後結束時，可以知道 s_t 到終點 s_T 中間得到的回饋是一個明確的值，因此得到 R_t 這個值如公式(6)：

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_{T-1} + r_T \qquad (6)$$

那麼這些值有什麼用呢，想像一下假如我們用的策略 π 是完美的，而且評估函數 V 也是完美的，那麼 $V(s_t)$、$R_t^{(1)}$、…、$R_t^{(n)}$、…、R_t，這些值應該都會是相等的，因為只有當走錯路時，才會使得到的回饋有所偏差。反過來說，如果這些值不同的話，則我們用的策略 π 或是評估函數 V 就不可能是完美的，因此就可以來修正使這些值盡量接近，所以我們就取 $R_t^{(1)}$、…、$R_t^{(n)}$、…、R_t 拿來調整 $V(s_t)$這個值，而實際上 $V(s_t)$應該會較接近 $R_t^{(1)}$，因為這兩個狀態較為接近，越遠的值越可能差越多，所以我們取的比例最好是由大到小，所以使用等比公式，等比為 λ 介於 0 到 1 之間，$R_t^{(1)}$的比例為 1−λ，往後每次比例都乘上公比 λ，最後為了比例總和要為 1 而作些調整，因此 R_t 的比例為 λ^{T-t-1}，如下圖。

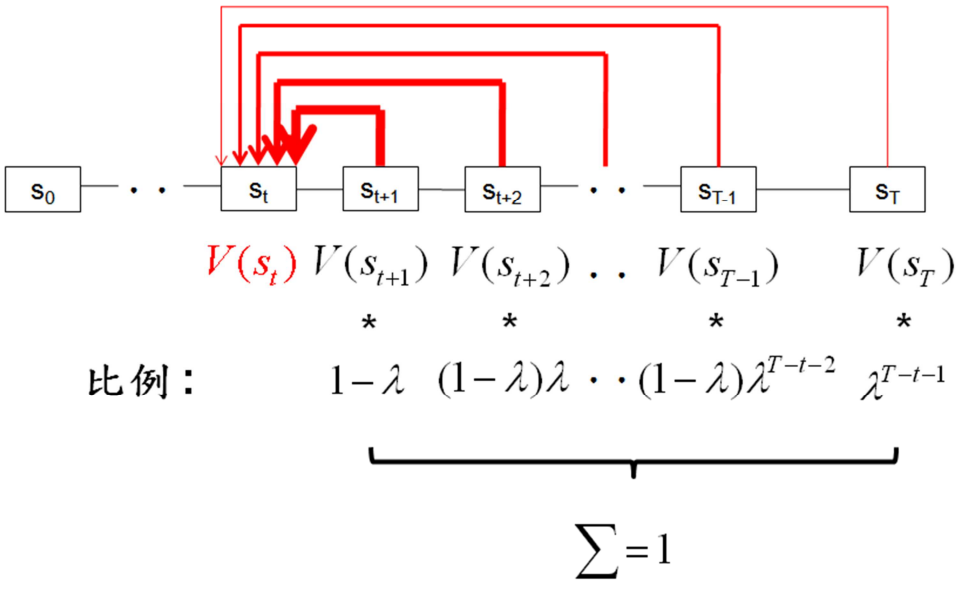


圖 10. TD(λ)更新示意圖

最後將 $R_t^{(1)}$、…、$R_t^{(n)}$、…、R_t 乘上各自的比例總和起來得到 R_t^{λ}：

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t \tag{7}$$

得到 R_t^{λ} 之後就可以拿來調整 $V(s_t)$，計算出修正量$\Delta V(s_t)$，α 是一個參數介於 0 到 1 之間，越高代表每次的修正量越多。

$$\Delta V(s_t) = \alpha[R_t^{\lambda} - V(s_t)] \tag{8}$$

而調整就只是很簡單的加上要修正的量

$$V(s_t) = V(s_t) + \Delta V(s_t) \tag{9}$$

這就是整個 TD(λ)的概念。

然而實際在許多學習的應用上，尤其是棋類遊戲，常常到終點狀態才得到回饋 r_T，中間得到的回饋都是 0，因此我們可以將公式做一些推導使其簡單化。例如公式(5)可以把中間得到的回饋都去掉得到公式(10)

$$R_t^{(n)} = V(s_{t+n}) \tag{10}$$

再額外定義 $V(s_T)$

$$V(s_T) = r_T \tag{11}$$

便可以把公式(6)簡化成公式(12)

$$R_t = V(s_T) \tag{12}$$

這個一來根據公式(10)和(12)就能把公式(7)簡化如下

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} V(s_{t+n}) + \lambda^{T-t-1} V(s_T) \tag{13}$$

接下來提到的時序差異學習都是指適用於遊戲的簡化更新為主。

而時序差異學習不只可以調整狀態分數，有一種變形稱為線性時序差異學習是用來調整特徵分數，進而影響狀態分數，其更新公式如下

$$\Delta\theta = \alpha[R_t^{\lambda} - V(s_t)]\varphi(s_t) \quad (14)$$

$$\theta = \theta + \Delta\theta \quad (15)$$

公式(14)只是將原本的公式(8)修改，將原本要調整的 s_t 分數，改成看 s_t 有哪些特徵，就調整那些特徵的分數。

2.4.5 TD(0)和 TD(1)

在 TD(λ)中有兩個極端的情況，一個是 TD(0)，也就是 λ=0 時，只取 $R_t^{(1)}$來調整 $V(s_t)$，公式和圖例如下：

$$\Delta V(s_t) = \alpha[V(s_{t+1}) - V(s_t)] \quad (16)$$

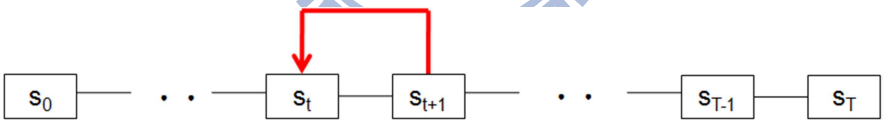


圖 11. TD(0)示意圖

另一個則是 TD(1)，也就是 λ=1 時，只取 R_t 來調整 $V(s_t)$，公式和圖例如下：

$$\Delta V(s_t) = \alpha[V(s_T) - V(s_t)] \quad (17)$$

圖 12. TD(1)示意圖

TD(1)實際上就和 monte-carlo 一樣，取最後的結果拿來更新，優點是更新的速度快，缺點是較不準。而 TD(0)則相反，優點是比較準，但更新的速度比較慢。

第三章、研究方法

雖然 2.4 節介紹了時序差異學習(Temporal Difference Learning)[3][14]是怎麼做學習,可是對一般學習者,可能還是太難了解,因此 3.1 節介紹了一些 TD(0)的簡單例子,讓讀者可以快速了解時序差異學習。3.2 節介紹實作時序差異學習演算法我們所用的虛擬程式碼,3.3 節則介紹一些實作上的議題。

3.1 簡單的時序差異學習

首先先設計一個簡單的遊戲,擲硬幣,出現正面得 100 分,反面則得 0 分。我們可以根據這個遊戲畫出樹狀圖如下,A 代表還沒擲之前的狀態。如果這是一個公正硬幣,則出現正面和反面的機率都相同,皆為 50%,則我們可以算出期望值 $V^{\pi}(A)$ = 50%*100+50%*0 = 50(分)。然而這是因為我們知道機率的情況下才可以計算,如果不知道機率的話就無法算出期望值。在一般的棋類遊戲等通常不太容易知道每一步確切的機率,這時候我們就可以用時序差異學習來學習 $V(A)$使其接近 $V^{\pi}(A)$,一開始先初始化 $V(A)$為 0,再慢慢根據結果調整期望值。

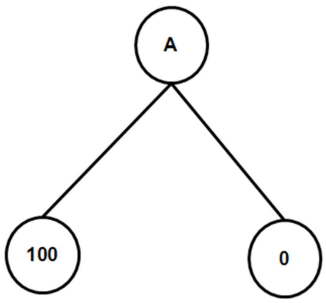


圖 13. 擲硬幣樹狀圖

首先假設第一擲擲出正面，我們設定 α 固定 0.01，則根據 TD(0)的更新公式(16)算出$\Delta V(A) = \alpha[100-V(A)] = 0.01[100-0] = 1$，更新完 $V(A) = V(A)+\Delta V(A) = 0+1 = 1$，因此第一輪結束後 $V(A)$更新成 1。

再來假設第二擲值出反面，則$\Delta V(A) = \alpha[0-V(A)] = 0.01[0-1] = -0.01$，更新完 $V(A) = V(A)+\Delta V(A) = 1+(-0.01) = 0.99$，因此第二輪結束後 $V(A)$更新成 0.99。

可以看到做了兩次的更新之後，$V(A)$從原本的 0，改變成 0.99，下圖是作者寫了一個小程式下去跑，跑了 1000 輪之後統計的結果。可以看出大約在 300 輪以後，$V(A)$的值就在 50 上下浮動，確實逼近了我們要得到的理想值 $V^{\pi}(A) = 50$。從這個例子我們可以看出在不斷地調整更新之後，狀態分數會學習趨近於其期望值。

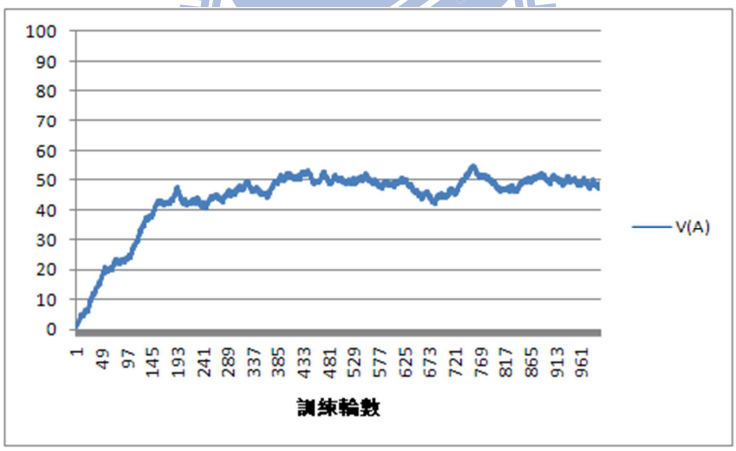


圖 14. 擲硬幣訓練結果

再舉第二個例子，樹狀圖如下圖 15，若採用均勻隨機策略(uniform random policy)，則 $V^{\pi}(A) = 50$，$V^{\pi}(B) = 200/3$，$V^{\pi}(C) = 100/3$。也是一樣一開始初始化 $V(A) = 0$，$V(B) = 0$，$V(C) = 0$。

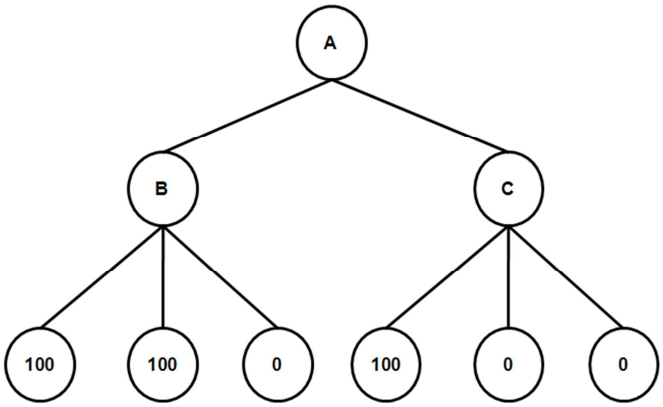


圖 15. 第二個例子樹狀圖

假設第一輪選到最左邊這條路，因為走了兩步到終點，所以我們可以做兩次更新，第一次更新上面的 $V(A)$，算出$\Delta V(A) = \alpha[V(B)-V(A)] = 0.01[0-0] = 0$，更新完 $V(A) = V(A)+\Delta V(A) = 0+0 = 0$。第二次更新下面的 $V(B)$，算出$\Delta V(B) = \alpha[100-V(B)] = 0.01[100-0] = 1$，更新完 $V(B) = V(B)+\Delta V(B) = 0+1 = 1$。因此第一輪更新完 $V(A)$、$V(B)$、$V(C)$分別是 0、1、0。

假設第二輪選到左邊數來第三條路，那麼也是可以做兩次更新，第一次更新上面的 $V(A)$，算出$\Delta V(A) = \alpha[V(B)-V(A)] = 0.01[1-0] = 0.01$，更新完 $V(A) = V(A)+\Delta V(A) = 0+0.01 = 0.01$，第二次更新下面的 $V(B)$，算出$\Delta V(B) = \alpha[0-V(B)] = 0.01[0-1] = -0.01$，更新完 $V(B) = V(B)+\Delta V(B) = 1+(-0.01) = 0.99$。因此第二輪更新完 $V(A)$、$V(B)$、$V(C)$分別是 0.01、0.99、0。

同樣也是寫一個小程式下去跑，跑了 1000 輪之後統計的結果如下圖，可以看到 ABC 最後分數都接近他們的期望值。而這邊也可以看到 $V(A)$的更新速度會比較慢一點，這是因為他要等其 children B 和 C 的分數更新到一定的程度，$V(A)$才會更新得比較合理，因此越接近 root node，更新會越慢。

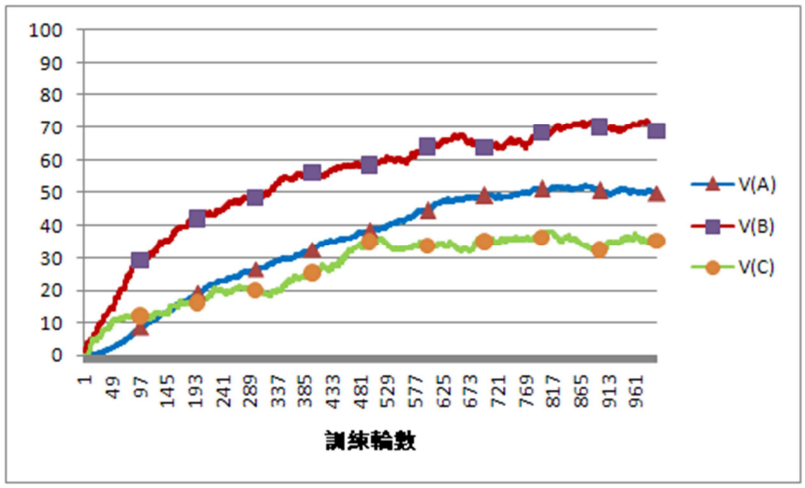


圖 16. 第二個例子訓練結果

再來是將前面的例子，改用線性時序差異學習方式來調整特徵分數，假設有三種特徵，ABC 的特徵數量分別為 $\varphi(A)=[2,1,1]$，$\varphi(B)=[1,2,1]$，$\varphi(C)=[1,1,2]$。一開始初始化特徵分數全為 0，$\theta=[0,0,0]$。

假設第一輪選到最左邊這條路，那我們同樣可以做兩次更新，第一次更新上面，算出$\Delta\theta=\alpha[V(B)-V(A)]\varphi(A)=0.01[0-0][2,1,1]=[0,0,0]$，更新完 $\theta=\theta+\Delta\theta=[0,0,0]+[0,0,0]=[0,0,0]$。第二次更新下面，算出 $\Delta\theta=\alpha[100-V(B)]\varphi(B)=0.01[100-0][1,2,1]=[1,2,1]$，更新完 $\theta=\theta+\Delta\theta=[0,0,0]+[1,2,1]=[1,2,1]$。因此第一輪更新完之後 $V(A)=\varphi(A)\cdot\theta=[2,1,1]\cdot[1,2,1]=2+2+1=5$，$V(B)=\varphi(B)\cdot\theta=[1,2,1]\cdot[1,2,1]=1+4+1=6$，$V(C)=\varphi(C)\cdot\theta=[1,1,2]\cdot[1,2,1]=1+2+2=5$。

同樣也是寫一個小程式下去跑，跑了 1000 輪之後統計的結果如圖 17，可以看到跟圖 16 相比，A 的分數明顯更新比較快，這是因為特徵分數是共用的原因，使得每次的更新都會影響到所有節點。

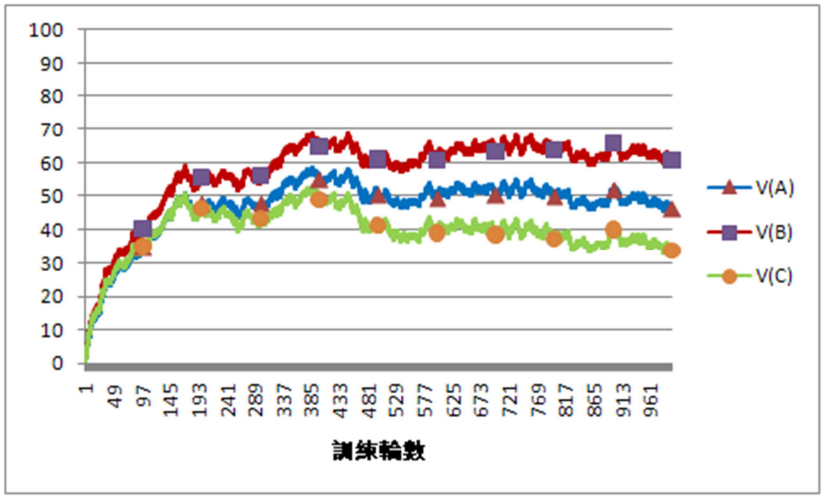


圖 17. 第三個例子訓練結果

3.2 虛擬程式碼

此篇研究在 NCTU6 上實作了 TD(0)演算法來學習調整特徵分數，這一節會來介紹我們實作時所用的虛擬程式碼，主要是參考自 Silver 論文[3]中的 Algorithm 3。接下來會一一介紹重要的函式和他的虛擬程式碼。

3.2.1 TD Learning

Algorithm 1 TD_Learning
Input: n (the number of iterations) **Procedure**: 1: i = 0 2: **while** i < n **do** 3: board.Initialise() 4: SelfPlay(board) 5: i++ 6: **end while**

TD_Learning，是最一開始進入的函式，TD_Learning 的虛擬程式碼列

在 Algorithm 1。此函式一直不斷重複同一個迴圈，在迴圈之內會執行 Initialise 函式，將盤面回到初始盤面。之後執行 SelfPlay 函式，會從初始盤面開始，選一條路直到結束，途中會調整特徵分數。到底之後重複迴圈，如此就能不斷重複下去，持續調整特徵分數。我們設定 n 為訓練盤數，當迴圈重複 n 次之後代表學習結束。

3.2.2 SelfPlay

Algorithm 2 SelfPlay

Input: board (board position)

Procedure:

```
1:   t = 0
2:   V0,φ0 = Eval(board)
3:   while not board.Terminal() do
4:      at = Greedy(board,ε)
5:      board.Play(at)
6:      t++
7:      Vt,φt = Eval(board)
8:      if t ≥ 1 then
9:         δ= Vt - Vt-1
10:        for all i ∈ φt-1 do
11:           θ[i] += αδφt-1[i]
12:        end for
13:     end if
14: end while
```

SelfPlay 會從輸入的盤面開始，選一條路直到結束，並根據所經過的這條路來調整特徵分數。可以看到 4~5 行會呼叫 Greedy 函式，來選擇接下來要下哪一步，並且下下去。而在經過的路上，每個盤面都會呼叫 Eval 函式來取得該盤面的分數與特徵數量。9~12 行是整個流程最重要的部分，也就是特徵分數的更新，會根據特徵數量和盤面分數差值來調整特徵分數。

整個迴圈不斷重複下去，直到盤面到達終點為止，便完成一盤訓練。

3.2.3 Greedy

Algorithm 3 Greedy

Input: board (board position), ε (epsilon)

Procedure:

```
1:  if Bernoulli(ε) = 1 then
2:      return Uniform(board)
3:  end if
4:  black = board.BlackToPlay()
5:  a* = Pass
6:  V* = black ? V_min : V_Max
7:  for all a ∈ board.Legal() do
8:      board.Play(a)
9:      V = Eval(board)
10:     if (black and V ≥ V*)
11:     or (not black and V ≤ V*) then
12:         V* = V
13:         a* = a
14:     end if
15:     board.Undo()
16: end for
17: return a*
```

Greedy，會根據目前盤面，來選擇一步要下的步，而選點的策略為貪婪策略(greedy policy)。因此一開始有 ε 的機率隨機選一步下，就會執行 Uniform 函式。如果沒有要隨機選的話，就會去找最好的一步。而要找到最好的那步，就必須每步都下下去之後，呼叫 Eval 函式計算盤面分數，就可以知道哪一步的分數是最好的，找到之後我們就可以回傳最好的那一步。

3.2.4 Eval

Algorithm 4 Eval

Input: board (board position)

Procedure:

```
1:  if board.Terminal() then
2:      return board.BlackWins();
3:  end if
4:  φ = board.GetActive()
5:  V = 0
6:  for all i ∈ φ do
7:      V += φ[i]θ[i]
8:  end for
9:  return V,φ
```

Eval，會根據輸入的盤面，來計算該盤面的分數和特徵數量。第 1~2 行先判斷盤面是否已經分出勝負，如果已經結束，便直接回傳分出勝負應有的分數。若還未分出勝負，則在第 4 行會去取得盤面的特徵數量，並且在第 8 行計算加總，得到盤面分數，最後會回傳盤面分數和特徵數量。

3.3 實作議題

實作上，有許多機制我們可以選擇是否要使用，這一節會來介紹實作時，有哪些可以拿來討論的重要議題。分別介紹特徵的選擇、特徵標準化、迫著空間搜尋(Threat Space Search)[9][10]以及讀高手棋譜。

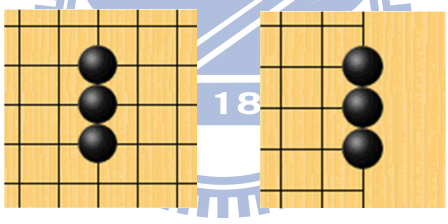
3.3.1 特徵的選擇

要選什麼來當作特徵是一個大問題，在六子棋中，最主要可以拿來當

特徵的就是棋型，例如活一、死二等。為什麼棋型很重要呢？舉例來說假如我方有一個活二，就代表我方可以下兩顆子形成雙迫著，這將會非常具有攻擊力，有一個死三的話，也可以下一顆子形成單迫著，這些棋型往往都能延伸出去形成攻擊的手段。

其餘也有一些可以拿來輔助的特徵，像是黑白兩方棋型的數量多寡，通常有用的棋型越多代表越有利，有利的那方可以多給一些額外分數，這個分數也可以拿來當作特徵分數訓練。

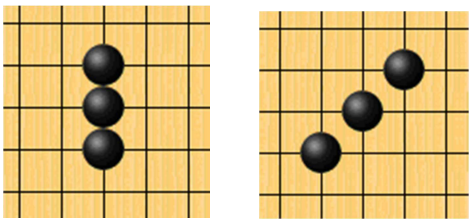
棋型離中心的距離也可以拿來當作特徵，通常越接近邊界是越不利的，因為發展容易受到限制，如圖 18.a，從橫向的角度來看的話是三個活一，而 18.b 則是三個死一，因此就算是同樣的棋型，距離中心越遠的分數理論上要越低。



(a)　　(b)

圖 18. (a)離中心較近 (b)離中心較遠

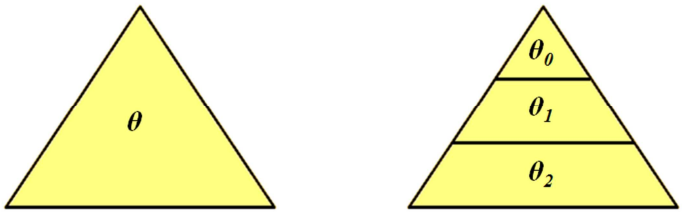
棋型的方向亦可當作特徵，通常斜的會比直的或橫的更具有攻擊力，因為延伸出去的距離較長，使涵蓋的攻擊範圍也會更著變大，如圖 19。



(a)　　(b)

圖 19. (a)直的棋型 (b)斜的棋型

是否要區分開局、中局、殘局也是特徵選擇的重要決定之一，每個階段注重的棋型都不盡相同，例如在六子棋，迫著常常可以拿來制衡活二，因為只要我方有一個迫著，對方就必須花一顆子來擋，便沒有辦法將活二下兩顆子形成 Double-Threat，可是在開局階段常常迫著數量不多，因此活二就比較重要，下到後期迫著比較容易出現，活二的重要性就會稍微降下來，因此區分不同的階段來獨立訓練理論上會比較好，如圖 20。



(a)　　(b)

圖 20. (a)不區分階段 (b)區分成開局、中局、殘局

最後結合以上這些，我們總共有 492 個特徵。

3.3.2 特徵標準化

原本的公式(14)，更新是根據特徵的數量為主，這麼一來可能會有一些問題，例如有一個盤面有九個活一，一個活二，沒有活三，則更新的特徵分數比例就是 9:1:0，導致活一更新的比例太高。在其他應用時，這個作法可能沒有問題，因為某個特徵常出現就代表這個特徵可能比較重要，但是在六子棋裡面，常常在遠一點的地方下一顆子，就產生好幾個活一，但活一明顯重要程度比其他棋型如活二等來的低。因此我們設計另一個更新公式如下：

$$\omega_i(s)=\begin{cases}1 & if\ \varphi_i(s)>0\\ 0 & if\ \varphi_i(s)=0\\ -1 & if\ \varphi_i(s)<0\end{cases} \tag{18}$$

$$\Delta\theta=\alpha[R_t^{\lambda}-V(s_t)]\omega(s_t) \tag{19}$$

這麼一來剛才的例子更新比例就會從 9:1:0 改成 1:1:0。

3.3.3 迫著空間搜尋

在訓練時，是否要使用迫著空間搜尋，也是重要議題之一，用原本的方法訓練時，我們發現一個奇怪的現象，就是迫著的分數會過高。這是因為在選擇一條路到結束時，常常最後一小段都是在走雙迫著追殺的必勝法形成的必勝路徑，如下圖。這會使得訓練時誤認為雙迫著很重要，因此這裡就需要使用迫著空間搜尋來做改善。

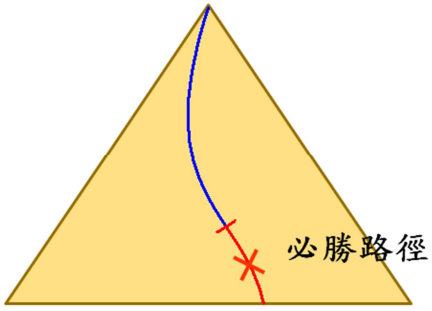


圖 21. 去掉必勝路徑示意圖

我們主要使用迫著空間搜尋在兩個地方，一個是程式選點時避開必敗步，另一個是如果有必勝步，就直接結束這一輪訓練。這麼一來就能提升程式選點時的準確度，也能去掉不必要的更新，不過缺點是因為要作迫著空間搜尋的關係，也會使訓練速度變慢。

3.3.4 讀高手棋譜訓練

原本的訓練程式,可以換個角度,想像成是程式根據策略 π 自己選點，再把最後選的路輸入到程式訓練。而這條路我們可以用高手曾經下過的棋譜來取代，也就能將策略 π 替換成這些高手的想法。當然讀棋譜時也會有前一小節的問題，因此也要使用迫著空間搜尋來去掉不必要的更新。

我們使用的高手棋譜，是從 Little Golem[11]網站上面，找出積分高於 1800 以上的高手們所下的棋譜集合而成。由於 Little Golem 網站上的對局思考時間非常長，每個棋士一場有 240 小時，每次下完又會補充 36 小時，因此思考時間相當充裕，較不會因時間不夠而粗心下錯，使得棋譜品質更好，我們總共蒐集了三萬多盤棋譜。

讀高手棋譜來訓練，有個好處是比程式自己選點來的快，因為原本程式自己選點時要作迫著空間搜尋來避開必敗步，而讀高手棋譜則可以想像成是高手已經幫忙避開必敗步，因此能夠省下不少時間，不過缺點可能是棋譜數量有限，無法像程式一樣不斷訓練下去。

第四章、實驗

這章會來實驗比較 3.3 當中各種方法是否使用的結果，首先 4.1 先補充實作時的一些細節，4.2 再來介紹實驗環境，接著幾節列出各種實驗結果。

4.1 實作細節

此篇研究在 NCTU6 上實作了 TD(0)演算法來學習調整特徵分數，基本架構如 3.2 的虛擬程式碼所示，而這一節會來講解一些在實作時的一些細節與設定。

4.1.1 更新比例

α是一個參數，用來調整更新的比例，越高代表每次的更新幅度越多。理論上α應該要由大變小，這麼一來才能讓一開始參數分數還不準的時候可以很快找到一個大略的值，到後面才去做微調的動作。然而在實作上，有時為了要取一個適當的函數來找合適的 α，這些計算可能反而會影響程式效率，因此我們最後是選取固定的α值 0.1。

4.1.2 選點策略

我們使用的選點策略，是採取 2.4.1 小節提到的貪婪策略(greedy policy)。因為跟均勻隨機策略(uniform random policy)相比，貪婪策略比較接近實際玩家的策略 Mini-Max Search，這麼一來得到的期望值也會和正式對戰時比較相符。同時我們又希望選點時不要太過於集中，因此選

點時還是有一定的機率隨機去選，確保每個節點都要有機率去選到，最後我們選擇貪婪策略中的 ε 值為 0.1。

4.1.3 更新標準化

我們希望每次更新時的比例是合理的，然而原本的更新公式(14)會有一個問題，如果特徵總數越多，則更新的總量也會越大。因此我們可以將特徵數量除以特徵總數 k，修改之後得到公式(20)，這麼一來就能改善上述問題。

$$\Delta\theta = \alpha[R_t^{\lambda} - V(s_t)]\frac{\varphi(s_t)}{\|\varphi(s_t)\|^2} \tag{20}$$

在虛擬程式碼部分，也只要修改一個地方即可，只要將 Algorithm 2 中的第 11 行，多除以特徵總數即可。

4.1.4 分數標準化

原本的狀態分數是由特徵數量和特徵分數線性組合而成，我們可以把狀態分數標準化到 0 至 1 之間。

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{21}$$

$$V(s) = \sigma(\varphi(s)\cdot\theta) \tag{22}$$

這麼一來有個好處，下圖是 $\sigma(x)$的函數圖，可以看到在 x = 0 的時候斜率最高，這可以反映在棋類遊戲上。例如一個盤面是優勢，另一個是非常優勢，這兩個在選擇時其實差異不大，因為通常下到最後的結果都是贏。但是如果一個盤面是優勢，另一個是劣勢，這兩個盤面來選擇時，一旦選錯

可能就是輸贏之差。而分數標準化就能把這個現象顯現出來。

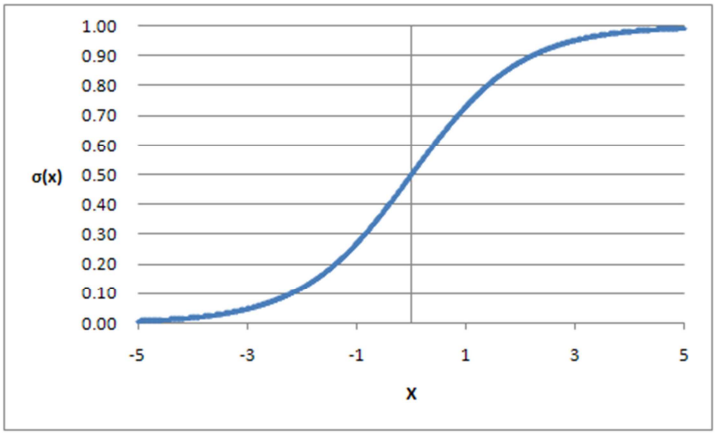


圖 22. σ(x)函數圖

在虛擬程式碼部分，也只要修改一個地方即可，只要將 Algorithm 4 中，在回傳結果之前，將分數經過公式(21)的轉換即可。

4.1.5 回饋(Reward)

回饋的設定其實大同小異，不過為了配合上一小節提到的分數標準化，我們必須設定贏得到的回饋為+1，輸得到的回饋為+0，平手得到的回饋為+0.5。$V^{\pi}(s)$是指期望獲得的回饋，照上面的方式設定回饋才可以把 $V^{\pi}(s)$也標準化到 0 至 1 之間，也才能讓 $V(s)$去學習逼近 $V^{\pi}(s)$。

4.1.6 兩層更新

原本 TD(0)的更新公式(16)都是拿下一個狀態的分數減掉目前狀態的分數，這個我們稱為一層更新，我們可以改變一下變成兩層更新，如以下公式和圖

$$\Delta V(s_t) = \alpha[V(s_{t+2}) - V(s_t)] \qquad (23)$$

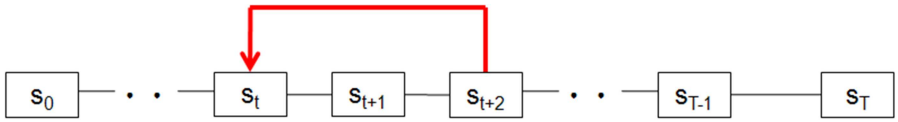


圖 23. 兩層更新示意圖

由於實際上的棋局是兩人互下，因此假如是採用一層更新，會導致算出來的調整量過大。舉六子棋來說，目前的盤面是黑方剛下完，黑方剛下完時，盤面上的黑子數量比白子多一顆，因此評估函數算出來的分數平均來說應該會對黑稍微有利，而下一個盤面則是白方剛下完，則白方剛下完時，盤面上的白子數量比黑子多一顆，因此評估函數算出來的分數應該會對白稍微有利。如果拿白有利的分數減掉黑有利的分數，自然會讓調整量過大。而改成兩層更新的話，就能讓兩個黑有利的盤面相減，算出來的調整量就會比較合理。

在虛擬程式碼部分，則將 Algorithm 2 中第 8~13 行，原本的一層更新改成兩層更新即可。

4.2 實驗環境

我們使用實際對戰的方式來檢驗學習的成果，讓兩個程式來對戰，一個是 NCTU6 使用原本手工設置的特徵分數，另一個是 NCTU6 使用學習得到的特徵分數，兩邊差別只有在特徵分數不同而已，因此可以比較哪一個特徵分數較為準確。

我們挑了 176 個盤面，每個盤面比兩場，比兩場的原因是因為要交換先手。這 176 個盤面的挑選方式是整理 Little Golem[11]網站上，積分高於 1800 的高手們所下的所有棋譜，當中出現超過 30 次以上的盤面。因為常下到的盤面，通常代表這個盤面是均勢，雙方才會想去下，不過也有例

外，因此把一些用 NCTU6 可以簡單確定必勝或必敗的盤面去掉，剩下的就是這 176 個盤面。

我們設定每場遊戲贏得 2 分，輸得 0 分，平手得 1 分，而勝率則為得分除以總分。

由於兩個程式每比一次就有三百多場遊戲要比，這在一台電腦上就需要好幾天才能跑完，因此在實驗時，我們使用了實驗室所開發的桌機格網系統，可以將許多場遊戲平行下去進行，藉此來加快實驗的進行。

4.3 區分階段

這一節來討論有無區分階段的實驗結果，結果如下圖，橫軸代表訓練的盤數，縱軸代表使用訓練完的特徵分數，對上原本使用手工設置的特徵分數時的勝率。

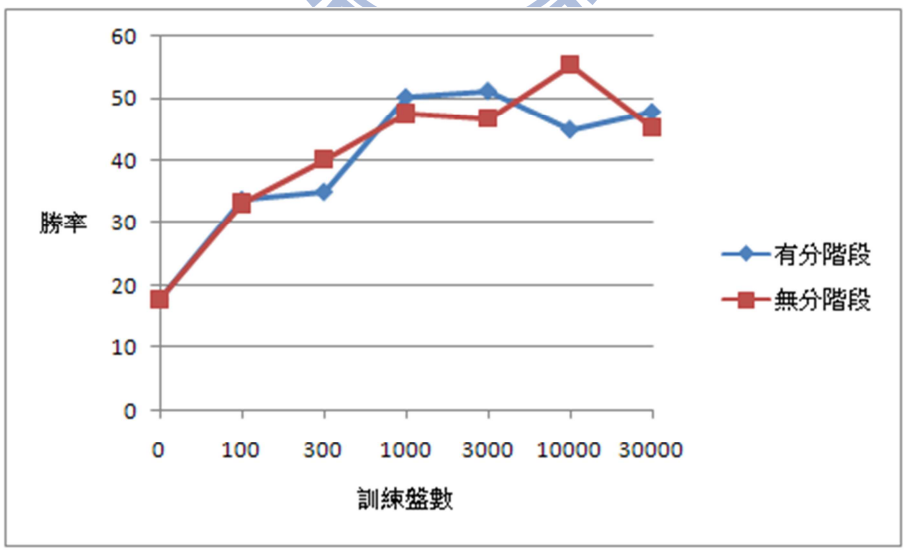


圖 24. 區分階段實驗結果

理想情況應該是有區分階段的前期勝率較低，後期勝率較高。但是從

實驗結果似乎沒有得到預期般的數據，因此經過我們繼續深入分析後，發現區分階段會有一個嚴重的問題，就是開局訓練出來的特徵分數不夠準，這可能是由於開局比較接近root，所以訓練會比較慢，儘管訓練了三萬盤，開局的特徵分數仍舊不夠理想。

因此我們使用了一個方法，就是把訓練出來的開局特徵分數用原本手工設置的特徵分數來取代，這麼一來就能解決開局不準的問題，除了開局以外全部使用訓練出來的特徵分數，我們稱這種方法為混合，在混合方法中，同樣也考慮是否要區分中局和殘局。

使用這個方法之後的勝率如下圖，可以看到使用混合的兩種方法，和沒有使用混合的兩種方法相比，勝率絕大部分都較高，明顯比較好。而且最高點在於訓練三千盤的勝率，高達 57.81%，超過 50%，代表經過訓練後的特徵分數，比原本手工設置的特徵分數還要好。並且也可以看到最高點的勝率，比開始訓練前的勝率高了 33.38%，可看出訓練前後棋力有相當明顯的成長。

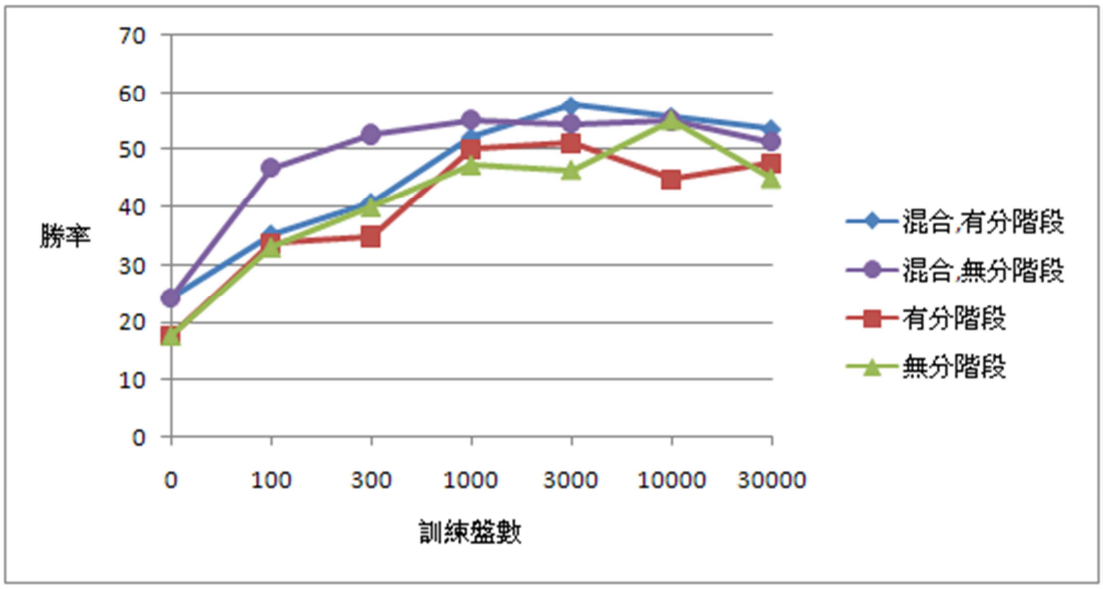


圖 25. 加入開局使用手工特徵分數實驗結果

而同樣是混合，有分階段和無分階段相比，前期勝率較低，而後期勝率較高。這也相當符合預期，因為區分階段會讓特徵數量變多，訓練起來就會比較久，因此前期勝率爬升較慢，不過好處是能學得更精細，因此到後來勝率會超過無分階段的勝率。

4.4 特徵標準化

這一節來比較有無使用特徵標準化的實驗結果。從下圖可以看到有無使用特徵標準化，並沒有太大的差別。有使用特徵標準化的勝率最高點，在於經過三千盤訓練的 57.81%，無使用特徵標準化的勝率最高點，在於經過三萬盤訓練的 57.95%。反而是無使用特徵標準化的略好 0.14%，因此在六子棋的學習中，使用特徵標準化並沒有預期般的效果，不過或許在其他應用上可以當作討論的議題之一。

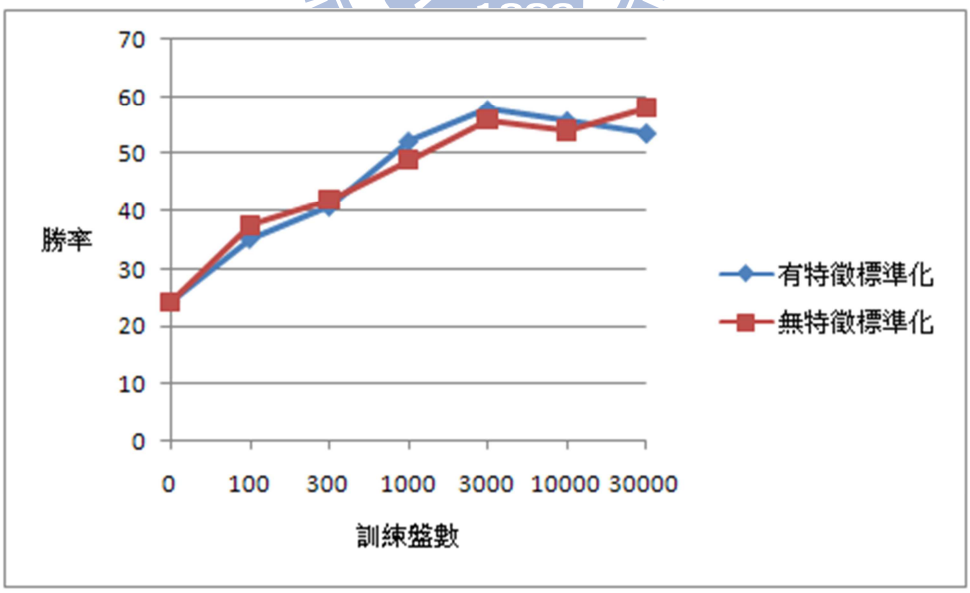


圖 26. 特徵標準化實驗結果

4.5 迫著空間搜尋

這一節來比較有無使用迫著空間搜尋的實驗結果，如下圖，可以看到有使用迫著空間搜尋的勝率，明顯比沒有使用迫著空間搜尋來的好很多。勝率差最多的地方，在於同樣訓練一千盤時，兩邊勝率差了 20.17%。因此可以看出六子棋在學習時使用迫著空間搜尋是相當重要的方法之一。

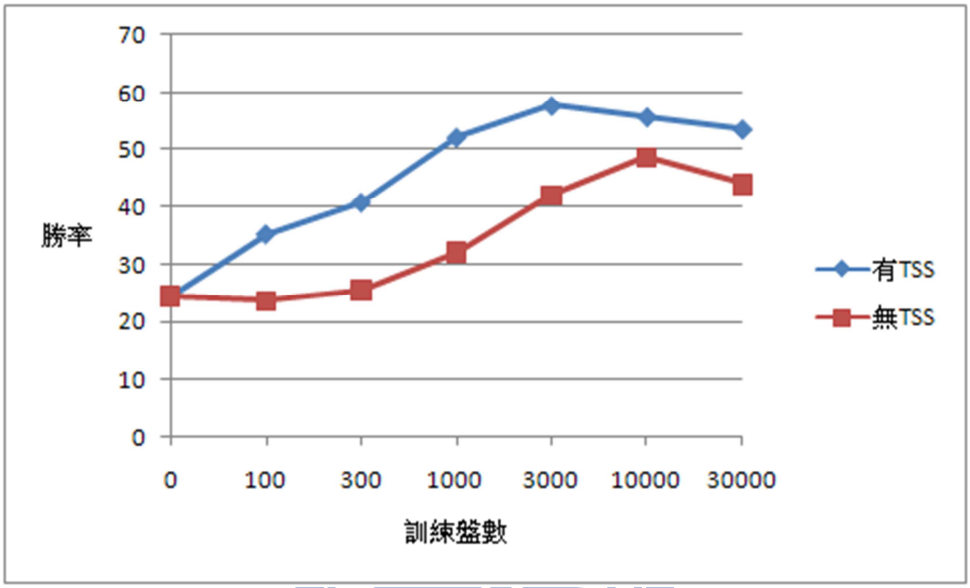


圖 27. 迫著空間搜尋實驗結果

另一方面，我們也觀察了訓練出來的特徵分數，發現沒有使用迫著空間搜尋的雙迫著分數過高，這應該就是導致勝率較低的原因。而使用迫著空間搜尋訓練出來的雙迫著分數，則只有未使用的三分之一不到，這也較符合一般棋士的想法，不會下沒有用的雙迫著。部分特徵分數列在下表。

特徵分數	有迫著空間搜尋	無迫著空間搜尋
雙迫著	0.52982	1.73220
單迫著	0.51070	0.83796
活三	0.49358	0.73046
死三	0.27506	0.25531
活二	0.20028	0.07715

表 3. 特徵分數比較表

4.6 讀高手棋譜訓練

這一章節來比較程式選點，和使用高手棋譜的勝率差別。如下圖，可以看到若是使用高手棋譜,但沒有用迫著空間搜尋時,勝率一樣普遍很低，而使用高手棋譜並且有用迫著空間搜尋的話，勝率雖然沒有比程式選點來的高，不過在訓練三萬盤時也高達 55.26%，仍舊有不錯的效果。因此就算是讀高手棋譜訓練，還是必須配合迫著空間搜尋把必勝路徑去除，這樣訓練出來的效果才會比較好。

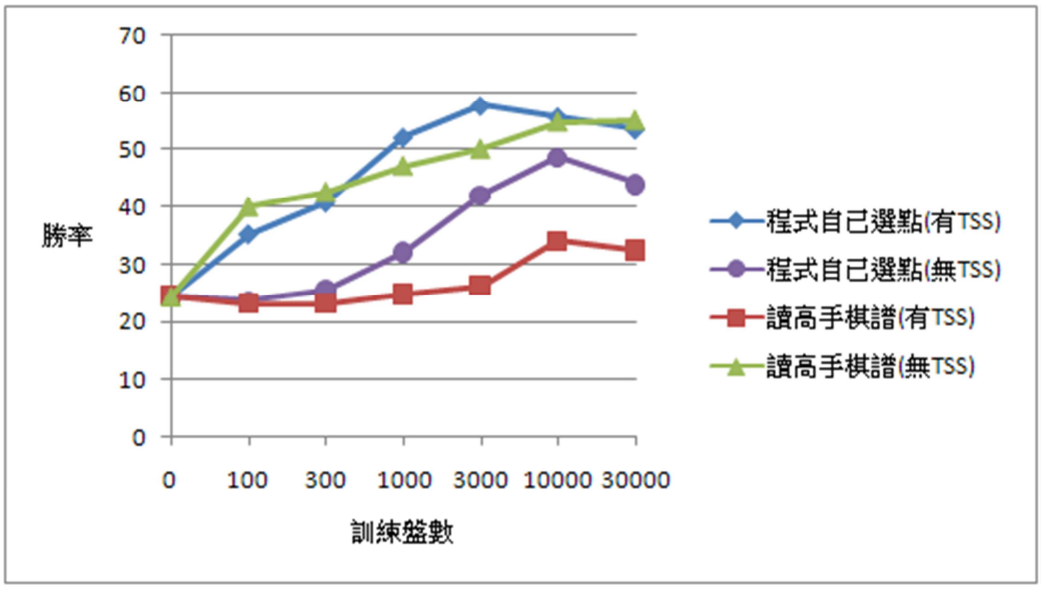


圖 28. 讀高手棋譜實驗結果

不過我們也可以從另一個角度來分析，也就是時間。可以看到下表，若是程式自己選點的話，有用迫著空間搜尋，一萬盤訓練需要花 11 個小時多，而讀高手棋譜，也有使用迫著空間搜尋，只需要約三十分鐘，時間大約程式選點的二十分之一。因此使用高手棋譜來學習可以說是快而且也算準的一個方法。

方法	訓練一萬盤所需時間
程式自己選點(有 TSS)	11 小時 17 分鐘
程式自己選點(無 TSS)	31 分鐘
讀高手棋譜(有 TSS)	32 分鐘
讀高手棋譜(無 TSS)	2 分鐘

表 4. 時間比較表

第五章、結論與未來展望

本篇論文中，實作了時序差異學習(Temporal Difference Learning)演算法在 NCTU6 上，從實驗的結果我們可以看到學習之後的 NCTU6 的確比原本的 NCTU6 還要強，勝率達到 57.95%，成功改良了特徵分數的設定。我們也將使用時序差異學習演算法的 NCTU6 以 TD6 的名稱參加了 2011 TCGA 電腦對局競賽六子棋組，並且得到了金牌的好成績。

同時我們也對以下這些議題作了分析：

- 區分階段:單純區分階段並沒有太大改善,因為開局的訓練效果不彰，不過這點可以用原本手工設定的參數來解決這個問題。
- 特徵標準化：雖然在本篇論文中的實驗結果並不明顯，不過或許在別的應用上可以考慮當作議題之一。
- 迫著空間搜尋(Threat Space Search):在六子棋中是相當重要的方法，雖然會使學習的速度變慢，但訓練完的結果會比沒有使用來的好很多。
- 讀高手棋譜訓練：訓練出來的特徵分數結果也不錯，重點在於使用高使棋譜訓練能夠使訓練所花的時間減少很多。

最後在未來展望方面,希望能找到適合的方法來學習開局的參數分數，使程式棋力更上一層樓。以及可能可以試著將時序差異學習套用到其他遊戲，如象棋等，幫助產生更強的程式。

參考文獻

[1] Connect6-六子棋網站. Available at http://www.connect6.org/

[2] D.E. Knuth and R.W. Moore, “An analysis of alpha-beta pruning,“ Artificial Intelligence, 6:293–326, 1975.

[3] D. Silver, “Reinforcement learning and simulation-based search in computer Go,” Ph.D. dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2009.

[4] H.J. van den Herik, J.W.H.M. Uiterwijk and J.V. Rijswijck, “Games solved: Now and in the future,” Artificial Intelligence, vol. 134 (1-2), pp. 277–311, 2002.

[5] I.-C. Wu and D.-Y. Huang, “A New Family of K-in-a-row Games,” Advances in Computer Games Conference (ACG2005), Taipei, Taiwan, 2005.

[6] I.-C. Wu and P.-H. Lin, “Relevance-Zone-Oriented proof search for Connect6,” IEEE Trans. Comput. Intell. AI Games, vol. 2, no. 3, Sep. 2010.

[7] I.-C. Wu, D.-Y. Huang and H.-C. Chang, “Connect6,” ICGA Journal, vol. 28(4), pp. 234–242, 2006.

[8] J.R. Slagle and J.K. Dixon, “Experiments with some programs that search game trees,” JACM 16, 2 189-207, 1969.

[9] L.V. Allis, H.J. van den Herik and M.P. H. Huntjens, “Go-Moku Solved by New Search Techniques,” Computational Intelligence, Vol. 12, pp. 7–23, 1996.

[10] L.V. Allis, “Searching for solutions in games and artificial intelligence,” Ph.D. Thesis, University of Limburg, Maastricht, 1994.

[11] Little Golem website. Available at http://www.littlegolem.net/

[12] Ludoteka website. Available at http://www.ludoteka.com/

[13] Pente website. Available at http://pente.org/

[14] R.S. Sutton and A.G. Barto, “Reinforcement Learning: An Introduction,” MIT Press, Cambridge, MA, 1998.

[15] T. Thomsen, ”Lambda-search in game trees - with application to Go,” ICGA Journal, Vol. 23 203–217, 2000.