Application of Temporal Difference Learning and Supervised Learning in the Game of Go

By

Horace Wai-kit, Chan

Supervised By :

Prof. Irwin King and Prof. Chi-sing Lui

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the
degree of Master of Philosophy
at the Chinese University of Hong Kong
July 1996

Acknowledgement

I would like to express my gratitude to my supervisors, Prof. Irwin King and Prof. Chi-sing Lui for their advice and guidance throughout this research. I also want to thank Prof. Lai-wan Chan who introduced me to the area of neural networks and gave me comments in the early stage of this research. Moreover, I need to thank Prof. Man-hon Wong, Prof. Jimmy Lee and Prof. Tony Marsland for reading, commenting and grading this thesis.

I appreciate very much the help from two Go servers, Internet Go Server (IGS) and No Name Go Server (NNGS), which provide abundant records of Go games. Without such resources, many experiments cannot be carried out. I have special thanks for Mr. Erik Van Riper, the system developer and administrator of NNGS, for his interest in my research and his generosity.

I am indebted to Mr. Pui-fai Sum, Mr. Kei-siu Ho, Mr. Chi-sing Leung and Mr. Wing-kei Yip for their kind discussions with me when I faced difficulties.

I want to give my special thanks to Miss Lam Laam for her endless support and care during the hard time. Thanks to my parents who have been doing a lot for me so that I can concentrate on the work presented in this thesis.

Abstract

In this thesis, we try to analyze the feasibility of applying neural networks, especially the temporal difference (TD) learning model, in the game of Go. First, a framework of decomposition of computer Go into smaller problems is given. Previous applications of neural networks and TD learning in Backgammon and Go are then given.

We derive a new updating rule for applying TD(λ) learning in multi-layer Perceptron and use the rule in performing several experiments. From the first experiment, we obtain a relationship between λ and the performance of the evaluation function after training. When trying to use TD learning from a fixed training set, performance of TD learning is worse than supervised learning.

Although we do not apply TD learning in Go successfully, using supervised learning, we can obtain good results with learning from a small set of training data. We propose a model that describes such extraordinary behavior. The model will be useful in reducing the complexity of space of many problems in computer Go.

Contents

Acknowledgement i

Abstract ii

1 Introduction 1

1.1 Overview 1

1.2 Objective 3

1.3 Organization of This Thesis 3

2 Background 5

2.1 Definitions 5

2.1.1 Theoretical Definition of Solving a Game 5

2.1.2 Definition of Computer Go 7

2.2 State of the Art of Computer Go 7

2.3 A Framework for Computer Go 11

2.3.1 Evaluation Function 11

2.3.2 Plausible Move Generator 14

2.4 Problems Tackled in this Research 14

3 Application of TD in Game Playing 15

3.1 Introduction 15

3.2 Reinforcement Learning and TD Learning 15

3.2.1 Models of Learning 16

3.2.2 Temporal Difference Learning 16
3.3 TD Learning and Game-playing 20
3.3.1 Game-Playing as a Delay-reward Prediction Problem 20
3.3.2 Previous Work of TD Learning in Backgammon 20
3.3.3 Previous Works of TD Learning in Go 22
3.4 Design of this Research 23
3.4.1 Limitations in the Previous Researches 24
3.4.2 Motivation 25
3.4.3 Objective and Methodology 26

4 Deriving a New Updating Rule to Apply TD Learning in Multi-layer Perceptron **28**

4.1 Multi-layer Perceptron (MLP) 28
4.2 Derivation of TD(λ) Learning Rule for MLP 31
4.2.1 Notations 31
4.2.2 A New Generalized Delta Rule 31
4.2.3 Updating rule for TD(λ) Learning 34
4.3 Algorithm of Training MLP using TD(λ) 35
4.3.1 Definitions of Variables in the Algorithm 35
4.3.2 Training Algorithm 36
4.3.3 Description of the Algorithm 39

5 Experiments **41**

5.1 Introduction 41
5.2 Experiment 1 : Training Evaluation Function for 7×7 Go Games by TD(λ) with Self-playing 42
5.2.1 Introduction 42
5.2.2 7×7 Go 42
5.2.3 Experimental Designs 43
5.2.4 Performance Testing for Trained Networks 44

5.2.5 Results . . . 44
5.2.6 Discussions . . . 45
5.2.7 Limitations . . . 47
5.3 Experiment 2 : Training Evaluation Function for 9×9 Go Games by TD(λ) Learning from Human Games . . . 47
5.3.1 Introduction . . . 47
5.3.2 9×9 Go game . . . 48
5.3.3 Training Data Preparation . . . 49
5.3.4 Experimental Designs . . . 50
5.3.5 Results . . . 52
5.3.6 Discussion . . . 54
5.3.7 Limitations . . . 56
5.4 Experiment 3 : Life Status Determination in the Go Endgame . . . 57
5.4.1 Introduction . . . 57
5.4.2 Training Data Preparation . . . 58
5.4.3 Experimental Designs . . . 60
5.4.4 Results . . . 64
5.4.5 Discussion . . . 65
5.4.6 Limitations . . . 66
5.5 A Postulated Model . . . 66

6 Conclusions **69**
6.1 Future Direction of Research . . . 71

A An Introduction to Go **72**
A.1 A Brief Introduction . . . 72
A.1.1 What is Go? . . . 72
A.1.2 History of Go . . . 72
A.1.3 Equipment used in a Go game . . . 73
A.2 Basic Rules in Go . . . 74

A.2.1 A Go game . . . 74
A.2.2 Liberty and Capture . . . 75
A.2.3 Ko . . . 77
A.2.4 Eyes, Live and Death . . . 81
A.2.5 Seki . . . 83
A.2.6 Endgame and Scoring . . . 83
A.2.7 Rank and Handicap Games . . . 85
A.3 Strategies and Tactics in Go . . . 87
A.3.1 Strategy vs Tactics . . . 87
A.3.2 Open-game . . . 88
A.3.3 Middle-game . . . 91
A.3.4 End-game . . . 92

B Mathematical Model of Connectivity **94**
B.1 Introduction . . . 94
B.2 Basic Definitions . . . 94
B.3 Adjacency and Connectivity . . . 96
B.4 String and Link . . . 98
B.4.1 String . . . 98
B.4.2 Link . . . 98
B.5 Liberty and Atari . . . 99
B.5.1 Liberty . . . 99
B.5.2 Atari . . . 101
B.6 Ko . . . 101
B.7 Prohibited Move . . . 104
B.8 Path and Distance . . . 105

Bibliography **109**

List of Tables

2.1 Comparison between Chess and Go 9

3.1 Performance of TD-Gammon 21

5.1 Input representation of a 7×7 board in experiment 1 43
5.2 Performance for different λ in experiment 1 45
5.3 Generalization error after training in experiment 2 53
5.4 Partial connectivity of MLP in experiment 3 62
5.5 Generalization error of the alive-or-dead problem in experiment 3 64

A.1 A summary of Strategy vs Tactics 88

List of Figures

4.1 A multi-layer Perceptron with 2 hidden layers. 29

5.1 8 ways of invariant transformation in experiment 2 51
5.2 Generalization summed square error (SSE) in experiment 2 54
5.3 Generalization error (discrete) in experiment 3 55
5.4 Extraction of a 8×8 corner pattern in experiment 3 59
5.5 Representation for a 8×8 corner in experiment 3 61
5.6 Connectivity between the input and first hidden layer in experiment 3 . . . 63

A.1 Liberties of a single stone 75
A.2 Liberties for a string of stones 76
A.3 Examples of capturing stones 76
A.4 Examples of cutting into separate strings 77
A.5 Examples of prohibited moves 77
A.6 Running out of liberty 78
A.7 Examples of Atari 78
A.8 Diagram showing cycle in Ko 79
A.9 Example of Ko and Ko threat 80
A.10 An example of termination of Ko 81
A.11 Examples of alive by having 2 eyes 82
A.12 Examples of big eyes 82
A.13 Examples of Seki 83
A.14 More examples of Seki 84

A.15 Positions for putting handicap stones 86
A.16 Two examples of Joseki 89
A.17 Example of a complicated Joseki with 33 moves 89
A.18 An example of Fuseki 90
A.19 Concrete territory vs Potential 91

B.1 Horizontal and vertical distance of some points 95
B.2 Major-adjacency and minor-adjacency of some points 96
B.3 Major-connectivity and minor-connectivity of some stones 97
B.4 Examples of string 98
B.5 Examples of link 99
B.6 Examples of liberty 100
B.7 Examples of capture 100
B.8 Examples of Atari 101
B.9 Examples of simple Ko 102
B.10 Examples that are NOT Ko 104
B.11 Examples of prohibited moves 105
B.12 Examples of major-path and major-distance 106
B.13 Examples of effective major-path and major-distance 107
B.14 Examples of string major-path and major-distance 108

Chapter 1

Introduction

1.1 Overview

The ability for computer to play intellectual games such as Chess, Checkers and Go, in a level better than human beings was thought as a breakthrough in artificial intelligence. It is because we commonly believe that being able to play such games well requires true "intelligence." Now, we have computer Chess programs that are able to beat our world Chess champion. We also have programs that can play Backgammon in a level very close to our world Backgammon champion. So, it is reasonable to think that the research for computer playing intellectual games will come to a ending stage soon. However, this thinking is refuted by the case in Go (*Wei-Qi* in Chinese).

The game Go has certain distinct properties that make it difficult for computer to play well [6]. First, it has a huge game space, which is much bigger than that of Chess. Second, it has a large branching factor, making a deep exhaustive search for good moves impractical. Furthermore, it is difficult to evaluate a situation of the game. As a result, it is not possible to adopt the techniques in computer Chess in computer Go to produce similar success. Current approach in solving the game is to design programs in a heuristic way that requires a lot of expert knowledge of the game. The resulting programs are very weak — at a very low level that even a novice player can beat the best computer program easily.

For all these difficulties, the problem of computer Go is a good problem in artificial

intelligence that we may need to model the way human being thinks in order to solve the problem. This is not the case in computer Chess because Chess programs play Chess in a very different way from human players. The Chess programs rely heavily on searching instead of on the knowledge of the game [6]. On the other hand, computer Go researchers have to build Go programs from our knowledge and experience in the game. However, from our current knowledge, we find that it is difficult for us to model our Go knowledge precisely because our reasoning in Go is in a very abstract form.

Aside from the current heuristic programming approach in solving computer Go, there are some alternate ways to tackle the problem. Machine learning, which tries to solve problem through learning from experience, will be a good direction in computer Go. Neural networks, which is a learning model that solves a problem by learning the implicit knowledge from a set of training examples, appears to fit the problem of computer Go because our knowledge of the game is too abstract to formalize and implement in a program. With this approach, we will not need to explicitly extract the knowledge that is important and organized it in a good structure. These will hopefully be solved implicitly by neural networks.

Temporal difference (TD) learning is a *reinforcement learning model*. It has been applied in Backgammon. Like Go, Backgammon is also a game that cannot be solved by the model used in computer Chess because of the huge space generated from the random process of dice rolling. Applying TD learning in Backgammon gives very good results: computer Backgammon programs using this approach are able to play in a level very close to the human world champion [28] [29]. The success of this application is comparable to the one in computer Chess as in both cases. The best programs reach the world champion level.

TD learning has been applied to Go following its success in Backgammon [24]. Some preliminary results show that applying TD learning in Go can obtain programs that can beat a commercial Go program (playing at its weak level) in games with reduced board size. Since then, there are no further investigation for this approach. However, there are some distinctive difference between Go and Backgammon: determinism. This may have

a critical effect on the performance of TD learning. In this research, we would like to investigate this, as well as using the classical supervised learning model, in the problem of computer Go.

1.2 Objective

In this research, we focus on applying neural networks in solving some problems in computer Go to investigate its feasibility. The problems that we try to tackle are board evaluation function and alive-or-death analysis. TD learning, as well as a classical model of learning: *supervised learning* are used to solve the problems.

1.3 Organization of This Thesis

In Chapter 2, we will give a *practical* definition for "solving" computer Go. Then as a background information, we will present the state of the art of computer Go and analyze the large difference between computer Go and computer Chess by the difference of the two games. Finally, we will introduce our framework that decomposes the problem of computer Go into smaller sub-problems. We will state the sub-problems that we are trying to focus in this research.

In Chapter 3, we focus on the previous neural network applications in games, especially for the TD learning model. Previous research for applying TD to Backgammon and Go are described. We state the insufficiencies of previous research and give the motivations of our research in applying neural network learning model in some of the sub-problems of Go.

In Chapter 4, we will give the derivation of a new updating rule for applying TD learning in a popular neural network model called multi-layer Perceptron (MLP) [22]. It is necessary for us to derive this updating rule as well as a training algorithm for the rule as they are needed in performing experiments in this research.

In Chapter 5, we will present three sets of experiments performed in this research.

We will give the objectives and designing issues for each experiment. The results and significance of each experiment are also discussed.

In Chapter 6, we will give the conclusions of this dissertation by giving a summary of contributions in this research. We also presents a list of extensions to this research.

Some supplementary information are given in the appendices. In Appendix A, we give a brief introduction to the game of Go. Some complicated characteristics of the game are demonstrated based on the author's experience in the game. Interested readers are recommended to refer to some Go books for extra knowledge of the game.[1]

In Appendix B, we present some definitions of basic concepts in Go using mathematical notations. To define the knowledge of Go in a precise form, we need to have a set of formal definitions for the basic concepts. The definitions we have done in this appendix will be useful for designing a rigorous model of Go knowledge in the future.

[1]Go books written in Chinese are extensively available. We recommend some English ones here: [5], [12], [18].

Chapter 2

Background

2.1 Definitions

2.1.1 Theoretical Definition of Solving a Game

There are extensively different forms of games. The class of games discussed here is *2-player zero-summed board games with perfect information*. Many of the recreational games we play belong to this class, such as Backgammon, Checkers, Chess, Chinese Chess, Go, Go-Moku (connect-5), Shogi, etc. A game can also be deterministic (like Go) or non-deterministic (like Backgammon). In a deterministic game, state transition is totally controlled by the will of the players. On the other hand, in a non-deterministic game, there are some random factors included in the state transition.

Human beings not only play games but also want to study the theories of the games so as to come up with better strategies. There are many theoretical definitions of solving a game. Here is a set of definitions with different levels of strength for solving a game:

1. **ultra-weakly solved**: for the initial position(s), game-theoretic value[1] has been determined.

2. **weakly solved**: for the initial position(s), a strategy [2] has been determined to obtain

[1]For a deterministic game, the game-theoretic value means the outcome of the game under perfect plays. For a non-deterministic game, it represents the probability of winning under perfect play.

[2]*Strategy* for a game here means a set of rules for state-transition for a player in the space of the game.

at least the game-theoretic value of the game, for both players, under *reasonable resources.*[3]

3. **strongly solved**: for each legal position, a strategy has been determined to obtain the game-theoretic value of the position, for both players, under *reasonable resources.*

The three definitions are in increasing order of strength. This means that to strongly-solve a game is the most difficult among the three.

However, most of the games we play, like the ones mentioned before, are too complicated to be solved theoretically using these definitions because their game spaces are huge. As a result, we need to solve the game in an easier but useful way. We define one more notation of solving a game :

practically solved : for the initial positions(s), a *high quality* strategy has been determined that will obtain at least the game-theoretic value of the game *in a high probability.*

This definition is even weaker than definition (2) as it accepts a sub-optimal strategy for the game.

This new definition arouses another problem : it is difficult to analytically evaluate whether a strategy has high quality. As a result, the strategy should be tested empirically by comparing it to some existing high-quality strategies, which are usually the strategies of the human expert player of that type of game.

In practice, a strategy is implemented in a computer program. Comparison of two strategies is done by matching between the two agents, with each agent implemented one of the strategies. (for example, a computer program against a human expert player) Therefore, the problem of *practically solving* a game can be understood as building a computer program that can play the game in a level higher than the human expert of that game.

[3] *Reasonable resources* may be, say, a few minutes' computation of the fastest available computer.

2.1.2 Definition of Computer Go

We define the problem of *computer Go* as to *practically solve* the Go game. The goal of solving the problem is therefore to *build a computer program* that can play Go in a level *comparable to the strongest human expert of Go*. Similar definition of the problems of computer Chess and computer Backgammon are used in this thesis.

Since the strategy implemented in a computer Go program is sub-optimal, it is always possible to build programs stronger than the existing best one until the optimal strategies for the initial positions have been found and implemented. We need a scale to measure the progress of computer Go instead of just comparing new programs with the best existing programs. Since currently, the strongest computer Go programs is far weaker than an average human player, we can monitor the performance of a computer Go program by the scale we use for human players. The grades in the scale can be roughly classified into 5 levels [4] :

1. **level 1** - Beginner's Level: understanding the basic rules.
2. **level 2** - Novice Level: average human player's level.
3. **level 3** - Human Expert Level: top 10% human players' level.
4. **level 4** - World Champion Level: competing with world champion.
5. **level 5** - Over Champion Level: stronger than all human players.

According to this scale, the progress of Computer Go is around level 2. The progress for another two games which are well-practically-solved : Chess and Backgammon, is around level 4.

2.2 State of the Art of Computer Go

Computer Go has a short history. Thus the research effort on this topic is little comparing to computer Chess. The first paper which talks about computer Go was published

[4] Please refer to Appendix A for the conventional ranking system.

in 1968 by Albert Zobrist, who built the first computer program that can play a complete Go game [31]. Since the first international computer Go congress took place in 1986, many programs have been developed. Here are some landmark programs :

1. *Go Nemesis* by Bruce Wilcox [30],
2. *Many Faces of Go* by David Fotland [14],
3. *Go Intellect* by Ken Chen [11] and
4. *Handtalk* [5] by Chen ZhiXing.

After many years of research effort, the performance of computer Go programs are increasing steadily but slowly. The current strongest computer Go program is only at a beginner to intermediate *amateur* player's level. This is a large contrast to the achievement in computer Chess, which has reached the world champion level.

The large progressive difference between computer Chess and computer Go is owing to the different natures between the two games and the techniques applied to solve the games. Some important properties of the two games are compared and summarized in Table 2.1 . We are trying to understand how the difference in the two games contributes to such a difference in performance.

In computer Chess, the approach used is basically the brute-force searching. Before a move is made, a large number of nodes in the game tree are searched and evaluated. Some well developed algorithms liked Minimax, α-β tree pruning and conspiracy number [19] are used in searching. Using this approach, with an accurate *evaluation function*, accuracy of the moves made by the program will improve if more levels and nodes in the game tree are examined. As a result, performance of a computer Chess programs using this approach will simply improve with the increase of computing power.

Learning from the difference between Chess and Go, it is found that techniques in computer Chess cannot be applied to computer Go because :

[5] *Handtalk* is the strongest computer program which won the computer Go congress in 1996. Its level is around 5-kyu to 10-kyu.

	Feature	**Chess**	**Go**
1	board size	8×8 squares	19×19 grid
2	# moves per game	≈ 40	≈ 200
3	branching factor	small (≈ 35)	large (≈ 200)
4	end of game and scoring	check-mate (simple definition - quick to identify)	counting territory (consensus by players hard to identify)
5	long range effects	pieces can move long distances (e.q., queen rooks, bishops)	stones do not move, patterns of stones have long range effects (e.g. ladders; life and death)
6	state of board	changes rapidly as pieces move	mostly changes incrementally (except for captures)
7	evaluation of board positions	good correlation with number and quality of pieces on board	poor correlation with number of stones on board or territory surrounded
8	programming approaches used	amenable to tree searches with good evaluation criteria	too many branches for brute brute force search, pruning is difficult due to lack of good evaluation measures
9	human lookahead	typically up to 10 moves	even beginners read up to 60 moves (deep but narrow searches e.g., ladders)
10	horizon effect	grand-master level	beginner level(e.g., ladders)
11	human grouping processes	hierarchical grouping [9]	stones belong to many groups simultaneously [21]
12	handicap system	none	good handicap system

Table 2.1: Comparison between Chess and Go [6].

1. **High branching factor of the game tree.** The average number of valid moves in a Go board position is larger than 200 while it is around 35 in Chess.

2. **Large depth of the game tree.** The average number of moves in a Go game is around 200 while in Chess, it is around 40.

3. **Difficult to evaluate a board position.** Unlike Chess, there is no strong correlation between winning and number of stones in Go. Strategic factors must be considered in evaluation, making the design of evaluation function difficult.

These properties of Go make the brute-force search model impractical to apply. Consequently, a different approach is needed for computer Go. Techniques used in the current strongest computer Go programs are mainly heuristic programming and knowledge engineering [14]. A lot of expert Go knowledge is implanted into the programs by the designer. Searching (lookahead) is only used selectively, usually in some local tactical analysis.

There are several problems in this approach.

1. *It is difficult to define the knowledge in Go clearly.* Human experts understand many concepts in Go in very abstract form. Some examples are *Moyo*, *group strength*, *influence*, *Aji*, etc. To define and formulate such knowledge precisely into rules is hard, especially when full understanding is not acquired by expert players.

2. *Engineering becomes difficult when the size of the knowledge-base increases.*

Since directly programming knowledge of Go has certain difficulties, it is reasonable to use the machine learning model to extract some useful knowledge from the Go game, especially when the knowledge is difficult to formalize and the knowledge is not well understood by the human experts themselves.

In our research, learning using neural networks is applied to solve some specific problems in computer Go. Decomposition of the problem into smaller subproblems will be discussed in the next section.

Before finishing the discussion, it is important to state that although computer can play very strong Chess, the brute-force searching model applied in computer Chess is very

different from the way human player thinks. On the other hand, in solving computer Go, researchers need to understand how expert players of Go think and model their knowledge representation and reasoning. Since modeling the human intelligence is a difficult job, computer Go is considered as the *grand challenge* of artificial intelligence (AI) [2].

2.3 A Framework for Computer Go

In this section, we propose a framework which is designed for solving the problem of computer Go. Theoretically, computer Go program can be developed with a search engine together with the following functions :

- **Evaluation function**: a function which gives the game value of any legal board positions.
- **Legal move generator**: a function which gives all legal (valid) moves for a given board position.

With these two functions, the search engine can select the move leading to the best result after searching n levels exhaustively. However, in practice, the search space of Go is too large for exhaustive search even when n is very small. As a result, it is an usual case that the search space has to be reduced by limiting the number of moves to search at each level. This is done by a *plausible move generator* which picks up moves that can be potentially good and ignores moves that are certainly bad for a board position. The evaluation function and plausible move generator are therefore the two important units in designing a computer Go program. We are going to analyze the design of these two units and propose our approach in solving some sub-problems in the design.

2.3.1 Evaluation Function

An evaluation function f_{Ev} is a function :

$$f_{Ev} : \mathcal{B} \rightarrow \mathcal{V}$$

where $\mathcal{B}$ is the set of all legal board positions and $\mathcal{V}$ is the set of legal game values. For deterministic game like Go and Chess, if f_{Ev} is perfect, $\mathcal{V} = \{-1, 0, 1\}$ with -1 means white wins, 0 means draw and 1 means black wins. It is usual to implement the function to give output values in a continuous range, with different values representing different extent of favorableness to both sides. In this case, $\mathcal{V} = [-1, 1]$. For example, for both board positions B_1 and B_2 with evaluations : $f_{Ev}(B_1) = 0.9$ and $f_{Ev}(B_2) = 0.1$, B_1 will be more favorable to the black side than B_2. For non-deterministic games like Backgammon, $\mathcal{V} = [0, 1]$ which represents the winning probability of one side for a board position. The range $[0, 1]$ is sufficient in this case because in non-deterministic zero-summed games, $P(\text{white will win}) = 1 - P(\text{black will win})$.

From the experience of computer Go research, it is found that designing an evaluation function for Go is very difficult [6]. Consequently, it is suitable to break down the large problem of evaluation function into smaller sub-problems by designing sub-functions for different time, space and properties in a Go game. Here are three types of possible decompositions that we propose:

A. Temporal Decomposition. Special sub-functions can be designed for evaluating different stages in a Go game. A Go game is commonly divided into three stages : *Open-game*, *Middle-game* and *End-game*. Special evaluation function can be designed for each stage of a Go game. We define h_{st} to be the classification function that identifies the stage of a board position. For a board position B, $h_{\text{st}}(B) = 1, 2$ or 3 if B is in the Open-game, Middle-game or End-game respectively. The design of evaluation using temporal decomposition is in the form of

$$f_{\text{Ev}}(B) = \begin{cases} f_{\text{Ev-open}}(B) & : \textbf{ if } \quad h_{\text{st}}(B) = 1 \\ f_{\text{Ev-mid}}(B) & : \textbf{ if } \quad h_{\text{st}}(B) = 2 \\ f_{\text{Ev-end}}(B) & : \textbf{ if } \quad h_{\text{st}}(B) = 3 \end{cases}$$

In this case, the three sub-functions : $f_{\text{Ev-open}}$, $f_{\text{Ev-mid}}$ and $f_{\text{Ev-end}}$ are applied independently of each other. Therefore, this is a simple decomposition method.

B. Spatial Decomposition. Different sub-functions can be designed for different local area of the board. A board position B is segmented into a set of sub-boards : $\{B_1, B_2, \ldots, B_n\}$. The sub-boards are distinct and totally cover the whole 19×19 board. That is :

$$\bigcup_i B_i = B \qquad \text{and} \qquad B_i \cap B_j \neq B_i, i \neq j$$

Sub-boards may be overlapping to avoid loss of information in the boundaries. Moreover, sub-boards in the similar region may share the same evaluation sub-function. (For examples, windows in around the central area will have similar properties and can use the same sub-function.) Each sub-board is evaluated separately and the results are combined :

$$f_{\text{Ev}}(B) = \Gamma\Big(\alpha_1(B_1), \alpha_2(B_2), \ldots, \alpha_n(B_n)\Big)$$

where $\alpha_i(B_i)$ is an analyzer for a sub-board position B_i and $\Gamma(.)$ is the function for combining the results from the n sub-boards. The best combination function has to be determined by tuning in the real games. A simple example of $\Gamma(\cdot)$ is a linear combination of the sub-functions.

C. Functional Decomposition. Special sub-functions can be designed to evaluate some special properties that will assist in board evaluation. Some examples are *group strength*, *alive-or-dead state*, *influence*, etc. Let the properties analyzed be $P_1, P_2, \ldots, P_m$. Evaluation of a board position B will be :

$$f_{\text{Ev}}(B) = \Phi\Big(\beta_{P_1}(B), \beta_{P_2}(B), \ldots, \beta_{P_m}(B)\Big)$$

where $\beta_{P_i}(B)$ is the analyzer for property P_i and $\Phi(.)$ is the function for combining results of m property analysis.

Considering the three compositions introduced, temporal decomposition is the simplest model because all sub-functions $f_{\text{Ev-open}}(\cdot)$, $f_{\text{Ev-mid}}(\cdot)$, $f_{\text{Ev-end}}(\cdot)$ are independent and there is no combination of sub-functions. In spatial and functional decomposition, results of the sub-functions (α_i and β_{P_j}) have to be recombined. The re-combination function is difficult to design analytically.

A good design using the three decompositions is with the temporal decomposition at the top level. For designing sub-function for each stage of the game, specific properties are analyzed in function decomposition and special segmentation method is applied in regional decomposition. The reason for this design is because from our understanding of Go, very different analyzing methods are applied on different stages. This hierarchical design fits our model of understanding of the game.

2.3.2 Plausible Move Generator

Plausible move generator selects the potentially good moves from a legal board position. For a board position B, let $M(B)$ be the set of all legal moves for B. The results from the plausible move generator, $g_{\text{pm}}(B)$, is a subset of M_B. The effective-factor $\frac{|g_{\text{pm}}(B)|}{|M_B|}$ determines the ratio of pruning and thus reduction of the search space.

In practice, the design of plausible move generator for a board position B can be more effective if it is *goal-directed*, in which the goal to be achieved is supplied to the generator before plausible moves are generated. Therefore, plausible move generator shall be based on the result of board evaluation. As a result, evaluation function plays a very important role in the problem of computer Go.

2.4 Problems Tackled in this Research

In the research, we adopt *machine learning* using *artificial neural networks* to solve the problems of evaluation function and a property analyser (alive-or-dead analyser). The motivations of using this approach will be discussed in the next chapter. We train neural networks to *evaluate board positions of reduced board size* directly since neural network is able to learn the internal relationship from the examples of a problem. We also train neural networks to do the *alive-or-dead analysis*, which is an important sub-problem in designing an evaluation function. The methodology of the applications will also be discussed in the next chapter.

Chapter 3

Application of TD in Game Playing

3.1 Introduction

As illustrated in [28] [29], *temporal difference* (TD) learning has been applied successfully in the game Backgammon via learning by neural networks. In this thesis, we apply neural network learning, especially the TD learning method, to solve problems in computer Go.

In this chapter, we first give a background of the neural network learning models and the TD learning. Then we will talk about the relevance of TD learning in game-playing and report the previous applications of TD learning in Backgammon and Go. Finally, learning from the previous applications mentioned, we will design our approaches in solving problems in computer Go by neural networks.

3.2 Reinforcement Learning and TD Learning

In this section, we will give a background of Reinforcement learning as one of the three fundamental categories of neural network learning. Derivation of TD learning, which is a reinforcement learning model invented by Sutton [25], will then be explained.

3.2.1 Models of Learning

Learning can be classified into three categories : *unsupervised* learning, *supervised* learning and *reinforcement* learning.

In *unsupervised learning*, the neural network receives a training set consists of input training patterns only. The network learns to adapt based on the experiences collected through the training patterns. This means that the network does not need to learn a specific example of function. Learning is made for a task-independent measure of the quality of representation. This type of learning can be applied in the clustering problems, competitive learning, PCA, etc.

In *supervised learning*, the training data set consists of input-output pairs : $X = \{(x_i, y_i) |\ i = 1, \ldots, n\}$. Each pair contains an input pattern x_i and the desired output y_i for that pattern. The error between the actual response (network's prediction) and desired responses for each pattern is used to adjust the parameters (weights) inside the network. Typically, supervised learning rewards accurate responses and punishes incorrect ones. The weights are modified in the opposite direction of the error gradient. Therefore, most supervised learning algorithms reduce to stochastic minimization of error in the multi-dimensional weight space. Supervised learning is the most widely used learning model with large number of successful applications.

In *reinforcement learning*, the learning agent produces a response for each training pattern. The response will be fed into a specific environment and a reinforcement signal reflecting the correctness of the response, instead of the correct response itself, will be returned. The reinforcement signal is usually a scalar value. The learner will use this reinforcement signal to learn. This model is different with supervised learning as no desired response will be provided for each input pattern.

3.2.2 Temporal Difference Learning

Temporal Difference (TD) learning, which is first introduced by Sutton [25], is a reinforcement learning method for *delay-reward* (or multi-step) prediction problem. In

a delay-reward prediction problem, observation-outcome sequences has the form $(o(1), o(2), \ldots, o(m), \chi)$ where each $o(t)$ is a vector of observation available at time t and χ is the target of the predictions which is not revealed until the m-th observation. For the observation sequence, the learning agent produces a corresponding sequence of predictions : $(P(1), P(2), \ldots, P(m))$. Each prediction is made to estimate the target χ.

Using a supervised learning model to solve the delay-reward prediction problem, the observation-outcome sequence will be treated as a set of independent pairs of input-target : $(o(1), \chi), (o(2), \chi), \ldots, (o(m), \chi)$. The target χ is needed for computing weight increment for each input pattern. For an observation $o(t)$, the network will produce a prediction $P(t)$. The weight increment for $P(t)$ using the prototypical supervised learning rule is :

$$\Delta w(t) = \eta(\chi - P(t))\nabla_w P(t) \tag{3.1}$$

where $\nabla_w P(t)$ is the partial derivative (gradient) of $P(t)$ with respect to the weight w and η is the learning rate. The value of $\nabla_w P(t)$ is easy to compute for a linear neural network. For a non-linear network like multi-layer Perceptron (MLP), computation is more complicated.

The total weight increment for all the predictions from $P(1)$ to $P(m)$ is

$$\begin{aligned} \Delta w &= w + \sum_{t=1}^{m} \Delta w(t) \\ &= w + \sum_{t=1}^{m} \eta(\chi - P(t))\nabla_w P(t) \end{aligned} \tag{3.2}$$

The problem for using the supervised learning model in solving the delay-reward prediction problem is that weight update depends critically on the target of the predictions χ which will not be available until all predictions have been made. No weight update can be computed until χ is revealed. Thus, all the observations and predictions made during a sequence must be stored until we obtain the target χ and compute all weight increments.

To solve the problem, it is necessary to reformulate the learning rule. First, the error term $(\chi - P(t))$ is changed to a sum of differences of successive predictions :

$$\chi - P(t) = \sum_{k=t}^{m} (P(k+1) - P(k)) \qquad \text{where } P(m+1) \text{ is defined as } \chi \tag{3.3}$$

Then we try to substitute this result into equation 3.2 :

$$\begin{aligned}
\Delta w &= w + \sum_{t=1}^{m} \eta(\chi - P(t))\nabla_w P(t) \\
&= w + \sum_{t=1}^{m} \eta \sum_{k=t}^{m} (P(k+1) - P(k))\nabla_w P(t) \quad \text{(Substitution by 3.3)} \\
&= w + \sum_{k=1}^{m} \eta \sum_{t=1}^{k} (P(k+1) - P(k))\nabla_w P(t) \\
&= w + \sum_{t=1}^{m} \eta (P(t+1) - P(t)) \sum_{k=1}^{t} \nabla_w P(t)
\end{aligned}$$

We obtain an equation of weight update for all predictions P_1, P_2, ..., P_m. The weight increment corresponding to a specific prediction made at time i can be computed by :

$$\Delta w(i) = \eta(P(i+1) - P(i)) \sum_{k=1}^{i} \nabla_w P(k) \tag{3.4}$$

which is an equation independent of χ (unless when $i = m$ where $P(m+1)$ is defined as χ). Unlike equation 3.1, computation of weight update is *incremental* using this equation because the value of χ is not involved. Weight increment for $P(i)$ can be computed immediately when prediction at time $(i+1)$ is available and there is no need to wait for the target χ for this calculation.

Weight increment using equation 3.4 is called the linear temporal difference procedure, TD(1). It produces the same result as the prototypical supervised learning procedure if both are in *batch updating mode* (weights are updated as in equation 3.2). In practice, the weight will usually be updated immediately after each training example, which is called *immediate updating mode*. In such case, this equation will not produce exactly the same effect as the supervised learning rule.

An exponential factor λ can be added to the term of gradient-sum to control the effect of 'experience' from past predictions with *recentness* :

$$\Delta w(t) = \eta(P(t+1) - P(t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P(k) \qquad \text{where } \lambda \in (0, 1). \tag{3.5}$$

The new parameter λ takes the value between 0 and 1. A family of learning procedures, $\mathbf{TD(\lambda)}$ is obtained with different values of λ. The parameter λ controls the effect of past

learning experience. The influence of the past predictions will be weaker if λ takes a smaller value. When $\lambda = 1$, the learning procedure takes all past gradient with the same importance and takes no credit of the temporal order of the past prediction sequence.

The term of gradient-sum can be computed incrementally. If $e(t)$ represents the gradient-sum at time t, i.e.

$$e(t) = \sum_{k=1}^{t} \nabla_w P(k),$$

the gradient-sum at time $t+1$ can be computed by

$$\begin{aligned} e(t+1) &= \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla_w P(k) \\ &= \nabla_w P(t+1) + \sum_{k=1}^{t} \lambda^{t+1-k} \nabla_w P(k) \\ &= \nabla_w P(t+1) + \lambda\, e(t) \end{aligned} \tag{3.6}$$

We can see that $e(t)$ forms a recurrence series. Therefore, it is a simple recursive calculation for computing the past gradient-sum, $e(t)$, during training as predictions are made in the temporal order : $P(1)$, $P(2)$, ..., $P(m)$.

In a special case when λ is equal to zero, the past gradient-term $e(t)$ will reduce to $\nabla_w P(t)$, which contains only the current gradient. The equation for weight increment for TD(0) then becomes

$$\Delta w_t = \eta(P(t+1) - P(t))\nabla_w P(t). \tag{3.7}$$

The weight updating here looks very similar to the prototypical supervised learning method in equation 3.1. The learning mechanism is the same. The difference is only in the error term — TD(0) learning uses the difference between two successive predictions as the error term while supervised learning uses the difference between the target and the prediction. As a result, learning algorithm for supervised learning can be used for TD(0) learning with simple modifications.

3.3 TD Learning and Game-playing

3.3.1 Game-Playing as a Delay-reward Prediction Problem

There is a close relationship between game-playing and *delay-reward prediction problem*, which is the type of problem that TD learning is aiming to tackle. Taking Go as an example, the problem of predicting the outcome of the game from a Go board position, which is the objective of the *evaluation function* [1] can be formulated as a delay-reward prediction problem. The observation-outcome sequence of the problem becomes $(B(1), B(2), \ldots, B(m), \chi)$ where $B(i)$ is the board position after the i-th move in a Go game and χ, which is the target of all predictions, is the outcome of the Go game. From each board position $B(i)$ $(i = 1, \ldots, m)$, a prediction $P(i)$ is made for predicting the outcome of the game. The game outcome χ is revealed to the network only after the final board position $B(m)$ is reached.

With this formulation, we can train neural networks to evaluate board positions of a specific game by TD learning. Sequences of board positions, in temporal order, are passed to the network for training. In practice, TD learning has been applied in Backgammon and Go. The application in Backgammon is very successful [29]. On the other hand, the application in Go, which is motivated by the success in Backgammon, is in a preliminary stage that requires more investigation. We will discuss these two applications in the following sections.

3.3.2 Previous Work of TD Learning in Backgammon

Application of TD learning in Backgammon is first demonstrated by Tesauro [27]. Before using TD learning, Tesauro has applied supervised learning to train neural networks to evaluate board positions in Backgammon [26]. In applying supervised learning, the training examples are selected using expert knowledge. The resulting program, *Neurogammon*, became the strongest computer Backgammon program at that time.

[1] Please recall section 2.3.1, for the definition of an evaluation function

Program	Training Games	Hidden Units	Results
TDG 1.0	300,000	80	-13 points in 51 games (-0.25 ppg)
TDG 2.0	800,000	40	-7 points in 38 games (-0.18 ppg)
TDG 2.1	1,500,000	80	-1 points in 40 games (-0.02 ppg)

Table 3.1: Results of testing TD-Gammon in play against world-class human opponents. (ppg = point per game)

After implementing the Neurogammon program, Tesauro applied TD(λ) learning to train neural networks to do Backgammon board evaluation by *self-playing* $\lambda = 0$ is used throughout the experiments [27] [28]. The resulting program, TD-Gammon achieves a level very close to the world champion [29].

During training by self-playing, the neural network is used to *predict and control* in the same time. The network is used to evaluate board positions in a 1-ply search (*predict*). The move with the best evaluation is selected and played and the board position is changed by the network's move (*control*). As a result, unlike training Neurogammon, there is no need to supply training example to the network. The network can *learn by itself.*

It should be noted that there is no a priori knowledge built in the architecture of network and every synaptic weight is initialized with a random number before training. Performance of the network after training with this design is very satisfactory. When a raw input representation is used, the trained network is able to play at a *strong intermediate level* comparable to the Neurogammon. When an extra set of hand-crafted features are added to the input representation, the trained network achieves *human expert level.*

Experimental results show that with more training games, the network performs better. Table 3.1 summarizes the performance of different versions of TD-Gammon. It should be noted that the performance of the strongest version of the TD-Gammon (2.1) loses only 0.02 point per game, which is a very small amount, when it plays against the human world champion of Backgammon. This result shows that the problem of computer Backgammon is almost *practically solved.*

The research performed by Tesauro shows that performance of TD learning using self-playing (as in TD-Gammon) surpasses the one of supervised learning model approach (as

in Neurogammon) for the game Backgammon. It is suggested that the stochastic nature of Backgammon is critical to the success of TD learning. The dice-rolling in Backgammon enforces a good amount of exploration of the state space, forcing the system into regions of the space that are new to the current evaluation function. In contrast, the state space explored by the network in supervised learning is limited to the training set. More training examples have to be supplied to increase the amount of state space exploration during training. Moreover, it is often difficult to design a "good" training set which provides more state space exploration for training neural networks.

On the other hand, in the case for deterministic games, it is unknown that training by TD learning can produce similar good results as in Backgammon. A neural network trained by self-playing could end up exploring only some very narrow portion of the state space. Consequently, the resulting network might develop some poor strategy that nevertheless gives self-consistent results when it plays against itself.

3.3.3 Previous Works of TD Learning in Go

The application of TD learning in Go, for training evaluation function for 9×9 board positions, has been studied by Schraudolph, et al.[24]. This is the only significant research on applying TD learning in Go (and also applying neural networks in Go) to the author's knowledge.

In their experiments, a small board size of 9×9 is used instead of the standard 19×19 in order to reduce the complexity of the problem. TD(0) is used in training which is the same as the application in Backgammon.

There are two special issues in the design. First, prediction is made on the *outcome of each position* on the board instead of the outcome of the game. The outcome of each position, which can either be a *black's territory* (or a black stone) or a *white's territory* (or a white stone), is available after a game is completely finished. Such information together can determine the outcome of the Go game. This is a special property in Go. Second, architecture of the network is *specially designed* which reflects the spatial organization on

the board.

Different opponents are used to train different networks. The opponents used are :

1. the network itself (self-playing),
2. random move generator,
3. a public-domain Go program: *Wally*,
4. a commercial Go program: *The Many Faces of Go*, in its weak playing level.

Gibbs sampling [16] is used to select moves stochastically during self-playing. This is to avoid the risk of trapping the network in some suboptimal fixed state. The probability of a move being chosen is exponentially related to the evaluation of the board position to which the move leads. A parameter called *temperature* is used to control the degree of randomness of the stochastic selection. When the temperature is high, moves will be chosen more randomly. When the temperature is low, moves will be less random : move with a better evaluation will be chosen with a much higher probability. The temperature is set to be very high in the beginning of training and is gradually cooled down during training.

The goal of the experiments is to compare the performance between two networks: *an undifferentiated network* and *a specially structured network* when they are trained with some available Go programs. Experimental results show that networks with specially designed structure performs better. Moreover, the networks are able to out-perform the Go programs used in training after certain number of training games.

3.4 Design of this Research

Our main focus in this research is in applying neural network learning in computer Go. In this section, first, some limitations of the past applications are stated. Then we talk about the motivations and objectives of this research. The design of solutions to the problems are also presented.

3.4.1 Limitations in the Previous Researches

The previous (which is also the only one) research of applying of TD learning in Go [24] has two major limitations :

1. **Only TD(0) learning has been used.** In both of the previous TD applications, only TD(0) learning has been used. *Simplicity* is the first concern. As mentioned in equation 3.7 in section 3.2.2, training algorithm for supervised learning can be used for TD(0) with simple modifications. Therefore programs for the training experiments can be easily implemented for TD(0).

   Moreover, *efficiency* is another concern for the choice of TD(0). From equation 3.5 and 3.6 in section 3.2.2, it can be observed that for general TD(λ) learning, the term for past gradient-sum, $e(t)$, has to be stored for computation of weight increment in the time $t+1$. Since each weight in the neural network has its own gradient-sum, the demand of storage during training is at least doubled. Furthermore, the time needed for computation of $e(t)$ for each synaptic weight will make the training process of general TD(λ) slower than TD(0).

   Since only TD(0) has been applied in previous experiment, performance of TD(λ) learning for different values of λ has not been investigated. If we try to apply TD(λ) for any values of λ in $[0, 1]$, a new updating rule and training algorithm have to be designed for a specific model of neural networks.

2. **Training and testing performance with existing Go program.** The previous experiment of applying TD in Go tries to train and test networks with some Go programs which are relatively weak among other existing computer Go programs. Consequently, the trained evaluation function may not perform uniformly well against other new opponents. The function may be biased to the playing styles of the Go programs that it is trained with. Moreover, results of beating the Go programs during training does not imply good general performance because the evaluation functions trained has not been tested with some objective scales. Actually, there are

more general measures for performance evaluation, such as the percentage of correct evaluation of board positions taken from real Go games (played by human players).

3.4.2 Motivation

Summarizing from the past works and from our own ideas, we have found the followings:

1. **TD Learning is good for training evaluation function.** As discussed before, in solving computer Go, designing a good evaluation is a very important problem. Since evaluation function can be modeled as a delay-reward prediction problem, the function can then be trained by TD learning in a straightforward way. As a result, applying TD learning in training evaluation function, which is the method applied in Backgammon and Go, has a good prospect.

2. **New updating rule is needed for general TD(λ) in multi-layer Perceptron.** The derivation of TD(λ) learning is independent of any neural network model. The gradient $\nabla_w P(t)$ has to be calculated for different models of neural networks. In the two previous applications of TD learning, *multi-layer Perceptron* (MLP) is used. However, weight updating rule for applying TD(λ), for λ greater than 0, in MLP has not been defined. As a result, we need to derive the updating rule in order to apply the general TD(λ) learning in Go.

3. **Abundant Go game records provide good source of implicit Go knowledge for machine learning.** Game records in electronic form are available in some Go servers on the Internet. They provide a good source of implicit Go knowledge for learning by neural network. Moreover, training examples can be generated from the existing game files automatically.

   Furthermore, the past application of neural network in Go only explores training by self-playing and by playing against existing Go programs. It will be a good way to train neural networks by using human games as examples because the games can provide high quality knowledge to the networks.

4. **Neural Networks can be used in solving difficult sub-problems in Go.** Not only for TD learning, the approach of learning by neural network has also not been applied in computer Go. In the current approach of computer Go, the techniques of *heuristic programming* plus *knowledge engineering* cannot effectively handle some concepts that are very *abstract* and even not fully understood by human experts. (Some of such concepts are pointed out in section A.3 of appendix A.) As a result, trying to learn such concepts from examples in human games by neural networks is a helpful direction to solve such problems.

3.4.3 Objective and Methodology

The main objective of this research is to apply machine learning using neural networks to solve some problems in computer Go. We try to analyze the feasibility of this approach and set a milestone by discovering good directions for further research in this area. This provides an alternative for solving problems in computer Go.

The approach in tackling the problems is summarized in the following points :

1. **New updating rule for TD(λ) in MLP.** First, we try to derive a new updating rule for applying TD(λ) learning in multi-layer Perceptron so that different values of λ can be used in TD(λ).

2. **TD(λ) learning of evaluation function.** Some experiments are then performed to use the TD(λ) learning algorithm in training MLP's for Go board evaluation. Both self-training and training with games played by human players are used. The experiment of training with human games is a new method in application of TD learning. Performance of difference values of λ are analyzed.

3. **Solving a pattern analysis sub-problems using MLP.** Finally, we try to apply MLP to do *alive-or-dead analysis*, which tries to predict the status (alive or dead) of a group of stones on the board. Patterns from human games are extracted as training examples. Since this is a static pattern recognition problem (patterns are

independent with each other and have no temporal order), supervised learning is used.

Chapter 4

Deriving a New Updating Rule to Apply TD Learning in Multi-layer Perceptron

In this chapter, we will derive a new updating rule for applying temporal difference (TD) learning in a class of neural networks called *multi-layer Perceptron* (MLP) [7] [8]. Before the derivation, we will give a brief introduction to MLP for readers who are not familiar with neural networks. Then we will present our derivation. After that, we will describe the training algorithm which is designed according to the new learning rule [8].

4.1 Multi-layer Perceptron (MLP)

General Architecture of an MLP Multi-layer Perceptron (MLP) is an important class of neural networks. An MLP consists of many processing elements called *neurons*. The neurons are arranged into layers and the layers are ordered. The layer that accepts input signal for the whole network is called the *input layer*. The layer that produces the network output signal is called the *output layer*. All layers in between are called *hidden layers*. An MLP should have at least one hidden layer. Figure 4.1 shows an example of MLP with two hidden layers. The architecture of the network is 4-4-5-3 which represents that there are 4 neurons in the input layer, 4 neurons in the first hidden layer, 5 neurons

in the second hidden layer and 3 neurons in the output layer.

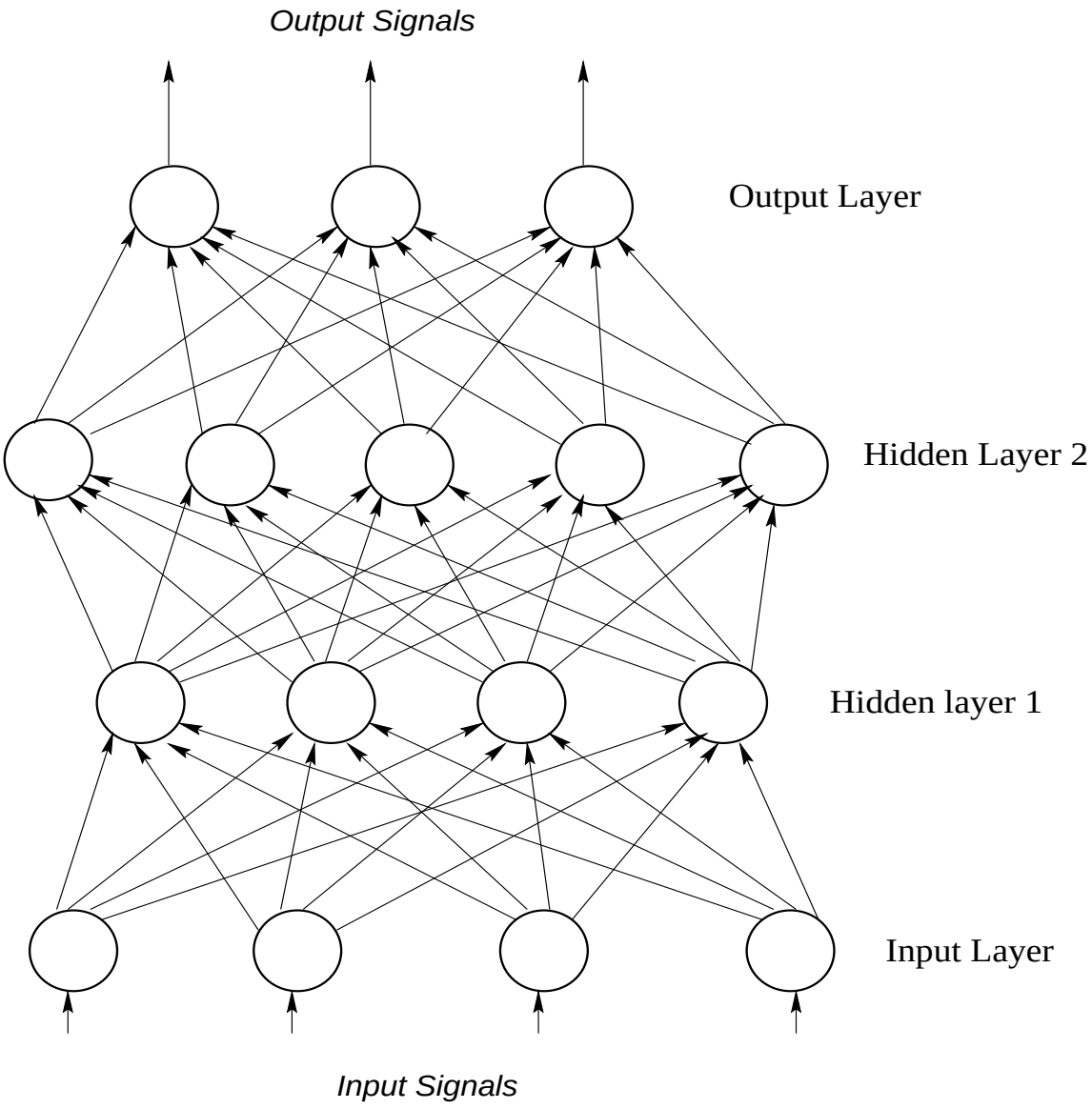


Figure 4.1: A multi-layer Perceptron with 2 hidden layers.

Signal Propagation inside an MLP Connection in an MLP only exists between neurons from two adjacent layers. Each connection between two neurons is associated with a weight, which is a real number that is multiplied to the signal that propagates up the connection from the lower layer to the upper layer. When an MLP receives an input

signal from its input layer, it propagates the signal up from its input layer, through one or more hidden layers, to its output layer and produces an output signal.

Training an MLP MLP is usually trained by *supervised learning*. During training, the network has to learn a set of training examples, with each example in the form of (X, O) where X is the input vector and O is the output vector expected from the network for X. After the network is properly trained for a problem by a set of examples, it can *generalize* by producing results with a good accuracy for examples that it has not encountered during training. There are some distinctive properties that characterize the problem solving capability of an MLP [17] :

1. Each neuron includes a *nonlinearity* at the output end. The nonlinearity is achieved by a *differentiable transfer function*[1], as opposed to the hard-limiting used in simple Perceptron.

2. The network contains *one or more layers of hidden neurons* which enable the network to learn complex tasks by extracting more meaningful features from the input vectors progressively.

3. The network exhibits a *high degree of connectivity*. In most cases, connectivity is *complete* between two adjacent layers of neuron. Each connection in the network is associated with a synaptic weight, which is the parameter to be tuned during training.

MLP's have been successfully applied to solve diverse problems after being trained in a supervised manner with a popular algorithm called *error back-propagation algorithm* or simply *backpropagation* [22]. Development of the algorithm represents a "landmark" in neural networks as it provides a computationally efficient method for the training of MLP.

[1]Transfer function is applied to the weighted sum of a neuron's input signals to produce that neuron's own output signal. A very common example of differentiable transfer function used in MLP is the *logistic function* :

$$f(n) = \frac{1}{1 + \exp(-n)}$$

which exhibits the sigmoidal nonlinearity.

4.2 Derivation of TD(λ) Learning Rule for MLP

4.2.1 Notations

Before we come to the derivation, let us define the following notations which we use in our derivation. For an MLP with k hidden layers :

layer 0	denotes the input layer.
layer i	denotes the corresponding i-th hidden layer, where $i = 1, \ldots, k$.
layer $k+1$	denotes the output layer.
$s(l)$	denotes the size of layer l (i.e. number of neurons).
M^T	represents the transpose of the matrix M.
w_l	represents the matrix of weights for connection between layer $(l-1)$ and layer l. It has a dimension of $s(l-1) \times s(l)$.
$[w_l]_{i,j}$	represents the specific weight of connection between the i-th node in layer $(l-1)$ and the j-th node in layer l.
$f_l(\cdot)$	is the transfer function used in layer l.
net_l	is the weighted-sum in layer l, which is equal to $w_l^T \cdot y_{l-1}$.
y_l	is the vector of output from layer l, which is equal to $f_l(\text{net}_l)$.

All vectors are column vectors unless stated otherwise.

4.2.2 A New Generalized Delta Rule

The *generalized delta rule* is a supervised learning rule for MLP [22]. We will make a small change to the original generalized delta rule to obtain a *new generalized delta rule*, which will be useful in the future derivation of updating rule for TD learning.

Original Delta Rule The original generalized delta rule for increment of a specific weight in an MLP [22] is:

$$[\Delta w_l]_{i,j} = \eta \, [y_{l-1}^T]_i \, [\delta_l]_j$$

where Δw_l is the matrix of weight increment of w_l. It has the same dimension as w_l,

η is the learning rate (a scalar),

δ_l is the $s(l) \times 1$ vector of error backpropagated from layer l to layer $l-1$.

The vector of backpropagated error, δ_l, is defined differently for output layer and for hidden layer :

$$\delta_l = \begin{cases} (T - P) \otimes f_l'(\text{net}_l) & \text{if } l \text{ is an output layer} \\ f_l'(\text{net}_l) \otimes w_{l+1}\delta_{l+1} & \text{if } l \text{ is a hidden layer} \end{cases}$$

where $\otimes$ denotes the element-wise vector multiplication

($[a_i] \otimes [b_i]$ gives $[a_i b_i]$ if a and b are 2 vectors of the same size.),

$f_l'(\cdot)$ denotes the derivative of the transfer function $f_l(\cdot)$,

T is the vector of desired output in supervised learning,

P is the output vector with length $|T|$.

From the equation, we notice that the calculation of delta at layer l, δ_l, requires the delta of its upper layer, layer $(l+1)$. As a result, the values of every delta should be computed in order : starting from the output layer down to the input layer. The delta computed in one layer should then be propagated back to its lower layer. This process continues until all values of delta in the network are computed. This algorithm for calculating weight increment in MLP is therefore called *backpropagation*.

New Delta Rule Here, we try to extract the term $(T - P)$ from the original delta, δ_l to obtain a *new delta*, δ_l^*. i.e.

$$\delta_l = \delta_l^* \cdot (T - P).$$

It can be observed that the new delta, δ_l^*, is a $s(l) \times |T|$ matrix. Like the original delta, the new delta is also defined differently for output layer and for hidden layer.

For the **output layer** $k+1$, it is defined as

$$\delta_{k+1}^* = \textbf{diag}\ [f_{k+1}'(net_{k+1})]$$

where $\mathbf{diag}(v)$ is a diagonal matrix with the diagonal is equal to v.[2]

For a **hidden layer** l, the new delta is defined as :

$$\delta_l^* = f_l'(\text{net}_l) \otimes w_{l+1} \cdot \delta_{l+1}^*$$

where $\otimes$ is an operator of element-wise multiplication applying on the vector $f_l'(\text{net}_l)$ and every column of $(w_{l+1}^T \cdot \delta_{l+1}^*)$.[3]

The equation for weight increment with this new delta will therefore be :

$$\begin{aligned} [\Delta w_l]_{ij} &= \eta\, [y_{l-1}]_i\, [\delta_l]_j \\ &= \eta\, [y_{l-1}]_i\, [\delta_l^*]_{j,:} \cdot (T - P) \\ &= \eta\, (T - P)^T \cdot ([y_{l-1}]_i\, [\delta_l^*]_{j,:})^T \end{aligned} \tag{4.1}$$

where $[\delta_l^*]_{j,:}$ is the j-th row vector of δ_l^* and

$[y_{l-1}]_i$ is the i-th element in vector y_{l-1} (a scalar).

This is the *new generalized delta rule* which will be used in the derivation in next section. Unlike the original delta, the new delta, δ_l^*, for layer l is a $s(l) \times |T|$ matrix instead of a vector. Moreover, the error between the desired and actual output, $(T - P)$, is required for calculating every weight increment.

[2]For example, $\mathbf{diag}(x)$, where $x = [x_1, x_2, \ldots, x_n]^T$, will be :

$$\begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

[3]In this case, $\otimes$ applies on a vector and a matrix (instead of 2 vectors). The operation it performs is :

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} = \begin{bmatrix} a_1 b_{11} & a_1 b_{12} & \cdots & a_1 b_{1m} \\ a_2 b_{21} & a_2 b_{22} & \cdots & a_2 b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_{n1} & a_n b_{n2} & \cdots & a_n b_{nm} \end{bmatrix}$$

4.2.3 Updating rule for TD(λ) Learning

If we compare two equations for weight increment

1. *prototypical supervised learning* (in Eq. 3.1) :

$$\Delta w(t) = \eta(\chi - P(t))\nabla_w P(t)$$

2. *new generalized delta rule for MLP* (in Eq. 4.1) :

$$[\Delta w_l]_{i,j} = \eta\,(T-P)^T([y_{l-1}]_i\,[\delta_l^*]_{j,:}])^T$$

We obtain the gradient term, i.e. the partial derivative of the network output with respect to a particular weight, for MLP :

$$\nabla_{w'} P = ([y_{l-1}]_i[\delta_l^*]_{j,:})^T.$$

If we consider the problem as a delay-reward prediction problem in which the input data come in a temporal sequence, the prediction P as well as y and δ are varying with the time t. We can rewrite the equation as :

$$\nabla_{w'} P(t) = ([y_{l-1}(t)]_i \cdot [\delta_l^*(t)]_{j,:})^T \tag{4.2}$$

where w' is a short form of $[w_l(t)]_{i,j}$ that represents a particular weight in the network at time t.

Recall the equation for TD(λ) Learning (in Eq. 3.5) :

$$\Delta w(t) = \eta\,(P(t+1) - P(t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P(k).$$

By substituting the expression of $\nabla_w P(t)$ in equation 4.2 and after some appropriate variable renaming, we obtain the weight increment for TD(λ) learning in MLP :

$$[\Delta w_l(t)]_{i,j} = \eta\,(P(t+1) - P(t))^T \sum_{k=1}^{t} \lambda^{t-k} ([y_{l-1}(k)]_i \cdot [\delta_l^*(k)]_{j,:})^T \tag{4.3}$$

where subscripts i and j denote the positions of elements in a matrix, l and $l-1$ denote the layer numbers, t denotes the time for the prediction and the corresponding weight

increment. The summation term in equation 4.3 is called the *history vector*, denoted by $[h_l(t)]_{ij}$. A history vector is associated for a particular weight in the network. It is a vector with length $|P(t)|$.

Equation 4.3 is a new updating rule of TD(λ) learning in MLP where $\lambda \in [0, 1]$. A training algorithm for this learning rule will be designed in next section.

4.3 Algorithm of Training MLP using TD(λ)

4.3.1 Definitions of Variables in the Algorithm

Each training example in TD learning is in the form (x_1, x_2, ..., x_n, o) where (x_1, x_2, ..., x_n) is a sequence of input vectors in temporal order. From each input vector x_i, which can be regarded as an observation made at time i, the network is supposed to make a prediction for that observation. The value of o is the target of all predictions. It is available after the n-th prediction has been made.

Assuming that the network to be trained is a fully-connected MLP with k hidden layers.[4] The size of each layer is $s(0)$, $s(1)$, ..., $s(k+1)$ respectively. Here are the representation of some variables :

1. y_0 is the input vector presented to the network,
2. y_l is the vector of output of layer l,
3. w_l is matrix of weight connecting layer $l-1$ and l,
4. b_l is vector of biases connecting to layer l,
5. η is the learning rate,
6. λ is the parameter in TD (λ) learning,

[4]For a fully-connected MLP, connections between neurons in two adjacent layers are *complete*. This means that there is connection between every two neurons from two adjacent layers respectively.

where $l = 1, \ldots, k, k+1$.

In practice, each neuron in an MLP will have a *bias*. It is a value assigned to a *fixed* connection to the neuron : a connection that has a constant signal of 1 at all time. The function of a bias is like a variable threshold value because the value of a bias will change during training. The vector b_l here represents the biases for all neurons in layer l. It is therefore a vector of length $s(l)$.

Aside from the above variables in the original backpropagation algorithm, we need some more storage for the following values during training :

1. a history vector $[hw_l]_{i,j}$, of length $s(k+1)$, for each synaptic weight $[w_l]_{i,j}$,
2. a history vector $[hb_l]_j$, of length $s(k+1)$, for each bias $[b_l]_j$,
3. a vector of prediction of previous observation, P_{prev}.

At each layer l, the *history matrix* of weight hw_l denotes a matrix of same dimension as the matrix w_l. Similarly, the history vector of bias hb_l denotes a vector of the same dimension as the vector b_l. Each element in hw_l and hb_l is a vector with length equal to the size of network output, $s(k+1)$.

Moreover, during training, some temporal storage is needed to store δ_l^* for each layer l, where $l = 1, \ldots, k+1$. δ_l^* is a $s(l) \times s(k+1)$ matrix which stores the error backpropagated from layer l to layer $l-1$.

We will present our training algorithm in next section. The description of the algorithm can be found in the section afterwards.

4.3.2 Training Algorithm

1. **If** `this is the first time of training`

   `Initialize weights` $w_1, w_2, \ldots, w_{k+1}$ `and`

   `biases` $b_1, b_2, \ldots, b_{k+1}$ `by small random values.`

   else

Adopt weights $w_1, w_2, \ldots, w_{k+1}$ and

biases $b_1, b_2, \ldots, b_{k+1}$ from last training.

2. **while** (error of training has not dropped below expected value)

for each training example in the form of $(x_1, x_2, \ldots, x_n, o)$:

(a) Initialize *history vectors* and *previous prediction*:

$P_{prev} = \vec{0}$

for $l = 1$ to $k+1$ **do**

for $j = 1$ to $s(l)$ **do**

for $i = 1$ to $s(l-1)$ **do**

$[hw_l]_{i,j} = \vec{0}$

for $l = 1$ to $k+1$ **do**

for $j = 1$ to $s(l)$ **do**

$[hb_l]_j = \vec{0}$

(b) **for** t = 1 to n **do**

i. Feed x_t into layer 0 of the network : $y_1 = x_t$

ii. Feed-forward :

for $l = 1$ to $k+1$ **do**

$y_l = f(y_{l-1}^T w_l + b_l)$

$P_t = y_{k+1}$

iii. Update weight and bias:

for $l = 1$ to $k+1$ **do**

for $j = 1$ to $s(l)$ **do**

for $i = 1$ to $s(l-1)$ **do**

$[w_l]_{i,j} = [w_l]_{i,j} + \eta\,(P_t - P_{prev})^T \cdot [hw_l]_{i,j}$

for $l = 1$ to $k+1$ **do**

for $j = 1$ `to` $s(l)$ **do**

$$[b_l]_j = [b_l]_j + \eta\,(P_t - P_{prev})^T \cdot [hb_l]_j$$

iv. `Compute the values of delta by backpropagation :`

$$\delta^*_{k+1} = \mathbf{diag}(f'(y_{k+1}))$$

for $l = k$ `to 1 step` -1 **do**

$$\delta^*_{l-1} = f'('(y_{l-1})) \otimes w_l \cdot \delta^{*T}_l$$

v. `Update history vectors (for both weight and bias) :`

for $l = 1$ `to` $k+1$ **do**

for $j = 1$ `to` $s(l)$ **do**

for $i = 1$ `to` $s(l-1)$ **do**

$$[hw_l]_{i,j} = [\delta^*_l]_j\,[y_{l-1}]_i + \lambda[hw_l]_{i,j}$$

for $l = 1$ `to` $k+1$ **do**

for $j = 1$ `to` $s(l)$ **do**

$$[hb_l]_j = [\delta^*_l]_j + \lambda[hb_l]_j$$

vi. `Update` *previous prediction* :

$$P_{prev} = P_i$$

(c) `Update weight and bias for target` o `(after` n `observations):`

for $l = 1$ `to` $k+1$ **do**

for $j = 1$ `to` $s(l)$ **do**

for $i = 1$ `to` $s(l-1)$ **do**

$$[w_l]_{i,j} = [w_l]_{i,j} + \eta\,(o - P_{prev})^T \cdot [hw_l]_{i,j}$$

for $l = 1$ `to` $k+1$ **do**

for $j = 1$ `to` $s(l)$ **do**

$$[b_l]_j = [b_l]_j + \eta\,(o - P_{prev})^T \cdot [hb_l]_j$$

4.3.3 Description of the Algorithm

The training algorithm is used to train a set of training data, each element of which contains a temporal sequence of input data and a target.

In `(1)`, initialization of the MLP is performed. If this is the first time of training, all weights and biases of the network will be randomly initialized. Otherwise, weights and biases from the previous training will be adopted.

Codes in `(2)` perform the training procedure. An *epoch* of training is completed when the whole set of data are presented to the network once. Usually, many epochs have to be performed before the network can produce acceptable results for a specific training data set. The stopping criterion for training is based on the summed square error (SSE) between the network's outputs and the target for each training data.

In `2(a)`, previous prediction and history vectors are initialized before a new training sequence is presented into the network.

In `2(b)`, each input vector of the sequence is passed to the network for training, as in `(i)`. Training mainly consists of *four* stages :

1. **Feed-forward** `(ii)` : The input vector is fed into the input layer. The signals are passing forward and finally to the output layer in the network. An output vector, which represents a prediction made by the network, is produced.

2. **Weight Update** `(iii)` : Using the difference between the current and the previous predictions, together with the history vectors, all weights and biases of the network are changed.

3. **Backprop** `(iv)` : Delta's for the current prediction are computed for the output layer. The values of delta are back-propagated to the lower layers one after another until delta for every layer is available. The operator $\otimes$ performs an element-wise multiplication [5] between a vector and every column of a matrix, which are of the same length.

[5] Refer to the previous section for more description.

4. **History Update** `(v)` : Using the the values of delta just computed, history vectors for all weights and biases in the network are updated for the current prediction.

After the above four stages, the variable storing the past prediction, P_{prev} is updated by the current prediction, which is done in `(vi)`.

In the codes of `2(c)`, after all input vectors in sequence is passed into the network, finally the target of all predictions, which is the reinforcement value, is used to compute the final update of the weights and biases.

The codes in `2(a), 2(b)` and `2(c)` will iterate in the for-loop for every training data in the training set. The codes inside the while-loop in `2` will perform a whole epoch of training. It will iterate until the training error drops below the expected limit.

Chapter 5

Experiments

5.1 Introduction

In this chapter, we present the designs and results of some experiments for applying neural networks in Go. Three sets of experiments have been performed to train multi-layer Perceptrons (MLP) using supervised learning and TD learning to solve some specific problems in computer Go.

In the first set of experiment, we try to apply TD (λ) learning and MLP's to evaluate 7×7 Go board positions. The networks are trained by self-playing with stochastic selection of moves. Different values of λ are used and their performance is compared.

In the second set of experiments, we train MLP's to evaluate 9×9 board positions in the Middle-game. Temporal sequences of board positions, which are extracted from a set of *human games*, are used as training examples. We investigate the effect of the length of the training sequences and the value of λ to the network's generalization performance after training. Results of the experiments are different with our original expectation.

In the last experiments, we try to train MLP's to determine the life status of a group of stones by supervised learning. Local patterns from human games are extracted as training examples and supervised learning model is used. Accuracy for the alive-or-dead classification after training is high. This result is very encouraging which gives a good prospect for similar applications in this area.

5.2 Experiment 1 : Training Evaluation Function for 7×7 Go Games by TD(λ) with Self-playing

5.2.1 Introduction

The objective of this experiment is to test the performance of different values of λ in TD(λ) learning when it is applied to solve the problem of board evaluation in Go. From the past application of TD learning in board evaluation of Go [24], only TD(0) has been employed. As a result, we want to investigate the effect of different λ in training by using the *new algorithm for TD learning using MLP* which has been derived in the last chapter.

We train multi-layer Perceptrons (MLP's) by TD(λ) learning and self-playing to evaluate 7×7 Go board positions. The 7×7 board size, instead of the standard 19×19, is used in order to reduce the complexity of the problem. We have trained several networks by different values of λ using TD (λ) learning and their performance are compared [8].

5.2.2 7×7 Go

Go has a favorable property that when the board size is reduced, all basic rules of the game still apply. Go with a smaller board size is excellent for beginners to handle and for computer to conquer because the complexity of the game is much smaller than the standard one.

With the complexity of the problem being reduced, we could use a smaller network in our experiments. Using the same raw representation scheme, only 51 input units are needed to represent a 9×9 Go board while 363 units are needed for the standard 19×19 board. Reducing the number of input units can decrease the number of necessary hidden neurons and thus the size of the whole network. Consequently, time and storage needed for the computation in training can be reduced to a manageable level.

Node	Information Stored	Representation
1-49	49 points in a 7×7 Go board	$+1$ - black stone 0 - empty -1 - white stone
50	Color of next move	$+1$ - black -1 - white
51	number of prisoners	[(no. of prisoners captured by black) - (no. of prisoners captured by white)] $\times \frac{1}{10}$

Table 5.1: Input representation of a 7×7 board in experiment 1

5.2.3 Experimental Designs

Network Architecture Multi-layer Perceptron with 1 hidden layer is used. The architecture of the network is 51–30–1.

Representation Schemes We use a raw representation for a 7×7 Go board. The information encoded in the 51 input nodes is illustrated in Table 5.1. Output of the network represents the evaluation of the board. The output value lies within the continuous range $[-1, 1]$, with -1 representing an absolute win for the white side and $+1$ representing an absolute win for the black side. Different values between -1 and $+1$ represents different degree of favorableness each side correspondingly.

Self Playing Since a network is trained by *self-playing* (playing against itself), we do not need to supply any training example to the network in this experiment. During self-playing, the network evaluates board positions after each legal move and plays according to the evaluations obtained. This is basically a 1-ply minimax search. Gibbs sampling [24] is used to select move stochastically from all legal moves. The probability of a move m_i being selected is calculated by :

$$P(m_i) = \frac{1}{Z} \exp\left(\frac{E(B_i)}{T}\right) \tag{5.1}$$

where B_i is the board position after the move m_i,

Z is a normalizing factor equal to $\sum_i \exp(\frac{E(B_i)}{T})$ for B_i after each legal moves m_i,

$E(B_i)$ is the evaluation of board position B_i made by the network,

T is the temperature determining the degree of randomness.

Totally 100,000 games have been played in self-training of each network. In the beginning of training, temperature T is set to a high value (1.0). When the training proceeds, T is reduced by a factor of α for every 20 games. The reduction is done by : $T_{\text{new}} = \alpha\, T_{\text{old}}$, where $0 < \alpha < 1$. The factor α is set to 0.92 in this experiment.

5.2.4 Performance Testing for Trained Networks

Several networks are trained by TD(λ) with $\lambda = 0.00, 0.25, 0.50, 0.75$ and 1.00 respectively. We try to compare performance of any two networks trained by different λ by letting them play against each other for 2000 testing games. Inside each testing game, the criterion for selecting a move is similar to the case in the training games. To avoid two networks from playing the same sequence of moves in all the 2000 testing games, stochastic Gibbs sampling with a very low temperature is used for move selection. All networks trained with different λ compete in a *round-robin tournament* for performance testing.

To obtain more reliable results, *four replications* have been performed. In each replication, the whole set of experiments starts from the beginning and the networks are initialized with a new set of random weights. We obtain the final results from averaging results of the four replications.

After testing performance with different λ, overall performance of the trained networks is estimated by the author, who is an expert Go player.

5.2.5 Results

Table 5.2 shows the summary of the outcomes of the testing games. Each entry of the table represents the winning percentage in the 2000 testing games between two different networks. For example, from the entry in the fifth row and the first column, we know

Networks $\lambda =$	Average Winning Percentage with Opponents :					
	$\lambda = 0.00$	$\lambda = 0.25$	$\lambda = 0.50$	$\lambda = 0.75$	$\lambda = 1.00$	relative score
0.00	—	46.11%	43.58%	37.38%	31.68%	161.74
0.25	53.89%	—	48.26%	43.41%	37.97%	183.53
0.50	56.43%	51.74%	—	46.19%	40.74%	195.09
0.75	62.63%	56.59%	53.81%	—	45.41%	228.44
1.00	68.32%	62.03%	59.26%	54.59%	—	244.20

Table 5.2: Results showing performance of networks trained by TD(λ) with different λ

that the network trained by $\lambda = 1.00$ wins 68.32% of the 2000 testing games against the network trained by $\lambda = 0.00$.

We sum up all the winning percentages for each network to obtain its *relative score* which reflects the accuracy of the network's predictions. If a network has a high relative score, it wins more games against other networks and is therefore a more accurate position evaluator.

From the table, we can see that *relative score increases with the value of* λ. This shows that static board evaluation of Go game will be more accurate if we put more emphasis on the past state information in evaluation.

On the other hand, *performance of the evaluator after training is still practically inaccurate*, an assessment made by an expert player, who is the author of this thesis. As a result, the moves played by the networks are usually of low-grade. This indicates that we need special design and training strategy for self-training an accurate evaluation function for Go board positions.

5.2.6 Discussions

The results of this experiment can be summarized as

1. *Better performance obtained with larger value of* λ. This is an empirical relationship between the value of λ and the performance of evaluation of Go board position. However, it is not obvious that which property of the Go game contributes to this relationship because of the limited knowledge on the problem of Go board evaluation.

Moreover, although TD(1) learning has a close resemblance with supervised learning, they are not the same as *immediate updating mode* is used in this experiment. [1]

2. *Evaluation is not satisfactory through a simple design of self-training with stochastic sampling of moves* within the training that is performed in this experiment. Since we mainly focus on the effect for different λ, the design in this experiment is simple and may not be sufficient for training practically good evaluation.

Self-training in TD learning has been shown a good strategy in the non-deterministic game Backgammon. However, applying this strategy in deterministic game needs some special design because the exploration of state space in the game during self-training is not as efficient as in non-deterministic games. Gibbs sampling plus a temperature cooling scheme, which are techniques applied in *simulated annealing* [1], are used to overcome this difficulty. However, there is no theoretical proof that this stochastic sampling in move selection will give good state space exploration because the use of the techniques here is different from its original application. This is a problem that needs further analysis.

A recent publication demonstrates a research that applies an improved TD learning in a deterministic game called *Nine Men's Morris* [20], which is a converging perfect-information game. Its state-space complexity is 10^{10}, which is the smallest among the *Olympic List of games*.[2] (In contrast, the state-complexity of 7×7 Go is 10^{23} and 9×9 Go is 10^{38}.) The game has been solved in 1993 by Ralpg Gasser in a way that game-theoretic value for every board position is generated and stored in a large database [15]. This is in an even stronger state than *strongly solved* which was explained in Chapter 2.

Unlike Go, training networks to learn the evaluation function in Nine Men's Morris is simpler because exact values of all possible game positions are available. Therefore, some better training strategies can be applied. In the experiment, a re-learning method has been used to improve the performance of TD learning. This is a method that iterates

[1] The original TD(1) learning is derived from supervised learning using batch updating mode (refer to section 3.2.2).

[2] The Olympic list contains most of the common games we play. Every year, there is a tournament for programs of each game to compete with each other.

the learning for a specific temporal sequence of training examples until the network can give accurate predictions for all the examples.

It is difficult to apply this technique in Go possibly because the exact value of the board positions of Go is not available. The state space of Go board positions, even in 7×7, is huge when compared to Nine Men's Morris. Therefore, it is not likely to solve Go in the same way as Nine Men's Morris.

Finally, we can conclude that the study of applying TD learning and self-playing in deterministic games like Go is still in an early stage. In this experiment, we have found an empirical relation between λ and training performance. However, further studies are needed for a better understanding of the problem.

5.2.7 Limitations

The objective of this experiment is to find out the effect of different values of λ. As a result, the design is not sophisticated for training evaluation of very good performance. We may try to improve the performance of the overall networks by investigating and designing special architecture of network for extraction of some useful information from a board position. However, that is not in the scope of this experiment.

5.3 Experiment 2 : Training Evaluation Function for 9×9 Go Games by TD(λ) Learning from Human Games

5.3.1 Introduction

In this experiment, we train MLP's by TD (λ) learning using board positions extracted from human game records (i.e. training by given examples). This is a new way of applying TD learning because in the previous experiments, only self-playing has been used.

MLP's are trained to evaluate board positions taken from the Middle-game and early

End-game stage of 9×9 games played by human players. Each training example contains a sequence of board positions taken from the last few moves of a 9×9 Go game. Sequences of different length are used to train different networks and the performance of the networks are compared. It is expected that after training, the network can capture the strategic complexity in the Middle-game and produce good evaluation for the board positions.

5.3.2 9×9 Go game

9×9 Go game, like 7×7, is a simplified version of the standard 19×19 Go game. This simplification *does not change* the rules of the game. On the other hand, the complexity of the game and the problem of board evaluation has been reduced by a very large degree.

We choose 9×9 as the board size in this experiment because abundant training examples for 9×9 Go games are available. This is not the case for Go game of other similar sizes. As neural networks are trained by human games, the abundance of available training examples is an important factor for the success of the experiment.

In usual cases, Go game with smaller board size is only for beginners and weaker players because of its reduced complexity. However, fortunately, large amount of high-quality 9×9 game records are available because of the recent interest from the professional players in this game. It is believed that even with a 5.5 point *Komi* [3], which is a comparatively large amount when the board size is small, black should have an advantage. However, strategy leading to an easy win for black is still not obvious for the professional players. This result draws much interest from strong players in researching 9×9 Go games. In one of the Go servers on the Internet, there is a *ladder* tournament dedicated for 9×9 Go games. As a result, we are able to obtain large number of high quality 9×9 game records from that server.

[3] The compensation given by the black side to the white side aimed to eliminate the advantage of playing first.

5.3.3 Training Data Preparation

Source A few thousands of 9×9 Go game records are down-loaded from a Go server called *No Name Go Server* (NNGS)[4]. The games are mainly from the 9×9 ladder tournament [5].

Classification and Selection Normally, a game is finished in either one of the 3 conditions :

1. *resignation* : when one player resigns.
2. *time forfeit* : when one player exceeds his/her time limit.
3. *scoring* : when there are no more valuable moves and both player agree on endgame.

In this experiment, only games finished by resignation are used for extraction of training examples. The reasons for not selecting the others are :

1. A game finished by time forfeit is not used because the game outcome may not correctly reflect the state of the final board position. It is very often that the final board position is favorable to the side who lost the game by time forfeit.
2. A game finished by scoring is not used because of the nature of the final board position. Counting the size of territories, instead of pattern shape analysis, should be applied in determining the outcome of the final board position. That is not the expected goal for the network to solve in this experiment.

Only games with certain number of moves (20 to 50 moves) are selected. It is because if a game has too few moves, the final board position will remain in the opening stage. If a game has too many moves, the final board position will already be in the late endgame stage. Both of them will not give the type of training examples we need for this experiment.

[4]NNGS is a newer and free Go server. It is located at York College, City University of New York. Connection to NNGS can be made to **ra.york.cuny.edu** through port number **9696**. The archive of game records can be obtained through FTP at the same address.

[5]9×9 ladder is a ranking tournament of 9×9 Go game. Each participant will be given a position in the ladder. Winning will make ones position higher and vice verse.

Extraction of Board Position Sequences After a game record has passed the above selection processes, we extract its last k-th board positions as a training example (or a testing example). Assuming there are n moves in that record : $m_1, m_2, \ldots, m_n$ and the sequence of board positions after each move is : $B_0, B_1, B_2, \ldots, B_n$ (where B_i represents the board position after move m_i), the sequence of board positions to be extracted will be $B_{n-k+1}, B_{n-k+2}, \ldots, B_n$. The values of k used in this experiment are $1, 2, \ldots, 6$.

After this procedure, we obtain a set of board position sequences which may be used as training and testing examples for the TD learning experiment.

Generating More Examples by Reflections and Rotations A Go board is 90°-*rotational invariant* and *reflection invariant*. In other words, the content of a Go board is unchanged when it is rotated by 90° or reflected over diagonals, horizontal edges and vertical edges. For a board position B, let $R(B)$ be the board after clockwise rotation by 90° of B and and $F(B)$ be the reflection over the lower horizontal edge of the board. We can totally produced 8 different variations of the board B with the same contents by repeatedly apply the rotation and reflection to the board :

1. B
2. $F(B)$
3. $R(B)$
4. $F(R(B))$
5. $R(R(B))$
6. $F(R(R(B)))$
7. $R(R(R(B)))$
8. $F(R(R(R(B))))$

With such application, we can increase the size of the data set. Figure 5.1 shows the 8 variations of an example board position.

5.3.4 Experimental Designs

Network Architecture The model of multi-layer Perceptron with 2 hidden layers is used. The network architecture is 83-50-20-1.

Theoretically, MLP with one hidden layer is sufficient for universal function approximation [13]. However, there is no limit for the number of hidden neurons and the time of convergence given by the theory. Practically, networks with more than one hidden layer

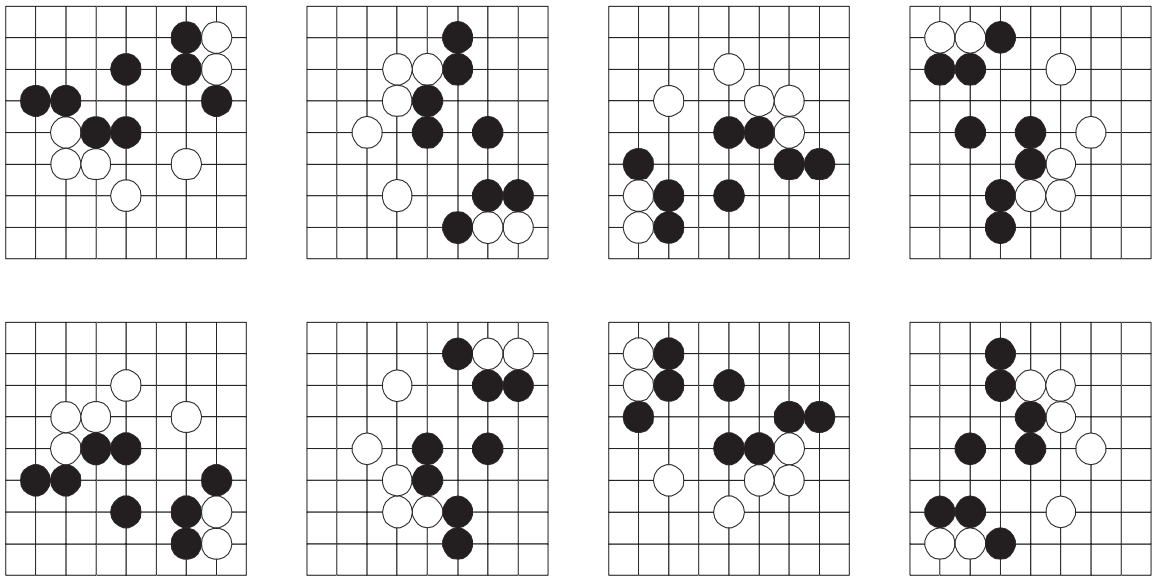


Figure 5.1: 8 ways of invariant transformation of a 9×9 Go board.

is often used for solving complicated problems. As a result, we try to use networks with two hidden layers instead of one.

Representation Schemes A Go board position is represented by 83 input units. 81 units are used to represents each position in the 9×9 Go board (-1 = white stone; 0 = empty; +1 = black stone). 1 unit is used to represent the color that plays next (-1 = white; +1 = black) and 1 unit is used to represent the number of prisoners by the value : [no. of stones captured by black - no. of stones captured by white] $\times \frac{1}{10}$.

Training and Testing Data Set The training set contains 600 different sequences of positions and the testing set contains 180 different sequences of positions. By using the 8-way transformation to produce more semantically equivalent board positions, we obtain $600 \times 8 = 4800$ training sequences and $180 \times 8 = 1440$ testing sequences.

Different Parameters Three sets of experiments have been trained by TD(λ) where $\lambda = 0.0, 0.5$ and 1.0 in each set respectively.

Measurement of Generalization Error Each testing example has the form (B_1, $B_2, \ldots, B_k$, O) where B_is are the last k-th board positions of a game and O is the outcome of the game. The outcome O only correctly reflects the evaluation of the final board position B_k, but not other board positions in the sequence. It is because some moves m_j contributing to the last k-th board transitions may be bad and largely change the value of the board positions. This phenomenon is called *volatility* of moves. In such cases, O will be a poor reflection of the board positions before the bad move m_j. To avoid such problems, we only use the final board position from each sequence for testing the generalization performance of the trained networks.

There are two measures of the generalization error. The first one is the conventional *Summed Square Error* (SSE) which is measured by $\sum_i (z_i - y_i)^2$ where $[z_i]$ is the vector of the expected output (Z in this experiment) and $[y_i]$ is the vector of the actual output from the network. This error measurement is useful in comparing performance of different training methods.

In the second measure, the output of the network is treated as a discrete decision. In our experiment, the network has two output values representing the decision of 'black wins" and 'white wins' respectively. The output of the network can be practically treated as a discrete decision by comparing the two output values. With such definition, a network can either predict correctly or wrongly for each testing example. We measure the total number and the percentage of incorrect predictions. This measurement is more significant in estimating the practical performance of the network.

5.3.5 Results

Table 5.3 shows the generalization error for different networks trained with different values of λ and different length of game sequence. Figure 5.2 and 5.3 are the graphical presentations of data in Table 5.3.

The column of *SSE* stores the summed square errors of the trained network on the testing data set. The column of *incorrect ratio* is the ratio of number of wrong prediction

game length	λ=0.0		λ=0.5		λ=1.0	
	SSE	incorrect ratio	SSE	incorrect ratio	SSE	incorrect ratio
1	0.2417	0.1333	0.2408	0.1333	0.2419	0.1333
2	0.4340	0.3049	0.3833	0.2632	0.3545	0.2444
3	0.3533	0.2132	0.3752	0.2556	0.3748	0.2278
4	0.4683	0.3208	0.3942	0.2785	0.3891	0.2778
5	0.4338	0.2944	0.3964	0.2667	0.3955	0.2674
6	0.4648	0.3444	0.4020	0.2868	0.3999	0.2986

Table 5.3: Generalization error after training in experiment 2

to total number of predictions. The network makes a wrong prediction for a training data when the discrete decision it makes (either black wins, draw or white wins) does not match the desired output. For both measures (SSE and incorrect ratio), the smaller the value is, the better the performance is.

From both graphs, we can observe trends showing that performance of the network degrades when length of the training games increases. The best performance is obtained when game length = 1, in which TD learning is reduced to supervised learning because each training example contains only one board position and each pattern is treated independently. This is a surprising result because we expect that prediction made by the network will become more accurate when the length of sequences of training board positions increases.

The performance for different values of λ agrees with the trend in experiment 1. Better performance is obtained with greater value of λ. However, since TD learning does not seem to be a good model for this application, this relationship between λ and performance is not important.

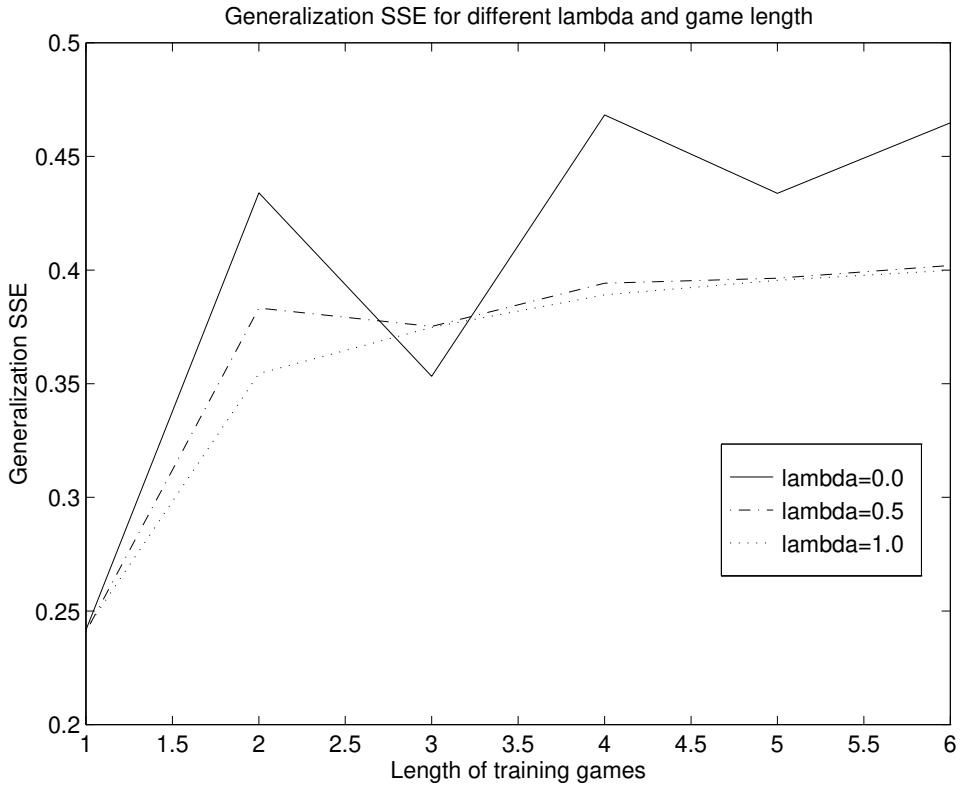


Figure 5.2: Generalization summed square error (SSE) in experiment 2

5.3.6 Discussion

Relationship with Length of Game Sequences

The result of this experiment is different from expected. We expect TD learning from temporal sequences of board position, extracted from human game records will produce better result than learning from a set of independent board positions. However, the empirical results support the opposite : *TD learning from sequences of board positions of a game performs worse than learning from independent board positions. The longer the sequences, the poorer the result.*

We shall understand this phenomenon from the characteristics of TD learning. TD learning is superior over conventional supervised learning by giving more accurate estimation of certain important statistical parameters and a more detailed modeling of the

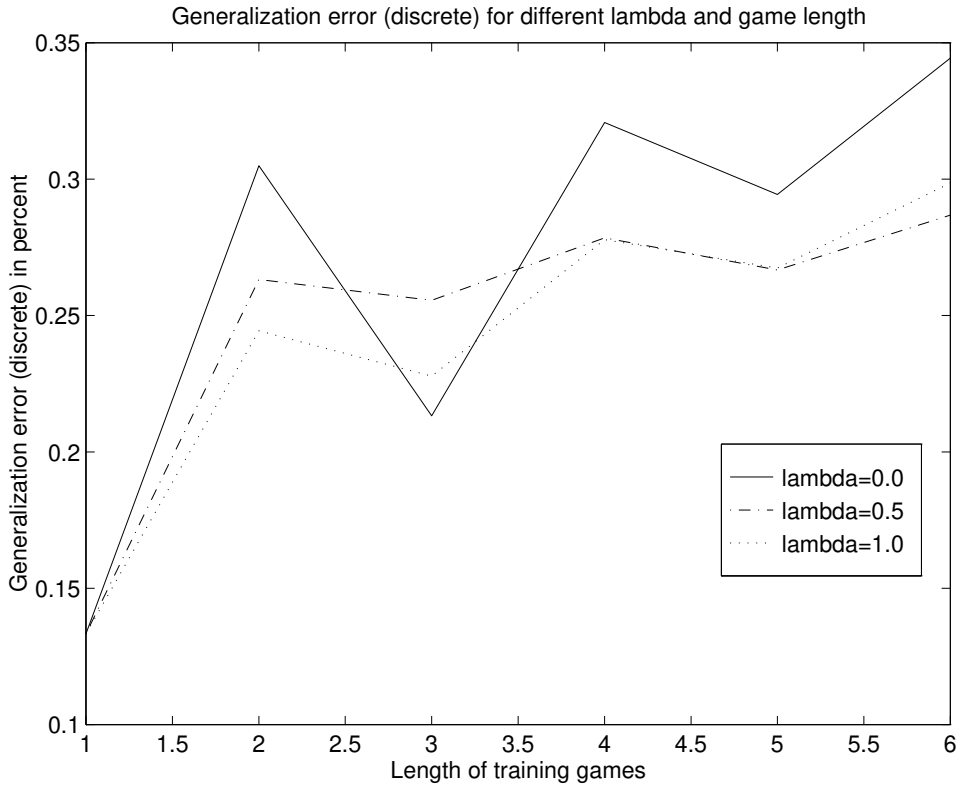


Figure 5.3: Generalization error measured by percentage of incorrect prediction

underlying probabilistic model. The superiority holds when the system to be estimated is a first order Markov model [3]. The difference of performance of TD learning in Backgammon and in Go suggests that determinism of the game is an important factor to the effectiveness of TD learning. Since transition of state of board positions is probabilistic in Backgammon, state transition probability from the current state to the terminal state is a good estimator of the game-theoretic value of current state. As a result, the problem of Backgammon can be modeled by the Markov model and solved effectively by TD learning. In contrast, state transition in Go is deterministic. TD learning may not train a good estimator because the value of a state cannot be estimated by the transition probability to the terminal states.

The reason for worse performance with longer sequences of board position is because of

the inability of modeling the probabilistic models from the limited training data. With self-playing and a good scheme for state space exploration, TD learning is able to understand the probabilistic model of the state space transitions. However, if the training examples are supplied and fixed during training, TD learning could only model the probabilistic model with the existing data set. This will result in poor estimation as the existing sequences do not provide thorough and even exploration of the state space.

In our experiment, when we increase the length of the board position sequences, the complexity of the problem increases. This is because the network is required to make predictions on board positions from the earlier stage of a Go game. The modeling of a more complex space with the fixed set of training data is worse, explaining the results obtained from the experiment.

Best Performance of network after Training

Performance is obtained when the length of each training sequence is equal to 1. The accuracy of prediction obtained from this training is over 85% (13.3% incorrect predictions), which is a practically acceptable figure. When we consider the number of training examples available in this experiment, this accuracy is not bad because the problem has a very high complexity. (Complexity of the state space is of the order 10^{38} but the size of training set is of the order of 10^3 to 10^4) This is an encouraging result that shows a good feasibility of using supervised learning model for solving the problem, even though TD learning does not give a satisfactory result.

5.3.7 Limitations

The number of training examples is limited and the performance of TD learning with more fixed training examples cannot be tested. With more training examples available, the performance can be expected to improve.

5.4 Experiment 3 : Life Status Determination in the Go Endgame

5.4.1 Introduction

Beside using TD learning, we may also apply the supervised learning model in solving some specific sub-problems in Go. Usually, problem that requires analysis of shape of static patterns is suitable to be solved by supervised learning in which training examples are learned separately and independently. However, research for solving problems in computer Go using this approach has not been done before.

We try to tackle the problem of alive-or-dead analysis in this experiment. It is not only an important sub-problem for designing a evaluation function of Go (recall section 2.3.1 for a detailed explanation) but also a representative problem that requires static shape analysis.

Training by supervised learning usually requires plenty of training examples. Therefore, the *availability* and *ease of preparation* of sufficient examples are two important aspects in designing experiments using neural network. In our experiment, we obtain large amount of records of human games from a Go server in the Internet. Moreover, training examples are produced by extracting corner patterns from End-game positions of a Go game automatically. (Some human selection process is needed for refining the training set but the process can be eliminated after some further change of design discussed in this section) As a result, there is no practical problem in setting up the training experiments.

We have trained MLP's with multiple hidden layers to learn the problem of life-status identification. The network have to judge whether a group of stones residing at the corner region is alive or dead. After training with a data set of reasonable size, the networks can perform accurate identification. This encouraging result gives a very good prospect for applying neural network to solve sub-problems in Go with the similar nature as this problem.

5.4.2 Training Data Preparation

Data Source We down-load thousands of game records from the *Internet Go Server* (IGS) [6]. Each game record stores a 19×19 game played by amateur Go players in the server.

Classification and Selection The large number of games are first classified by the *rank of their players*. Only games played by *certified high-ranked*[7] players (above 3-*kyu*) are used in this experiment. This is for maintaining the quality of information supplied by the games because weak players make mistakes in judging the life status of a group of stones more easily.

The qualified games will be further classified by the *finishing method*. As mentioned in experiment 2, a game can be finished by *resignation*, by *time forfeit* or by *scoring*.

For the games ended by scoring, alive-or-dead information of every stone on the End-game position is available from the game record. Such information is absent in the games ended by resignation and by time forfeit. Therefore, only game finished by scoring will be used for extracting training examples.

Pattern Extraction The process of pattern extraction from game records can be divided into the following four steps :

1. The game is replayed from the game record by a computer until the final board position is reached.

2. From the information in the game record, life status of every stone on the final board position is assigned.

[6] IGS is a Go server in the Internet. It is a site for Go players from all around the world to connect and find opponent to play Go. Connection to IGS should be made to **igs.nuri.net** through port number **6969**. The game records used in this experiment are obtained through FTP from the same address.

[7] The ranks of amateur Go players are divided into 2 classes : *kyu (k)* and *dan (d)*. The order is like this :

$$\text{(beginner) } 25k \rightarrow 24k \rightarrow \cdots \rightarrow 2k \rightarrow 1k \rightarrow 1d \rightarrow 2d \rightarrow \cdots \rightarrow 6d \text{ (expert)}$$

6-dan is the highest amateur rank. Professional players are ranked by a different system. Currently, the strongest computer Go programs have ranks around the level of 10-kyu to 5-kyu.

3. Stones of the same color connecting together are arranged as group.

4. If a group of stone is residing with any one of the 7×7 corner of the board, pattern of the corresponding 8×8 corner region is extracted. This extraction process is illustrated in Figure 5.4.

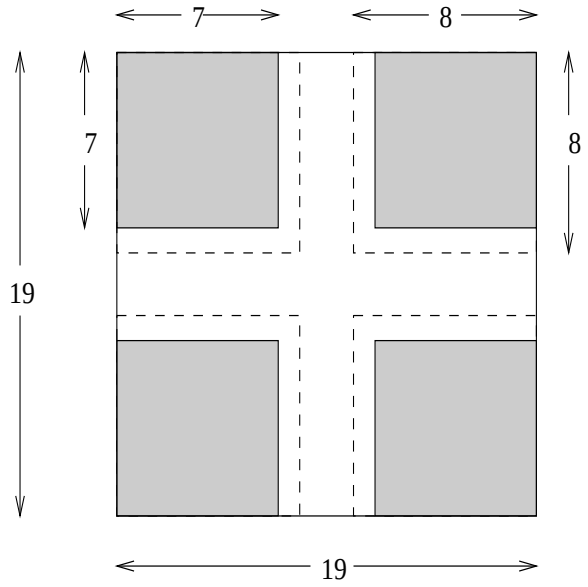


Figure 5.4: Extraction of a 8×8 corner pattern : if a group of stones reside in either one of the 7×7 corner (marked by gray), the corresponding 8×8 corner pattern (marked by dotted lines) will be extracted as a training example

After passing the above four steps, a set of corner patterns are obtained from the game records. Information stored in each extracted corner pattern is :

1. stone at each point of the 7×7 pattern (black, white or empty).

2. locations of the group whose life status needs to be identified.

3. life status of the group marked in (2).

The representation of each training data is then standardized. First, the corner point of the pattern should be at the upper left (i.e. uniform orientation). Patterns not at

this orientation are rotated. Second, color of stones in the group to be examined by the network is black in color (i.e. uniform stone-color). Color reversal[8] is applied on patterns that do not satisfy this condition. These two processes (rotation and color reversal) will not change the actual content of the pattern. However, complexity of the problem can be reduced with this standardization of pattern representation.

Expert Selection Inside a 8×8 pattern, there *may not* be sufficient information for determining whether a group residing there is alive or dead. Some necessary information located outside the 8×8 corner region may be lost during the extraction process. Therefore, to provide a good set of training data, such patterns should not be included in the training set. In this experiment, we need inspection of human expert[9] to find out and discard the 8×8 corner patterns that do not contain sufficient information.

5.4.3 Experimental Designs

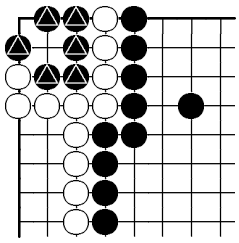
Network Architecture Multi-layer Perceptrons with 4 hidden layers are used. The architecture is 128–150–70–40–30–2. The first two hidden layers are aimed for extraction of *eye* patterns, which is assisted by the specially designed connectivity in the first hidden layer. The last two hidden layers aimed for high level analysis using the information extracted.

Representation Schemes 128 input units are used to represent two 8×8 maps for each training example. The first 8×8 map (Board Map) represents the state of each point in the pattern. For each point, -1 represents a white stone, 0 represents a empty point and $+1$ represents a black stone. The second 8×8 map (Location Map) locates the group of stones for the alive-or-dead test. Since there may be more than one group of stones of the same color inside the 8×8 corner, this extra map is needed to indicate to the network

[8]Color reversal is performed by changing color : black $\rightarrow$ white and white $\rightarrow$ black for every stone one the specified region of the board.

[9]The author of this paper is an expert in Go. He is an amateur 6-dan and has been the champion of Hong Kong Grand Go Tournament from 1991 to 1995.

the locations of the group to be test. Figure 5.5 shows an example of the representation.



$\Downarrow$

0	1	1	−1	1	0	0	0	0	1	1	0	0	0	0	0
1	0	1	−1	1	0	0	0	1	0	1	0	0	0	0	0
−1	1	1	−1	1	0	0	0	0	1	1	0	0	0	0	0
−1	−1	−1	−1	1	0	1	0	0	0	0	0	0	0	0	0
0	0	−1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	−1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	−1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	−1	1	0	0	0	0	0	0	0	0	0	0	0	0

Board Map Location Map

Figure 5.5: Representation for a training example : we want to identify the alive-or-dead status of the group marked by ▲. The *Board Map* represents the 8×8 corner pattern and the *Location Map* indicates the group for identification.

The network has two output units, one corresponding to the alive decision and the other to the dead decision for the input pattern. The first one fires if the group is alive and the second one fires when the group is dead. The difference between the two output values is the alive-or-dead decision made by the network.

Partial Connectivity In our design, the first hidden layer is used for the primary extraction of *eyes*, which is an essential factor in deciding whether a group of stones is alive or dead. Connectivity between the input and first hidden layer is specially designed to facilitate the eye extraction process.

The neurons in the first hidden layer is divided into 2 groups. The first group has 90 neurons and the second group has 60 neurons. In the first group, each hidden neuron is

Window Size	No. of different windows	Hidden nodes for all windows	No. of connections for each node to the input layer
3×3	36	1-36	18
4×4	25	37-61	32
5×5	16	62-77	50
6×6	9	78-86	72
7×7	4	87-90	98

Table 5.4: Connectivity of 1st group of neurons in the 1st hidden layer

connected only to a local window of the 8×8 corner pattern. The size of the local windows ranges from 3×3 to 7×7, together covering the whole 8×8 region. For example, a hidden neuron connecting to a 3×3 local window will be connected to 18 different neurons in the input layers : 9 neurons in the 3×3 window of the board map. and 9 neurons to the corresponding 3×3 window in the location map. The detail of connectivity of the first group of hidden neurons is summarized in Table 5.4.

In the second group, each neuron connects fully to all neurons in the input layer, i.e. to the whole 8×8 corner pattern. This group is used to capture some irregular shapes of stones which contribute to some special cases in alive-or-death problem. Figure 5.6 shows a brief structure of partial connectivity between the input layer and the first hidden layer. The details of the region of partial connectivity in the figure has been described in the previous table (5.4).

In practice, training of networks with this partial connectivity in the first hidden layer is 1.5 times faster than the totally fully connected ones.

Training method The training data set consists of corner patterns extracted from different human games. The patterns do not have any relation with one another. During training, they can be presented to the network in any arbitrary order. As a result, the model of supervised learning is applied in this experiment. Backpropagation is used to train MLP's, instead of TD learning, to solve this problem .

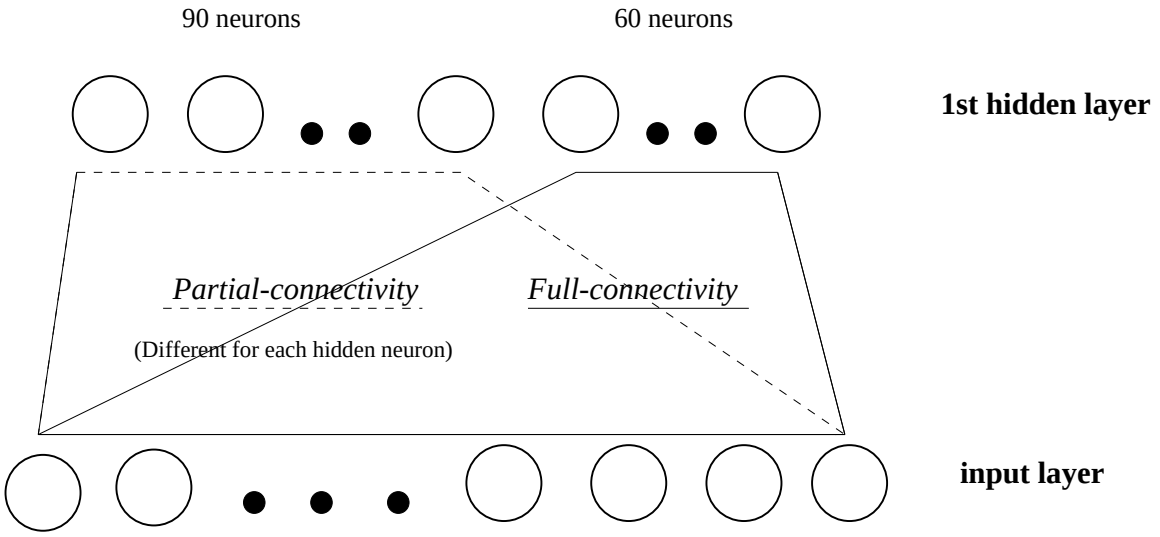


Figure 5.6: Connectivity between the input and first hidden layer in experiment 3

Strategies for Faster Convergence Some strategies are used to set the parameters of the neural network in order to make convergence faster [23].

1. **learning rate :** each neuron has its own learning rate. Neuron in last layers,which tend to have larger local gradients, should have smaller learning rates than the front-end layers. Moreover, a neuron with higher fan-in factor should also have a smaller rate.

2. **initial weight :** weights and biases are initialized with random numbers that are uniformly distributed inside a small range. The range, which is determined by the fan-in number (F_i) of the neuron, is $\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right)$.

Training Data Sets Three training data sets are used to train neural networks. The first data set contains 1391 patterns directly extracted from human games without passing and expert selection process. The second data set contains 969 patterns which are selected by human expert. The third data set is a super set of the second set. It contains 1495 patterns which are also selected by human expert. Training data selected by human expert

Training Data Set	No. of training examples	Human Selection	Total SSE	Average SSE	Wrongly Classified	Correct Percentage
(1)	1391	No	24.1032	0.06228	16	95.87%
(2)	969	Yes	20.0245	0.05174	12	96.90%
(3)	1495	Yes	11.5450	0.02983	6	98.45%

Table 5.5: Generalization results of different training data sets in the corner alive-or-dead problem

is noiseless because the process ensures that in each pattern, there should be sufficient information for judgment of the live-or-death status.

Testing Data Set The testing data set is used for testing the performance of generalization of the trained networks. It contains 387 patterns that are completely different from the training patterns. All the data in the testing set have passed the selection process by the expert and therefore can be used to test generalization accurately.

5.4.4 Results

Table 5.5 records the best performance of generalization of three networks that are trained with different training sets. The error of a network for each testing data is measured by the summed square error (SSE). It is calculated from summing up the square of difference between every desired output and actual output values. If the SSE for testing example is too large (> 0.3), the network is considered as making a wrong decision for that example. For each network, the total number of wrong decisions is counted and the correct percentage is calculated.

If we compare results between networks trained by data set (1) and the one trained by data set (2), it can be observed expert selection has a positive effect on the accuracy of generalization. It is because the selection process removes noisy data from the training set.

Comparing results of (2) and (3) in Table 5.5, the increase in the number of training

data results in a significant improvement in the accuracy of generalization. We foresee that better result can be obtained with more training data, which is a usual case in neural network learning. However, elimination of noisy examples through human inspection requires a lot of effort. The corresponding time for training an epoch and for convergence will also increase.

5.4.5 Discussion

The result of experiment 3 is very encouraging : accuracy in predicting the alive-or-dead state of a corner group is over 95%. This shows that with an appropriate problem decomposition, and specially designed experiments, neural network can be successfully applied in solving some problems in computer Go.

The techniques for the success of this application are :

1. *Reduction of the problem complexity to a manageable size.* Instead of using a whole board, only patterns in corner are used. This decreases the number of input units to about one sixth of the whole board.

2. *Nature of the problem fits the properties of the neural networks.* Alive-or-dead analysis of a group of stones is mainly a problem of pattern matching and analysis. This type of problems are suitable to be solved by neural networks. Another problem having high potential to be solved by neural networks is strength analysis of stones in Go.

Moreover, it should be noted that the size of the training data set is compact. The result of obtaining network with good performance from relatively small training set suggests some special properties in the game of Go. Similar results also occurs in experiment 2. In the next section, we will try to explain these phenomena with a postulated model of the Go game that has properties leading to these results.

5.4.6 Limitations

Since this is the first application in the area of solving game-specific problem of Go using neural network, our emphasis is mainly put on the design of the experiment. Instead of training a neural network, in the first time, that can be directly used in building a computer Go program, we aim at testing the feasibility of our approach. As a result, a reduced problem that handles 8×8 corner pattern is solved. In a real Go game, being able to analyze alive-or-dead state of stones at a corner is not sufficient because many groups of stones will be lying across that boundary.

With the extraction of 8×8 corner pattern, human examination, which is a slow and expensive process, is needed for selecting training patterns containing sufficient information. As a result, the number of training examples is limited. We suggest the use of group of stones in 9×9 or 13×13 Go board as an extension of this experiment such that no extraction of local region is needed. As a result, preparation of training examples can be done automatically.

5.5 A Postulated Model

In this section, we try to describe a postulated model based on the our experimental results and our knowledge in the game of Go. The implication of this model is giving a significant complexity reduction for solving some problems in Go, This gives a good direction for future research in computer Go.

Good generalizations obtained from experiment 2 and 3 has shown that even with relatively small training set, generalization can still be reached to a practically good level. In the two experiments, data in the training sets and the testing sets are obtained from games played by experienced (above-novice) human players. From this, we come to a hypothesis that *the state space of Go board positions can be functionally classified into 2 sets* : **ordinary set** and **bizarre set**. Most of the board positions occurring in human games are in the **ordinary set**. On the other hand, the **bizarre set** contains board positions that seldom occur in human games.

If we describe the concept from the human knowledge of Go, an *ordinary set* means the set of board positions reachable from sensible moves (i.e. moves with certain quality). *Bizarre set* is the set of board positions results from random and non-sensible moves. Size of the *ordinary set* should be much smaller than the *bizarre set*, as explained by the concept of *entropy*. It is because board positions in the *ordinary set* shall have some forms of order which implicitly represents the Go knowledge of the human players. On the other hand, *bizarre set* contains board positions without such order and are therefore more random. From the theory of entropy, the probability of occurrence of chaotic states without any order is much higher than the probability of the highly-ordered ones. Therefore the *ordinary set* shall occupy only a small portion in the state space of Go.

Since most of the board positions occuring in the human games belong to the *ordinary set*. If we train and test neural networks with examples restricted to the *ordinary set*, we might expect the network can generalize well. It is because the actual space complexity of the problem is much smaller.

For human players, the ability of classifying board positions from the two sets gives a complexity reduction of the problem. Supporting evidences for this complexity reduction techniques among human players in Go are

1. Most beginners find it hard to play Go because they find the number of moves they needed to consider is too large to handle. This is the observation from the author who has taught many introductory courses of Go. The difficulty can be explained as the lack of techniques of complexity reduction, assisted by knowledge in Go.

2. Experienced player can make instant differentiation of board positions played by human players and by random generation. This is because randomly generated board positions will produce some irregular shapes, which can be easily identified.

3. Strength of strong Go players can be retained in lightning game, in which a move is played within a very short period of time. Difference of strength between two players will often be magnified but never diminished in lightning games. This can be explained by the pattern recognition ability of the strong players. Such ability is

helpful in complexity reduction of space by recognizing unlikely-to-be-good patterns and ignore them in searching.

This above phenomenon positively verify the fact that strong human players acquire a good complexity reduction technique, which can be explained as the ability to classify board positions in the *ordinary set* and the *bizarre set*. In our experiments, since training data extracted from human games belong to the *ordinary set*, the space explored in training and testing is relatively small compared to the theoretical state space. This explains the extraordinary training performance obtained from a relative small training set.

If such hypothesis is true, it will be a good direction of research in solving the problem of *ordinary set* and *bizarre set* classification. This problem is suitable to be solved by neural networks because :

1. The classification is a problem of pattern analysis and neural network is strong in solving such problem.

2. Training examples for both set can be easily and automatically generated. Moreover, the supply is unlimited because board positions of any games played by strong human players will be suitable.

3. It is difficult to solve the problem analytically because that requires understanding of all properties characterizes patterns occuring in human games.

We propose this as an extension to our research.

Chapter 6

Conclusions

In this research, we have done some analysis and performed several experiments to investigate the application of neural networks in some problems of computer Go. We conclude our work in the following paragraphs :

Framework in Computer Go We use a practical definition of solving a game in the problem of computer Go. A framework is designed for decomposition of a large problem into smaller sub-problems in computer Go. In this research, we try to focus on some specific sub-problems using neural network learning.

A New Updating Rule We have derived a new updating rule for TD(λ) learning using multi-layer Perceptron which is a common model of neural networks. An algorithm for the new updating rule has been designed which is used in the experiments of TD learning in this research.

TD(λ) Learning for Nonzero λ in Go The first set of experiments are for applying TD(λ) learning, with nonzero λ (using the newly derived updating rule) to train evaluation function in 7×7 Go game. Self-playing is used in generating example board positions to train the networks. From the experiments, we find that the larger the value of λ within $[0, 1]$, the better the performance.

TD(λ) Learning by Human Games in Go The second set of experiments are for applying TD(λ) learning to train evaluation function for 9×9 Go game. The approach is different that examples from human games are used to train the network, instead of using self-playing. Results show that TD learning does not perform well in this case. Better results are obtained from using a set of independent examples. In such case, it is reduced to the supervised learning model. From these results, we suggest that TD learning with a provided set of training examples is not a good strategy in solving deterministic game like Go.

Alive-or-dead analysis In the last set of experiments, we try to solve an important sub-problem in board evaluation : alive-or-dead analysis of a group of stones on the board. The complexity of the problem is reduced by imposing some restrictions to the training examples. Successful results show that neural network using supervised learning model is very suitable for solving this problem. We propose some extended experiments for some further investigation in this area.

A Postulated Model for Complexity Reduction Base on the results of the second and the third set of experiments, we find that performance of trained neural networks can be surprisingly good with a training set of moderate size. Consequently, we postulate a model of classification of board positions into an *ordinary set* and a *bizarre set*. Solving problems which focusing on the ordinary set that contains board position frequently occuring in human games would be an effective way for space reduction for Go, which has a huge state space.

Overall Conclusions Concluding the whole research, we have started an alternative area for computer Go and explored some learning methods for solving sub-problems in computer Go. Although application of TD learning in Go is practically difficult because of the determinism in the game, some encouraging results are obtained from training using the supervised learning model. As a result, we find that there is a good prospect in applying supervised learning model in some specific problems of analysis in computer Go.

6.1 Future Direction of Research

Extended Alive-or-dead analysis and Strength analysis First, we can extend experiments of alive-or-dead analysis to groups on a board position of 9×9 Go games with no spatial restriction. The use of 9×9 Go board can maintain the complexity of the problem in similar level. Moreover, there will be no need for human inspection for selecting noise-less training data. On the other hand, solving the problem of *strength analysis* for stones in open and early-middle game, which is a problem that has some common properties as the alive-or-dead analysis, is also of good prospects.

Classification of the ordinary and bizarre set Training neural networks to classify board positions into the ordinary set and bizarre set will be an experiment of great significance. Being able to do the classification can result in reduction of complexity of many problems in computer Go by focusing on board positions mainly from the ordinary set.

In a practical sense, the classification problem is suitable to be solved using neural network. It is because finding an analytical solution requires a complete understanding of properties that appear in human game. This is too difficult from our current knowledge in Go. Moreover, it is easy to generate training data and the amount is almost unlimited because board positions from any human games can be used to train the neural networks for solving this problem.

Appendix A

An Introduction to Go

In this chapter, we will give a brief introduction to the game of Go. Then we will describe some of the complex strategies in Go.

A.1 A Brief Introduction

A.1.1 What is Go?

Go is board game played by two persons. In technical term, it is a deterministic two-player zero-summed game with perfect information. It is known as *Wei-Qi* in China and *Baduk* in Korea. The name *Go* is a Japanese translation.

The goal of the game Go is to seize more territories than the opponent. It is a difficult game with sophisticated tactics and strategies.

The game is originated in the Orient. Because of its interesting properties, it is now gaining popularity in the Western world.

A.1.2 History of Go

According to myths, Go was invented by Emperor Yao who was the ruler of the Han race in China around 4000 years ago. Originally, it was just a tool for fortune telling and gambling. Later it became an interesting and challenging recreational game.

The game was learned by the Korean and then Japanese around the 5th century. In the 16th Century, Go was first institutionalized and became professionally studied in Japan. Japanese Emperor started to subsidize 4 professional Go institutes for teaching and studying Go. The skills in Go was then largely improved as a result of the systematic study and the keen competition between the institutes. Consequently the professional Go players in Japan achieved a very high playing level since then.

In the past of China, Go was like a traditional arts. Some nobles, officials and rich people liked to employ some strong Go players for fame and for using them to play betting-match with other players. The skills of Go in ancient China therefore relied heavily on the prosperity of the society. The situation continued until Go was treated as a sports activity in the past few decades. Since then, Go has become more popular in China and the skills of the top professional players are increasing quickly. Nowadays, the best Chinese professional Go players are at a competing level with the top professionals in Japan.

Korea (South) started to develop Go as one of their National cultural activity after the World War II. Korea is now having the greatest density of Go population in the world. Recently, a few top Korean Go professionals started winning a lot of international tournaments and becoming competitive with the professionals from China and Japan.

Besides the 3 largest "Go countries," Go is also very popular in Oriental regions like Taiwan, Singapore and Hong Kong. It is estimated that the population of Go around the whole world is more than 25 millions now! This interesting and challenging game is also becoming more popular outside the Oriental countries. The interesting properties of the game are the reasons for its large popularity.

A.1.3 Equipment used in a Go game

A complete set of Go equipment contains a board and 2 sets of stones with color black and white respectively and a pair of bowls for containing the stones.

A standard Go board is square in shape. It is usually made of wood. It has 19 horizontal lines and 19 vertical lines, forming totally 361 intersections called *points*. (There are also

small 13×13 and 9×9 boards for beginners.) Inside a 19×19 Go board, there are 9 specially marked points called *stars*. They are for easy visualization and for putting handicap stones in a handicap game.

There are 2 types of Go stones: black and white. A complete set of stone has 181 black stones and 180 white stones and stones of the same color are put inside a bowl-shaped container. The stones are circular in shape. Some are swelling on both side (Japanese style) and some are swelling on one side and flat on the other side (Chinese style). Stones are usually made from plastic, glass or genuine stones. Some expensive stones are made from shell and even jade!

A.2 Basic Rules in Go

Go is a board game with simple basic rules but complicated strategies. It takes only a few minutes for one to learn the rules but years for a player to master! Since our knowledge of the game is still very limited, new variations and strategies are discovered by human experts every year. It seems that even some of our basic concepts in Go are still in the current of change.

A.2.1 A Go game

A Go game requires 2 players to play. One player takes black stone and another takes white stone in a game. In an even game, Black player always plays first. In a handicap game, the weaker side will take black and put a certain a number of stones on the board first. Then white will start to play. Each player plays alternately. A move is played by putting a stone at an empty point on the board and removing dead stones from the board, if any.

The goal of the game is to surround and control as many unoccupied points as possible. The empty points once surrounded by one side become the *territories* of that side. The player who surrounds more territory than his/her opponent wins the game. In the competition for territory, there will often be battles between stones. Some stones may

be captured and removed from the board. We will explain these ideas in the following sections.

A.2.2 Liberty and Capture

A *liberty* of a stone is an empty adjacent point of that stone. In Figure A.1, the black stone at the corner has two liberties, labeled "1" and "2". Similarly, the black stone at the edge has three liberties and the one in the center has four liberties.

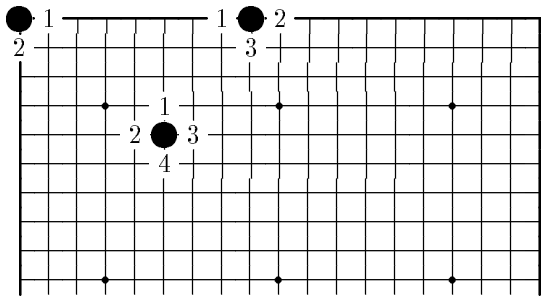


Figure A.1: Liberties of a single stone

Two stones of the same color are *connected* if they are adjacent to each other. This connectivity relation is transitive. Stones of the same color form a *string* if they are connected to one another. The liberties of a string is the empty adjacent points of every stone in that string. Liberties, shared by many stones of a string, at the same point count only once. In Figure A.2, the string at the corner has three liberties counted by number 1, 2 and 3. The string of three stones at the edge has five liberties. The 3-stone string in the center has eight liberties.

If all the liberties of a string are filled up by opponent stones, the string will be *captured* and removed from the board. Stone removed from the board is called *prisoner* which is returned to the player of that color. (according to the *Chinese Rule* [1]). Note that a string can only be captured *as a whole* if all its liberties are filled by opponent stones. Stones of a string cannot be captured separately. In Figure A.3, at the corner, after the move ① is played, two black stones are captured and removed. At the edge, after ① is played, three

[1] In Japanese rule, prisoners are kept by the player who captures them in

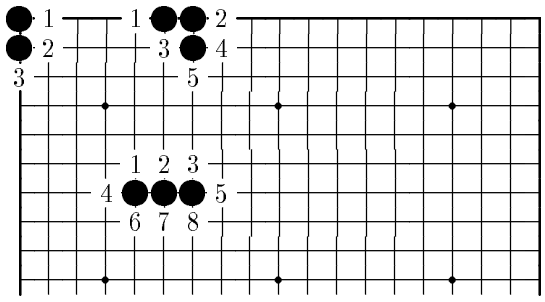


Figure A.2: Liberties for a string of stones

black stones are captured. The case is similar in the center where four black stones are captured after ①.

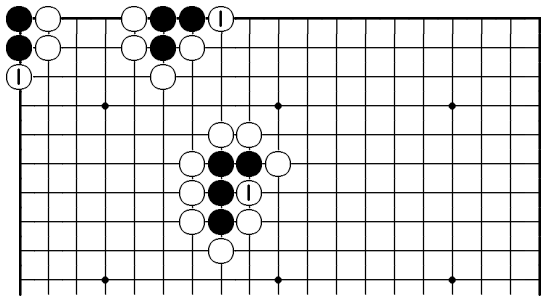


Figure A.3: Examples of capturing stones

In many situations, stones of both players are cutting each other into separated strings. Usually, battle will start when this cutting situation occurs. In Figure A.4, at all the corner area, there are battles between black strings and white strings.

A player cannot make a move at a location that makes any of his/her own string out of liberty without capturing any opponent's stone. Such move is *prohibited* by the rule. In Figure A.5, all points labeled from *a* to *e* are prohibited moves for black.

A move is allowed if it makes both one's string and some opponent's strings out of liberty simultaneously. In such case, the opponent string(s) will be captured and removed but not one's own string. In Figure A.6, all the moves marked *a* are allowed for white though the white stone at *a* seems to be out of liberty at a moment. The outcome will be corresponding black stones marked ▲ being captured after white plays a move at either position *a*.

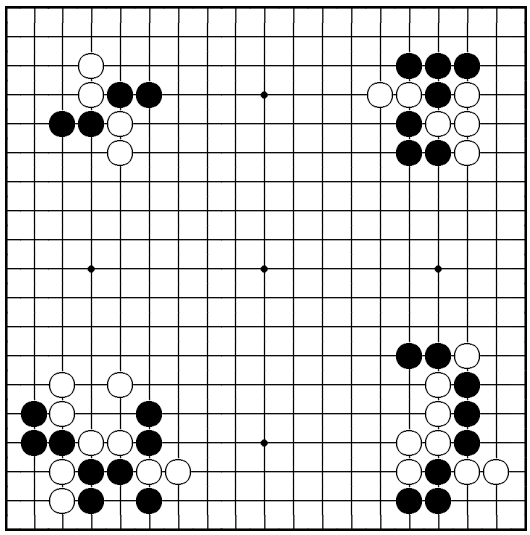


Figure A.4: Examples of cutting into separate strings

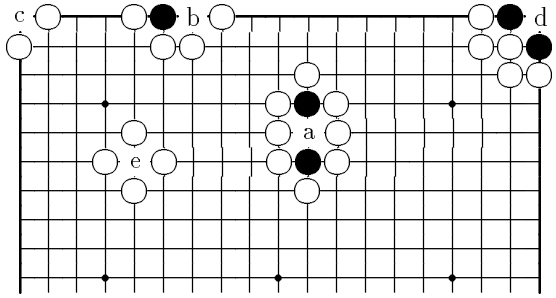


Figure A.5: Examples of prohibited moves

If a string has only one liberty left, it is in the state of *Atari*. In Figure A.7, all black stones ⬤ are in Atari since they only have one liberty, marked by *a*).

A.2.3 Ko

Ko is a shape of that cyclic capture-and-recapture of a stone is possible. Figure A.8 shows three examples of Ko. If both player insist on capturing the opponent stone in the Ko, the same situation will go on forever and the game cannot be finished. As a result, there is a rule forbidding the immediate re-capture of the stone in Ko. Once the Ko starts and player *A* captures the stone in the Ko, the stone he/she just played becomes a *hot*

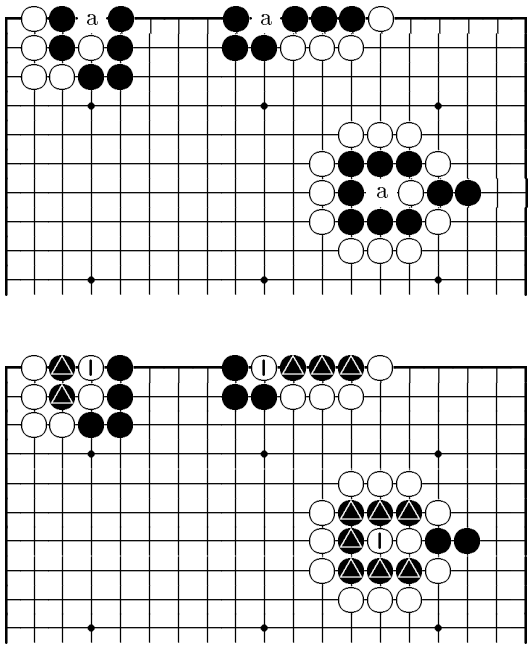


Figure A.6: Examples of two strings running out of liberty at the same time

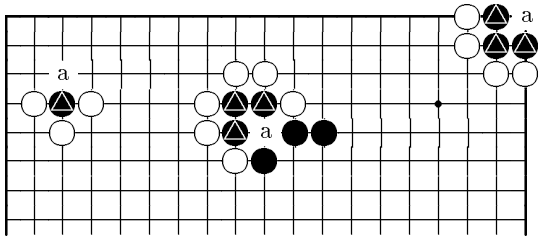


Figure A.7: Examples of Atari

stone. The other player (B) cannot re-capture the hot stone immediately. He/She must play somewhere else. If the player A also plays somewhere else, player B can capture the "hot stone" as it has been "cooled down". For the same reason, B's stone in the Ko becomes a new hot stone and player A must play somewhere else before capturing the hot stone. According to this rule of Ko, situations of immediate recapture in Figure A.8 cannot occur in a real game. Figure A.9 demonstrates an example of Ko fight in a real situation. When white plays ① and captures the black stone in Ko, black cannot capture back immediately. Black plays at ❷, threatening to cut the white group in the left side into two pieces. White responds with ③ and forms a string. Now the hot stone ① has

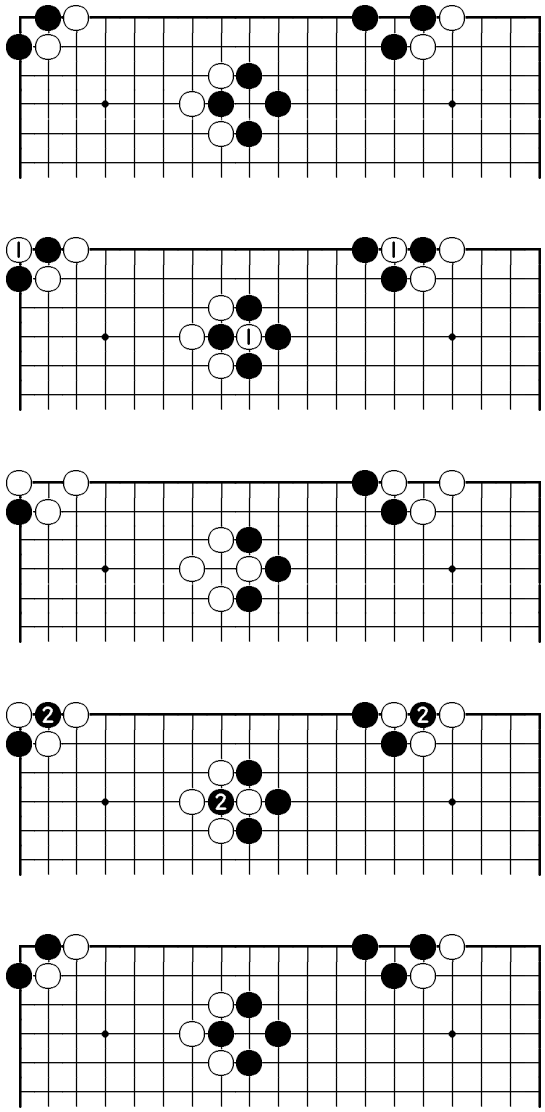


Figure A.8: Diagram showing cycle in Ko

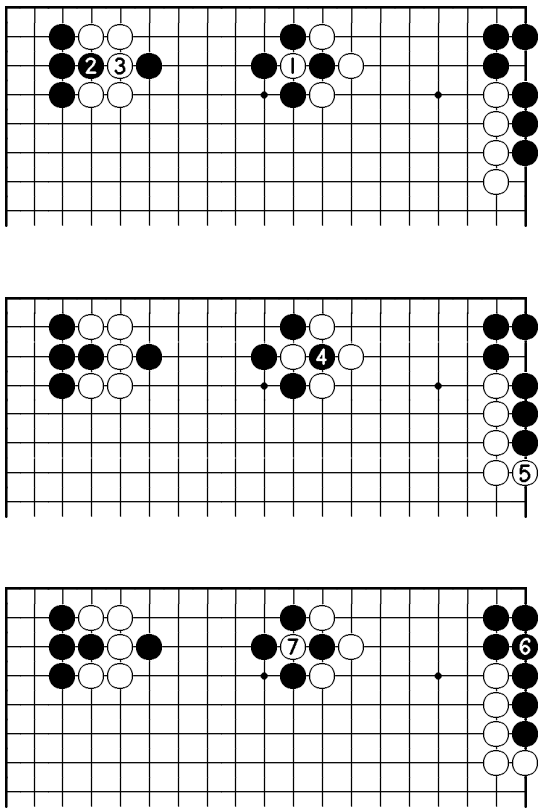


Figure A.9: Example of Ko and Ko threat

cooled down and Black can play ❹. Similarly white has to play ⑤ and black plays ❻ before white can capture the Ko in ⑦. Moves like ❷ and ⑤ are called *Ko threat*. Ko threat is a move that opponent will likely respond when it is played. Otherwise, there will be some adverse effects for the opponent. Therefore, Ko threats are necessary in a Ko fight. The side with more Ko threats will have a higher chance to win the Ko.

With the Ko rule just described, cyclic situation in Ko can be solved because the Ko can finally be terminated when one player ignores the opponent's Ko threat.

Figure A.10 demonstrated this situation. When white capture the Ko stone by ①, black make an *Atari* by ❷, which is a Ko threat. In usual cases, white will save the three stones by connecting them to the other white group. However, in this game, white judges that he/she can get more profit in winning the Ko than in saving the three stones. As a result, white ignore that *Ko threat* and terminate the Ko by playing ③ and capture the

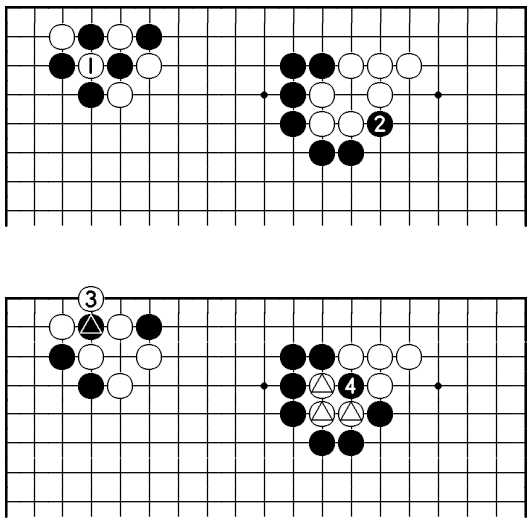


Figure A.10: An example of termination of Ko

black stone ▲. In return, black can now capture the three white stones (marked by △) by ❹. This case has similar meaning as *material exchange* in Chess but is less trivial here.

A.2.4 Eyes, Live and Death

Basically, an *eye* is one empty points (or many-connected points) connected point that is surrounded by stones of one side. If a group of stones possesses of 2 eyes, it can be alive even if surrounded by the opponent stones. In Figure A.11, all the three black groups have 2 eyes, labeled by a and b for each group. White cannot play at either point of black's eye because the move will violate the rule (prohibited move). As a result, all three groups of black stones are alive even though they are completely surrounded by white.

When surrounded by opponent stones, group of stones having only one eye will be dead. It is often that some surrounded space containing more than one point can form only one eye. Figure A.12 shows some examples of this case. After playing ① in each pattern, each black groups can essentially form one eye in the space surrounded by black stones. Therefore, they are dead.

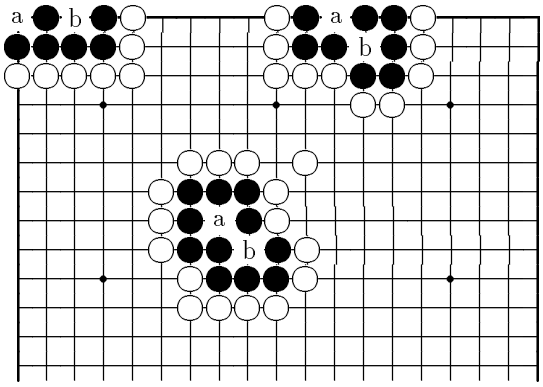


Figure A.11: Examples of alive by having 2 eyes

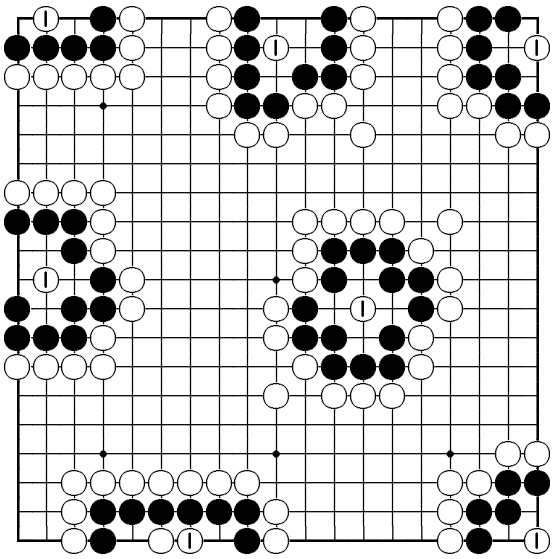


Figure A.12: Examples of big eyes

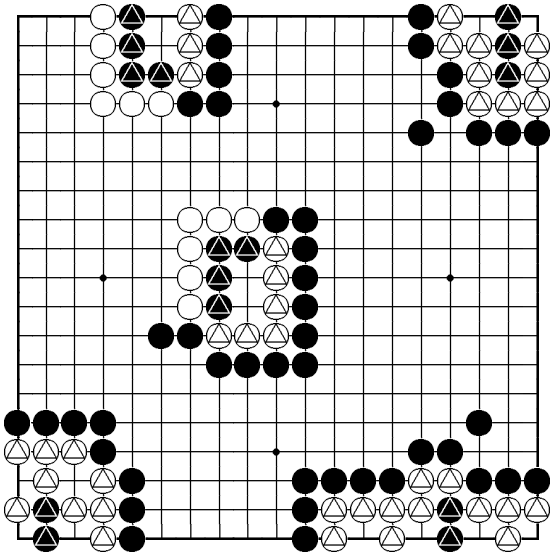


Figure A.13: Examples of Seki

A.2.5 Seki

In some special cases called *Seki*, strings without two eyes can still be alive. Existence of *mutual liberties* (i.e. liberties shared by some stones of different color) is a necessary condition. In Figure A.13, all black strings ▲ and white strings △ are alive, though they do not have 2 complete eyes.

Whether a situation is a Seki depends on the shape of the involved stones. Some similar-looking patterns may result in completely different outcomes. For example, in Figure A.14, the three patterns in the upper diagram looks very similar to the corresponding ones in the lower diagram. However, all white strings △ in the upper diagram are alive but all white strings △ in the lower diagram are dead. This is because of the slight difference in shape of black and white stones. Please refer to some Go books for a detail description [12], [5], [18].

A.2.6 Endgame and Scoring

Normally, a Go game is finished under one of two situations :

1. When either player resigns the game.

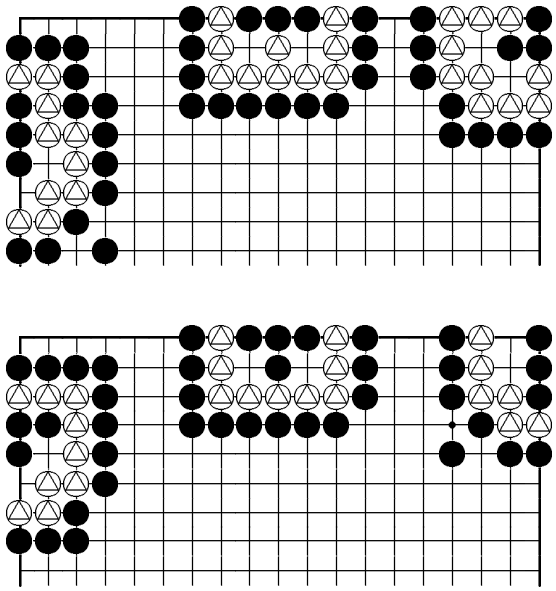


Figure A.14: More examples of Seki

2. When all empty points except territories are filled.

In a game with time limit for both players, there may be a third situation : when either player used up all his/her time.

In the second situation, scoring is needed to determine which player has won the game. Players have to agree on which stones on the board are dead under consensus. The dead stones are then removed from the board and returned to the original player. After that, they can count the total number (n) of (stones + points of territory) of either side, say black. If the sum is greater than $180\frac{1}{2}$ (which is the half of total number of points on a 19×19 Go board), the player of that side (black in this case) wins the game. The difference between the number n and $180\frac{1}{2}$, in terms of *stone*, is the amount of winning. This is the Chinese scoring method.

Since Go is a game of competition for territory on the board, the black player who plays first always has certain advantages over the white player. In a tournament match, the black side has to give some compensation, called *Komi*, to the white side in order to make the winning chance for both sides equal. The size of Komi is usually $2\frac{3}{4}$ stones (which is equivalent to 5.5 points of territory).

A.2.7 Rank and Handicap Games

The playing strength of a player can be estimated by his/her rank. The *Japanese ranking system* consists of *kyu* and *dan* is a universally accepted ranking system. In the systems, professional players and amateur players are ranked by different scales. For amateur players, when a person has learned the basic rules of Go, he/she will be a *15-kyu*, which is the entry level. The smaller the number in the kyu level, the stronger the player. The highest kyu level is 1-kyu, which is the strongest novice level.

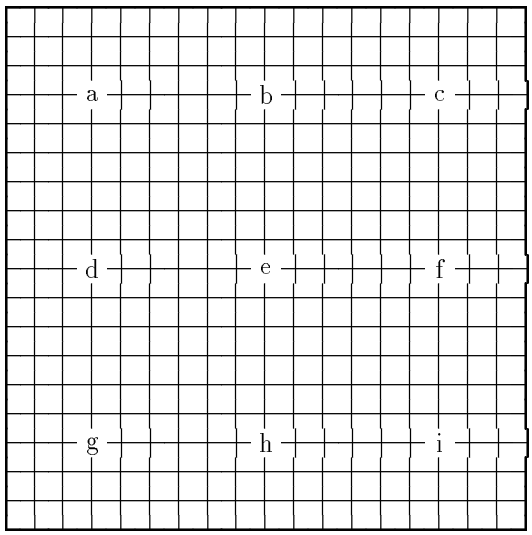
After the kyu levels, it comes to the *dan* levels. A dan level represents an expertise in Go, which is a respectable symbol in the amateur Go community. 1-dan (called *Shodan* in Japanese) is the weakest dan level. Unlike the kyu levels, the larger the number in dan, the stronger the player. The strongest dan level for an amateur player is 6-dan. (In some rare situations, amateur 7-dan will be granted for ultra strong amateur players) The scale is summarized as (where k = kyu and d = dan) :

$$\text{(beginner) } 25k \to 24k \to \cdots \to 2k \to 1k \to 1d \to 2d \to \cdots \to 6d \text{ (expert)}$$

The scale for professional players are different. A player can enter the professional community only after he/she can be qualified in some special selection tournaments. The are different selection tournaments organized by different professional Go institutes in China, Japan, Korea and Taiwan. Usually, a person has to be intensively trained in Go when he/she is very young (around 10 years old) if he/she wants to become a professional Go player. The entry level is 1-dan and the highest rank is 9-dan. The professional dan levels is much stronger than the amateur ones. Usually, a professional 1-dan is even stronger than an amateur 6-dan!

There is a complete and effective handicap system in Go. When two players with different ranks play a Go game, the weaker player can put certain number stones on the predefined locations on the board before they start playing. Such stones are called *handicap stones*. Since the weaker player should take black stones by convention, the handicap stones are always black stones.

The number of handicap stones is proportional to the difference in strength between



Number of Handicap Stones	Positions of Handicap Stones
2	a,i
3	a,c,i
4	a,c,g,i
5	a,c,e,g,i
6	a,c,d,f,g,i
7	a,c,d,e,f,g,i
8	a,b,c,d,f,g,h,i
9	a,b,c,d,e,f,g,h,i

Figure A.15: Positions for putting handicap stones

the two players.. The larger the difference, the more the handicap stones. The number of handicap stones usually ranges from 2 to 9. The ways of putting handicap stones are illustrated in the Figure A.15

For amateur players, when the rank difference between two players is one, the weaker player can take black without giving Komi (compensation). This is equivalent to taking one handicap stone. When the difference in rank is two, two handicap stones are needed. The number of handicap stones is equal to the difference in their ranks.

The strategies and tactics in a handicap game is very similar to an even game. The major difference is that the black (weaker) player is always advised to play concretely and safely while the white (stronger) player should use some unusual and tricky strategies to

stir up the mud in a handicap game.

With this effective handicap system, players of different ranks can play Go games with a fair and enjoyable environment.

A.3 Strategies and Tactics in Go

We are trying to give brief explanations to some advanced concepts in Go in order to demonstrate the complexity of the game and the ways human experts trying to understand the game.

A.3.1 Strategy vs Tactics

Human skills in Go can be roughly classified into two classes : *strategy* [2] and *tactics*. Strategy refers to the abilities to analyze the board and plan the corresponding actions according to the analysis. Human players perform this in a highly abstract manner. Analysis can be can be spatial (deciding which region on the board is the most important) or functional (deciding what action should be performed like offense, defense, building influence, occupying more territories, etc.) As the state space of Go is too large for detail evaluation of each strategy's outcome, heuristic is used by human players to find a satisfactory strategy in a situation. Prior knowledge on semantically similar situations and abstract rules will often give good hints for designing strategies.

On the other hand, tactics are the skills applied in some local regions of the board where stones from both sides come into spatially close interaction. The depth of searching (lookahead) is an important factor in tactics. Usually, a specific goal has to be accomplished for a local battle. (e.g. Killing an opponent group, attacking a weak group, escaping from opponent's web, etc.) Therefore, tactical searching is goal-directed. Good experience of effects of local patterns will help to prune many moves that are unlikely to be good. As a result, player with richer knowledge of common patterns can search deeper in the same amount of time. This is the major difference in tactical skills between a beginner and an

[2]The meaning of strategy in here is different with the one in game theory.

	Strategy	**Tactics**
Analogy	War	Battle
Spatial Effect	Global	Local
Important in	Opening and Middle Game	Middle and Endgame
Searching Property	Wide and shallow	Deep and narrow
Important Skills	knowledge of similar global situation	Familiarity of group patterns
Training method	Studying more professional games and understand the strategies used inside the games	Do more exercises which requires deep lookahead such as alive-or-dead problems (Tsume-Go)

Table A.1: A summary of Strategy vs Tactics

experienced player though they may put similar effort in the tactical analysis in playing a game.

A.3.2 Open-game

In the stage of open-game, there are systematically organized knowledge like Joseki and Fuseki. There are also some abstract concepts for board analysis, like Concrete-territory and Influence.

Joseki The term *Joseki* means "standard variation". A *Joseki* is a sequence of excellent moves that will give acceptable results for both players in the local region of the board. Almost all Josekis are designed for the corner regions of the board. This is because the corner regions are very important for it is more easy to occupy territory there than in the edge and center regions. They are usually invented by strong professional players from their sparkling insights in their games or through studying. As every move in a Joseki is good, a Joseki will lead to locally agreeable results for both players. However, the global effect of a Joseki in a game will not be that simple. Combination of different Josekis from different corners will give various effects. Therefore, the choice of Joseki is very important. Choosing an appropriate

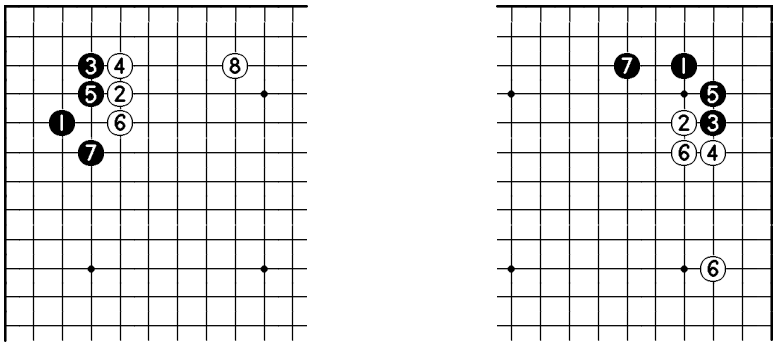


Figure A.16: Two examples of Joseki

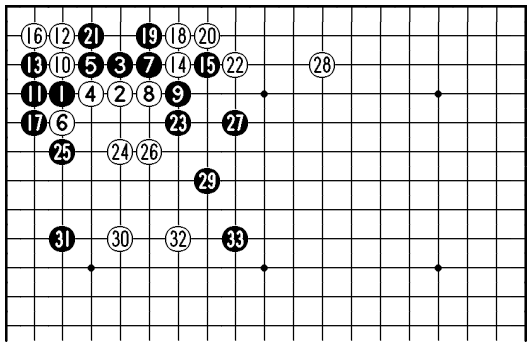


Figure A.17: Example of a complicated Joseki with 33 moves

one can lead to good global result and choosing an inappropriate one will give bad global result.

The reason for using Joseki is just like playing according to some standard opening styles in Chess. Since the complexity of moves restricted to a corner is not as high as the whole board, deep study on different variations is possible for strong and dedicated professional players. A large collection of Josekis has been developed and many new ones are discovered every year.

Fuseki *Fuseki* means "opening style". The area for a Fuseki to apply is larger than the area for a Joseki (Usually, at least half of the whole board is involved). Knowledge in a particular Fuseki consists of a combination of patterns in corner and edge regions, different solutions against opponent's invasion and methods for further expansion, if the opponent does not intrude. In Figure A.18, ❶,❸ and ❺ is a famous Fuseki

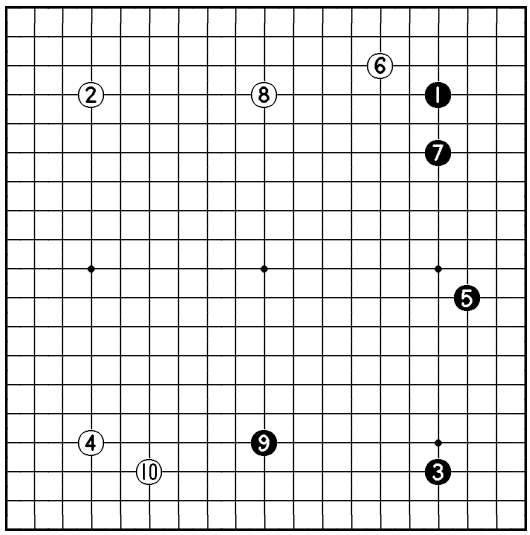


Figure A.18: An example of Fuseki

called **Chinese Style**.

The degree of freedom in Fuseki is much larger than in Joseki. Usually players have to handle many new situations that have not been encountered in previous games.

Concrete Territory vs Influence When some empty points on the board is almost surrounded by one side which can never intruded by opponent, the surrounded region becomes that side's *concrete territory*. The size of the concrete territories is fixed which has to be counted during board evaluation.

On the other hand, there are assets having value which are not as certain as concrete territories. *Influence* is an example. We can imagine that each stone on the board radiates some influence to the surrounding empty points. Radiation of a stone decays with distance and will be blocked by other stones [10]. In some region, if radiation accumulated from stones of ,say, black side is larger than the radiation from another side, the region will be under the influence of the black side. For example, in Figure A.19, all the radiation of black stones to the upper area are blocked by white stones. Therefore, there is unbalanced influence of white side in the upper area.

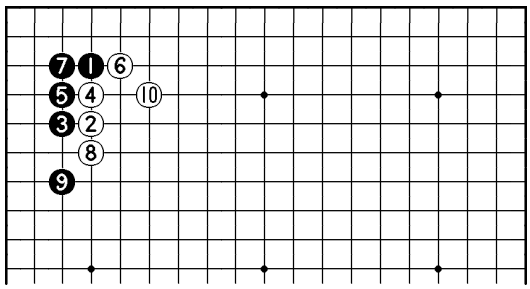


Figure A.19: In this Joseki, black gains some concrete territory while white gains large influence to the upper area of the board

Having influence on a region is advantageous. For example, battle inside a region under black's influence will be favorable to the black side. Moreover, region with a very high unbalanced influence of one side will more likely be that side's territory.

Although influence is a advantageous factor, Go is a game of struggle for territory. Unable of transforming influence back to territory at later stage of a game will result in a loss of the game. As a result, how to make use of influence becomes a very important skill in Go, which is not easy to acquire.

Difference of concrete territory and influence is analogous to the difference between cash and investment. Cash is real but fixed while investment can grow but can also shrink depends on the investor's techniques. However, in a Go game, it is always emphasized that maintaining a balance between the two is necessary. The difficult usually lies in evaluating the value of influence in different situations. The value often depends the skills of the player, which become very uncertain when compared to the value of concrete territory.

A.3.3 Middle-game

Middle game is the stage that is richest in strategies. To study them needs a lot of effort. Here, we only give the explanations of a few common and basic concepts involved :

Offense an action that tries to threaten the life of an opponent group of stone.

Defense an action that tries to increase the safeness of a peer group (the player's group of stone).

Aji Flaw in a group of stone that does not cause any damage at the current moment. However skillful opponent may design some plots to made use of the flaw to get benefit.

Thickness Flawlessness of some groups of stones. One side with "thick" groups can play strongly by offending without the need of defense.

To devise a good strategy for a board position in the middle-game, certain skills have to be acquired :

1. *Understand the meaning of each basic strategy.* For example, knowing what is an offense (attack) and its purpose.

2. *Knowing how to implement an action.* For example, when the player knows it is a right time to attack, he/she should know how to play to accomplish the task.

3. *Analyzing the board position.* For example, identify whether a group is weak or whether a group has some serious flaw.

4. Deriving a set of actions for attaining a certain goal. For example, when decided to kill a group of opponent stones, a set of actions have to be done before the act, like thickening some peer groups, repairing some flaws of the groups that surround the opponent.

A.3.4 End-game

In the stage of end game, the goal is mainly to finish the boundaries of territories in a favorable way. Usually, the size and shape of the territories are roughly formed. Moves are played to grow some of existing territory or to reduce the opponent's territory. As a result, boundaries will become more clear as more moves are played in the end-game.

To play a strong end game, exact value of every plausible move has to be known. The move that worths most should be played first. The value of moves in some simple patterns can be computed mechanically [4]. However, estimating value of moves in real situations is often so difficult and expert knowledge is needed to solve the problem.

Appendix B

Mathematical Model of Connectivity

B.1 Introduction

In this appendix, we are trying to mathematically define the concept of connectivity from the basics. Some simple concepts like *capture* and *Ko* are defined with the definition of connectivity. Definitions in this form will be useful for designing a rigorous mathematical model of Go knowledge.

B.2 Basic Definitions

1. The set of valid coordinate of a $n \times n$ Go board is :

$$\mathcal{B} \equiv \{1, 2, \ldots, n\}$$

where $n = 19$ for a standard board. Other common values of n are 9 and 13.

2. The set of stone color is :

$$\mathcal{C} \equiv \{black, white\}$$

3. The set of points, $\mathcal{P}$, on a $n \times n$ board is defined as

$$\mathcal{P} \equiv \mathcal{B} \times \mathcal{B}$$

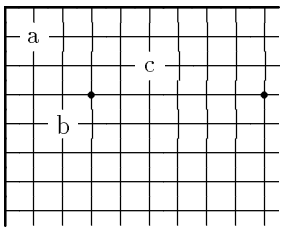


Figure B.1: Horizontal and vertical distance of some points

4. The set of stones is :

$$\mathcal{S} \equiv \mathcal{P} \times \mathcal{C}$$

For example, a stone s of color c at point (x, y) is denoted by s=((x,y),c) where $(x, y) \in \mathcal{P}$ and $c \in \mathcal{C}$.

5. A stones $s_1 = ((x_1, y_1), c_1)$ is the **peer** stone of another stone $s_2 = ((x_2, y_2), c_2)$ if $c_1 = c_2$. Otherwise s_1 is called the **opponent** stone of s_2.

6. status(p) denotes the status of a point p :

$$\textbf{status}(p) \in \{black, white, empty\}$$

7. A **move** in a Go game , m, can be represented by :

$$m = (p, c, n) \in \mathcal{P} \times \mathcal{C} \times \{1, 2, \ldots\}$$

where p is the point of the move, c is the color of the stone and n is an positive integer denoting the move number.

8. Horizontal and vertical distance (Δ_x and Δ_y) between 2 points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, are defined as

$$\Delta_x(p_1, p_2) \equiv |x_1 - x_2|$$

$$\Delta_y(p_1, p_2) \equiv |y_1 - y_2|$$

In figure B.1, $\Delta_x(a, b) = 1 \quad \Delta_y(a, b) = 3$

$\Delta_x(a, c) = 4 \quad \Delta_y(a, c) = 1$

$\Delta_x(b, c) = 3 \quad \Delta_y(b, c) = 2$

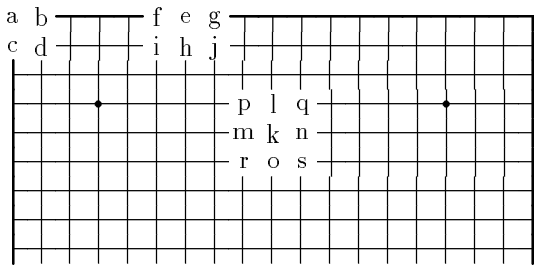


Figure B.2: Major-adjacency and minor-adjacency of some points

B.3 Adjacency and Connectivity

Adjacency is a relation for points while connectivity is a relation for stones. Both relations are commutative. Here are the definitions of some forms of adjacency and connectivity :

1. The relation of *major-adjacent* (**major-adj**) between two points p_1, p_2 is defined as :

$$\textbf{maj-adj}(p_1, p_2) \equiv (\Delta_x(p_1, p_2) = 1 \wedge \Delta_y(p_1, p_2) = 0) \vee (\Delta_x(p_1, p_2) = 0 \wedge \Delta_y(p_1, p_2) = 1)$$

2. The relation of *minor-adjacency* (**minor-adj**) is defined as :

$$\textbf{min-adj}(p_1, p_2) \equiv \Delta_x(p_1, p_2) \leq 1 \wedge \Delta_y(p_1, p_2) \leq 1$$

In figure B.2,point a is *major-adjacent* to b, c and
is *minor-adjacent* to b, c and d.
point e is *major-adjacent* to f, g and h and
is *major-adjacent* to f, g, h, i and j.
point k is *major-adjacent* to l, m, n and o and
is *minor-adjacent* to l, m, n, o, p, q, r and s.

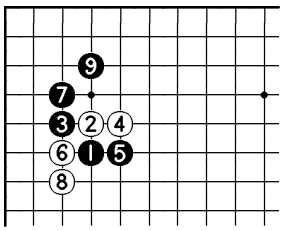


Figure B.3: Major-connectivity and minor-connectivity of some stones

3. The relation of *major-connect* (**major-con**) between two stones $s_1 = (p_1, c_1)$ and $s_2 = (p_2, c_2)$ is defined as :

$$\textbf{maj-con}(s_1, s_2) \equiv \textbf{maj-adj}(p_1, p_2) \wedge c_1 = c_2$$

4. The relation of *minor-connect* (**minor-con**) between two stones $s_1 = (p_1, c_1)$ and $s_2 = (p_2, c_2)$ is defined as :

$$\textbf{min-con}(s_1, s_2) \equiv \textbf{min-adj}(p_1, p_2) \wedge c_1 = c_2$$

where $s_1 = (p_1, c_1)$ and $s_2 = (p_2, c_2)$, $s_1, s_2 \in \mathcal{S}$

In figure B.3, ❶ is major-connected to ❺ and minor-connected to ❸.
⑥ is major-connected to ⑧ and minor-connected to ②.
④ is major-connected to ②.
❼ is major connected to ❸ and minor-connected to ❾.

5. Major adjacency and connectivity are more strict than the minor ones :

$$\textbf{maj-adj}(p_1, p_2) \Rightarrow \textbf{min-adj}(p_1, p_2)\,, p_1, p_2 \in \mathcal{P}$$

and

$$\textbf{maj-con}(s_1, s_2) \Rightarrow \textbf{min-con}(s_1, s_2)\,, s_1, s_2 \in \mathcal{S}$$

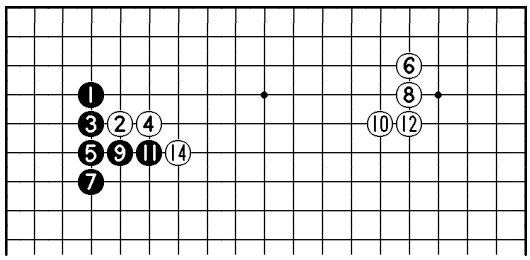


Figure B.4: Examples of string

B.4 String and Link

B.4.1 String

A **string** is a set of **peer** stones which are *major-connected* together. Classification of stones into string is very important because individual stone of a string cannot be captured separately. Once a string is out of *liberty* (defined later), the whole strong will be captured altogether. Here is a recursive definition of string :

1. $\{s\}$ is a string where $s \in \mathcal{S}$
2. $St_1 \cup St_2$ is a string if and only if
   (a) Both St_1 and St_2 are strings
   (b) $\exists s_1 \in St_1 \wedge s_2 \in St_2$, **maj-con** (s_1, s_2)
   where $s_1, s_2 \in \mathcal{S}$

Let **color** (St) be the color of a string St. A String St_1 is a *peer string* of St_2 if **color** (St_1) = **color** (St_2). Otherwise, St_1 is a *opponent string* of St_2.

In figure B.4, ❶,❸,❺, ❼,❾ and ❶❶ form a black string. ② and ④ form a white string. ⑥,⑧,⑩,⑫ form another white string. However ④ and ⑭ do not form a string.

B.4.2 Link

A link is a set of stone of the same color which are *minor-connected* together. Identification of links from stones is important because to capture opponent stones, a link has

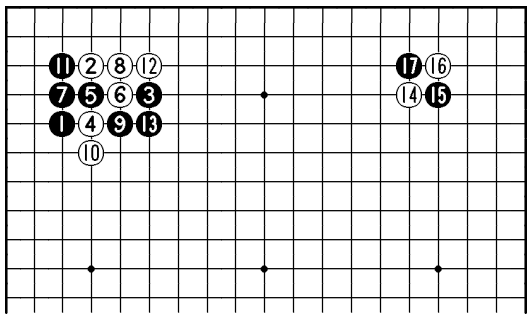


Figure B.5: Examples of link

to be formed. Here is a recursive definition of link :

1. $\{s\}$ is a link where $s \in \mathcal{S}$
2. $Lk_1 \cup Lk_2$ is a link if and only if
   (a) Both Lk_1 and Lk_2 are links
   (b) $\exists s_1 \in Lk_1 \wedge s_2 \in Lk_2, \textbf{min-con}(s_1, s_2)$
   where $s_1, s_2 \in \mathcal{S}$

If a set of stones form a string, they also forms a link but the converse is not true. In figure B.5, ②,④,⑥,⑧,⑩ and ⑫ form a link in the upper left corner. Similarly, black stones ❶,❸,❺, ❼,❾,❶❶ and ❶❸ also from a link. ⑭ and ⑯ form another link in the upper right corner and so are ❶❺ and ❶❼.

B.5 Liberty and Atari

B.5.1 Liberty

A liberty of a string is an empty point that is major-adjacent to that string. The definition of a *liberty-set* (**lib-set**) for a given string St is defined as :

$$\textbf{lib-set}(St) \equiv \{p | p \in \mathcal{P} \wedge \textbf{status}(p) = \text{empty} \ \wedge \exists s \in St, \textbf{maj-adj}(s, p)$$

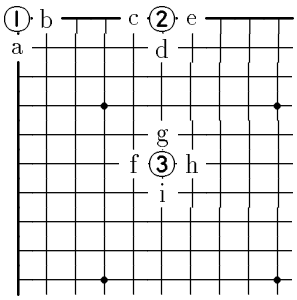


Figure B.6: Examples of liberty

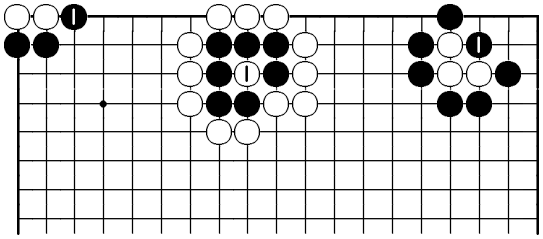


Figure B.7: Examples of capture

A point p is a liberty of a string St if and only if $p \in$ **lib-set**(St). The number of liberties for a string $St =$ | **lib-set** (St)|, where $|\cdot|$ is the set cardinality.

In figure B.6, stone ① has 2 liberties: a and b. Stone ② has 3 liberties: c, d and e. Stone ③ has 4 liberties: f, g, h and i.

When all the liberties of a string of color c_1 are filled by opponent stones (color $= c_2$) after a move made by the opponent, the string is **captured**. The string should then be removed from the board by the opponent (color $= c_2$) immediately after the move.

In figure B.7, at the left corner, the two white stones are captured after ❶. In the middle, the 7-stone black string is captured after ①. At the right corner, the 3 white stones are captured after ❶.

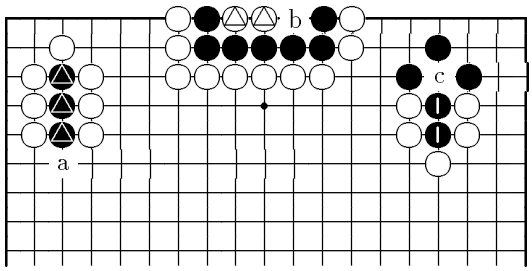


Figure B.8: Examples of Atari

B.5.2 Atari

A string is in a state of *Atari* if its number of liberty is only 1. The function **Atari** (St) states whether a string is in Atari :

$$\textbf{Atari}\ (St) \equiv (|\textbf{lib-set}(St)| = 1)$$

In figure B.8, the three black stones marked by △ in the left side are in Atari. The remaining liberty of the stones is at point a. In the middle, Both of the white string marked with △ and the black string are in Atari. Both of them have only one liberty at point b. At the right corner, two black stones marked with ❙ are in Atari with remaining liberty at point c.

B.6 Ko

Ko is a capture-and-recapture situation in a Go game such that if both players try to capture an opponent stone involved, the same situation will repeat forever.

In figure B.9, the basic shapes of Ko at different locations of the board are shown. Formally, Ko exists at a point p_1 of the board if the following conditions hold after a move $m_1 = (p_1, c_1, i)$ is played (move m_1 is the i-th move played by color c_1) :

Condition A (C_{a1}) The move m_1 can capture exactly one opponent stone (p_2, c_2).

Condition B (C_{b1}) The stone (p_1, c_1) does not major-connect to any peer stones.

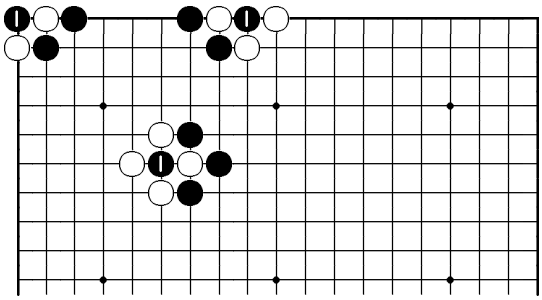


Figure B.9: Examples of simple Ko

Condition C (C_{c1}) Number of liberty for the stone (p_1, c_1) is *one* (after captured stone is removed)

We will try to prove that after move m_1 leading to condition (C_{a1}), (C_{b1}) and (C_{c1}), move $m_2 = (p_2, c_2, i+1)$ will lead to to condition (C_{a2}), (C_{b2}) and (C_{c2}) which are the same as the previous conditions after replacing 1 by 2 and 2 by 1 :

Condition A' (C_{a2}) The move m_2 can capture exactly one opponent stone (p_1, c_1).

Condition B' (C_{b2}) The stone (p_2, c_2) does not major-connect to any peer stones.

Condition C' (C_{c2}) Number of liberty for the stone (p_2, c_2) is **one** (after captured stone is removed)

That is :

$$(C_{a1})(C_{b1})(C_{c1}) \stackrel{(p_2,c_2,i+1)}{\longrightarrow} (C_{a2})(C_{b2})(C_{c2})$$

Without loss of generosity, we also get :

$$(C_{a2})(C_{b2})(C_{c2}) \stackrel{(p_1,c_1,i+2)}{\longrightarrow} (C_{a1})(C_{b1})(C_{c1})$$

since the move number $i+1$ and $i+2$ does not have any relation to the conditions. Repeatedly applying the two conditions, we can get

$$\ldots (C_{a1})(C_{b1})(C_{c1}) \stackrel{(p_2,c_2,i+1)}{\longrightarrow} (C_{a2})(C_{b2})(C_{c2}) \stackrel{(p_1,c_1,i+2)}{\longrightarrow} (C_{a1})(C_{b1})(C_{c1}) \ldots$$

which causes an infinite loop when play c_1 insists on playing at position p_1 and c_2 insists on playing at position p_2 and so on.

Now, we show under conditions $(C_{a1}), (C_{b1})$ and (C_{c1}) that move m_2 will lead to condition $(C_{a2}), (C_{b2})$ and (C_{c2}) :

1. By condition (C_{a1}), we know that move $m_1 = (p_1, c_1, i)$ captures stone (p_2, c_2). Therefore, p_1 and p_2 must be major-adjacent. By condition C_{b1}, we know that stone (p_1, c_1) is a 1-stone string. By condition C_{c1}, we know that stone (p_1, c_1) has only one liberty which must be at p_2 because p_1 and p_2 are major-adjacent. Therefore the move $(p_2, c_2, i + 1)$ will fill the last liberty of stone (p_1, c_1) and capture it. As a result condition C_{a2} holds.

2. Since m_1 capture (p_2, c_2) by condition (C_{a1}), p_1 must be the last liberty of stone (p_2, c_2) before move m_1. By condition C_{a1}, we know that move m_1 only captures stone (p_2, c_2). Since we just show that move m_2 only captures one stone (p_1, c_1), the new stone (p_2, c_2) after move m_2 still has only one liberty (at position p_1) because no other stones of color c_1 have been removed. Therefore condition C_{c2} holds.

3. Position p_2 is all surrounded by stones of color c_1 after move m_1 because by condition C_{a1}, move m_1 captures exactly one stone at p_2. After move m_2 captures one stone (p_1, c_1), all its major-adjacent points are all occupied by stones of color c_1 except at p_1 which is now an empty point after stone (p_1, c_1) is captured. As a result, stone (p_2, c_2) is not connecting to any peer stones of color c_2 and is therefore a 1-stone string. Consequently, condition C_{b2} holds.

In figure B.10, the examples shown are *not* Ko. At the upper left corner, white can play cat point a can capture 2 stones, violating condition (C_{a1}). In the middle, white can play at b and capture one black stone but the string containing stone at b will have 3 stones, violating condition (C_{b1}). At the right corner, after playing at c, the new white string will have 2 stones and violating condition (C_{b1}). At the lower left corner, after playing at d, the new white stone will have 2 liberties, violating condition (C_{c1}).

Since the shape satisfying all the above 3 conditions will loop forever if both player insist on capturing stones at p_1 and p_2 respectively, to avoid this situation goes on endlessly,

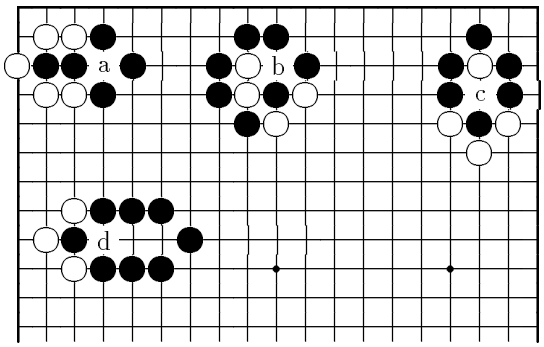


Figure B.10: Examples that are NOT Ko

the rule of Go prohibits this capture-and-recapture moves. The rule states that if it is a shape of Ko and player c_1 plays at p_1 and capture stone at p_2, player c_2 cannot play at p_2 immediately. Player c_2 should play at some other position. If player c_1 do not play a move to stop the Ko situation (for example, play at p_2 and fill the hole), player c_2 can then at $p2$. The similar prohibition for immediate re-capture will then apply on player c_1 after player c_2 plays at p_2.

B.7 Prohibited Move

A move is prohibited if it is in one of the following conditions :

1. **Committing suicide** : A move $m = (p, c, i)$ is committing suicide if after the move m is played and stones captured by m (if any) are removed, there is a string St on the board such that **lib-set** $(St) = \{\}$. Figure B.11 shows four examples of committing suicide. All moves marked by ❶ are prohibited because they make some of the black stones out of liberty after playing the move.

2. **Immediate Ko take-back** : recapture of a stone in Ko immediate, as mentioned in last section.

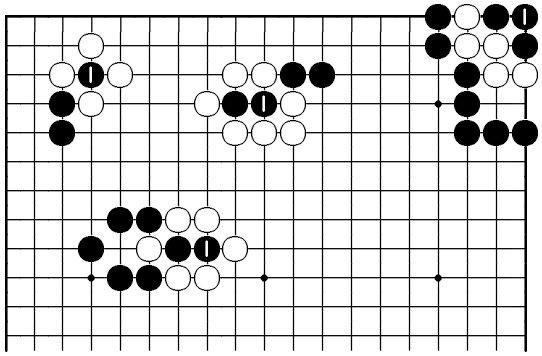


Figure B.11: Examples of prohibited moves

B.8 Path and Distance

The path and distance between two points are useful in computing radiation of influence from a stone to the outside area of the board [10].

1. A **major-path** between 2 points is a list of empty points that are **major-connected** together one after another. We define the set of all possible **major-path**s here:

$$\begin{aligned}\textbf{maj-path-set}(p_a, p_b) = \{(p_1, p_2, \ldots, p_n) |\ & (\forall i, 1 \leq i \leq n, \textbf{status}(p_i) = \text{ empty}) \ \wedge \\ & (\forall j, 1 \leq j < n, \textbf{maj-adj}(p_j, p_{j+1})) \ \wedge \\ & \textbf{maj-adj}(p_a, p_1) \wedge \textbf{maj-adj}(p_n, p_b) \ \wedge \\ & (p_1, p_2, \ldots, p_n) \text{ is a } \textbf{simple path}\}\end{aligned}$$

   where $(p_1, p_2, \ldots, p_n)$ is a **simple path** iff $\forall i, \forall j, 1 \leq i, j \leq n, i \neq j \rightarrow p_i \neq p_j$

   The **maj-path-set** is a finite set since all paths are cycle-free and the board size is finite.

2. The **major-distance** between 2 point is the cardinality of the shortest ordered list in the set of all **major-path**s.

   $\textbf{maj-distance}(p_a, p_b) = |l|$

   where $l \in \textbf{maj-path-set}(p_a, p_b) \ \wedge$

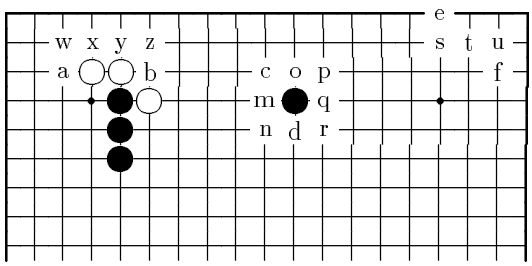


Figure B.12: Examples of major-path and major-distance

$$\forall l' \in \textbf{maj-path-set}(p_a, p_b), |l| \leq |l'|$$

The corresponding list, l forms the **shortest major-path**. Both **major-path** and **major-distance** are used to define the influence radiation of a stone to other empty space on a Go board.

In figure B.12, (w, x, y, z) forms a major-path between points a and b. The path is the shortest possible one. So the major-distance between a and b is 4.

(m, n) and (o, p, q, r) are 2 major-paths between points c and d. (m, n) is the shortest possible path. Therefore, the major-distance is 2.

(s, t, u) is one of the shortest major-path between points e and f. The major distance between e and f is 3.

3. The **effective-major-path** is a list points between 2 peer stones. Each point is either empty or occupied by a peer stone. The points p_i and p_{i+1} are **major-connected**. We define the set of all possible **effective-major-path**s here : is the set of all ordered lists of points which is defined as :

$$\begin{aligned}\textbf{eff-maj-path-set}(s_a, s_b) = \{(p_1, p_2, \ldots, p_n) | & c_a = c_b \quad \wedge \\ & \forall i, 1 \leq i \leq n, \textbf{status}\ (p_i) = \{c_a, \text{empty}\} \quad \wedge \\ & \forall j, 1 \leq j < n, \textbf{maj-adj}\ (p_j, p_{j+1}) \quad \wedge \\ & \textbf{maj-adj}(p_a, p_1) \wedge \textbf{maj-adj}(p_n, p_b) \quad \wedge \\ & (p_1, p_2, \ldots, p_n) \text{ is a } \textbf{simple path} \quad \}\end{aligned}$$

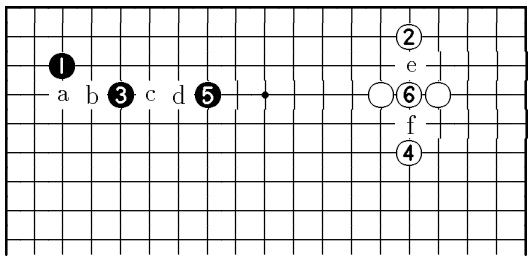


Figure B.13: Examples of effective major-path and major-distance

where $s_a = (p_a, c_a)$ and $s_b = (p_b, c_b)$

Along one of the **effective-major-path**, each point is either empty or occupied by a peer stone.

4. The **effective-major-distance** between 2 stones is defined as :

   eff-major-distance$(s_a, s_b) = \langle l \rangle$

   where $l \in$ **eff-maj-path-set**(p_a, p_b) $\wedge$

   $\langle l \rangle = |\{p_i \in l | \textbf{status}(p_i) = \text{empty}\}|$ $\wedge$

   $\forall l' \in$ **eff-maj-path-set**$(p_a, p_b), \langle l \rangle | \leq \langle l' \rangle$

   The corresponding list, l forms the **shortest effective-major-path**.

   In figure B.13, the path $(a, b, ❸, c, d)$ forms an effective major-path between stones ❶ and ❺. $(e, ⑥, f)$ is an effective major-path between ② and ④.

5. A **string-path** between 2 peer strings is the **effective-major-path** from a stone in one string to a stone from another string. We define the set of all possible **string-path**s :

   str-path-set$(St_1, St_2) = \bigcup_{\forall s_i \in St_1, \forall s_j \in St_2}$ **eff-maj-path**(s_1, s_2)

6. The **string-distance** between 2 strings is defined as :

   str-d$(St_1, St_2) = \langle l \rangle$

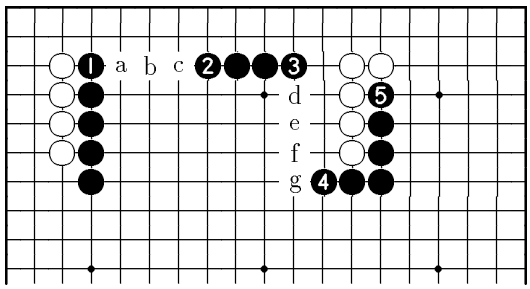


Figure B.14: Examples of string major-path and major-distance

where $l \in$ **str-path-set**(St_1, p_2) $\wedge$

$\forall l' \in$ **str-path-set**$(St_1, St_2), \langle l \rangle \leq \langle l' \rangle$

The corresponding list, l forms the **shortest string-path**.

In figure B.14, (a, b, c) forms a shortest string path between two black strings, one containing ❶ and another containing ❷ and ❸. The string distance is 3.

(d, e, f, g) forms another shortest string path between two black strings, one containing ❷,❸ and another containing ❹,❺. The shortest effective major-path is formed between ❸ and ❹ because between ❸ and ❺ there is some white stones blocking them. The string distance of these 2 strings is 4.

To prove that the above situation will lead to a infinite loop of move sequence : $(\ldots, (p_1, c_1, i),\ (p_2, c_2, i+1),\ (p_1, c_1, i+2),\ (p_1, c_1, i), \ldots)$ it is sufficient to show that a new move $m_2 = (p_2, c_2, i+1)$ will cause the following new conditions :

Bibliography

[1] Emile H. L. Aarts and Jan Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing.* Chichester ; New York : Wiley, 1989.

[2] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, Maastricht, July 1994.

[3] Etienne Barnard. Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2), March/April 1993.

[4] Elwyn R. Berlekamp and David Wolfe. *Mathematical Go Endgames: Nightmares for the Professional Go Players.* Ishi Press International, 1994.

[5] Richard Bozulich. *The second book of Go.* Ishi Press International, 1987.

[6] J. Burmeister and J Wiles. The challenge of Go as a domain: A comparison between Go and Chess. In *Proceeding of the Third Australian and New Zealand Conference on Intelligent Information Systems*, pages 181–186. IEEE Western Australia Section, November 1995.

[7] Wai-kit Chan and Irwin King. Temporal difference learning and its application in Go. In *Computer Strategy Game Programming Workshop.* The Chinese University of Hong Kong, May 1995.

[8] Wai-kit Chan, Irwin King, and C.S. Lui. Performance analysis of a new updating rule for TD (λ) learning in feedforward networks for position evaluation in Go game. In *Proceedings of ICNN 96*, volume 3, pages 1716–1720. IEEE, June 1996.

[9] W. G. Chase and H. A. Simon. Perception in Chess. *Cognitive Psychology*, 4:55–81, 1973.

[10] Ken Chen. Group identification in computer Go. In D.F.Beal Levy, D.N.L., editor, *Heuristic Programming in Artificial Intelligence, the first Computer Olympaid*, pages 195–210. Computer Olympaid, Ellis Horwood, September 1989.

[11] Ken Chen. The move decision process of Go Intellect. *Computer Go*, pages 9–17, 1990.

[12] Chi-hun Cho. *The magic of Go: a complete introduction to the game of Go.* Ishi Press, Tokyo, 1988.

[13] G. Cybenko. Approximation by superpositions of a Sigmoid function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[14] David Fotland. Knowledge representation in the Many Faces of Go. available through FTP from bsdserver.ucsf.edu /pub/Go/comp, February 1993.

[15] R. Gasser and G. Nievergelt. Es ist entschieden: Das mühlespiel ist unentschieden. *Informatik Spektrum*, 17(5):314–317, 1994.

[16] S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1984.

[17] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Macmillian, 1994.

[18] K. Iwamoto. *Go for Beginners.* Ishi Press, Tokyo, 1972.

[19] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[20] T. Ragg, H. Braun, and J. Feulner. Improving temporal difference learning for deterministic sequential decision problems. In F. Fogelman-Soulie and P. Gallinari, editors, *ICANN '95 : proceedings of the International Conference on Artificial Neural Networks*, volume 2, pages 117–122, October 1995.

[21] J. S. Reitman. Skilled perception in Go: Deducing memory structures from inter-response times. *Cognitive Psychology*, 8, 1976.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing (PDP): Exploration in Microstructure of Recognition (Vol. 1)*, chapter 8. MIT Press, Cambridge, Massachusetts, 1986.

[23] A.P. Russo. Neural networks for sonar signal processing, tutorial no. 8. In *IEEE Conference on Neural Networks for Ocean Engineering*, 1991.

[24] Nicol N. Schraudolph, Peter Dayan, and T.J. Sejnowski. Temporal difference learning of positon evaluation in the game of Go. In Cowanm J.D., G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing*, volume 6. Morgan Kaufmann, San Francisco, 1994.

[25] Richard S. Sutton. Learning to predict by the methods of temproal differences. *Machine Learning*, 3:9–44, 1988.

[26] G. Tesauro. Neurogammon: a neural network Backgammon program. In *IJCNN Proceedings, III*, pages 33–39, 1990.

[27] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–278, 1992.

[28] G. Tesauro. TD-gammon, a self-teaching Backgammon program, achieves master-level play. *Neural Computation*, 6(2), 1994.

[29] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of ACM*, 38(3), March 1995.

[30] Bruce Wilcox. Reflections on building two Go programs. *SIGART Newsletter*, 94:29–43, October 1985.

[31] A. Zobrist. *Feature Extraction and Representation for Pattern Recognition and the game of Go*. PhD thesis, University of Wisconson, 1970.