Abstract

In this thesis we develop a unified framework for reinforcement learning and simulation-based search. We introduce the *temporal-difference search* algorithm, a new approach to search that is able to generalise between related states, and efficiently update the values of those states. The temporal-difference learning algorithm acquires general domain knowledge from its past interactions with the world. In contrast, the temporal-difference search algorithm acquires local knowledge that is specialised to the current state, by simulating future interactions with the world. We combine both forms of knowledge together in the *Dyna-2* architecture.

We apply our algorithms to the game of 9×9 Go. Using temporal-difference learning, with a million binary features matching simple patterns of stones, we learnt the strongest static evaluation function without significant prior knowledge of the game. Using temporal-difference search with the same representation produced a dramatic improvement: without any explicit search tree, and with equivalent domain knowledge, it achieved better performance than a Monte-Carlo tree search. When combined together using the Dyna-2 architecture, our Go program outperformed all handcrafted programs, traditional search programs, and traditional machine learning approaches.

We also use our framework to extend the Monte-Carlo tree search algorithm. By generalising between related parts of the search tree, and incorporating knowledge learnt offline, we were able to significantly improve the performance of the Go program *MoGo*. Using these enhancements, *MoGo* became the first 9×9 Go program to achieve human master level.

Acknowledgements

The following document uses the first person plural to indicate the collaborative nature of much of this work. In particular, Rich Sutton has been a constant source of inspiration and wisdom. Martin Müller has provided invaluable advice on many topics, and his formidable Go expertise has time and again proven to be priceless. I'd also like to thank Gerry Tesauro for his keen insights and many constructive suggestions.

The results presented in this thesis were generated using the Computer Go programs *RLGO* and *Mogo*. The *RLGO* program was developed on top of the SmartGo library by Markus Enzenberger and Martin Müller. I would like to acknowledge contributions to *RLGO* from numerous individuals, including Anna Koop and Leah Hackman. In addition I'd like to thank the members of the Computer Go Mailing list for their feedback and ideas.

The *Mogo* progam was originally developed by Sylvain Gelly and Yizao Wang at the University of South Paris, with research contributions from Remi Munos and Olivier Teytaud. The heuristic UCT and RAVE algorithms described in Chapter 9 were developed in collaboration with Sylvain Gelly, who should receive most of the credit for developing these ideas into practical and effective techniques. Subsequent work on *Mogo*, including massive parallelisation and a number of other improvements, has been led by Olivier Teytaud and his team at the University of South Paris, but includes contributions from Computer Go researchers around the globe.

I'd like to thank Jessica Meserve for her enormous depths of patience, love and support that have gone well beyond reasonable expectations. Finally, I'd like to thank Elodie Silver for bringing joy and balance to my life. Writing this thesis has never been a burden, when I know I can return home to your smile.

Table of Contents

1 Introduction **1**

1.1 Computer Go 1
1.2 Reinforcement Learning 2
1.3 Simple Ideas for Big Worlds 2
1.3.1 Value function 3
1.3.2 State abstraction 3
1.3.3 Temporality 3
1.3.4 Bootstrapping 3
1.3.5 Sample-based planning 4
1.4 Game-Tree Search 4
1.4.1 Alpha-beta search 4
1.4.2 Monte-Carlo tree search 5
1.5 Overview 6

I Literature Review **9**

2 Reinforcement Learning **10**

2.1 Learning and Planning 10
2.2 Markov Decision Processes 10
2.3 Value-Based Reinforcement Learning 11
2.3.1 Dynamic Programming 12
2.3.2 Monte-Carlo Evaluation 12
2.3.3 Temporal Difference Learning 13
2.3.4 TD(λ) 13
2.3.5 Sarsa(λ) 14
2.3.6 Value Function Approximation 14
2.3.7 Linear Temporal-Difference Learning 15
2.4 Policy Gradient Reinforcement Learning 15
2.5 Exploration and Exploitation 17

3 Search and Planning 18
3.1 Introduction . . . 18
3.2 Planning . . . 18
3.2.1 Model-Based Planning . . . 18
3.2.2 Sample-Based Planning . . . 19
3.2.3 Dyna . . . 19
3.3 Search . . . 19
3.3.1 Full-Width Search . . . 19
3.3.2 Sample-Based Search . . . 20
3.3.3 Simulation-Based Search . . . 21
3.3.4 Monte-Carlo Simulation . . . 21
3.3.5 Monte-Carlo Tree Search . . . 21

4 Computer Go 24
4.1 The Challenge of Go . . . 24
4.2 The Rules of Go . . . 24
4.3 Go Ratings . . . 26
4.4 Position Evaluation in Computer Go . . . 26
4.5 Static Evaluation in Computer Go . . . 27
4.5.1 Symmetry . . . 27
4.5.2 Handcrafted Heuristics . . . 28
4.5.3 Temporal Difference Learning . . . 28
4.5.4 Comparison Training . . . 29
4.5.5 Evolutionary Methods . . . 29
4.6 Dynamic Evaluation in Computer Go . . . 30
4.6.1 Alpha-Beta Search . . . 30
4.6.2 Monte Carlo Simulation . . . 30
4.6.3 Monte-Carlo Tree Search . . . 31
4.7 Summary . . . 33

II Temporal Difference Learning and Search 34

5 Temporal Difference Learning with Local Shape Features 35
5.1 Introduction . . . 35
5.2 Shape Knowledge in Go . . . 37
5.3 Local Shape Features . . . 37
5.4 Weight Sharing . . . 39
5.5 Learning algorithm . . . 39

5.6 Training . . . 41
5.7 A Case Study in 9×9 Computer Go . . . 41
5.7.1 Experimental Setup . . . 42
5.7.2 Local Shape Features in 9×9 Go . . . 43
5.7.3 Weight Evolution . . . 45
5.7.4 Logistic Temporal-Difference Learning . . . 45
5.7.5 Self-Play . . . 46
5.7.6 Logistic TD(λ) . . . 47
5.7.7 Extended Representations in 9×9 Go . . . 48
5.7.8 Alpha-Beta Search . . . 49
5.8 Conclusion . . . 49

6 Temporal-Difference Search 52
6.1 Introduction . . . 52
6.2 Temporality . . . 53
6.3 Simulation-Based Search . . . 54
6.4 Monte-Carlo Search . . . 54
6.5 Temporal-Difference Search . . . 55
6.6 Temporal-Difference Search in Computer Go . . . 56
6.7 Experiments in 9×9 Go . . . 59
6.7.1 Default Policy . . . 59
6.7.2 Local Shape Features . . . 61
6.7.3 Parameter Study . . . 62
6.7.4 TD(λ) Search . . . 62
6.7.5 Temporality . . . 63
6.8 Conclusion . . . 64

7 Long and Short-Term Memories 65
7.1 Introduction . . . 65
7.2 Long and Short-Term Memories . . . 66
7.3 Dyna-2 . . . 66
7.4 Dyna-2 in Computer Go . . . 70
7.5 Dyna-2 and Heuristic Search . . . 72
7.6 Conclusion . . . 75

III Monte-Carlo Tree Search 76

8 Reusing Local Search Trees in Simulation-Based Search 77

8.1 Introduction 77
8.2 Local and Global Search in Computer Go 77
8.3 Local Monte-Carlo Search 79
8.3.1 State representation 79
8.3.2 Position evaluation 80
8.3.3 Simulation 81
8.4 Local Regions in Computer Go 81
8.5 Experiments in 9×9 Computer Go 82
8.6 Scalability 85
8.7 Related work 86
8.8 Conclusions 86

9 Heuristic MC–RAVE **87**
9.1 Introduction 87
9.2 Monte-Carlo Simulation 87
9.3 Monte-Carlo Tree Search 88
9.4 Heuristic Prior Knowledge 89
9.5 Rapid Action Value Estimation (RAVE) 89
9.6 MC–RAVE 91
9.7 Minimum MSE Schedule 93
9.7.1 Assumptions 93
9.7.2 Derivation 93
9.8 Heuristic MC–RAVE 95
9.9 Exploration and Exploitation 96
9.10 Heuristics and RAVE in Dyna-2 97
9.10.1 Heuristic Monte-Carlo tree search in Dyna-2 97
9.10.2 RAVE in Dyna-2 97
9.11 Conclusions 98

10 Monte-Carlo Simulation Balancing **101**
10.1 Introduction 101
10.2 Learning a Simulation Policy in 9×9 Go 102
10.2.1 Stochastic Simulation Policies 102
10.2.2 Strength of Simulation Policies 102
10.2.3 Accuracy of Simulation Policies in Monte-Carlo Simulation 103
10.3 Strength and Balance 104
10.4 Softmax Policy 107
10.5 Optimising Strength 108

10.5.1 Apprenticeship Learning 108
10.5.2 Policy Gradient Reinforcement Learning 109
10.6 Optimising Balance 109
10.6.1 Policy Gradient Simulation Balancing 109
10.6.2 Two-Step Simulation Balancing 110
10.7 Experiments in Computer Go 111
10.7.1 Balance of shapes 114
10.7.2 Mean squared error 114
10.7.3 Performance in Monte-Carlo search 115
10.8 Conclusions 116

11 Conclusions **118**
11.1 The Future of Computer Go 118
11.2 The Future of Sequential Decision-Making 119

Bibliography **120**

A Logistic Temporal Difference Learning **126**
A.1 Logistic Monte-Carlo Evaluation 126
A.2 Logistic Temporal-Difference Learning 127

List of Tables

4.1 Approximate Elo ratings of the strongest 9×9 programs using various paradigms on the 9x9 Computer Go Server. 33
4.2 Approximate Elo ratings of the strongest 19x19 Go programs using various paradigms on the Kiseido Go Server. 33

5.1 Number of local shape features of different sizes in 9×9 Go. 40
5.2 Performance of static evaluation function in alpha-beta search 49
5.3 Elo ratings established by *RLGO v1.0* on the Computer Go Server. 49

7.1 The Elo rating established by RLGO on the Computer Go Server. 72

9.1 Winning rate of *MoGo* against GnuGo 3.7.10 (level 10) when the number of simulations per move is increased 98

10.1 Winning rate of the basic UCT algorithm in *MoGo* against GnuGo 104
10.2 Elo rating of simulation policies in 5×5 Go and 6×6 Go tournaments 115

List of Figures

3.1 Five simulations of Monte-Carlo tree search. 23

4.1 The rules of Go 25
4.2 Performance ranks in Go, in increasing order of strength from left to right. 25

5.1 Shape Knowledge in Go 36
5.2 Location dependent and location independent weight sharing 38
5.3 Evaluating a position by local shape features 41
5.4 Multiple runs using the default learning algorithm and local shape features. 42
5.5 Histogram of feature occurrences during a training run of 1 million games. 43
5.6 Learning curves for local shape features 44
5.7 Weight evolution during training 45
5.8 Comparison of linear and logistic-linear temporal-difference learning 46
5.9 Learning curves for one-ply and two-ply updates (left), and for different exporation rates ϵ (right). 46
5.10 Learning curves for different values of λ, using accumulating traces (left) and replacing traces (right). 47
5.11 Differentiating the colour to play (left) and differentiating the stage of the game, by binning the move number at a variety of resolutions (right) 48
5.12 Local shapes with greatest absolute weight after training on a 9×9 board 50
5.13 Two example games played online 50

6.1 Temporal-difference search applied to Computer Go. 57
6.2 Generalisation in temporal-difference search 57
6.3 The *blood vomiting game* 58
6.4 Comparison of temporal-difference search and UCT (left). Performance of temporal-difference search when switching to the *Fuego* default policy (right). The number of moves at which the switch occurred was varied between 1 and 64. 60
6.5 Performance of temporal-difference search with different sizes of local shape feature 61
6.6 Performance of temporal-difference search with different learning rates α (left) and exploration rates ϵ (right). 62

6.7 Performance of TD(λ) search for different values of λ, using accumulating traces (left) and replacing traces (right). 63
6.8 Temporal-difference search with an old value function 63

7.1 The Dyna-2 architecture 68
7.2 Examples of long and short-term memories in 5×5 Go 70
7.3 Winning rate of RLGO against GnuGo in 9×9 Go, using Dyna-2-Shape for different simulations/move 71
7.4 Winning rate of RLGO against GnuGo in 9×9 Go, using a hybrid search based on both Dyna-2 and alpha-beta 73

8.1 Reusing local searches in 9×9 Go 78
8.2 Two methods for partitioning the board in the game of Go 82
8.3 The performance of local temporal-difference tree search against *GnuGo* in 9×9 Go 83
8.4 A comparison of local Monte-Carlo tree search and local temporal-difference tree search 84
8.5 The performance of local Monte-Carlo tree search and local temporal-difference tree search for different sizes of Go board 85

9.1 An example of using the RAVE algorithm to estimate the value of black c3 90
9.2 Winning rate of MC–RAVE with 3000 simulations per move against GnuGo, for different settings of the equivalence parameter 91
9.3 Winning rate of *MoGo*, using the heuristic MC–RAVE algorithm, with 3000 simulations per move against GnuGo 95
9.4 Scalability of *MoGo* 99

10.1 The relative strengths of each class of default policy, against the random policy and against a handcrafted policy 103
10.2 The MSE of each policy π when Monte Carlo simulation is used to evaluate a test suite of 200 hand-labelled positions 103
10.3 Monte-Carlo simulation in an artificial two-player game 105
10.4 Weight evolution for the 2×2 local shape features 112
10.5 Monte-Carlo evaluation accuracy of different simulation policies in 5×5 Go (top) and 6×6 Go 113

Chapter 1

Introduction

This thesis investigates the game of Go as a case study for artificial intelligence in large, challenging domains.

1.1 Computer Go

Computer Go provides an interesting case study for AI precisely because it represents the best-case for AI. The rules of the game are known, deterministic[1], and extremely simple. The state is fully observable; the state space and action space are both discrete. Games are of finite length, and always terminate with a clearly defined win or loss outcome. The state changes slowly and incrementally, with a single stone added at every move[2]. The expertise of human players provides a proof of existence that a finite and compact program for playing strong Go must exist. And yet until recently, and despite significant effort, Computer Go has resisted significant progress, and was viewed by many as an almost impossible challenge.

Certainly, the game of Go is big. It has approximately 10^{170} states and up to 361 actions. Its enormous search space is orders of magnitude too big for the search algorithms that have proven so successful in Chess and Checkers. Although the rules are simple, the emergent complexity of the game is profound. The long-term effect of a move may only be revealed after 50 or 100 additional moves. Professional Go players accumulate Go knowledge over a lifetime; mankind has accumulated Go knowledge over several millennia. For the last 30 years, attempts to precisely encode this knowledge in machine usable form have led to a positional understanding that is at best comparable to weak-amateur level humans. But are these properties really exceptional to Go? In real-world planning and decision-making problems, most actions have delayed, long-term consequences, leading to surprising complexity and enormous search-spaces that are intractable to traditional search algorithms. Furthermore, also just like Go, in the majority of these problems, expert knowledge is either unavailable, unreliable, or unencodable.

[1] In practice both the opponent's unknown policy, and the agent's own stochastic policy, provide elements of randomness.
[2] Except for captures, which occur relatively rarely.

So before we consider any broader challenges in artificial intelligence, and attempt to tackle continuous action, continuous state, partially observable, and infinite horizon problems, let's consider Computer Go.

1.2 Reinforcement Learning

Reinforcement learning is the study of optimal decision-making in natural and artificial systems. In the field of artificial intelligence, it has been used to defeat human champions at games of skill (Tesauro, 1994); in robotics, to fly stunt manoeuvres in robot-controlled helicopters (Abbeel et al., 2007). In neuroscience it is used to model the human brain (Schultz et al., 1997); in psychology to predict animal behaviour (Sutton and Barto, 1990). In economics, it is used to understand the decisions of human investors (Choi et al., 2007), and to build automated trading systems (Nevmyvaka et al., 2006). In engineering, it has been used to allocate bandwidth to mobile phones (Singh and Bertsekas, 1997) and to manage complex power systems (Ernst et al., 2005).

A reinforcement learning task requires decisions to be made over many time steps. At each step an agent selects actions (e.g. motor commands); receives observations from the world (e.g. robotic sensors); and receives a reward indicating its success or failure (e.g. a negative reward for crashing). Given only its actions, observations and rewards, how can an agent improve its performance?

Reinforcement learning (RL) can be subdivided into two fundamental problems: *learning* and *planning*. The goal of learning is for an agent to improve its policy from its interactions with the world. The goal of planning is for an agent to improve its policy without further interaction with the world. The agent can deliberate, reason, ponder, think or search, so as to find the best behaviour in the available computation time.

Despite the apparent differences between these two problems, they are intimately related. During learning, the agent interacts with the real world, by executing actions and observing their consequences. During planning the agent can interact with a *model* of the world: by simulating actions and observing their consequences. In both cases the agent updates its policy from its experience. Our thesis is that an agent can both learn and plan effectively using reinforcement learning algorithms.

1.3 Simple Ideas for Big Worlds

Artificial intelligence research often focuses on toy domains: small microworlds which can be easily understood and implemented, and are used to test, compare, and develop new ideas and algorithms. However, the simplicity of toy domains can also be misleading: many sophisticated ideas that work well in small worlds do not, in practice, scale up to larger and more realistic domains. In contrast, big worlds act as a form of Occam's razor, with only the simplest and clearest ideas achieving success. This can be true not only in terms of memory and computation, but also in terms of the practical challenges of implementing, debugging and testing a large program in a challenging domain.

In this thesis we combine five simple ideas for achieving high performance in big worlds. Four of the five ideas are well-established in the reinforcement learning community; the fifth idea of *temporality* is a contribution of this thesis. All five ideas are brought together in the *temporal-difference search* algorithm (see Chapter 6).

1.3.1 Value function

The *value function* estimates the expected outcome from any given state, after any given action. The value function is a crucial component of efficient decision-making, as it summarises the long-term effects of the agent's decisions into a single number. The best action can then be selected by simply maximising the value function.

1.3.2 State abstraction

In large worlds, it is not possible to store a distinct value for every individual state. *State abstraction* compresses the state into a smaller number of features, which are then used in place of the complete state. Using state abstraction, the value function can be approximated by a much smaller number of parameters. Furthermore, state abstraction enables the agent to *generalise* between related states, so that a single outcome can update the value of many states.

1.3.3 Temporality

In very large worlds, state abstraction cannot provide an accurate approximation to the value function. For example, there are 10^{170} states in the game of Go. Even if the agent can store 10^{10} parameters, it is compressing the values of 10^{160} states into every parameter. The idea of *temporality* is to focus the agent's representation on the *current* region of the state space, rather than attempting to approximate the entire state space. This thesis introduces the idea of temporality as an explicit concept for reinforcement learning and artificial intelligence.

1.3.4 Bootstrapping

Large problems typically entail making decisions with long-term consequences. Hundreds or thousands of time-steps may elapse before the final outcome is known. These outcomes depend on all of the agent's decisions, and on the world's uncertain responses to those decisions, throughout all of these time-steps. *Bootstrapping* provides a mechanism for reducing the variance of the agent's evaluation. Rather than waiting until the final outcome is reached, the idea of bootstrapping is to make an evaluation based on subsequent evaluations. For example, the *temporal-difference learning* algorithm estimates the current value from the estimated value at the next time-step.

1.3.5 Sample-based planning

The agent's experience with its world is limited, and may not be sufficient to achieve good performance in the world. The idea of *sample-based planning* is to simulate hypothetical experience, using a *model* of the world. The agent can use this simulated experience, in place of or in addition to its real experience, to learn to achieve better performance.

1.4 Game-Tree Search

The challenge of search is to find, by a process of computation, the optimal action from some root state. The importance of search is most clearly demonstrated in two-player games, where *game-tree search* algorithms such as alpha-beta search and Monte-Carlo tree search have achieved remarkable success.

1.4.1 Alpha-beta search

In classic games such as Chess (Campbell et al., 2002), Checkers (Schaeffer et al., 1992) and Othello (Buro, 1999), traditional search algorithms have exceeded human levels of performance. In each of these games, master-level play has also been achieved by a reinforcement learning approach (Baxter et al., 1998; Schaeffer et al., 2001; Buro, 1999):

- A position is broken down into many features, each of which identifies a particular piece or configuration of pieces.
- Positions are evaluated by summing the values of all features that are present in the current position.
- The value of each feature is learned offline, from many training games of self-play.
- The learned evaluation function is used in a high-performance alpha-beta search.

Despite these impressive successes, there are many domains in which traditional search methods have failed. In very large domains, it is often difficult to construct an evaluation function with any degree of accuracy. We cannot reasonably expect to accurately approximate the value of all distinct states in the game of Go; all attempts to do so have achieved a position evaluation that, at best, corresponds to weak-amateur level humans (Müller, 2002).

We introduce a new approach to position evaluation in large domains. Rather than trying to approximate the entire state space, our idea is to specialise the evaluation function to the current region of the state space. Instead of evaluating every position that could ever occur, we focus on the positions that are likely to occur in the subgame starting from *now*. In this way, the evaluation function can represent much more detailed knowledge than would otherwise be possible, and can adapt to the nuances and exceptional circumstances of the current position. In Chess, it could know

that the black Rook should defend the unprotected queenside and not be developed to the open file; in Checkers that a particular configuration of checkers is vulnerable to the opponent's dynamic King; or in Othello that two adjacent White discs at the top of the board give a crucial advantage in the embattled central columns.

We implement this new idea by a simple modification to the above framework:

- The value of each feature is learned *online*, from many training games of self-play *from the current position*.

In prior work on learning to evaluate positions, the evaluation function was trained offline, typically over weeks or even months of computation (Tesauro, 1994; Enzenberger, 2003). In our approach, this training is performed in real-time, in just a few seconds of computation. At the start of each game the evaluation function is initialised to the best global weights. But after every move, the evaluation function is retrained online, from games of self-play that start from the current position. In this way, the evaluation function evolves dynamically throughout the course of the game, specialising more and more to the particular tactics and strategies that are relevant to *this* game and *this* position. We demonstrate that this approach can provide a dramatic improvement to the quality of position evaluation; in 9×9 Go it increased the performance of our alpha-beta search program by 800 Elo points in competitive play (see Chapter 7).

1.4.2 Monte-Carlo tree search

Monte-Carlo tree search is a new paradigm for search, which has revolutionised Computer Go, and is rapidly replacing traditional search algorithms as the method of choice in challenging domains. Many thousands of games are simulated from the current position, using self-play. New positions are added into a search tree, and each node of the tree contains a value that predicts whether the game will be won or lost from that position. These predictions are learnt by Monte-Carlo simulation: the value of a node is simply the average outcome of all simulated games that visit the position. The search tree is used to guide simulations along promising paths, by selecting the child node with the highest value. This results in a highly selective search that very quickly identifies good move sequences.

The evaluation function of Monte-Carlo tree search is grounded in experience: it depends only on the observed outcomes of simulations, and does not require any human knowledge. Additional simulations continue to improve the evaluation function; given infinite memory and computation, it will converge on the true minimax value (Kocsis and Szepesvari, 2006). Furthermore, also unlike full-width search algorithms such as alpha-beta search, Monte-Carlo tree search is highly selective, and develops the most promising regions of the search space much more deeply.

However, despite its revolutionary impact, Monte-Carlo tree search suffers from a number of serious deficiencies:

- The first time a position is encountered, its value is completely unknown.
- Each position is evaluated independently, with no generalisation between similar positions.
- Many simulations are required before Monte-Carlo can establish a high confidence estimate of the value.
- The overall performance depends critically on the rollout policy used to complete simulations.

This thesis extends the core concept of Monte-Carlo search into a new framework for simulation-based search, which specifically addresses these weaknesses:

- New positions are assigned initial values using a learned, global evaluation function (Chapters 7, 9).
- Positions are evaluated by a linear combination of features (Chapters 6, 8, 7), or by generalising between the value of the same move in similar situations (Chapter 9).
- Positions are evaluated by applying temporal-difference learning, rather than Monte-Carlo, to the simulations (Chapters 6, 7).
- The rollout policy is learnt and optimised automatically by simulation balancing (Chapter 10).

1.5 Overview

In the first part of the thesis, we survey the relevant research literature:

- In Chapter 2 we review the key concepts of reinforcement learning.
- In Chapter 3 we review sample-based planning and simulation-based search methods.
- In Chapter 4 we review the recent history of Computer Go, focusing in particular on reinforcement learning approaches and the Monte-Carlo revolution.

In the second part of the thesis, we introduce our general framework for learning and search.

- In Chapter 5 we investigate how a position evaluation function can be learned for the game of Go, with no prior knowledge except for the basic grid structure of the board. We introduce the idea of *local shape features*, which abstract the state into a large vector of binary features, and we use temporal-difference learning to train the weights of these features. Using this approach, we were able to learn the strongest static position evaluation function without significant prior knowledge.

- In Chapter 6 we develop temporal-difference learning into a high-performance search algorithm. The *temporal-difference search* algorithm is a new approach to simulation-based search that uses state abstraction and bootstrapping to search more efficiently in large domains. We demonstrate that temporal-difference search substantially outperforms temporal-difference learning in 9×9 Go. In addition, we show that temporal-difference search, without any explicit search tree, outperforms a Monte-Carlo tree search with equivalent domain knowledge.

- In Chapter 7 we combine temporal-difference learning and temporal-difference search, using long and short-term memories, in the *Dyna-2* architecture. We implement Dyna-2, using local shape features in both the long and short-term memories, in our Go program *RLGO*. We also introduce a *hybrid search* that combines Dyna-2 with alpha-beta. Using Dyna-2 in 9×9 Go, *RLGO* outperformed all handcrafted programs, traditional search programs, and traditional machine learning approaches; the hybrid search performed even better.

In the third part of the thesis, we apply our general framework to Monte-Carlo tree search.

- In Chapter 8 we extend the representation used by temporal-difference search to include local search trees. We introduce the *local Monte-Carlo tree search* and *local temporal-difference tree search* algorithms, in which the global position is represented by the local states in a number of overlapping search trees. We demonstrate that local temporal-difference tree search outperforms a global Monte-Carlo tree search in 9×9 Go, and scales better with increasing board sizes.

- In Chapter 9 we introduce two extensions to Monte-Carlo tree search. The *RAVE* algorithm rapidly generalises between related parts of the search-tree. The *heuristic Monte-Carlo tree search* algorithm incorporates prior knowledge into the nodes of the search-tree. The new algorithms were implemented in the Monte-Carlo program *MoGo*. Using these extensions, *MoGo* became the first program to achieve *dan*-strength at 9×9 Go, and the first program to beat a professional human player at 9×9 Go. *MoGo* became the highest rated program on the Computer Go Server for both 9×9 and 19×19 Go, and won the gold medal at the 2007 Computer Go Olympiad.

- In Chapter 10 we introduce the paradigm of *Monte-Carlo simulation balancing*, and develop the first algorithms for optimising the performance of a rollout policy in Monte-Carlo search. On small boards, given equivalent representations and equivalent training data, we demonstrate that rollout policies learned by our new paradigm exceed the performance of both supervised learning and reinforcement learning paradigms, by a margin of more than 200 Elo.

Finally, we provide supplementary material in the appendices:

- In Appendix A we introduce the *logistic temporal-difference learning* algorithm. This algorithm is specifically tailored to worlds, such as games or puzzles, in which there is a binary outcome for success or failure. By treating the value function as a *success probability*, we extend the probabilistic framework of logistic regression to temporal-difference learning.

Part I

Literature Review

Chapter 2

Reinforcement Learning

2.1 Learning and Planning

A wide variety of tasks in artificial intelligence and control can be formalised as sequential decision-making processes. We refer to the decision-making entity as the *agent*, and everything outside of the agent as its *environment*. At each time-step t the agent receives observations $s_t \in \mathcal{S}$ from its environment, and executes an action $a_t \in \mathcal{A}$ according to its behaviour policy. The environment then provides a feedback signal in the form of a reward $r_{t+1} \in \mathbb{R}$. This time series of actions, observations and rewards defines the agent's *experience*. The goal of *reinforcement learning* is to improve the agent's future reward given its past experience.

2.2 Markov Decision Processes

If the next observation and reward depend only on the current observation and action,

$$Pr(s_{t+1}, r_{t+1}|s_1, a_1, ..., s_t, a_t) = Pr(s_{t+1}, r_{t+1}|s_t, a_t), \tag{2.1}$$

then the task is a *Markov decision-making process* (MDP). The current observation s_t summarises all previous experience and is described as the *Markov state*. If a task is *fully observable* then the agent receives a Markov state s_t at every time-step; otherwise the task is described as *partially observable*. This thesis is concerned primarily with fully observable tasks; unless otherwise specified all states s are assumed to be Markov. It is also primarily concerned with MDPs in which both the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are finite.

The dynamics of an MDP, from any state s and for any action a, are determined by *transition probabilities* $\mathcal{P}^a_{ss'}$ specifying the next state distribution s', and a *reward function* $\mathcal{R}^a_{ss'}$ specifying the expected reward for a given state transition,

$$\mathcal{P}^a_{ss'} = Pr(s_{t+1} = s'|s_t = s, a_t = a) \tag{2.2}$$

$$\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1}|s_t = s, s_{t+1} = s', a_t = a]. \tag{2.3}$$

Model-based reinforcement learning methods, such as dynamic programming, assume that the dynamics of the MDP are known. *Model-free* reinforcement learning methods, such as Monte-Carlo evaluation or temporal-difference learning, learn directly from experience and do not assume any knowledge of the environment's dynamics.

In episodic (finite horizon) tasks there is a distinguished terminal state. The *return* $R_t = \sum_{k=t}^{T} r_k$ is the total reward accumulated in that episode from time t until the episode terminates at time T[1]. For example, the reward function for a game could be $r_t = 0$ at every move $t < T$, and $r_T = z$ at the end of the game, where z is the final score or outcome; the return would then simply be the score for that game.

The agent's action-selection behaviour can be described by a *policy*, $\pi(s, a)$, that maps a state s to a probability distribution over actions, $\pi(s, a) = Pr(a_t = a|s_t = s)$. In any MDP there is an *optimal policy* $\pi^*(s, a)$ that maximises the expected return from every state in the MDP, $\pi^*(s, a) = \underset{\pi}{\operatorname{argmax}} \mathbb{E}_\pi[R_t|s_t = s]$, where $\mathbb{E}_\pi$ indicates the expectation over episodes of experience generated with policy π.

2.3 Value-Based Reinforcement Learning

Most successful examples of reinforcement learning use a *value function* to summarise the long-term consequences of a particular decision-making policy (Abbeel et al., 2007; Tesauro, 1994; Schaeffer et al., 2001; Singh and Bertsekas, 1997; Ernst et al., 2005).

The value function $V^\pi(s)$ is the expected return from state s when following policy π. The action value function $Q^\pi(s, a)$ is the expected return after selecting action a in state s and then following policy π,

$$V^\pi(s) = \mathbb{E}_\pi[R_t|s_t = s] \tag{2.4}$$

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a]. \tag{2.5}$$

Value-based reinforcement learning algorithms iteratively update an estimate $V(s)$ or $Q(s, a)$ of the value function for the agent's current policy. The updated value function can then be used to *improve* the policy, for example by selecting actions greedily with respect to the new value function. This cyclic process of policy evaluation and policy improvement is at the heart of all value-based reinforcement learning methods (Sutton and Barto, 1998).

The value function is updated by an appropriate *backup* operator. In model-based reinforcement learning algorithms such as dynamic programming, the value function is updated by a *full backup*, which uses the model to perform a full-width lookahead over all possible actions and all possible state transitions. In model-free reinforcement learning algorithms such as Monte-Carlo evaluation

[1] In continuing (infinite horizon) tasks, it is common to discount the future rewards. For clarity of presentation, we restrict our attention to episodic tasks with no discounting.

and temporal-difference learning, the value function is updated by a *sample backup*. At each time-step a single action is sampled from the agent's policy, and a single state transition and reward are sampled from the environment. The value function is then updated from this sampled experience.

2.3.1 Dynamic Programming

The *optimal value function* $V^*(s)$ is the value function when following an optimal policy $\pi^*(s,a)$. An important property of the optimal value function is that it maximises the expected value following from any action. This recursive property is known as the *Bellman equation*,

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + V^*(s') \right] \tag{2.6}$$

The idea of *dynamic programming* is to iteratively update the value function using full backups, so as to satisfy the Bellman equation. The *value iteration* algorithm updates the value function using an *expectimax backup*,

$$V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + V(s') \right] \tag{2.7}$$

If all states are updated by expectimax backups infinitely many times, value iteration converges on the optimal value function.

2.3.2 Monte-Carlo Evaluation

Monte-Carlo evaluation provides a particularly simple, model-free method for policy evaluation. The value function for each state s is estimated by the average return from all episodes that visited state s,

$$V(s) = \frac{1}{n(s)} \sum_{i=1}^{n(s)} R^i(s), \tag{2.8}$$

where $R^i(s)$ is the return following the ith visit to s, and $n(s)$ counts the total number of visits to state s. Monte-Carlo evaluation can equivalently be implemented by an incremental sample backup at each time-step t,

$$n(s_t) \leftarrow n(s_t) + 1 \tag{2.9}$$

$$V(s_t) \leftarrow V(s_t) + \frac{1}{n(s_t)}(R_t - V(s_t)), \tag{2.10}$$

where $n(s)$ and $V(s)$ are initialised to zero.

At each time-step, Monte-Carlo updates the value of the current state towards the return. However, this return depends on the particular action and particular state transition that were sampled in every subsequent state, which may be a very noisy signal. In general, Monte-Carlo provides an unbiased, but high variance estimate of the true value function.

2.3.3 Temporal Difference Learning

Bootstrapping is a general method for reducing the variance of an estimate, by updating a guess from a guess. *Temporal-difference learning* is a model-free method for policy evaluation that bootstraps the value function from subsequent estimates of the value function.

In the TD(0) algorithm, the value function is bootstrapped from the very next time-step. Rather than waiting until the complete return has been observed, the value function of the next state is used to approximate the expected return. The *TD-error* $\delta_t = r_{t+1} + V(s_{t+1}) - V(s_t)$ is measured between the value at state s_t, and the value at the subsequent state s_{t+1}, plus any reward r_{t+1} accumulated along the way. For example, if the agent thinks that Black is winning one move, but that White is winning at the next move, then this inconsistency generates a TD-error. The TD(0) algorithm adjusts the value function so as to correct the TD-error and make it more consistent with the subsequent value,

$$\delta_t = r_{t+1} + V(s_{t+1}) - V(s_t) \tag{2.11}$$

$$\Delta V(s_t) = \alpha \delta_t \tag{2.12}$$

where α is a step-size parameter controlling the learning rate.

2.3.4 TD(λ)

The idea of the TD(λ) algorithm is to bootstrap the value of a state from the subsequent values many steps into the future. The parameter λ determines the temporal span over which bootstrapping occurs. At one extreme, TD(0) bootstraps the value of a state only from its immediate successor. At the other extreme, TD(1) updates the value of a state from the final return; it is equivalent to Monte-Carlo evaluation.

To implement TD(λ) incrementally, an eligibility trace $z(s)$ is maintained for each state. The eligibility trace represents the total credit that should be assigned to a state for any subsequent errors in evaluation. It combines a *recency heuristic* with a *frequency heuristic*: states which are visited most frequently and most recently are given the greatest eligibility (Sutton, 1984). The eligibility trace is incremented each time the state is visited, and decayed by a constant parameter λ at every time step (Equation 2.13). Every time a difference is seen between the predicted value and the subsequent value, a *TD-error* δ_t is generated. The value function for all states is updated in proportion to both the TD-error and the eligibility of the state,

$$z_t(s) = \begin{cases} \lambda z_{t-1}(s) & \text{if } s \neq s_t \\ \lambda z_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{2.13}$$

$$\delta_t = r_{t+1} + V_t(s_{t+1}) - V_t(s_t) \tag{2.14}$$

$$\Delta V_t(s) = \alpha \delta_t z_t(s). \tag{2.15}$$

This form of eligibility update is known as an *accumulating* eligibility trace. An alternative update, known as a *replacing* eligibility trace, can be more efficient in some environments (Singh and Sutton, 2004),

$$z_t(s) = \begin{cases} \lambda z_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{2.16}$$

$$\delta_t = r_{t+1} + V_t(s_{t+1}) - V_t(s_t) \tag{2.17}$$

$$\Delta V_t(s) = \alpha \delta_t z_t(s). \tag{2.18}$$

If all states are visited infinitely many times, and with appropriate choice of step-size, temporal-difference learning converges on the true value function V^π for all values of λ (Dayan, 1994).

2.3.5 Sarsa(λ)

The Sarsa algorithm (Rummery and Niranjan, 1994) combines temporal difference learning with policy improvement. An action-value function is updated by the TD(λ) algorithm, and the latest action values are used to select actions. An ϵ-*greedy* policy is used to combine exploration (selecting a random action with probability ϵ) with exploitation (selecting $\operatorname{argmax}_a Q(s,a)$ with probability $1-\epsilon$). The action-value function is updated from each tuple (s, a, r, s', a') of experience, using the TD(λ) update rule for state-action values. If all states are visited infinitely many times, and with appropriate choice of step-size, the Sarsa algorithm converges on the optimal policy (Singh et al., 2000).

2.3.6 Value Function Approximation

In large environments, it is not possible or practical to learn a value for each individual state. In this case, it is necessary to represent the state more compactly, by using some set of *features* $\phi(s)$ of the state s. The value function can then be approximated by a function of the features and parameters θ. For example, a set of binary features $\phi(s) \in \{0,1\}^n$ can be used to abstract the state space, where each binary feature $\phi_i(s)$ identifies a particular property of the state. A common and successful methodology (Sutton, 1996) is to use a linear combination of features and parameters to approximate the value function, $V(s) = \phi(s).\theta$.

We refer to the case when no value function approximation is used, in other words when each state has a distinct value, as *table-lookup*. Linear function approximation subsumes table-lookup as a special case. To see this, consider binary features $\phi(s) = e(s)$, where $e(s)$ is a unit vector of size $|S|$ with a one in the sth component and zeros elsewhere. In this special case the sth parameter θ_s represents the value function $V(s)$ for the individual state s.

When the value function is approximated by a function of features, errors could be attributed to any or all of those features. *Gradient descent* provides a principled approach to this problem of credit assignment: the parameters are updated in the direction that minimises the mean-squared

evaluation error. Monte-Carlo evaluation with linear function approximation provides a particularly simple case. The parameters are updated by stochastic gradient descent, with a step-size of α,

$$\Delta\theta = -\frac{\alpha}{2}\nabla_\theta(R_t - V(s_t))^2 \tag{2.19}$$

$$= \alpha(R_t - V(s_t))\nabla_\theta V(s_t) \tag{2.20}$$

$$= \alpha(R_t - V(s_t))\phi(s_t) \tag{2.21}$$

2.3.7 Linear Temporal-Difference Learning

Temporal-difference learning can be applied to any differentiable approximation to the value function. The key idea is to minimise the mean-squared TD-error (MSTDE) by stochastic gradient descent. However, when bootstrapping the target value is treated as a constant that does not depend on the parameters. This ensures that the current value is moved towards the future value, rather than the other way around[2]. Temporal-difference learning with linear function approximation is a particularly simple case. The parameters are updated in proportion to the TD-error and the feature value,

$$\Delta\theta = -\frac{\alpha}{2}\nabla_\theta(V(s_{t+1}) - V(s_t))^2 \tag{2.22}$$

$$= \alpha(V(s_{t+1}) - V(s_t))\phi(s_t) \tag{2.23}$$

The linear Sarsa algorithm combines linear temporal-difference learning with the Sarsa algorithm. The parameters of the linear combination are updated by stochastic gradient descent so as to minimise the mean-squared TD-error, while using an epsilon-greedy policy to select actions. The complete linear Sarsa algorithm is shown in Algorithm 1.

2.4 Policy Gradient Reinforcement Learning

Instead of updating a value function, the idea of policy gradient reinforcement learning is to directly update the parameters of the agent's policy by gradient ascent, so as to maximise the agent's average reward per time-step. Policy gradient methods are typically higher variance and therefore less efficient than value-based approaches, but they have three significant advantages. First, they are able to learn *mixed strategies* that are a stochastic balance of different actions. Second, they have better convergence guarantees than value-based methods. Finally, they are able to learn a parameterised policy even in problems with continuous action spaces.

[2]The *residual gradient algorithm* (Baird, 1995) follows the gradient with respect to both current and future values. Unfortunately, there are simple counterexamples, even in the table-lookup case, where this algorithm evaluates states incorrectly (Dayan, 1994).

Algorithm 1 Sarsa(λ)

```
1:  procedure SARSA(λ)
2:      θ ← 0                                              ▷ Clear weights
3:      loop
4:          s ← s_0                                        ▷ Start new episode in initial state
5:          z ← 0                                          ▷ Clear eligibility trace
6:          a ← ε-greedy action from state s
7:          while s is not terminal do
8:              Execute a, observe reward r and next state s'
9:              a' ← ε-greedy action from state s'
10:             δ ← r + Q(s', a') − Q(s, a)                ▷ Calculate TD-error
11:             θ ← θ + αδz                                ▷ Update weights
12:             z ← λz + φ(s, a)                           ▷ Update eligibility trace
13:             s ← s', a ← a'
14:         end while
15:     end loop
16: end procedure
```

Williams' REINFORCE algorithm updates the parameters of the agent's policy by stochastic gradient ascent. Given a differentiable policy $\pi_p(s, a)$ that is parameterised by a vector of adjustable weights p, the REINFORCE algorithm updates those weights at every time-step t,

$$\Delta p = \beta(R_t - b(s_t)) \log \nabla_p \pi_p(s_t, a_t) \tag{2.24}$$

where β is a step-size parameter and b is a *reinforcement baseline* that does not depend on the current action a_t.

The *policy gradient theorem* (Sutton et al., 2000) extends the REINFORCE gradient to use the action value function in place of the actual return,

$$\Delta p = \beta(Q^\pi(s_t, a_t) - b(s_t)) \log \nabla_p \pi_p(s_t, a_t) \tag{2.25}$$

This theorem is used by *actor-critic* algorithms to combine the advantages of policy gradient methods with the efficiency of value-based reinforcement learning. They consist of two components: an *actor* that updates the agent's policy, and a *critic* that updates the action value function. When value function approximation is used, care must be taken to ensure that the critic's parameters θ are *compatible* with the actor's parameters p. The compatibility requirement is that $\nabla_\theta Q_\theta(s, a) = \nabla_p \log \pi_p(s, a)$.

The reinforcement baseline does not affect the expectation of the gradient, but can reduce its variance. The variance is minimised by using the value function as a baseline, $b(s) = V^\pi(s)$. The policy parameters are then updated in proportion to the *advantage function* $Q^\pi(s, a) - V^\pi(s)$.

2.5 Exploration and Exploitation

The ϵ-greedy policy used in the Sarsa algorithm provides one simple approach to balancing exploration with exploitation. However, more sophisticated strategies are also possible. We mention two of the most common approaches here.

First, exploration can be skewed towards more highly valued states, for example by using a *softmax policy*,

$$\pi(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_b e^{Q(s,b)/\tau}} \tag{2.26}$$

where τ is a parameter controlling the temperature (level of stochasticity) in the policy.

A second approach is to apply the principle of *optimism in the face of uncertainty*, for example by adding a bonus to the value function that is largest in the most uncertain states. The UCB1 algorithm (Auer et al., 2002) follows this principle, by maximising an upper confidence bound on the value function,

$$Q^{\oplus}(s,a) = Q(s,a) + \sqrt{\frac{2\log n(s)}{n(s,a)}} \tag{2.27}$$

$$\pi(s,a) = \operatorname*{argmax}_b Q^{\oplus}(s,b) \tag{2.28}$$

where $n(s)$ counts the number of visits to state s, and $n(s,a)$ counts the number of times that action a has been selected from state s.

Chapter 3

Search and Planning

3.1 Introduction

Planning is the process of computation by which an action is selected. Typically, the agent has some amount of thinking time during which it can plan its next action, without any further interaction with the environment. The goal of planning is to use this computation to improve its action selection policy $\pi(s, a)$. This policy can then be applied from any state s in the problem. *Search* refers to the process of computation that specifically improves the action selected from a root state s_0, for example the agent's current state. By focusing on this single state, search methods can be considerably more efficient than general planning methods.

3.2 Planning

Most planning methods use a model of the environment. This model can either be solved directly, by applying model-based reinforcement learning methods, or indirectly, by sampling the model and applying model-free reinforcement learning methods.

3.2.1 Model-Based Planning

As we saw in the previous chapter, fully observable environments can be represented by an MDP M with state transition probabilities $\mathcal{P}_{ss'}^a$ and a reward function $\mathcal{R}_{ss'}^a$. In general, the agent does not know the true dynamics of the environment, but it may know or learn an approximate *model* of its environment, represented by state transition probabilities $\hat{\mathcal{P}}_{ss'}^a$ and a reward function $\hat{\mathcal{R}}_{ss'}^a$.

The idea of model-based planning is to apply model-based reinforcement learning methods, such as dynamic programming, to the MDP $\hat{M}$ described by the model $\hat{\mathcal{P}}_{ss'}^a, \hat{\mathcal{R}}_{ss'}^a$. The success of this approach depends largely on the accuracy of the model. If the model is accurate, then the policy learned for $\hat{M}$ will also perform well in the agent's actual environment M. If the model is inaccurate, the learned policy can perform arbitrarily poorly in M.

3.2.2 Sample-Based Planning

In reinforcement learning, the agent samples experience from the real world: it executes an action at each time-step, observes its consequences, and updates its policy. In sample-based planning the agent samples experience from a model of the world: it *simulates* an action at each computational step, observes its consequences, and updates its policy. This symmetry between learning and planning has an important consequence: algorithms for reinforcement learning can also become algorithms for planning, simply by substituting simulated experience in place of real experience.

Sample-based planning requires a *generative model* that can sample state transitions and rewards from $\hat{\mathcal{P}}^a_{ss'}$ and $\hat{\mathcal{R}}^a_{ss'}$ respectively. However, it is not necessary to know these probability distributions; the next state and reward could, for example, be generated by a black box simulator. In some environments, such as two-player games, it can be much easier to provide a generative model (e.g. a program simulating the opponent's behaviour) than to describe the complete probability distribution.

Given a generative model, the agent can sample experience and receive a hypothetical reward. The agent's task is to learn how to maximise its total expected reward, from this hypothetical experience. Thus, each model specifies a new reinforcement learning problem, which itself can be solved by model-free reinforcement learning algorithms.

3.2.3 Dyna

The Dyna architecture (Sutton, 1990) combines reinforcement learning with sample-based planning. The agent learns a model of the world from real experience, and updates its action-value function from both real and sampled experience. Before each real action is selected, the agent executes some number of iterations of sample-based planning.

The *Dyna-Q* algorithm utilises a memory-based model of the world. It remembers all state transitions and rewards from all visited states and selected actions. During each iteration of planning, a previously visited start state and action is selected, and a state transition and reward are sampled from the memorised experience. Temporal-difference learning is used to update the action-value function after each sampled transition (planning), and also after each real transition (learning).

3.3 Search

Most search algorithms construct a *search tree* of states that can be reached from s_0, and evaluate candidate actions by their future consequences within the search tree.

3.3.1 Full-Width Search

A *full-width search* considers all possible actions and all successor states from each internal node of the search tree. The states of the search tree can be developed *selectively*, by expanding leaf nodes according to some criterion, or *exhaustively*, by expanding all nodes up to some fixed depth.

Expansion may proceed in a depth-first, breadth-first, or best-first order, where the latter utilises a *heuristic function* to guide the search towards the most promising states (Russell and Norvig, 1995).

Full-width search can be applied to MDPs, so as to find the sequence of actions that leads to the maximum expected return from the current state. Special case searches can also be applied in deterministic environments, or in two-player games. In each case, heuristic search algorithms operate in a very similar manner. Leaf nodes are evaluated by the heuristic function, and interior nodes are evaluated by an appropriate backup operation that updates each parent value from all of its children: an expectimax backup in MDPs, a max backup in deterministic environments, or a minimax backup in two-player games.

This very general framework can be used to categorise a number of well-known search algorithms: for example A* (Hart et al., 1968) is a best-first search with max backups; *expectimax search* (Davies et al., 1998) is a depth-first search with expectimax backups; and alpha-beta (Rivest, 1988) is a depth-first search with minimax backups.

A value function (see Chapter 2) can be used as a heuristic function. In this approach, leaf nodes are evaluated by estimating the expected return or outcome from that node (Davies et al., 1998).

3.3.2 Sample-Based Search

In *sample-based search*, instead of considering all possible actions and all possible successor states, actions are sampled from the agent's policy, and successor states and rewards are sampled from a generative model. These samples are typically used to construct a tree, and the value of each interior node is updated by an appropriate backup operation. In environments that are stochastic or have large branching factors, sample-based search can be much more effective than full-width search.

Sparse lookahead (Kearns et al., 2002) is a depth-first approach to sample-based search. A state s is expanded by executing each action a, and sampling C successor states from the model, to generate a total of $|\mathcal{A}|C$ children. Each child is expanded recursively in depth-first order, and then evaluated by a *sample max backup*,

$$V(s) \leftarrow \max_{a \in \mathcal{A}} \frac{1}{C} \sum_{i=1}^{C} V(child(s, a, i)) \tag{3.1}$$

where $child(s, a, i)$ denotes the ith child of state s for action a. Leaf nodes at maximum depth D are evaluated by a fixed value function. Finally, the action with maximum evaluation at the root node s_0 is selected. Given sufficient depth D and breadth C, this approach will generate a near-optimal policy for any MDP.

Sparse lookahead can be extended to use a more informed exploration policy. Rather than uniformly sampling each action C times, the UCB1 algorithm (see Chapter 2) is used to select the next action to sample (Chang et al., 2005). This ensures that the best actions are tried most often, but the actions with greatest uncertainty are also explored.

3.3.3 Simulation-Based Search

The basic idea of *simulation-based search* is to sequentially sample episodes of experience that start from the root state s_0. At each step t of simulation, an action a_t is selected according to a *simulation policy*, and a new state s_{t+1} and reward r_{t+1} is generated by the model. After every simulation, the values of states or actions are updated from the simulated experience.

Like sparse lookahead, simulation-based search algorithms can be used to selectively construct a search tree, but in a best-first rather than depth-first manner. We refer to this approach as *simulation-based tree search*. Each simulation starts from the root of the search tree, and the best action is selected at each step according to the current value function[1]. After each simulation, every visited state is added to the search tree, and the values of these states are backed up through the search tree, for example by a sample max backup (Péret and Garcia, 2004).

As the simulations progress, the value function becomes more accurate and the simulation policy becomes better informed. Unlike the depth-first approach used in sparse lookahead, this allows the search to continually refocus its attention on the most promising regions of the state space.

3.3.4 Monte-Carlo Simulation

Monte-Carlo simulation is a very simple simulation-based search algorithm for evaluating candidate actions from a root position s_0. The search proceeds by simulating complete episodes from s_0 until termination, using a fixed simulation policy. The action-values $Q(s_0, a)$ are estimated by the mean outcome of all simulations with candidate action a.

In its most basic form, Monte-Carlo simulation is only used to evaluate actions, but not to improve the simulation policy. However, the basic algorithm can be extended by progressively favouring the most successful actions, or by progressively pruning away the least successful actions (Billings et al., 1999; Bouzy and Helmstetter, 2003)

The principle advantage of Monte-Carlo simulation is that it can dynamically evaluate states and actions, often more accurately than can be achieved by a statically stored approximation to the value function. However, in some environments it can be beneficial to terminate simulation before the end of the episode, and bootstrap from the estimated value at termination. This approach is known as *truncated Monte-Carlo simulation*, and is most effective in environments for which highly accurate value functions can be constructed (Tesauro and Galperin, 1996; Sheppard, 2002).

3.3.5 Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) is a simulation-based tree search algorithm that uses Monte-Carlo simulation to evaluate the nodes of a search tree (Coulom, 2006). There is one node in the tree for each state-action pair (s, a), containing a value $Q(s, a)$. This value is estimated by the mean outcome of all simulations in which action a was selected from state s. Each simulation starts from

[1]This procedure is *recursively best-first*: it selects the best child at every step, rather than the best overall leaf node.

the root state s_0, and is divided into two stages: a *tree policy* is used while within the search tree; and a *default policy* is used once simulations leave the scope of the search tree. The simplest version of MCTS uses a greedy tree policy during the first stage, which selects the action with the highest value, $\operatorname*{argmax}_a Q(s,a)$; and a uniform random default policy during the second stage, which rolls out simulations until completion. After each simulation, one new node is added to the search tree, containing the first state visited in the second stage. Figure 3.1 illustrates several steps of the MCTS algorithm.

MCTS can be enhanced by improving the tree policy in the first stage of simulation, or by improving the default policy in the second stage of simulation.

The UCT algorithm (Kocsis and Szepesvari, 2006) improves the greedy action selection in Monte-Carlo Tree Search. Each state of the search tree is treated as a multi-armed bandit, and actions are chosen by using the UCB1 algorithm (see Chapter 2). The action-value is augmented by an exploration bonus that is highest for rarely visited state-action pairs,

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}, \tag{3.2}$$

where c is a scalar constant. Once all actions from state s are represented in the search tree, the tree policy selects the action maximising the augmented action-value, $\operatorname*{argmax}_a Q^{\oplus}(s,a)$. Otherwise, the default policy is used to select actions.

MCTS can also be extended to incorporate domain knowledge in the default policy, rather than behaving completely randomly (Gelly et al., 2006). Much of the research in Monte-Carlo tree search has been developed in the context of Computer Go, and is discussed in more detail in the next Chapter.

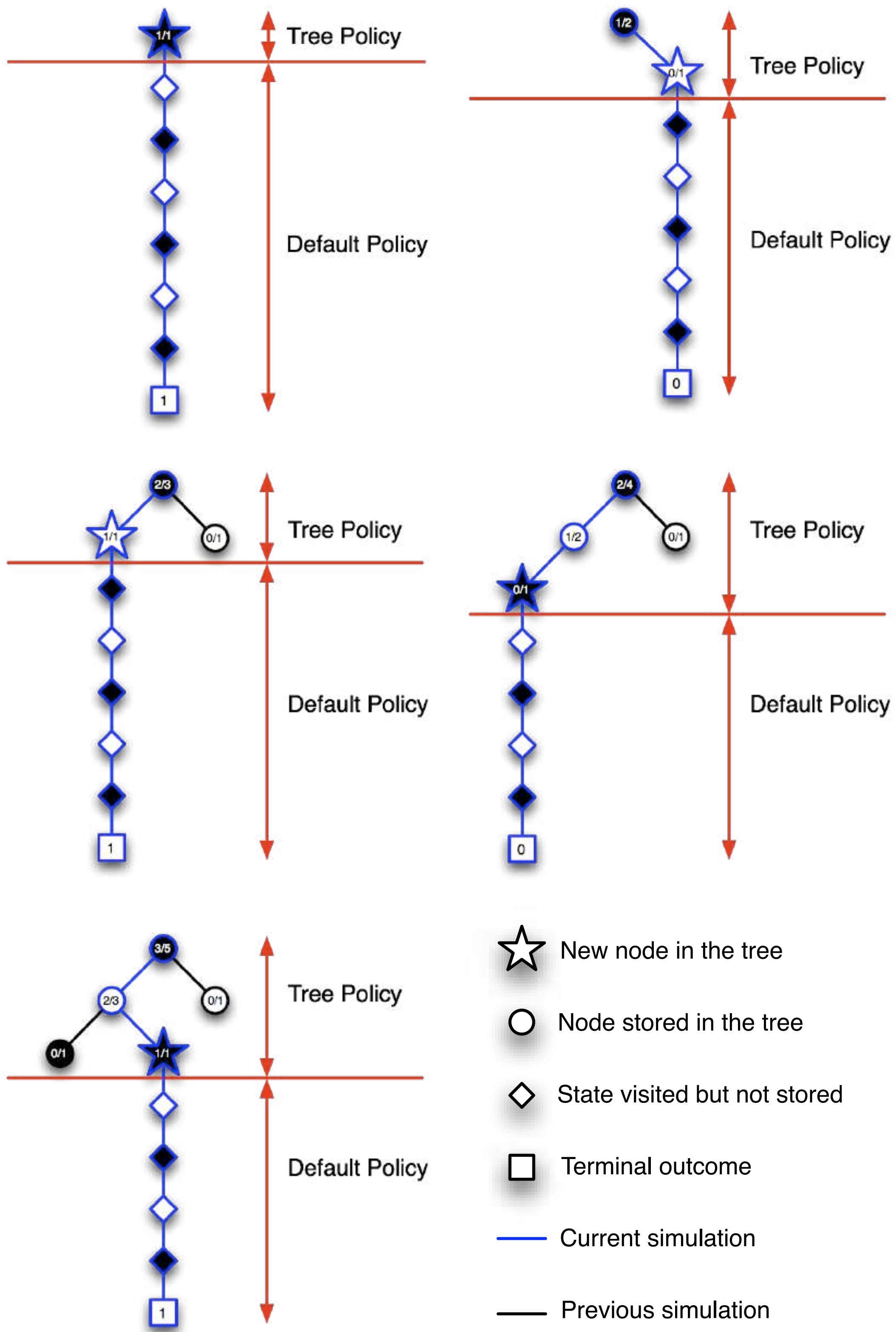

Figure 3.1: Five simulations of Monte-Carlo tree search.

Chapter 4

Computer Go

4.1 The Challenge of Go

For many years, Computer Chess was considered to be the *drosophila* of AI[1] and a grand challenge task (McCarthy, 1997). It provided a simple sandbox for new ideas, a straightforward performance comparison between algorithms, and measurable progress against human capabilities. With the dominance of alpha-beta search programs over human players now conclusive (McClain, 2006), many researchers have sought out a new challenge. Computer Go has emerged as the new drosophila of AI (McCarthy, 1997), a task *par excellence* (Harmon, 2003), and a grand challenge task for our generation (Mechner, 1998).

In the last few years, a new paradigm for AI has been developed in Computer Go. This approach, based on Monte-Carlo simulation, has provided dramatic progress and led to the first master-level programs (Gelly and Silver, 2007; Coulom, 2007). Unlike alpha-beta search, these algorithms are still in their infancy, and the arena is still wide open to new ideas. In addition, this new approach to search requires little or no human knowledge in order to produce good results. Although this paradigm has been pioneered in Computer Go, it is not specific to Go, and the core concepts are widely applicable. Ultimately, the study of Computer Go may illuminate a path towards high performance AI in a wide variety of challenging domains.

4.2 The Rules of Go

The game of Go is usually played on a 19×19 grid, with 13×13 and 9×9 as popular alternatives. Black and White play alternately, placing a single stone on an intersection of the grid. Stones cannot be moved once played, but may be captured. Sets of adjacent, connected stones of one colour are known as *blocks*. The empty intersections adjacent to a block are called its *liberties*. If a block is reduced to zero liberties by the opponent, it is captured and removed from the board (Figure 4.1A). Stones with just one remaining liberty are said to be in *atari*. Playing a stone with zero liberties is illegal (Figure 4.1B), unless it also reduces an opponent block to zero liberties. In this case the

[1] *Drosophila* is the fruit fly, the most extensively studied organism in genetics research.

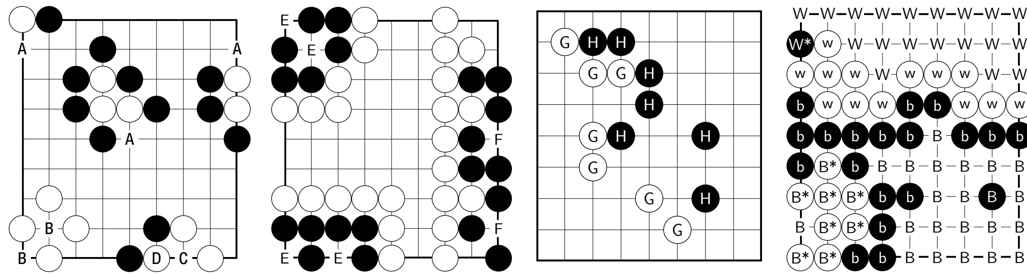

Figure 4.1: a) The White stones are in *atari* and can be captured by playing at the points marked A. It is illegal for Black to play at B, as the stone would have no liberties. Black may, however, play at C to capture the stone at D. White cannot recapture immediately by playing at D; as this would repeat the position - it is a *ko*. b) The points marked E are *eyes* for Black. The black groups on the left can never be captured by White, they are *alive*. The points marked F are *false eyes*: the black stones on the right will eventually be captured by White and are *dead*. c) *Groups* of loosely connected white stones (G) and black stones (H). d) A final position. Dead stones (B*,W*) are removed from the board. All surrounded intersections (B,W) and all remaining stones (b,w) are counted for each player. If *komi* is 6.5 then Black wins by 8.5 points in this example.

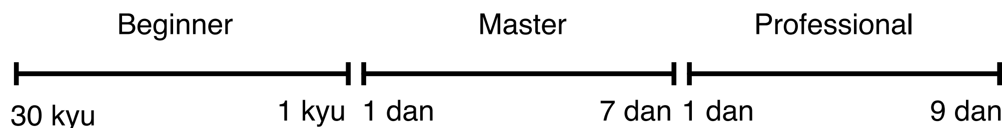

Figure 4.2: Performance ranks in Go, in increasing order of strength from left to right.

opponent block is captured, and the player's stone remains on the board (Figure 4.1C). Finally, repeating a previous board position is illegal. A situation in which a repeat could otherwise occur is known as *ko* (Figure 4.1D).

A connected set of empty intersections that is wholly enclosed by stones of one colour is known as an *eye*. One natural consequence of the rules is that a block with two eyes can never be captured by the opponent (Figure 4.1E). Blocks which cannot be captured are described as *alive*; blocks which will certainly be captured are described as *dead* (Figure 4.1F). A loosely connected set of stones is described as a *group* (Figure 4.1G). Determining the life and death status of a group is a fundamental aspect of Go strategy.

The game ends when both players pass. Dead blocks are removed from the board (Figure 4.1B*,W*). All alive stones, and all intersections that are enclosed by a player, are counted as a point of *territory* for that player (Figure 4.1B,W)[2]. Black always plays first in Go; White receives compensation, known as *komi*, for playing second. The winner is the player with the greatest territory, after adding *komi* for White.

4.3 Go Ratings

Human Go players are rated on a three-class scale, divided into *kyu* (beginner), *dan* (master), and *professional dan* ranks (see Figure 4.2). *Kyu* ranks are in descending order of strength, whereas *dan* and *professional dan* ranks are in ascending order. At amateur level, the difference in rank corresponds to the number of handicap stones required by the weaker player to ensure an even game[3].

The statistical Elo scale is also used to evaluate human Go players. This scale assumes that each player's performance in a game is a random variable, and that the player with higher performance will win the game. The original Elo scale assumed that the player's performance is normally distributed; modern incarnations of the Elo scale assume a logistic distribution. In either case, each player's Elo rating is their mean performance, which is estimated and updated from their results. Unfortunately, several different Elo scales are used to evaluate human Go ratings, based on different assumptions about the performance distribution.

The majority of Computer Go programs compete on the Computer Go Server (CGOS). This server runs an ongoing rapid-play tournament of 5 minute games, and evaluates the Elo rating of each program. The Elo scale on CGOS, and the Elo ratings reported in this thesis, assume a logistic distribution with winning probability $Pr(A \text{ beats } B) = \frac{1}{1+10^{\frac{\mu_B - \mu_A}{400}}}$, where μ_A and μ_B are the Elo ratings for player A and player B respectively. On this scale, a difference of 200 Elo corresponds to a 75% winning rate for the stronger player, and a difference of 500 Elo corresponds to a 95% winning rate.

4.4 Position Evaluation in Computer Go

A rational Go player selects moves so as to maximise an *evaluation function* $V(s)$. We denote this greedy move selection strategy by a deterministic function $\pi(s)$ that takes a position $s \in \mathcal{S}$ and produces the move $a \in \mathcal{A}$ with the highest evaluation,

$$\pi(s) = \operatorname*{argmax}_a V(s \circ a) \tag{4.1}$$

where $s \circ a$ denotes the position reached after playing move a from position s.

The evaluation function is a summary of positional Go knowledge, and is used to estimate the goodness of each move. A *heuristic function* is a measure of goodness, such as the material count in chess, that is presumed but not required to have some positive correlation with the outcome of the game. A *value function* (see Chapter 2) specifically defines the goodness of a position s to be the expected outcome z of the game from that position, $V(s) = \mathbb{E}[z|s]$. A *static evaluation function* is

[2] The Japanese scoring system is somewhat different, but usually has the same outcome.

[3] The difference between 1 *kyu* and 1 *dan* is normally considered to be 1 stone.

stored in memory, whereas a *dynamic evaluation function* is computed by a process of search from the current position s.

4.5 Static Evaluation in Computer Go

Constructing an evaluation function for Go is a challenging task. First, as we have already seen, the state space is enormous. Second, the evaluation function can be highly volatile: changing a single stone can transform a position from lost to won or vice versa. Third, interactions between stones may extend across the whole board, making it difficult to decompose the global evaluation into local features.

A static evaluation function cannot usually store a separate value for each distinct position s. Instead, it is represented by *features* $\phi(s)$ of the position s, and some number of adjustable *parameters* θ. For example, a position can be evaluated by a neural network that uses features of the position as its inputs (Schraudolph et al., 1994; Enzenberger, 1996; Dahl, 1999; Enzenberger, 2003).

4.5.1 Symmetry

The Go board has a high degree of symmetry. It has eight-fold rotational and reflectional symmetry, and it has colour symmetry: if all stone colours are inverted and the colour to play is swapped, then the position is exactly equivalent. This suggests that the evaluation function should be invariant to rotational, reflectional and colour inversion symmetries. When considering the status of a particular intersection, the Go board also exhibits translational symmetry: a local configuration of stones in one part of the board has similar properties to the same configuration of stones in another part of the board, subject to edge effects.

Schraudolph et al. (1994) exploit these symmetries in a convolutional neural network. The network predicts the final territory status of a particular target intersection. It receives one input from each intersection (-1, 0 or $+1$ for White, Empty and Black respectively) in a local region around the target, contains a fixed number of hidden nodes, and outputs the predicted territory for the target intersection. The global position is evaluated by summing the territory predictions for all intersections on the board. Weights are shared between rotationally and reflectionally symmetric patterns of input features[4], and between all target intersections. In addition, the input features, squashing function and bias weights are all antisymmetric, and on each alternate move the sign of the bias weight is flipped, so that network evaluation is invariant to colour inversion.

A further symmetry of the Go board is that stones within the same block will live or die together as a unit, sometimes described as the *common fate* property (Graepel et al., 2001). One way to make use of this invariance (Enzenberger, 1996; Graepel et al., 2001) is to treat each complete block or empty intersection as a unit, and to represent the board by a *common fate graph* containing a node for each unit and an edge between each pair of adjacent units.

[4] Surprisingly this impeded learning in practice (Schraudolph et al., 2000).

4.5.2 Handcrafted Heuristics

In many other classic games, handcrafted heuristic functions have proven highly effective. Basic heuristics such as *material count* and *mobility*, which provide reasonable estimates of goodness in Checkers, Chess and Othello (Schaeffer, 2000), are next to worthless in Go. Stronger heuristics have proven surprisingly hard to design, despite several decades of endeavour (Müller, 2002).

Until recently, most Go programs incorporated very large quantities of expert knowledge, in a *pattern database* containing many thousands of manually inputted patterns, each describing a rule of thumb that is known by expert Go players. Traditional Go programs used these databases to recommend expert moves in commonly recognised situations, typically in conjunction with local or global alpha-beta search algorithms. In addition, they can be used to encode knowledge about connections, eyes, opening sequences, or promising search extensions. The pattern database accounts for a large part of the development effort in a traditional Go program, sometimes requiring many man-years of effort from expert Go players.

However, pattern databases are hindered by the knowledge acquisition bottleneck: expert Go knowledge is hard to interpret, represent, and maintain. The more patterns in the database, the harder it becomes to predict the effect of a new pattern on the overall playing strength of the program.

4.5.3 Temporal Difference Learning

Reinforcement learning can be used to estimate a value function that predicts the eventual outcome of the game. The learning program can be rewarded by the score at the end of the game, or by a reward of 1 if Black wins and 0 if white wins. Surprisingly, the less informative binary signal has proven more successful (Coulom, 2006), as it encourages the agent to favour risky moves when behind, and calm moves when ahead. Expert Go players will frequently play to minimise the uncertainty in a position once they judge that they are ahead in score; this behaviour cannot be replicated by simply maximising the expected score. Despite this shortcoming, the final score is widely used as a reward signal (Schraudolph et al., 1994; Enzenberger, 1996; Dahl, 1999; Enzenberger, 2003).

Schraudolph et al. (1994) exploit the symmetries of the Go board (see Section 4.5.1) to predict the final territory at an intersection. They train their multilayer perceptron using $TD(0)$, using a reward signal corresponding to the final territory value of the intersection. The network outperformed a traditional Go program set to a low playing level, in 9×9 Go, after just 3,000 self-play training games.

Dahl's Honte (1999) and Enzenberger's NeuroGo III (2003) use a similar approach to predicting the final territory. However, both programs learn intermediate features that are used to input additional knowledge into the territory evaluation network. Honte has one intermediate network to predict local moves and a second network to evaluate the life and death status of groups. NeuroGo III uses intermediate networks to evaluate connectivity and eyes. Both programs achieved single-digit *kyu* ranks; NeuroGo won the silver medal at the 2003 9×9 Computer Go Olympiad.

Although a complete game of Go typically contains hundreds of moves, only a small number of moves are played within a given local region. Enzenberger (2003) suggests for this reason that $TD(0)$ is a natural choice of algorithm. Indeed, $TD(0)$ has been used almost exclusively in reinforcement learning approaches to position evaluation in Go (Schraudolph et al., 1994; Enzenberger, 1996; Dahl, 1999; Enzenberger, 2003; Runarsson and Lucas, 2005; Mayer, 2007), perhaps because of its simplicity and its proven efficacy in games such as backgammon (Tesauro, 1994).

4.5.4 Comparison Training

If we assume that expert Go players are rational, then it is reasonable to infer the expert's evaluation function V_{expert} by observing their move selection decisions. For each expert move a, rational move selection tells us that $V_{expert}(s \circ a) \geq V_{expert}(s \circ b)$ for any legal move b. This can be used to generate an error metric for training an evaluation function $V(s)$, in an approach known as *comparison training* (Tesauro, 1988). The expert move a is compared to another move b, randomly selected; if the non-expert move evaluates higher than the expert move then an error is generated.

Van der Werf et al. (2002) use comparison training to learn the weights of a multilayer perceptron, using local board features as inputs. Following Enderton (1991), they compute an error function E of the form,

$$E(s, a, b) = \begin{cases} [V(s \circ a) + \epsilon - V(s \circ b)]^2 & \text{if } V(s \circ a) + \epsilon > V(s \circ a) \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

where ϵ is a positive control parameter used to avoid trivial solutions. The trained network was able to predict expert moves with 37% accuracy on an independent test set; the authors estimate its strength to be at strong *kyu* level for the task of local move prediction. The learned evaluation function was used in the Go program *Magog*, which won the bronze medal in the 2004 9×9 Computer Go Olympiad.

4.5.5 Evolutionary Methods

A common approach is to apply evolutionary methods to learn a heuristic evaluation function, for example by applying genetic algorithms to the weights of a multilayer perceptron. The fitness of a heuristic is typically measured by running a tournament and counting the total number of wins. These approaches have two major sources of inefficiency. First, they only learn from the result of the game, and do not exploit the sequence of positions and moves used to achieve the result. Second, many games must be run in order to produce fitness values with reasonable discrimination. Runarsson and Lucas compare temporal difference learning with coevolutionary learning, using a basic state representation. They find that TD(0) both learns faster and achieves greater performance in most cases (Runarsson and Lucas, 2005). Evolutionary methods have not yet, to our knowledge, produced a competitive Go program.

4.6 Dynamic Evaluation in Computer Go

An alternative method of position evaluation is to construct a search tree from the root position, and dynamically update the evaluation of the nodes in the search tree.

4.6.1 Alpha-Beta Search

Despite the challenging search space, and the difficulty of constructing a static evaluation function, alpha-beta search has been used extensively in Computer Go. One of the strongest traditional programs, *The Many Faces of Go*, uses a global alpha-beta search to select moves in 19×19 Go. Each position is evaluated by a combination of local alpha-beta searches to determine the status of individual blocks and groups. Other traditional Go programs also use specialised search routines for determining local subgoals such as capture, connection, and eye formation (Müller, 2002).

However, even determining the status of individual blocks can be a challenging problem. In addition, the local searches are not usually independent, and the search trees can overlap significantly. Finally, the global evaluation often depends on more subtle factors than can be represented by simple local subgoals (Müller, 2001).

4.6.2 Monte Carlo Simulation

In contrast to traditional search methods, Monte-Carlo simulation does not require a static evaluation function. This makes it an appealing choice for Go, where as we have seen, position evaluation is particularly challenging.

Bouzy and Helmstetter developed the first competitive Go programs based on Monte-Carlo simulation (Bouzy and Helmstetter, 2003). Their basic framework simulates many games of self-play from the current position s, for each candidate action a, using a uniform random simulation policy; the value of a is estimated by the average outcome of these simulations. The only domain knowledge is to prohibit moves within eyes; this ensures that games terminate within a reasonable timeframe. Bouzy and Helmstetter also investigated a number of extensions to Monte-Carlo simulation, several of which are precursors to the more sophisticated algorithms now used in the strongest Go programs:

1. *Progressive pruning* is a technique in which statistically inferior moves are removed from consideration (Bouzy, 2005b).

2. The *all-moves-as-first* heuristic (first introduced in the Go program *Gobble* (Bruegmann, 1993)) assumes that the value of a move is not significantly affected by changes elsewhere on the board. The value of playing move a immediately is computed from the average outcome of all simulations in which move a is played *at any time*.

3. The *temperature* heuristic uses a softmax simulation policy to bias the random moves towards the strongest evaluations. The softmax policy selects moves with a probability $\pi(s, a) =$

$\frac{e^{V(s \circ a)/\tau}}{\sum e^{V(s \circ b)/\tau}}$, where τ is a constant temperature parameter controlling the overall level of randomness[5].

4. The *minimax enhancement* constructs a full width search tree, and separately evaluates each node of the search tree by Monte-Carlo simulation. Selective search enhancements were also tried (Bouzy, 2004).

Bouzy also tracks statistics about the final territory status of each intersection after each simulation (Bouzy, 2006). This information is used to influence the simulations towards disputed regions of the board, by avoiding playing on intersections which are consistently Black territory or consistently White territory. Bouzy also incorporates pattern knowledge into the simulation player (Bouzy, 2005a). Using these enhancements his program *Indigo* won the bronze medal at the 2004 and 2006 19×19 Computer Go Olympiads.

It is surprising that a Monte-Carlo technique, originally developed for stochastic games such as Backgammon (Tesauro and Galperin, 1996), Poker (Billings et al., 1999) and Scrabble (Sheppard, 2002) should succeed in Go. Why should an evaluation that is based on random play provide any useful information in the precise, deterministic game of Go? The answer, perhaps, is that Monte-Carlo methods successfully manage the uncertainty in the evaluation. A random simulation policy generates a broad distribution of simulated games, representing many possible futures and the uncertainty in what may happen next. As the search proceeds and more information is accrued, the simulation policy becomes more refined, and the distribution of simulated games narrows. In contrast, deterministic play represents perfect confidence in the future: there is only one possible continuation. If this confidence is misplaced, then predictions based on deterministic play will be unreliable and misleading.

4.6.3 Monte-Carlo Tree Search

Within just three years of their introduction, Monte-Carlo tree search algorithms have revolutionised Computer Go, leading to the first strong programs that are competitive with human players. Work in this field is ongoing; in this section we outline some of the key developments.

Monte-Carlo tree search, as described in Chapter 3, was first introduced in the Go program *Crazy Stone* (Coulom, 2006). The true value of each move is assumed to have a Gaussian distribution centred on the current value estimate, $Q^\pi(s,a) \sim \mathcal{N}(Q(s,a), \sigma^2(s,a))$. During the first stage of simulation, the tree policy selects each move according to its probability of being better than the current best move, $\pi(s,a) \approx Pr(\forall b, Q^\pi(s,a) > Q(s,b))$. During the second stage of simulation, the default policy selects moves with a probability proportional to an *urgency* value encoding domain specific knowledge. In addition, *Crazy Stone* used a hybrid backup to update values in the tree, which is intermediate between a minimax backup and a expected value backup. Using these techniques,

[5] Gradually reducing the temperature, as in simulated annealing, was not beneficial.

Crazy Stone exceeded 1800 Elo on CGOS and won the gold medal at the 2006 9×9 Computer Go Olympiad.

The Go program *MoGo* introduced the UCT algorithm to Computer Go (Gelly et al., 2006; Wang and Gelly, 2007). *MoGo* treats each position in the search tree as a multi-armed bandit[6]. There is one arm of the bandit for each legal move, and the payoff from an arm is the outcome of a simulation starting with that move. During the first stage of simulation, the tree policy selects moves using the UCB1 algorithm (see Chapter 3). During the second stage of simulation, *MoGo* uses a default policy based on specialised domain knowledge. Unlike the enormous pattern databases used in traditional Go programs, *MoGo*'s patterns are extremely simple. Rather than suggesting the best move in any situation, these patterns are intended to produce local sequences of plausible moves. They can be summarised by four prioritised rules following an opponent move a:

1. If a put our stones into atari, play a saving move at random.
2. Otherwise, if one of the 8 intersections surrounding a matches a simple pattern for cutting or *hane*, randomly play one.
3. Otherwise, if any opponent stone can be captured, play a capturing move at random.
4. Otherwise play a random move.

Using these patterns in the UCT algorithm, *MoGo* significantly outperformed all previous 9×9 Go programs, exceeding 2100 Elo on the Computer Go Server.

The UCT algorithm in *MoGo* was subsequently replaced by the heuristic MC-RAVE algorithm (Gelly and Silver, 2007) (see Chapter 9). In 9×9 Go *MoGo* reached 2500 Elo on CGOS, achieved *dan*-level play on the Kiseido Go Server, and defeated a human professional in an even game for the first time (Gelly and Silver, 2008). These enhancements also enabled *MoGo* to perform well on larger boards, winning the gold medal at the 2007 19×19 Computer Go Olympiad.

The default policy used by *MoGo* is handcrafted. In contrast, a subsequent version of *Crazy Stone* uses supervised learning to train the pattern weights for its default policy (Coulom, 2007). The relative strengths of patterns is estimated by assigning them Elo ratings, much like a tournament between games players. In this approach, the pattern selected by a human player is considered to have won against all alternative patterns. In general, multiple patterns may match a particular move, in which case this team of patterns is considered to have won against alternative teams. The strength of a team is estimated by the product of the individual pattern strengths. The probability of each team winning is assumed to be proportional to that team's strength, using a generalised Bradley-Terry model (Hunter, 2004). Given a data set of expert moves, the maximum likelihood pattern strengths can be efficiently approximated by the minorisation-maximisation algorithm. This algorithm was

[6] In fact, the search tree is not a true multi-armed bandit, as there is no cost associated with exploration. In addition the simulation policy continues to change as the search tree is updated, which means that the payoff is non-stationary.

Program	*Description*	*Elo*
Indigo	Handcrafted patterns, Monte-Carlo simulation	1700
Magog	Supervised learning, neural network, alpha-beta search	1700
GnuGo, Many Faces of Go	Handcrafted patterns, local search	1800
NeuroGo	Reinforcement learning, neural network, alpha-beta search	1850
RLGO	Dyna-2, alpha-beta search	2150
MoGo, Fuego, Greenpeep	Handcrafted patterns, heuristic MC-RAVE	2500+
CrazyStone, Zen	Supervised learning of patterns, heuristic MC-RAVE	2500+

Table 4.1: Approximate Elo ratings of the strongest 9×9 programs using various paradigms on the 9x9 Computer Go Server.

Program	*Description*	*Rank*
Indigo	Handcrafted patterns, Monte-Carlo simulation	6 kyu
GnuGo, Many Faces of Go	Handcrafted patterns, local search	6 kyu
Mogo, Fuego	Handcrafted patterns, heuristic MC-RAVE	2 kyu
CrazyStone, Zen	Supervised learning of patterns, heuristic MC-RAVE	1 kyu, 1 dan

Table 4.2: Approximate Elo ratings of the strongest 19x19 Go programs using various paradigms on the Kiseido Go Server.

used to train the strengths of 17,000 patterns that were harvested from the data set. Using these patterns in its default policy, and progressively widening the search tree (Chaslot et al., 2007), *Crazy Stone* achieved a rating of 1 *kyu* at 19×19 Go against human players on the Kiseido Go Server.

The *Zen* program has combined ideas from both *MoGo* and *Crazy Stone*, using more sophisticated domain knowledge. *Zen* has achieved a 1 *dan* rating, on full-size boards, against human players on the Kiseido Go Server.

Monte-Carlo tree search can be parallelised much more effectively than traditional search techniques (Chaslot et al., 2008b). Recent work on *Mogo* has focused on full size 19×19 Go, using massive parallelisation (Gelly et al., 2008) and incorporating additional expert knowledge into the search tree. A version of *MoGo* running on 800 processors defeated a 9-*dan* professional player with 7 stones handicap. The latest version of *Crazy Stone* and a new, Monte-Carlo version of *The Many Faces of Go* have also achieved impressive victories against professional players on full-size boards[7].

4.7 Summary

We provide a summary of the current state of the art in Computer Go, based on ratings from the Computer Go Server (see Table 4.1) and the Kiseido Go Server (see Table 4.2). The Go programs to which this thesis has directly contributed are highlighted in bold[8].

[7] Nick Wedd maintains a website of all human versus computer challenge matches: *http://www.computer-go.info/h-c/index.html*.

[8] In fact many of the top Go programs, including *The Many Faces of Go*, *Crazystone*, *Fuego*, *Greenpeep* and *Zen* now use variants of the RAVE and heuristic UCT algorithms (see Chapter 9).

Part II

Temporal Difference Learning and Search

Chapter 5

Temporal Difference Learning with Local Shape Features

5.1 Introduction

A number of notable successes in artificial intelligence have followed a straightforward strategy: the state is *represented* by many simple features; states are *evaluated* by a weighted sum of those features, in a high-performance search algorithm; and weights are *trained* by temporal-difference learning. In two-player games as varied as Chess, Checkers, Othello, Backgammon and Scrabble, programs based on variants of this strategy have exceeded human levels of performance.

- In each game, the position is broken down into a large number of *features*. These are usually binary features that recognise a small, local pattern or configuration within the position: material, pawn structure and king safety in Chess (Campbell et al., 2002); material and mobility terms in Checkers (Schaeffer et al., 1992); configurations of discs in Othello (Buro, 1999); checker counts in Backgammon (Tesauro, 1994); and single, duplicate and triplicate letter rack leaves in Scrabble (Sheppard, 2002).

- The position is evaluated by a linear combination of these features with *weights* indicating their value. Backgammon provides a notable exception: *TD-Gammon* evaluates positions with a non-linear combination of features, using a multi-layer perceptron[1]. Linear evaluation functions are fast to compute; easy to interpret, modify and debug; and they have good convergence properties in many learning algorithms.

- Weights are trained from games of self-play, by temporal-difference or Monte-Carlo learning. The world champion Checkers program *Chinook* was hand-tuned by experts over 5 years. When weights were trained instead by self-play using temporal difference learning, the program equalled the performance of the original version (Schaeffer et al., 2001). A similar approach attained master level play in Chess (Baxter et al., 1998). *TD-Gammon* achieved world

[1] In fact, Tesauro notes that evaluating positions by a linear combination of Backgammon features is a "surprisingly strong strategy" (Tesauro, 1994).

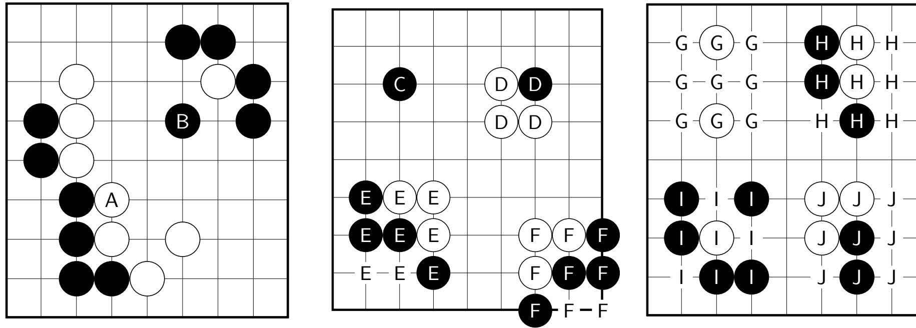

Figure 5.1: a) The sequence of stones from A forms a common *joseki* that is specific to the 4-4 intersection. Black B captures the white stone using a *tesuji* that can occur at any location. b) In general a stone on the 3-3 intersection (C) helps secure a corner. If it is surrounded then the corner is insecure (D), although with sufficient support it will survive (E). However, the same shape closer to the corner is unlikely to survive (F). c) Professional players describe positions using a large vocabulary of shapes. In this position we see examples of the *one-point jump* (G), *hane* (H), *net* (I) and *turn* (J).

class Backgammon performance after training by temporal-difference learning and self-play (Tesauro, 1994). Games of self-play were also used to train the weights of the world champion Othello and Scrabble programs, using least squares regression and a domain specific solution respectively (Buro, 1999; Sheppard, 2002)[2].

- A linear evaluation function is combined with a suitable search algorithm to produce a high-performance game playing program. Alpha-beta search variants have proven particularly effective in Chess, Checkers, Othello and Backgammon (Campbell et al., 2002; Schaeffer et al., 1992; Buro, 1999; Tesauro, 1994), whereas Monte-Carlo simulation has been most successful in Scrabble and Backgammon (Sheppard, 2002; Tesauro and Galperin, 1996).

In contrast to these games, the ancient oriental game of Go has proven to be particularly challenging. Handcrafted and machine learnt evaluation functions have so far been unable to achieve good performance (Müller, 2002). It has often been speculated that position evaluation in Go is uniquely difficult for computers because of its intuitive nature, and requires an altogether different approach to other games.

In this chapter, we return to the strategy that has been so successful in other domains, and apply it to Go. We develop a systematic approach for representing Go knowledge using local shape features. We evaluate positions using a linear combination of these features, and learn weights by temporal-difference learning and self-play. Finally, we incorporate a simple alpha-beta search algorithm.

5.2 Shape Knowledge in Go

The concept of shape is extremely important in Go. A good shape uses local stones efficiently to maximise tactical advantage (Matthews, 2003). Professional players analyse positions using a large vocabulary of shapes, such as *joseki* (corner patterns) and *tesuji* (tactical patterns). The joseki at the bottom left of Figure 5.1a is specific to the white stone on the 4-4 intersection, whereas the tesuji at the top-right could be used at any location. Shape knowledge may be represented at different scales, with more specific shapes able to specialise the knowledge provided by more general shapes (Figure 5.1b). Many Go proverbs exist to describe shape knowledge, for example "*ponnuki* is worth 30 points", "the one-point jump is never bad" and "*hane* at the head of two stones" (Figure 5.1c).

Commercial Computer Go programs rely heavily on the use of pattern databases to represent shape knowledge (Müller, 2002). Many man-years have been devoted to hand-encoding professional expertise in the form of local pattern rules. Each pattern recommends a move to be played whenever a specific configuration of stones is encountered on the board. The configuration can also include additional features, such as requirements on the liberties or strength of a particular stone. Unfortunately, pattern databases suffer from the knowledge acquisition bottleneck: expert shape knowledge is hard to quantify and encode, and the interactions between different patterns may lead to unpredictable behaviour.

Prior work on learning shape knowledge has focused on predicting expert moves by supervised learning (Stoutamire, 1991; van der Werf et al., 2002). Although success rates of around 40% have been achieved in predicting expert moves, this approach has not led to strong play in practice. This may be due to its focus on mimicking rather than evaluating the consequences of the local shapes.

A second approach has been to train a multi-layer perceptron, using temporal-difference learning by self-play (Schraudolph et al., 1994). The networks implicitly contain some representation of local shape, and utilise weight sharing to exploit the natural symmetries of the Go board. This approach has led to stronger Go playing programs, such as Dahl's *Honte* (Dahl, 1999) and Enzenberger's *NeuroGo* (Enzenberger, 2003). However, these networks utilise a great deal of sophisticated Go knowledge in the network architecture and input features. Furthermore, knowledge learnt in this form cannot be directly interpreted or modified in the manner of pattern databases.

5.3 Local Shape Features

We introduce a much simpler approach for representing shape knowledge, which requires no prior knowledge of the game, except for the basic grid structure of the board.

A state in the game of Go, $s \in \{\cdot, \circ, \bullet\}^{N \times N}$, consists of a state variable for each intersection of a size $N \times N$ board, with three possible values for empty, black and white stones respectively[3]. We

[2]Sheppard reports that temporal-difference learning performed poorly, due to insufficient exploration (Sheppard, 2002).
[3]Technically the state also includes the full board history, so as to avoid repetitions (known as *ko*).

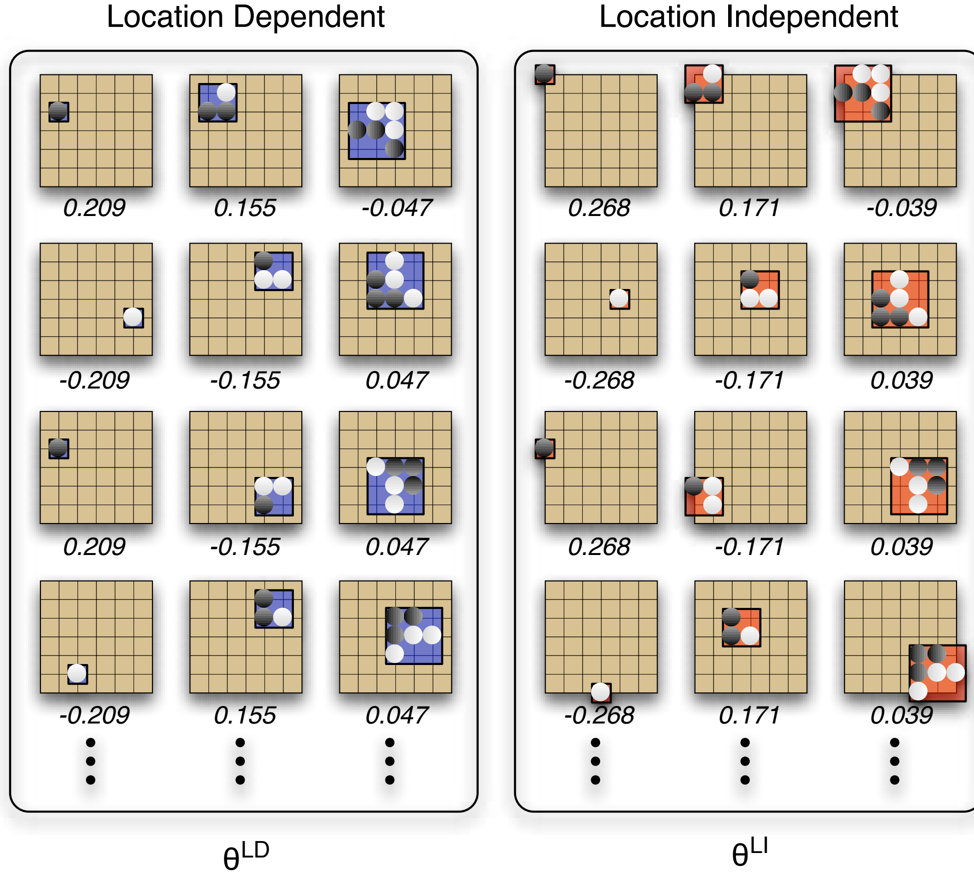

Figure 5.2: Examples of location dependent and location independent weight sharing.

define a *local shape* to be a specific configuration of state variables within some square region of the board. We exhaustively enumerate all possible local shapes within all possible square regions up to size $m \times m$. The *local shape feature* $\phi_i(s)$ has value 1 in position s if the board exactly matches the ith local shape, and value 0 otherwise.

The local shape features are combined into a large feature vector $\phi(s)$. This feature vector is very sparse: exactly one local shape is matched in each square region of the board; all other local shape features have value 0.

A vector of weights θ indicates the value of each local shape feature. A position s is evaluated by a linear combination of features and their corresponding weights, squashed into the range $[0, 1]$ by a logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$.

$$V(s) = \sigma(\phi(s).\theta) \tag{5.1}$$

5.4 Weight Sharing

We use weight sharing to exploit the symmetries of the Go board (Schraudolph et al., 1994). We define an equivalence relationship over local shapes, such that all rotationally and reflectionally symmetric local shapes are placed in the same equivalence class. In addition, each equivalence class includes *inversions* of the local shape, in which all black and white stones are flipped to the opposite colour[4].

The local shape with the smallest index j within the equivalence class is considered to be the canonical example of that class. Every local shape feature ϕ_i in the equivalence class shares the weight θ_j of the canonical example, but the sign may differ. If the local shape feature has been inverted from the canonical example, then it uses negative weight sharing, $\theta_i = -\theta_j$, otherwise it uses positive weight sharing, $\theta_i = \theta_j$. In certain equivalence classes (for example empty local shapes), an inverted shape is identical to an uninverted shape, so that either positive or negative weight sharing could be used, $\theta_i = \theta_j = -\theta_j \Rightarrow \theta_i = \theta_j = 0$. We describe these local shapes as *neutral*, and assume that they are equally advantageous to both sides. All neutral local shapes are removed from the representation (see Figure 5.3.

The rotational, reflectional and inversion symmetries define the vector of *location dependent* weights θ^{LD}. The vector of *location independent* weights θ^{LI} also incorporates translation symmetry: all local shapes that have the same configuration, regardless of its location on the board, are included in the same equivalence class. Figure 5.4 shows some examples of both types of weight sharing.

For each size of square up to 3×3, all local shape features are exhaustively enumerated, using both location dependent and location independent weights. This provides a hierarchy of local shape features, from very general configurations that occur many times each game, to specific configurations that are rarely seen in actual play. Smaller local shape features are more general than larger ones, and location independent weights are more general than location dependent weights. The more general features and weights provide no additional information, but may offer a useful abstraction for rapid learning. Table 5.4 shows the total number of local shape features of each size; the total number of distinct equivalence classes, and therefore the total number of unique weights; and the maximum number of active features (features with value of 1) in the feature vector.

5.5 Learning algorithm

Our objective is to win games of Go. This goal can be expressed by a binary reward function, which gives a reward of $r = 1$ if Black wins and $r = 0$ if White wins, with no intermediate rewards. The value function $V^\pi(s)$ is defined to be the expected total reward from board position s when following policy π. This value function is Black's *winning probability* from state s. Black seeks

[4]The first player advantage and *komi* mean that the game is not truly colour symmetric.

Local shape features	Total features	Unique weights	Max active features
1×1 Location Independent	243	1	81
1×1 Location Dependent		15	81
2×2 Location Independent	5184	8	64
2×2 Location Dependent		344	64
3×3 Location Independent	964467	1418	49
3×3 Location Dependent		61876	49
Total	969894	63303	388

Table 5.1: Number of local shape features of different sizes in 9×9 Go.

to maximise his winning probability, while White seeks to minimise it. We approximate the value function by a linear combination of local shape features and both location dependent and location independent weights (see Figure 5.3),

$$V(s) = \sigma(\phi(s).\theta^{LI} + \phi(s).\theta^{LD}) \tag{5.2}$$

We measure the TD-error between the current value $V(s_t)$, and the value after both player and opponent have made a move, $V(s_{t+2})$. In this approach, which we refer to as a *two-ply update*, the value is updated between successive moves with the same colour to play. The current player is viewed as the agent, and his opponent is viewed as part of the environment. We contrast this approach to a *one-ply update*, used in prior work such as *TD-Gammon* (Tesauro, 1994) and *NeuroGo* (Enzenberger, 2003), that measures the TD-error between Black and White moves.

We update both location dependent and location independent weights by logistic temporal-difference learning (see Appendix A). For each feature ϕ_i, the shared value for the corresponding weights θ_i^{LI} and θ_i^{LD} is updated. This can lead to the same shared weight being updated many times in a single time-step. As in the NLMS algorithm (Haykin, 1996), we normalise the update by dividing by the total number of active features $|\phi(s_t)|$ in the current state s_t,

$$\Delta\theta^{LD} = \Delta\theta^{LI} = \alpha \frac{\phi(s_t)}{|\phi(s_t)|}(V(s_{t+2}) - V(s_t)) \tag{5.3}$$

As in the Sarsa algorithm (see Chapter 2) the policy is updated after every move t, by using an ϵ-greedy policy. With probability $1 - \epsilon$ the player selects the move that maximises (Black) or minimises (White) the value function $a = \underset{a'}{\operatorname{argmax}}\ V(s \circ a')$. With probability ϵ the player selects a move with uniform random probability.

Because the local shape features are sparse, only a small subset of features need be evaluated and updated. This leads to an an efficient $O(k)$ implementation, where k is the total number of active features. This requires just a few hundred operations, rather than evaluating or updating a million components of the full feature vector.

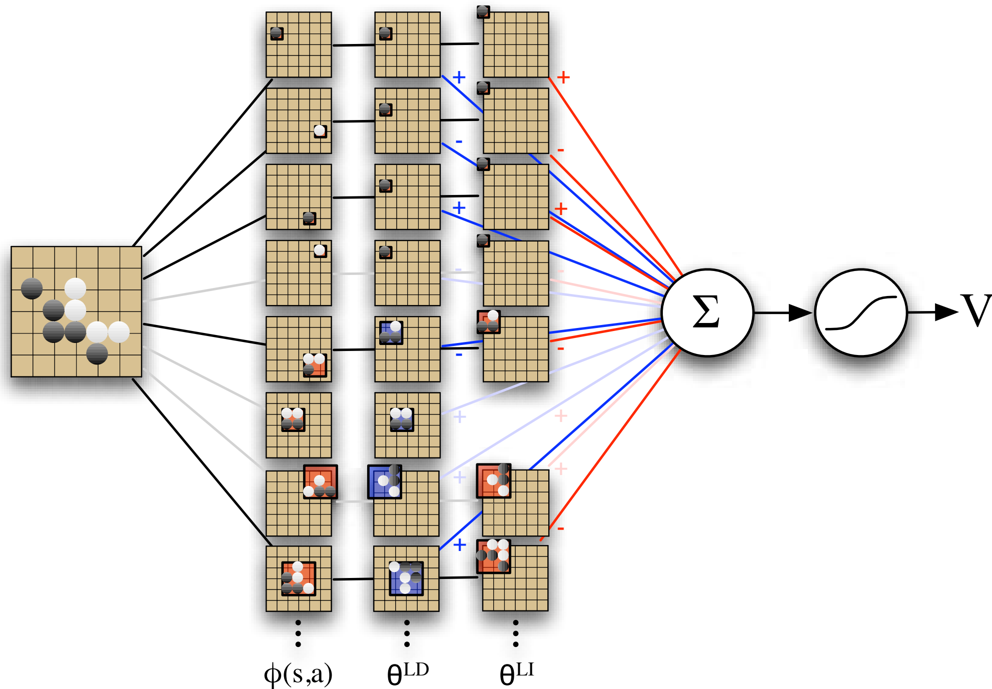

Figure 5.3: Evaluating a position by local shape features, using location dependent and location independent weight sharing. Neutral weights (e.g. the gap in the location independent weights) are assumed to be zero and are removed from the representation.

5.6 Training

We initialise all weights to zero. We train the weights by executing a million games of self-play, in 9×9 Go. Both Black and White select moves using an ϵ-greedy policy over the same value function $V(s)$. The same weights are used by both players, and updated after both Black and White moves by Equation 5.3.

All games begin from the empty board position, and terminate when both players pass. To prevent games from continuing for an excessive number of moves, we prohibit moves within single-point eyes, and only allow the pass move when no other legal moves are available. In addition, any game that exceeds 1000 moves is declared a draw, in which case both players are given a reward of $r = 0.5$. Games which successfully terminate are scored by Chinese rules, using a *komi* of 7.5.

5.7 A Case Study in 9×9 Computer Go

We implemented the learning algorithm and training procedure described above in our Computer Go program *RLGO*.

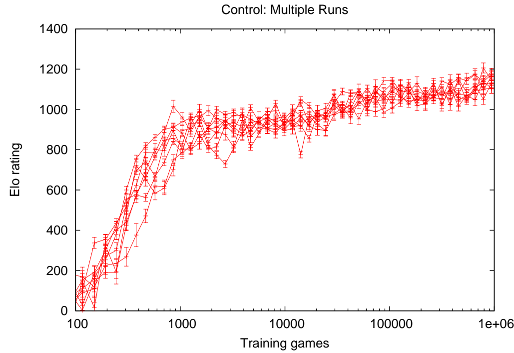

Figure 5.4: Multiple runs using the default learning algorithm and local shape features.

5.7.1 Experimental Setup

In this case study we compare learning curves for *RLGO*, using a variety of parameter choices. This requires some means to evaluate the performance of our program for thousands of different combinations of parameters and weights. Measuring performance online against human or computer opposition is only feasible for one or two programs. Measuring performance against a standardised computer opponent, such as *GnuGo*, is only useful if the opponent is of a similar standard to the program. When Go programs differ in strength by many orders of magnitude, this leads to a large number of matches in which the result is an uninformative whitewash. Furthermore, we would prefer to measure performance robustly against a variety of different opponents.

In each experiment, after fully training *RLGO* with several different parameter settings, we ran a tournament between multiple versions of *RLGO*. Each version used the weights learnt after training from n games with a particular parameter setting. Each tournament included a variety of different n, for a number of different parameter settings. In addition, we included the program *GnuGo* 3.7.10 (level 10) in every tournament, to anchor the absolute performance between different tournaments. Each tournament consisted of at least 1000 games for each version of *RLGO*. After all matches were complete, the results were analysed by the *bayeselo* program to establish an Elo rating for every program. Following convention, *GnuGo* was assigned an anchor rating of 1800 Elo in all cases.

Unless otherwise specified, we used default parameter settings of $\alpha = 0.1$ and $\epsilon = 0.1$. All local shape features were used for all square regions from 1×1 up to 3×3, using both location dependent and location independent weight sharing. The logistic temporal-difference learning algorithm was used, with two-ply updates (see Equation 5.3).

Due to limitations on computational resources, just one training run was executed for each parameter setting. However, Figure 5.4 demonstrates that our learning algorithm is remarkably consistent, producing the same performance over the same timescale in 8 different training runs. Thus the conclusions that we draw from single training runs, while not definitive, are very likely to be repeatable.

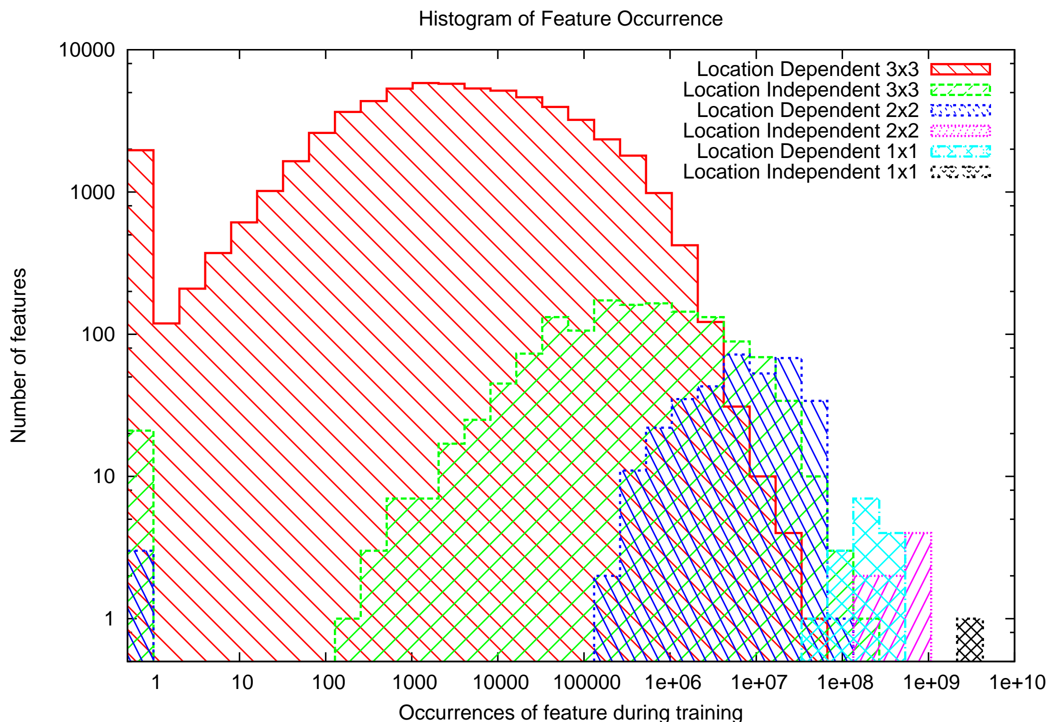

Figure 5.5: Histogram of feature occurrences during a training run of 1 million games.

5.7.2 Local Shape Features in 9×9 Go

Perhaps the single most important property of local shape features is their huge range of generality. To assess this range, we counted the number of times that each equivalence class of feature occurs during training, and plotted a histogram for each size of local shape and for each type of weight sharing (see Figure 5.5). Each histogram forms a characteristic curve in log-log space. The most general class, the location independent 1×1 feature representing the material value of a stone, occurred billions of times during training. At the other end of the spectrum, there were tens of thousands of location dependent 3×3 features that occurred just a few thousand times, and 2000 that were never seen at all. In total, each class of feature occurred approximately the same amount overall, but these occurrences were distributed in very different ways. Our learning algorithm must cope with this varied data: high-powered signals from small numbers of general features, and low-powered signals from a large number of specific features.

We ran several experiments to analyse how different combinations of local shape features affect the learning rate and performance of *RLGO*. In our first experiment, we used a single size of square region (see Figure 5.6, top-left). The 1×1 local shape features, unsurprisingly, performed poorly. The 2×2 local shape features learnt very rapidly, but their representational capacity was saturated at around 1000 Elo after approximately 2000 training games. Surprisingly, performance appeared to decrease after this point, perhaps suggesting that the policy converged to a somewhat stable solution

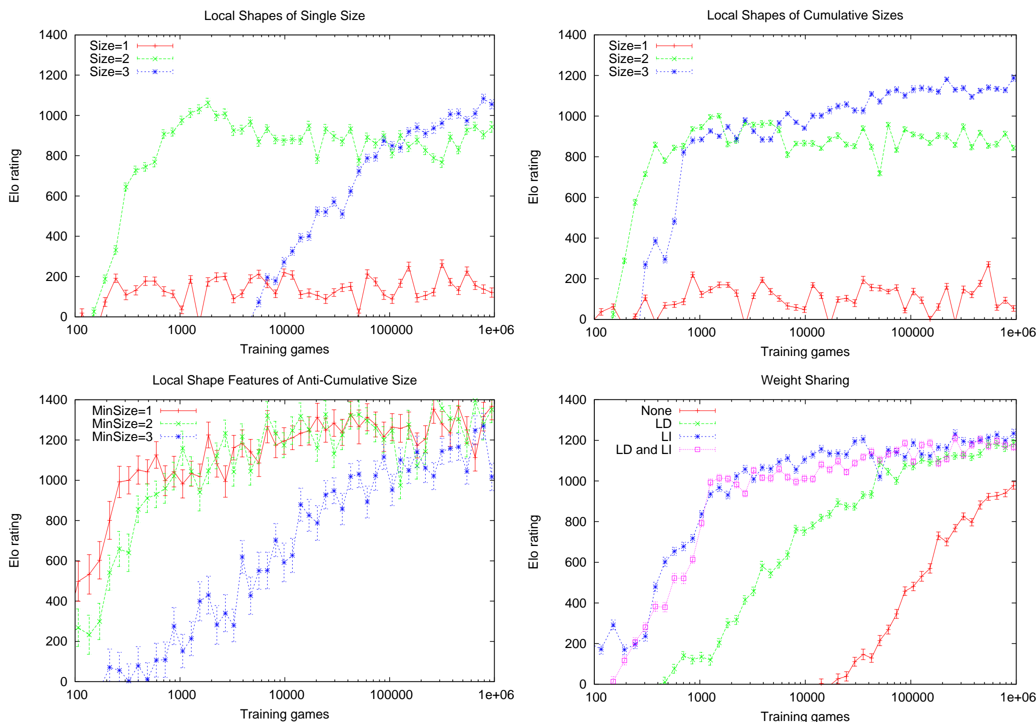

Figure 5.6: Learning curves for just one size of local shape feature: 1×1, 2×2 or 3×3 (top-left), 1×1. Learning curves for cumulative sizes of local shape feature: 1×1; 1×1 and 2×2; or 1×1, 2×2 and 3×3 (top-right). Learning curves for anti-cumulative sizes of local shape feature: 1×1, 2×2 and 3×3; 1×1 and 2×2; or 3×3 (bottom-left). Learning curves for different weight sharing rules (bottom-right).

that is less representative of the range of opponents encountered during tournament play. The 3×3 local shape features learnt very slowly, but exceeded the performance of the 2×2 features after around 100000 training games.

In our next experiment, we combined multiple sizes of square region (see Figure 5.6, top-right and bottom-left). Using all features up to 3×3 effectively combined the rapid learning of the 2×2 features with the better representational capacity of the 3×3 features; overall performance was around 200 Elo better than for any single shape set, reaching 1200 Elo, and apparently still improving slowly. We conclude that a redundant representation, in which the same information is represented at multiple levels of generality, confers a significant advantage during learning.

In our final experiment with local shape features, we compared a variety of different weight sharing schemes (see Figure 5.6, bottom-right). Without any weight sharing, learning was very slow, eventually achieving 1000 Elo after a million training games. Location dependent weight sharing provided an intermediate rate of learning, and location independent weights provided the fastest learning. Perhaps surprisingly, the eventual performance of the location independent weights was equivalent to the location dependent weights, and combining both types of weight sharing together offered no additional benefits. This suggests that the additional knowledge offered by location

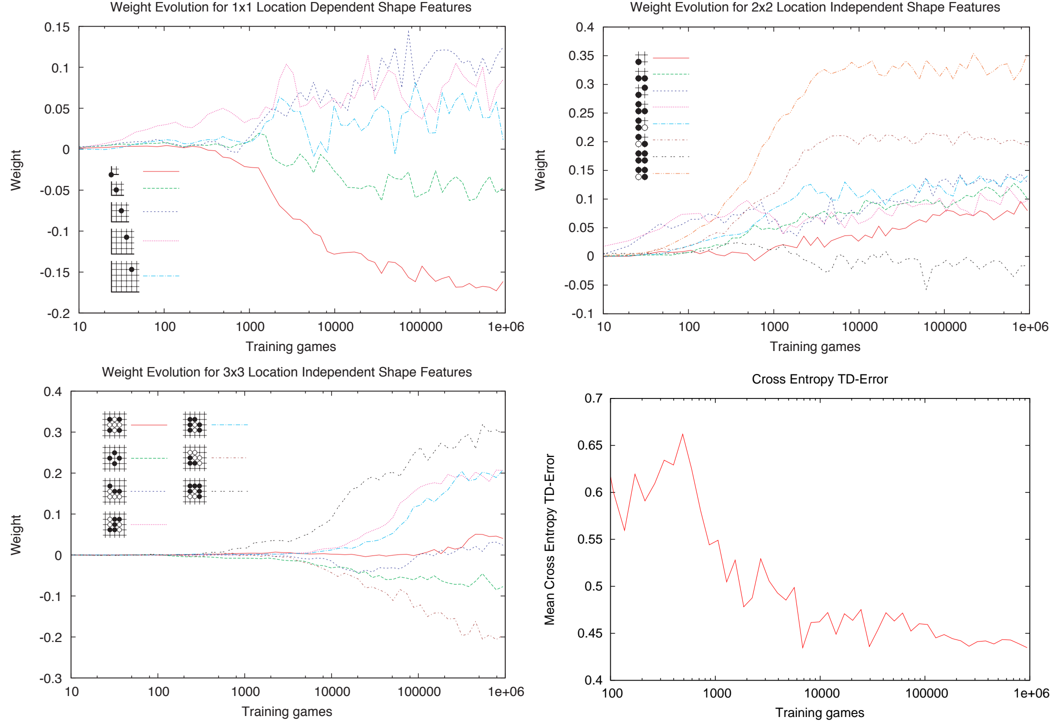

Figure 5.7: Weight evolution during training: several location independent 1×1 weights (top-left); the location dependent 2×2 weights (top-right); several several location independent 1×1 weights (bottom-left); and the evolution of the cross entropy TD-error (bottom-right).

dependent shapes, for example patterns that are specific to edge or corner situations, was either not useful or not successfully learnt within the training time of these experiments.

5.7.3 Weight Evolution

Figure 5.7 shows the evolution of several feature weights during a single training run. Among the location independent 2×2 features, the efficient *turn* and *hane* shapes were quickly identified as the best, and the inefficient *dumpling* as the worst. The location dependent 1×1 features quickly established the value of stones in central board locations over edge locations. The 3×3 weights took several thousand games to move away from zero, but appeared to have stabilised ttowards the end of training.

The logistic TD algorithm (see Appendix A) reduces the cross entropy TD-error $-V(s_{t+1}logV(s_t) - (1 - V(s_{t+1})log(1 - V(s_t))$. Figure (see Figure 5.7d) shows how how the average cross entropy TD-error decreases with additional training.

5.7.4 Logistic Temporal-Difference Learning

In Figure 5.8 we compare our logistic temporal-difference learning algorithm to linear temporal-difference learning algorithm, for a variety of different step-sizes α. In the latter approach, the value function is represented directly by a linear combination of features, with no logistic function; the

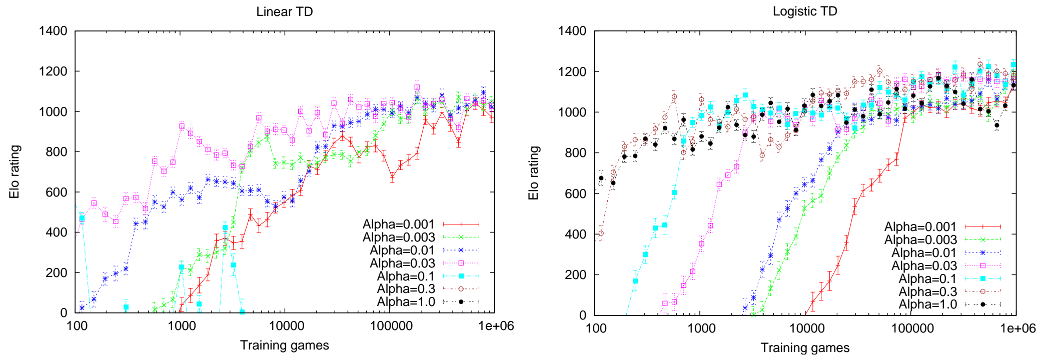

Figure 5.8: Comparison of linear (top-left) and logistic-linear (top-right) temporal-difference learning. Linear temporal-differencing learning diverged for step-sizes of $\alpha \geq 0.1$.

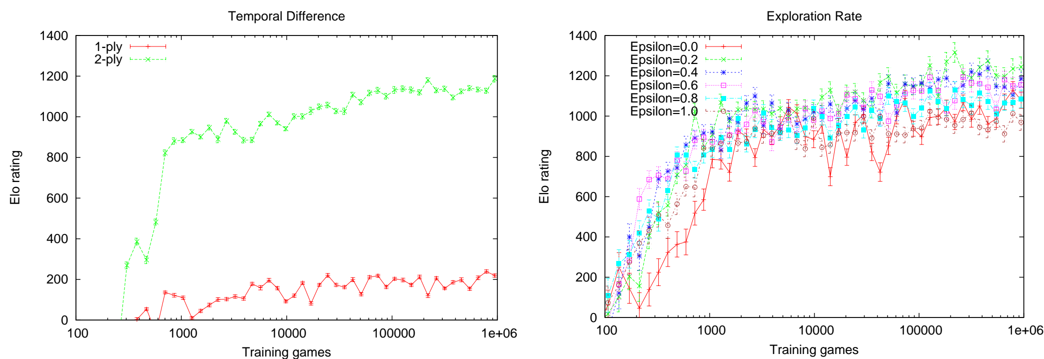

Figure 5.9: Learning curves for one-ply and two-ply updates (left), and for different exporation rates ϵ (right).

weight update equation is otherwise identical to Equation 5.3.

Logistic temporal-difference learning is considerably more robust to the choice of step-size. It achieved good performance across three orders of magnitude of step-size, and improved particularly quickly with an aggressive learning rate. With a large step-size, the value function steps up or down the logistic function in giant strides. This effect can be visualised by zooming out of the logistic function until it looks much like a step function. In contrast, linear temporal-difference learning was very sensitive to the choice of step-size, and diverged when the step-size is too large.

Logistic temporal-difference learning also achieved better eventual performance. This suggests that, much like logistic regression for supervised learning (Jordan, 1995), the logistic representation is better suited to representing probabilistic value functions.

5.7.5 Self-Play

When training from self-play, temporal-difference learning can use either one-ply or two-ply updates (see Section 5.5). We compare the performance of these two updates in Figure 5.9a. Surprisingly, one-ply updates, which were so effective in *TD-Gammon*, performed very poorly in *RLGO*. This is due to our more simplistic representation: *RLGO* does not differentiate the colour to play. Because of

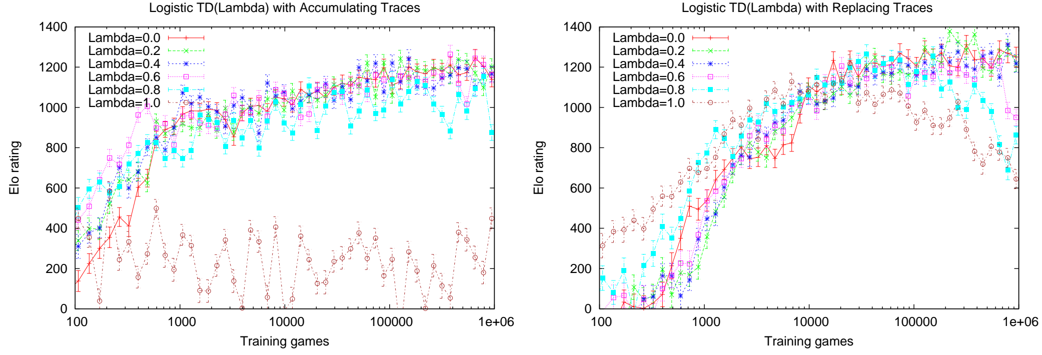

Figure 5.10: Learning curves for different values of λ, using accumulating traces (left) and replacing traces (right).

this, whenever a player places down a stone, the value function is improved for that player. This leads to a large TD-error corresponding to the current player's advantage, which cannot ever be corrected. This error signal overwhelms the information about the relative strength of the move, compared to other possible moves. By using two-ply updates, this problem can be avoided altogether[5].

Figure 5.9b compares the performance of different exploration rates ϵ. As might be expected, the performance decreases with increasing levels of exploration. However, without any exploration learning was much less stable, and $\epsilon > 0$ was required for robust learning. This is particularly important when training from self-play: without exploration the games are perfectly deterministic, and the learning process may become locked into local, degenerate solutions.

5.7.6 Logistic TD(λ)

Logistic temporal-difference learning algorithm can be extended to incorporate a λ parameter that determines the time-span of the temporal difference. When $\lambda = 1$, learning updates are based on the final outcome of the complete game, which is equivalent to logistic Monte-Carlo (see Appendix A). When $\lambda = 0$, learning updates are based on a one-step temporal difference, which is equivalent to the basic logistic temporal-difference learning update. We implement logistic TD(λ) by maintaining a vector of eligibility traces z that measures the credit assigned to each feature during learning (see Chapter 2), and is initialised to zero at the start of each game. We consider two eligibility update equations, based on accumulating and replacing eligibility traces,

$$z_{t+1} \leftarrow \lambda z_t + \frac{\phi(s_t)}{|\phi(s_t)|} \qquad \text{using accumulating traces} \qquad (5.4)$$

$$z_{t+1} \leftarrow (1 - \phi(s_t))\lambda z_t + \frac{\phi(s_t)}{|\phi(s_t)|} \qquad \text{using replacing traces} \qquad (5.5)$$

$$\Delta\theta_t = \alpha(V(s_{t+2}) - V(s_t))z_t \qquad (5.6)$$

[5] Mayer also reports an advantage to two-ply TD(0) when using a simple multi-layer perceptron architecture (Mayer, 2007).

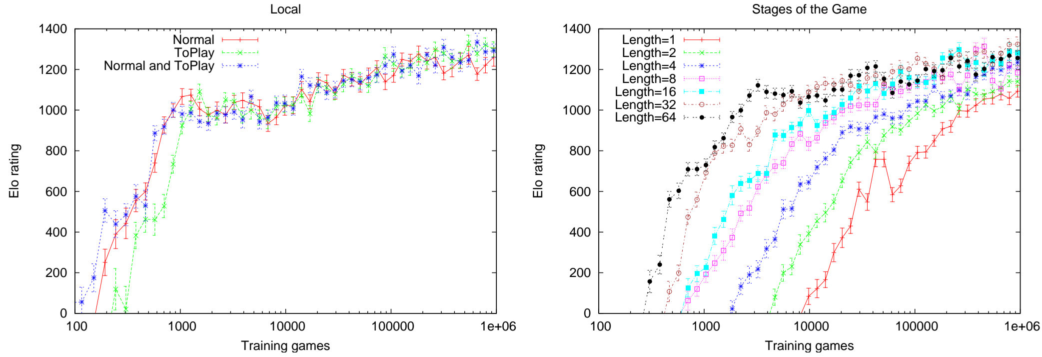

Figure 5.11: Differentiating the colour to play (left) and differentiating the stage of the game, by binning the move number at a variety of resolutions (right)

We compared the performance of logistic TD(λ) for different settings of λ. High values of λ, especially $\lambda = 1$, performed substantially worse with accumulating traces. With replacing traces, high values of λ were initially beneficial, but the performance dropped off with more learning, suggesting that the high variance of the updates was less stable in the long run. The difference between lower values of λ, with either type of eligibility trace, was not significant.

5.7.7 Extended Representations in 9×9 Go

Local shape features are sufficient to represent a wide variety of intuitive Go knowledge. However, this representation of state is very simplistic: it does not represent which colour is to play, and it does not differentiate different stages of the game.

In our first experiment, we extend the local shape features so as to represent the colour to play. Three vectors of local shape features are used: $\phi^B(s)$ only match local shapes when Black is to play, $\phi^W(s)$ only match local shapes when White is to play, and $\phi^{BW}(s)$ matches local shapes when either colour is to play. We append these feature vectors together in three combinations:

1. $[\phi^{BW}(s)]$ is our basic representation, and does not differentiate the colour to play.

2. $[\phi^B(s); \phi^W(s)]$ differentiates the colour to play.

3. $[\phi^B(s); \phi^W(s); \phi^{BW}(s)]$ combines features that differentiate colour to play with features that do not.

Figure 5.11a compares the performance of these three approaches, showing no significant differences.

In our second experiment, we extend the local shape features so as to represent the stage of the game. Each local shape feature $\phi^Q(s_t)$ only matches local shapes when the current move t is within the Qth stage of the game. Each stage of the game lasts for T moves, and the Qth stage lasts from move QT until $(Q+1)T$. We consider a variety of different timescales T for the stages of the game, and analyse their performance in Figure 5.11b.

Search depth	Elo rating	Error
Depth 1	859	± 23
Depth 2	1067	± 20
Depth 3	1229	± 18
Depth 4	1226	± 20
Depth 5	1519	± 19
Depth 6	1537	± 19
GnuGo (level 10)	1800	± 26
GnuGo (level 0)	1638	± 19

Table 5.2: Performance of full width, fixed depth, alpha-beta search, using the learnt weights as an evaluation function. Weights were trained using default settings for 1 million training games. All depths were evaluated in a tournament, including *GnuGo* as a benchmark player.

Search depth	**Elo rating on CGOS**
1	1050
5	1350

Table 5.3: Elo ratings established by *RLGO v1.0* on the Computer Go Server.

Surprisingly, differentiating the stage of the game was strictly detrimental. The more stages that are used, the slower learning proceeds. The additional representational capacity did not offer any benefits within a million training games.

5.7.8 Alpha-Beta Search

To complete our study of position evaluation in 9×9 Go, we used the learnt value function $V(s)$ as a heuristic function to evaluate the leaf positions in a fixed-depth alpha-beta search. We ran a tournament between *GnuGo* and several versions of *RLGO* using an alpha-beta search up to various depths; the results are shown in Figure 5.2.

Alpha-beta search tournaments with the same program often exaggerate the performance differences between depths. To gain some additional insight into the performance of our program, *RLGO* played online in tournament conditions against a variety of different opponents on the Computer Go Server[6]. The Elo rating established by *RLGO* is shown in Table 5.3.

5.8 Conclusion

The knowledge learnt using local shape features represents a broad library of common-sense Go intuitions. Figure 5.1 displays the weights with the highest absolute magnitude within each class. The 1×1 shapes encode the basic material value of a stone. The 2×2 shapes measure the value of connecting and cutting; they encourage efficient shapes such as the turn, and discourage inefficient shapes such as the empty triangle. The 3×3 shapes represent several ways to split the opponent's stones, and three different ways to form two eyes in the corner.

[6]This was an older version of *RLGO*, trained from 100000 games of self-play. The Computer Go Server is now dominated by much stronger Monte-Carlo programs, and it is no longer possible to establish informative ratings for weaker players.

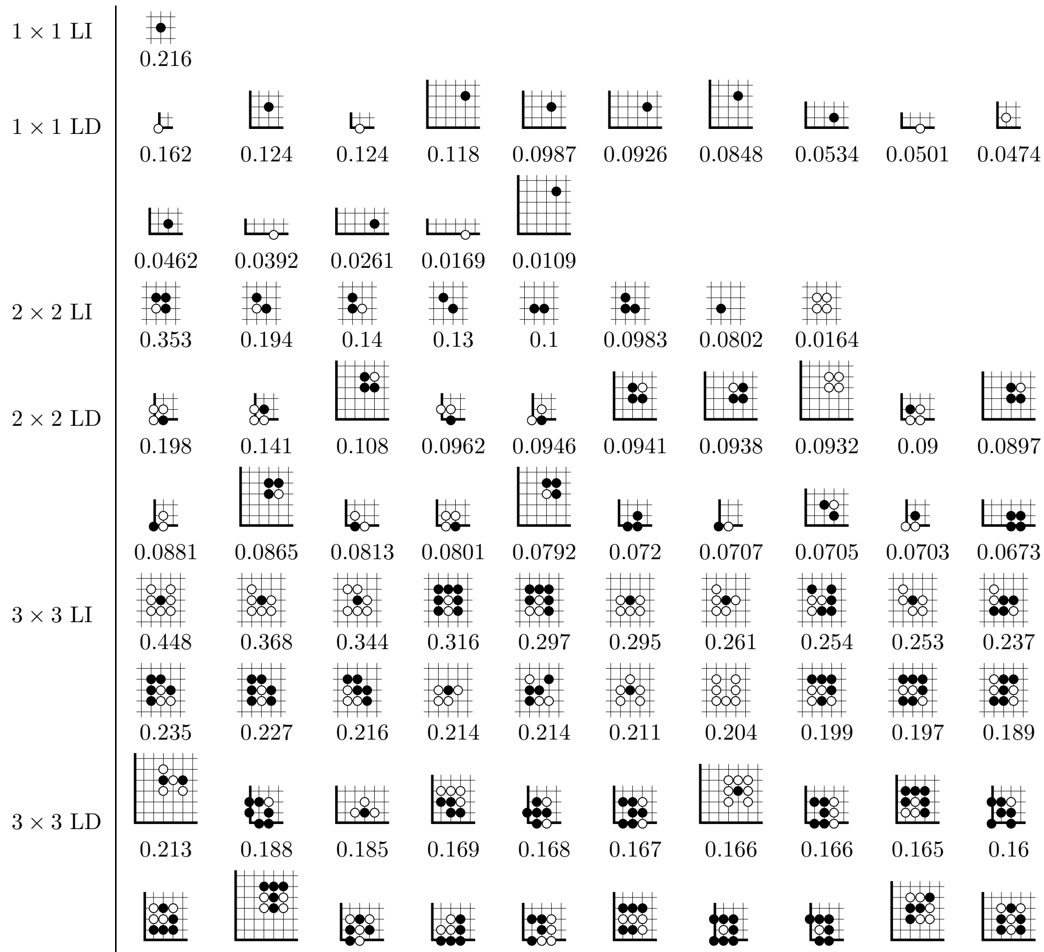

Figure 5.12: The top 20 shapes in each set from 1×1 to 3×3, location independent and location dependent, with the greatest absolute weight after training on a 9×9 board.

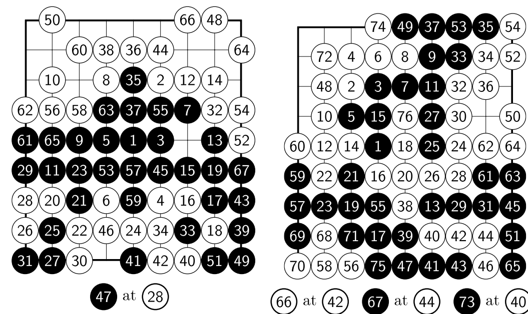

Figure 5.13: a) A game on the Computer Go Server between *RLGO v1.0* (white) and DingBat-3.2 (rated at 1577 Elo). RLGO plays a nice opening and develops a big lead. Moves 48 and 50 make good eye shape locally, but for the wrong group. DingBat takes away the eyes from the group at the bottom with move 51 and goes on to win. b) A game between *RLGO v1.0* (white) and the search based Liberty-1.0 (rated at 1110 Elo). RLGO plays good attacking shape from moves 24-37. It then extends from the wrong group, but returns later to make two safe eyes with 50 and 62 and ensure the win.

However, the whole is greater than the sum of its parts. Weights are learnt for tens of thousands of shapes, and the agent's play exhibits global behaviours beyond the scope of any single shape, such as territory building and control of the corners. Its principle weakness is its myopic view of the board; the agent will frequently play moves that look beneficial locally but miss the overall direction of the game, for example adding stones to a group that has no hope of survival.

Nevertheless, the performance achieved by this approach is very far from human levels. We tentatively conclude that the straightforward strategy of linear evaluation, temporal-difference learning, and alpha-beta search, which has proven so successful in other classical games, is insufficient to produce high-performance in Computer Go. In subsequent chapters, we develop a new paradigm for combining temporal-difference learning and search, which is much more successful in this challenging domain.

Chapter 6

Temporal-Difference Search

6.1 Introduction

Temporal-difference learning (Sutton, 1988) has proven remarkably successful in a wide variety of domains. In two-player games it has been used by the world champion Backgammon program *TD-Gammon* (Tesauro, 1994), the world champion Checkers program *Chinook* (Schaeffer et al., 2001), the master-level Chess program *Knightcap* (Baxter et al., 1998), and the strongest machine learned evaluation function in Go (Enzenberger, 2003). In every case, an evaluation function was learned offline, by training from thousands of games of self-play, and no further learning was performed online during actual play.

In this chapter we develop a very different paradigm for temporal-difference learning. In this approach, learning takes place *online*, so as to find the best evaluation function *for the current state*. Rather than training a very general evaluation function offline over many weeks or months, the agent trains a much more specific evaluation function online, in a matter of seconds or minutes.

In any game, the current position s_t defines a new game, that is specific to this position. In this subgame, the rules are the same, but the game always starts from position s_t. It may be substantially easier to solve the subgame than to solve the original game: the search space is reduced and a much smaller class of positions will typically be encountered. The subgame can have very different properties to the original game: certain patterns or features will be successful in this particular situation, which may not in general be a good idea. The idea of *temporal-difference search* is to apply temporal-difference learning to games of self-play, using the subgame for the current position s_t.

More generally, temporal-difference search can be applied to any problem for which the rules are known, a simulator is provided, or a model can be learnt. Rather than trying to learn a policy that covers every possible eventuality, the agent focuses on the subproblem that arises from the current state: how to perform well *now*. Life is full of such situations: we don't need to know how to climb every mountain in the world; but we'd better have a good plan for the one we are scaling right now.

6.2 Temporality

Much of machine learning can be characterised as the search for a solution that, once found, no longer needs be changed. Throughout conventional supervised and unsupervised learning the common presumption is that learning will be done in a separate phase and only after it is complete will the system be used. No learning is expected during the normal operation of the learned system.

In supervised learning, of course, it is often not possible for learning to continue during normal operation because appropriate training data are no longer be available. Reinforcement learning, on the other hand, is not strongly limited in this way; training data is continually available during normal operation. Nonetheless, the focus in the overwhelming majority of current work in these areas has been on finding a single, stationary solution. When reinforcement learning is applied in a game-playing context, as in the world's best backgammon player TD-Gammon (Tesauro, 1994), the objective is to find a single, static, high-quality evaluation function. When reinforcement learning is applied to learning to fly a helicopter (Ng et al., 2004), the search for a good policy is done in simulation and no learning is done while the helicopter is actually flying. When reinforcement learning is applied to learning a good gait for a robot dog (Kohl and Stone, 2004), the learning occurs during an extensive self-training period and does not continue after the dog is playing soccer. In many of these cases, even if the learning could continue during normal operation, the prior experience is so extensive and the adaptation so slow that no significant learning would occur during normal operation. The standard paradigm of machine learning, with a few notable exceptions, is about learning systems that converge to an optimal or good solution. In this sense, machine learning has been more concerned with the results of learning than with the ongoing process of learning.

Focusing only on the results of learning is of course not adequate when dealing with a non-stationary environment. If the environment may change over time, then no prior learning could ever be sufficient on its own. When the environment changes, mistakes will be made and, if learning does not continue, they will be made over and over again. The *temporality* of the environment must be acknowledged and solutions must be continually be updated so as to perform well in the environment as it is *now*. Nevertheless, while most researchers would acknowledge that temporality is important in non-stationary domains, this approach has not been extensively pursued. For the most part researchers have chosen to focus on the stationary case, in part because it is clearer, and in part perhaps because it is seen as a step best completed before moving on to the nonstationary case.

We suggest that temporality is important, even in stationary problems, and that focusing on convergence to a single solution may not be good enough. If training data comes from a system with evolving states, such as a Markov Decision Process, then temporality may become important just because the environment is very large. The learning agent encounters different parts of the environment at different times, but cannot fully represent the entire environment. In this case, rather than finding a single global solution, it may be advantageous for the agent to adapt to the local[1]

[1] By local here we mean temporally local.

environment — the specific part of the state space it finds itself encountering right *now* (Sutton et al., 2007).

6.3 Simulation-Based Search

In this chapter we introduce a general framework for simulation-based search. The basic idea is to generate episodes of experience, starting from the agent's current state, by sampling from the agent's own policy and from a model of the environment. By doing so, the agent samples episodes from the distribution of future experience that it would encounter if the model was correct. By learning from this distribution of future experience, rather than the distribution of all possible experience, the agent exploits the temporality of the environment[2]. It can focus its limited resources on what is likely to happen from *now* onwards, rather than learning about all possible eventualities.

Simulation-based search requires a *generative model* of the environment: a black box process for sampling a state transition from $\hat{\mathcal{P}}^a_{ss'}$ and a reward from $\hat{\mathcal{R}}^a_{ss'}$. The effectiveness of simulation-based search depends on the accuracy of the model, and learning a model can in general be a challenging problem.

In this thesis we sidestep the model learning problem and consider the special case of two-player zero sum games. In these games, the opponent's behaviour can be modelled by the agent's own policy. As the agent's policy improves, so the model of the opponent also improves. In addition, we assume that the rules of the game are known. By combining the rules of the game with our model of the opponent's behaviour, we can generate complete two-ply state transitions for each possible action. We refer to this approach as a *self-play model*. In addition, the rules of the game can be used to generate rewards. In two-player games there is typically a final reward that depends on the terminal outcome of the game (e.g. winning, drawing or losing), with no intermediate rewards.

By simulating experience from a model of the environment, the agent creates a new reinforcement learning problem. At every computational step u the agent receives a state s_u from the model, executes an action a_u according to its current policy $\pi_u(s, a)$, and then receives a reward r_{u+1} from the model. The idea of simulation-based search is to learn a policy that maximises the total future reward, in this simulation of the environment. Unlike other sample-based planning methods, such as Dyna (Sutton, 1990), simulation-based search seeks the specific policy that maximises expected total reward from the *current* state onwards.

6.4 Monte-Carlo Search

Monte-Carlo tree search provides one example of a simulation-based search algorithm. States are represented individually, with the value of each state stored in the nodes of a large search-tree. The

[2]In non-ergodic environments, such as episodic tasks, this distribution can be very different. However, even in ergodic environments, the short-term distribution of experience, generated by discounting or by truncating the simulations after a small number of steps, can be very different from the stationary distribution. This local *transient* in the problem can be exploited by an appropriately specialised policy.

value of each state is estimated by Monte-Carlo evaluation, from the mean return of all simulations that pass through the state.

Monte-Carlo tree search has been highly successful in a variety of domains. However, other approaches to simulation-based search are possible, and may be preferable in large domains. In particular, Monte-Carlo tree search represents all states individually, and cannot generalise between related states. We introduce a framework for generalisation in simulation-based search, by approximating the value function by a linear combination of features. In this approach, the outcome from a single position can be used to update the value function for a large number of similar states, leading to a much more efficient search.

Furthermore, Monte-Carlo methods must wait many time-steps until the final return is known. This return depends on all of the agent's decisions, and on the environment's uncertain responses to those decisions, throughout all of these time-steps. We introduce a lower variance, more efficient learning algorithm, by using temporal-difference learning to update the value function in simulation-based search.

6.5 Temporal-Difference Search

Algorithm 2 Linear TD Search

```
1:  θ ← 0                                              ▷ Initialise parameters
2:  procedure SEARCH(s_0)
3:      while time available do
4:          z ← 0                                      ▷ Clear eligibility trace
5:          s ← s_0
6:          a ← ε-greedy(s; Q)
7:          while s is not terminal do
8:              s' ~ P^a_{ss'}                         ▷ Sample state transition
9:              r ← R^a_{ss'}                          ▷ Sample reward
10:             a' ← ε-greedy(s'; Q)
11:             δ ← r + Q(s', a') − Q(s, a)            ▷ TD-error
12:             θ ← θ + αδz                            ▷ Update weights
13:             z ← λz + φ(s, a)                       ▷ Update eligibility trace
14:             s ← s', a ← a'
15:         end while
16:     end while
17:     return argmax_a Q(s_0, a)
18: end procedure
```

Temporal-difference search is a simulation-based search algorithm in which the value function is updated online, from simulated experience, by temporal-difference learning. Each search begins from a root state s_0. The agent simulates many episodes of experience from s_0, by sampling from its current policy $\pi_u(s, a)$, and from its current transition model A and reward model B, until each episode terminates.

The agent approximates the value function by using features $\phi(s, a)$ and adjustable parameters θ_u, using a linear combination $Q_u(s, a) = \phi(s, a).\theta_u$. After every step u of simulation, the agent updates the parameters by temporal-difference learning, using the TD(λ) algorithm. The first time a search is performed, the parameters are initialised to zero. For all subsequent searches, the parameter values are reused, so that the value function computed by the last search is used as the initial value function for the next search.

The agent selects actions by using an ϵ-greedy policy $\pi_u(s, a)$ that with probability $1 - \epsilon$ maximises the current value function $Q_u(s, a)$, or with probability ϵ selects a random action. As in the Sarsa algorithm, this interleaves policy evaluation with policy improvement, with the aim of finding the policy that maximises expected total reward from s_0, given the current model of the environment.

Temporal-difference search provides a spectrum of different behaviours. At one end of the spectrum, we can set $\lambda = 1$ to give Monte-Carlo search algorithms, or alternatively we can set $\lambda < 1$ to bootstrap from successive values. We can reproduce Monte-Carlo tree search by using table-lookup features $\phi(s, a) = e(s, a)$ (see Chapter 2)[3]; or we can generalise between states by using more abstract features.

6.6 Temporal-Difference Search in Computer Go

As we saw in Chapter 5, local shape features provide a simple but effective representation for intuitive Go knowledge. The value of each shape can be learnt offline, using temporal-difference learning and training by self-play, to provide general knowledge about the game of Go. However, the value function learned in this way is rather myopic: each square region of the board is evaluated independently, without any knowledge of the global context.

Local shape features can also be used during temporal-difference search. Although the features themselves are very simple, temporal-difference search takes account of the distribution of positions encountered during simulations. The value of each feature represents its contribution to future success, from this position onwards. Using local shape features, the value function is no longer myopically focused on each square region, but is highly tailored to the global context. This can significantly increase the representational power of local shape features: a shape may be bad in general, but good in the current situation (see Figure 6.3). By training from simulated experience, starting from the current state, the agent can focus on what works well *now*.

Local shape features provide a simple but powerful form of generalisation between similar positions. Unlike Monte-Carlo tree search, which evaluates each state independently, the value θ_i of a local shape ϕ_i is reused in a large class of related positions $\{s : \phi_i(s) = 1\}$ in which that particular shape occurs. This enables temporal-difference search to learn an effective value function from

[3]To exactly reproduce the behaviour of Monte-Carlo tree search we would incrementally add one new feature per simulation. The value of each state-action pair (s, a) should also be set to the mean return of all simulations in which action a was taken from state s. This can be achieved in Algorithm 2 by using a step-size schedule $\alpha(s, a) = 1/n(s, a)$, where $n(s, a)$ counts the number of times that action a has been taken from state s.

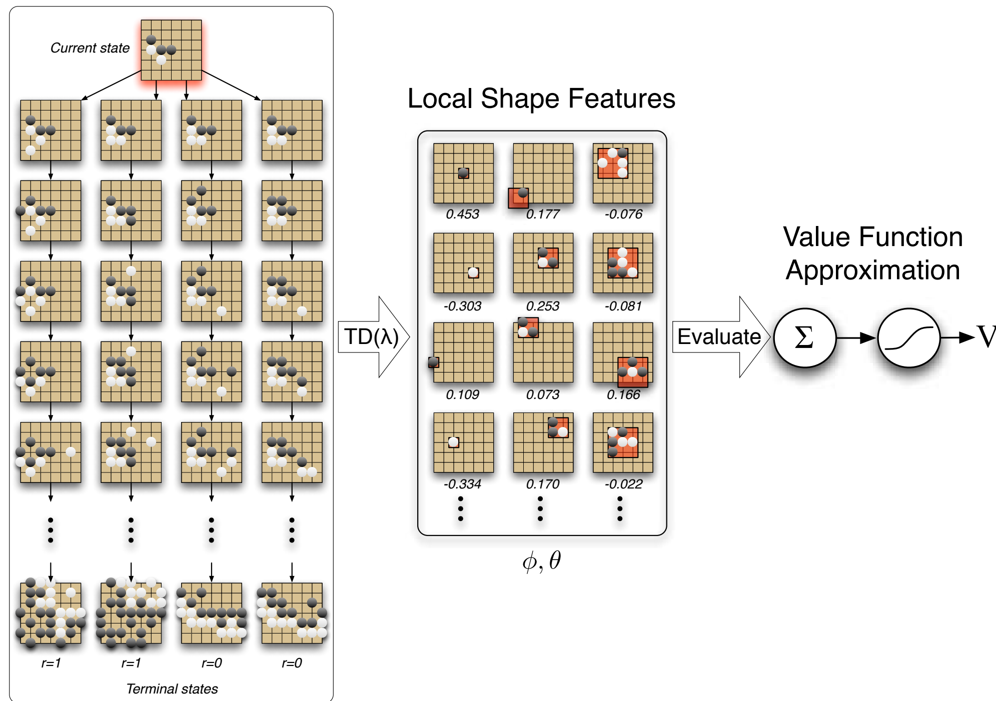

Figure 6.1: Temporal-difference search applied to Computer Go.

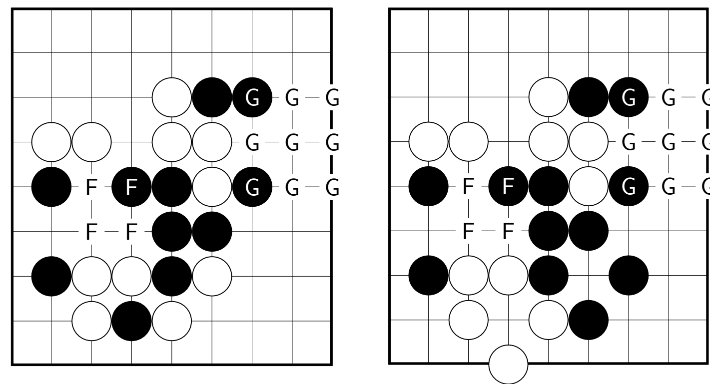

Figure 6.2: a) White threatens to cut blacks stones apart at F and G. 2×2 and 3×3 local shape features can represent the local continuations at F and G respectively. Temporal-difference search can evaluate the global effect of each local continuation, in the context of this specific position. b) A position encountered when searching from (a): Temporal-difference search is able to generalise, by using local shape features. It can re-use its knowledge about the value of each local continuation at F and G. In contrast, Monte-Carlo tree search evaluates each state independently, and must re-search from scratch.

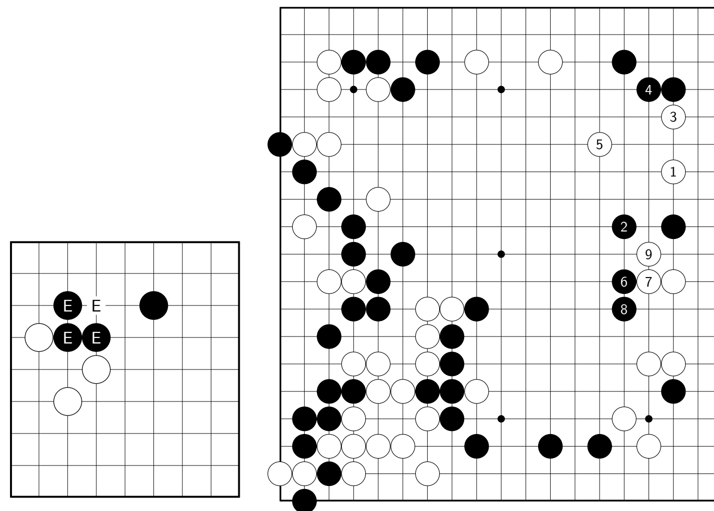

Figure 6.3: a) The *empty triangle* (E) is normally considered a bad shape; any other arrangement of three stones is usually more efficient. Using temporal-difference learning from self-play games, the empty triangle acquires a low value (see Chapter 5). However, in this particular position the empty triangle makes a good shape, known as *guzumi*. Using temporal-difference search, the empty triangle at E instead acquires a high value, as it is contributes to successful outcomes in games from this position. b) The *blood-vomiting game*, one of the most famous games from the history of Go. Honinbo Jowa played three miraculous moves, supposedly brought to him by his ghost. These moves enabled him to defeat his opponent, who ended the exhausting 4-day match by vomiting blood on the board, and died 2 months later. The third ghost move, 9, forms an empty triangle but begins a powerful splitting attack against black's stones. A less fanciful interpretation might be that, before attacking with 1, Honinbo Jowa imagined a variety of possible continuations, and observed that the empty triangle was a successful component of many of these continuations.

many fewer simulations than is possible with Monte-Carlo tree search. Figures 6.2 and 6.3 illustrate how local shape features can generalise between states during temporal-difference search.

In Chapter 5 we were able to exploit the symmetries of the Go board by using weight sharing. However, by starting our simulations from the current positions, we break these symmetries. The vast majority of Go positions are asymmetric, so that for example the value of playing in the top-left corner will be significantly different to playing in the bottom-right corner. Thus, we do not utilise any form of weight-sharing during temporal-difference search.

We apply the temporal-difference search algorithm to 9×9 Computer Go using 1×1 to 3×3 local shape features, excluding local shape features that consist entirely of empty intersections. We use a self-play model, an ϵ-greedy policy, and default parameters of $\lambda = 0$, $\alpha = 0.1$, and $\epsilon = 0.1$. We use a binary reward function at the end of the game: $r = 1$ if Black wins and $r = 0$ otherwise. We modify the basic temporal-difference search algorithm to exploit the probabilistic nature of the value function, by using logistic temporal-difference learning (see Appendix A). As in Chapter 5 we normalise the step-size by the total number of active features $|\phi(s)|$, and we use a two-ply temporal-difference update,

$$V(s) = \sigma(\phi(s).\theta) \tag{6.1}$$

$$\Delta\theta = \alpha \frac{\phi(s)}{|\phi(s)|}(V(s_{t+2}) - V(s_t)) \tag{6.2}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic function.

6.7 Experiments in 9×9 Go

We implemented the temporal-difference search algorithm in our Go program *RLGO v2.0*. We ran a tournament between different versions of *RLGO*, using different numbers of simulations per move, and for a variety of different parameter settings. In addition, we included two benchmark programs in each tournament. First, we included *GnuGo 3.7.10* (level 10). Second, we used the UCT implementation from *Fuego 0.1* (Müller and Enzenberger, 2009)[4], using the handcrafted default policy based on the rules described in (Gelly et al., 2006), with parameters set to the best reported values (exploration constant $= 1$, first play urgency $= 1$). Each Swiss-style tournament[5] consisted of at least 200 games for each version of *RLGO*. After all matches were complete, the results were analysed by the *bayeselo* program to establish an Elo rating for every program. Following convention, *GnuGo* was assigned an anchor rating of 1800 Elo in all cases.

6.7.1 Default Policy

Monte-Carlo tree search uses an *incomplete* representation of the state: only some subset of states are represented in the search tree. In our temporal-difference search algorithm, local shape features

[4]Here we compare against the vanilla UCT implementation in *Fuego*, with the RAVE and prior knowledge extensions turned off. These extensions will be developed and discussed further in Chapter 9

[5]Matches were randomly selected with a bias towards programs with a similar number of wins.

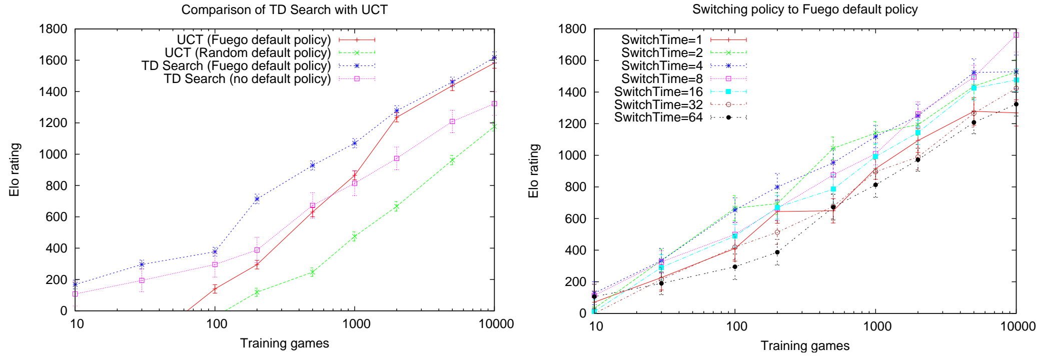

Figure 6.4: Comparison of temporal-difference search and UCT (left). Performance of temporal-difference search when switching to the *Fuego* default policy (right). The number of moves at which the switch occurred was varied between 1 and 64.

provide a *complete* representation of the state: every state is represented by a non-zero feature vector. One consequence of this completeness is that simulations never encounter states beyond the represented knowledge base, and so there is in principle no need for a default policy, or for a separate second stage of simulation. However, in practice a default policy can provide advantages in terms of computational or learning efficiency.

In temporal-difference search, an ϵ-greedy policy is used throughout the simulations. However, this simulation policy requires positions to be evaluated for all candidate moves, which can be significantly more expensive than a fast, handcrafted default policy. We ran an experiment to determine the effect on performance of switching to a default policy after a constant number T of moves. We used the default policy from *Fuego* to provide a fair comparison with the UCT algorithm. The results are shown in Figure 6.4 (right).

Switching policy was consistently most beneficial after 2-8 moves, providing around a 300 Elo improvement over no switching. This suggests that the knowledge contained in the local shape features is most effective when applied close to the root, and that the handcrafted default policy is more effective in positions far from the root.

We also compared the performance of temporal-difference search against the UCT implementation in *Fuego*. We considered two variants of each program, with and without a handcrafted default policy. The same default policy from *Fuego* was used in both programs. When using the default policy, the temporal-difference search algorithm switched to the *Fuego* default policy after $T = 6$ moves. When not using the default policy, the epsilon-greedy policy was used throughout all simulations. The results are shown in Figure 6.4.

The basic temporal-difference search algorithm, which utilises minimal domain knowledge based only on the grid structure of the board, significantly outperformed UCT with a random default policy. When using the *Fuego* default policy, temporal-difference search again outperformed the UCT algorithm, although the difference was not significant beyond 2000 simulations per move.

In our subsequent experiments, we switched to the *Fuego* default policy after $T = 6$ moves. This

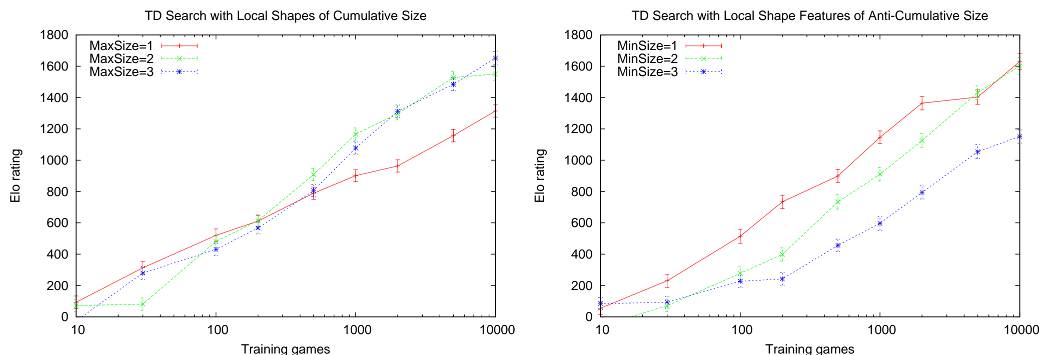

Figure 6.5: Performance of temporal-difference search with different sizes of local shape feature. Learning curves for cumulative sizes of local shape feature: 1×1; 1×1 and 2×2; or 1×1, 2×2 and 3×3 (left). Learning curves for anti-cumulative sizes of local shape feature: 1×1, 2×2 and 3×3; 2×2 and 3×3; or 3×3 (right).

increased the speed of our program by an order of magnitude, from around 200 simulations/move to 2000 simulations/move on a 2.4 GHz processor. For comparison, the UCT implementation in *Fuego* executed around 6000 simulations/move.

6.7.2 Local Shape Features

The local shape features that we use in our experiments would normally be considered quite naive in the domain of Go: the majority of shapes and tactics described in Go textbooks span considerably larger regions of the board than 3×3 squares. Indeed, when used in a traditional reinforcement learning context, the local shape features achieved a rating of around 1200 Elo (see Chapter 5). However, when the same representation is used in temporal-difference search, combining the 1×1 and 2×2 local shape features achieved a rating of almost 1700 Elo with just 10,000 simulations per move, more than UCT with an equivalent number of simulations (Figure 6.5). These simple features, based on neighbouring points, help to generalise on the basis of local moves and responses: for example, quickly learning the importance of White connecting when Black threatens to cut (Figures 6.2a and 6.2b).

The importance of temporality is aptly demonstrated by the 1×1 features. A static evaluation function based only on these features achieved a rating of just 200 Elo (see Chapter 5). However, when the feature weights are adapted dynamically, these simple features are often sufficient to identify the critical moves in the current position. Temporal-difference search increased the performance of the 1×1 features by 1000 Elo, and achieved a similar level of performance to a static evaluation function of one million features.

Surprisingly, including the more detailed 3×3 features provided no statistically significant improvement. However, we recall from Figure 5.6, when using the standard paradigm of temporal-difference learning, that there was an initial period of rapid 2×2 learning, followed by a slower period of learning the 3×3 shapes. Furthermore we recall that, without weight sharing, this transition

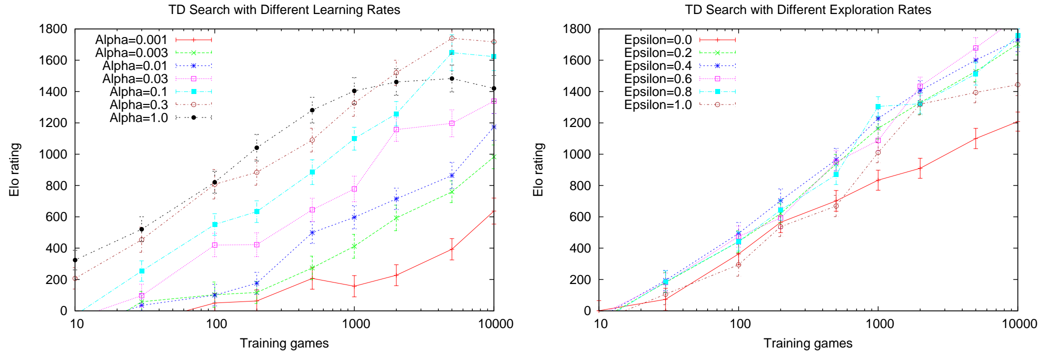

Figure 6.6: Performance of temporal-difference search with different learning rates α (left) and exploration rates ϵ (right).

took place after many thousands of simulations. This suggests that our temporal-difference search results correspond to the steep region of the learning curve. With sufficient simulations, we expect that the 3×3 shapes will provide a significant advantage, but the rate of improvement will flatten out.

6.7.3 Parameter Study

In our next experiment we varied the step-size parameter α (Figure 6.6). The results clearly show that an aggressive learning rate is most effective across a wide range of simulations per move, but the rating improvement for the most aggressive learning rates flattened out with additional computation. The rating improvement for $\alpha = 1$ flattened out after 1000 simulations per move, while the rating improvement for $\alpha = 0.1$ and $\alpha = 0.3$ appeared to flatten out after 5000 simulations per move.

We also evaluated the effect of the exploration rate ϵ. As in logistic temporal-difference learning (Figure 5.8), the algorithm performed poorly with either no exploratory moves ($\epsilon = 0$), or with only exploratory moves ($\epsilon = 1$). The difference between intermediate values of ϵ was not statistically significant.

6.7.4 TD(λ) Search

We extend the temporal-difference search algorithm to utilise eligibility traces, using the accumulating and replacing traces from Chapter 5. We study the effect of the temporal-difference parameter λ in Figure 6.7. With accumulating traces, bootstrapping ($\lambda < 1$) provided a significant performance benefit. With replacing traces, $\lambda = 1$ performed well for the first 2000 simulations per move, but bootstrapping performed much better with more simulations per move.

Previous work in simulation-based search has largely been restricted to Monte-Carlo methods (Tesauro and Galperin, 1996; Kocsis and Szepesvari, 2006; Gelly et al., 2006; Gelly and Silver, 2007; Coulom, 2007). Our results suggest that generalising these approaches to temporal- difference learning methods may provide significant benefits when value function approximation is used.

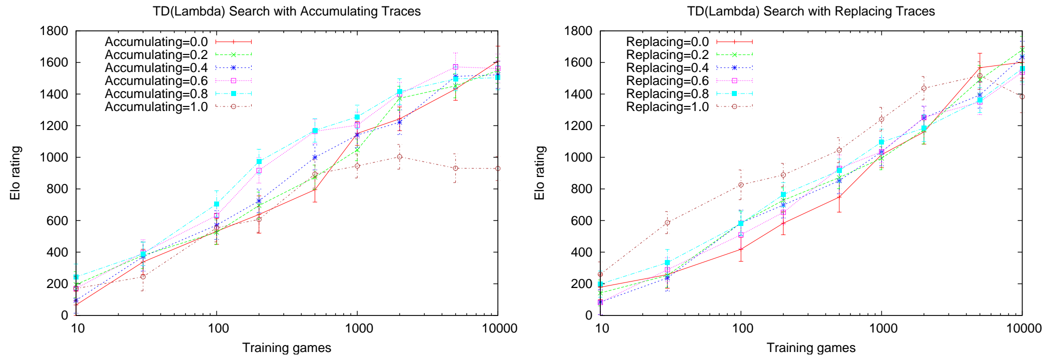

Figure 6.7: Performance of TD(λ) search for different values of λ, using accumulating traces (left) and replacing traces (right).

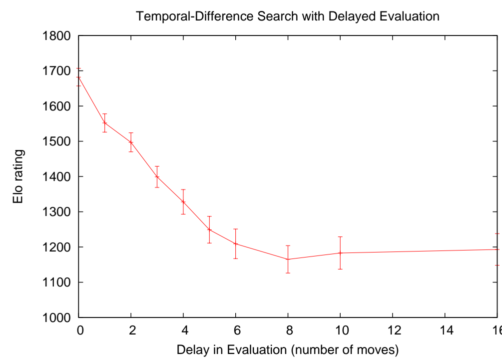

Figure 6.8: Performance of temporal-difference search with 10000 simulations/move, when the results of the latest search are only used after some additional number of moves have elapsed.

6.7.5 Temporality

Successive positions are particularly coherent in the game of Go. Each position changes incrementally, by just one new stone at every non-capturing move. Groups and fights develop, providing specific shapes and tactics that may persist for a significant proportion of the game, but are unique to this game and are unlikely to ever be repeated. We conducted two experiments to disrupt this temporal coherence, so as to gain some insight into its effect on temporal-difference search.

In our first experiment, we selected moves according to an old value function from a previous search. At move number t, the agent selects the move that maximises the value function that it computed at move number $t - k$, for some move gap $0 \leq k < t$. The results, shown in Figure 6.8, indicate the rate at which the global context changes. The value function computed by the search is highly specialised to the current situation. When it was applied to the position that arose just 6 moves later, the performance of *RLGO*, using 10000 simulations per move, dropped to 1200 Elo, the same level of performance that was achieved by global temporal-difference learning.

In our second experiment, we reset the parameters θ to zero at the beginning of every search, so as to disrupt any transfer of knowledge between successive moves. With 2000 simulations per move,

this reduced the performance of our program by around 400 Elo. This suggests that an important aspect of temporal-difference search is its ability to accumulate knowledge over many successive, highly related positions.

6.8 Conclusion

Reinforcement learning is often considered a slow procedure. Outstanding examples of success have, in the past, learned a value function from months of offline computation. However, this does not need to be the case. Many reinforcement learning methods, such as Monte-Carlo learning and temporal-difference learning, are fast, incremental, and scalable. When such a reinforcement learning algorithm is applied to experience simulated from the current state, it produces a high performance search algorithm.

Monte-Carlo search algorithms, such as UCT, have recently received much attention. However, this is just one example of a simulation-based search algorithm. There is a spectrum of algorithms that vary from table-lookup to highly abstracted state representations, and from Monte-Carlo learning to temporal-difference learning. Value function approximation provides rapid generalisation in large domains, and bootstrapping is advantageous in the presence of function approximation. By varying these dimensions in the temporal-difference search algorithm, we have achieved better search efficiency per simulation than UCT.

In addition, temporal-difference search with a complete state representation offers two potential advantages over Monte-Carlo tree search. First, simulations never exit the agent's knowledge-base: all positions can be evaluated, so that there is no requirement for a distinct default policy. Second, search knowledge from previous time-steps can be reused much more effectively, simply by using the previous value function to initialise the new search. Thus temporal-difference search can generalise both from past positions and from future positions during simulation.

The UCT algorithm retains several advantages over temporal-difference search. It is faster, simpler, and given unlimited time and memory algorithms it will converge on the optimal policy. In many ways our initial implementation of temporal-difference search is more naive: it uses straightforward features, a simplistic epsilon-greedy exploration strategy, a non-adaptive step-size, and a constant policy switching time. The promising results of this basic strategy suggest that the full spectrum of simulation-based methods, not just Monte-Carlo and table-lookup, merit further investigation.

Chapter 7

Long and Short-Term Memories

7.1 Introduction

In many problems, *learning* and *search* must be combined together in order to achieve good performance. Learning algorithms extract knowledge, from the complete history of training data, that applies very generally throughout the domain. Search algorithms both use and extend this knowledge, so as to evaluate local states more accurately. Learning and search often interact in a complex and surprising fashion, and the most successful approaches integrate both processes together (Schaeffer, 2000; Fürnkranz, 2001).

In Computer Go, the most successful learning methods have used reinforcement learning algorithms to extract domain knowledge from games of self-play (Schraudolph et al., 1994; Enzenberger, 1996; Dahl, 1999; Enzenberger, 2003; Silver et al., 2007). The value of a position is approximated by a multi-layer perceptron, or a linear combination of binary features, that forms a compact representation of the state space. Temporal-difference learning is used to update the value function, slowly accumulating knowledge from the complete history of experience.

The most successful search methods are simulation based, for example using the Monte-Carlo tree search algorithm (see Chapter 4). This algorithm begins each new move without any domain knowledge, but rapidly learns the values of positions in a temporary search tree. Each state in the tree is explicitly represented, and the value of each state is learned by Monte-Carlo simulation, from games of self-play that start from the current position.

In this chapter we develop a unified architecture, *Dyna-2*, that combines both reinforcement learning and simulation-based search. Like the Dyna architecture (Sutton, 1990), the agent updates a value function both from real experience, and from simulated experience that is sampled from a model of the environment. The new idea is to maintain two separate memories: a long-term memory that is learned from real experience; and a short-term memory that is used during search, and is updated from simulated experience. Both memories use linear function approximation to form a compact representation of the state space, and both memories are updated by temporal-difference learning.

7.2 Long and Short-Term Memories

Domain knowledge contains many general rules, but even more special cases. A grandmaster Chess player once said, "I spent the first half of my career learning the principles for playing strong Chess and the second half learning when to violate them" (Schaeffer, 1997). Long and short-term memories can be used to represent both aspects of knowledge.

We define a *memory* $M = (\phi, \theta)$ to be a vector of features ϕ, and a vector of corresponding parameters θ. The feature vector $\phi(s, a)$ compactly represents the state s and action a, and provides an abstraction of the state and action space. The parameter vector θ is used to approximate the value function, by forming a linear combination $\phi(s, a).\theta$ of the features and parameters in M.

In our architecture, the agent maintains two distinct memories: a *long-term memory* $M = (\phi, \theta)$ and a *short-term memory* $\overline{M} = (\overline{\phi}, \overline{\theta})$[1]. The agent also maintains two distinct approximations to the value function. The *long-term value function*, $Q(s, a)$, uses only the long-term memory to approximate the true value function $Q^\pi(s, a)$. The *short-term value function*, $\overline{Q}(s, a)$, uses *both* memories to approximate the true value function, by forming a linear combination of both feature vectors with both parameter vectors,

$$Q(s, a) = \phi(s, a).\theta \tag{7.1}$$

$$\overline{Q}(s, a) = \phi(s, a).\theta + \overline{\phi}(s, a).\overline{\theta} \tag{7.2}$$

The long-term memory is used to represent *general* knowledge about the domain, i.e. knowledge that is independent of the agent's current state. For example, in Chess the long-term memory could know that a Bishop is worth 3.5 pawns. The short-term memory is used to represent *local* knowledge about the domain, i.e. knowledge that is specific to the agent's current region of the state space. The short-term memory is used to *correct* the long-term value function, representing adjustments that provide a more accurate local approximation to the true value function. For example, the short-term memory could know that, in the current closed position, the black Bishop is worth 1 pawn less than usual. These corrections may actually hurt the global approximation to the value function, but if the agent continually adjusts its short-term memory to match its current state, then the overall quality of approximation can be significantly improved.

7.3 Dyna-2

The core idea of Dyna-2 is to combine temporal-difference learning with temporal-difference search, using long and short-term memories. The long-term memory is updated from real experience, and the short-term memory is updated from simulated experience, in both cases using the TD(λ) algorithm. We denote short-term parameters with a bar $\overline{x}$, and long-term parameters with no bar x.

[1]These names are suggestive of each memory's function, but should not be taken to imply a literal interpretation of the biological long and short-term memory systems.

Algorithm 3 Episodic Dyna-2

```
1:  procedure LEARN
2:      Initialise P^a_{ss'}, R^a_{ss'}                    ▷ State transition and reward models
3:      θ ← 0                                              ▷ Clear long-term memory
4:      loop
5:          s ← s_0                                        ▷ Start new episode
6:          θ̄ ← 0                                          ▷ Clear short-term memory
7:          z ← 0                                          ▷ Clear eligibility trace
8:          SEARCH(s)
9:          a ← ε-greedy(s; Q̄)
10:         while s is not terminal do
11:             Execute a, observe reward r, state s'
12:             P^a_{ss'}, R^a_{ss'} ← UPDATEMODEL(s, a, r, s')
13:             SEARCH(s')
14:             a' ← ε-greedy(s'; Q̄)
15:             δ ← r + Q(s', a') − Q(s, a)                ▷ TD-error
16:             θ ← θ + αδz                                ▷ Update weights
17:             z ← λz + φ                                 ▷ Update eligibility trace
18:             s ← s', a ← a'
19:         end while
20:     end loop
21: end procedure

22: procedure SEARCH(s)
23:     while time available do
24:         z̄ ← 0                                          ▷ Clear eligibility trace
25:         a ← ε̄-greedy(s; Q̄)
26:         while s is not terminal do
27:             s' ∼ P^a_{ss'}                             ▷ Sample state transition
28:             r ← R^a_{ss'}                              ▷ Sample reward
29:             a' ← ε̄-greedy(s'; Q̄)
30:             δ̄ ← r + Q̄(s', a') − Q̄(s, a)                ▷ TD-error
31:             θ̄ ← θ̄ + ᾱδ̄z̄                                ▷ Update weights
32:             z̄ ← λ̄z̄ + φ̄                                 ▷ Update eligibility trace
33:             s ← s', a ← a'
34:         end while
35:     end while
36: end procedure
```

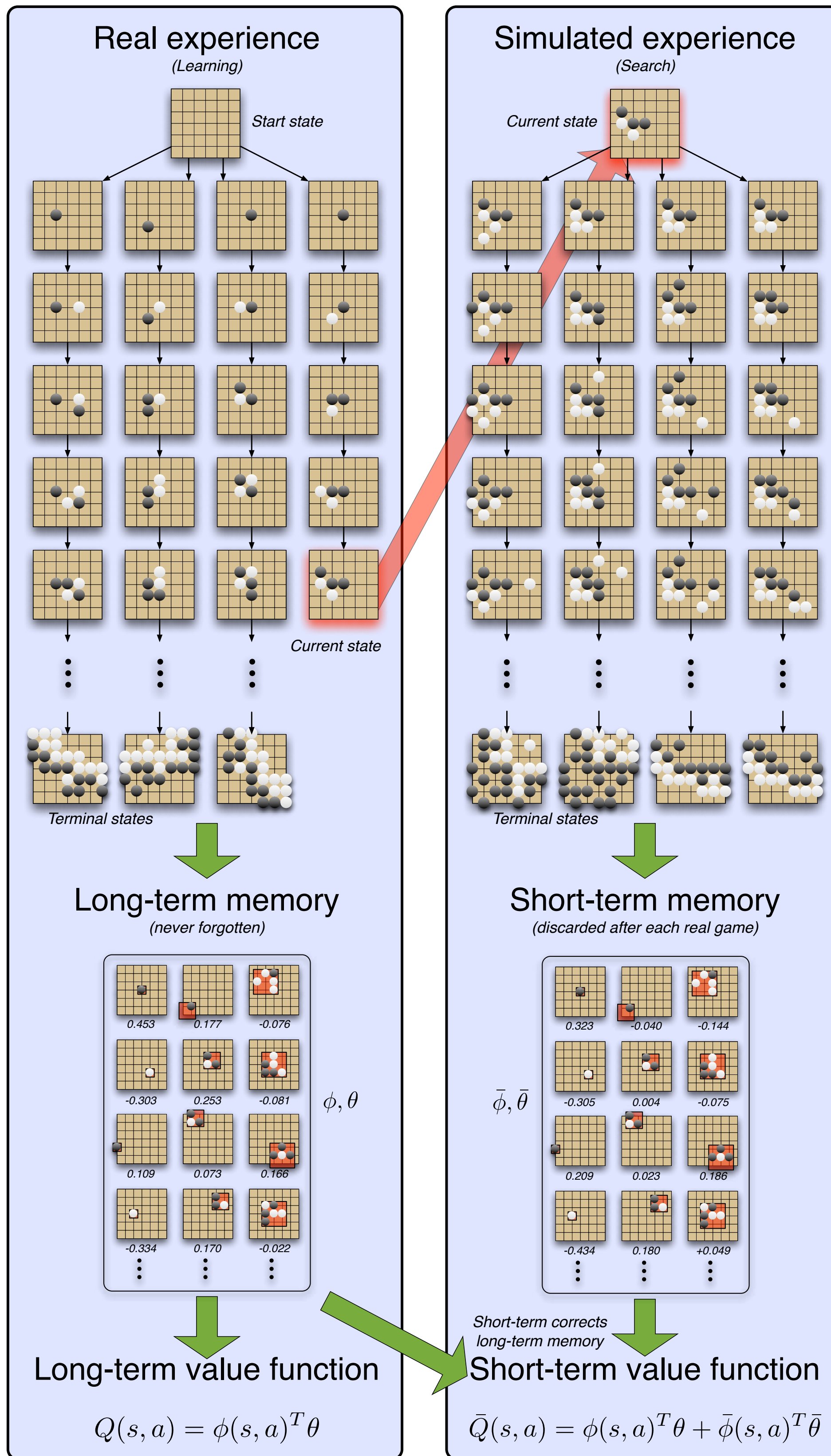

Figure 7.1: The Dyna-2 architecture

At the beginning of each real episode, the contents of the short-term memory are cleared, $\overline{\theta} = 0$. At each real time-step t, before selecting its action a_t, the agent executes a simulation-based search. Many simulations are launched, each starting from the agent's current state s_t. After each step of computation u, the agent updates the weights of its short-term memory from its simulated experience $(s_u, a_u, r_{u+1}, s_{u+1}, a_{u+1})$, using the TD($\lambda$) algorithm. The TD-error is computed from the short-term value function, $\overline{\delta}_u = r_{u+1} + \overline{Q}(s_{u+1}, a_{u+1}) - \overline{Q}(s_u, a_u)$. Actions are selected using an $\overline{\epsilon}$-greedy policy that maximises the short-term value function $a_u = \underset{b}{\operatorname{argmax}}\ \overline{Q}(s_u, b)$. This search procedure continues for as much computation time as is available.

When the search is complete, the short-term value function represents the agent's best local approximation to the optimal value function. The agent then selects a real action a_t using an ϵ-greedy policy that maximises the short-term value function $a_t = \underset{b}{\operatorname{argmax}}\ \overline{Q}(s_t, b)$. After each time-step, the agent updates its long-term value function from its real experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, again using the TD($\lambda$) algorithm. This time, the TD-error is computed from the long-term value function, $\delta_t = r_{t+1} + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$. In addition, the agent uses its real experience to update its state transition model A_t and its reward model B_t. The complete algorithm is described in Algorithm 3.

The Dyna-2 architecture learns from both the past and the future. The long-term memory is updated from the agent's actual past experience. The short-term memory is updated from sample episodes of what could happen in the future. Combining both memories together provides a much richer representation than is possible with a single memory.

A particular instance of Dyna-2 must specify learning parameters: a set of features ϕ for the long-term memory; a temporal-difference parameter λ; an exploration rate ϵ and a learning rate α. Similarly, it must specify the equivalent search parameters: a set of features $\overline{\phi}$ for the short-term memory; a temporal-difference parameter $\overline{\lambda}$; an exploration rate $\overline{\epsilon}$ and a learning rate $\overline{\alpha}$.

The Dyna-2 architecture subsumes a large family of learning and search algorithms. If there is no short-term memory, $\phi = \emptyset$, then the search procedure has no effect and may be skipped. This results in the linear Sarsa algorithm (see Chapter 2). If there is no long-term memory, $\overline{\phi} = \emptyset$, then Dyna-2 reduces to the temporal-difference search algorithm. As we saw in Chapter 6, this algorithm itself subsumes a variety of simulation-based search algorithms such as Monte-Carlo tree search.

Finally, we note that real experience may be accumulated offline prior to execution. Dyna-2 may be executed on any suitable training environment (e.g. a helicopter simulator) before it is applied to real data (e.g. a real helicopter). The agent's long-term memory is learned offline during a preliminary training phase. When the agent is placed into the real environment, it uses its short-term memory to adjust to the current state. Even if the agent's model is inaccurate, each simulation begins from its true current state, which means that the simulations are usually fairly accurate for at least the first few steps. This allows the agent to dynamically correct at least some of the misconceptions in the long-term memory.

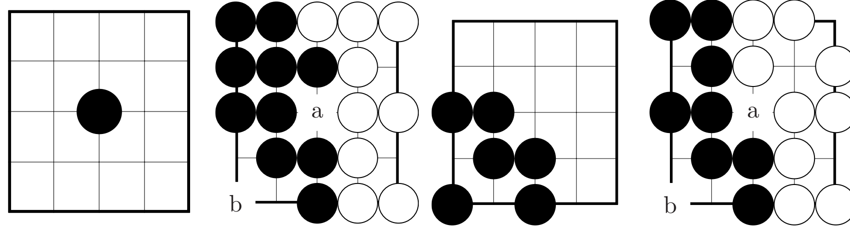

Figure 7.2: a) A 1×1 local shape feature with a central black stone. This feature acquires a strong positive value in the long-term memory. b) In this position, move b is the winning move. Using only 1×1 local shape features, the long-term memory suggests that move a should be played. The short-term memory will quickly learn to correct this misevaluation, reducing the value of a and increasing the value of b. c) A 3×3 local shape feature making two eyes in the corner. This feature acquires a positive value in the long-term memory. d) Black to play, using Chinese rules, move a is now the winning move. Using 3×3 features, the long-term memory suggests move b, believing this to be a good shape in general. However, the short-term memory quickly realises that move b is redundant in this context (black already has two eyes) and learns to play the winning move at a.

7.4 Dyna-2 in Computer Go

In general, it may be desirable for the long and short-term memories to utilise different features, that are best suited to representing either general or local knowledge. In our Computer Go experiments, we focus our attention on the simpler case where both vectors of features are identical[2], $\phi = \overline{\phi}$.

We have already seen that local shape features can be used with temporal-difference learning, to learn general Go knowledge (see Chapter 5). We have also seen that local shape features can be used with temporal-difference search, to learn the value of shapes in the current situation (see Chapter 6). The Dyna-2 architecture lets us combine the advantages of both approaches, by using local shape features in both the long and short-term memories.

Figure 7.2 gives a very simple illustration of long and short-term memories in 5×5 Go. It is usually bad for Black to play on the corner intersection, and so long-term memory learns a negative weight for this feature. However, Figure 7.2 shows a position in which the corner intersection is the most important point on the board for Black: it makes two eyes and allows the Black stones to live. By learning about the particular distribution of states arising from this position, the short-term memory learns a large positive weight for the corner feature, correcting the long-term memory.

We apply Dyna-2 to 9×9 Computer Go using 1×1 to 3×3 local shape features. We use a self-play model, and default parameters of $\lambda = \overline{\lambda} = 0$, $\alpha = 0.1/|\phi(s,a)|$, $\overline{\alpha} = 0.1/|\overline{\phi}(s,a)|$, $\epsilon = 0$, and $\overline{\epsilon} = 0.1$. We modify the Dyna-2 algorithm slightly to utilise logistic temporal-difference

[2]We note that in this special case, the Dyna-2 algorithm can be implemented somewhat more efficiently, using just one memory during search. At the start of each real game, the contents of the short-term memory are initialised to the contents of the long-term memory, $\overline{\theta} = \theta$. Subsequent searches can then proceed using only the short-term memory, just as in temporal-difference search (see Algorithm 2).

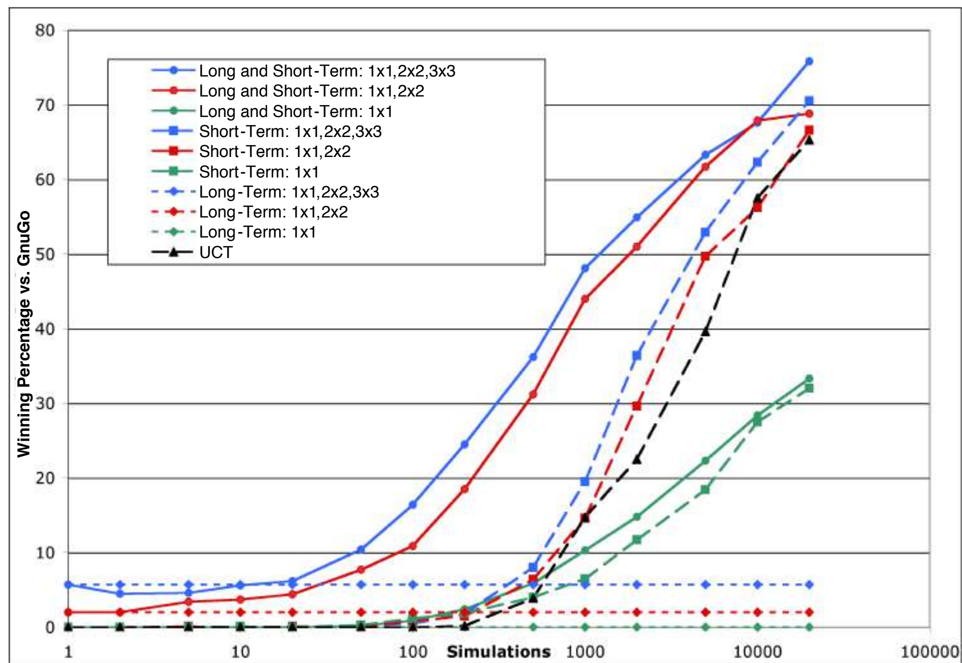

Figure 7.3: Winning rate of RLGO against GnuGo 3.7.10 (level 0) in 9×9 Go, using Dyna-2-Shape for different simulations/move. Local shape features are used in either the long-term memory (dotted lines), the short-term memory (dashed lines), or both memories (solid lines). The long-term memory was trained offline from 100,000 games of self-play. Local shape features varied in size from 1×1 up to 3×3. Each point represents the winning percentage over 1000 games (less than 1 second per move).

Search algorithm	Memory	Elo rating on CGOS
Alpha-beta	Long-term	1350
Dyna-2	Long and short-term	2030
Dyna-2 + alpha-beta	Long and short-term	2130

Table 7.1: The Elo rating established by RLGO on the Computer Go Server.

learning (see Appendix A), by replacing the value function approximation in (7.1) and (7.2),

$$Q(s,a) = \sigma(\phi(s,a).\theta) \tag{7.3}$$

$$\overline{Q}(s,a) = \sigma(\phi(s,a).\theta + \overline{\phi}(s \circ a).\overline{\theta}) \tag{7.4}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic function.

In addition we ignore local shape features consisting of entirely empty intersections, and we use the *Fuego* default policy after the first $T = 10$ moves of each simulation. We use weight sharing to exploit symmetries in the long-term memory, but we do not use any weight sharing in the short-term memory. We refer to the complete algorithm as *Dyna-2-Shape*, and implement this algorithm in our program *RLGO*, which executes almost 2000 complete episodes of simulation per second on a 3 GHz processor.

We compare our algorithm to the UCT algorithm, using the *Fuego* 0.1 program (Müller and Enzenberger, 2009). We use an identical default policy to the Dyna-2-Shape algorithm, to select moves when outside of the search tree, and a first play urgency of 1. We evaluate both programs by running matches against GnuGo, a standard benchmark program for Computer Go.

We compare the performance of local shape features using only a long-term memory (i.e. temporal-difference learning); using only a short-term memory (i.e. temporal-difference search); and in both long and short-term memories. We also compare the performance of local shape features of different sizes (see Figure 7.3). Using only the short-term memory, Dyna-2-Shape outperformed UCT by a small margin. Using Dyna-2-Shape with both long and short-term memories provided the best results, and outperformed UCT by a significant margin.

7.5 Dyna-2 and Heuristic Search

In games such as Chess, Checkers and Othello, master-level play has been achieved by combining a heuristic evaluation function with α-β search. The heuristic is typically approximated by a linear combination of binary features, and can be learned offline by temporal-difference learning and self-play (Baxter et al., 1998; Schaeffer et al., 2001; Buro, 1999). In Chapter 5, we saw how this approach could be applied to Go, by using local shape features.

In this chapter we have developed a significantly more accurate approximation of the value function, by combining long and short-term memories, using both temporal-difference learning and temporal-difference search. Can this more accurate value function be successfully used in a traditional alpha-beta search?

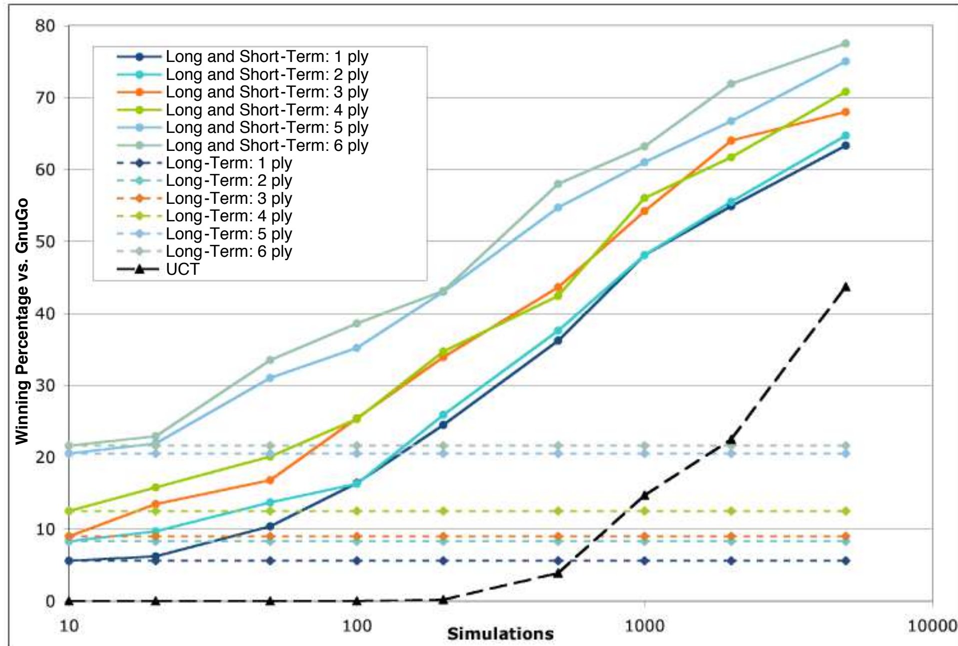

Figure 7.4: Winning rate of RLGO against GnuGo 3.7.10 (level 0) in 9×9 Go, using a hybrid search based on both Dyna-2 and alpha-beta. A full-width α-β search is used for move selection, using a value function based on either the long-term memory alone (dotted lines), or both long and short-term memories (solid lines). Using only the long-term memory corresponds to a traditional alpha-beta search. Using both memories, but only a 1-ply search, corresponds to the Dyna-2 algorithm. The long-term memory was trained offline from 100,000 games of self-play. Each point represents the winning percentage over 1000 games.

We describe this approach, in which a simulation-based search is followed by a traditional search, as a *hybrid search*. We extend the Dyna-2 algorithm into a hybrid search, by performing an alpha-beta search after each temporal-difference search. As in Dyna-2, after the simulation-based search is complete, the agent selects a real move to play. However, instead of directly maximising the short-term value function, an alpha-beta search is used to find the best move in the depth d minimax tree, where the leaves of the tree are evaluated according to the short-term value function $\overline{Q}(s, a)$.

The hybrid algorithm can also be viewed as an extension to alpha-beta search, in which the evaluation function is dynamically updated. At the beginning of the game, the evaluation function is set to the contents of the long-term memory. Before each alpha-beta search, the evaluation function is re-trained by a temporal-difference search. The alpha-beta search then proceeds as usual, but using the updated evaluation function.

We compared the performance of the hybrid search algorithm to a traditional search algorithm. In the traditional search, the long-term memory $Q(s, a)$ is used as a heuristic function to evaluate leaf positions, as in Chapter 5. The results are shown in Figure 7.3.

Dyna-2 outperformed traditional search by a wide margin. Using only 200 simulations per move, Dyna-2-Shape exceeded the performance of a full-width 6-ply search. For comparison, a 5-ply search took approximately the same computation time as 1000 simulations. When combined with alpha-beta in the hybrid search algorithm, the results were even better. Alpha-beta provided a substantial performance boost of around 15-20% against GnuGo, which remained approximately constant throughout the tested range of simulations per move. With 5000 simulations per move, the hybrid algorithm achieved a winning rate of almost 80% against GnuGo. These results suggest that the benefits of alpha-beta search are largely complementary to the simulation-based search.

Finally, we implemented a high-performance version of our hybrid search algorithm in *RLGO*. In this tournament version, time was dynamically allocated between the two search algorithms, according to the time controls. We extended the temporal-difference search to use multiple processors, by sharing the long and short-term memories between processes, and to use pondering, by simulating additional games of self-play during the opponent's thinking time. We extended the alpha-beta search to use several well-known extensions: iterative deepening, transposition table, killer move heuristic, and null-move pruning (Schaeffer, 2000). RLGO competed on the 9×9 Computer Go Server, which uses 5 minute time controls, for several hundred games in total. The ratings established by RLGO are shown in Table 7.1.

Using only an alpha-beta search, based on the long-term memory alone, RLGO established a rating of 1350 Elo. Using Dyna-2, using both long and short-term memories, but no alpha-beta search, RLGO established a rating of 2030 Elo. Using the hybrid search algorithm, including Dyna-2 and also an alpha-beta search, RLGO established a rating of 2130 Elo. For comparison, the highest rating achieved by any previous traditional search program was around 1850 Elo. Furthermore these

previous programs incorporated a great deal of sophisticated handcrafted knowledge about the game of Go, whereas the handcrafted Go knowledge in RLGO is minimal. If we view the hybrid search as an extension to alpha-beta, then we see that dynamically updating the evaluation function offers dramatic benefits, improving the performance of RLGO by 800 Elo. If we view the hybrid search as an extension to Dyna-2, then the performance improves by a more modest, but still significant 100 Elo.

7.6 Conclusion

Learning algorithms update the agent's knowledge about the environment, from its real experience with the environment. Search algorithms dynamically update the agent's knowledge of its local sub-problem, using a model of the environment. The Dyna-2 algorithm provides a principled approach to learning and search that effectively combines both forms of knowledge.

In the game of Go, the consequences of a particular move or shape may not become apparent for tens or even hundreds of moves. In a traditional, limited depth search these consequences remain beyond the horizon, and will only be recognised if explicitly represented by the evaluation function. In contrast, Dyna-2 only uses the long-term memory as an initial guide, and learns to identify the consequences of particular patterns in its short-term memory. However, it lacks the precise global lookahead required to navigate the full-board fights that can often engulf a 9×9 board. The hybrid search successfully combines the deep knowledge of Dyna-2 with the precise lookahead of a full-width search, outperforming all handcrafted, traditional search, and traditional machine learning approaches.

Part III

Monte-Carlo Tree Search

Chapter 8

Reusing Local Search Trees in Simulation-Based Search

8.1 Introduction

Prior research on Monte-Carlo tree search has focused on developing the search tree more efficiently: balancing exploration and exploitation (Kocsis and Szepesvari, 2006), incorporating prior knowledge (Gelly and Silver, 2007), pruning the search tree (Gelly et al., 2006) or progressively widening the search space (Chaslot et al., 2007). In this chapter we focus instead on the issue of *representation*: can the state space be represented more efficiently than by a single, monolothic search tree?

A typical Go position contains many local fights, each with its own particular tactics and strategies. However, these fights usually overlap, and cannot be exactly decomposed into independent subgames (Müller, 1995). Traditional Go programs construct local search trees for each separate fight, which can then be reused in many related positions. In contrast, a monolithic global search, such as Monte-Carlo tree search, must duplicate the work of these local searches exponentially many times.

We introduce a new approach to Monte-Carlo tree search, which forms multiple, overlapping search trees, each of which is local to a specific region of the state space. The global position is evaluated by a linear combination of the local search states, with values learned online by temporal-difference search. Like traditional Go programs, this efficiently reuses the work of each local state; but like Monte-Carlo tree search, evaluation is grounded in simulations, and continues to improve with additional computation.

8.2 Local and Global Search in Computer Go

We motivate our approach with a typical position from a game of 9×9 Go (Figure 8.1), in which several tactical battles are being fought concurrently, in different regions of the board. One approach to evaluating this position, used by the majority of commercial and traditional Go programs, is to

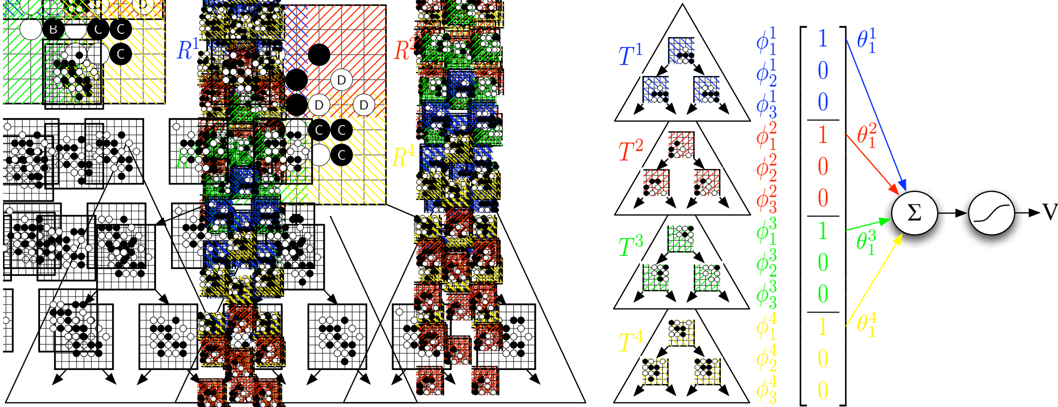

Figure 8.1: (Top-left) A 9×9 Go position (black to move), containing concurrent, overlapping local tactics. White A can be captured, but still has *aji*: the potential to cause trouble. Black B is caught in a ladder, but local continuations will influence black C, which is struggling to survive in the bottom-right. White D is attempting to survive in the top-right, in an overlapping battle with black C. (Bottom-left) A global search tree that includes several subtrees containing the same moves in R^1. After each move in regions R^2, R^3 and R^4, the region R^1 is re-searched, resulting in an exponential number of duplicated subtrees. (Right) The position is broken down into local search trees for the overlapping regions R^1, R^2, R^3 and R^4. The global value is estimated by summing the value of each local state, and squashing into $[0, 1]$ by a logistic function.

evaluate each region independently (Müller, 2002, 2001). A number of local searches are performed, each directed towards a binary subgoal such as the life or death, connection, or eye status of each block of stones. This approach suffers from several drawbacks. First, exact computation of each local search is computationally infeasible for the vast majority of non-trivial positions. Second, the local searches are interdependent: local battles frequently spill out across the board, often affecting the outcome of many other searches. Third, these simple subgoals are inadequate to capture the complex global effect of each local state, for example the influence it exerts on the rest of the board, or its *aji* (potential for troublemaking later in the game).

This position can also be evaluated by Monte-Carlo tree search (MCTS). A global search tree is constructed, with each position in the tree evaluated by its mean outcome, over many thousands of games of self-play. This addresses, at least in principle, all three of the above problems: approximate evaluation is possible for any position in the search tree, even in tactically complex positions; the global search can represent and evaluate the complex interactions between regions; and global concepts such as aji (Figure 8.1, region A) and influence (Figure 8.1, region B) are captured by the outcomes of the simulations. Go programs based on MCTS have significantly outperformed traditional search programs (Gelly and Silver, 2007, 2008; Coulom, 2007).

Unfortunately, the global nature of MCTS can also make it highly inefficient. In order to solve the tactics associated with capturing white A, it must search a subtree that contains global states for all possible continuations within the top-left region R^1. However, if a single move is played elsewhere, then the global state is changed, and a new subtree must be constructed for R^1. Every

time a move is played elsewhere, R^1 must be re-searched, leading to an exponential number of subtrees within the global search tree, each duplicating the work of solving the fight in a different global context (Figure 8.1, bottom-left). This exponential redundancy is repeated for all local regions, constructing approximately $k(bq)^d$ duplicate subtrees, where d is the global search depth, k is the number of local regions, b is the global branching factor, and q is the proportion of moves that are irrelevant to the outcome of each local search.

We extend the idea of Monte-Carlo search, so as to efficiently reuse local search-trees. Instead of representing the position by a single, global state, the new idea is to represent the position by several local search states. A separate value is associated with each local state, indicating its average contribution to winning or losing the game. These values are learned online from simulated games of self-play, and summed together to evaluate the global state. Thus, rather than explicitly seeking to solve a specific subgoal, each local search tree is used to provide an informative set of positional features for global evaluation.

This approach retains the key advantages of Monte-Carlo search, while addressing all three problems of traditional local search algorithms. First, approximate global evaluation is possible for a broad class of positions, whenever one or more regions match a local state from one of the search trees. Second, local regions can overlap without adverse consequences. For example, when a local state in the region R^4 is evaluated from simulated experience, it averages over the distribution of local states encountered in region R^2 (and elsewhere). Third, the global effect of each local state is explicitly represented by its value. The influence of certain local states in region R^3 leads to more black wins during simulation; the aji in region R^1 leads to more white wins during simulation. In both cases, the value of the local state, learned from the outcomes of simulations, captures its long-term global consequences.

Unlike MCTS, this approach is able to generalise between related positions. Once the value of a local state has been determined (for example, capturing the black stone in R^1), it contributes to the evaluation of all global positions in which that local state occurs. This ability to generalise also means that exponentially more states can be evaluated, enabling the highly informed first phase of simulation to last significantly longer, and therefore increasing the effective search depth.

8.3 Local Monte-Carlo Search

Local Monte-Carlo search is a particular form of simulation-based search (see Chapter 6) in which the features are defined by local states in a local search tree.

8.3.1 State representation

In many problems, the global state space can be completely described by a set of *state variables*, for example the set of intersections in the game of Go. A *region* is a subset of these state variables, for example a set of neighbouring intersections in Go. A *local state* is a complete configuration of

the state variables within a region, for example a local pattern of black, white and empty points in Go. A region can be viewed as an *abstraction* of the state space, which takes account of some state variables, and ignores all others. This form of state abstraction has proven particularly effective in traditional search algorithms (Culberson and Schaeffer, 1996).

The key idea of our algorithm is to represent state by a composition of many different abstractions. We represent a state s by a large feature vector $\phi(s)$ that consists of many local states for many different regions (see Figure 8.1, right). Each component of the vector is a binary feature, $\phi^i_j(s)$, that matches the jth local state of the ith region R^i. Specifically, if the values of all state variables in this local state match the corresponding values in s, then $\phi^i_j(s) = 1$, otherwise $\phi^i_j(s) = 0$. The subvector $\phi^i(s)$ consists of binary features for many different local states in region R^i. The overall feature vector is composed of the subvectors for all regions,

$$\phi(s) = \begin{bmatrix} \phi^{(1)}(s) \\ . \\ . \\ . \\ \phi^{(k)}(s) \end{bmatrix} \tag{8.1}$$

We assume that the set of regions is given. The local states in each region R^i are then chosen by incrementally developing a local search tree T^i during the course of the simulations. At the beginning of the game, all local search trees are cleared, $T^i = \emptyset$. Every simulation, for every region R^i, the first local state that is visited which is not already in the local search tree T^i, is added into that search tree. The feature subvector ϕ^i is also grown correspondingly, so that it includes one binary feature for every local state in the local search tree T^i.

8.3.2 Position evaluation

We assume that the outcome of the game is binary, e.g. 1 if black wins and 0 if white wins. We evaluate a global state s by estimating the expected outcome, or equivalently the probability of winning the game, using a *value function* $V(s)$. We represent the contributing value of each binary feature ϕ^i_j by a corresponding weight θ^i_j. We can then approximate the overall value of state s by forming a linear combination of all binary features with all weights (see Figure 8.1, right). We squash the result into $[0, 1]$ by using a logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$,

$$V(s) = \sigma(\phi(s)^T \theta) \tag{8.2}$$

The idea of Monte-Carlo search is to estimate the value function from simulated games from the current position. In local Monte-Carlo search, the value function is approximated by the vector of weights θ, which are learnt from simulated games. This allows the outcome of a single simulation to update the evaluation of a whole class of positions that share the same local state. This enables local Monte-Carlo search to learn much more quickly than global Monte-Carlo search.

In return for its increased generalisation and learning speed, local Monte-Carlo search introduces a new problem: that of credit assignment. Given that many local states are matched in the

course of a simulation, which local states should be given credit for winning or losing the game? We consider two well-known solutions to the credit assignment problem: logistic regression, and temporal-difference learning.

Our first algorithm, *local Monte-Carlo tree search* (LMCTS), updates the weights θ at every step t of each simulation, so as to maximise the likelihood that the value function predicts the correct outcome,

$$\Delta\theta = \alpha\left(z - V(s_t)\right)\frac{\phi(s_t)}{|\phi(s_t)|} \tag{8.3}$$

where z is the outcome of the simulation, and α is a step-size parameter. This algorithm can be viewed as a direct extension of MCTS to local search-trees, using logistic regression to estimate the contribution of each local state.

In many domains the value function $V(s)$ can be learned more efficiently by *bootstrapping* from successive estimates of the value (Sutton and Barto, 1998). When the value function is approximated by multiple features, bootstrapping can significantly improve the performance of simulation-based search algorithms (see Chapter 6, Silver et al. (2008)). Our second algorithm, *local temporal difference tree search* (LTDTS), updates the weights θ so as to maximise the likelihood that the predicted value matches the successive predicted value,

$$\Delta\theta = \alpha\left(V(s_{t+1}) - V(s_t)\right)\frac{\phi(s_t)}{|\phi(s_t)|} \tag{8.4}$$

LTDTS is a special case of temporal-difference search (Chapter 6), in which the representation is developed incrementally by using local search trees.

8.3.3 Simulation

Like MCTS, simulations are started from the current position and continue until the game terminates. Moves are selected by self-play, with both players using the same policy to select moves, in two distinct phases. During the first phase of simulation, moves are selected by an ϵ-greedy policy that maximises the global value of the successor state, $V(s \circ a)$, or selects a move at random with probability ϵ. The second phase begins after T steps, or whenever a state s is encountered for which no local states are matched, such that $\phi(s) = 0$. Moves are then selected by a default policy, such as uniform random, until the end of the game.

Since the simulations are predominantly greedy with respect to the global value, the local search trees grow in a highly selective fashion, so as to include those local states that actually occur when playing to win the game.

8.4 Local Regions in Computer Go

A state in the game of Go, $s \in \{\cdot, \circ, \bullet\}^{N\times N}$, consists of a state variable for each intersection of a size $N \times N$ board, with three possible values for empty, black and white stones respectively[1].

[1]Technically the state also includes the full board history, so as to avoid repetitions (known as *ko*).

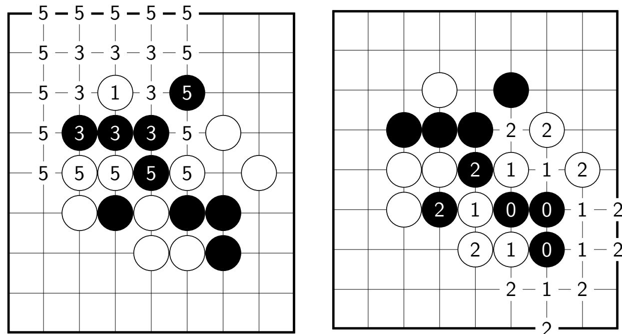

Figure 8.2: Two methods for partitioning the board in the game of Go. (Left) A hierarchy of overlapping square regions, at all possible locations on the board. An example of $R^{square(1)}$, $R^{square(3)}$, and $R^{square(5)}$ regions are shown. (Right) A hierarchy of regions within increasing Manhattan distance from each connected block of stones. An example of $R^{block(0)}$, $R^{block(1)}$ and $R^{block(2)}$ regions are shown for the black block in the bottom-right.

We partition the board into overlapping regions, each containing some local subset R^i of the state variables. We compare two partitioning methods for Computer Go.

Our first method uses a hierarchy of overlapping grids, containing square regions $R^{square(m)}$ from size 1×1 to $m \times m$ (Figure 8.2), at all possible locations on the board. Each local state ϕ^i_j matches a specific pattern of empty, black and white stones for all intersections within the ith square region $R^{square(m)}_i$ (Silver et al., 2007).

Our second partitioning method exploits the significance of connected blocks of stones in the game of Go. As we saw in Figure 8.1, much of the game revolves around the life and death of blocks, or the connection and disconnection of different blocks. We construct regions $R^{block(m)}$ surrounding each current block up to some Manhattan distance m. These regions vary dynamically: if additional stones are added to the ith block, then the corresponding region $R^{block(m)}_i$ grows larger. This allows tactical battles, such as ladders, to be tracked across the board. Again, we use a hierarchy of overlapping regions, with Manhattan distance from 0 to m (Figure 8.2).

We also compare our approach to the *local shape features* that were defined in Chapter 5. Unlike local search trees, local shape features enumerate all possible configurations within a region, and are therefore limited to small regions. In 9×9 Go, it is only practical to use local shape features for up to 3×3 square regions.

8.5 Experiments in 9×9 Computer Go

We compared our approach to the UCT implementation from the open source *Fuego* 0.1 project Müller and Enzenberger (2009). For a fair comparison, we use the default policy from *Fuego* in both our program and the MCTS program.

After a preliminary investigation into the parameters that are most effective in 9×9 Go, we set the step-size to $\alpha = 0.1$, and the length of the first phase to $T = 10$. Our current implementation of

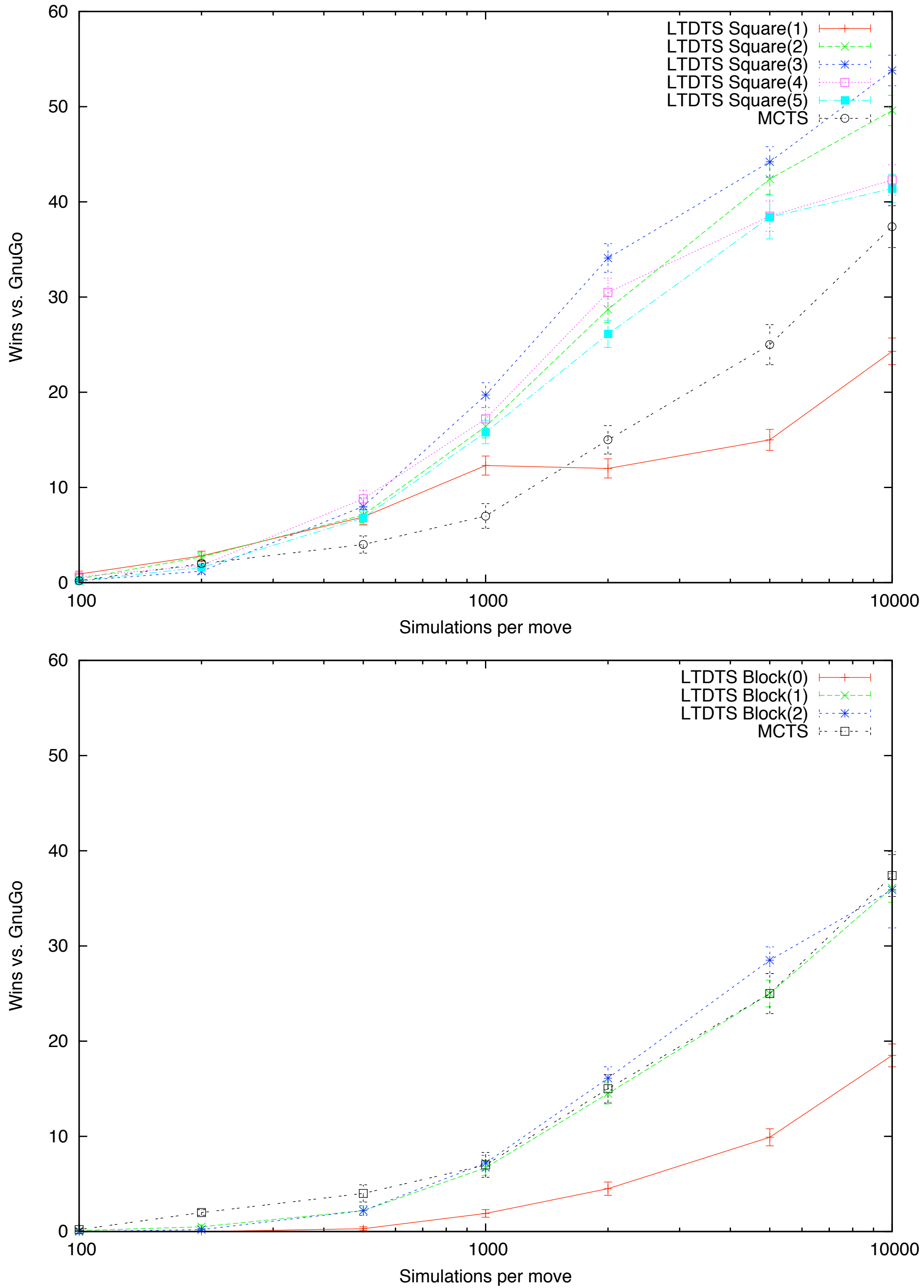

Figure 8.3: The performance of LTDTS against *GnuGo* in 9×9 Go, for increasing numbers of simulations per move. Each point represents the percentage of wins in a match of 1000 games. The board is partitioned into a) square regions $R^{square(m)}$, b) block regions $R^{block(m)}$.

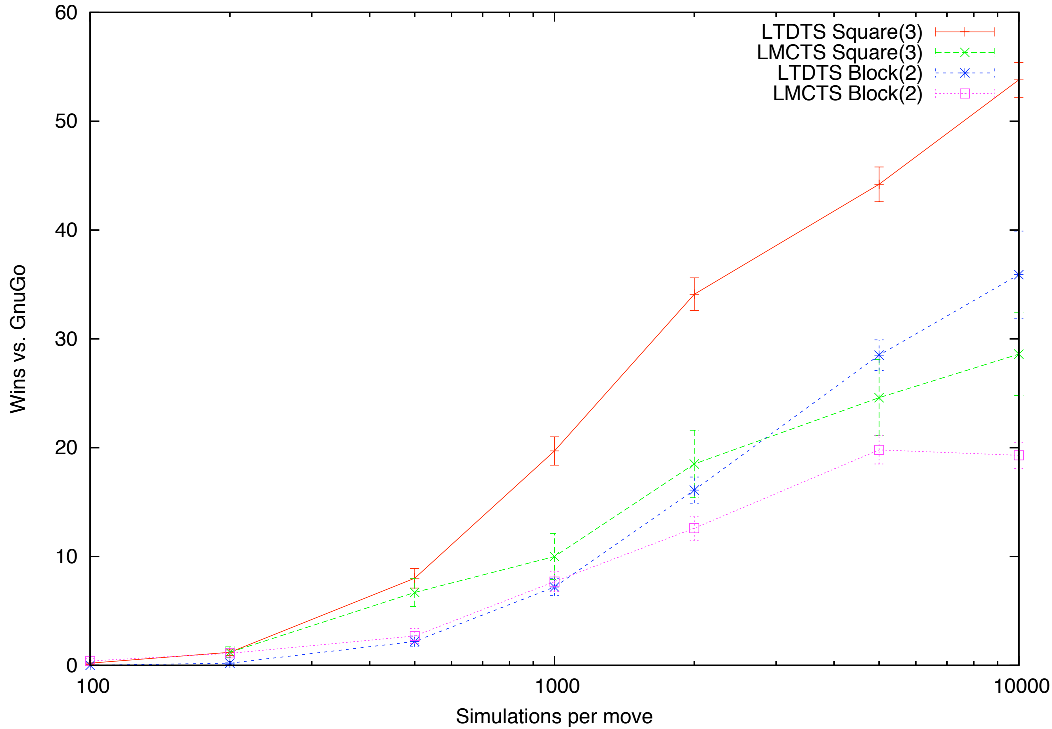

Figure 8.4: A comparison of local Monte-Carlo tree search (LMCTS) and local temporal-difference tree search (LTDTS). Each point represents the percentage of wins against *GnuGo* in 9×9 in a match of 1000 games.

LTDTS and LMCTS executes approximately 1000 simulations per second in 9×9 Go, on a single 2Ghz processor, compared to 10000 simulations per second for the MCTS program. We evaluated the performance of our algorithms against *GnuGo 3.7.10* (level 0), the standard benchmark opponent in Computer Go.

In our first experiment, we evaluated the performance of LTDTS in 9×9 Go, for a variety of different partitioning methods, including both square regions and block regions. The results are shown in Figure 8.3.

As with other Monte-Carlo search methods, LTDTS scaled well with increasing numbers of simulations per move. Using the best partitioning method, it outperformed MCTS throughout the tested range, and narrowly exceeded the performance of *GnuGo* after 10,000 simulations per move.

Perhaps surprisingly, a combination of many smaller, overlapping regions appears to give better overall performance than fewer, larger and more complex regions, with the 3×3 square regions $R^{square(3)}$ achieving the highest winning rate. This demonstrates a trade-off between representation and generalisation: larger regions can in principle represent the value function more accurately, but each local state is quite specific and occurs less frequently, reducing the level of generalisation and thus slowing down learning.

In our second experiment, we investigated the effect of bootstrapping, by comparing the perfor-

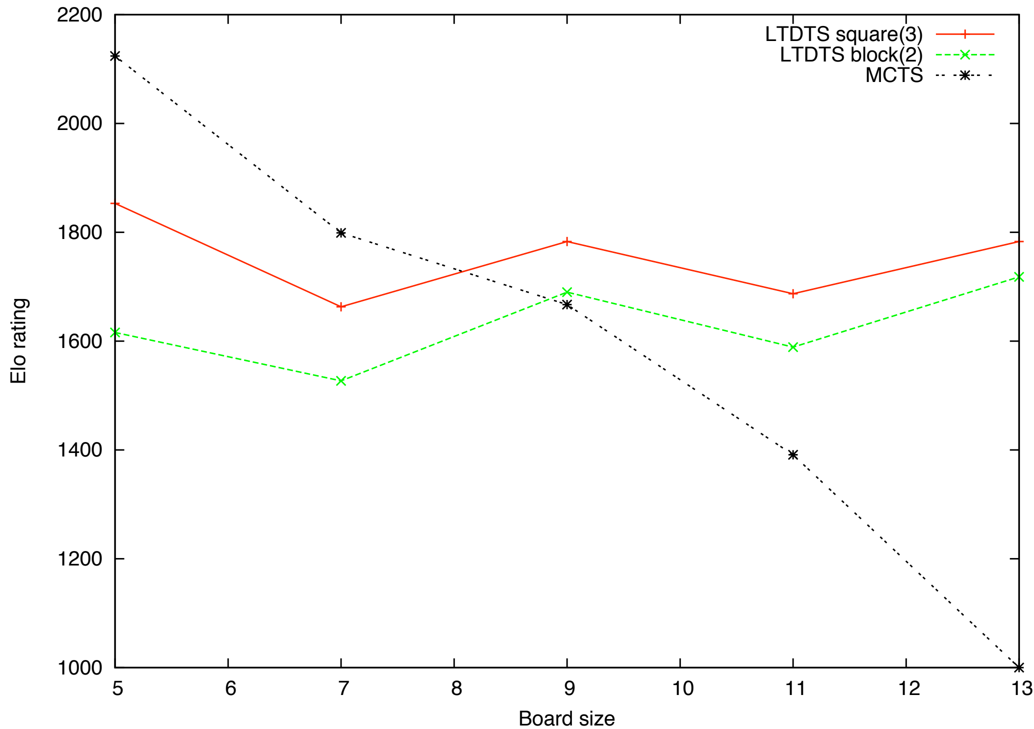

Figure 8.5: The performance of LTDTS and MCTS for different sizes of Go board, when executing 5000 simulations per move. Elo ratings were computed from a tournament for each board size, and including *GnuGo* as an anchor program. 1000 game tournaments were run for board sizes 5, 7 and 9, and 200 game tournaments for board sizes 11 and 13.

mance of LMCTS with LTDTS (Figure 8.4). Temporal difference learning outperformed Monte-Carlo learning, for both square and block regions, throughout the tested range of simulations per move.

8.6 Scalability

In our final experiment, we investigated the scalability of our approach to higher dimensional state spaces. We ran tournaments between LTDTS, MCTS, and *GnuGo*, on a variety of different board sizes from 5×5 up to 13×13. We used the same parameters for the algorithms as in the 9×9 Go experiments, with no additional tuning. The Elo rating of each program was computed from the outcome of the tournament, with *GnuGo* assigned an anchor rating of 1800 Elo. The results are shown in Figure 8.5.

On small 5×5 and 7×7 boards, the whole board is strongly interdependent, and the global search of MCTS performed best. However, MCTS was unable to maintain its level of performance in larger state spaces, and achieved just 1 win in 200 games on 13×13 boards. In contrast, the local search tree representation scaled well with larger board sizes, and the LTDTS algorithm maintained good performance against *GnuGo*. These results were achieved with just 5000 simulations per move; in

comparison Mogo required 70000 simulations per move to attain a similar level of performance on size 13×13 boards (Gelly et al., 2006).

8.7 Related work

The Go program *Mogo* implemented a scheme for partitioning the board into *zones* (Gelly et al., 2006). Each zone was evaluated separately by Monte-Carlo tree search, by restricting the simulations to moves within the zone. However, zones could not overlap, and the zone boundaries caused unrealistic artifacts. The zone algorithm was dropped from subsequent versions of *Mogo*.

8.8 Conclusions

We have demonstrated that an abstract representation of the state space, based on local search trees, can outperform a monolithic representation of the state space. Unlike traditional approaches to local search, our new algorithms scale well with increasing levels of computation. Unlike Monte-Carlo tree search, our new algorithms also scale well with increasing dimensionality.

Recently, Monte-Carlo search has been extended to incorporate domain-specific knowledge and a variety of other enhancements, achieving impressive results in Computer Go (see Chapter 9 and (Gelly et al., 2006; Gelly and Silver, 2007; Coulom, 2007; Chaslot et al., 2007)). Combining local Monte-Carlo search with these search enhancements may well provide the most fertile avenue for progress in large, challenging domains such as 19×19 Go.

Chapter 9

Heuristic MC–RAVE

9.1 Introduction

Simulation-based search has revolutionised Computer Go and is certain to have a dramatic impact on many other challenging domains. As we have seen in previous chapters, simulation-based search can be significantly enhanced: both by generalising between different states, using a short-term memory; and by incorporating general knowledge about the domain, using a long-term memory. In this chapter we apply these two ideas specifically to Monte-Carlo tree search, by introducing two extensions to the basic algorithm.

Our first extension, *heuristic Monte-Carlo tree search*, uses a heuristic function as a long-term memory. This heuristic is used to initialise the values of new positions in the search tree. As in previous chapters, we use a heuristic function that has been learned by temporal-difference learning and self-play; however, in general any heuristic can be provided to the algorithm.

Our second extension, the *RAVE* algorithm, uses a very simple generalisation between the nodes of each subtree. The value of each move is estimated, regardless of when that move was played during simulation. The RAVE algorithm forms a very fast but sometimes inaccurate estimate of the value; whereas normal Monte-Carlo is slower but more accurate. The MC–RAVE algorithm combines these two value estimates in a principled fashion, so as to minimise the mean-squared error.

We applied our algorithms in the program *MoGo*, and achieved a dramatic improvements to its performance. The resulting program became the first program to achieve *dan*-strength at 9×9 Go.

9.2 Monte-Carlo Simulation

We briefly review the key ideas of Monte-Carlo simulation, using terminology that will be helpful in explaining our new extensions.

A policy $\pi(s, a)$ represents a stochastic strategy for selecting a move a in position s. We define the value function $Q^\pi(s, a)$ to be the expected outcome of a self-play game that starts with move a in position s, and then follows policy π for both players.

Monte-Carlo simulation provides a simple method for estimating the value of positions and moves. N games are simulated using self-play, starting from position s and trying a candidate move a, and then continuing until the end of the game using policy π. The value $Q(s, a)$ is estimated from the average outcome of all simulations in which move a was selected in position s,

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N} I_i(s,a) z_i, \tag{9.1}$$

where z_i is the outcome of the ith simulation; $I_i(s, a)$ is an indicator function returning 1 if move a was selected in position s at any step during the ith simulation, and 0 otherwise; and $N(s, a) = \sum_{i=1}^{N} I_i(s, a)$ counts the total number of simulations in which move a was selected in position s.

Monte-Carlo simulation can be accelerated by using the *all-moves-as-first* (AMAF) heuristic (Bruegmann, 1993). In incremental games such as Computer Go, the value of a move is often unaffected by moves played elsewhere on the board. The idea of the AMAF heuristic is to have one general value for each move, regardless of when it is played in a simulation. In this approach, an approximate value $\tilde{Q}(s, a)$ is estimated from the average outcome of all simulations in which move a is selected *at any time* after s is encountered, regardless of what has happened elsewhere on the board.

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N} \tilde{I}_i(s,a) z_i, \tag{9.2}$$

where $\tilde{I}_i(s, a)$ is an indicator function returning 1 if position s was encountered at any step t of the ith simulation, and move a was selected at any step $u >= t$, or 0 otherwise; and $\tilde{N}(s, a) = \sum_{i=1}^{N} \tilde{I}_i(s, a)$ counts the total number of simulations used to estimate the AMAF value. Note that Black moves and White moves are considered to be distinct actions, even if they are played at the same intersection.

In order to select the best move with reasonable accuracy, Monte-Carlo simulation requires at least one simulation from every candidate move, and many more in order to estimate the value with any accuracy. The AMAF heuristic provides orders of magnitude more information: every move will typically have been tried on several occasions, after just a handful of simulations. If the value of a move really is unaffected, at least approximately, by moves played elsewhere, then this can result in a much more efficient estimate of the value.

9.3 Monte-Carlo Tree Search

Monte-Carlo tree search builds a search tree $\mathcal{T}$ containing nodes $(s, a) \in \mathcal{T}$ for some or all of the positions s and moves a encountered during simulations. Each node stores a count $n(s, a)$ and a value $Q(s, a)$ that is estimated by Monte-Carlo simulation.

Each simulation starts from the real, root position, and is divided into two stages: a *tree policy* and a *default policy*. If all moves from the current state s are represented in the tree, $\forall a \in$

$\mathcal{A}(s), (s,a) \in \mathcal{T}$, then the tree policy is used, for example to choose the move with the highest value. Otherwise the default policy is used to select an move from among all unrepresented moves $\{a|(s,a) \notin \mathcal{T}\}$.

At the end of a simulation $s_1, a_1, s_2, a_2, ..., s_T$ with outcome z, the mean value of each node in the search tree, $(s_t, a_t) \in \mathcal{T}$, is incrementally updated,

$$n(s_t, a_t) \leftarrow n(s_t, a_t) + 1 \tag{9.3}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{n(s_t, a_t)}, \tag{9.4}$$

which is equivalent to Equation 9.1.

In addition, the tree is grown by one node per simulation, by adding the first position and move in the second stage.

9.4 Heuristic Prior Knowledge

If a particular position s and move a is rarely encountered during simulation, then its Monte-Carlo value estimate is highly uncertain and very unreliable. Furthermore, because the search tree branches exponentially, the vast majority of nodes in the tree are only experienced rarely. The situation at the leaf nodes is worst of all: by definition each leaf node has been visited only once (otherwise a child node would have been added).

In order to reduce the uncertainty for rarely encountered positions, we incorporate prior knowledge by using a *heuristic evaluation function* $H(s,a)$ and a *heuristic confidence function* $C(s,a)$. When a node is first added to the search tree, it is initialised according to the heuristic function, $Q(s,a) = H(s,a)$ and $N(s,a) = C(s,a)$. The confidence in the heuristic function is measured in terms of *equivalent experience*: the number of simulations that would be required in order to achieve a Monte-Carlo value of similar accuracy to the heuristic value[1]. After initialisation, the value and count are updated as usual, using standard Monte-Carlo simulation.

9.5 Rapid Action Value Estimation (RAVE)

Monte-Carlo tree search learns a unique value for each node in the search tree, and cannot generalise between related positions. The RAVE algorithm provides a simple way to share knowledge between related nodes in the search tree, resulting in a rapid, but biased value estimate.

The RAVE algorithm combines Monte-Carlo tree search with the all-moves-as-first heuristic. Instead of evaluating each node (s,a) of the search-tree by Monte-Carlo (Equation 9.1), each node is evaluated by the all-moves-as-first heuristic (Equation 9.2).

[1] This is equivalent to a beta prior when binary outcomes are used.

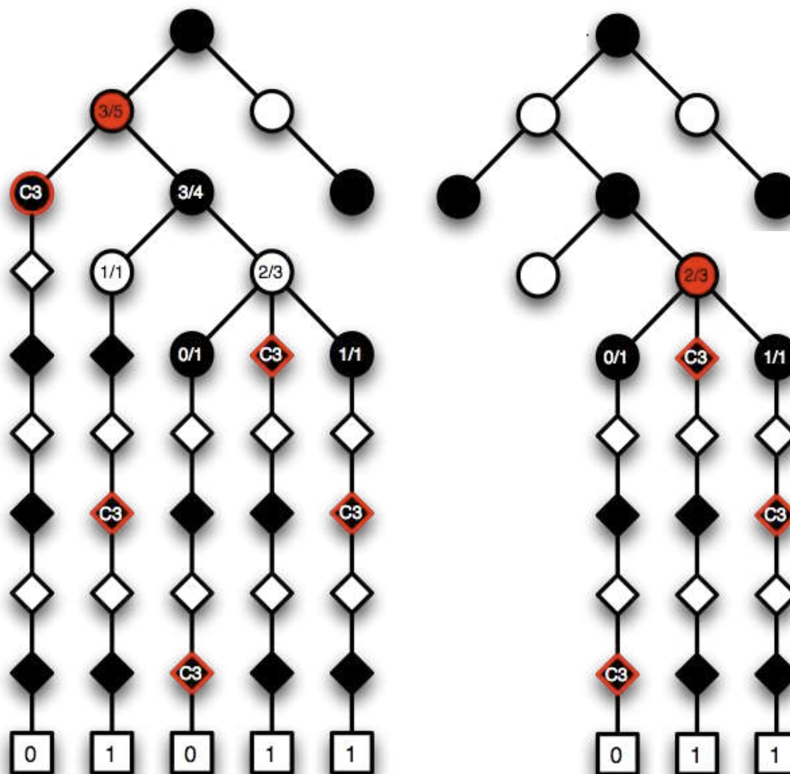

Figure 9.1: An example of using the RAVE algorithm to estimate the value of black c3. Several simulations have already been performed and are shown in the search tree. During the next simulation, black uses RAVE to select his next move to play, first from the solid red node in the left diagram, and then from the solid red node in the right diagram. The RAVE values for Black c3 are shown for the subtree beneath the solid red node. (Left) Black c3 has been played once, and resulted in a loss; its Monte-Carlo value is 0/1. Black c3 has been played 5 times in the subtree beneath the red node, resulting in 3 wins and two losses; its RAVE value is 3/5. (Right) Black c3 has been played once, and resulted in a win; its Monte-Carlo value is 1/1. It has been played 3 times in the subtree, resulting in 2 wins and one loss; its RAVE value is 2/3.

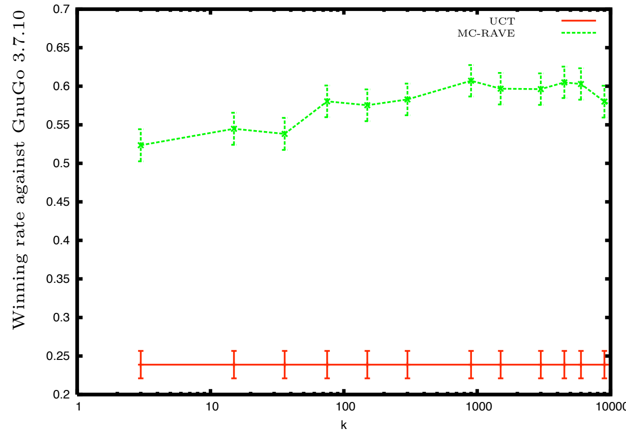

Figure 9.2: Winning rate of MC–RAVE with 3000 simulations per move against GnuGo 3.7.10 (level 10), for different settings of the equivalence parameter k. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

Every position in the search tree, $s \in \mathcal{T}$, is the root of a subtree $\tau(s) \subseteq \mathcal{T}$. If a simulation visits the root position s at step t, then all subsequent nodes visited by the first stage of simulation, (s, a) where $u \geq t$, are in this subtree, $(s, a) \in \tau(s)$. The second stage of the simulation can also be considered to be part of the subtree: even though these positions are not currently stored, $(s, a) \notin \tau(s)$, given enough further simulations they will eventually become part of the subtree.

The basic idea of RAVE is to generalise over subtrees. The assumption is that the value of move a will be similar from all positions within a subtree s. Thus, the value of a is estimated from all simulations starting from s, regardless of exactly when a is played.

When the RAVE values are used to select a move, the most specific subtree is used, $\tilde{Q}(s, a)$. In our experiments, combining value estimates from more general subtrees did not confer any advantage.

RAVE is closely related to the *history heuristic* in alpha-beta search (Schaeffer, 1989). During the depth-first traversal of the search tree, the history heuristic remembers the success[2] of each move at various depths; the most successful moves are tried first in subsequent positions. RAVE is similar, but because the search is not developed in a depth-first manner, it must explicitly store a separate value for each subtree. In addition, RAVE takes account of the success of all moves in the simulation, not just moves within the subtree.

9.6 MC–RAVE

The RAVE algorithm learns very quickly, but it is often wrong. The principal assumption of RAVE, that a particular move has the same value across an entire subtree, is frequently violated. There are many situations, for example during tactical battles, in which nearby changes can completely change

[2] A successful move in alpha-beta either causes a cut-off, or has the best minimax value.

the value of a move: sometimes rendering it redundant; sometimes making it even more vital. Even distant moves can significantly affect the value of a move, for example playing a ladder-breaker in one corner can radically alter the value of playing a ladder in the opposite corner.

To overcome this issue, we develop an algorithm, MC–RAVE, which combines the rapid learning of the RAVE algorithm with the accuracy and convergence guarantees of Monte-Carlo tree search. The basic idea is to store *two* values with every node (s, a) of the search tree, using both the MC value and the AMAF value. To estimate the overall value of move a in position s, we use a weighted sum of the Monte-Carlo value $Q(s, a)$ and the AMAF value $\tilde{Q}(s, a)$. When only a few simulations have been seen, we weight the AMAF value more highly. When many simulations have been seen, we weight the Monte-Carlo value more highly. The schedule is different for each node in the tree, depending on the number of simulations from that particular node.

We can now describe the complete MC–RAVE algorithm. As with Monte-Carlo tree search, each simulation is divided into two stages. During the first stage, for positions $s \in \mathcal{T}$ within the search tree, moves are selected so as to maximise the combined MC and AMAF value $Q_\star(s, a)$,

$$Q_\star(s, a) = \beta Q(s, a) + (1 - \beta)\tilde{Q}(s, a) \tag{9.5}$$

During the second stage of simulation, for positions $s \notin \mathcal{T}$ beyond the search tree, moves are selected according to a default policy.

After each simulation $s_1, a_1, s_2, a_2, ..., s_T$ with outcome z, both the MC and AMAF values are updated. For every position s_t and action a_t in the simulation, if there is a corresponding node in the search tree, $(s_t, a_t) \in \mathcal{T}$, then its Monte-Carlo value is updated,

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1 \tag{9.6}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{N(s_t, a_t)} \tag{9.7}$$

In addition, the AMAF value is updated for every subtree. For every position s_t in the simulation that is represented in the tree, $s_t \in \mathcal{T}$, and for every subsequent move of the simulation a, such that $u \geq t$, the AMAF value of (s_t, a) is updated according to the simulation outcome z,

$$\tilde{N}(s_t, a) \leftarrow \tilde{N}(s_t, a) + 1 \tag{9.8}$$

$$\tilde{Q}(s_t, a) \leftarrow \tilde{Q}(s_t, a) + \frac{z - \tilde{Q}(s_t, a)}{\tilde{N}(s_t, a)} \tag{9.9}$$

If the same move is played multiple times during a simulation, then this update is only performed the first time. Moves which are illegal in s_t are ignored.

One possible schedule for MC–RAVE uses an *equivalence parameter* k to specify the number of simulations at which the Monte-Carlo value and the AMAF value should be given equal weight,

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}} \tag{9.10}$$

where $N(s)$ is the total number of simulations from position s, $N(s) = \sum_a N(s,a)$.

We tested MC–RAVE in the Go program *MoGo*, using the schedule in Equation (9.10) and the default policy described in (Gelly et al., 2006), for different settings of the equivalence parameter k. For each setting, we played a 2300 game match against GnuGo 3.7.10 (level 10). The results are shown in Figure 9.2, and compared to Monte-Carlo tree search, using 3000 simulations per move for both algorithms. The winning rate using MC–RAVE varied between 50% and 60%, compared to 24% without RAVE. Maximum performance is achieved with an equivalence parameter of 1000 or more, indicating that the rapid action value estimate is more reliable than standard Monte-Carlo simulation until several thousand simulations have been executed from position s.

9.7 Minimum MSE Schedule

The schedule presented in Equation 9.10 is somewhat heuristic in nature. We now develop a more principled schedule, which selects $\beta(s,a)$ so as to minimise the mean squared error in the combined estimate $Q_\star(s,a)$.

9.7.1 Assumptions

We make three simplifying assumptions to derive our algorithm:

1. The MC and AMAF values for each node are independent.
2. The MC value is unbiased.
3. The outcomes of the simulations are binomially distributed.

In fact, the same simulations used to compute the MC value are also used to compute the AMAF value, which means that the values are correlated. Furthermore, as the tree develops over time, the simulation policy changes. This leads to bias in the MC value, and also means that outcomes are not in fact binomially distributed. Nevertheless, we believe that these simplifications do not significantly affect the performance of the schedule in practice.

9.7.2 Derivation

To simplify our notation, we consider a single position s and move a, and denote the estimated mean, bias and variance of the MC, AMAF and combined values for this position and move by $\mu, \tilde{\mu}, \mu_\star$; $b, \tilde{b}, b_\star$ and $\sigma^2, \tilde{\sigma}^2, \sigma^2_\star$, respectively. We denote the number of Monte-Carlo simulations by $n = N(s,a)$ and the number of simulations used to compute the RAVE value by by $\tilde{n} = \tilde{N}(s,a)$.

We start by decomposing the mean-squared error of the combined value into the bias and variance of the MC and AMAF values respectively, making use of our first and second assumptions,

$$MSE = \sigma_\star^2 + b_\star^2 \tag{9.11}$$
$$= \beta^2\tilde{\sigma}^2 + (1-\beta)^2\sigma^2 + (\beta\tilde{b} + (1-\beta)b)^2 \tag{9.12}$$
$$= \beta^2\tilde{\sigma}^2 + (1-\beta)^2\sigma^2 + \beta^2\tilde{b}^2 \tag{9.13}$$

Differentiating with respect to β and setting to zero,

$$0 = 2\beta\tilde{\sigma}^2 - 2(1-\beta)\sigma^2 + 2\beta\tilde{b}^2 \tag{9.14}$$
$$\beta = \frac{\sigma^2}{\sigma^2 + \tilde{\sigma}^2 + \tilde{b}^2} \tag{9.15}$$
$$\tag{9.16}$$

We now make use of our third assumption, that the outcome of the n simulations used to estimate the MC value is binomially distributed. Similarly, we assume that the outcome of the $\tilde{n}$ simulations used to estimate the AMAF value is binomially distributed. However, we don't know the true mean of these distributions. Instead, we approximate the true mean by using the current best estimate $\mu_\star$ of the value.

$$\sigma^2 = \mu_\star(1-\mu_\star)/n \tag{9.17}$$
$$\tilde{\sigma}^2 = \mu_\star(1-\mu_\star)/\tilde{n} \tag{9.18}$$
$$\beta = \frac{\tilde{n}}{n + \tilde{n} + n\tilde{n}\tilde{b}^2/\mu_\star(1-\mu_\star)} \tag{9.19}$$

This equation still includes one unknown constant: the RAVE bias $\tilde{b}$. This can either be evaluated empirically (by testing the performance of the algorithm with various values of $\tilde{b}$), or by machine learning (by learning to predict the average difference between the AMAF value and the MC value, after many simulations). The former method is simple and effective; but the latter method could allow different biases to be identified for different types of position.

Putting it all together, we arrive at the following equation to updated the schedule and estimate the combined value $Q_\star(s,a)$ of each node,

$$Q_\star(s,a) = \beta\tilde{Q}(s,a) + (1-\beta(s,a))Q(s,a) \tag{9.20}$$
$$\beta(s,a) = \frac{\tilde{N}(s,a)}{N(s,a) + \tilde{N}(s,a) + N(s,a)\tilde{N}(s,a)\tilde{b}(s,a)^2/Q_\star(s,a)(1-Q_\star(s,a))} \tag{9.21}$$

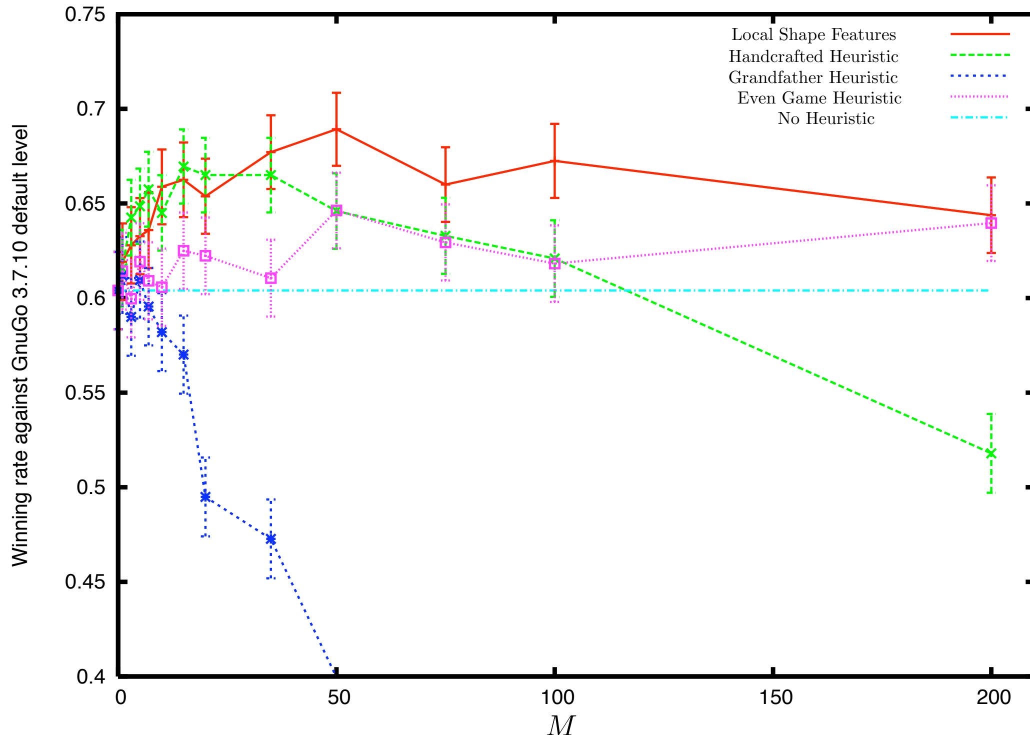

Figure 9.3: Winning rate of *MoGo*, using the heuristic MC–RAVE algorithm, with 3000 simulations per move against GnuGo 3.7.10 (level 10). Four different forms of heuristic function were used (see text). The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

We implemented the minimum error MC–RAVE algorithm in *MoGo*[3]. On a 9×9 board, the performance of MoGo increased by around 60 Elo. On a 19×19 board, the improvement was substantially larger.

9.8 Heuristic MC–RAVE

The heuristic Monte-Carlo tree search algorithm can be combined with the MC–RAVE algorithm. When a new position and move (s, a) is added to the tree $\mathcal{T}$, we initialise both the MC and AMAF values to the heuristic evaluation function, and initialise both counts according to heuristic confidence functions C and $\tilde{C}$ respectively,

[3]The implementation of this algorithm in *MoGo* ignores the $Q(s, a)(1 - Q(s, a))$ term, with no noticable decrease in playing strength.

$$Q(s,a) \leftarrow H(s,a) \tag{9.22}$$

$$N(s,a) \leftarrow C(s,a) \tag{9.23}$$

$$\tilde{Q}(s,a) \leftarrow H(s,a) \tag{9.24}$$

$$N(s,a) \leftarrow \tilde{C}(s,a) \tag{9.25}$$

We compare four heuristic evaluation functions in 9×9 Go, using the heuristic MC–RAVE algorithm in the program *MoGo*.

1. The *even-game* heuristic, $Q_{even}(s,a) = 0.5$, makes the assumption that most positions encountered between strong players are likely to be close.

2. The *grandfather* heuristic, $Q_{grand}(s_t,a) = Q(s_{t-2},a)$, sets the value of each node in the tree to the value of its grandfather. This assumes that the value of a Black move is usually similar to the value of that move, last time Black was to play.

3. The *handcrafted* heuristic, $Q_{mogo}(s,a)$, is based on the pattern-based rules that were successfully used in *MoGo's* default policy. The heuristic was designed such that moves matching a "good" pattern were assigned a value of 1, moves matching a "bad" pattern were given value 0, and all other moves were assigned a value of 0.5. The good and bad patterns were identical to those used in *MoGo*, such that selecting moves greedily according to the heuristic, and breaking ties randomly, would exactly produce the default policy π_{mogo}.

4. The *local shape* heuristic, $Q_{rlgo}(s,a)$, is computed from a linear combination of local shape features. This heuristic is learned offline by temporal difference learning, from games of self-play, exactly as described in Chapter 5.

For each heuristic evaluation function, we assign a heuristic confidence $\tilde{C}(s,a) = M$, for various constant values of equivalent experience M. We played 2300 games between *MoGo* and GnuGo 3.7.10 (level 10). The MC–RAVE algorithm executed 3000 simulations per move (see Figure 9.3).

The value function learned from local shape features, Q_{rlgo}, outperformed all the other heuristics and increased the winning rate of *MoGo* from 60% to 69%. Maximum performance was achieved using an equivalent experience of $M = 50$, which indicates that Q_{rlgo} is worth about as much as 50 simulations using all-moves-as-first. It seems likely that these results could be further improved by varying the heuristic confidence according to the particular position, based on the variance of the heuristic evaluation function.

9.9 Exploration and Exploitation

The performance of Monte-Carlo tree search is greatly improved by carefully balancing exploration with exploitation. The UCT algorithm significantly outperforms a greedy tree policy in Computer

Go (Gelly et al., 2006). Surprisingly, this result does not extend to the heuristic MC–RAVE algorithm: the optimal exploration rate in our experiments was zero, i.e. a greedy tree policy.

We believe that the explanation lies in the nature of the RAVE algorithm. Even if a move a is not selected immediately from position s, it will often be played at some later point in the simulation. This greatly reduces the need for explicit exploration, as the values for all moves are continually updated, regardless of the initial move selection.

9.10 Heuristics and RAVE in Dyna-2

In this section we show that the two extensions to Monte-Carlo tree search are special cases of the Dyna-2 architecture (see Chapter 7). We make this connection explicit in order to underline the similarities between the new algorithms developed in this chapter, and the ideas we have explored in previous chapters. We view the Dyna-2 architecture as a common framework for understanding and improving simulation-based search algorithms, even if the special cases used in *MoGo* can be implemented in a more direct and efficient manner.

9.10.1 Heuristic Monte-Carlo tree search in Dyna-2

Heuristic Monte-Carlo tree search can be exactly implemented by the Dyna-2 algorithm, by using different feature vectors in the long and short-term memories. The heuristic evaluation function is represented and learned by the long-term memory in Dyna-2. For example, the evaluation function $Q_{rlgo}(s, a)$ was learned by using local shape features in the long-term memory. The short-term memory in Dyna-2 contains the search tree. This can be represented in Dyna-2 by using an individual feature $\bar{b}\phi(s, a) = \phi^{S,A}(s, a)$ to match each node in the search tree $(S, A) \in \mathcal{T}$,

$$\phi^{S,A}(s, a) = \begin{cases} 1 & \text{if } s = S \text{ and } a = A; \\ 0 & \text{otherwise.} \end{cases} \tag{9.26}$$

The temporal difference parameter is set to $\lambda = 1$ so as to replicate Monte-Carlo evaluation[4]. The heuristic confidence function $C(s, a)$ provides the step-size schedule for Dyna-2,

$$\alpha(s, a) = \frac{1}{C(s, a) + N(s, a)}. \tag{9.27}$$

9.10.2 RAVE in Dyna-2

The RAVE algorithm can be implemented in the Dyna-2 architecture, but using features of the history and not just the current state.

We define a history h_t to be a sequence of states and actions $h_t = s_1 a_1 ... s_t a_t$, including the current action a_t. We define a *RAVE feature* $\tilde{\phi}^{S,A}(h)$ that matches move A in or beyond the subtree

[4] In our experiments with Monte-Carlo tree search, where each state of the search tree is represented individually, bootstrapping slightly reduced performance. When the state is represented by multiple features, bootstrapping was always beneficial.

Simulations	Wins .v. GnuGo	CGOS rating
3000	69%	1960
10000	82%	2110
70000	92%	2320*

Table 9.1: Winning rate of *MoGo* against GnuGo 3.7.10 (level 10) when the number of simulations per move is increased. The asterisked version used on CGOS modifies the simulations/move according to the available time, from 300000 games in the opening to 20000.

from S, $\tau(S)$. This binary feature has value 1 iff S occurs in the history and A matches the current action a_t,

$$\tilde{\phi}^{S,A}(s_1 a_1 s_2 a_2 ... s_t a_t) = \begin{cases} 1 & \text{if } a_t = a \text{ and } \exists i \text{ s.t. } s_i = s; \\ 0 & \text{otherwise.} \end{cases} \tag{9.28}$$

To implement RAVE in Dyna-2, we use a short-term memory consisting of one RAVE feature for every node in the search tree. This set of features provides the same abstraction over the search tree as the RAVE algorithm in *MoGo*, generalising over the same move within each subtree. We again set the temporal difference parameter to $\lambda = 1$. However, the RAVE algorithm used in *MoGo* makes two additional simplifications. First, the value of each RAVE feature is estimated independently, whereas in Dyna-2 the values of the RAVE features are learned by linear or logistic regression. This shares the credit for each win or loss among the RAVE features for all subtrees visited during that simulation. Second, *MoGo* selects moves using only the most specific subtree, whereas in Dyna-2 the evaluation function takes account of subtrees of all levels.

In principle it could be advantageous to combine RAVE features from all levels of the tree. However, in our experiments with *MoGo*, learning or combining value estimates from multiple subtrees did not appear to confer any clear advantage.

9.11 Conclusions

The performance of Monte-Carlo tree search can be significantly improved by generalising over subtrees, and by incorporating prior knowledge in the form of a heuristic function. Each technique increased the winning rate of *MoGo* against GnuGo, from 24% for the basic search algorithm, up to 69% using both techniques. However, these results are based on executing just 3000 simulations per move. When the number of simulations increases, the overall performance of *MoGo* improves correspondingly. Table 9.1 shows how the performance of heuristic MC–RAVE scales with additional computation.

Subsequent versions of *MoGo* improved the handcrafted heuristic function, and tweaked a number of parameters and optimisations in the MC–RAVE algorithm. The scalability of the improved version is shown in Figure 9.4, based on the results of a study over many thousands of computer hours (Dailey, 2008). The improved version of *MoGo* became the first program to achieve *dan*-

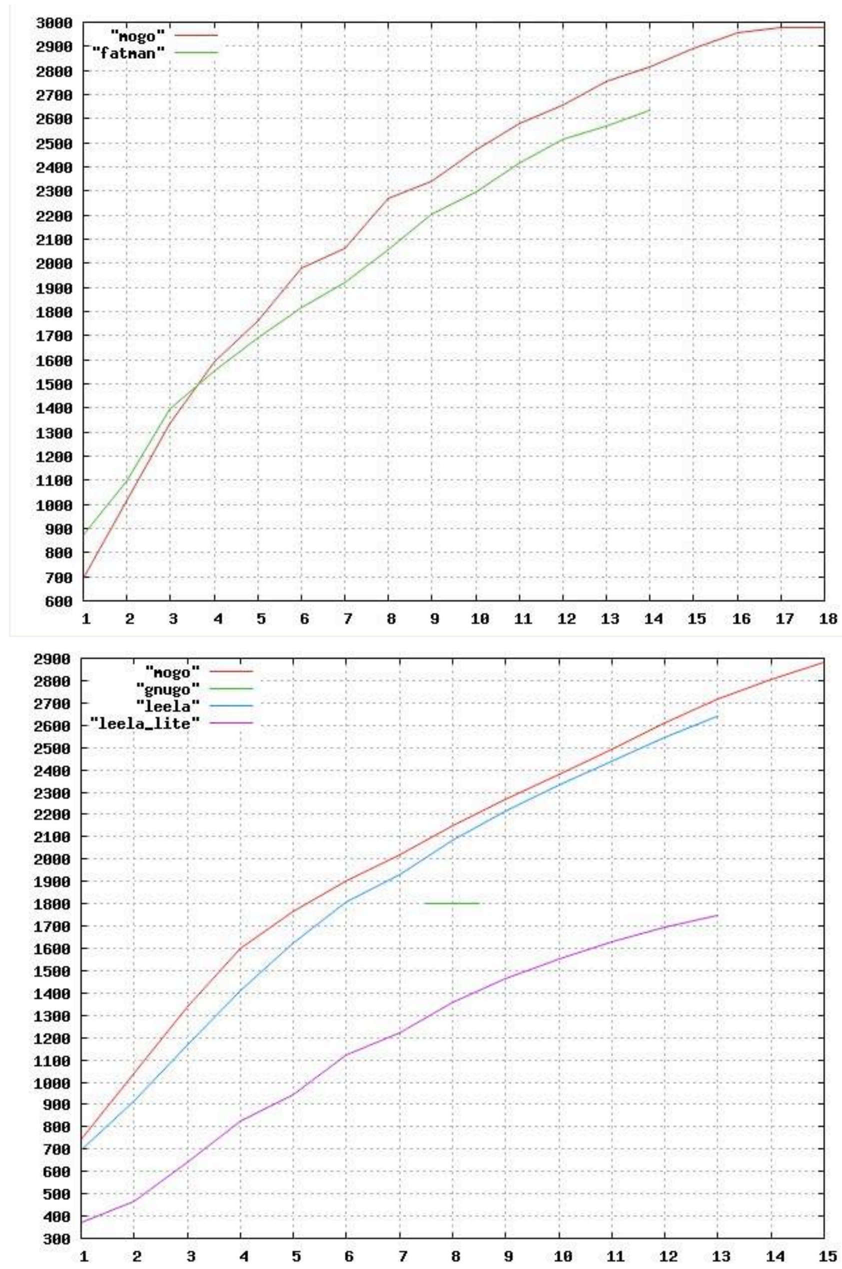

Figure 9.4: Scalability of *MoGo*, reproduced with thanks from (Dailey, 2008). The x-axis represents successive doublings of computation. Elo ratings were computed from a large tournament, consisting of several thousand games for each player, for each program with different levels of computation. a) Scalability of *MoGo* and the program *Fatman* in 9×9 Go. *MoGo* uses 2^{x+5} simulations per move; *Fatman* uses 2^{x+9}. b) Scalability of *MoGo* and the program *Leela* in 13×13 Go, in a tournament among each program with different levels of computation. *MoGo* and *Leela* both use 2^{x+8} simulations per move.

strength at 9×9 Go; the first program to beat a professional human player at 9×9 Go; the highest rated program on the Computer Go Server for both 9×9 and 19×19 Go; and the gold medal winner at the 2007 Computer Go Olympiad.

Chapter 10

Monte-Carlo Simulation Balancing

10.1 Introduction

Monte-Carlo search has dramatically outperformed traditional search methods in two-player games such as Go (Gelly and Silver, 2007; Coulom, 2007), Backgammon (Tesauro and Galperin, 1996) and Scrabble (Sheppard, 2002). The performance of Monte-Carlo search is primarily determined by the quality of the simulation policy. However, a policy that always plays strong moves does not necessarily produce the diverse, well-balanced simulations that are desirable for a Monte-Carlo player. In this chapter we introduce a new paradigm for learning a simulation policy, which explicitly optimises a Monte-Carlo objective.

Broadly speaking, two approaches have previously been taken to improving the simulation policy. The first approach is to directly construct a *strong* simulation policy that performs well by itself, either by hand (Billings et al., 1999), reinforcement learning (Tesauro and Galperin, 1996; Gelly and Silver, 2007), or supervised learning (Coulom, 2007). Unfortunately, a stronger simulation policy can actually lead to a weaker Monte-Carlo search (Gelly and Silver, 2007), a paradox that we explore further in this chapter. One consequence is that human expertise, which typically provides domain knowledge for maximising strength, can be particularly misleading in this context.

The second approach to learning a simulation policy is by trial and error, adjusting parameters and testing for improvements in the performance of the Monte-Carlo player, either by hand (Gelly et al., 2006), or by hill-climbing (Chaslot et al., 2008a). However, each parameter evaluation typically requires many complete games, thousands of positions, and millions of simulations to be executed. Furthermore, hill-climbing methods do note scale well with increasing dimensionality, and fare poorly with complex policy parameterisations.

Handcrafting an effective simulation policy has proven to be particularly problematic in Go. Most of the top Go programs utilise a small number of simple patterns and rules, based largely on the default policy used in *Mogo* (Gelly et al., 2006). Adding further Go knowledge without breaking *Mogo's* "magic formula" has proven to be surprisingly difficult.

We take a new approach to learning a simulation policy. We define an objective function, which

we call *balance*, that explicitly measures the performance of a simulation policy for Monte-Carlo evaluation. We introduce two new algorithms that optimise the balance of a simulation policy by gradient descent. These algorithms require very little computation for each parameter update, and are able to learn expressive simulation policies with hundreds of parameters.

We evaluate our simulation balancing algorithms in the domain of 5×5 and 6×6 Go. We compare them to reinforcement learning and supervised learning algorithms for maximising strength, and to a well-known simulation policy for this domain, handcrafted by trial and error. The simulation policy learnt by our new algorithms significantly outperforms prior approaches.

10.2 Learning a Simulation Policy in 9×9 Go

In Chapter 5, we learned an evaluation function for the game of Go, by reinforcement learning and self-play. In the previous chapter, we successfully applied this evaluation function in the search tree of a Monte-Carlo tree search. It is natural to suppose that this same evaluation function, which provides a fast, simple heuristic for playing reasonable Go, might be successfully applied to the default policy of a Monte-Carlo search.

10.2.1 Stochastic Simulation Policies

In a deterministic domain, Monte-Carlo simulation requires a stochastic simulation policy. If there is insufficient diversity in the simulations, then averaging over many simulations provides no additional benefit. We consider three different approaches for generating a stochastic simulation policy from the learned evaluation function Q_{rlgo}.

First, we consider an ϵ-greedy policy,

$$\pi_\epsilon(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a = \operatorname*{argmax}_{a'} Q_{rlgo}(s,a') \\ \frac{\epsilon}{|A(s)|} & \text{otherwise} \end{cases}$$

Second, we consider a greedy policy, where the evaluation function has been corrupted by Gaussian noise $\eta(s,a) \sim N(0,\sigma^2)$,

$$\pi_\sigma(s,a) = \begin{cases} 1 & \text{if } a = \operatorname*{argmax}_{a'} Q_{rlgo}(s,a') + \eta(s,a') \\ 0 & \text{otherwise} \end{cases}$$

Third, we select moves using a softmax distribution with an explicit *temperature* parameter τ controlling the level of stochasticity,

$$\pi_\tau(s,a) = \frac{e^{Q_{rlgo}(s,a)/\tau}}{\sum_{a'} e^{Q_{rlgo}(s,a')/\tau}}$$

10.2.2 Strength of Simulation Policies

We compared the performance of each class of simulation policy π_ϵ, π_σ, and π_τ, with *MoGo's* default policy π_{mogo}, and the uniform random policy π_{random}. Figure 10.1 assesses the strength of

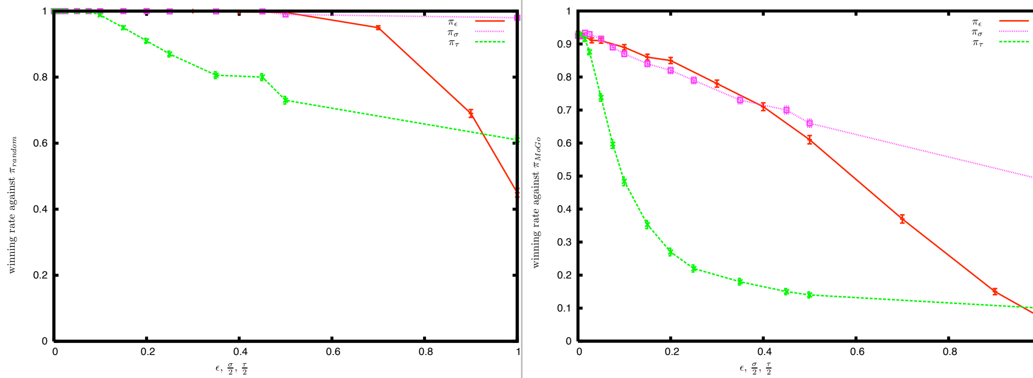

Figure 10.1: The relative strengths of each class of default policy, against the random policy π_{random} (left) and against a handcrafted policy π_{MoGo} (right). The x axis represents the degree of randomness in each policy.

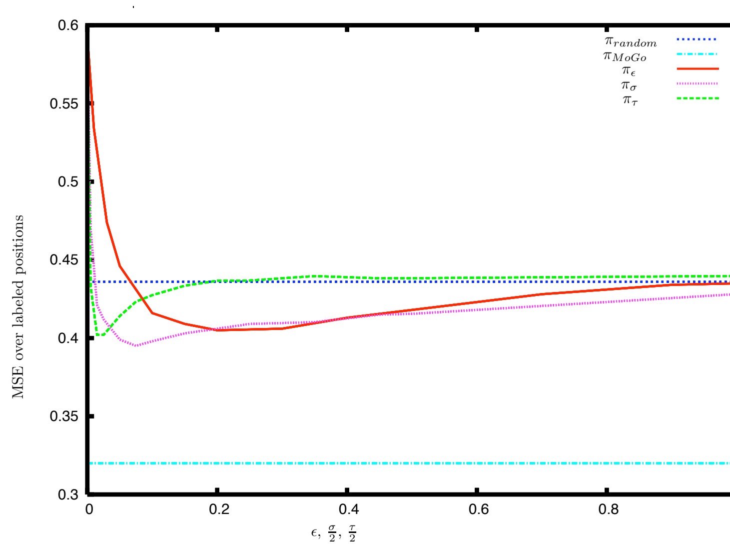

Figure 10.2: The MSE of each policy π when Monte Carlo simulation is used to evaluate a test suite of 200 hand-labelled positions. The x axis indicates the degree of randomness in the policy.

each policy, directly as a Go player, in a round-robin tournament of 6000 games between each pair of policies. With little or no randomisation, the policies based on Q_{rlgo} were by far the strongest, and outperformed both the random policy π_{random} and *MoGo's* handcrafted policy π_{mogo} by a margin of over 90%. As the level of randomisation increased, the policies degenerated towards the random policy π_{random}.

10.2.3 Accuracy of Simulation Policies in Monte-Carlo Simulation

In general, a good simulation policy for Monte-Carlo tree search is one that can evaluate each position accurately, when using Monte-Carlo simulation with no search tree. We measured the performance of Monte-Carlo simulation on a test suite of 200 mid to late-game positions, each of which was hand-labelled as a win or loss by a human expert. 1000 simulations were played from each test

Default Policy	Wins .v. GnuGo
π_{random}	8.88 $\pm$ 0.42 %
π_{σ}	9.38 $\pm$1.9%
π_{mogo}	48.62 $\pm$1.1%

Table 10.1: Winning rate of the basic UCT algorithm in *MoGo* against GnuGo 3.7.10 (level 0), given 5000 simulations per move, using different default policies. The numbers after the $\pm$ correspond to the standard error from several thousand complete games. π_σ is used with $\sigma = 0.15$.

position using each simulation policy. A Monte-Carlo value was estimated by the mean outcome of these simulations. For each simulation policy, we measured the mean-squared error (MSE) between the Monte-Carlo value and the hand-labelled value (see Figure 10.2)[1].

A stronger and appropriately randomised simulation policy achieved a lower MSE than uniform random simulations. However, if the default policy was too deterministic, then Monte-Carlo simulation failed to provide any benefits and the MSE increased dramatically. If the default policy was too random, then it became equivalent to the random policy π_{random}.

Surprisingly, the accuracy of π_ϵ, π_σ and π_τ never come close to the accuracy of the handcrafted policy π_{MoGo}, despite the fact that these policies were much stronger Go players. To verify that the default policies based on Q_{rlgo} were indeed stronger in our particular suite of test positions, we re-ran the round-robin tournament, starting from each of these positions in turn, and found that the relative strengths of the default policies remained very similar. We also compared the performance of a complete Monte-Carlo tree search, using the program *MoGo* and plugging in the simulation policy that minimised MSE as a default policy (see Table 10.1). Again, despite being significantly stronger than π_{mogo}, the simulation policy based on Q_{rlgo} performed much worse overall, performing no better than the uniform random policy.

From these results, we conclude that the spread of outcomes produced by the simulation policy is more important than its strength. Each policy has its own bias, leading it to a particular distribution of simulations. If this distribution is skewed away from the minimax value, then the overall performance can be significantly poorer. In the next section, we develop this intuition into a formal objective function for simulation policies.

10.3 Strength and Balance

We consider deterministic two-player games of finite length with a terminal outcome or score $z \in \mathbb{R}$. During simulation, move a is selected in state s according to a stochastic simulation policy $\pi_p(s, a)$ with parameter vector p, that is used to select moves for both players. The goal is to find the parameter vector p^* that maximises the overall playing strength of a player based on Monte-Carlo search. Our approach is to make the Monte-Carlo evaluations in the search as accurate as possible, by minimising the mean squared error between the estimated values $V(s) = \frac{1}{N}\sum_{i=1}^{N} z_i$ and the

[1] During the development of *MoGo*, the MSE on this test suite was usually a good indicator of overall playing strength.

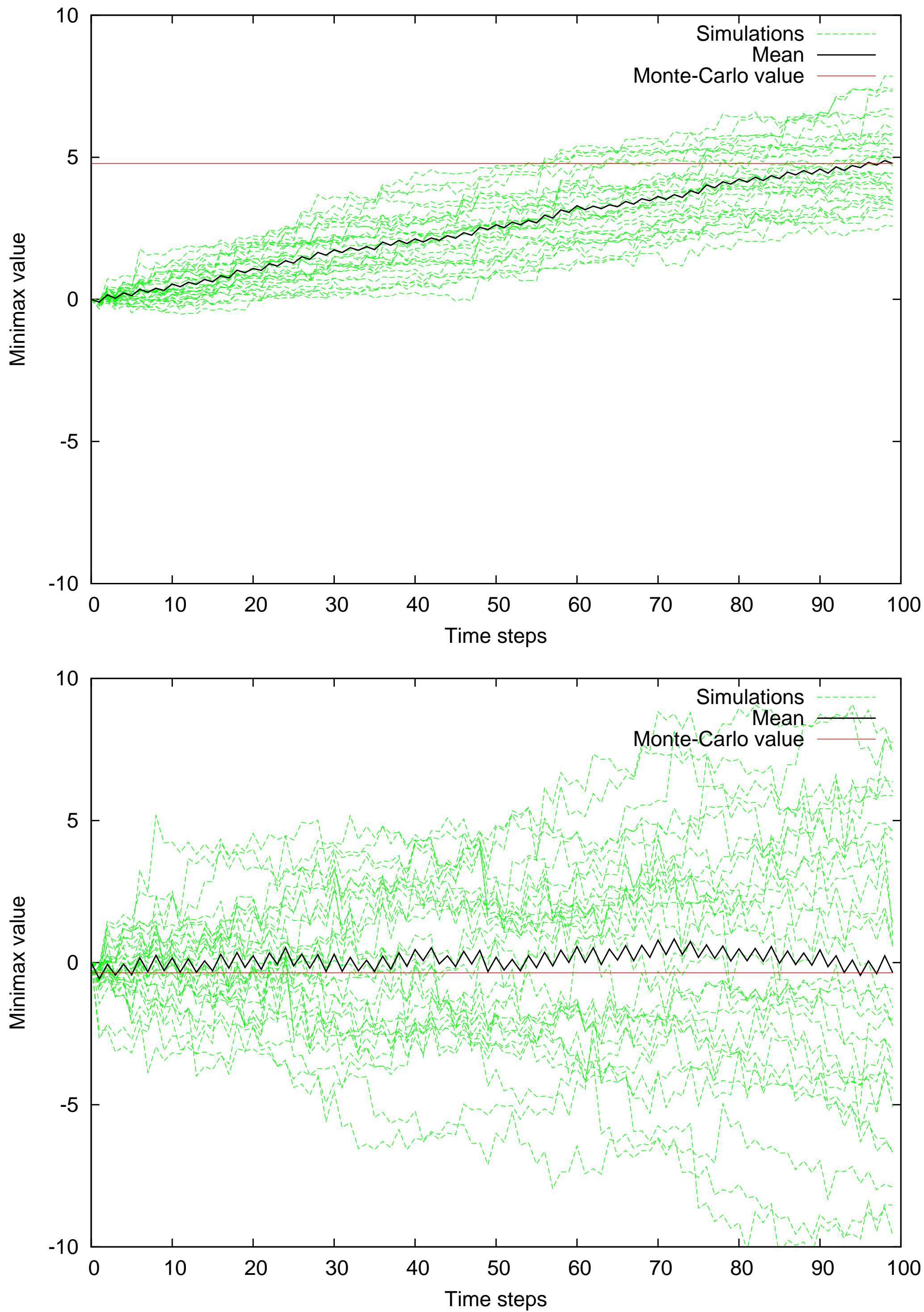

Figure 10.3: Monte-Carlo simulation in an artificial two-player game. 30 simulations of 100 time steps were executed from an initial state with minimax value 0. Each player selects moves imperfectly during simulation, with an error that is exponentially distributed with respect to the minimax value, with rate parameters λ_1 and λ_2 respectively. a) The simulation players are strong but imbalanced: $\lambda_1 = 10, \lambda_2 = 5$, b) the simulation players are weak but balanced: $\lambda_1 = 2, \lambda_2 = 2$. The Monte-Carlo value of the weak, balanced simulation players is significantly more accurate.

minimax values $V^*(s)$.

When the number of simulations N is large, the mean squared error is dominated by the bias of the simulation policy with respect to the minimax value, $V^*(s) - \mathbb{E}_{\pi_p}[z|s]$, and the variance of the estimate (i.e. the error caused by only seeing a finite number of simulations) can be ignored. Our objective is to minimise the mean squared bias, averaged over the distribution of states $\rho(s)$ that are evaluated during Monte-Carlo search.

$$p^* = \operatorname*{argmin}_p \mathbb{E}_\rho \left[\left(V^*(s) - \mathbb{E}_{\pi_p}[z|s] \right)^2 \right] \tag{10.1}$$

where $\mathbb{E}_\rho$ denotes the expectation over the distribution of actual states $\rho(s)$, and $\mathbb{E}_{\pi_p}$ denotes the expectation over simulations with policy π_p.

In real-world domains, knowledge of the true minimax values is not available. In practice, we use the values $\hat{V}^*(s)$ computed by deep Monte-Carlo tree searches, which converge on the minimax value in the limit (Kocsis and Szepesvari, 2006), as an approximation $\hat{V}^*(s) \approx V^*(s)$.

At every time-step t, each player's move incurs some error $\delta_t = V^*(s_{t+1}) - V^*(s_t)$ with respect to the minimax value $V^*(s_t)$. We will describe a policy with a small error as *strong*, and a policy with a small expected error as *balanced*. Intuitively, a strong policy makes few mistakes, whereas a balanced policy allows many mistakes, as long as they cancel each other out on average. Formally, we define the strength $J(p)$ and k-step imbalance $B_k(p)$ of a policy π_p,

$$J(p) = \mathbb{E}_\rho \left[\mathbb{E}_{\pi_p} \left[\delta_t^2 | s_t = s \right] \right] \tag{10.2}$$

$$\begin{aligned} B_k(p) &= \mathbb{E}_\rho \left[\left(\mathbb{E}_{\pi_p} \left[\sum_{j=0}^{k-1} \delta_{t+j} | s_t = s \right] \right)^2 \right] \qquad (10.3) \\ &= \mathbb{E}_\rho \left[\left(\mathbb{E}_{\pi_p} \left[V^*(s_{t+k}) - V^*(s_t) | s_t = s \right] \right)^2 \right] \end{aligned}$$

We consider two choices of k in this paper. The *two-step imbalance* $B_2(p)$ is specifically appropriate to two-player games. It allows errors by one player, as long as they are on average cancelled out by the other player's error on the next move. The *full imbalance* B_∞ allows errors to be committed at any time, as long as they cancel out by the time the game is finished. It is exactly equivalent to the mean squared bias that we are aiming to optimise in Equation 10.1,

$$\begin{aligned} B_\infty(p) &= \mathbb{E}_\rho \left[\left(\mathbb{E}_{\pi_p} \left[V^*(s_T) - V^*(s) | s_t = s \right] \right)^2 \right] \\ &= \mathbb{E}_\rho \left[\left(\mathbb{E}_{\pi_p}[z | s_t = s] - V^*(s) \right)^2 \right] \qquad (10.4) \end{aligned}$$

where s_T is the terminal state with outcome z. Thus, while the direct performance of a policy is largely determined by its strength, the performance of a policy in Monte-Carlo simulation is determined by its full imbalance.

If the simulation policy is optimal, $\mathbb{E}_{\pi_p}[z|s] = V^*(s)$, then perfect balance is achieved, $B_\infty(p) = 0$. This suggests that optimising the strength of the simulation policy, so that individual moves become closer to optimal, may be sufficient to achieve balance. However, even small errors can rapidly accumulate over the course of long simulations if they are not well-balanced. It is more important to maintain a diverse spread of simulations, which are on average representative of strong play, than for individual moves or simulations to be low in error. Figure 10.3 shows a simple scenario in which the error of each player is i.i.d and exponentially distributed with rate parameters λ_1 and λ_2 respectively. A weak, balanced simulation policy ($\lambda_1 = 2, \lambda_2 = 2$) provides a much more accurate Monte-Carlo evaluation than a strong, imbalanced simulation policy ($\lambda_1 = 10, \lambda_2 = 5$).

In large domains it is not usually possible to achieve perfect strength or perfect balance, and some approximation is required. Our hypothesis is that very different approximations will result from optimising balance as opposed to optimising strength, and that optimising balance will lead to significantly better Monte-Carlo performance.

To test this hypothesis, we implement two algorithms that maximise the strength of the simulation policy, using apprenticeship learning and reinforcement learning respectively. We then develop two new algorithms that minimise the imbalance of the simulation policy by gradient descent. Finally, we compare the performance of these algorithms in 5×5 and 6×6 Go.

10.4 Softmax Policy

We use a *softmax policy* to parameterise the simulation policy,

$$\pi_p(s, a) = \frac{e^{\phi(s,a)^T p}}{\sum_b e^{\phi(s,b)^T p}} \tag{10.5}$$

where $\phi(s, a)$ is a vector of features for state s and action a, and p is a corresponding parameter vector indicating the preference of the policy for each feature.

The softmax policy can represent a wide range of stochasticity in different positions, ranging from near deterministic policies with large preference disparities, to uniformly random policies with equal preferences. The level of stochasticity is very significant in Monte-Carlo simulation: if the policy is too deterministic then there is no diversity and Monte-Carlo simulation cannot improve the policy; if the policy is too random then the overall accuracy of the simulations is diminished. Existing paradigms for machine learning, such as reinforcement learning and supervised learning, do not explicitly control this stochasticity. One of the motivations for simulation balancing is to tune the level of stochasticity to a suitable level in each position.

We will need the gradient of the log of the softmax policy, with respect to the policy parameters,

$$
\begin{aligned}
\nabla_p \log \pi_p(s,a) &= \nabla_p \log e^{\phi(s,a)^T p} - \nabla_p \log \left(\sum_b e^{\phi(s,b)^T p} \right) \\
&= \nabla_p \left(\phi(s,a)^T p \right) - \frac{\nabla_p \sum_b e^{\phi(s,b)^T p}}{\sum_b e^{\phi(s,b)^T p}} \\
&= \phi(s,a) - \frac{\sum_b \phi(s,b) e^{\phi(s,b)^T p}}{\sum_b e^{\phi(s,b)^T p}} \\
&= \phi(s,a) - \sum_b \pi_p(s,b)\phi(s,b) \qquad (10.6)
\end{aligned}
$$

which is the difference between the observed feature vector and the expected feature vector. We denote this gradient by $\psi(s,a)$.

10.5 Optimising Strength

We consider two algorithms for optimising the strength of a simulation policy, by supervised learning and reinforcement learning respectively.

10.5.1 Apprenticeship Learning

Our first algorithm optimises the strength of the simulation policy by apprenticeship learning. The aim of the algorithm is simple: to find a simulation policy that behaves as closely as possible to a given expert policy $\mu(s,a)$.

We consider a data-set of L training examples (s_l, a_l^*) of actions a_l^* selected by expert policy μ in positions s_l. The apprenticeship learning algorithm finds parameters maximising the likelihood, $\mathcal{L}(p)$, that the simulation policy $\pi_p(s,a)$ produces the actions a_l^*[2]. This is achieved by gradient ascent of the log likelihood,

$$
\begin{aligned}
\mathcal{L}(p) &= \prod_{l=1}^{L} \pi_p(s_l, a_l^*) \\
\log \mathcal{L}(p) &= \sum_{l=1}^{L} \log \pi_p(s_l, a_l^*) \\
\nabla_p \log \mathcal{L}(p) &= \sum_{l=1}^{L} \nabla_p \log \pi_p(s_l, a_l^*) \\
&= \sum_{l=1}^{L} \psi(s_l, a_l^*) \qquad (10.7)
\end{aligned}
$$

This leads to a stochastic gradient ascent algorithm, in which each training example (s_l, a_l^*) is used to update the policy parameters, with step-size α,

[2] These parameters are the log of the ratings that maximise likelihood in a generalised Bradley-Terry model (Coulom, 2007).

$$\Delta p = \alpha \psi(s_l, a_l^*) \tag{10.8}$$

10.5.2 Policy Gradient Reinforcement Learning

Our second algorithm optimises the strength of the simulation policy by reinforcement learning. The objective is to maximise the expected cumulative reward from start state s. *Policy gradient* algorithms adjust the policy parameters p by gradient ascent, so as to find a local maximum for this objective.

We define $\mathcal{X}(s)$ to be the set of possible games $\xi = (s_1, a_1, ..., s_T, a_T)$ of states and actions, starting from $s_1 = s$. The policy gradient can then be expressed as an expectation over game outcomes $z(\xi)$,

$$\begin{aligned}
\mathbb{E}_{\pi_p}[z|s] &= \sum_{\xi \in \mathcal{X}(s)} Pr(\xi) z(\xi) \\
\nabla_p \mathbb{E}_{\pi_p}[z|s] &= \sum_{\xi \in \mathcal{X}(s)} \nabla_p (\pi_p(s_1, a_1) ... \pi_p(s_T, a_T)) z(\xi) \\
&= \sum_{\xi \in \mathcal{X}(s)} \pi_p(s_1, a_1) ... \pi_p(s_T, a_T) \\
&\quad \left(\frac{\nabla_p \pi_p(s_1, a_1)}{\pi_p(s_1, a_1)} + ... + \frac{\nabla_p \pi_p(s_T, a_T)}{\pi_p(s_T, a_T)} \right) z(\xi) \\
&= \mathbb{E}_{\pi_p} \left[z \sum_{t=1}^{T} \nabla_p \log \pi_p(s_t, a_t) \right] \\
&= \mathbb{E}_{\pi_p} \left[z \sum_{t=1}^{T} \psi(s_t, a_t) \right]
\end{aligned} \tag{10.9}$$

The policy parameters are updated by stochastic gradient ascent with step-size α, after each game, leading to a REINFORCE algorithm (Williams, 1992),

$$\Delta p = \frac{\alpha z}{T} \sum_{t=1}^{T} \psi(s_t, a_t) \tag{10.10}$$

10.6 Optimising Balance

We now introduce two algorithms for minimising the full imbalance and two-step imbalance of a simulation policy. Both algorithms learn from $\hat{V}^*(s)$, an approximation to the minimax value function constructed by deep Monte-Carlo search.

10.6.1 Policy Gradient Simulation Balancing

Our first simulation balancing algorithm minimises the full imbalance B_∞ of the simulation policy, by gradient descent. The gradient breaks down into two terms. The *bias*, $b(s)$, indicates the direction

in which we need to adjust the mean outcome from state s: e.g. does black need to win more or less frequently, in order to match the minimax value? The *policy gradient*, $g(s)$, indicates how the mean outcome from state s can be adjusted, e.g. how can the policy be modified, so as to make black win more frequently?

$$
\begin{aligned}
b(s) &= V^*(s) - \mathbb{E}_{\pi_p}[z|s] \\
g(s) &= \nabla_p \mathbb{E}_{\pi_p}[z|s] \\
B_\infty(p) &= \mathbb{E}_\rho\left[b(s)^2\right] \\
\nabla_p B_\infty(p) &= \mathbb{E}_\rho\left[b^2\right] = -2\mathbb{E}_\rho\left[b(s)g(s)\right]
\end{aligned} \tag{10.11}
$$

We estimate the bias, $\hat{b}(s)$, by sampling M simulations $\mathcal{X}_M(s)$ from state s,

$$
\hat{b}(s) = \hat{V}^*(s) - \frac{1}{M}\sum_{\xi \in \mathcal{X}_M(s)} z(\xi) \tag{10.12}
$$

We estimate the policy gradient, $\hat{g}(s)$, by sampling N additional simulations $\mathcal{X}_N(s)$ from state s and using Equation 10.9,

$$
\hat{g}(s) = \sum_{\xi \in \mathcal{X}_N(s)} \frac{z(\xi)}{NT}\sum_{t=1}^{T}\psi(s_t, a_t) \tag{10.13}
$$

In general $\hat{b}(s)$ and $\hat{g}(s)$ are correlated, and we need two independent samples to form an unbiased estimate of their product (Algorithm 1). This provides a simple stochastic gradient descent update, $\Delta p = \alpha \hat{b}(s)\hat{g}(s)$.

Algorithm 4 Policy Gradient Simulation Balancing

$p \leftarrow 0$
for all $s_1 \in$ training set **do**
  $V \leftarrow 0$
  for $i = 1$ to M **do**
    simulate $(s_1, a_1, ..., s_T, a_T; z)$ using π_p
    $V \leftarrow V + \frac{z}{M}$
  end for
  $g \leftarrow 0$
  for $j = 1$ to N **do**
    simulate $(s_1, a_1, ..., s_T, a_T; z)$ using π_p
    $g \leftarrow g + \frac{z}{NT}\sum_{t=1}^{T}\psi(s_t, a_t)$
  end for
  $p \leftarrow p + \alpha(V^*(s_1) - V)g$
end for

10.6.2 Two-Step Simulation Balancing

Our second simulation balancing algorithm minimises the two-step imbalance B_2 of the simulation policy, by gradient descent. The gradient can again be expressed as a product of two terms. The

two-step bias, $b_2(s)$, indicates whether black needs to win more or less games, to achieve balance between time t and time $t+2$. The *two-step policy gradient*, $g_2(s)$, indicates the direction in which the parameters should be adjusted, in order for black to improve his evaluation at time $t+2$.

$$
\begin{aligned}
b_2(s) &= V^*(s) - \mathbb{E}_{\pi_p}[V^*(s_{t+2})|s_t = s] \\
g_2(s) &= \nabla_p \mathbb{E}_{\pi_p}[V^*(s_{t+2})|s_t = s] \\
B_2(s)(p) &= \mathbb{E}_\rho\left[b_2(s)^2\right] \\
\nabla_p B_2(s)(p) &= -2\mathbb{E}_\rho\left[b_2(s) g_2(s)\right] \qquad (10.14)
\end{aligned}
$$

The two-step policy gradient can be derived by applying the product rule,

$$
\begin{aligned}
g_2(s) &= \nabla_p \mathbb{E}_{\pi_p}[V^*(s_{t+2})|s_t = s] \\
&= \nabla_p \sum_a \sum_b \pi_p(s_t, a)\pi_p(s_{t+1}, b) V^*(s_{t+2}) \\
&= \sum_a \sum_b \pi_p(s_t, a)\pi_p(s_{t+1}, b) V^*(s_{t+2}) \\
&\quad \left(\frac{\nabla_p \pi_p(s_t, a)}{\pi_p(s_t, a)} + \frac{\nabla_p \pi_p(s_{t+1}, b)}{\pi_p(s_{t+1}, b)} \right) \\
&= \mathbb{E}_{\pi_p}\left[V^*(s_{t+2})(\psi(s_t, a_t) + \psi(s_{t+1}, a_{t+1}))|s_t = s\right] \qquad (10.15)
\end{aligned}
$$

Both the two-step bias $b_2(s)$ and the policy gradient $g_2(s)$ can be calculated analytically, with no requirement for simulation, leading to a simple gradient descent algorithm (Algorithm 2), $\Delta p = \alpha b_2(s) g_2(s)$.

Algorithm 5 Two Step Simulation Balancing

```
p ← 0
for all s_1 ∈ training set do
    V ← 0, g_2 ← 0
    for all a_1 ∈ legal moves from s_1 do
        s_2 = s_1 ∘ a_1
        for all a_2 ∈ legal moves from s_2 do
            s_3 = s_2 ∘ a_2
            p = π_p(s_1, a_1) π_p(s_2, a_2)
            V ← V + p V*(s_3)
            g_2 ← g_2 + p V*(s_3)(ψ(s_1, a_1) + ψ(s_2, a_2))
        end for
    end for
    p ← p + α(V*(s_1) − V) g_2
end for
```

10.7 Experiments in Computer Go

We applied each of our algorithms to learn a simulation policy for 5×5 and 6×6 Go. For the apprenticeship learning and simulation balancing algorithms, we constructed a data-set of positions

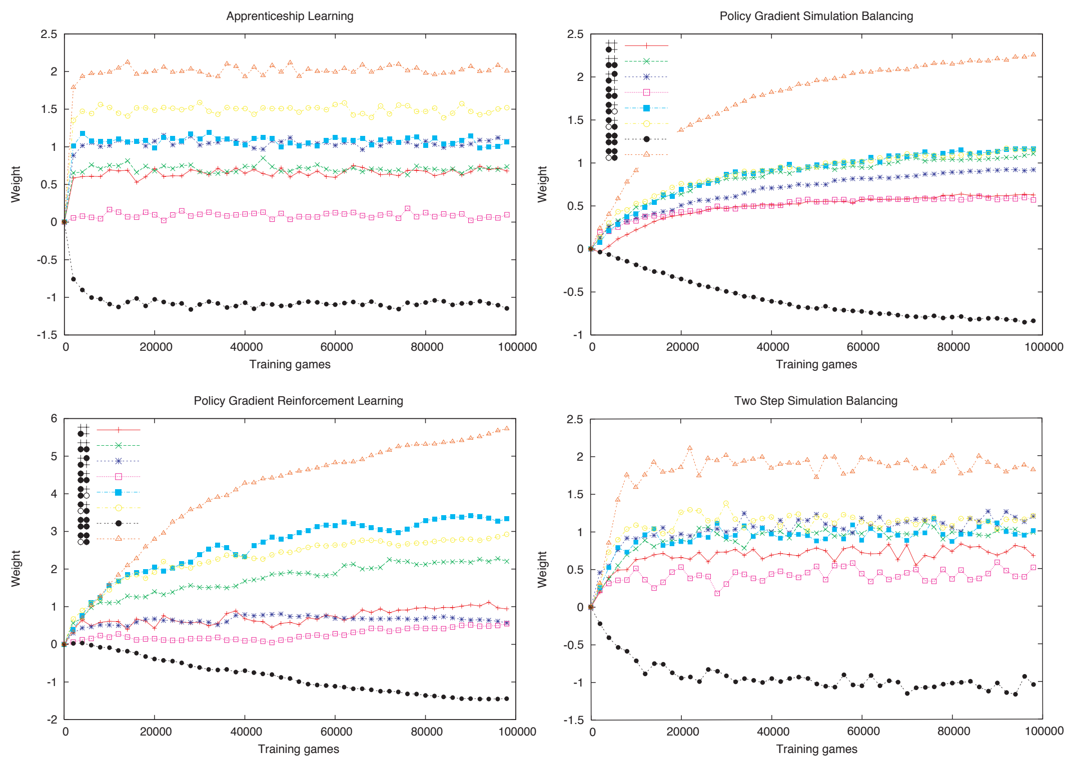

Figure 10.4: Weight evolution for the 2×2 local shape features: (top left) apprenticeship learning, (top right) policy gradient simulation balancing, (bottom left) policy gradient reinforcement learning, (bottom right) two-step simulation balancing.

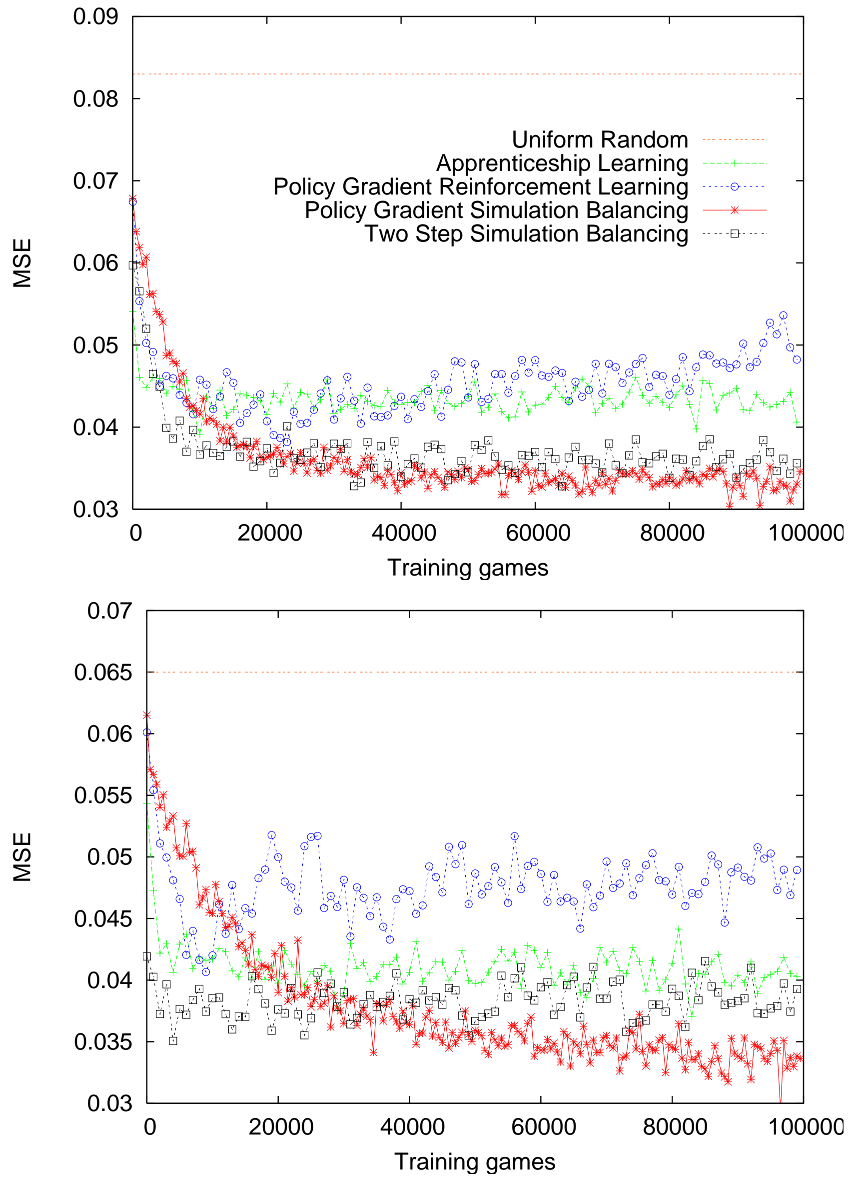

Figure 10.5: Monte-Carlo evaluation accuracy of different simulation policies in 5×5 Go (top) and 6×6 Go (bottom). Each point is the mean squared error over 1000 positions, between Monte-Carlo values from 100 simulations, and deep rollouts using the Go program *Fuego*.

from 1000 games of randomly played games. We used the open source Monte-Carlo Go program *Fuego* to evaluate each position, using a deep search of 10000 simulations from each position. The results of the search are used to approximate the optimal value $\hat{V}^*(s) \approx V^*(s)$. For the two-step simulation balancing algorithm, a complete tree of depth 2 was also constructed from each position in the data-set, and each leaf position evaluated by a further 2000 simulations. These leaf evaluations are used in the two-step simulation balancing algorithm, to approximate the optimal value after each possible move and response.

We parameterise the softmax policy (Equation 10.5) with *local shape features* (Silver et al., 2007). Each of these features has a value of 1 if it matches a specific configuration of stones within a square region of the board, and 0 otherwise. The feature vector $\phi(s, a)$ contains local shape features for all possible configurations of stones, in all 1×1 and 2×2 squares of the board, for the position following action a in state s. Two different sets of symmetries are exploited by weight-sharing. *Location dependent* weights are shared between equivalent features, based on rotational, reflectional and colour inversion symmetries. *Location independent* weights are shared between these same symmetries, but also between translations of the same configuration. Combining both sets of weights results in 107 unique parameters for the simulation policy, each indicating a preference for a particular pattern.

10.7.1 Balance of shapes

We trained the simulation policy using 100000 training games of 5×5 Go, starting with initial weights of zero. The weights learnt by each algorithm are shown in Figure 10.4. All four algorithms converged on a stable solution. They quickly learnt to prefer capturing moves, represented by a positive preference for the location independent 1×1 feature, and to prefer central board intersections over edge and corner intersections, represented by the location dependent 2×2 features. Furthermore, all four algorithms learnt patterns that correspond to basic Go knowledge: e.g. the *turn* shape attained the highest preference, and the *dumpling* and *empty triangle* shapes attained the lowest preference.

In our experiments, the policy gradient reinforcement learning algorithm found the most deterministic policy, with a wide spectrum of weights. The apprenticeship learning algorithm converged particularly quickly, to a moderate level of determinism. The two simulation balancing algorithms found remarkably similar solutions, with the turn shape highly favoured, the dumpling shape highly disfavoured, and a stochastic balance of preferences over other shapes.

10.7.2 Mean squared error

We measured the accuracy of the simulation policies every 1000 training games by selecting 1000 random positions from an independent test-set, and performing a Monte-Carlo evaluation from 100 simulations. The mean squared error (MSE) of the Monte-Carlo values, compared to the deep search

	5x5		6x6	
Simulation Policy	Direct	MC	Direct	MC
Uniform random	0	1031	0	970
Apprenticeship learning	671	1107	569	1047
Policy gradient RL (20k)	816	1234	531	1104
Policy gradient RL (100k)	**947**	1159	**850**	1023
Policy gradient sim. balancing	719	**1367**	658	**1301**
Two-step simulation balancing	720	1357	444	1109
GnuGo 3.7.10 (level 10)	1376	N/A	1534	N/A
Fuego simulation policy	356	689	374	785

Table 10.2: Elo rating of simulation policies in 5×5 Go and 6×6 Go tournaments. The first column shows the performance of the simulation policy when used directly. The second column shows the performance of a simple Monte-Carlo search using the simulation policy. Each program played a minimum of 8000 games, and bayesian Elo ratings were computed from the results, with a 95% confidence interval of approximately +/-10 points.

values, is shown in Figure 10.5, for 5×5 and 6×6 Go.

All four algorithms significantly reduced the evaluation error compared to the uniform random policy. The simulation balancing algorithms achieved the lowest error, with less than half the MSE of the uniform random policy. The reinforcement learning algorithm initially reduced the MSE, but then bounced after 20,000 steps and started to increase the evaluation error. This suggests that the simulation policy became too deterministic, specialising to weights that achieve maximum strength, rather than maintaining a good balance. The apprenticeship learning algorithm quickly learnt to reduce the error, but then converged on a solution with significantly higher MSE than the simulation balancing algorithms. Given a source of expert evaluations, this suggests that simulation balancing can make more effective use of this knowledge, in the context of Monte-Carlo simulation, than a supervised learning approach.

10.7.3 Performance in Monte-Carlo search

In our final experiment, we measured the performance of each learnt simulation policy in a Monte-Carlo search algorithm. We ran a tournament between players based on each simulation policy, consisting of at least 5000 matches for every player. Two players were included for each simulation policy: the first played moves directly according to the simulation policy; the second used the simulation policy in a Monte-Carlo search algorithm. Our search algorithm was intentionally simplistic: for every legal move a, we simulated 100 games starting with a, and selected the move with the greatest number of wins. We included two simulation policies for the policy gradient reinforcement learning algorithm, firstly using the parameters that maximised performance (100k games of training), and secondly using the parameters that minimised MSE (20k and 10k games of training in 5×5 and 6×6 Go respectively). The results are shown in Table 1.

When the simulation policies were used directly, policy gradient RL (100k) was by far the strongest, around 200 Elo points stronger than simulation balancing. However, when used as a

Monte-Carlo policy, simulation balancing was much stronger, 200 Elo points above policy gradient RL (100k), and almost 300 Elo stronger than apprenticeship learning.

The two simulation balancing algorithms achieved similar performance in 5×5 Go, suggesting that it suffices to balance the errors from consecutive moves, and that there is little to be gained by balancing complete simulations. However, in the more complex game of 6×6 Go, Monte-Carlo simulation balancing performed significantly better than two-step simulation balancing.

Finally, we compared the performance of our Monte-Carlo search to GnuGo, a deterministic Go program with sophisticated, handcrafted knowledge and specialised search algorithms. Using the policy learnt by simulation balancing, our simple one-ply search algorithm achieved comparable strength to GnuGo. In addition, we compared the performance of the *Fuego* simulation policy, which is based on the well-known *MoGo* patterns, handcrafted for Monte-Carlo search on larger boards. Surprisingly, the Fuego simulation policy performed poorly, suggesting that handcrafted patterns do not generalise well to different board sizes.

10.8 Conclusions

In this chapter we have presented a new paradigm for simulation balancing in Monte-Carlo search. Unlike supervised learning and reinforcement learning approaches, our algorithms can balance the level of stochasticity to an appropriate level for Monte-Carlo search. They are able to exploit deep search values more effectively than supervised learning methods, and they maximise a more relevant objective function than reinforcement learning methods. Unlike hill-climbing or handcrafted trial and error, our algorithms are based on an analytical gradient based only on the current position, allowing parameters to be updated with minimal computation. Finally, we have demonstrated that our algorithms outperform prior methods in small board Go.

We are currently investigating methods for scaling up the simulation balancing paradigm both to larger domains, using *actor-critic* methods to reduce the variance of the policy gradient estimate; and to more sophisticated Monte-Carlo search algorithms, such as UCT (Kocsis and Szepesvari, 2006). In complex domains, the quality of the minimax approximation $\hat{V}^*(s)$ can affect the overall solution. One natural idea is to use the learned simulation policy in Monte-Carlo search, and generate new deep search values, in an iterative cycle.

One advantage of apprenticeship learning over simulation balancing is that it optimises a convex objective function. This suggests that the two methods could be combined: first using apprenticeship learning to find a global optimum, and then applying simulation balancing to find a local, balanced optimum.

For clarity of presentation we have focused on deterministic two-player games with terminal outcomes. However, all of our algorithms generalise directly to stochastic environments and intermediate rewards.

Chapter 11

Conclusions

11.1 The Future of Computer Go

For the last 30 years, Computer Go programs have evaluated positions by using handcrafted heuristics that are based on human expert knowledge of shapes, patterns and rules. However, professional Go players often play moves according to intuitive feelings that are hard to express or quantify. Precisely encoding their knowledge into machine-understandable rules has proven to be a dead-end: a classic example of the knowledge acquisition bottleneck in AI. Furthermore, traditional search algorithms, which are based on these handcrafted heuristics, cannot cope with the enormous search space and branching factor in the game of Go, and are unable to make effective use of additional computation time. This approach has led to Go programs that are at best comparable to weak-amateur level humans.

In contrast, simulation-based search requires no human knowledge in order to understand a position. Instead, positions are evaluated from the outcome of thousands of simulated games of self-play from that position. These simulated games are progressively refined to prioritise the selection of positions with promising evaluations. Over the course of many simulations, attention is focused selectively on narrow regions of the search space that are correlated with successful outcomes. Unlike traditional search algorithms, this approach scales well with additional computation time.

The performance of Monte-Carlo tree search algorithms speak for themselves. On the Computer Go Server, using 9×9, 13×13 and 19×19 board sizes, traditional search programs are rated at around 1800 Elo, whereas Monte-Carlo programs, enhanced by RAVE and domain knowledge, are rated at around 2500 Elo using standard hardware[1] (see Table 4.1). On the Kiseido Go Server, on full-size boards against human opposition, traditional search programs have reached 5-6 Kyu, whereas the best Monte-Carlo programs are rated at 1-*dan* using standard hardware (see Table 4.2). The top programs are now competitive with top human professionals at 9×9 Go, and are winning handicap games against top human professionals at 19×19 Go.

However, it is the scalability of simulation-based search that is perhaps most exciting. In the

[1] A difference of 700 Elo corresponds to a 99% winning rate.

Go program *Mogo*, every doubling in computation power leads to an increase in playing strength of approximately 100 Elo points in 13×13 Go (see Figure 9.4), and perhaps even more in 19×19 Go. So in a sense it is already all over for humankind. Even if Computer Go researchers were all to retire, we can rest assured that Moore's law will dramatically improve the performance of existing Computer Go programs; only a few more doublings may be required before we have a computer world champion. However, Computer Go research is more active than ever. Simulation-based search is in its infancy, and we can expect exciting new developments over the next few years, suggesting that the inevitable may happen sooner than anyone expects.

11.2 The Future of Sequential Decision-Making

In this thesis we have introduced a general framework for simulation-based search. We have developed the temporal-difference learning algorithm into a high performance search algorithm (see Chapter 6). We have combined both long and short-term memories together, so as to represent both general knowledge about the whole domain, and specialised knowledge about the current situation (see Chapter 7. We have applied these methods to Monte-Carlo tree search, so as to reuse local search trees (see Chapter 8, to incorporate prior knowledge, and to provide a rapid generalisation between subtrees (see Chapter 9).

The heuristic MC-RAVE algorithm has proven particular successful in Computer Go. While this particular algorithm exploits Go specific properties, such as the *all-moves-as-first* heuristic, our general framework for simulation-based search is applicable to a much wider range of applications. The key contributions of this thesis are to combine simulation-based search with state abstraction, with bootstrapping, and with long-term learning. Each of these ideas is very general: given an appropriate model and state representation, they could be applied to any large decision-making problem. The Dyna-2 algorithm brings all of these ideas together, providing a general-purpose framework for learning and search in very large worlds.

In real-world planning and decision-making problems, most actions have long-term consequences, leading to enormous search-spaces that are intractable to traditional search algorithms. Furthermore, also just like Go, in the majority of these problems, expert knowledge is unavailable or unreliable. Simulation-based search offers a hope for new progress in these formidable problems. Three years ago, experts agreed that *dan*-strength Computer Go programs were 50 years away; now they are a reality. But perhaps the revolution is just beginning: a revolution in how we search, plan and act in challenging domains.

Bibliography

Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems 19*, pages 1–8.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2-3):235–256.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann.

Baxter, J., Tridgell, A., and Weaver, L. (1998). Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99.

Billings, D., Castillo, L. P., Schaeffer, J., and Szafron, D. (1999). Using probabilistic knowledge and simulation to play poker. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 697–703.

Bouzy, B. (2004). Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In *4th Computers and Games conference*, pages 67–80.

Bouzy, B. (2005a). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, 175(1):14–27.

Bouzy, B. (2005b). Move pruning techniques for Monte-Carlo Go. In *11th Advances in Computer Games conference*, pages 104–119.

Bouzy, B. (2006). History and territory heuristics for monte-Carlo Go. *New Mathematics and Natural Computation*, 2(2).

Bouzy, B. and Helmstetter, B. (2003). Monte-Carlo Go developments. In *10th Advances in Computer Games conference*, pages 159–174.

Bruegmann, B. (1993). Monte-Carlo Go. *http://www.cgl.ucsf.edu/go/Programs/Gobble.html*.

Buro, M. (1999). From simple features to sophisticated evaluation functions. In *First International Conference on Computers and Games*, pages 126–145.

Campbell, M., Hoane, A., and Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, 134:57–83.

Chang, H., Fu, M., Hu, J., and Marcus, S. (2005). An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139.

Chaslot, G., Winands, M., Szita, I., and van den Herik, H. (2008a). Parameter tuning by the cross-entropy method. In *Proceedings of the 8th European Workshop on Reinforcement Learning*.

Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., and Bouzy, B. (2007). Progressive strategies for Monte-Carlo tree search. In *Proceedings of the 10th Joint Conference on Information Sciences*, pages 655–661.

Chaslot, G., Winands, M., and van den Herik, J. (2008b). Parallel monte-carlo tree search. In *Proceedings of the 6th International Conference on Computer and Games*.

Choi, J., Laibson, D., Madrian, B., and Metrick, A. (2007). Reinforcement learning and savings behavior. Technical Report ICF Working Paper 09-01, Yale.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computer and Games, 2006-05-29*, pages 72–83.

Coulom, R. (2007). Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop*.

Culberson, J. and Schaeffer, J. (1996). Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081*, pages 402–416. Springer-Verlag.

Dahl, F. (1999). Honte, a Go-playing program using neural nets. In *Machines that learn to play games*, pages 205–223. Nova Science.

Dailey, D. (2008). 9x9 scalability study. *http://cgos.boardspace.net/study/index.html*.

Davies, S., Ng, A., and Moore, A. (1998). Applying online-search to reinforcement learning. In *15th European Conference on Artificial Intelligence*, pages 753–760.

Dayan, P. (1994). TD(λ) converges with probability 1. *Machine Learning*, 14:295–301.

Enderton, H. (1991). The Golem Go program. Technical report, School of Computer Science, Carnegie-Mellon University.

Enzenberger, M. (1996). The integration of a priori knowledge into a go playing neural network. *http://www.cs.ualberta.ca/ emarkus/neurogo/neurogo1996.html*.

Enzenberger, M. (2003). Evaluation in Go by a neural network using soft segmentation. In *10th Advances in Computer Games Conference*, pages 97–108.

Ernst, D., Glavic, M., Geurts, P., and Wehenkel, L. (2005). Approximate value iteration in the reinforcement learning context. application to electrical power system control. *International Journal of Emerging Electric Power Systems*, 3(1).

Fürnkranz, J. (2001). Machine learning in games: A survey. In *Machines That Learn to Play Games, chapter 2*, pages 11–59. Nova Science Publishers.

Gelly, S., Hoock, J., Rimmel, A., Teytaud, O., and Kalemkarian, Y. (2008). The parallelization of Monte-Carlo planning. In *6th International Conference in Control, Automation and Robotics*.

Gelly, S. and Silver, D. (2007). Combining online and offline learning in UCT. In *17th International Conference on Machine Learning*, pages 273–280.

Gelly, S. and Silver, D. (2008). Achieving master level play in 9 x 9 computer Go. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*, pages 1537–1540.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

Graepel, T., Kruger, M., and Herbrich, R. (2001). Learning on graphs in the game of go. In *Artificial Neural Networks -ICANN 2001*, pages 347–352. Springer.

Harmon, A. (2003). Queen, captured by mouse; more chess players use computers for edge. *New York Times*.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107.

Haykin, S. (1996). *Adaptive Filter Theory*. Prentice Hall.

Hunter, R. (2004). Mm algorithms for generalized bradley-terry models. *The Annals of Statistics*, 32(1):384406.

Jordan, M. (1995). Why the logistic function? A tutorial discussion on probabilities and neural networks. Technical Report Computational Cognitive Science Report 9503, Massachusetts Institute of Technology.

Kearns, M., Mansour, Y., and Ng, A. (2002). A sparse sampling algorithm for near-optimal planning in large Markovian decision processes. *Machine Learning*, 49(2-3):193–208.

Kocsis, L. and Szepesvari, C. (2006). Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning*, pages 282–293.

Kohl, N. and Stone, P. (2004). Machine learning for fast quadrupedal locomotion. In *19th National Conference on Artificial Intelligence*, pages 611–616.

Matthews, C. (2003). *Shape up*. GoBase.

Mayer, H. (2007). Board representations for neural go players learning by temporal difference.

McCarthy, J. (1997). AI as sport. *Science*, 276(5318):1518–1519.

McClain, D. (2006). Once again, machine beats human champion at chess. *New York Times*.

Mechner, D. (1998). All systems go. *The Sciences*, 38(1).

Müller, M. (1995). *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich.

Müller, M. (2001). Global and local game tree search. *Information Sciences*, 3-4(135):187–206.

Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134:145–179.

Müller, M. and Enzenberger, M. (2009). Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. Technical Report TR09-08, University of Alberta, Department of Computing Science.

Nevmyvaka, Y., Feng, Y., and Kearns, M. (2006). Reinforcement learning for optimized trade execution. In *23rd International Conference on Machine learning*, pages 673–680.

Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2004). Autonomous inverted helicopter flight via reinforcement learning. In *9th International Symposium on Experimental Robotics*, pages 363–372.

Péret, L. and Garcia, F. (2004). On-line search for solving markov decision processes via heuristic sampling. In *16th Eureopean Conference on Artificial Intelligence*, pages 530–534.

Rivest, R. L. (1988). Game tree searching by min/max approximation. *Artificial Intelligence*, 34:77–96.

Rummery, G. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

Runarsson, T. and Lucas, S. (2005). Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. *IEEE Transactions on Evolutionary Computation*, 9(6).

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *EEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer Verlag.

Schaeffer, J. (2000). The games computers (and people) play. *Advances in Computers*, 50:189–266.

Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P., and Szafron, D. (1992). A world championship caliber checkers program. *Artificial Intelligence*, 53:273–289.

Schaeffer, J., Hlynka, M., and Jussila, V. (2001). Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 529–534.

Schraudolph, N., Dayan, P., and Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of Go. In *Advances in Neural Information Processing 6*, pages 817–824.

Schraudolph, N. N., Dayan, P., and Sejnowski, T. J. (2000). *Learning To Evaluate Go Positions Via Temporal Difference Methods*, volume 62, pages 77–96. Springer-Verlag.

Schultz, W., Dayan, P., and Montague, P. (1997). A neural substrate of prediction and reward. *Science*, 16:1936–1947.

Sheppard, B. (2002). World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275.

Silver, D., Sutton, R., and Müller, M. (2007). Reinforcement learning of local shape in the game of Go. In *20th International Joint Conference on Artificial Intelligence*, pages 1053–1058.

Silver, D., Sutton, R., and Müller, M. (2008). Sample-based learning and search with permanent and transient memories. In *25th International Conference on Machine Learning*, pages 968–975.

Singh, S. and Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems 9*, pages 974–982.

Singh, S., Jaakkola, T., Littman, M., and Szepesvari, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38:287–308.

Singh, S. and Sutton, R. (2004). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Stoutamire, D. (1991). *Machine Learning, Game Play, and Go*. PhD thesis, Case Western Reserve University.

Sutton, R. (1984). *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts.

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3(9):9–44.

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *7th International Conference on Machine Learning*, pages 216–224.

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044.

Sutton, R. and Barto, A. (1990). Time-derivative models of pavlovian reinforcement. *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497–537.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: an Introduction*. MIT Press.

Sutton, R., Koop, A., and Silver, D. (2007). On the role of tracking in stationary environments. In *17th International Conference on Machine Learning*, pages 871–878.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press.

Tesauro, G. (1988). Connectionist learning of expert preferences by comparison training. In *NIPS*, pages 99–106.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219.

Tesauro, G. and Galperin, G. (1996). On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing 9*, pages 1068–1074.

van der Werf, E., Uiterwijk, J., Postma, E., and van den Herik, J. (2002). Local move prediction in Go. In *3rd International Conference on Computers and Games*, pages 393–412.

Wang, Y. and Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Appendix A

Logistic Temporal Difference Learning

Many tasks are best described by a binary goal of success or failure: winning a game, solving a puzzle or achieving a goal state. These tasks can be described by the classic expected-reward formulation of reinforcement learning. The agent receives a binary reward of $r = 1$ for success, $r = 0$ for failure, and no intermediate rewards. The value function $V^\pi(s)$ is defined to be the expected total reward from board position s when following policy π. This value function also defines the *success probability* from state s,

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[\sum_{t=1}^{T} r_t | s_t = s\right] \\ &= \mathbb{E}_\pi[r_T | s_t = s] \\ &= Pr(r_T = 1 | s_t = s). \end{aligned} \tag{A.1}$$

We can form an approximation $p(s)$ to the success probability by taking a linear combination of a feature vector $\phi(s)$ and a corresponding weight vector θ. However, a linear value function, $p(s) = \phi(s)^T\theta$, is not well suited to modelling probabilities. Instead, we use the logistic function $\sigma(x) = \frac{1}{1+e^x}$ to squash the value to the desired range $[0, 1]$,

$$p(s) = \sigma(\phi(s)^T\theta). \tag{A.2}$$

A.1 Logistic Monte-Carlo Evaluation

At each time-step t of each episode i we define a random variable $Y_t^i \sim \text{Bernoulli}(p(s_t^i))$ representing the agent's prediction of success or failure from state s_t^i. This corresponds to the agent flipping a coin with probability $p(s_t^i)$ of heads, and predicting success if the coin lands on heads, and failure if it lands on tails. We seek the weights that maximise the likelihood $L(\theta)$ that all predictions across

all episodes are correct,

$$L(\theta) = \prod_{i=1}^{N}\prod_{t=1}^{T} Pr(Y_t^i = z_i|\theta) \quad (A.3)$$

$$= \prod_{i=1}^{N}\prod_{t=1}^{T} p(s_t^i)^{z_i}(1-p(s_t^i))^{1-z_i} \quad (A.4)$$

$$\log L(\theta) = \sum_{i=1}^{N}\sum_{t=1}^{T} z_i \log p(s_t^i) + (1-z_i)\log(1-p(s_t^i)), \quad (A.5)$$

which is the (negative) cross-entropy between the success probability and the actual outcome. We maximise the log likelihood by gradient ascent, or equivalently minimise the cross-entropy by gradient descent. We proceed by applying the well-known identity for the derivative of the logistic function, $\nabla_\theta(\sigma(x)) = \sigma(x)(1-\sigma(x))$,

$$\nabla_\theta L(\theta) = \sum_{i=1}^{N}\sum_{t=1}^{T} z_i \nabla_\theta \log p(s_t^i) + (1-z_i)\nabla_\theta \log(1-p(s_t^i)) \quad (A.6)$$

$$= \sum_{i=1}^{N}\sum_{t=1}^{T} z_i \frac{p(s_t^i)(1-p(s_t^i))\phi(s_t^i)}{p(s_t^i)} - (1-z_i)\frac{p(s_t^i)(1-p(s_t^i))\phi(s_t^i)}{1-p(s_t^i)} \quad (A.7)$$

$$= \sum_{i=1}^{N}\sum_{t=1}^{T} z_i(1-p(s_t^i))\phi(s_t^i) - (1-z_i)p(s_t^i)\phi(s_t^i) \quad (A.8)$$

$$= \sum_{i=1}^{N}\sum_{t=1}^{T} (z_i - p(s_t^i))\phi(s_t^i). \quad (A.9)$$

If we sample this gradient at every time-step, using stochastic gradient ascent, then we arrive at a logistic regression algorithm, applied to the outcomes of episodes,

$$\Delta\theta = \alpha(z_i - p(s_t))\phi(s_t), \quad (A.10)$$

where α is a step-size parameter. We can also view this algorithm as a modification of Monte-Carlo evaluation (see Chapter 2) to estimate probabilities rather than expected values. We call this algorithm *logistic Monte-Carlo evaluation*.

A.2 Logistic Temporal-Difference Learning

We extend the idea of logistic regression to temporal-difference learning, by bootstrapping from successive estimates of the success probability. We introduce the *logistic temporal-difference learning* algorithm, which maximises the likelihood $M(\theta)$ that all predictions Y_t^i match the subsequent prediction at the next time-step Y_{t+1}^i,

$$
\begin{aligned}
M(\theta) &= \prod_{i=1}^{N}\prod_{t=1}^{T} Pr(Y_t^i = Y_{t+1}^i|\theta) && \text{(A.11)}\\
&= \prod_{i=1}^{N}\prod_{t=1}^{T} p(s_t^i)^{Y_{t+1}^i}(1-p(s_t^i))^{1-Y_{t+1}^i} && \text{(A.12)}\\
\log M(\theta) &= \sum_{i=1}^{N}\sum_{t=1}^{T} Y_{t+1}^i \log p(s_t^i) + (1-Y_{t+1}^i)(1-p(s_t^i)), && \text{(A.13)}
\end{aligned}
$$

which is the (negative) cross-entropy between the success probability and the subsequent prediction of success or failure. As in the standard temporal-difference learning algorithm (see Chapter 2), we derive a gradient descent algorithm by treating the subsequent prediction as a constant that does not depend on θ,

$$
\begin{aligned}
\nabla_\theta M(\theta) &= \sum_{i=1}^{N}\sum_{t=1}^{T} Y_{t+1}^i \nabla_\theta \log p(s_t^i) + (1-Y_{t+1}^i)\nabla_\theta \log(1-p(s_t^i)) && \text{(A.14)}\\
&= \sum_{i=1}^{N}\sum_{t=1}^{T} Y_{t+1}^i \frac{p(s_t^i)(1-p(s_t^i))\phi(s_t^i)}{p(s_t^i)} - (1-Y_{t+1}^i)\frac{p(s_t^i)(1-p(s_t^i))\phi(s_t^i)}{1-p(s_t^i)} && \text{(A.15)}\\
&= \sum_{i=1}^{N}\sum_{t=1}^{T} (Y_{t+1}^i - p(s_t^i))\phi(s_t^i) && \text{(A.16)}
\end{aligned}
$$

which is equal in expectation to $\sum_{i=1}^{N}\sum_{t=1}^{T}(p(s_{t+1}^i) - p(s_t^i))\phi(s_t^i)$. This leads to a stochastic gradient descent algorithm that we call *logistic TD(0)*,

$$
\Delta\theta = \alpha(p(s_{t+1}) - p(s_t))\phi(s_t) \qquad \text{(A.17)}
$$

We note that this update is identical in form to the standard, linear update for TD(0), except that the value function is now approximated by a logistic function.