

LEARNING COORDINATION STRATEGIES USING REINFORCEMENT LEARNING

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Information Technology at George Mason University

By

Myriam Z. Abramson

Director: Dr. Harry Wechsler
Professor, Department of Computer Science

Spring Semester 2003
George Mason University
Fairfax, VA

Copyright 2003 Myriam Z. Abramson
All Rights Reserved

Dedication

To my daughter Sarah

Acknowledgements

I want to acknowledge the total dedication Harry Wechsler has shown towards me during my tenure as a graduate student. With great patience, he tried to understand my often half-baked ideas and challenged them with constructive arguments. He taught me how to write coherently and communicate my ideas. He provided me with precise directions and steps to follow, which helped me complete this dissertation. More importantly, he gave me the intellectual guidance necessary to grow in my field of research. I am grateful to all my committee members for agreeing to participate in my research. I want especially to acknowledge Larry Hunter who first showed me how to conduct research and provided me with many years of interesting and stable work. I also want to thank all my employers who put up with me throughout my studies.

I am grateful to Mike Stein for sharing with me his expert knowledge of Go, stimulating discussions, and unyielding support. I am also very grateful to all the researchers who took the time to answer my questions and give me feedback without knowing me or getting anything in exchange. Last but not least, I am grateful to the Free Software Foundation for providing me with the tools to write this dissertation and Sun Microsystems for the Java language as a good mainstream alternative to Lisp.

I want to thank posthumously my father for motivating me to study, my mother for teaching me what is really important, and Dan Briggs for teaching me that it is important not to be afraid.

Of course, none of it would have been possible without the support of my daughter Sarah.

Table of Contents

	Page
List of Figures	x
List of Tables	xiii
List of Algorithms	xiv
Abstract	xvi
1 Introduction	1
1.1 Motivation and Objectives	1
1.1.1 Coordination Strategies Examples	2
1.1.2 The Coordination Strategy Problem	4
1.1.3 Why Is Coordination Hard?	5
1.2 Background	5
1.2.1 The Promise of Reinforcement Learning	5
1.2.2 Game Learning	6
1.2.3 Behavior-based Learning	7
1.3 Overview	8

2	Reinforcement Learning	9
2.1	From Dynamic Programming to Reinforcement Learning	10
2.1.1	Value Iteration	11
2.1.2	Policy Iteration	11
2.1.3	Reinforcement Learning	12
2.2	From A* to Reinforcement Learning	13
2.3	From Genetic Algorithms to Reinforcement Learning	16
2.4	Discussion	17
3	Sarsa Learning Vector Quantization (SLVQ)	18
3.1	Learning Framework	18
3.1.1	Reinforcement Learning Policies	19
3.1.2	Self-Organization and Learning Vector Quantization	21
3.1.3	Hybrid Learning	24
3.2	The SLVQ Algorithm	24
3.3	Knobs	28
3.3.1	The Number of Codebook Vectors	28
3.3.2	The Discount Factor γ	29
3.3.3	The Eligibility Trace λ	30
3.3.4	The Learning Rate α	31
3.4	Sources of Instability	32

4	Computer Go	34
4.1	The Complexity of Go	34
4.2	Game Theory vs. Game Playing	35
4.3	Feature Weighting vs. Feature Discovery	36
4.4	Look-Ahead Search	37
4.5	Plan-Based Search	38
4.6	Decomposition-Based Search	38
4.7	Proof-Number Search	39
4.8	Pattern-Based Search	40
4.9	Discussion	41
5	SLVQ Go	42
5.1	Representation and Evaluation of the Board	42
5.2	Matching	44
5.3	Experimental Methodology	46
5.3.1	Go Players	46
5.3.2	Comparison Metric	48
5.3.3	Experimental Setup	48
5.3.4	Experimental Results	49
5.3.5	Move Analysis	51

6	Continuous Control Tasks	56
6.1	Motivation	56
6.2	SLVQ with Minimum Spanning Tree Topology (SLVQ-MST)	57
6.3	The Cart Centering Problem	59
6.3.1	Problem Description	60
6.3.2	Distance Function	61
6.3.3	Empirical Evaluation	63
6.4	The Mountain Car Problem	65
6.4.1	Problem Description	65
6.4.2	Empirical Evaluation	66
6.5	Comparison with QLVQ	68
7	Intelligent Exploration	71
7.1	Motivation	71
7.2	Exploration Methods	72
7.2.1	Bonus/Penalty Approaches	74
7.2.2	Comparative Results of Bonus/Penalty Approaches	75
7.2.3	Training	76
7.3	Tabu Search Exploration for Reinforcement Learning	78
7.3.1	Meta-Level Search	79
7.3.2	Aspiration Criteria	81

7.3.3	Tabu List Strategies	82
7.3.4	Empirical Evaluation	87
7.4	Discussion	88
8	Learning by Parts	91
8.1	Related Research	92
8.2	The Task Decomposition Problem	93
8.3	The Knowledge Transfer Problem	94
8.4	Methodology	95
8.5	Motivation for a Game-Theoretical Approach to an Arbitration Function . .	96
8.5.1	Nash Equilibrium and Mixed Strategy	97
8.5.2	Stochastic Arbitration Function	99
8.6	Empirical Evaluation	100
8.6.1	Adaptive Wally	100
8.6.2	Approach	101
8.6.3	Experimental Results	101
8.7	Discussion	103
9	Conclusions	104
A	Overview of the Game of Go	107
B	Overview of Computer Go	115

Bibliography 121

List of Figures

Figure		Page
2.1	Trial-and-error cycle	10
2.2	Adaptive heuristic critic	13
3.1	Learning framework	19
3.2	On and off policies	21
3.3	SLVQ architecture	25
3.4	SLVQ adaptive heuristic critic architecture	26
3.5	Hill-climbing bias	33
5.1	Influence representation	44
5.2	Similarity of the codebook vectors	45
5.3	5x5 games against Minimax	50
5.4	7x7 games against Wally	51
5.5	7x7 games against “Heuristic”	52
5.6	7x7 games against Minimax	52
5.7	SLVQS opening against Wally	53
5.8	SLVQS vs. Wally complete game record	54
6.1	Cart centering problem	61
6.2	Cosine and Euclidean distance function comparison	62

6.3	Optimal trajectories	64
6.4	SLVQ trajectories	64
6.5	The mountain car problem	65
6.6	SLVQ codebook trajectories for the mountain car problem	67
6.7	SLVQ mountain car policy	68
6.8	QLVQ mountain car policy	70
7.1	Entropy and exploration relationship	73
7.2	Bonus factor for frequency-based exploration strategy	74
7.3	5x5 bonus/penalty exploration methods vs. Wally	76
7.4	7x7 bonus/penalty exploration methods vs. Wally	77
7.5	Self-play training with softmax in 7x7 games vs. Wally and minimax	78
7.6	Knowledge transfer from two different training methods	79
7.7	Candidate moves in life-and-death pattern	86
7.8	Exploration methods vs Minimax for pattern in Figure 7.7	88
7.9	Tabu exploration methods vs Wally on a 7x7	89
7.10	Self-play training with STS vs. Wally and Minimax	90
8.1	Learning by parts principles	96
8.2	Game theory framework for strategic games	97
9.1	Family of Algorithms	105
A.1	Territory	108
A.2	Life	109
A.3	Ko	110
A.4	Influence radiating from a stone	113

A.5 Seki 114

List of Tables

Table	Page
1.1 Types of coordination problems	3
4.1 Go vs. Chess	35
5.1 7x7 baseline results averaged over 100 games	47
5.2 Critical moves on a 5x5 board	53
7.1 Tabu lists for the first 2 moves in Figure 7.7	86
8.1 The Battle of the Sexes example	98
8.2 Overlapping decomposition	102
8.3 Comparative performance of knowledge transfer methods	103
A.1 Liberties	108
A.2 Interaction between friendly stones	111
A.3 Interaction between enemy stones	111

List of Algorithms

Algorithm	Page
2.1 Value iteration	11
2.2 Policy iteration	12
2.3 Model-based reinforcement learning	14
2.4 Direct reinforcement learning	14
2.5 LRTA*	16
2.6 Simple classifier system	17
3.1 Sarsa	22
3.2 LVQ	23
3.3 SLVQ	27
5.1 Influence Propagation Algorithm	44
6.1 SLVQ-MST	59
6.2 QLVQ	69
7.1 Tabu Search	81
7.2 STS Move selection	84
8.1 Stochastic SLVQ	100

ABSTRACT

LEARNING COORDINATION STRATEGIES

Myriam Abramson

George Mason University, 2003

Thesis Director: Dr. Harry Wechsler

How do we learn to accomplish complex tasks without knowledge of the end state? This absence of teleological explanation characterizes complex systems. Rather, intelligent behavior to accomplish complex tasks emerges from the coordination of many actions or many agents. Reinforcement learning is a general and powerful methodology to learn complex tasks by acquiring complementary behaviors from local interactions. To represent the actions of other agents, an efficient encoding of the state space must be found with a function approximator. This thesis claims that learning coordination strategies of localized behaviors can be achieved through self-organized and distributed reinforcement learning. The distributed representation found in Learning Vector Quantization (LVQ) enables reinforcement learning (RL) methods to cope with a large decision search space defined in terms of equivalence classes of input patterns like those found in complex problems. In particular, this thesis introduces S[arsa]LVQ, a new hybrid reinforcement learning algorithm, and shows its feasibility for Go, a coordination strategy game, and control tasks. As the distributed LVQ representation corresponds to a (quantized) codebook of compressed

and generalized pattern templates, the state space requirements for reinforcement methods are significantly reduced, thus decreasing the complexity of the decision space and consequently improving performance. Exploration can drastically influence the speed and convergence of this incremental algorithm. To this end, novel approaches to exploration based on tabu search (TS) are introduced. TS is a flexible memory-based search mechanism. This thesis shows how TS can be integrated with an RL algorithm like SLVQ and how to avoid cycling through previous solutions. The *learning by parts* paradigm inspired from behavioral psychology scales up SLVQ to partially observable states with a mixed-strategy approach.

Chapter 1

Introduction

make yourself empty before each move

Some natural phenomenon like a beehive or an ant colony have evolved their behavior over millions of years to survive by coordinating their actions to one another. Implicit coordination is a characteristic of intelligent and robust systems composed of simple agents¹. Coordination is one way to solve the fundamental survival problem and is also a way to solve many other complex problems. Coordination problems are characterized by the tight coupling between the elements of the problem. One way to tame this resulting complexity is by a top-down decomposition of the problem space. Complex behaviors, however, can emerge from the interactions of simple behaviors without hierarchical organization or knowledge of the end state.

1.1 Motivation and Objectives

The claim of this thesis is that learning coordination strategies of localized behaviors can be achieved through self-organized and distributed reinforcement learning. Here, strategy is viewed “not simply [...] as a sequence of actions but a prescription for choosing an

¹Simple agents, as opposed to complex agents, have only one behavior.

action”[61]. The goal of this research, in support of this claim, is to develop and analyze a self-organized, distributed reinforcement learning methodology, using the game of Go as a testbed. The relevance of this method is also shown by its successful application to control tasks such as the cart centering and the mountain car problem. The learning task includes the on-line partitioning of the problem space into equivalence classes to formulate a policy mapping states to actions. Finally, the locality of this method is extended to partially observable states with the *learning by parts* paradigm. The overarching goal of this thesis is to reflect the decentralization of the mind hypothesis[14]. As problems and models grow more complex, there is a need for decentralized methods.

The need for coordination arises with the co-dependency of single agents for survival or for the successful completion of a task. Coordination problems have been solved by modeling those dependencies[90]. A coordination algorithm is a planning algorithm for the management of different agents seeking to achieve a predefined desirable joint state. This dissertation addresses the problem of learning coordination of actions *without* a model and without knowledge of the end state in a pattern-based approach.

1.1.1 Coordination Strategies Examples

Coordination strategies are always about multiple agents and occur in adversarial or non-adversarial situations (Table 1.1). In the synchronous case where one action occurs at a time, which this dissertation addresses, the sequential decision task problem is formulated as a coordination problem involving multiple agents. The asynchronous case with concurrent actions can be reduced to a non-deterministic synchronous problem where the effects of actions are not always predictable. More than the locality of the interactions, what is common in those examples is the agent-based decision model and the absence of commu-

Table 1.1: Types of coordination problems

	asynchronous	synchronous
adversarial	predators/prey	game of Go
non-adversarial	swarms	game of Life

nication. The goal or end product is implicit in their actions. Handcrafted solutions for some of those problems have been found[42, 66].

The predators/prey pursuit problem In this classic example, the goal of the predators is to completely surround a prey. Like in the prisoner’s dilemma, the predators and the prey move simultaneously. If there is no deliberation among the predators, there must be an analytical solution for each predator based on the current positions of the prey and the other predators. Coordination is viewed as an emergent property of simple greedy agents subject to environmental constraints[42].

Swarms Simple homogenous (identical) agents produce a time-dependent behavior from local interactions. For example, the flocking behavior [66] exemplifies an equilibrium emerging from the local interactions of boids². Simple rules governing the steering behavior of boids with respect to their neighbors are enough to produce a pattern of aggregation.

Game of Go Go is the quintessential example of coordination strategy because the stones receive their meaning from the other stones on the board (see Appendix A for an overview). Unlike chess, the game starts with an empty board and, once put on the board, a stone rarely moves. There is no intrinsic value in a move by itself like in chess. As we’ll see in Chapter

²Boids are computer simulated creatures in Craig Reynolds’ distributed behavioral model.

5, this game suggests that learning coordination strategies involves representing the current state as a collection of moves, past and present, and their connectivity relationships.

Game of Life In this cellular automata game, the status of cells at time t are determined by the status of its neighbors at time $t - 1$ according to some predefined rules. Learning cellular automata rules is also a way to solve the coordination strategy problem[53].

1.1.2 The Coordination Strategy Problem

This problem encompasses the action selection problem. How to choose the right action with knowledge of only the local situation so that it works together with other actions in neighboring situations and without relying on knowledge of the end state? This thesis will argue that coordination can be learned reactively through experience and that it can lead to the emergence of higher-level skills. Problem solving is a practical skill like swimming[63] and learning problem solving skills is like learning how to swim. One indication that a problem has been correctly solved is that all the data is used to arrive at the solution, including recognizing and eliminating irrelevant data. Complex problems require sometimes the use of auxiliary operations but, in the end, all the relevant data blend harmoniously in the solution. Coordination is an essential trait of problem solving and complex problems can be reformulated as coordination problems.

The coordination problem is different from the planning problem where the object is to learn how best to map out a sequence of subgoals in order to achieve a goal state. Here, the goals are not reducible but are “induced” from the sequence of actions. For example, there is no representation of a winning board to achieve in the game of Go as in the n -tile puzzle where distance-to-goal methods such as A* are applicable.

1.1.3 Why Is Coordination Hard?

Learning coordination strategies is hard because the co-dependency between actions makes the problem intractable. In addition, there is a path-dependence situation, that is, an action influences the course of further actions. A hybrid learning framework is needed to (1) infer the relevant dependencies between actions and (2) act in accordance to those dependencies. Such a framework is described in Chapter 3. Previous approaches in distributed artificial intelligence have assumed a model of those dependencies[90] while evolutionary approaches has concentrated on learning rules of behaviors[33, 30].

1.2 Background

This thesis draws from several AI disciplines, namely reinforcement learning, game learning, and behavior-based learning.

1.2.1 The Promise of Reinforcement Learning

Reinforcement learning (RL) is our core learning methodology for coordination strategies. RL was shown to promote coordination by learning complementary behaviors without sharing information for cooperative or non-cooperative tasks[77, 32]. The update of a state-action pair value, or backup operation, transfers information from successor states in a bootstrapping process[88]. The basic stimulus/reward loop assumes an underlying learning system. With RL the learning system is the policy itself: which actions get activated/inhibited determines what gets learned (the bias) which in turn determines the policy. RL has this self-referential property of human learning. Sampling is inextricably part of

the learning process. Chapter 3 will introduce the two different models of sampling interactions in RL, on-policy and off-policy.

RL's performance in game learning has been demonstrated quite successfully with TD-Gammon[92], but not in the game of Go[74]. It has been argued that the success of TD-Gammon is less due to RL than to the coevolutionary nature of self-play triggered by the non-deterministic domain landscape of backgammon[62] and that any locally successful strategy such as hill-climbing will do. Indeed, in its simplest form, temporal difference learning is a local "hill-following" method. The beauty of RL is that an optimal policy can be found which will reduce to a local greedy strategy encapsulating look-ahead searches. Other features of backgammon, such as the number of opponents needed to defeat a non-optimal player might have contributed to the success of coevolution in this domain[68]³. Reasons for the failure of the application of TD-Gammon to Go include the lack of smoothness in the problem space and the difficulty in representing the global interactions of the stones on the board. This thesis will present a self-organized, distributed approach to address those problems.

1.2.2 Game Learning

Machine learning in games has been directed early on toward learning aspects of the evaluation function guiding the search[71]. Those aspects include the recognition of abstract features, the selection of operational features, and the weight of those features in the evaluation function. Handcrafted features provide a terrific boost to the learning process but

³The Internet Go Server (IGS) rating system mimics the coevolution process: The probability of the games and the players' rating coevolve in a bayesian way so that the likelihood of the system as a whole is maximized.

they are descriptive of the situation and do not have any prescriptive power⁴ by themselves. Learning coordination strategies, that is determining which move is appropriate given other moves, casts the learning problem in an agent-based framework instead of a parameter-based framework. This approach requires operational features which characterize the pattern of interactions and that have prescriptive power.

Evolutionary methods has had mixed success in evolving behavioral strategies for the multi-agent coordination problem in the predators/prey pursuit problem[33]. A similar issue to learning coordination strategies is found in the related problem of evolving “teams”. Evolutionary computation explores the space of behaviors and assigns credit to actions only implicitly[56]. A consequent pitfall is to learn the next action to execute only in the context of a specific task or game. Another pitfall of evolutionary computation in game learning is the danger of learning the opponent’s weaknesses instead of the game itself due to the contingency aspect of evolution or local minima issue. Competitive coevolution between adversaries has been used to address this training issue but do not replace the need for an existing strategy learning algorithm[68].

1.2.3 Behavior-based Learning

This type of learning exploits the decomposition of a task into patterns of behavior. Learning specific low-level behaviors occurs off-line and are integrated into a high-level behavior either by a planner[3] or within a subsumption architecture[50]. Those approaches solve the coordination strategy problem by selecting or filtering in the local actions that accomplish a global, high-level behavior. Chapter 8 builds on this distributed perception approach. One

⁴A feature has prescriptive power when it is operational (i.e. can be recognized easily) and provides a direction for action. For example, the safety of a group of stones (see Appendix A) is a descriptive feature. The symmetry of a formation has the prescriptive power to play at the center of the formation.

problem with those approaches has been to learn the pattern of interactive behaviors rather than handcoding the interaction network to produce a high-level behavior.

1.3 Overview

Chapter 2 introduces RL in a historical context. Its similarities and differences to other approaches are examined. Chapter 3 presents a detailed description of Sarsa Learning Vector Quantization (SLVQ), a self-organized algorithm in the reinforcement learning framework to learn the mapping of states to actions. Chapter 4 gives a brief background on Go and computer Go, with Appendix A and B providing greater details. In Chapter 5, the experimental methodology in the game of Go within the SLVQ framework is introduced along with experimental results. The goal of this thesis is not to learn to play the game of Go as a tournament class program but to learn from Go how to approach the coordination strategy problem. Chapter 6 applies the SLVQ algorithm to continuous control tasks and shows the relevance of the algorithm to other problems. Chapter 7 discusses issues in intelligent exploration for reinforcement learning and shows how tabu search methods apply to this algorithm along with some results in the game of Go. Chapter 8 presents the *learning by parts* paradigm to scale up this framework to partially observable states.

Chapter 2

Reinforcement Learning

A meijin needs no joseki.

Reinforcement learning used to describe a class of problems characterized by trial-and-error cycles such as pole balancing and illustrated by the basic feedback loop in Figure 2.1. It has been formalized as the n -armed bandit problem: given a machine with two levels that independently pays some amount of money each time a level is pulled, develop a strategy to select which lever to pull that gains a maximum payoff over time based only on past experience. In the n -armed bandit problem, the current state is abstracted away as the accumulation of rewards. Reinforcement learning has grown to describe a learning method more adequately called temporal-difference learning applied to Markovian problems. As Brooks noted[7], artificial intelligence paradigms are driven by advances in technology. The rise of embedded computer systems, physical or computational, with ongoing interactions with the world fuels the need for incremental learning methods such as reinforcement learning. This chapter introduces reinforcement learning by describing some of the paradigm shifts that this technique involves.

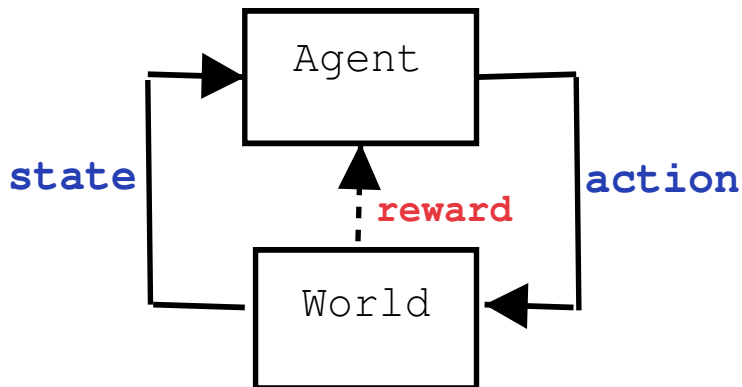


Figure 2.1: Trial-and-error cycle

2.1 From Dynamic Programming to Reinforcement Learning

Dynamic Programming (DP) is an optimization technique for problems that can be solved recursively with the characteristic that the subproblems are not independent and can overlap. Because of its recursive nature, the optimality of a solution is based on the optimality of the solution to subproblems. Solving a dynamic programming problem means to find a recurrence relation describing the problem. Bellman's optimality equation (2.1) is such a recurrence relation for sequential decision tasks where $V^*(s)$ is the optimal value of state s , a the action taken from s to s' such that $\sum_{s'} P_{ss'}^a = 1$, $P_{ss'}^a = P(s'|s, a)$, $a \in A$, the set of actions, r the current reward, and γ the discount factor, $0 < \gamma \leq 1$, weighing the future rewards.

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^*(s')] \quad (2.1)$$

One can choose from two algorithms, value iteration and policy iteration, to implement this recurrence relation in an iterative, bottom-up fashion, which is more efficient than the top-down recursive approach. For both of those algorithms, a model of the transition probabilities $P_{ss'}^a$ must be provided apriori.

2.1.1 Value Iteration

The value iteration algorithm computes the optimal value of a state before computing a policy based on this value. Algorithm 2.1 illustrates the two main loops of the algorithm. Step 1 has to terminate before Step 2 can begin. The value function $V^*(s)$ and policy function $\pi^*(s)$ can be computed concurrently in a parallel architecture or in a fixed order as in the Gauss-Seidel iteration. The Prioritized sweeping algorithm[55] optimizes this order according to the size of the update step.

Algorithm 2.1 Value iteration

1. For all $s \in S$, repeat until convergence

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^*(s')]$$
 2. for all $s \in S$

$$\pi^*(s) = \arg \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V^*(s')]$$
-

2.1.2 Policy Iteration

The policy iteration algorithm (Algorithm 2.2) interleaves the determination of a policy function $\pi(s)$ with the evaluation of the value function $V(s)$ for this policy. The value “determination” at step 2 of the algorithm should converge faster than in the value iteration algorithm since only the action specified by the policy function is used. The policy

improvement at step 3 interacts with the policy evaluation at step 2 in a series of relaxation steps.

Algorithm 2.2 Policy iteration

1. Given an initial policy π
 2. for all $s \in S$, repeat until convergence

$$V(s) = \sum_{s'} P_{ss'}^{\pi(s)} [r_{ss'}^{\pi(s)} + \gamma V(s')]$$
 3. for all $s \in S$

$$\pi(s) = \arg \max_a \sum_{s'} P_{ss'}^a [r_{ss'}^a + \gamma V(s')]$$
 4. if $\pi(s)$ is stable, stop; else go to 1
-

2.1.3 Reinforcement Learning

Reinforcement learning replaces the model based expectation of dynamic programming with learning from sample estimates. Reinforcement learning does not necessarily learn all possible mappings of state to action unless visited. It trades off sweeps of the entire state space in dynamic programming for the price of visiting each state infinitely often for convergence to occur. In practice, only the states needed to reach the goal need to be visited for a satisficing solution. This approximation to an optimal solution rather than an accurate optimal solution at greater computational cost enables the algorithm to adapt to changing goals. Three basic RL types are distinguished: the adaptive heuristic critic model, model-based reinforcement learning and direct reinforcement learning. The adaptive heuristic critic is derived from the policy iteration algorithm where the policy evaluation acts as a critic of the actor's behavior (Figure 2.2). Model-based reinforcement learning (see Algorithm 2.3) learns a model from experience by computing the transition probabilities $P_{ss'}^a$ as $\frac{\text{Number of transitions } (s,a) \rightarrow s'}{\text{Number of times in state } s}$ while learning the value function $V(s)$. This model in turn

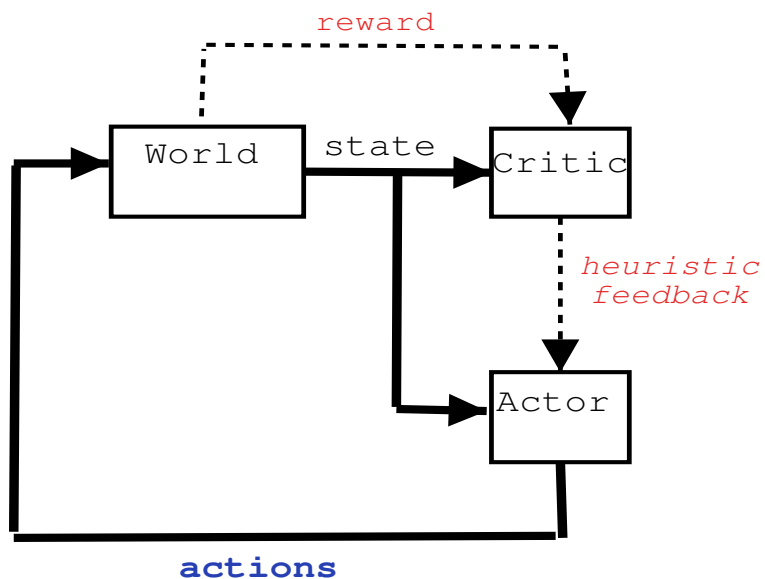


Figure 2.2: Adaptive heuristic critic

can be used to compute the policy function π or to replay experience as in Dyna[87] when experience is scarce. This is a general approach to learning and planning as a simulation of experience. But is computing a model really needed? Direct reinforcement learning (Algorithm 2.4), of relevance to this thesis, learns an action value $Q(s, a)$. Learning an optimal policy with direct reinforcement learning can be faster since there can be many such optimal policies for one optimal value function.

2.2 From A* to Reinforcement Learning

The A* algorithm is a best-first search algorithm where the evaluation of a state s , $s \in S$, is the sum of the incurred cost $g(s)$ of reaching that state and the estimated cost $h(s)$ of reaching the goal from that state.

Algorithm 2.3 Model-based reinforcement learning

- An initial behavior π is given
 - Initialize the learning rate α
 - Initialize $Count_{ss'}^a$, $Count_s$
 - Initialize $V(s)$ randomly
 - Randomly generate s
 - While stopping condition is false
 - $Count_s \leftarrow Count_s + 1$
 - Select an action a to execute according to π
 - Execute action a . Let r be the reward received and s' the next state
 - $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$
 - $Count_{ss'}^a \leftarrow Count_{ss'}^a + 1$
 - $P_{ss'}^a \leftarrow \frac{Count_{ss'}^a}{Count_s}$
 - $s \leftarrow s'$
 - For each $s \in S$
 - $\pi(s) = \arg \max_a P_{ss'}^a [r_{ss'}^a + \gamma V(s')]$
-

Algorithm 2.4 Direct reinforcement learning

- An initial behavior π is given
 - Initialize the learning rate α
 - While stopping condition is false
 - Select an action a to execute according to π
 - Execute action a . Let r be the reward received, s' the next state and a' the next action
 - $Q(s, a) \leftarrow Q(s, a) + \alpha [r_{ss'}^a + \gamma Q(s', a') - Q(s, a)]$
 - $s \leftarrow s'$
 - For each $s \in S$
 - $\pi(s) \leftarrow \arg \max_a Q(s, a)$
-

$$f(s) = g(s) + h(s) \quad (2.2)$$

The function h has to be an admissible heuristic, i.e. it guarantees to find the optimal solution, provided it exists, regardless of the start state, by never overestimating (or underestimating if solving a maximum problem) the distance to the goal. For example, the Euclidean distance to the goal is such an admissible heuristic for path planning in robotics. Lookahead and pruning can drastically speed up this algorithm. A* is an off-line algorithm requiring a planning or simulation phase all the way to the goal before making the first move. Real Time A*[41] interleaves execution with search and Learning Real Time A* (LRTA*)[41] extends the A* algorithm to repeated trials (Algorithm 2.5). The heuristic values $h(s)$ converges to their true values after learning. LRTA* learns the optimal solution by backing up the heuristic function from the evaluation of successor states in a one-step learning approach similar to that found in DP and RL. However, finding the heuristic function is often problematic, brittle, and requires a great deal of domain knowledge. Reinforcement learning learns the heuristic function $h(s)$ without a model and knowledge of the end goal simply by keeping statistics on what led to a good outcome. The idea of admissibility in the heuristic function carries over to RL with Q-learning and action penalties[39, 20]. The Q-values are admissible if they satisfy the following condition:

$$Q(s, a) \leq Q(s', a') \quad (2.3)$$

Algorithm 2.5 LRTA*

Input an adjacency matrix of S , $s \in S$

Generate a random state s

Repeat

1. For each neighbor s' of s
 $f(s') \leftarrow g(s, s') + h(s')$
2. $h(s) \leftarrow \min f(s')$
3. $s \leftarrow \arg \min_{s'} f(s')$

until s is a terminal

2.3 From Genetic Algorithms to Reinforcement Learning

Genetic algorithms (GAs) are a different model-free approach to the reinforcement learning problem as formalized in the n -armed bandit problem. GAs are a technique to solve optimization problems by encoding the elements of the solution space as binary strings. GAs are population-based algorithms, learning off-line through internal interactions, via the selection, mutation, and crossover operators, and only indirectly through interaction with the environment via a fitness measure. GAs search the space of policies rather than learn individual behavior. The credit assignment problem is addressed only implicitly[56] since the policy itself, as a collection of actions, is assigned a fitness and not the actions themselves.

Classifier systems[34] are a different kind of GAs. They consist of a population of rules, mapping states to actions as condition-action pairs, and encoded as binary strings. In response to an input, several rules fire in parallel and in succession. The bucket brigade algorithm, characteristic of classifier systems, addresses the credit assignment issue by

propagating back the feedback from the environment and has been shown to be equivalent to a temporal difference method[23]. Figure 2.6 illustrates a simple classifier system. A classifier can refer to any type of generalization algorithm.

Algorithm 2.6 Simple classifier system

Input A collection of classifiers as condition-action rules C

Initialize the strength S of each rule c for each classifier

Repeat

1. Read the current state s
 2. $M \leftarrow$ matching set of rules
 3. Fire a rule c of classifier x with a probability proportional to its strength
 4. $S_{t+1}(c_x, a_x) = (1 - \alpha)S_t(c_x, a_x) + R + \alpha(S_{t+1}(c_y, a_y))$
-

2.4 Discussion

This chapter described the progression and relationship of off-line algorithms, DP, A* and GAs, to the on-line learning method of reinforcement learning as a way of scaling up from a closed world to an open and dynamic world. Leveraging from prior knowledge promises in turn to speed up the convergence of reinforcement learning algorithms. The rest of this dissertation will further flesh out aspects of reinforcement learning as they apply to SLVQ.

Chapter 3

Sarsa Learning Vector Quantization (SLVQ)

Play at the centre of three stones

This chapter shows that the competitive learning rule found in Learning Vector Quantization (LVQ) serves as a promising function approximator to enable reinforcement learning methods to cope with a large decision search space, defined in terms of equivalence classes of input patterns. In particular, this chapter describes S[arsa]LVQ, a novel hybrid reinforcement learning algorithm while Chapter 5 will show its feasibility in the game of Go and Chapter 6 in continuous control tasks. As the distributed LVQ representation corresponds to a (quantized) codebook of compressed and generalized pattern templates, the state space requirements for on-line reinforcement methods are significantly reduced, thus decreasing the complexity of the decision space while preserving a table lookup approach to reinforcement learning.

3.1 Learning Framework

This section describes briefly the reinforcement learning and self-organization learning principles required for the SLVQ algorithm (see Section 3.2). The overall framework is illustrated in Figure 3.1.

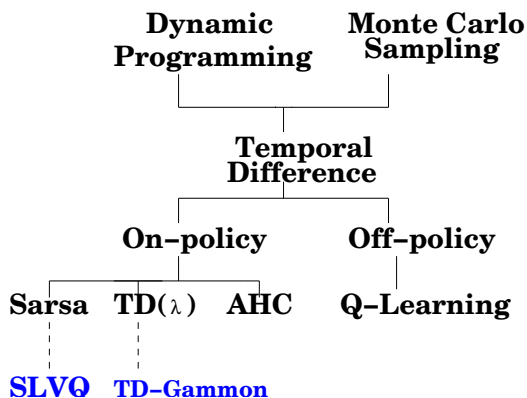


Figure 3.1: Learning framework

3.1.1 Reinforcement Learning Policies

In reinforcement learning, a learning agent interacts with its environment by taking actions and accepting input from the environment. Input from the environment constitutes the state of the environment followed by an immediate reward. State information passed to the agent summarizes all currently relevant information about the environment. In contrast to a purely reactive agent, a learning agent is endowed with an internal state that summarizes past history of its interactions with the environment. The environmental state and the internal state of the agent together are the state of the system upon which the learning agent bases its actions. An internal state enables an agent to generalize from previous experience which is missing from purely reactive architecture such as the subsumption architecture[6]. The reward passed to the learning agent is a scalar reinforcement that serves to evaluate current and past actions. While interacting with the environment, the agent follows a policy to determine what actions to take. A policy is a function, denoted as π , that maps the system state to an action to be taken by the agent. Through interaction with the environment, the agent learns either the value function $V^\pi(s)$ of a state s given a fixed policy π ,

or the action-value function, denoted as $Q(s, a)$, of a state-action pair. What is learned is a mapping to a “long-term” reward $E[\sum_{t=0}^{\infty} \gamma^t r^{t+1}]$ where $0 < \gamma \leq 1$ and r^t is the reward at time t . This form of expected long-term reward is called “discounted future” reward over an infinite horizon and has a finite value.

There are two basic ways of using experience in reinforcement learning: off-policy and on-policy¹. They differ only by the update rule used to arrive at an optimal policy (Figure 3.2). In an on-policy, the policy being updated (target policy) affects the selection of the next move. In an off-policy, the move evaluation of the next best move affects the update of the current move but the move selection does not depend on the policy being updated and can come from a completely different policy (behavior policy). Both policies reflect the bootstrapping strategy of dynamic programming to update the prediction for s_t from the next prediction s_{t+1} . The off-policy, embodied in the Q-learning algorithm[98], uses the estimate of the optimal policy for update of the existing policy and consequently separates exploration from control. The on-policy, embodied in the Sarsa algorithm[70, 88], uses the current estimate of an existing non-optimal policy for refinement towards a *better* policy $\hat{\pi}^*$ (see Algorithm 3.1) and combines exploration and control. The only guarantee to arrive at an optimal policy with Sarsa is possible only if the control policy progressively inches itself towards the optimal policy[81] as the exploration tapers off during training. In both policies, convergence has been proved in the discrete, tabular case if each action is selected infinitely often[99, 15]. Convergence has also been proven for TD(λ) in the linear representation case[17].

An on-policy approach for game learning has more opportunities for active learning in the exploration of moves as well as better on-line performance in “teaching” games. In

¹This is not to be confused with off-line and on-line updating. An offline updating scheme waits till the end of a cycle to do batch update. An online updating scheme is incremental.

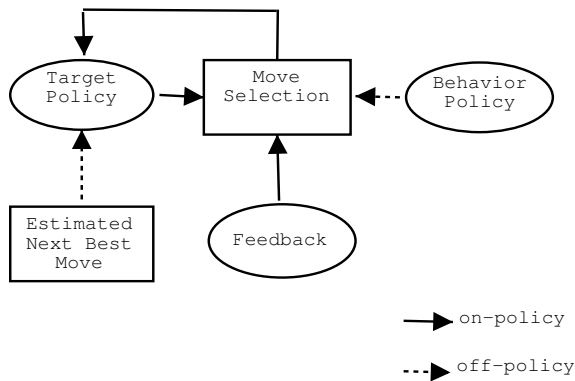


Figure 3.2: On and off policies

addition, credit assignment to previous moves, or *eligibility trace*, is not limited as it is the case for the off-policy approach where greedy control moves have to coincide with the exploratory policy. Better informed exploratory policy will be the subject of Chapter 7.

3.1.2 Self-Organization and Learning Vector Quantization

Self-organization involves the ability to learn and organize (cluster) sensory information without the benefit of a teacher. Learning is driven by measures of fitness, possibly evolved over time. If the task to be learned is that of clustering, one example of such a fitness measure is that of similarity. The process of self-organization consists of iteratively modifying synaptic weights in response to sensory patterns until an optimal configuration, according to some closeness measure, eventually develops. One particular class of self-organizing systems that are of interest to us are the Self-Organizing Feature Maps (SOFM)[40], which are driven by competitive learning. In the competitive learning scheme, the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron or one neuron per group is on at any one time. A neighborhood function,

Algorithm 3.1 Sarsa

Initialize $Q(s, a)$, the learning rate α , and the discount factor γ .

Observe the current state s

While stopping condition is false

1. Select an action a to execute according to π
 2. Receive immediate reward r
 3. Observe the new state s' and new action a'
 4. Update $Q(s, a)$

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$
 5. Update π towards $\hat{\pi}^*$
 6. Set s to s'
-

decaying with time, activates “other” neurons to learn from the same input. Consequently, the locations of the winning neurons tend to become ordered with respect to each other in such a way that a meaningful lattice-like coordinate system eventually emerges and faithfully represents the sensory input.

There are many situations where the clusters derived as a result of self-organization have to be appropriately labeled as it would be the case for information retrieval. Towards that end, one expands SOFM using a supervised learning scheme as it is the case with Learning Vector Quantization (LVQ). In the case of LVQ, the labeled clusters collection correspond to a (quantized) codebook of compressed pattern templates mapping the continuous space \mathbb{R}^n into the discrete topological space of the codebook vectors.

The LVQ algorithm[40] is a supervised clustering method in which each output unit represents a particular class or category. The weight vector for an output unit is often referred to as a prototype or *codebook* vector for the class that the unit represents. It is also assumed that a set of training patterns with known class labels is provided, along with an initial distri-

bution (“seed”) of prototype vectors. After training, the neural net classifies an input vector by assigning it to the same class as the (labeled) output unit that has its weight vector closest to the input vector (see Algorithm 3.2). After learning, the probability density function of the input is approximated by the modified set of discrete decoders or codebook vectors. The distributed representation of LVQ into codebook vectors as generalization of the input patterns significantly reduces the state space requirements and has a close correspondence to a tabular representation of state-action pairs. The generalized Lloyd algorithm[46] is a related algorithm, albeit without the topological ordering property, applying batch updates rather than on-line updates to the codebook vectors as centroids of their respective partition.

Algorithm 3.2 LVQ

$x \leftarrow$ input vector

$W_j \leftarrow$ weight vector for the j th output unit

$\|x - W_j\| \leftarrow$ Euclidean distance between the input and weight vectors.

- Assign a fixed number of reference vectors (“weights”) to each of the pattern classes. Initialize the weights and the learning rate α .
 - While stopping criteria is false
 1. Select a sample x from the training set.
 2. Find index j so that $\|x - W_j\|$ is at a minimum;
 3. If j belongs to the correct class
 - $W_j(t + 1) = W_j(t) + \alpha [x - W_j(t)]$
 - else
 - $W_j(t + 1) = W_j(t) - \alpha [x - W_j(t)]$
 4. Reduce monotonically the learning rate α as a function of t .
-

3.1.3 Hybrid Learning

Learning algorithms provide different biases for searching the hypothesis space which gives preference to certain generalizations. Only if the target concept matches the bias would a learning algorithm be successful. For example, if a learning algorithm only searches the space of conjunctive expressions, the target concepts it can learn are limited to this reduced hypothesis space. Inductive learning is inherently biased because a choice must be made between possible generalizations. In learning a policy mapping states S to actions A , the hypothesis space is the total number of state-action pairs $\sum_{s \in S} |A|$ or policy space. In large state spaces, the learning task also includes searching the hypothesis representation S . For example, the hypothesis space for the weights of a feature vector is \mathfrak{R}^n where n is the number of features. Hybrid systems try to build a synergy from complementary biases. In the case of SLVQ, the bias of LVQ, or competitive learning, is to reduce the dimensionality of the state space \mathfrak{R}^n according to a Voronoi tessellation into “prototypes” and to minimize the overall distance from the prototypes to the input distribution. The bias of Sarsa, or temporal difference learning, is to minimize the distance from one action value to the next temporal action value[89]. Together they define a new kind of function approximator based on recognizing when to apply a certain action rather than evaluating the resulting state of an action through simulation. This hybridization makes a powerful tool for intelligent control and other applications.

3.2 The SLVQ Algorithm

SLVQ integrates Sarsa with LVQ. This integration loosely ties the estimation of the action-value Q to the pattern recognition task of the situation (see Figure 3.3). This is in contrast

to value function approximators which uses pattern recognition techniques to directly associate a state description to a value, e.g. the probability of success. An SLVQ codebook vector is a tuple $\{m, a, Q, \alpha\}$, composed of the following:

1. a weight vector m
2. the action value Q , $-1.0 \leq Q \leq 1.0$
3. the local learning rate α , $0 < \alpha \leq 1$
4. an action a corresponding to the pattern class of LVQ, $a \in A$

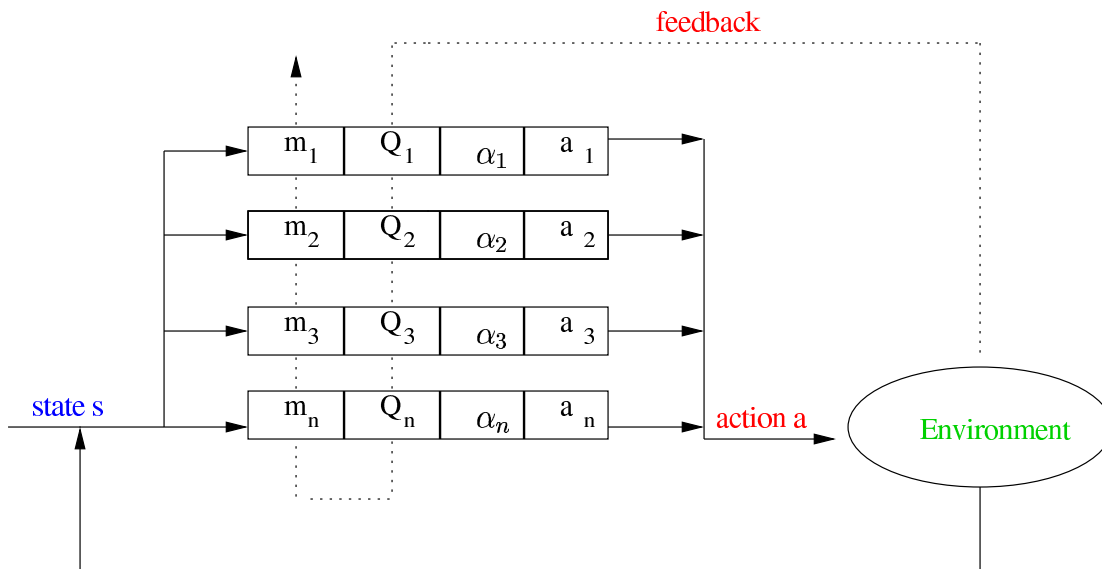


Figure 3.3: SLVQ architecture

In the off-policy control algorithm, the action value of the *best* move a according to the current estimate of the optimal policy is the one used for updating the state-action pair $Q(s, a)$ of the previous step. In contrast, in an on-policy control algorithm, the action value of the *next* move taken a' will be used to update the state-action pair $Q(s, a)$ (see 3.1.1).

The update of the weight vectors is a function of the change in the action-value of the move. Let $\Delta Q(m_t, a)$ be the change in action value at time t with action a and discount factor γ . The weight vector m that matched s_t most closely then moves closer to or away from s_t accordingly:

$$m_t = \arg \max_m \text{similarity}(s_t, m) \quad (3.1)$$

$$\Delta Q(m_t, a) = \alpha_{m_t} [r_t + \gamma Q(m', a') - Q(m_t, a)] \quad (3.2)$$

$$m(t+1) = m(t) + \Delta Q(m_t, a) [s_t - m_t] \quad (3.3)$$

Figure 3.4 shows an adaptive heuristic critic view of SLVQ where reinforcement learning provides the critics through the Q-values and the LVQ codebook tuples are the actors.

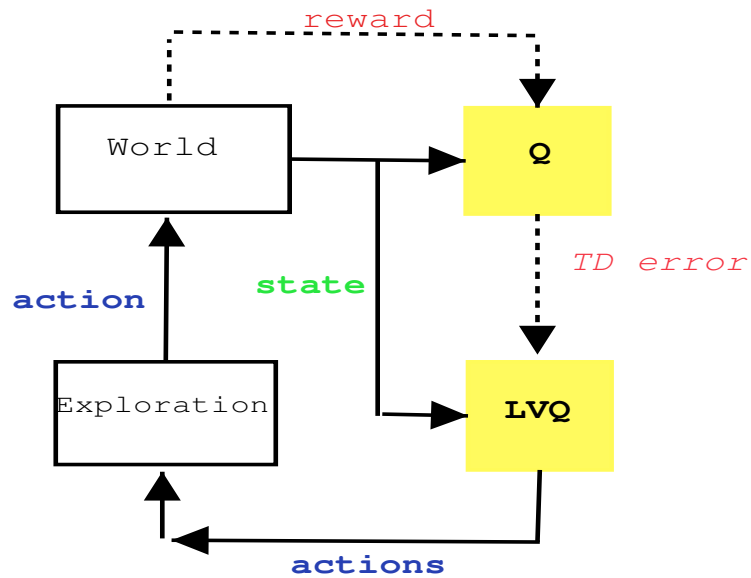


Figure 3.4: SLVQ adaptive heuristic critic architecture

The choice of an on-policy approach gives us better on-line performance at the expense maybe of less flexibility in exploration. The learning rate α is local to m and should decay proportionally as a function of the number of times m won the competition for convergence. Algorithm 3.3 shows the backward implementation view of SLVQ where the updates are done at the end of an episode. This implementation holds constant the codebook tuples until the end of the episode.

Algorithm 3.3 SLVQ

Input outcome is the reward at T , the end of an episode

Procedure SLVQ(outcome)

while ($T > 0$)

Codebook $\leftarrow \{m, a, Q, \alpha\}_T$
 $\delta \leftarrow r + \text{outcome} - Q_{\text{Codebook}}$
 $t \leftarrow T$
 trace $\leftarrow 1$

while ($t > 0$)

lastCodebook $\leftarrow \{m, a, Q, \alpha\}_t$
 lastCodebook.backup(trace δ , s_t)
 trace $\leftarrow \text{trace} \lambda$
 $t \leftarrow t - 1$

$T \leftarrow T - 1$

outcome $\leftarrow \gamma Q_{\text{Codebook}}$

Procedure BACKUP(delta, state)

$m_{t+1} \leftarrow m_t + \alpha_t \text{delta}(\text{state} - m_t)$

$Q_{t+1} \leftarrow Q_t + \alpha_t \text{delta}$

decay α

The theoretical justification of using an on-policy approach with LVQ rather than an off-policy approach is that LVQ tries to approximate the probability distribution of its input by minimizing its distortion:

$$E = \int distance(s, m_c)p(s)ds \quad (3.4)$$

where s is the input state and m_c is the winning codebook vector. The error E will be minimized if $p(s)$, the input probability distribution during training, corresponds to the target policy. The convergence of the algorithm is predicated upon the convergence of the series $\sum_{t=0}^{\infty} \Delta Q(m_t, a)$ in 3.2.

3.3 Knobs

Successful setting of some of the parameters can drastically influence the learning speed and convergence of this algorithm. In SLVQ, some parameters are set locally for each codebook vector such as the learning rate α and the exploration rate τ and some are set globally such as the discount factor γ , the eligibility trace decay parameter λ , and the number of codebook vectors m . A goal of further research is to localize those global parameters to enhance the distributed capability of the algorithm. The influence of those “knobs” are examined below.

3.3.1 The Number of Codebook Vectors

There is a trade-off between the number of codebook vectors m and the learning speed. If more codebook vectors are used, the internal representation of the task is more detailed

and it takes longer to learn their relative relevance. This is the curse of dimensionality for an algorithm such as SLVQ. States have to be visited infinitely often for a stochastic approximation process to converge and a small number of codebook vectors will ensure that states are visited often. With less codebook vectors, convergence is faster and often the algorithm generalizes better. The number of codebook vectors to use can be dynamically found with an unsupervised SOM preprocessing step[9] of example states. The action label of the clusters found can be determined by a majority vote while preprocessing the unclustered vectors again to arrive at a principled number of codebook vectors.

3.3.2 The Discount Factor γ

The γ parameter, or discount parameter ($0 < \gamma \leq 1$), balances the importance of anticipated rewards versus the importance of local rewards. Its role is to minimize the number of steps it takes to obtain the actual reward since steps closer to the goal will be rewarded more than others depending on the size of γ . A discount of 0 implements a greedy strategy. The number of iterations necessary for convergence is exponential as a function of γ . It also works to eliminate fast but ill-fated moves. However, quick but mediocre moves might appear more favorable than slower moves with better results since they appear closer to the goal and are reinforced more often. R-learning[76] proposes to learn average performance instead of maximizing expected discounted rewards to remedy to this problem. A careful computation of the final outcome can also help in this problem. When there is no clear intermediate reward in an episodic task and the outcome at the end of the task is the only scalar feedback, γ is generally 1.

3.3.3 The Eligibility Trace λ

The λ parameter, or eligibility trace decay parameter ($0 \leq \lambda \leq 1$), controls the *temporal* credit assignment by rewarding state-action pairs that appear more recently in a solution. It extends temporal-difference learning algorithms beyond the next step approach. It was shown in [83] that the recency heuristic was more accurate than the frequency heuristic for credit assignment in prediction problems. With $\lambda > 0$, the temporal difference δ between two successive predictions is applied to past state-action pairs. When $\lambda = 1$, TD methods are equivalent to Monte Carlo methods[88], i.e. each update is based on the final outcome rather than on the next successive prediction (see Figure 3.1 for the relationships of the different RL methods). This (a) speeds up the learning process by spreading the outcome to other close codebook vectors and (b) reduces the bias from the next prediction to several predictions[89]. When coupled with LVQ, the eligibility trace defines a temporal neighborhood around the winning codebook vector. Patterns that do well together will be reinforced and tend to occur temporally together in an hebbian fashion.

Let the reinforcement learning feedback at time t be:

$$\delta_t(0) = r_t + \gamma Q_{t+1} - Q_t \quad (3.5)$$

Let k a situation that occurred before t , its credit assignment $\delta_k(t)$ at time t is as follows:

$$\delta_k(t) = \lambda^{t-k} \delta_t(0) \quad (3.6)$$

The total update at the end of an episode T is then:

$$\delta_k = \sum_{s=k}^T \lambda^{s-k} \delta_s \quad (3.7)$$

The total change for the weight vector m at the end of an episode becomes:

$$\Delta m = \alpha(s - m)\delta_k$$

The credit assignment δ_k depends on the size of the temporal differences δ_s and the recency of k . The basic assumption of operand conditioning in psychology, on which reinforcement learning is based, is that temporally related events are causally related. At the beginning of the training process, temporally related events occur without justifiable causality assumptions. It does make sense therefore to start with a large λ to speed up learning and incorporate a more reliable final reward and slowly decrease it as the next temporal prediction becomes more accurate [89] like the radius of the topological neighborhood function in SOFM (3.1.2). A second assumption in eligibility trace is that the greater the temporal difference δ , the further removed is its cause which does not account well for “blunders”, that is short-term mistakes with great consequences.

3.3.4 The Learning Rate α

Convergence in reinforcement learning, and stochastic approximation in general, has been predicated upon an appropriately decaying learning rate such that $\sum_{t=0}^{\infty} \alpha(t) = \infty$ and $\sum_t^{\infty} [\alpha(t)]^2 < \infty$ [99]. An optimal decaying rate for LVQ where codebook vectors are modified according to a positive/negative scalar feedback $s(t)$ has been shown to be $\alpha(t) = \frac{\alpha(t-1)}{1+s(t)\alpha(t-1)}$ [40] where the learning rate decreases if the feedback is positive and increases

if the feedback is negative. The process converges only when the feedback becomes mostly positive. A decaying learning rate gives equal weight to earlier and later examples and thus assume a stable distribution of the examples. When the learning rate is $\frac{1}{t}$, where t is the number of times a codebook has been updated, the weight vector reflects the exact mean of the input weight vectors. Convergence is obtained mostly at the expense of optimality without an appropriate exploration policy. The exploration changes the distribution of the examples (as in boosting² for example) and the codebook vectors should adapt to later rather than earlier examples which motivates a constant learning rate as in TD-Gammon[93]. A compromise can be achieved by dynamically adding new codebook vectors from the input examples when no good fit is found instead of oversaturating the existing weight vectors.

3.4 Sources of Instability

There are two sources of instability in SLVQ: (a) if the feedback is negative, the winning tuple $\{m, a, Q, \alpha\}$ might oscillate between two competing codebook vectors leading to two alternating sequences of actions; and (b) if the feedback is positive, a codebook vector might become more general and attract more input states at the expense of other codebook vectors thereby making the mapping from states to actions less specific (see “Collective Entropy” in Chapter 7) and less optimal. Those instabilities are due to a mismatch between the weight vectors distribution and the input distribution in the quantization of the state space. Bootstrapping from a semi-clustered search space (see 3.3.1) mitigates those oscillation problems.

There is an inherent bias in temporal difference learning methods that favors hill-climbing

²Boosting is a resampling algorithm coupled with a weighted majority scheme that redistributes the proportion of positive/negative examples to train a learner on different region of the problem space[73].

at the expense of finding a global optimum. The Markov chain in Figure 3.5 illustrates this problem. A longer path will be favored if successively “better” actions can be found. The algorithm can get stuck at the first successful but suboptimal sequence of actions. Discounting can reduce the magnitude of this discrepancy but this problem is compounded for SLVQ where the update of the Q-value influences the update of the codebook vectors as well. An efficient exploration scheme has to sample the space to reflect the true expected return at any time to avoid this instability.

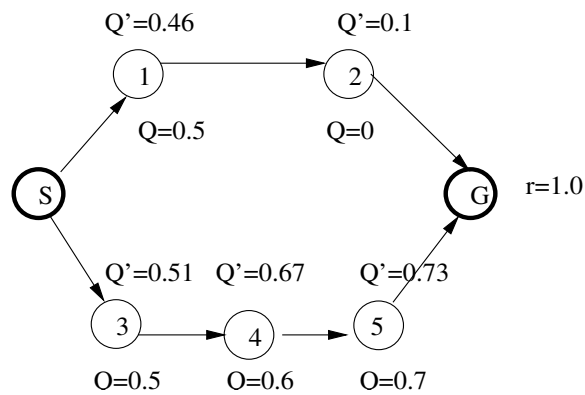


Figure 3.5: Hill-climbing bias

Assuming identical initial conditions, a longer path will be favored if each of its actions has been part of a successful sequence.

$$(\alpha = 0.1, \gamma = 1.0, \lambda = 0)$$

Chapter 4

Computer Go

if it has a name, know it

This chapter summarizes the issues and directions that are being followed in computer Go. An overview of the game and the main concepts are given in Appendix A. A detailed description of the implementation of different approaches is given in Appendix B.

4.1 The Complexity of Go

While several programs have become world champions¹, Go is still a challenge for a computer program and has replaced chess as the “drosophila” of AI. Although human expertise might provide stronger programs, a knowledge-free approach will provide some insight into achieving human-like intelligence and help augment other programs. Since Go is a perfect-information, deterministic game, “solving” the game might seem like a possibility. “Solving” a game means determining the game value of a move, win or lose, from any arbitrary board position. This is different from finding a strategy for playing the game. It has been shown however that Go is P-space hard[44] and therefore “solving” the game

¹Chinook for checkers, TD-Gammon for backgammon, Bill for othello, and Deep Blue for chess.

Table 4.1: Go vs. Chess

	Rubik's Cube	Chess	Go	Go	Go	Go	Go
Board Size		8x8	5x5	7x7	9x9	13x13	19x19
Avg Branching Ratio		30 – 40	9	18	30	61	180
Avg Number of Moves		50	40	89	140	220	250
Space Complexity	10^{19}	10^{43} [78]	10^{11}	10^{23}	10^{39}	10^{80}	10^{172}

with an algorithm computing an exact evaluation function is probably not practical. Table 4.1 illustrates the complexity of Go compared with that of chess. The complexity of a game is usually measured in terms of its branching ratio and number of moves. Space complexity[2] is a compact measure representing the number of legal board configurations. This number is quite large but can be estimated from Monte Carlo simulations. Go is also hard for machine learning because there are no clearly identifiable features (see [5] for a state-of-the-art survey). Rather, each feature is a pattern that, like fractals, grows into larger and more detailed patterns.

4.2 Game Theory vs. Game Playing

Game playing has been firmly ensconced in game theory. Game tree searches owes its origin to the Von Neuman's minimax theorem which states that there is a minimum value to a move, no matter what the opponent does. The minimax theorem assumes a zero-sum game, that is, a player's optimal strategy is to minimize an opponent's gain. Game tree searches based on the minimax theorem have been paradoxically extended to non-zero sum games like Chess and Go. Such a generalization of the minimax theorem has been made

on the assumption that, to the extent that there is a winner and a loser, all games can be thought of as zero-sum. For strategic games, a win or a loss has to be read over a sequence of moves and requires a search. As long as the evaluation function is a good approximation of the value of the board, even if the game tree does not reach a terminal state, the zero-sum generalization holds.

The duality of acquiring territory vs. influence in the opening (joseki) moves is an illustration of the non-zero sum aspect of the game of Go and the dichotomy of game theory and game playing. Obtaining territory in one corner gives influence to the opponent and determines whether in the next joseki move the opponent will fight for territory instead. If fighting is to be avoided because of the players' relative skills and the current situation, the next best move might not be the optimal move in a game-theoretical sense and the player should settle for influence.

4.3 Feature Weighting vs. Feature Discovery

Early efforts in game learning[71] has been in weighting the features of an evaluation function driving a game tree search. By hand-crafting the features of an evaluation function, human expertise is transferred to a program. In contrast, a pattern recognition approach tries to induce an evaluation function from an unbiased "raw" representation of the game. Features can be then be extracted for understandability as a post process from a neural net for example. Feature weighting is notably more efficient in reaching a competent level of play but can quickly reach a ceiling. On the other hand, a pattern recognition approach is capable of construing new features based on the current input and the stored patterns. Together, they can provide a powerful synergy. For example, adding a set of hand-crafted

features to a raw board representation has enabled TD-Gammon to reach the expert play level[93].

4.4 Look-Ahead Search

Game playing has traditionally been viewed as a search problem based on the minimax theorem and learning approaches have been oriented toward making this search more efficient. The minimax theorem assumes a zero-sum game where one player's win is another player's loss. Strategic games violate this assumption in the sense that a win or a loss are read over a sequence of moves and require a deeper search. In addition, a large branching factor in the game of Go contributes not only to the impracticality of a pure minimax search but also to the pathology of game trees[61]. If we look at the evaluation of a node as a probability of winning rather than an absolute value, then the deeper the search, the worse the error in the evaluation as values are backed-up the tree with minimax. This problem of error propagation with the evaluation of a node as an approximation also occurs when using function approximators with reinforcement learning[84]. Short look-ahead searches might be useful to avoid "tactical blunders"[13], or specifically in the game of Go, to recognize patterns such as *ladders*. No other factors such as improved visibility and node dependence seem to occur in the game of Go to justify deep searches. Heuristics or rules of thumb are the traditional alternative to search in large state space. Go lends itself well to this approach as testified by the number of Go proverbs but heuristics conflict in their application.

4.5 Plan-Based Search

A goal-driven approach to the game of Go is based on the intuition that the purpose of a good Go move should not only be a reaction to the present situation but should carry some purpose. Those proactive moves are known as *sente* (Appendix A). Goal-driven adversarial planning requires to find a goal or set of goals for which no countergoals can be instantiated[79]. The difficulty in instantiating countergoals is usually a function of the number of conjunctive goals G that can be satisfied and the number of moves m required to execute them or $\frac{G}{m}$. If an achievable countergoal can be found, backtracking occurs. Because the search is focused, this approach can be very quick even with large state space. Testing for feasibility of a plan can also be fast since not all possible moves need to be examined. There is however a computational tradeoff between focusing the search in the state space and enlarging the search in the plan space with added knowledge. The difficulty with this approach is in the evaluation of the goals themselves. There is no clear evaluation of the urgency of the context except in relation to the urgency of the goal itself. An integration of the pattern and goal approach seems better suited for Go. Gobi[102] allocates demons or plan *critics* over parts of the board to detect certain urgent situations and insert appropriate plans into the agenda. This hybrid approach integrates tactics and strategy.

4.6 Decomposition-Based Search

Since Go is a perfect information game, it is conceivable that a combinatorial-based exact evaluation exists without requiring look-ahead. In combinatorial games² such as Nim, the

²A combinatorial game is defined[31] as an episodic, perfect information, zero-sum game where the winner is the last one to play.

winner has the last move. Each move decompose the game into successive independent subgames³. In Go, the board can be decomposed into independent subgames, that is a move in one subgame does not affect the outcome of other subgames, towards the endgame or when expert knowledge about the safety of a group can be used. Decomposition search[57] evaluates the value of a game, i.e. a win or loss for the player whose turn it is to move, by the combinatorial game value of each subgame. If from state s with n subgames, black can move to $b_1 \dots b_n$ and white can move to $w_1 \dots w_n$, the combinatorial game value of s , $C(s)$, is given by

$$C(s) = \{C(b_1), \dots, C(b_n) | C(w_1), \dots, C(w_m)\}$$

This combinatorial expression expands until a terminal expression where either black moves first or white moves first occurs. This terminal value is then backed up the game tree. Dominated solutions on either side are eliminated until a unique game value remains. Assuming independent subgames can be found, decomposition search gives an accurate solution of the value of a game.

4.7 Proof-Number Search

Proof-number search (pn-search) is a game-tree search algorithm comparable to alpha-beta search. The game is played by continuously proving or disproving it. Pn-search is attractive for Go because the evaluation of a node does not depend on prior knowledge but simply on its contribution in solving a tree. A terminal node has proof number 0, 1, or

³Subgames in game theory are subtrees of a game tree.

∞ , corresponding to true, false or unknown. Unknown values denote also frontier nodes, i.e. nodes that haven't been developed. Proving or disproving a tree involves developing frontier nodes and backing up their values up the tree. A proof or disproof number is the number of nodes that must be developed in order to prove or disprove the tree. Those numbers can be used to direct the search towards the smallest subset of frontier nodes that would solve the tree. As a result, the total number of node development to solve a tree is reduced by focusing on potential solutions involving the smallest number of nodes. Pn-search is equivalent to the heuristic of playing the move that would result in the smallest number of legal moves for the opponent.

4.8 Pattern-Based Search

The pattern database forms the core of many Go programs (e.g. [28]). Patterns are represented as bitstrings for efficient bitwise matching or rule-based constraints between salient stones. Associated with each pattern is a move value representing its relative urgency. Scanning a board for patterns of different sizes can be done with a deterministic finite state automata built from the pattern database. Starting from each intersection, the neighboring intersections are scanned following a predefined path in search of a pattern. *Tuning* the pattern database requires a great deal of skill and expert knowledge and the management of over a thousand patterns. The automatic acquisition of patterns is an important research area in Go which this research addresses.

4.9 Discussion

The different avenues of research being pursued in the game of Go are an indication of the difficulty of the game. Pattern matching is essential to speed up the search and has been used especially at the beginning of the game where move sequences have been standardized. Combinatorial game theory are particularly useful in the endgame. The middle game has not been reduced to any single method and a combination of the different approaches might be required. The next chapter applies SLVQ as a pattern-based inference approach to Go.

Chapter 5

SLVQ Go

Learn to play under the stones

This section shows the feasibility of SLVQ Go and its comparative merits compared to other potential Go players. The interactive nature of reinforcement learning is particularly appealing for game learning since the early days of Samuel's checker player[71]. What is learned through interaction with the environment is an optimal policy mapping states to actions maximizing the total expected reward. Compared to other learning paradigms, reinforcement learning has some nice properties for game learning: no expert knowledge is needed and it is incremental, that is continuous learning against a variety of opponents is possible. From a reinforcement learning point of view, Go is an episodic ¹ sequential-task problem with discrete states.

5.1 Representation and Evaluation of the Board

The board is represented as a square matrix of stone objects. A list of legal moves is generated and passed to the players. Players play until no legal moves exist. A legal move

¹Episodic tasks are considered infinite horizon tasks by treating the outcome as an absorbing state with transition to itself [88]. This approach unifies the study of episodic tasks and continuing tasks.

must not duplicate a previous board configuration and players must not play in their own eyes². This last constraint is necessary for the game to terminate without recognizing when to pass. The evaluation of the board is done using the Chinese rules which allow a player to fill its territory without penalty³ and a program to play without recognizing life or death patterns. No komi points were added for white and no prisoners were included in the final score. Because of the large state space, it is helpful to view the game with continuous states using the influence value propagation of the stones. After each move, the influence value of the move (+1.0 for black and -1.0 for white) is propagated to neighboring stones (Figure 5.1). This representation was first used by Zobrist[103]. However, our propagation algorithm is different and conveys the spatial connectivity between the stones. Each neighbor (including diagonals), if it's an empty point, computes its influence value by summing the influence value of its neighbors (8-connected neighbors) with equal weight until no changes occur in a ripple-like fashion (Algorithm 5.1). The influence representation of the board as a one-dimensional vector is used for matching against the codebook vectors. At the end of the game, the difference in territory (including the number of stones on the board) won over the maximum territory is propagated as the reward. This somewhat simplified version of the game still retains its main strategic aspects.

In this representation, each codebook tuple $\{m, a, Q, \alpha\}$, as a simple agent, has an associated weight vector m representing the move of the other agents on the board and their relationships through the influence value. Learning coordination strategies involves representing the actions of other agents and their relationships as part of the state of the world. Beyond the game of Go, coordination strategies for simple agents can be represented as an adjacency matrix of their relationships or *grid* structure.

²Including half eyes, but does not look-ahead for a potential capture of an enemy stone at one of the diagonals.

³We used Robert Jasiek's rules at <http://home.snafu.de/jasiek/simple.html>.

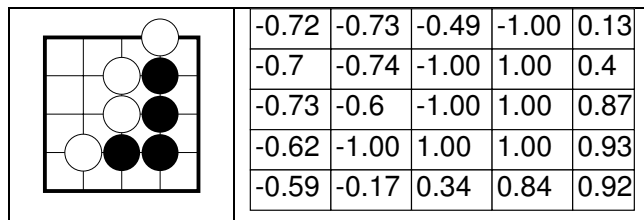


Figure 5.1: Influence representation
of a 5x5 pattern

Algorithm 5.1 Influence Propagation Algorithm

Influence Propagation of a Stone

push stone in stack

while (stack)

stone \leftarrow pop stack

Foreach empty neighboring stone s

influence $_s \leftarrow \frac{1}{8} \sum_j influence_j$ where j is a neighbor of s

if $\Delta influence > \epsilon$

push s in stack

5.2 Matching

Matching of a board configuration against a weight vector is done using the fuzzy contrast model [72, 97]. This distance is not a metric distance but takes into account the presence or absence of certain stones in a pattern. This characteristic makes it extremely well suited for the recognition of the vital stones in a game. Let a pattern P be represented by the set of influence values p_{ij} (after normalization) at each of its points. The similarity S between patterns P and R can be computed as a function of their commonality and their reflexive differences:

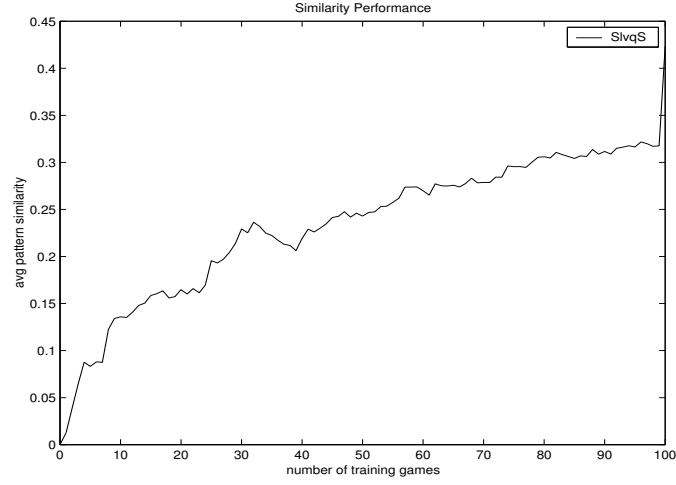


Figure 5.2: Similarity of the codebook vectors

$$Closeness(P, R) = \sum_i \sum_j \min \{p_{ij}, r_{ij}\} \quad (5.1)$$

$$Contrast(P, R) = \sum_i \sum_j \max \{p_{ij} - r_{ij}, 0\} \quad (5.2)$$

$$S(P, R) = Closeness(P, R) - \alpha Contrast(P, R) - \beta Contrast(R, P) \quad (5.3)$$

α and β represents the relative saliency of the prototype and the variant (input pattern).

The saliency of a board pattern P represented as a set of quantitative features by the influence values of the stones can be determined, for example, as the density $\sum_i \sum_j w_{ij} p_{ij}$ where w_{ij} is proportional to the inverse distance relationship of the move associated with the pattern. Rotation and mirror symmetry of the board (8 transformations) are performed on the input pattern P to compute a best match. Figure 5.2 shows the average similarity of the codebook vectors selected in random 5x5 games slowly increasing with the number of training games. The range of this fitness measure is between -1.0 and 1.0.

5.3 Experimental Methodology

The initialization of the weight vectors is done from real games played against Gnugo⁴. Bootstrapping from legal board games instead of randomly initializing the codebook vectors reduces drastically the complexity of this game. Each board configuration makes up a weight vector after the propagation of the influence values. The softmax exploration policy was used with SLVQ (SLVQS) where action selection is implemented by adding a small random number decaying with time to the action evaluation and picking the action with the largest sum[88]. Learning from the opponent, made possible by the color invariance property of Go, provides an even source of positive and negative examples.

5.3.1 Go Players

The following players are the adversaries, in addition to a random player, used in training games in this research. Table 5.1 shows the relative strength of the players on a 7x7 board.

Wally Wally[52] is a low-level player but its moves are surprisingly good despite no knowledge of the *whole board* situation. It uses simple heuristics: 1) capture; 2) put in atari; 3) search the board for a pattern matching a table of “good” moves ordered by urgency and breaking ties randomly; and 4) if nothing else was found to do, play a random move. The program[58] was modified to play white as well as black, to play on different board sizes, and to reseed the random number generator every 100 moves.

“Heuristic” This player deterministically chooses the move generating the highest sum of influence values. This evaluation is consonant with the input representation of the

⁴<http://www.gnu.org/software/gnugo/gnugo.html>

Table 5.1: 7x7 baseline results averaged over 100 games
 NN is a nearest-neighbor lookup based on the similarity function and
 the initial codebook vectors

black\white	NN	Random	Heuristic	Minimax	Wally
NN	-1.0	1.0	-0.18	-0.10	-0.57
Random	-1.0	0.75	-0.75	-0.74	-0.79
Heuristic	0.13	1.0	0.47	-1.0	-0.62
Minimax	1.0	1.0	1.0	-0.05	-0.23
Wally	1.0	1.0	-0.09	0.48	0.55

board. This heuristic has much more strategic than tactical value but can narrowly beat Wally when playing white.

Minimax This player chooses the move generating the highest sum of influence values in 2 ply assuming the opponent chooses the same strategy. It adds a lookahead to the “Heuristic” player.

Monte Carlo Go (MCGo) This player learns to play Go with no prior knowledge in a Monte Carlo approach. Each move has an associated weight and the policy is to pick the move with the greatest weight. During training, moves are selected stochastically according to a simulated annealing schedule. The weights w are updated with a moving average of the outcome of the game, $w_i = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{k=0}^{N-1} r^k$. This program is based on the observation that the outcome of a game is more consistent for earlier than later moves. This approach is loosely based on Gobble (see Appendix B). This player is included for comparison purposes with another learning scheme.

5.3.2 Comparison Metric

It is hard to compare different reinforcement learning experiments since the end, performance of the learner might matter more than how it got there. Moreover, the parameters need to be tuned to the exploration method which makes comparison more problematic. It is however useful to compare the learning power of an on-line algorithm and not just its off-line results after training with a t -test. The Wilcoxon's matched pair signed rank test measures the statistical difference between two non-parametric distributions by measuring the significance probability that the median of two matched samples are equal. It assumes that, if the methods are not different, then the probability of a positive or negative difference is equal, i.e. the differences are symmetric.

5.3.3 Experimental Setup

The parameters had to be tuned to the task. Developing a set of defaults is the subject of further research. The parameters were initialized as follows for the different experiments:

5x5 $\lambda = 0.07$
 $\gamma = 1.0$
 $\alpha = 1.0$ decays as a function of $\frac{1}{\sqrt{h}}$ where h is the number of times a codebook vector has won the competition.
 exploration $\tau = 0.09$
 Total number of codebook vectors $m = 300$
 MCGo temperature=5.0
 MCGo temperature decrement=0.015

7x7 against the Heuristic Player:

$$\lambda = 0.07$$

$$\gamma = 1.0$$

$\alpha = 1.0$ decaying as above.

exploration $\tau = 0.09$

Total number of codebook vectors $m = 1000$

MCGo temperature=25.0

MCGo temperature decrement=0.015

7x7 against Wally and Minimax:

$$\lambda = 0.5$$

$$\gamma = 1.0$$

$\alpha = 1.0$ decaying as above.

exploration $\tau = 0.13$

Total number of codebook vectors $m = 1000$

MCGo temperature=30.0

MCGo temperature decrement=0.015

5.3.4 Experimental Results

The off-line results with the greedy strategy show the average territory difference over 3 games for SLVQS (see 5.3) and MCGo playing as Black. Training was done against the same opponent as the games played off-line unless specified. Smoothing of the graphs (10%) ensures an easier readability of the performance trend in this complex game where one move difference can drastically alter the outcome of the game. Winning corresponds to a positive score (territory) difference over the opponent. Please note that Go is not a zero-sum game and that winning rarely involves wiping out the opponent's territory.

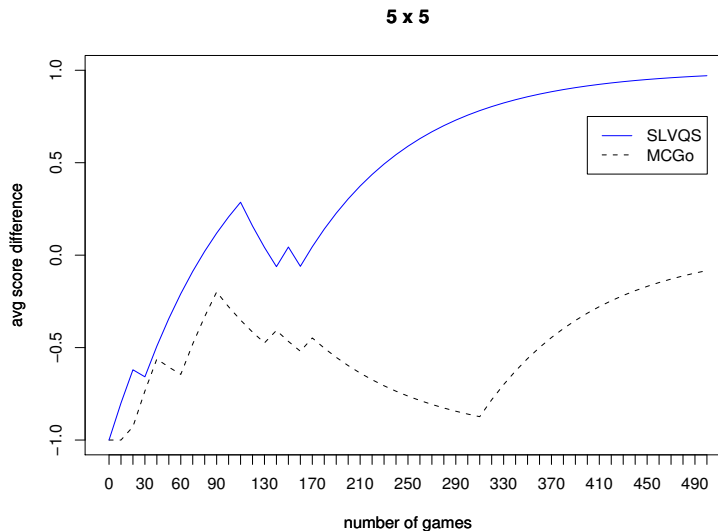


Figure 5.3: 5x5 games against Minimax

5x5 A 5x5 game has approximately 10^{11} possible configurations under rotational invariance. The legal move constraints bring this figure down somewhat but the state space still remains huge. A game has an average of 20 moves for each player. The codebook vectors were initialized with sample boards from 17 games for a total of 186 vectors and the rest filled with random weights. Figure 5.3 shows the comparative performance of SLVQS and MCGo. SLVQS clearly learns to defeat the Minimax player and is better than MCGo.

7x7 Increasing the dimensions of the board from 5x5 to 7x7 makes the game noticeably more difficult. Table 5.1 shows the relative strengths of the players. The credit assignment problem is more acute since a good move in a game that resulted in a loss will not be recognized as such. A game now has an average of 40 moves for each player. The codebook vectors were initialized from samples of 42 games or 659 codebook vectors and the rest filled with random weights. Figure 5.4 shows results against Wally. The performance

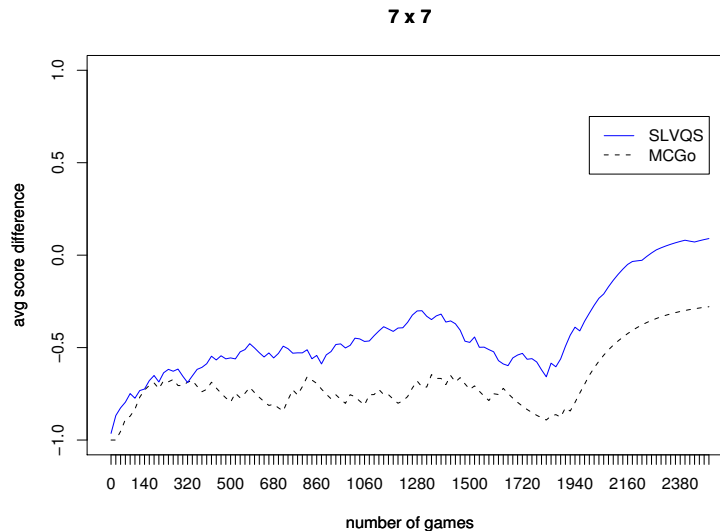


Figure 5.4: 7x7 games against Wally

of SLVQS improves as training progresses and converges to win narrowly against Wally. SLVQS again clearly outperforms MCGo.

Figure 5.5 and 5.6 show SLVQS winning against the “Heuristic” player and Minimax respectively. Clearly, it is easier to learn a strategy against a deterministic player and harder to learn to play against a tactical and random player like Wally since SLVQ has only a generalized representation of the board and here again, SLVQS outperforms MCGo.

5.3.5 Move Analysis

It is instructive to examine the kind of moves that can be learned with SLVQS. Driven only by the outcome as a global reinforcement signal and pattern inference, SLVQS learns from random weight initialization to defend, connect, and capture (see Table 7.7, (a) and (b)). It

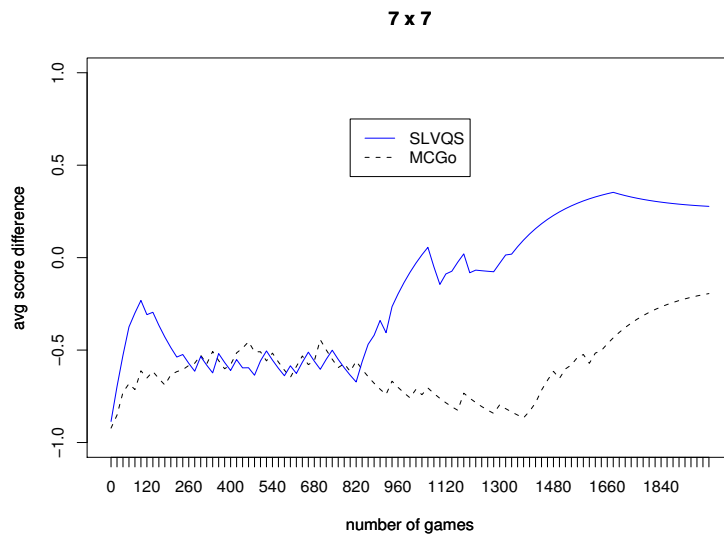


Figure 5.5: 7x7 games against “Heuristic”

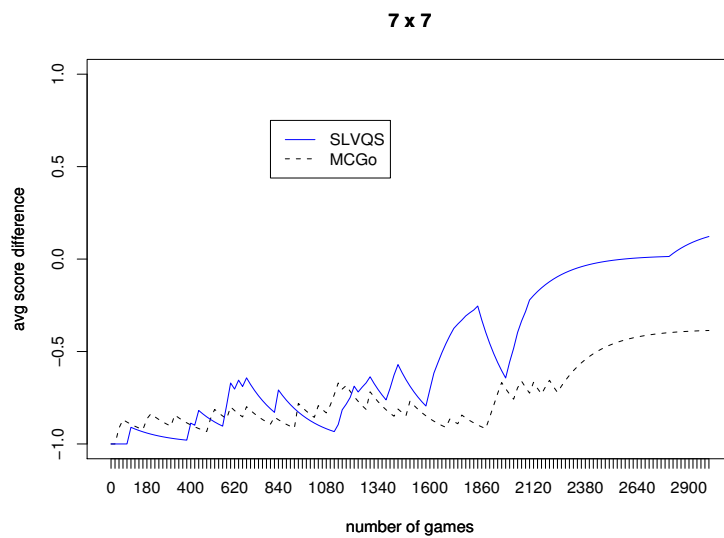
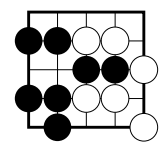
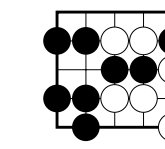
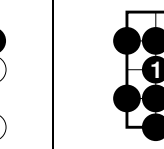
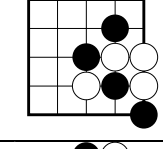
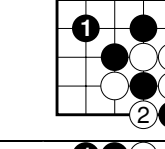
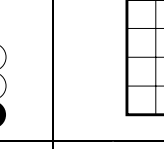
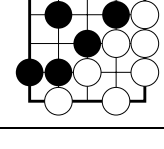
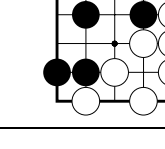
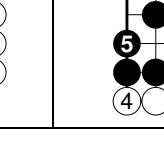


Figure 5.6: 7x7 games against Minimax

Table 5.2: Critical moves on a 5x5 board
SLVQS plays black

	pattern	before training	after ~500 games
(a)			
(b)			
(c)			

also learns how to live by playing the correct sequence of moves (Table 7.7(c)). SLVQS plays a good opening (Figure 5.7). It quickly learns the value of the center on a small board and of the corners but it has some amount of senseless *throw-ins* which are necessary in the endgame under the program's rules which requires to play until no legal moves exist.

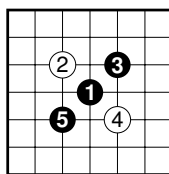


Figure 5.7: SLVQS opening against Wally
SLVQS plays black

Figure 5.8 shows a complete game record of SLVQS against Wally after ~2000 training games. SLVQS wins with 16 points and a score of 0.34. A move-by-move comment of the

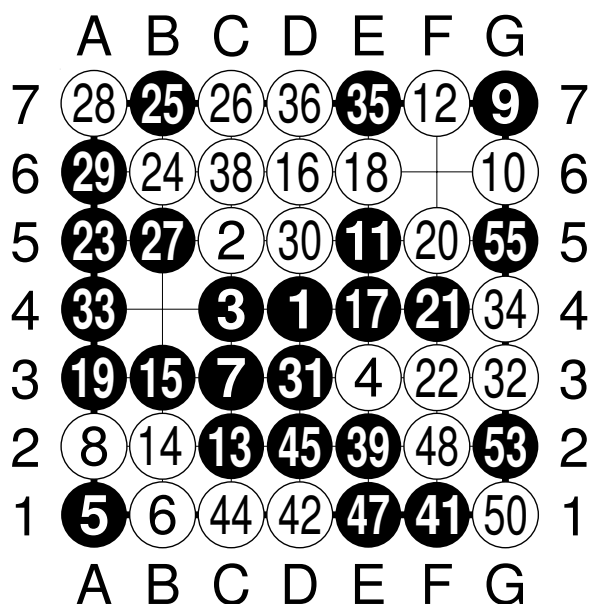


Figure 5.8: SLVQS vs. Wally complete game record

game follows.

With Move 1 and 3, SLVQS starts at the center of the board.

Move 5 is a poor move. Even if a tenuki⁵ were desirable, a play at the corner is worthless since it can't enclose any territory. This type of move is due to matching an endgame pattern. Fortunately, Wally follows it blindly with Move 6.

Move 7 is a poor shape.

Move 9-10 are similar to 5-6 above.

Move 11 is a reasonable move.

SLVQS succeeds in separating Wally's stones with the sequence of Moves 11 through 21.

Move 21 might just be the decisive winning move of the game.

⁵A tenuki is a move outside of the local area of play.

Move 23 is a blunder. It should have been at B5 which came later with move 27.

Neither side see the critical move at G4.

Move 39 is a good move. It makes territory for black and restrict territory for white.

Afer Move 50, SLVQS is in danger. It has two choices. Either fill in the *ko* in the upper left corner or capture the white stones at the lower left corner. It captures, makes its second eye, and lives.

In summary, SLVQS plays a strong opening in the center. There are some problems in the influence value representation in distinguishing opening and endgame patterns. A phase-of-the-game distinguishing feature should help. Wally fell victim of its own tactical strategies and got its stones separated. So how good is SLVQ in reality? Chapter 7 and 8 will forge the road ahead in finding ways to improve on those initial results.

Chapter 6

Continuous Control Tasks

play slow, win slow; play fast, lose fast ...

Examples of control tasks include everyday's tasks like parking a car, adjusting the temperature of the water when preparing a bath, keeping a fire burning in the fireplace, etc. Control tasks require constant adjustment by the operator much like taking turns in a game where the adversary is the environment. The process is composed of a succession of *states* and moves or actions influence the next state stochastically or deterministically. Those tasks are characterized by a continuous search space.

6.1 Motivation

Continuous control tasks typically try to influence a physical process[19]. In simulation, they can be represented as Markov Decision Processes (MDPs) with a limited number of relevant variables and small steps to represent continuous time. Those tasks are different from combinatorial tasks such as the game of Go where the representation is inherently discrete and was “transformed” into a continuous representation using the influence value. The smoothness assumption underlies continuous control tasks. It is the assumption that similar actions apply for situations close in input space except at some boundary conditions.

Those tasks define different conceptual neighborhoods and thus require different similarity functions. What control tasks and combinatorial tasks have in common is their assumption of the Markov property and their deterministic aspect. The cart centering and the mountain car problems examined below are 2-dimensional control tasks varying position and velocity. Both of those problems require a long sequence of steps and the credit assignment problem is compounded by the fact that the state trajectories can contain cycles. Previous approaches to those problems have included genetic programming and the discretization of the input space. The cart centering problem is compared against a genetic programming approach and the mountain car problem against a tiling approach based on discretization.

A task can be represented by state-action pairs where the states are real-valued variables describing the physical process at a moment in time and the action desired is typically a non-linear function of those states. It is difficult to compute an exact solution to those problems. The cart centering problem is one of the few problems for which an exact mathematical solution is known. A solution for those problems involves computing a step function for each action which divides the input space into continuous regions. A good similarity function for those problems would therefore represent the graded correlation of a prototype vector and an input vector.

6.2 SLVQ with Minimum Spanning Tree Topology (SLVQ-MST)

The continuous control task problem assumes that input vectors that are close from a distance point of view will be in the same neighborhood and trigger the same action except for those that lie at the boundary. It does therefore make sense for those problems to activate

neurons defined by a neighborhood function or kernel to learn from the same input. Spreading the reward around makes learning converge faster and less sensitive to changes in the parameters. By doing so, a *structural* credit assignment is propagated to similar states and similar actions. The Minimal Spanning Tree (MST) algorithm defines a topology connecting the codebooks with a minimum total distance and thus defines local neighborhoods with global optimality. The SLVQ control equation where m_c is the winning codebook vector at t gets propagated to neighboring codebook vectors according to a neighborhood function h_{ci} and move label. Under SLVQ-MST, potentially all codebook vectors learn from an episode. The update equations are modified as follows:

$$\delta = \gamma Q(m_c(t+1), a') - Q(m_c(t), a) \quad (6.1)$$

$$m_c(t+1) = m_c(t) + \alpha_c(t) \delta [s(t) - m_c(t)] \quad (6.2)$$

$$h_{ci}(t) = \begin{cases} \text{Similarity}(m_c, m_i), & \text{if neighbor}(c, i) \text{ at } t \\ 0, & \text{otherwise} \end{cases} \quad (6.3)$$

$$\eta = \begin{cases} h_{ci}(t), & \text{if same move} \\ 0, & \text{if different move} \end{cases} \quad (6.4)$$

$$m_i(t+1) = m_i(t) + \alpha_i(t) \delta \eta [s(t) - m_i(t)] \quad (6.5)$$

MST has been shown to fit better to certain distributions and to provide more flexibility to changes in the input distribution than a fixed spatial topology[40]. The computational time for the MST algorithm is $O(n^2)$ and might be prohibitive for large problems. Since the relative order of the codebook vectors changes slowly, the MST neighborhood reordering needs to take place only a few times. This modified SLVQ algorithm (Algorithm 6.1) is used in the control tasks described below.

Algorithm 6.1 SLVQ-MST

Input outcome is the reward at T, the end of an episode

SLVQ (outcome,T)

while (T>0)

Codebook $\leftarrow \{m,a,Q,\alpha\}_T$

$\delta \leftarrow r + \text{outcome} - Q_{\text{Codebook}}$

t \leftarrow T

trace \leftarrow 1

Codebook.MST_PROPAGATE(δ, s_t)

while (t>0)

codebook $\leftarrow \{m,a,Q,\alpha\}_t$

codebook.BACKUP(trace δ, s_t)

trace \leftarrow trace λ

t \leftarrow t - 1

T \leftarrow T - 1

outcome $\leftarrow \gamma Q_{\text{Codebook}}$

BACKUP (delta, state)

$m_{t+1} \leftarrow m_t + \alpha_t \text{delta}(\text{state} - m_t)$

$Q_{t+1} \leftarrow Q_t + \alpha_t \text{delta}$

decay α

MST_PROPAGATE (delta, state)

neighbors \leftarrow neighbors(radius)

for each neighbor

$h \leftarrow \text{Similarity}(\text{state}, \text{neighbor})$

neighbor.BACKUP(hdelta, state)

6.3 The Cart Centering Problem

The cart centering problem is a control problem where the goal is to find a policy or set of rules towards a fixed point in the state space from any other point rather than finding an

optimal trajectory from an initial starting point as in the mountain car problem.

6.3.1 Problem Description

The cart centering problem involves pushing a frictionless cart of mass m on a one-dimensional track until the cart becomes centered (Figure 6.1). A force of fixed magnitude F , a *bang-bang* force, is applied either to the right or to the left at each time step t . The two state variables for this system are the position x and velocity v . The object is to find an optimal policy to apply these bang-bang forces from initial random conditions until the cart becomes centered at approximately position 0.0 and velocity 0.0. Under the optimal policy, the bang-bang force is to be applied to the right if

$$-x > \frac{v^2 \text{Sign}(v)}{2|\frac{F}{m}|} \quad (6.6)$$

and to the left otherwise.

The movement of the cart is governed by the following equations:

$$a(t) = \frac{F(t)}{m}$$

$$v(t + 1) = \text{bound}(v(t) + \tau a) \quad (6.7)$$

$$x(t + 1) = \text{bound}(x(t) + \tau v(t)) \quad (6.8)$$

where $a(t)$ is the acceleration computed by Newton's Law at time t , τ is the size of the time step (0.02 s), m the mass of the cart (2.0kg), and $|F|$, the magnitude of the force (1.0N). The control variable, left (-1.0) or right (+1.0), acts as a multiplier on the force F at time t .

Positions are initialized randomly between -0.75 meters and $+0.75$ meters and velocity is initialized to 0. The bound operation enforces the range of the variables.

The optimal solution to this problem was found with genetic programming[43] where the experimental settings of this problem were taken. The payoff value was the time it took to center the cart. The optimal solution was found after 1500 evaluations of S-expressions composed of a limited function set $\{+, -, *, /, \%, \text{ABS}, \text{GT}\}$.

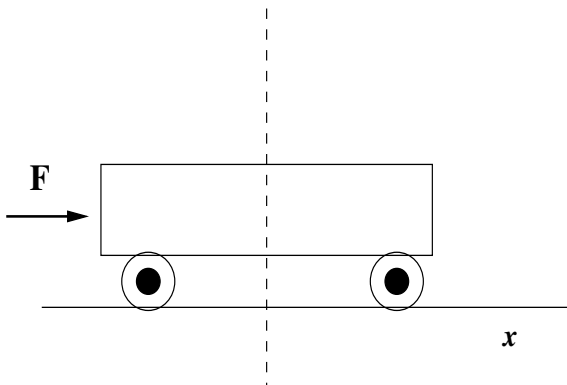


Figure 6.1: Cart centering problem

6.3.2 Distance Function

To evaluate the predictive power of the cosine and euclidean distance function for continuous control tasks, we randomly generated 1000 codebook vectors and set the prototype label as the action to take according to the optimal solution. We then tested the distance functions with 100 randomly generated patterns and compared the prototype label they returned when matching with the distance function against the action to take under the optimal solution.

Each prototype vector m has two dimensions, i.e. the position x and the velocity v . The

cosine distance function ($-1 < d < 1$) between two normalized prototype vectors m_1 and m_2 is defined as:

$$\frac{x_1x_2 + v_1v_2}{\sqrt{x_1^2 + v_1^2}\sqrt{x_2^2 + v_2^2}} \quad (6.9)$$

When the magnitude of the vectors is important, the euclidean distance function ($d > 0$) is more appropriate:

$$\sqrt{(x_1 - x_2)^2 + (v_1 - v_2)^2} \quad (6.10)$$

The results in Figure 6.2 shows the superiority of the Euclidean distance function in this problem and that the number of codebook vectors does not significantly affect the accuracy of the function.

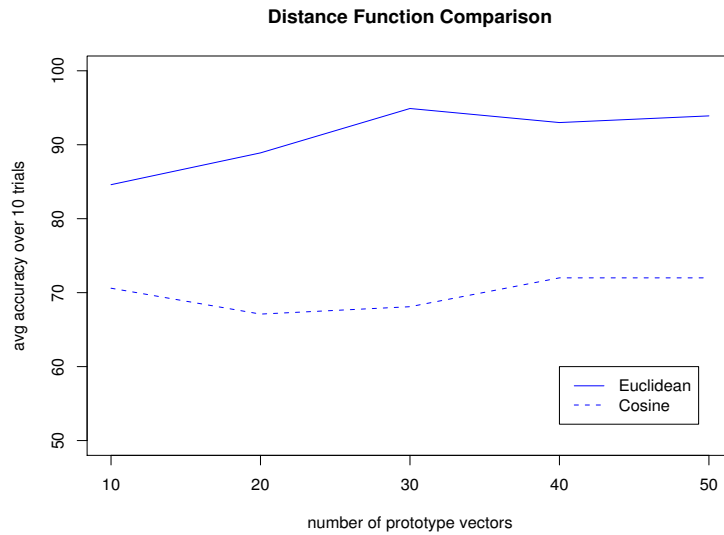


Figure 6.2: Cosine and Euclidean distance function comparison

6.3.3 Empirical Evaluation

The optimal solution to this problem starting from different random positions and null velocity averages 76 steps in 100 trials. In our experiments, the Euclidean distance to the goal (0,0) is propagated back as a negative reward if the cart does not get approximately centered (± 0.05) within the maximum number of steps allowed (110 steps). Otherwise, +1.0 is propagated back as a positive reward. The two-dimensional codebook vectors (position and velocity) are initialized randomly within some certain distance of each other by partitioning the input space into equal 3x3 areas. The Pareto optimality of the moves along the pattern similarity and action value dimension is used for selecting the next move:

$$eval(s, a) = \frac{Similarity(s, m)Q(m, a_m)}{\sum_m Similarity(s, m)Q(m, a_m)} \quad (6.11)$$

where s is the current state, a the candidate action, and m the candidate codebook vector. This approach combines discrimination and the reinforcement feedback and boosts exploration at the beginning of the learning process. The learning rate α decayed monotonically. SLVQ averages 77 steps in 100 trials after training for 1500 episodes over 3 runs with an average accuracy of 88% measured as the average number of completed trajectories within the maximum number of steps. Figure 6.3 shows the trajectories given by the optimal solution for points starting at various positions and null velocity. Figure 6.4 shows the SLVQ trajectories for the same initial starting points. Due to finite accuracy both the computed optimal and SLVQ solutions end within an area of small uncertainty (± 0.05) around the optimum point (0,0).

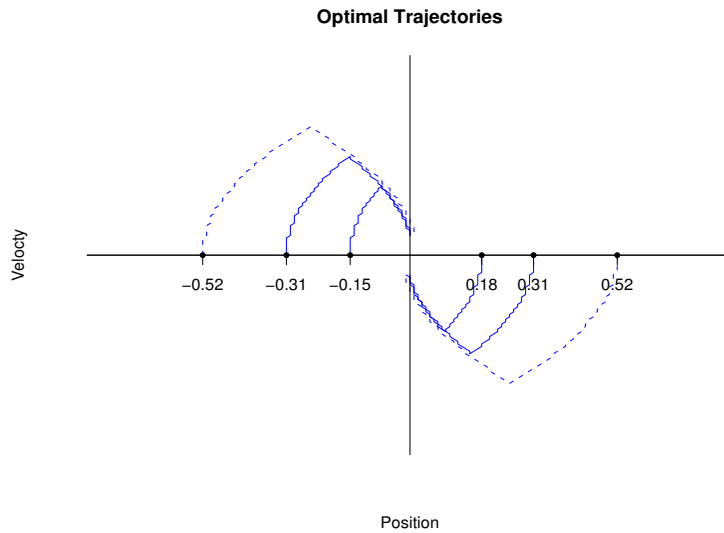
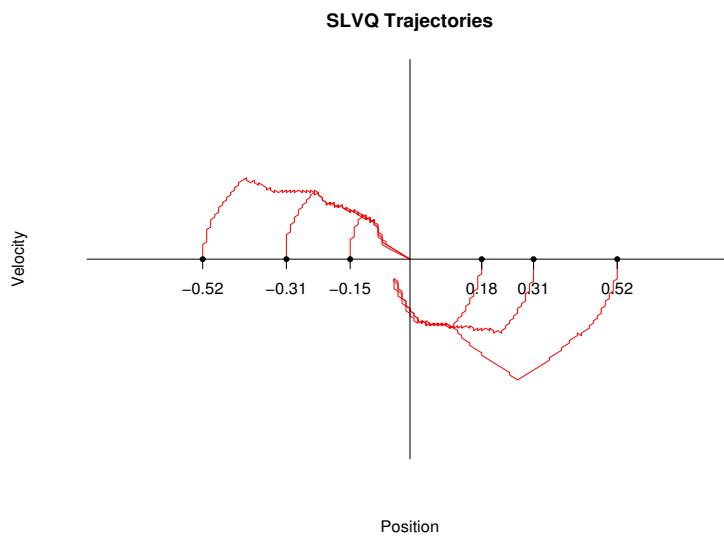


Figure 6.3: Optimal trajectories

Figure 6.4: SLVQ trajectories
 $\alpha=1.0, \lambda=0.07, \gamma=1.0, m=200$

6.4 The Mountain Car Problem

This control problem was first introduced as the *puck-on-the-hill* problem in [54]. A frictionless puck moves on a bumpy surface acted on by gravity and a thruster. This problem has been reworked as the *mountain car* problem in [83] as an undiscounted task. The objective is to drive past the top of the mountain but, even at maximum thrust, the engine is not strong enough to get up the steep slope (Figure 6.5). The only way to solve this problem is to move *away* from the goal. This is a classic example of delayed rewards where a greedy strategy would fail.

6.4.1 Problem Description

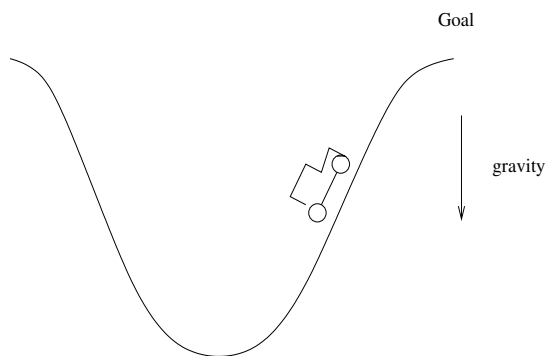


Figure 6.5: The mountain car problem

Like the cart centering problem, this problem has two continuous state variables: the position x_t in the range $[-1.2, 0.5]$ and the velocity v_t in the range $[-0.07, 0.07]$. However, there are three possible actions: forward, backward and none, i.e. $a_t \in \{+1, -1, 0\}$. The absence of direction reduces the velocity just like when we stop accelerating before stopping. The reward is -1 on all time steps in [83] and as an exponentially decreasing function

of the proximity of the goal in [54]. The position and velocity are initialized randomly at the beginning of each episode. The physics of the problem are as follows:

$$x_{t+1} = \text{bound}(x_t + v_{t+1}) \quad (6.12)$$

$$v_{t+1} = \text{bound}(v_t + 0.001a_t - 0.0025\cos(3x_t)) \quad (6.13)$$

The bound operation enforces the range of the variables.

6.4.2 Empirical Evaluation

A solution to this problem using Sarsa and tile coding is given in [88, 83]. Tile coding discretizes a continuous input space into finer degrees of resolution with a fixed number of binary features. This encoding of the state space evolved from the state representation of CMACs[1] where the input space is mapped onto overlapping binary features which are activated depending on the size of their receptive fields. A weight representing the expected value of each tile is associated with a binary feature and the approximate value function can then be computed as a linear combination of those weights. The reward is -1 on each step and the total feedback is divided evenly among each tiles. A near optimal policy (104-109 steps) is achieved in less than 100 episodes with 9 tilings of 9x9 offsets starting at position -0.5 and velocity 0.

In our experiments with SLVQ, the relative position to the goal is propagated back as a negative reward if the car does not make it within the maximum number of steps allowed (1000 steps). Otherwise, the relative number of steps is propagated back as a positive reward since the starting position remains fixed. There is no intermediate reward/penalty and no discount. The two-dimensional codebook vectors m (position and velocity) are

initialized randomly within some certain distance of each other by dividing the input space into equal 3x4 areas. The Pareto optimality of the moves was used to provide a source of exploring starts (see 6.3.3). The Euclidean distance determines the winning codebook vector. The learning rate α was held constant. Figure 6.6 shows the trajectories of the codebook vectors in the learning phase at 111 steps. Figure 6.7 shows the action taken by the greedy policy with the active codebook vectors. A near optimal policy (111-119 steps) is achieved in less than 100 episodes starting at position -0.5 and velocity 0.

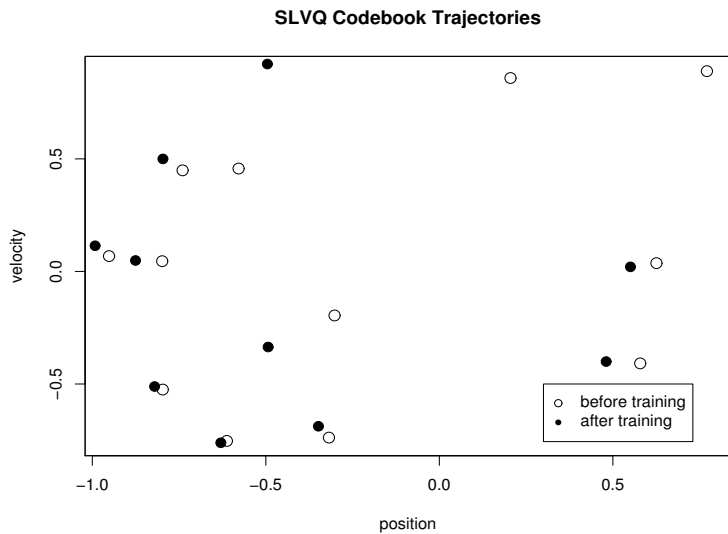


Figure 6.6: SLVQ codebook trajectories for the mountain car problem
 $\alpha=0.1, \lambda=0.07, \gamma=1.0, m=12$

SLVQ learns to partition the input space into a Voronoi tessellation according to the empirical distribution of the data and the requirements of the task. It is therefore doing more than the fixed representation of tile coding by trying to learn the representation as well as the task but provides a more flexible computational approach for problems of high dimensionality. The fixed size discretization of tile coding is not desirable when different granularity

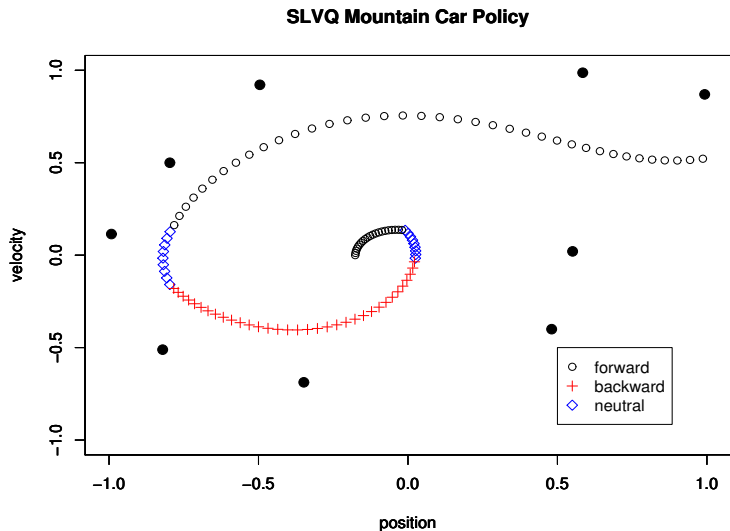


Figure 6.7: SLVQ mountain car policy and relevant codebook vectors at 111 steps

is needed in different regions of the state space. It is also possible with the SLVQ approach to bias the search space with a proper initialization of the codebook vectors.

6.5 Comparison with QLVQ

The theoretical justification of an on-policy approach versus an off-policy approach was outlined in Chapter 3. Indeed, QLVQ[11], as an off-policy procedure combined with LVQ (see Algorithm 6.2), fails to converge in the mountain car problem with similar parameters and initializations of the codebook vectors. Figure 6.8 shows the policy obtained by QLVQ at 182 steps. Likewise, the performance of QLVQ in the cart centering problem[11] is far from optimal with an average of 94 steps in 100 trials and 32% accuracy. For problems

where delayed rewards is important, on-policy learning with eligibility trace has a faster performance and better convergence properties.

Algorithm 6.2 QLVQ

- Initialize all $Q(m, a)$, the learning rates α and β , and discount factor γ
 - While stopping condition is false
 1. Randomly generate state s .
 2. $m \leftarrow f(s)$
 3. Select an action a to execute.
 4. Execute action a , and let s' be the next state, a' the next action, and r the reward received.
 5. $m' \leftarrow f(s')$
 6. Update $Q(m, a)$

$$\delta_Q \leftarrow [r + \gamma \max_b Q(m', a') - Q(m, a)]$$

$$Q(m, a) \leftarrow Q(m, a) + \alpha(t)\delta_Q$$
 7. $m(t+1) \leftarrow m(t) + \beta(t) [s-m(t)]$ if $\delta_Q > 0$
 $m(t+1) \leftarrow m(t) - \beta(t) [s-m(t)]$ if $\delta_Q < 0$
 8. Reduce monotonically the learning rates α and β as a function of t .
 9. Update the policy function π such that

$$\pi(m) \leftarrow \arg \max_{a \in A} Q(m, a)$$
-

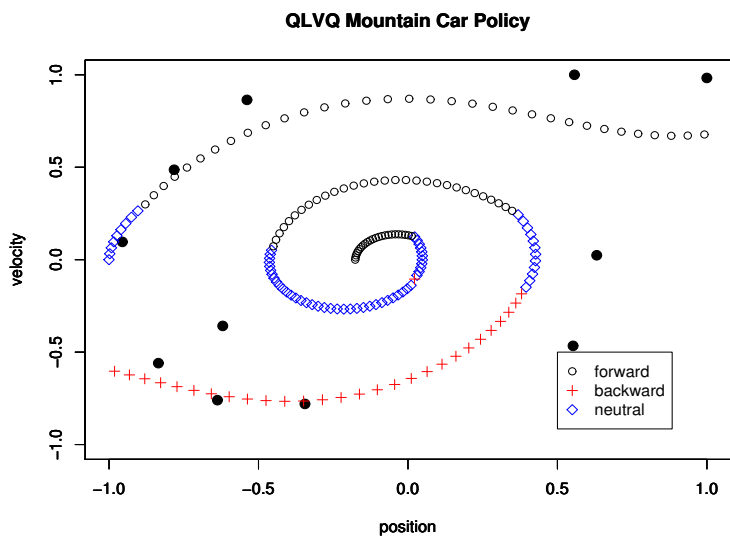


Figure 6.8: QLVQ mountain car policy and relevant codebook vectors at 182 steps

Chapter 7

Intelligent Exploration

with less than 15 stones in danger, tenuki.

On-policy reinforcement learning provides adaptive on-line performance which is a characteristic of intelligent systems and lifelong learning. For efficient and “believable” on-line performance, an exploration strategy also has to avoid cycling through previous solutions and know when to stop without getting stuck in a local optimum. This chapter stresses action inhibition as a characteristic of intelligent exploration.

7.1 Motivation

Unlike dynamic programming, an exhaustive sweep of the state space is not necessary for convergence in reinforcement learning with an efficient exploration strategy. However, the use of function approximation introduces a new dimension in the exploration strategy problem. The modification of the weight vectors in the trial and error exploration of the search space introduces some information loss. Some weight vectors become undifferentiable. This is the learning cost of exploration. An intelligent exploration of the search space would minimize this cost by (1) focusing the search by examining only relevant portions of the problem space and (2) sustain diversity by relevant modification of the weight vectors.

The overall diversity of a set of codebook vectors, or collective entropy, can be computed as $-\sum_i^n p_i \log_2 p_i$ where p_i is computed as the similarity of a codebook i to the normalized identity vector. Figure 7.1 shows the collective entropy for different exploration rates with the softmax exploration method (see 5.3) in 7x7 games against Wally.

The claim of this chapter is that exploration can be a destructive operation when reinforcement learning is coupled with a function approximator and, unlike genetic algorithms, there is no redundancy in the overlap of generations to protect against this entropy. For on-policy approaches, exploration guides the convergence of the learning process since it controls the update mechanism. In other words, the choice of a move affects the future choice of other moves. It is also an opportunity to integrate prior knowledge with the learning process. This chapter examines different exploration methods and issues and presents exploration strategies for reinforcement learning based on tabu search which specifically refrains from unnecessary exploration. With an adaptive exploration strategy, a behavior policy evolves into an optimal policy.

7.2 Exploration Methods

The trade-off dilemma between exploration and exploitation, as illustrated by the two-armed bandit problem, has been shown[35] to have an optimal strategy in the allocation of an exponential number of trials to the observed better arm with respect to the observed worse arm. With respect to sampling in reinforcement learning, this means that the observed best action should progressively receive more trials. In this context, two types of on-policy control methods are distinguished:

- *Diffusion* of the target policy towards other candidate actions. As training progresses,

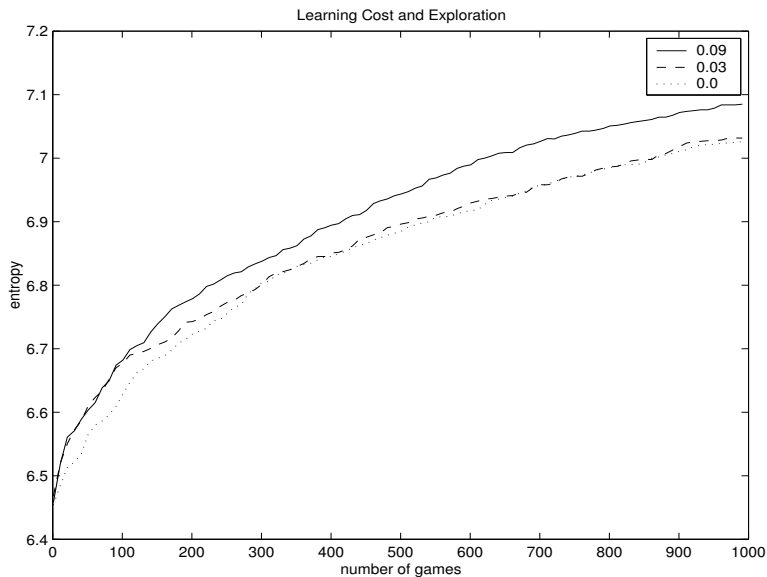


Figure 7.1: Entropy and exploration relationship

this diffusion lessens. The metaphor here is that of looking at something from far away and then getting closer.

- *Permutation* of the behavior policy toward the target policy. The metaphor here is to turn each stone systematically before moving forward.

This distinction is typical of problem solving approaches to exploration: instead of picking the best, or what we think is the best, pick something slightly different if possible or try to permute the steps. One approach might be more natural than others depending on the domain. A basic distinction has been made between *undirected* and *directed* exploration approaches[96] where randomness might or might not be an ingredient to exploration. However, even directed exploration based on search statistics can appear random if content difference is ignored.

7.2.1 Bonus/Penalty Approaches

Those approaches are typical of a *diffusion* of the target policy. A bonus/penalty decaying with time is added to the evaluation of a move $eval(s, a)$. Bonus/penalty based approaches combine exploration and exploitation by maximizing a linear combination:

$$eval(s, a) \pm \tau \cdot explor(s, a)$$

where τ is the exploration rate ($0 \leq \tau \leq 1$) and $explor(s, a)$ is the bonus/penalty strategy followed and $\tau \cdot explor(s, a)$ decays in function of the number of times a tuple $\{m, a, Q, \alpha\}$ was selected to be part of the solution.

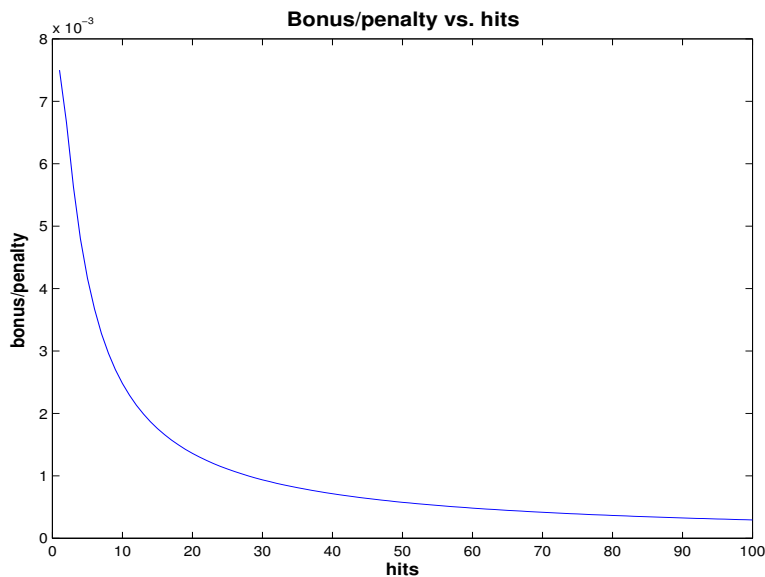


Figure 7.2: Bonus factor for frequency-based exploration strategy
($\tau = 0.03$)

Bonus-based exploration strategies have the characteristic that the exploration rate depends

on the uncertainty of the local environment, i.e. it is stronger when there is no apparent clear winner. It is therefore self-starting and self-adjustable. Unfortunately, it does not escape easily a local minimum if there is a clear local winning move. The following strategies experimented with were as follows:

Softmax bonus $random(0, 1)$

Recency bonus $\frac{1.0}{(1.0+Last(c))}$

Frequency bonus $\frac{1.0}{(1.0+Hits(c))}$

Conscience penalty[21] $\frac{Hits(c)}{Candidate(c)}$

where c , is the codebook tuple $\{m, a, Q, \alpha\}$ under evaluation, $Last(c)$ is the last episode (game) that c was selected, $Hits(c)$ is the number of times that c was selected and $Candidate(c)$ is the number of times that c was a possible candidate for selection. The recency, frequency and conscience methods constitute memory-based methods and hence are *directed* methods.

7.2.2 Comparative Results of Bonus/Penalty Approaches

Following the experimental methodology outlined in 5.3, Figure 7.3 shows the results on a 5x5 against Wally with a random initialization of the codebook vectors and comparing with MCGo (see Chapter 3). There is a statistical difference in the performance of all the exploration methods vs. MCGo.

Figure 7.4 shows the results on a 7x7 against Wally and comparing with MCGo with initialization of the codebook vectors with real games. For the parameters shown, there is a

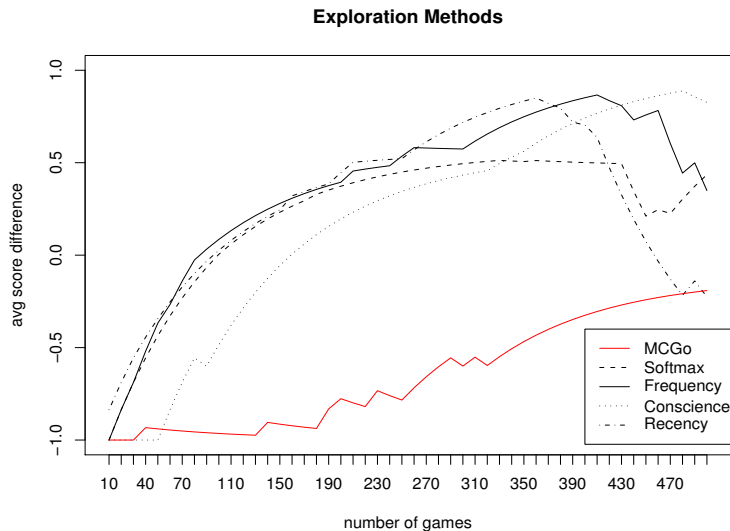


Figure 7.3: 5x5 bonus/penalty exploration methods vs. Wally with random initialization on a 5x5 board ($\tau = 0.09, \alpha = 0.05, \gamma = 0.99, \lambda = 0.05$)

statistical difference between MCGo and the conscience strategy ($p < 0.0005$), the softmax strategy ($p = 0.03464$), and the frequency strategy ($p = 0.003074$) but not with the recency strategy.

7.2.3 Training

Training is a behavioral type of directed exploration as opposed to a memory-based type of exploration. Good training should provide non-random variation from expert play[27]. Self-play has been shown to provide little variation in Go and overlook important areas of exploration[74]. Figure 7.5 shows the result of self-play training against Wally and Minimax in 7x7 games with the softmax exploration strategy 7.2.1. Although SLVQ's play

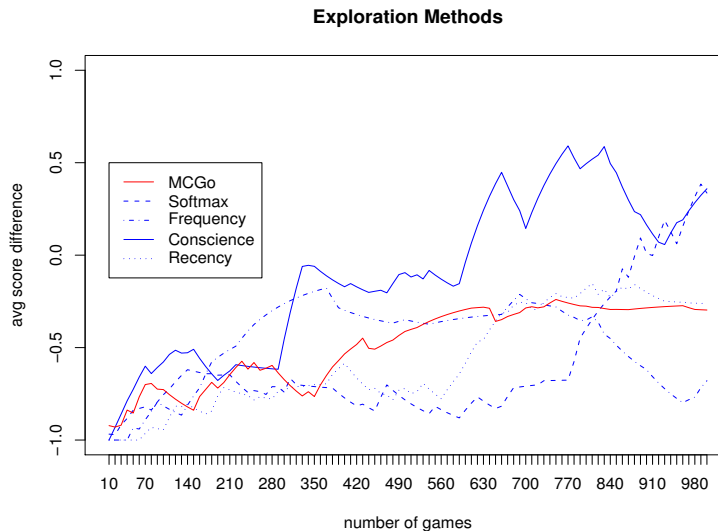


Figure 7.4: 7x7 bonus/penalty exploration methods vs. Wally with real games initialization on a 7x7 board ($\tau = 0.09, \alpha = 0.2, \gamma = 1.0, \lambda = 0.07, m = 1000$)

performance improves slowly through self-play and narrowly wins over Wally, it reaches a plateau after about 10,000 games.

Does training on simpler problems help in solving the original hard problem? This idea is known as shaping in behavioral psychology and has been shown to boost the reinforcement learning process for complex problems[64]. Handicap Go where stones of the weaker player are placed on the board prior to the game makes the game simpler (by making the goal, i.e. winning, easier to reach) without breaking it up into independent subtasks. Figure 7.6 shows that playing against a random player is more conducive to learning with SLVQ than making the game itself simpler with handicap stones. The diversity provided by a random player prevents overfitting and thus enables a knowledge transfer to take place.

Training works because it provides positive feedback. Both types of feedback are necessary



Figure 7.5: Self-play training with softmax in 7x7 games vs. Wally and minimax ($\tau = 0.13$, *decaying* α , $\gamma = 1.0$, $\lambda = 0.5$)

for learning to take place. Active learning, which purposely look for informative examples, is necessary when one type of feedback is scarce. The necessary experience must be sought out sometimes. For example, boosting[73] changes the distribution of the examples to give more weight in the learning process to the misclassified examples. As described above, each unfruitful example has a cost in competitive learning because of the reweighting of the codebook vector. This cost can be minimized if truly different examples are sampled.

7.3 Tabu Search Exploration for Reinforcement Learning

The convergence of reinforcement learning based on stochastics depends on visiting every state infinitely often. The Bellman optimality equation is predicated on the value function

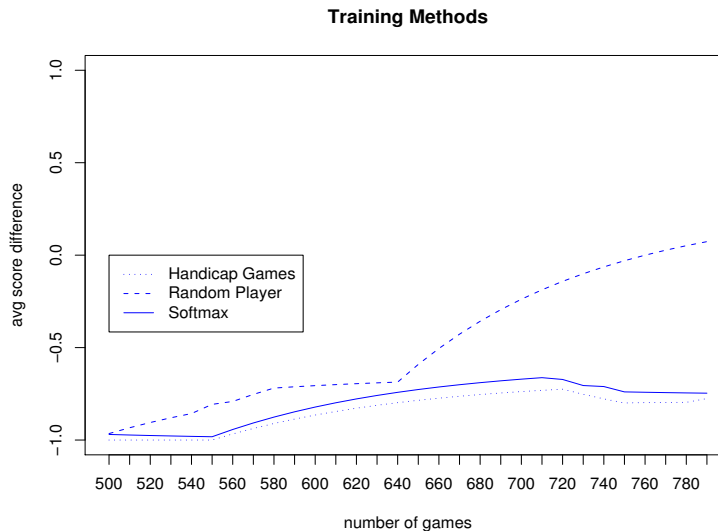


Figure 7.6: Knowledge transfer from two different training methods for the first 500 games: 2 handicap stones and random opponent ($\tau = 0.09$, $\alpha = 0.1$, $\gamma = 0.99$, $\lambda = 0.07$)

representing a good approximation of its expected return.¹ This is not obviously feasible for large state space. Exploration helps with convergence not only in guiding what to explore but also in deciding what not to explore. Tabu search (TS) explicitly addresses this aspect.

7.3.1 Meta-Level Search

Meta-level search addresses the question of how to search in order to learn. Genetic algorithms, simulated annealing and tabu search are considered metaheuristics, or general

¹Recall in dynamic programming, the Bellman optimality equation for V^* is

$$V^*(s) = \max_a \sum_{s'} p_{ss'}^a [R_{ss'}^a + \gamma V^*(s')]$$

Reinforcement learning replaces prior knowledge of the transition probabilities by an estimation based on sampling next-state returns.

search strategies, built on heuristics for the application domain.

Genetic algorithms evolve a population of solutions. They search through the space of policies, i.e. collections of mappings from state to actions. For example, SANE[56] evolves neural networks blueprints, i.e. combinations of neurons, from partial neuron solutions. Each individual in the population, as a combination of neurons, is a potential optimal policy. In this approach, the next move is always the next best move. Genetic algorithms search at the population level through selection, mutation and crossover. In contrast, simulated annealing and tabu search are exploration strategies of the environment through the selection of moves. They search for promising moves and not promising combinations of moves, i.e. policies. The best policy emerges from interactions of the task with the environment only and not from internal interactions as in genetic algorithms.

Tabu search[29] is a meta-heuristic that, unlike simulated annealing and genetic algorithms, uses knowledge of the history of the search instead of blind randomization to navigate through the search space and escape local optimum solutions. It uses knowledge by remembering characteristics of state-action pairs. It generalizes the memory-based methods described above (7.2.1). We must be able to recognize a solution (aspiration criteria) and we must be able to focus our attention on promising regions of the search space while avoiding cycling through previous solutions. Technically, a tabu search approach involves picking the best candidate that is not marked *tabu* from a candidate list, i.e. that is not on the tabu list, and override the tabu restriction if the candidate satisfies the *aspiration criteria* (see Algorithm 7.1). Theoretically, a tabu search approach involves recognizing what is right, what is new, and what is similar in the candidate solutions. In contrast, randomization is considered a weak diversification method. The tabu search approach has a principled way of choosing exploitation over exploration. For problems where the state space itself is

exponential in nature, the speed-up of frequency-based methods as noted in [95] does not reduce the search to a polynomial case.

Algorithm 7.1 Tabu Search

1. Choose an initial feasible solution i
 2. Generate candidate solutions in the neighborhood of i
 3. Filter out solutions that do not meet the aspiration criteria and that are marked *tabu*
 4. Select the best candidate solution j .
 5. If candidate solution j improves on solution i , set $i = j$
 6. Update tabu status of candidate moves and aspiration criteria
 7. If stopping criteria is met, stop, else go to 2
-

Tabu search and simulated annealing belong to the *permutation* type of exploration methods where the selection of moves is not a function of their evaluation. The “least” tabu move is selected if no moves satisfy the aspiration criteria and all moves are tabu. A tabu search entails the definition of the aspiration criteria and the specification of the tabu list structure which are discussed below.

7.3.2 Aspiration Criteria

What could be an aspiration criterion in a reinforcement learning approach? In other words, how do we recognize such a compelling move in an on-line learning approach versus a *best* move according to current estimate? The consistency of a move with a positive outcome can be measured after a certain number of trials within a certain window size if a non-overlapping confidence interval with other candidate moves can be found:

$$\left[\bar{Q} - t_{n-1} \frac{s}{\sqrt{n}}, \bar{Q} + t_{n-1} \frac{s}{\sqrt{n}} \right]$$

given $c = \{m, a, Q, \alpha\}$, a codebook tuple, where \bar{Q} is the sample average of Q_c , s the sample standard deviation of Q_c and t_{n-1} the *critical value* of the t distribution with $n - 1$ degrees of freedom for a specified confidence level. For a 95% confidence level and $n = 30$, $t_{n-1} = 2.0452$. The best move with a non-overlapping confidence interval constitutes such a compelling move.

An aspiration criterion defined by the predictability of a positive outcome models the preferential treatment of attentive behavior to states with high outcome reliability[85]. Selective attention, the capability to discriminate between input states, has been shown to increase with the consistency of the outcome and decrease with variability. An aspiration criterion is necessary to focus the search for convergence.

7.3.3 Tabu List Strategies

Restrictions on the candidate moves structure the search space to explore variations in a local neighborhood. This is equivalent to systematically turning each stone in a region of the search space before looking elsewhere. When restrictions are temporal restrictions, a tabu list stands for a short-term memory. Otherwise, a tabu list is similar to a nearest-neighbor type of approach where the neighborhood can be defined in different ways. The tabu list structure is the source of diversification as well as intensification modulated by the tabu tenure parameter. Some different approaches to structuring a tabu list in the context of SLVQ are introduced below.

Simple Tabu Search (STS)

Similarly to the temperature in simulated annealing, a tabu tenure is defined over the moves on the tabu list. The policy becomes greedier as the tabu tenure declines to 0. Candidate moves are ordered according to their evaluation and the best move that is not tabu, i.e. not on the tabu list, is picked and thereafter becomes tabu for the length of time specified by the tabu tenure parameter by setting its tabu status. All other moves on the tabu list see their tabu status decline (Algorithm 7.2). To avoid cycling through previous sequences of moves, the tabu tenure of each state-action pair is set with some random variation of each other. As opposed to the bonus/penalty approach, the rate of exploration can be controlled to match the training time as in simulated annealing by adjusting the length of the tabu tenure as an initial global parameter. The length of the tabu tenure determines the degree of intensification or diversification and can vary adaptively according to the reinforcement feedback. The tabu tenure parameter decays locally in proportion of the number of times h that an action has been used:

$$TabuStatus = TabuTenure * 0.99^h$$

The Elite List Strategy

Aspiration status and tabu status are often mirror images of each other. Exclusively selecting one type of move renders the others “tabu”. This strategy combines seamlessly an on-line and batch approach to reinforcement learning. The best moves are selected repeatedly and updates are stored in a buffer until the aspiration tenure of the moves expires. Once updated, their aspiration status might change and other moves get a chance to be selected.

Algorithm 7.2 STS Move selection

 PICKMOVE(board, legalMoves)

1. matches \leftarrow all legal move tuples matching the board
 2. bestMatch \leftarrow $\arg \max_{similarity}(matches)$
 3. tabuMatches \leftarrow tuples with tabu moves (tabu status > 0)
 4. permissibleMatches \leftarrow non-tabu move tuples
 5. if bestMatch satisfies aspiration criteria
 tuple \leftarrow bestMatch
 jump to 7
 6. tuple \leftarrow $\arg \max_{similarity}(permissibleMatches)$
 7. tuple.tabuStatus \leftarrow tabuTenure
 8. for all tuple \in tabuMatches
 tuple.tabuStatus \leftarrow $\max(tuple.tabuStatus - 1, 0)$
 9. return tuple.move
-

This buffering of the backups has the added benefit to stem out noise from subsequent moves of the game.

$$Q = Q + \sum_{t=1}^{\text{tabu tenure}} \Delta Q_t$$

$$m = m + \sum_{t=1}^{\text{tabu tenure}} \Delta m_t$$

Concept-based Tabu search

A tabu tenure can be computed not only on search attributes but also on the move attributes themselves. This is where the opportunity to integrate domain knowledge arises and the

capability to select truly “different” moves based on content arises.

The operational characteristics of moves in Go can express different purposes or strategies (see Appendix A):

Claim Increase the player’s sum of influence values

Defend Increase the player’s sum of liberties

Connect Join two groups

Invade Decrease the opponent’s sum of influence values

Attack Decrease the opponent’s sum of liberties

More abstract features, such as safe or expert move, could also be determined off-line with a supervised learning approach[13]. Here, we do not try to evaluate the moves themselves but use their categorization to guide the exploration of the search space.

Since the prototype vector m gets updated, action a can represent different move strategies and the determination of the move strategy has to be made in relation to the input vector. Once a tuple $\{m, a, Q, \alpha, T, TabuTenure\}$ is selected and put on the tabu list as well as candidate moves of the same strategy, the tabu status of tuples with a different move strategy decline. Because of the generality of the attribute, if nothing can be selected, a new tuple with a legal move attribute that is not tabu can be generated before selecting the least tabu move attribute that match more closely the input vector. An example follows.

The categorization of the candidate moves in Figure 7.7 is as follows:

Attack A,B,C,D,E,F

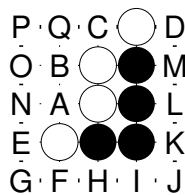


Figure 7.7: Candidate moves in life-and-death pattern

Invade N,O,P,G,Q

M, L, K, I, J and H do not accomplish any purpose and are therefore removed from the candidate list if included. If move F is selected, the tabu status of the corresponding tuple is set to the tabu tenure parameter. The tabu status of moves A, B, C, and D are set as well. Table 7.1a shows the tabu list after the first move with a global tabu tenure parameter of 3. Let's assume White plays A afterwards. Black will then play an invading move if any are on the candidate list since all attacking moves are tabu. Let's assume Black plays O . Table 7.1b shows the tabu list after this second Black move assuming codebook vectors for all the invading moves are found. Tuples with a tabu status of 0 are removed from the tabu list. The tabu status of the tuple associated with move F won't be reconsidered until it becomes a candidate move again.

Table 7.1: Tabu lists for the first 2 moves in Figure 7.7

a	{F,3} {B,3}{C,3}{D,3}{E,3},{A,3}
b	{O,3}{N,3}{P,3}{Q,3}{F,2}{B,2}{C,2}{D,2}{E,2}

7.3.4 Empirical Evaluation

The experimental methodology outlined in Chapter 5 is followed here as well. Figure 7.8 shows the performance of the TS methods for the pattern in Figure 7.7. In this life-and-death pattern, SLVQ plays black against minimax. Even that black starts at a disadvantage, it wins against its minimax opponent under all the TS strategies but not with the softmax strategy. The elite strategy focuses the fastest on this problem. Figure 7.9 shows the performance of TS exploration methods compared with the softmax strategy in 7x7 games. The TS strategies search the space more effectively than the softmax exploration strategy and generally arrive at a solution faster. There is a statistical difference between concept-based TS and softmax ($p = 0.0003046$). For computational purposes, concept-based TS categorizes a move by considering only its direct 4 neighbors and the influence values of the current board configuration:

CLAIM move into friendly influence value

DEFEND at least one friendly neighbor and at least 2 liberties

CONNECT exactly 2 friendly neighbors

INVADE move into unfriendly influence value

ATTACK at least 1 unfriendly neighbor

Figure 7.10 shows the offline results from self-play training against Wally and Minimax over 3000 games with STS. There is a statistical difference ($p = 0.03521$ and $p = 0$, respectively) over the results obtained with the softmax exploration strategy (7.5).

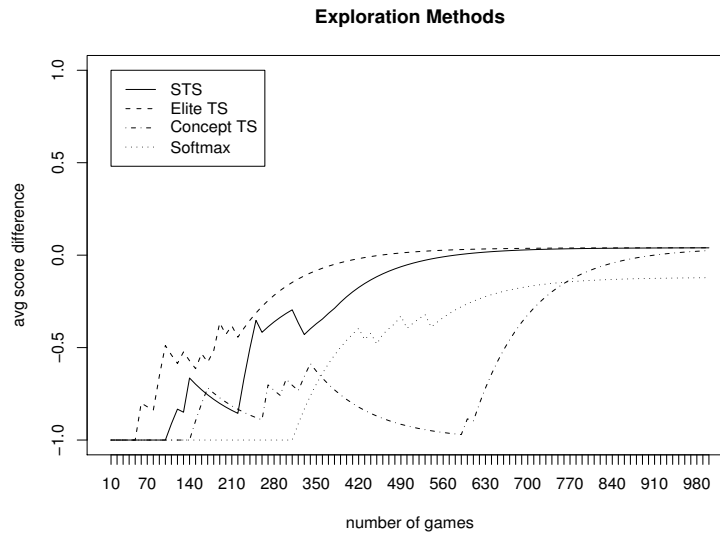


Figure 7.8: Exploration methods vs Minimax for pattern in Figure 7.7
 ($TabuTenure = 5, \alpha = 0.1, \gamma = 1.0, \lambda = 0.2$)

7.4 Discussion

A thorough exploration of the search space as found in tabu search(TS) gives a mechanism for escaping local minimum states of gradient search algorithms and avoid cycling through previous sequences of solutions. As a directed search method based on content rather than search statistics, TS provides an opportunity to integrate domain knowledge and concept learning in RL algorithms. By focusing on promising regions of the search space quickly, TS methods speed up the convergence of on-policy RL algorithms. In addition, TS serves also as a training method for game playing by suggesting non-random variations around the best candidate move.

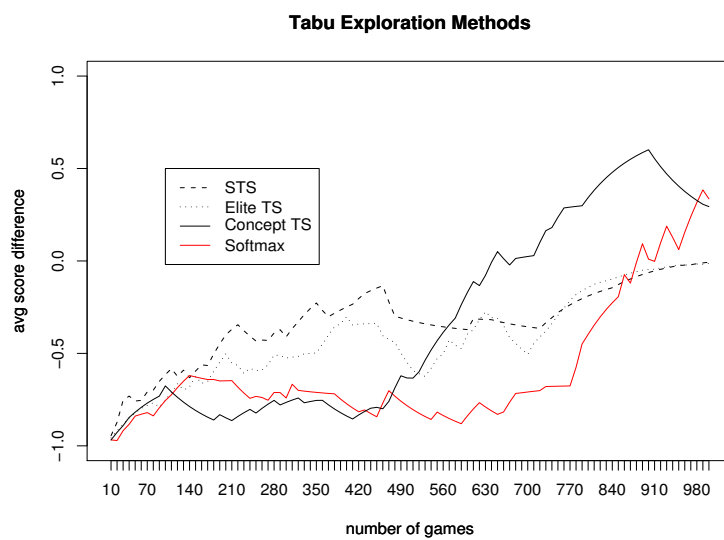


Figure 7.9: Tabu exploration methods vs Wally on a 7x7 with game initialization of the board ($TabuTenure = 3, \alpha = 0.2, \gamma = 1.0, \lambda = 0.07, \tau = 0.09$)

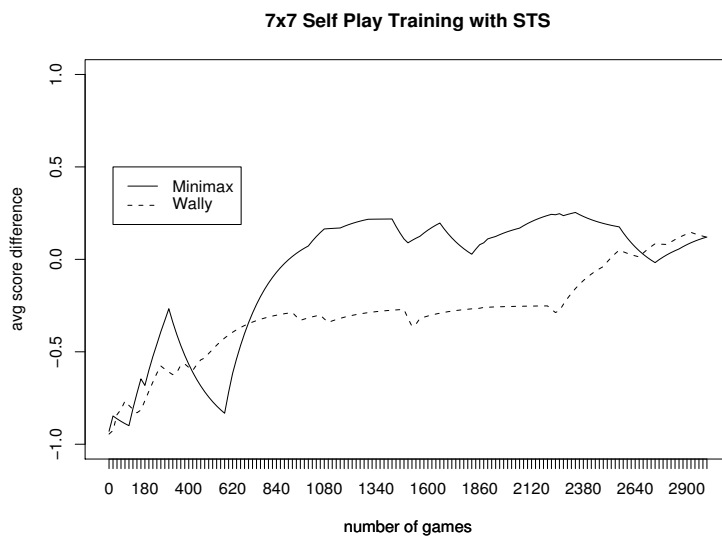


Figure 7.10: Self-play training with STS vs. Wally and Minimax
($TabuTenure = 5$, decaying $\alpha, \gamma = 1.0$, $\lambda = 0.5$)

Chapter 8

Learning by Parts

corner, side, centre.

The holy grail to the game of Go and other complex problems has been to learn on simpler problems and reuse this knowledge recursively or hierarchically. This approach has deeply-seated cognitive roots in problem-solving tasks. It is useful to take the multi-agent systems perspective in distributed problem solving¹ because of the social agency metaphor and its implied absence of global control in its division of labor. The SLVQ approach can be extended to heterogenous agents with different views of the current situation. Each candidate move and supporting pattern reference vector represents an agent vying for action. In the game of Go, this will allow us to represent the relative urgency of tactics, i.e. local moves, and strategy, i.e. global moves. This chapter extends SLVQ with learning by parts using a game-theoretical approach. An adaptive Wally is built using this approach. Adaptive Wally is then compared against the single-agent approach.

¹See[25] for a comparison between multi-agent systems and distributed problem solving.

8.1 Related Research

It can be said that research related to interaction of parts in a decentralized manner has its origin in the subsumption architecture[6] and behavior-based intelligence[48, 7]. This cognitive architecture encodes the search space with different behaviors or competences. Each behavior is attentive to a subset of the state space. Behaviors are prioritized in the sense that if one behavior fires, it will preempt other behaviors from firing. The applicability as well as the prioritization (arbitration) of those behaviors is handcoded. It was found in the box-pushing robot example [50], that learning each behavior separately with Q-learning in a subsumption architecture framework is superior to learning the entire task as one monolithic behavior and is a way to scale up reinforcement learning to complex tasks and to multiple tasks. In the subsumption architecture, the task decomposition into behaviors is given apriori. However, work has been done on learning the parameters controlling the applicability and prioritization parameters as an immediate reinforcement function[51, 49], as a response threshold determined by features in the environment[4, 65] or internal motivational forces[60].

In summary, a behavioral view ties local actions into a coherent whole and thus imposes a coarse granularity to the search space. Each behavior is considered a black-box module which can be learned separately. Similarly, training SLVQ on specific tasks produces a collection of behaviors. It is in leveraging from this modularity that lies the promise of scaling up reinforcement learning algorithms.

8.2 The Task Decomposition Problem

In decomposing a task the Markovian property assumed in reinforcement learning is lost since only a partial state is known to each memoryless agent accomplishing a task, namely the immediate sensory state. *Perceptual aliasing* is a related problem where a state cannot be discriminated adequately from other states based on observable features. One solution for this kind of non-markovian problem has been to augment the observable state with a history of past states within a certain window size[101] or to learn the value of a joint action[10] by maintaining a belief state representation. In the decomposition problem the non-markovian characteristic can be compensated by learning how to compose the subtasks and their limited range of applicability[38]. It was also shown in [80] that learning the composition and decomposition of elementary subtasks are two facets of the same problem, i.e. learning the association of a behavior to a desired subtask. For sequential tasks, temporal sequence can serve as a guide to subtask decomposition[86].

Because Go is a combinatorial game, decomposition has a special place in this game (see 4.6). Combinatorial game theory[12] can “solve” games which can be decomposed into independent subgames². This situation exists only at the endgame where each move affects only one subgame. In this context, a board configuration can be evaluated as a sum of games. Generalizing this approach to phases before the endgame hasn’t been successful so far. Identifying secure territories is an important step in identifying independent subgames[57]. Finding relevant subpatterns, not necessarily disjoint, is another approach to decomposition in Go for pattern-directed inference methods such as learning by parts.

²Subgames in game theory are subtrees of a game tree.

8.3 The Knowledge Transfer Problem

Task composition implies the possibility of a knowledge transfer between the subtasks and the more complex task. The knowledge transfer problem is the problem of transforming or adapting the knowledge acquired in learning a task to another task. For example, learning from a simulator always requires some adaptation to the real task. The amount of adaptation required to learn a new task given knowledge of another task versus no a priori knowledge is the knowledge transfer. Knowledge transfer also occurs when learning across multiple tasks. This has been shown in [18] to build up generalization by learning in an unsupervised manner a labeling symbol across multiple sensory modalities. After associating a random label to a pattern in one sensory modality, the simultaneous presentation of different sensory modalities reinforce each other until coordination in the labeling occurs. In summary, knowledge transfer can be thought of as another task[16, 80] or coordination behavior[24] that need also to be learned. What is learned in the transfer of control knowledge from subtasks is an arbitration function replacing the greedy function of the optimal policy in the global task.

Learning an arbitration function requires a goal-oriented representation of the top-level task. When a change of representation is possible as in a hierarchical representation, then compositional learning is possible. For example, a representation of a navigational task in terms of landmarks[38] composes the local navigational actions for each landmark in reaching the goal. However, when this change of representation is not given a priori or is hard to obtain, self-organization based on local interaction rules is a bottom up approach to complex tasks. In this context, arbitration functions based on heuristic search defines those local interaction rules. In a Kohonen network those local interaction rules give rise to a lateral network that reduces the high-dimensionality of the input into a 2-dimensional

representation suitable for visualization.

One such arbitration function for reinforcement learning tasks, for example, tries to maximize collective happiness[37, 100] by selecting the action with the highest expected value across subtasks, assuming the subtasks share some actions together:

$$f(s, g) = \arg \max_a \sum_i Q_{g_i}(s_i, a)$$

where s is the current state, g the set of subtasks, and $a \in A$, the set of actions. The actions selected with such an arbitration function will tend to satisfy multiple goals.

8.4 Methodology

Learning by parts is a shaping strategy. Borrowed from behavioral psychology, shaping rewards successive approximations to the desired behavior by increasing the difficulty of the intermediate target behaviors. Similarly, learning by parts assumes that learning a complex task will emerge from learning and coordinating the prerequisite simpler subtasks. Learning by parts combines learning and search. It is a basic reinforcement learning strategy that seeks positive reinforcement as a way to shape behaviors towards a higher-level target behavior. Its paradigm, based on Thorndike's identical elements theory of transfer[94], is outlined in Table 8.1.

Thorndike first identified learning as a behavior associating stimulus and response according to the strength of the feedback obtained. In this context, transfer of learning, i.e. the rapid identification of an appropriate response given a stimulus, is possible only when a similarity between stimuli is found. The similarity need not be identical but elements in

1. A decomposition of prerequisite subtasks is available
2. Learning the prerequisite subtasks occurs independently or concurrently
3. The stimulus/responses in the subtasks can be applied to the task
4. An arbitration function composes the subtasks

Figure 8.1: Learning by parts principles

common must be found. Learning by parts addresses the knowledge transfer problem as an isomorphism between stimuli and the focusing mechanism of an arbitration function. The subtasks need not be mutually exclusive and independent. Learning by parts differs from hierarchical learning like Max-Q[22] because of the absence of a top-down high-level representation. It differs from layered learning[86] because the stimulus/responses are not abstracted at the next level but is similar to it because the decomposition of behaviors can be learned in a separate task.

8.5 Motivation for a Game-Theoretical Approach to an Arbitration Function

From a game-theoretical viewpoint, the game of Go can be viewed as an intra, non-zero-sum, cooperative game between stones of the *same* color and as an extra, zero-sum, competitive game for stones of *different* color. Game theory is concerned with competitive and non-strictly-competitive (cooperative and non-cooperative) games of imperfect information (Figure 8.2). Typically, players move simultaneously with no knowledge of the other players' strategies. Likewise, by decomposing the board, the perfect information and cooperative aspect of the game is lost. It is therefore worthwhile looking again at the

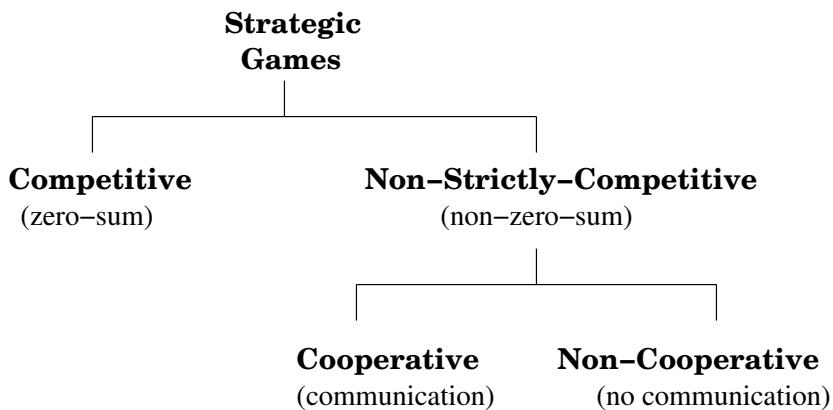


Figure 8.2: Game theory framework for strategic games

game theoretical approach for evaluating the joint actions of multiple agents because of the equilibrium property of such an approach that leads to convergence.

8.5.1 Nash Equilibrium and Mixed Strategy

A Nash equilibrium exists when each move is a *best* response given the other moves with respect to a global outcome. With perfect information, such equilibria can be achieved with a *pure* strategy. A pure strategy is a deterministic policy in RL terms. With imperfect information, playing a “best” move according to a local utility function does not always produce a favorable outcome. The Battle of the Sexes example (Table 8.1) illustrates this problem. In this game, given the information of the other player’s stated choice of music, it is better to compromise than not going to a concert at all. In the absence of information about the other player’s choice, a *mixed* strategy tries to obtain an equal payoff regardless of the other player’s choice by randomizing between pure strategies. A mixed strategy is expressed as a probability vector (one for each possible action) and tries to maximize a player’s security level by hedging the bets, so to speak. It is a maximin strategy[47]

Table 8.1: The Battle of the Sexes example
with different choices of music
 $(\frac{2}{3}, \frac{1}{3})$ and $(\frac{1}{3}, \frac{2}{3})$ are mixed strategy equilibria
for the row player and column player respectively.

	<i>Bach</i>	<i>Stravinsky</i>
<i>Bach</i>	(2,1)	(0,0)
<i>Stravinsky</i>	(0,0)	(1,2)

that ensures that a player gets at least a certain payoff under all circumstances. While pure strategies do not necessarily have a Nash equilibrium, it has been shown that every strategic game³ has a mixed strategy Nash equilibrium. It might seem strange to use a mixed strategy for games where a deterministic winning strategy should exist but, as was noted in [47, 45], a mixed strategy keeps the opponent from learning and exploiting a pure strategy's weaknesses.

It has been shown, in the Prisoner's dilemma for example, that a Nash equilibrium does not always produce pareto-optimal payoffs (that is the maximum payoff for the players as a whole) with instantaneous rewards for non-zero-sum games. It is rather the development of a strategy based on path dependence, such as tit-for-tat, that produces cooperation. The learning by parts methodology claims that with multi-agent reinforcement learners, this pitfall is avoided since an equilibrium is sought for *expected* future rewards and not instantaneous rewards.

³A strategic game is formally defined[59] as consisting of a set of players, a set of actions for each player and a preference relation on the actions.

8.5.2 Stochastic Arbitration Function

A game-theoretical approach suggests a stochastic arbitration function for composing SLVQ reinforcement learners[82] instead of a greedy strategy:

$$E(a_{s_{g_i}}) = \text{similarity}(s_g, m_{g_i})Q_{g_i}(s_{g_i}, a) \quad (8.1)$$

$$P(a|s_g) = \frac{e^{\frac{E(a_{s_{g_i}})}{T}}}{\sum_i e^{\frac{E(a_{s_{g_i}})}{T}}} \quad (8.2)$$

$$f(s, g) = P(a|s_g) \quad (8.3)$$

Learning by parts builds on the pattern recognition achieved in the subtasks. The candidate actions $a \in A$ are those with maximum similarity (s_g, m_{g_i}) in a state decomposition g_i . The expected value $E(a_{s_{g_i}})$ is the product of the similarity (s_g, m_{g_i}) and expected discounted reward $Q_{g_i}(s_{g_i}, a)$. The probability of action a is taken from the Boltzman distribution over its expected value within a state decomposition and annealed according of the temperature T . The adaptation process stops at a temperature T that will give good results given the opponent's responses. Previous game-theoretical approaches in reinforcement learning[36, 45] have used linear programming and quadratic optimization to compute a stochastic policy for matrix games⁴ taking into account the opponent's actions as well. In their approach, Markov games⁵ are treated as unrelated sequences of matrix games. This approach, an iterative version for large state space, learns the probabilities of a mixed strategy by adapting the Q-values to a global reinforcement signal to arrive at a stochastic policy to maximize rewards.

⁴Matrix games are strategic games with no history and no current state defined by a matrix of instantaneous rewards. The game of "rock, paper, scissors" is one example of zero-sum matrix games.

⁵Markov games are strategic games with no history like matrix games but with a current state.

Algorithm 8.1 Stochastic SLVQ

input the current state s , the set of legal moves A , a decomposition function g , a temperature T

output $a \in A$

procedure PICKMOVE (s, A, g, T)

1. $s_g \leftarrow$ a state decomposition
 2. $m \leftarrow$ all legal moves tuples $\{m, a, Q, \alpha\}$ maximally matching s_g
 3. for each m
 compute expected value $E(a_{s_i})$
 4. select a_{m_i} according to the Boltzman distribution
-

8.6 Empirical Evaluation

Learning by parts applies the results of learning on multiple subgames to 7×7 games.

8.6.1 Adaptive Wally

Wally's strategy is to search the board for common patterns around a candidate move in a table look-up and select the move with the highest *urgency*. Wally has approximately 50 hand-coded patterns and urgencies. Patterns vary in size and shape. Matching occurs on salient stones (black, white or empty) relative to the location of the candidate move while leaving other stones unspecified. This rule-type matching has been adopted by other programs. Depending on the urgency of the patterns found according to some pre-determined threshold, attack and defense tactics based on liberties count are used. While SLVQ does not have precise tactics, it can learn common patterns and their associated move response. In learning by parts, SLVQ learns the relative urgency of the patterns found to produce an adaptive Wally.

8.6.2 Approach

In applying the learning by parts principles, the first learning task is to learn subgames on multiple patterns and games. A second learning task is to learn the adaptation of the Q-values in those subgames to 7x7 games and a third learning task is to learn the right probabilities for a mixed strategy by adjusting the temperature in equation 8.2. In the adaptation phase, the board is decomposed into overlapping subgames (offset=1), as shown in Table 8.2, and candidate moves can span multiple subtasks. For a board of length n , subgames of length m , and offset o , there are $(\frac{n-m}{o} + 1)^2$ overlapping patterns and the offset has to be less than or equal $n - m$ to cover all possible legal moves. This exhaustive partition of the search space roughly corresponds to the corner, side and center topological representation of common Go patterns. Although no representation of the whole board exists, a mass of the opponent's stones will cast a shadow on the virtual boards through the influence values as an indirect communication. The following knowledge transfer methods were tested for the adaptation phase of the Q-values.

1. *Keep* the Q-values intact (no adaptation)
2. *Reset* the Q-values and learn the Q-values in a partially observable state.
3. *Adapt* the Q-values learned in the subgames to a partially observable state.

8.6.3 Experimental Results

5x5, 4x4 and 3x3 subgames trained against Wally were applied to 7x7 games against Wally. Table 8.3 summarizes the results. The results show the percentage of wins in 100 7x7 games

Table 8.2: Overlapping decomposition
shown with offset=2

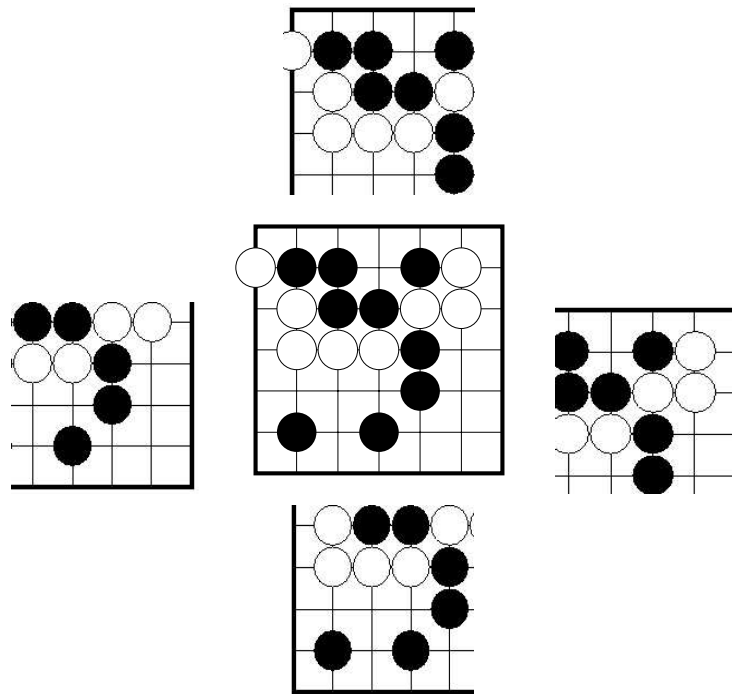


Table 8.3: Comparative performance of knowledge transfer methods

Knowledge Transfer method	Temperature	%wins vs Wally (7x7)
<i>keep</i>	n/a	0
<i>reset</i>	0.01	70%
<i>adapt</i>	0.25	100%

for the different knowledge transfer methods at the temperature that gave the best results. The temperature was set to 7.0 and slowly decreasing for a total of 1000 games.

As expected, merely decomposing the game does not produce good results. Decomposing the game and adapting the Q-values gives encouraging results for an adaptive Wally approach and has a better performance than monolithic SLVQS with 80% wins.

8.7 Discussion

The SLVQ approach has been shown to extend to multi-agent RL with partial knowledge of the state of the world. The Q-values adapt to the global task while using the pattern recognition of the subtasks. Final probabilities for a mixed strategy are arrived at by annealing the stochastic selection of candidate moves.

Chapter 9

Conclusions

The combination of LVQ and reinforcement learning is a rich learning framework for coordination strategies. Each codebook vector adapts to its task and to its neighbors through self-organization in a harmonious whole. The main contribution of this thesis was to develop SLVQ as a hybrid learning system and to introduce a family of algorithms extending its capabilities. Figure 9.1 shows the taxonomy of the algorithms. A chapter-by-chapter summary of the main contributions and open avenues for future research follows.

In Chapter 1 the action selection problem was framed in the broader problem of coordination strategy. The challenge was whether a novel learning algorithm, SLVQ (Chapter 3), based only on a distributed representation of the state space, self-organizing principles and reinforcement feedback could learn higher-level cognitive tasks such as game playing. The game of Go was chosen as a testbed, not only because a successful machine learning approach has not been found yet, but also for its specific coordination features between moves. Results in Chapter 5 indicate that this is a new and promising approach for this game and can be used to complement search-based programs such as Many Faces of Go (see Appendix B). Developing a set of default parameters for the algorithm should be the subject of further research especially the number of codebook vectors necessary for generalization. Extending the eligibility trace as a SOFM temporal neighborhood parameter

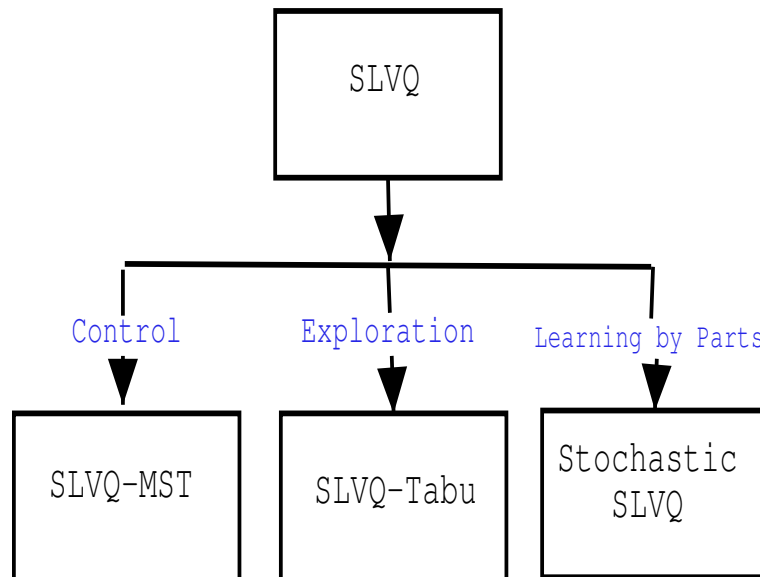


Figure 9.1: Family of Algorithms

decaying with time is also another promising avenue of research.

Chapter 6 demonstrated the applicability of the algorithm to control tasks problems such as the cart centering problem and the mountain car problem. Control tasks are time-consuming to encode and are well-suited to a learning approach but learning is often done in simulation because real experience is expensive. Bridging the gap is still a research area. An on-policy algorithm such as SLVQ improves upon an adaptive existing working policy without transitioning delays.

In Chapter 7, the issue of exploration for on-policy algorithms was examined and novel approaches based on tabu search were presented. Concept-based tabu search is a promising area to integrate domain knowledge and concept learning in reinforcement learning algorithms. Further research should be done on relating the entropy of a codebook vector as a metric for diversity with the exploration process and the learning cost as well as a principled determination of the tabu tenure parameter. A proof of the admissibility of tabu search

as a metaheuristic for on-policy reinforcement learning would be useful.

Finally, Chapter 8 expanded SLVQ to stochastic policies for partially observable states under the learning by parts paradigm. Decomposition coupled with a mixed-strategy policy proved to be a robust way to scale up SLVQ. More work needs to be done to learn and apply different subgames to obtain an intelligent and adaptive Wally.

Appendix A

Overview of the Game of Go

Go is a deterministic perfect-information 2-player game usually played on a 19x19 board. It can also be played on a 9x9 board or on a 13x13 board. The board is empty at the beginning of the game and the players alternate placing stones on the board. Stones are placed on the intersection of the lines. Once moved, a stone never moves unless captured. A stone alone on the board has 4 “liberties” (adjacent points) (Figure A.1a). Connecting stones in a group increase the number of liberties (Figure A.1b). Groups are the basic building blocks of a board configuration. A stone is captured when it has no liberties left (Figure A.1c). The aim of the game is to enclose territory (vacant points) (Figure A.1). The only way to achieve this goal is to make your stones “alive”. Here, alive is taken in the weak sense meaning that your stones can’t be captured. It has been found that the 2-eye shape is sufficient for life (Figure A.2). A good starting point for more information is www.usgo.org. In addition to the battle/war metaphors, economic metaphors can help understand some of the underlying strategies in the game of Go. For example, an investment is made when playing a stone and the object of the game can be thought of as making a large return on a small investment.

Table A.1: Liberties

<p>a. 1 stone, 4 liberties</p>	<p>b. 2 stones, 6 liberties</p>	<p>c. the white stone is captured</p>

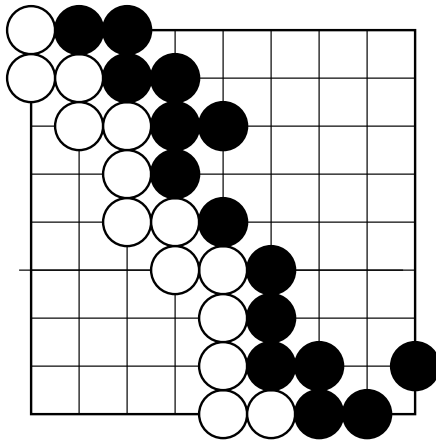


Figure A.1: Territory

Black has 32 points of territory and White 20 points.

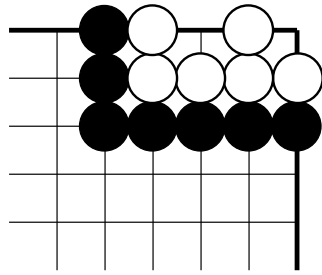


Figure A.2: Life

The white stones are alive in a 2-eye shape even though surrounded by the black stones.

Basic Rules of Go

1. A player may pass his turn
2. Suicide is not allowed.
3. A player cannot immediately recapture in ko.
4. The game ends when both players pass.

The third rule arises out of a situation, called ko (Figure A.3), that could cause an impasse in the game.

Go is hard because the simplicity of the rules allow complex interactions of the stones on the board. Stones can be connected in subtle ways as long as disconnection can be rebuked. There are definitively three stages of the game: the opening where stones claim territory, the middle game where claims are being fought for, and the endgame where the size of the territory conquered is established.

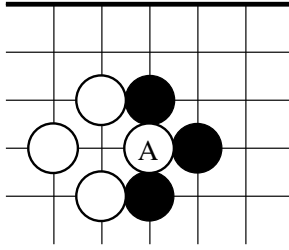


Figure A.3: Ko

A could be captured but White can't recapture immediately.

The Coordination Aspects of the Game

Unlike Chess, stones are added to the board and do not move afterwards unless captured. The concept of a move in Go is therefore different from other games and more dependent on other stones on the board. The impact of a move in chess can be evaluated for itself. A stone added to the board in Go acquire its meaning from its relationships to other stones.

Stones that work together are said to have “good shape”. Just like in the game of life,¹ certain common patterns (Table A.2 and A.3) arise when playing Go that describe the way the stones move across the board. Joseki and fuseki patterns arise at the beginning of the game.

¹The game of life is a cellular automata game where the status of a cell at time t is derived from the status of its neighbors at time $t + 1$ according to some fixed rules. From those local interactions, patterns emerge. See [68] for a cellular automata approach to the game of Go.

Table A.2: Interaction between friendly stones

bamboo joint	knight's move	tiger mouth

Table A.3: Interaction between enemy stones

joseki	ladder	fuseki

Some Heuristic Concepts

Connection

Even if there is not a solid connection, players can infer that a connection can be made if necessary. There exist typical connection patterns (Table A.2). As a rule of thumb, any stone 1 or 2 points away is connectable. When to connect (solidly) and when not to connect? Making each move count is important so that the opponent does not take the advantage. Connected stones of the same color form a “group” or “string”.

Influence

This concept measures the estimated value of a move in terms of territorial claim. This concept is computationally tractable. Each stone “radiates” a numerical value to contiguous nonoccupied points as a decreasing function of the distance[103]. Typically, white stones radiate positive values and black stones negative values. The influence value computed from each stone is added to the value of each point (Figure A.4). Summing the value of each point on the board can give a good estimate of the value of a move in terms of territory and can serve as a static evaluation of the board. The question is when to play such moves? Professional players seem to alternate between strategic moves that “claim” territory and have a high influence evaluation and tactical moves which don’t.

Life and Death and Neither (Seki)

In the weak sense, a group of stones is alive if it can’t be killed. A 2-eye shape is sufficient but not necessary to make a group alive. There are safe and unsafe patterns without much

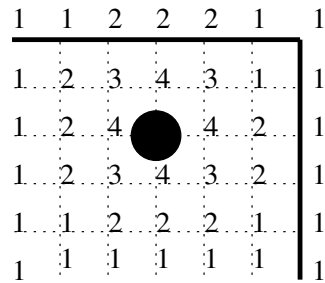


Figure A.4: Influence radiating from a stone

uncertainty about it. What's uncertain is the pattern of the opponent's stones, i.e., the environment, and the right sequence of stones to achieve a 2-eye shape. The canonical patterns are rarely achieved in practice. Life and death issues are tactical issues and can be studied on a small 9x9 board. One tactic in the game of life and death is the throw-in: a stone can be sacrificed to kill a group. An evaluation of the board has to be projected several moves ahead to model this situation. Life and death issues arise in the endgame. This has been found to be the most difficult part of the game to learn inductively because of the precise nature of the play. Connecting groups of stones makes it easier to live since fewer eye shapes will be necessary.

A third option to life and death, *seki*, figure A.5, is possible in the game. There are situations where it is in the interest of both players to stop fighting and live together.

Sente

Sente is maybe the most important characteristic of a tactical move. It means that the opponent cannot ignore it and must respond to it under the threat of losing an important chunk of territory. Keeping the initiative is a sure way of winning the game. The importance

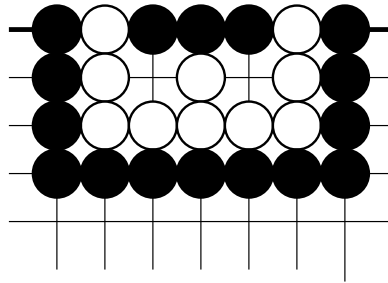


Figure A.5: Seki
The black and white stones live together

of a sente move is in destroying the ability of the opponent to strategize. The value of sente can be derived as the evaluation differential of a board from the point of view of the two players. Deciding whether to reply to a *sente* move and lose the initiative is a dilemma.

Multi-Purpose Moves

A rule of thumb is to select a move that has more than one purpose to be considered effective suggesting corroborative evidence from different perspectives. The main purposes of a move can be operationally defined in term of liberties² and influence as follows:

1. Attack: reduce opponent's liberties
2. Defend: increase own's liberties
3. Claim: increase own's influence
4. Invade: decrease opponent's influence
5. Conquer: enclose liberties.

²Here, liberties are loosely counted as the 8-neighbor intersections of a stone

Appendix B

Overview of Computer Go

Computer Go has been approached from two different perspectives: the knowledge intensive approach using domain heuristics and search or the machine learning approach with no pre-programmed knowledge. Below is a summary of some of the programs using AI techniques.

Many Faces of Go (MFOG)[28] This highly popular program is based on traditional AI techniques, such as alpha-beta search, rule-based expert systems and pattern matching into a pattern database of 8x8 points and an opening move (Joseki) database. Lookahead search goes up to 7 ply. It uses domain knowledge of various possible shapes to evaluate connectivity, eye space and territory. It has different move generators for the different phases of the game and meta-rules for strategic prioritization of the rules. The sente value of a move decreases as the game progress. The rule-based system for move suggestions has text associated with each rule to explain the move. Because of that, MFOG doubles up as a great teaching program. Associated with each pattern is a move (game) tree to suggest move sequences. This program can be easily extended by storing more rules or patterns. It has now approximately 1800 patterns.

Gobble [8] This program evaluates a move based on the outcome of Monte Carlo simulations with random games. It was the first program to play Go without prior knowledge. Simulated annealing is used for the Monte Carlo simulation of a random variable. Each move is assigned the average value of all the games in which it was played. After every update, the moves are ordered in descending order according to their average values. The modification for simulated annealing is to introduce a finite probability for a move to be played out of order which depends on the exponential of the value difference divided by the temperature. The ordered list of moves is swept once from best to worst move and two neighboring moves are switched with probability p_{swap} . The probability $p(n)$ that a move is shifted $n \geq 1$ steps down the list is

$$p(n) = (p_{swap})^n = \exp\left(\frac{-n}{T}\right)$$

The game is then played according to this permuted list of possible moves. The temperature decreases slowly at the end of each game.

Golem[26] This program also uses a knowledge-based approach to Go but claims not to use patterns. Instead it has general high-level rules that are applied to strings of stones that need defending or capturing and an evaluation function to estimate the gain in territory in a one-ply search. In turn, the candidate moves and the low-level features of their immediate context in canonical form are fed into a neural net for evaluation. The neural net has been previously trained with moves taken from professional games vs. random moves from the same board positions. It might be said that this use of a neural net is equivalent to the use of patterns to recognize good moves vs. bad moves. This program is estimated to play at the intermediate level. One insight of this program is to couple a global evaluation, the

estimated territory, with a local evaluation of the goodness of a move.

Classifier Systems An interesting representation of Go as a cellular automata problem gives rise to learning a strategy using a classifier approach[69]. The immediate neighborhood of a candidate move is encoded as the condition of a rule and whether to move or not as the action. The rule with the highest bid (strength) fires and gives part of its strength to the rule(s) that activated it in a producer/consumer fashion. At the end of the game the reward is given to the rule that fired last and will be added to its strength. As games are played the rewards are propagated back to the intermediate rules using the bucket brigade algorithm, thus solving the credit assignment problem. A new generation of condition/action pairs is then computed until the emergence of relevant rules. It has been shown that simple classifiers are equivalent to Q-learning[23].

SANE[67] This approach uses genetic algorithms to evolve neural networks. SANE evolves partial solutions, i.e. neurons, represented as a collection of labelled weight links to input and output units. Evolving the label as well as the weight enables the genetic algorithm to evolve the structure of the network as well as the parameters. Those neurons, in turn, are grouped at random to form a network. One neuron can participate in more than one network. Hierarchical SANE evolves those groups of neurons as well as the neurons themselves. The value of a node is computed as the sum of its input values multiplied by their connecting weights and passed through the sigmoid activation function. The output units are linear.

Each line intersection on a Go board is represented by 2 binary input units, 10, 01, and 00, representing Black, White, and Empty. The architecture is a three-layer feedforward network with a random number of hidden nodes. The fitness function is the score in terms of

territory points difference at the end of the game. To get a better resolution, the score was averaged over several games. This program beats Wally on a 9x9 board after 260 generations. Scaling up to a 13x13 board was estimated to require several thousand generations and is one of the main problem of this approach. It was found that the program learned to exploit weaknesses in the opponent (and thus did better against a deterministic opponent) instead of learning how to play Go. Nevertheless, this program showed examples of good play like taking corner territory first.

Temporal Difference Learning In an adaptation of TD-Gammon to the game of Go[75], an entire 9x9 “raw” board is mapped into a 82x40x1 backpropagation network. The output predicts the probability of winning for black starting from a given state. There is no immediate reward until the end of the game. For intermediate steps, the error propagated back using gradient descent is $Y(t+1) - Y(t)$ where Y is the probability of winning of the board[92], and $Y(t)$ and $Y(t+1)$ are two temporally successive predictions. The exploration policy is done by Gibbs sampling. An exploration scheme wasn’t needed for TD-Gammon since Backgammon is a nondeterministic game. The probability of a move conditioned on the rest of the board is obtained with a short 1-ply search. Gibbs sampling ensures that the distribution of the examples reflects the distribution of the value function as predicted. Annealing is done to reduce exploration as learning progresses. This program is reported to reliably beat Wally, a low-level player, after training against Many Faces of Go, a better player. One insight of this program has been that self-play was not conducive to learning in the game of Go. It would be interesting to know how Backgammon differs from Go since the TD(0) method has vastly different results between the two games. This result points to the complexity of Go. One issue is whether the probability of winning is sufficient to produce good moves[91]. Further work is to be done on a hybrid approach

integrating a connectivity map.

The success of TD-Gammon has been attributed[62] to the coevolutionary nature of self-play and the dynamics of Backgammon rather than to the temporal difference learning method. Self-play learning is prone to getting stuck in local maxima if the game allows it. In other words, a player could easily learn to “draw” itself and stop learning thereby achieving equilibrium. Apparently, the nature of Backgammon prevents those loops of self-complacency. That would explain why the TD-Gammon approach fails in the game of Go. The behavior of the program depends on the landscape of the domain. The question remains as to whether training can influence the landscape of the domain and overcome those difficulties.

Honte, a MultiStrategy Approach In Honte [13], neural nets and other traditional AI techniques such as alpha-beta search are used to evaluate features of a global (whole board) evaluation function. Such features include the number of strings that can be captured, the groups’ safety, territory estimate, etc. Candidate moves are suggested from a neural net trained to recognize expert moves from 7×7 patterns. Thereafter, a more detailed evaluation of the resulting board concerning the status of the groups is obtained with a lookahead search. When groups are not well defined because potential connections are still in question, a neural net trained with TD-learning evaluates their safety value. The groups are represented in terms of operational features such as the number of liberties, the number of eyes and the influence value of the liberties. Groups are considered safe if the stones remains on the board at the end of the game. Honte combines learning and search as follows. The evaluation of abstract strategic features such as safety and potential territory (moyo) is learned but a lookahead search evaluates the effect of tactical moves. Those evaluations are combined into a global evaluation of the board. One noted weakness is that this program

does not evaluate the *sente* property of a move which is an important characteristic of a feature-driven evaluation. It also does not learn when to play a strategic move versus a tactical move.

Bibliography

Bibliography

- [1] J.S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10:15–61, 1971.
- [2] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Maastricht University, the Netherlands, 1994.
- [3] R. C. Arkin and T. Balch. Aura: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence*, 1997.
- [4] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence, From Natural to Artificial Systems*. Oxford University Press, 1999.
- [5] B. Bouzy and T. Cazenave. Computer go: an ai oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.
- [6] R. A. Brooks. Technical Report 864, MIT AI Lab Memo, September 1985.
- [7] R. A. Brooks. Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.
- [8] B. Brugmann. Monte carlo go. available at <ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z>, 1993.
- [9] E. Cervera and A. P. del Pobil. A som-based sensing approach to robotic manipulation tasks. In Oja E. and Kaski S., editors, *Kohonen Maps*. Elsevier, 1999.

- [10] C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *National Conference on Artificial Intelligence*, 1998.
- [11] C. Clausen, S. Gutta, and H. Wechsler. Reinforcement algorithms using functional approximation. In *16th International Joint Conference on Artificial Intelligence*, 1999.
- [12] J. Conway. *On Numbers and Games*. Academic Press, London/New York, 1976.
- [13] F. A. Dahl. Honte, a go-playing program using neural nets. In *Workshop on Machine learning in Game Playing*, 1999.
- [14] A. R. Damasio. *Descartes' Error*. Avon Books, New York, 1995.
- [15] P. Dayan. The convergence of td(λ) for general λ . *Machine Learning*, (8):341–362, 1992.
- [16] P. Dayan and G. Hinton. Feudal reinforcement learning. In *Neural Information Processing Systems*, 1995.
- [17] P. Dayan and T. J. Sejnowski. Td(λ) converges with probability 1. *Machine Learning*, (14):295–301, 1994.
- [18] V. de Sa. *Prerational Intelligence: Adaptive Behavior and Intelligent Systems without Symbols and Logic, Vol. 2*, chapter Combining Uni-Modal Classifiers to Improve Learning, pages 702–722. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.
- [19] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufman, San Mateo, CA, 1991.

- [20] G. DeJong. Hidden strengths and limitations: an empirical investigation of reinforcement learning. In *International Conference on Machine Learning*. Morgan Kaufmann, 2000.
- [21] D. DeSieno. Adding a conscience to competitive learning. In *IEEE International Conference on Neural Networks*, volume 1, pages 117–124, 1988.
- [22] T. Dietterich. The maxq method for hierarchical reinforcement learning. In *International Conference on Machine Learning*, 1998.
- [23] M. Dorigo and H. Bersini. A comparison of q-learning and classifiers systems. In *From Animals to Animats: Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*, 1994.
- [24] M. Dorigo and M. Colombetti. *Robot Shaping, an Experiment in Behavior Engineering*. MIT Press, 1997.
- [25] E. H. Durfee and J. Rosenschein. Distributed problem solving and multiagent systems: Comparisons and examples. In M. Klein, editor, *Proceedings of the 13th International Workshop on DAI*, pages 94–104, Lake Quinalt, WA, 1994.
- [26] H. D. Enderton. The golem go program. Technical report, CMU-CS-92-101, 1992.
- [27] S. Epstein. Toward an ideal trainer. *Machine Learning*, (15):251–277, 1994.
- [28] D. Fotland. Knowledge representation in the many faces of go. available at <http://www.smart-games.com/knowpap.txt>, 1993.
- [29] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

- [30] J. J. Grefenstette. The evolution of strategies for multiagent environments. *Adaptive Behavior*, 1(1):65–90, 1992.
- [31] R. K. Guy. *Games of No Chance*, chapter What Is a Game? Cambridge University Press, 1998.
- [32] M. Harmon and L. Baird. Residual advantage learning applied to a differential game. In *Neural Information Processing Systems*, number 7, 1995.
- [33] T. Haynes et al. Evolving multiagent coordination strategies with genetic programming. Technical report, University of Tulsa, 1995.
- [34] J. H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach*, volume 2, chapter 20. Morgan Kaufmann, 1986.
- [35] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 2nd edition, 1992.
- [36] J. Hu and M. P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *International Conference on Machine Learning*, pages 242–250. Morgan Kaufman, 1998.
- [37] M. Humphrys. *Action Selection Methods using Reinforcement Learning*. PhD thesis, Trinity Hall, 1997.
- [38] L. Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *International Conference on Machine Learning*, 1993.

- [39] S. Koenig and R. G. Simmons. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithm. *Machine Learning*, 22:227–250, 1996.
- [40] T. Kohonen. *Self-Organizing Maps*. Springer, 2nd edition, 1997.
- [41] R. Korf. Real-time heuristic search. *Artificial Intelligence*, pages 189–211, 1990.
- [42] R. Korf. A simple solution to pursuit games. In *Proceedings of the 11th International Workshop on Distributed Artificial Intelligence*, Glen Arbor, MI, 1992.
- [43] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
- [44] D. Lichtenstein and M. Sipser. Go is polynomial-space hard. *Journal of the Association for Computing Machinery*, 27(2):393–401, 1980.
- [45] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 157–163. Morgan Kaufman, 1994.
- [46] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, (28):127–135, 1982.
- [47] R. D. Luce and H. Raiffa. *Games and Decisions*. Dover Publications, Inc., New York, 1957.
- [48] P. Maes. Behavior-based artificial intelligence. In *Proceedings of the Fifteenth Conference on Cognitive Science*. Lawrence Erlbaum Associates, 1993.

- [49] P. Maes and R. Brooks. Learning to coordinate behaviors. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 796–802, Boston, MA. AAAI Press/ MIT Press.
- [50] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, (55):311–365, 1992.
- [51] M. J. Mataric. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, 1997.
- [52] J. K. Millen. Programming the game of go. in *Byte Magazine*, April 1981.
- [53] M. Mitchell, P. T. Hraber, and J. P. Crutchfield. Revisiting the edge of chaos: Evolving cellular automata to perform computations. Technical Report Santa Fe Institute Working Paper 93-03-014, 1993.
- [54] A. W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Machine Learning: Proceedings of the Eighth International Workshop*, San Mateo, CA, 1991. Morgan Kaufmann.
- [55] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.
- [56] D. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, (22):11–23, 1996.
- [57] M. Muller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgames. In *IJCAI*, volume 1, pages 578–583, 1999.

- [58] W. H. Newman. available at <ftp://www.joy.ne.jp/welcome/igs/Go/computer/shally.sh.Z>, 1988.
- [59] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.
- [60] L. E. Parker. Adaptive heterogeneous multi-robot teams. *Neurocomputing*, 28:75–92, 1999.
- [61] J. Pearl. *Heuristics*. Addison-Wesley Publishing Company, 1988.
- [62] J. Pollack et al. Coevolution of a backgammon player. In *Fifth Artificial Life Conference*, Nara, Japan, 1996.
- [63] G. Polya. *How to Solve It*. Princeton University Press, Princeton, New Jersey, 1973.
- [64] J. Randlov and P. Alstrom. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufman.
- [65] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1997.
- [66] C. W. Reynolds. Flocks, herds, and schools: a distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.
- [67] N. Richards, D. Moriarty, P. McQuesten, and R. Miikkulainen. Evolving neural networks to play go. In *Proceedings of the Eight International Conference on Genetic Algorithm*, 1997.
- [68] C. Rosin. *Coevolutionary Search among Adversaries*. PhD thesis, University of California, San Diego, 1997.

- [69] C. D. Rosin and R. K. Below. Methods for competitive coevolution: Finding opponents worth beating. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, 1995.
- [70] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University Engineering Dept., 1994.
- [71] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 1959.
- [72] S. Santini and R. Jain. Similarity measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9), 1999.
- [73] R. E. Schapire. A brief introduction to boosting. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.
- [74] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. *Computational Intelligence in Games*, chapter Learning to Evaluate Go Positions via Temporal Difference Methods. Springer-Verlag, 2001.
- [75] N. N. Schraudolph et al. Temporal difference learning of position evaluation in the game of go. In *Neural Information Processing Systems*, number 6. Morgan Kaufman, 1994.
- [76] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, MA.

- [77] S. Sen, M. Sekaran, and J. Hale. Learning to coordinate without sharing information. In *Proceedings of the National Conference on Artificial Intelligence*, pages 426–431, 1994.
- [78] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.
- [79] H. Shui. *MultiPurpose Adversary Planning in the Game of Go*. PhD thesis, George Mason University, 1995.
- [80] S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.
- [81] S. Singh, T. Jaakkola, M. L. Littman, and Csaba Szepesvari. Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, 38(3):287–308, 2000.
- [82] S. P. Singh, T. Jaakkola, and M. I. Jordan. Learning without state-estimation in partially-observable markovian decision processes. In *International Conference on Machine Learning*, 1994.
- [83] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.
- [84] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.
- [85] J. E. R. Staddon and R. H. Ettinger. *An Introduction to the Principles of Adaptive Behavior*. Harcourt Brace Jovanovich, 1989.

- [86] P. Stone. Layered learning in multiagent systems. In *AAAI/IAAI*, page 819, 1997.
- [87] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the International Conference on Machine Learning*, pages 216–224. Morgan Kaufman, 1990.
- [88] R. S. Sutton and A. Barto. *Reinforcement Learning: an Introduction*. MIT Press, Cambridge, MA, 1998.
- [89] R. S. Sutton and S. P. Singh. On step-size and bias in temporal-difference learning. In *Proceedings of the Eight Yale Workshop on Adaptive and Learning Systems*, pages 91–96, 1994.
- [90] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1998.
- [91] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.
- [92] G. Tesauro. Temporal difference learning and td-gammon. *Communication of the ACM*, 38(3), 1995.
- [93] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence Journal*, 134(1-2):181–199, January 2002.
- [94] E.L. Thorndike. *Principles of teaching*. New York: MacMillan, 1906.
- [95] S. B. Thrun. Efficient exploration in reinforcement learning. Technical Report TR CMU-CS-92-102, Carnegie Mellon University, 1992.

- [96] S. B. Thrun. The role of exploration in learning control. In David A. White and Donald A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, New York, NY, 1992.
- [97] A. Tvesky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.
- [98] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King’s College, 1989.
- [99] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, (8):279–292, 1992.
- [100] S. Whitehead, J. Karlsson, and J. Tenenbergh. *Robot Learning*, chapter Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging. Kluwer Academic Publishers, 1993.
- [101] S. D. Whitehead and L. Lin. Reinforcement learning of non-markov decision processes. *Artificial Intelligence*, 73(1-2):271–306, 1995.
- [102] S. Willmott, J. Richardson, A. Bundy, and J. Levine. An adversarial planning approach to go. In *Proceedings of the First International Conference on Computer and Games*. Springer-Verlag, 1998.
- [103] L. Zobrist. *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. PhD thesis, University of Wisconsin, 1970.

Curriculum Vitae

Myriam Abramson, born March 8, 1952, graduated from high school in Israel with a passion for philosophy and especially ethics. She went to the Sorbonne to satisfy this passion and obtained a bachelor's degree in philosophy. She left somewhat disillusioned of having learned anything more than rhetorics. Later, she went back to school to study computer science as something that she could do well while earning a living. She earned a bachelor's degree and a master's degree in computer science also at George Mason University. It wasn't until she took "Intro to AI" in graduate school that she was able to rekindle this fire about theories of intelligence, representation and action. In her dissertation, *Learning Coordination Strategies*, she addresses the problem of action selection in a computational framework. She has published the following articles:

Abramson M., Wechsler H., A Distributed Reinforcement Learning Approach to Pattern Inference in Go, International Conference on Machine Learning Applications, Los Angeles, CA (2003).

Abramson M., Pachowicz P., Wechsler H., Competitive Reinforcement Learning in Continuous Control Tasks, Proceedings of the International Joint Neural Network Conference, Portland, OR (2003).

Abramson M., Wechsler H., Tabu Search Exploration for On-policy Reinforcement Learning, Proceedings of the International Joint Neural Network Conference, Portland, OR (2003).

Abramson M., Wechsler H., Competitive Reinforcement Learning for Combinatorial Problems, Proceedings of the International Joint Neural Network Conference, Washington, DC (2001).

Abramson M., Hunter L., Classification using Cultural Co-evolution and Genetic Programming. Proceedings of the 1996 Genetic Programming Conference (1996).

Abramson M., Bennet S., et al. Predictive Analysis System: A Case Study of AI Techniques for Counternarcotics. Proceedings of the Tenth IEEE Conference in Artificial Intelligence for Applications (1994).