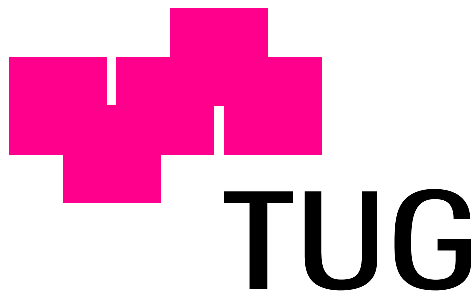
Machine Learning Applications in Computer Games

Diplomarbeit am
Institut für Grundlagen der Informationsverarbeitung
Technisch-Naturwissenschaftliche Fakultät
der



Technischen Universität
Erzherzog-Johann-Universität
(University of Technology)
Graz

vorgelegt von

Michael Pfeiffer

Studienrichtung: Technische Mathematik / Informationsverarbeitung

Betreuer und Begutachter: *O. Univ.-Prof. Dr.rer.nat. DI Wolfgang Maass*

Graz, Mai 2003

Contents

Abstract **vii**

1 Machine Learning in Computer Games **1**
1.1 Artificial Intelligence in Games 1
1.1.1 Artificial Intelligence Methods 2
1.1.2 Machine Learning Methods 6
1.2 Research vs. Industry: The Dilemma 8
1.2.1 Conclusion 10
1.3 Machine Learning Applications in Classical Games 10
1.3.1 Checkers 10
1.3.2 Chess 11
1.3.3 Go 11
1.3.4 Backgammon 11
1.3.5 Poker 12
1.4 Machine Learning Applications in Commercial Games 13
1.4.1 Creatures 13
1.4.2 Black & White 14
1.4.3 Games using Neural Networks 14

2 Reinforcement Learning **17**
2.1 Reinforcement Learning Model 17
2.1.1 Value Functions 18
2.1.2 The Learning Task 19
2.2 Learning Algorithms 19
2.2.1 Policy Iteration 19
2.2.2 Action selection 20
2.2.3 Temporal Difference Learning 20
2.3 Generalization and Function Approximation 23
2.3.1 Function Approximation 23
2.3.2 Performance of the Approximator 24
2.3.3 Gradient Descent Methods 24
2.3.4 Function Approximation in Games 25
2.4 Hierarchical Reinforcement Learning 25
2.4.1 Module-Based Reinforcement Learning 26
2.4.2 Options 27
2.4.3 MAXQ 27
2.4.4 Feudal Q-Learning 27
2.4.5 Discovery of Subgoals 27
2.4.6 Other Approaches 28
2.5 Reinforcement Learning in Games 28

3 Regression and Model Trees 31
3.1 Decision Trees 31
3.2 Regression Trees 32
3.3 Model Trees 32
3.3.1 Multivariate Linear Regression 32
3.4 Quinlan's M5 Algorithm 34
3.5 Reinforcement Learning with Regression and Model Trees 36
3.5.1 Earlier Results 37
3.5.2 A Q-Learning Algorithm using Model Trees 39
3.5.3 Conclusion 40

4 Settlers of Catan 41
4.1 Rules of the game 41
4.2 Tactics of the Game 44
4.2.1 Starting Phase 44
4.2.2 Intermediate Phase 46
4.2.3 Endgame 48
4.3 Artificial Intelligence for Settlers of Catan 48

5 Learning to Play Settlers of Catan 51
5.1 Program Architecture 51
5.2 Modular Agent Architecture 51
5.2.1 Behaviours 51
5.2.2 Actions 52
5.2.3 Primitive Actions 53
5.2.4 Trading 53
5.2.5 Passive Actions 54
5.2.6 Initial Buildings 54
5.3 Reinforcement Learning Framework 55
5.3.1 State Space 55
5.3.2 Hierarchical Learning 55
5.3.3 References to Related Work 56
5.3.4 Function Approximation 57
5.4 Training the Model Trees 58
5.4.1 Construction of the Model Trees 59
5.5 Training Procedure 61
5.5.1 General Training Procedure 61
5.5.2 Alternative Architectures 61

6 Results 63
6.1 Evaluation of the Policies 63
6.2 Feudal Approach 64
6.3 Module-Based Approach 66
6.4 Heuristic High-Level Strategies 69
6.5 Heuristically Guided Learning 73
6.6 Strengths and Weaknesses of the Approaches 75
6.7 Possible Applications in Computer Games 78
6.8 Future Work 79

7 Conclusion 81

A Heuristics for Low-Level Actions 83
A.1 Low-Level Actions 83
A.2 Goal Selection 85

B Definition of Features **87**

C List of Abbreviations **89**

D Summary of Notation **91**

Abstract

In this thesis the application of machine learning methods in modern, complex computer games is studied. First we present interesting approaches from the scientific world and the game industry, which are later applied in order to learn strategies for the game *Settlers of Catan*. A combination of hierarchical reinforcement learning and simple heuristics is used for this task. Since existing algorithms for learning and function-approximation are not very well-suited for problems of this size and complexity, new modifications were developed, which make learning in complex game environments easier. As a result this paper presents the first use of model trees for value-function approximation in a sophisticated computer game. Furthermore it is demonstrated how a-priori knowledge about the game can simplify the learning task, thereby reducing the learning time and improving the performance of the learned strategies. The design of the artificial agents could be kept particularly simple by using machine learning for those components, for which it would otherwise be very difficult to develop good heuristic policies. The performance of several different learning approaches was compared, and it turned out that, despite the simplicity of the architecture, a combination of learning and built-in knowledge yielded strategies that were able to challenge and even beat human players in a complex game like this.

Keywords: Computer Games, Machine Learning, Reinforcement Learning, Settlers of Catan, Function Approximation, Model Trees.

Zusammenfassung

In dieser Arbeit wird der Einsatz von Methoden des Maschinellen Lernens in modernen, komplexen Computerspielen studiert. Nach einer Präsentation interessanter Ansätze aus dem wissenschaftlichen und kommerziellen Bereich wird gezeigt, wie diese Ergebnisse angewandt werden können, um Strategien für das Spiel *Die Siedler von Catan* zu erlernen. Dabei kommen Techniken des hierarchischen Reinforcement Learning, kombiniert mit einfachen heuristischen Ansätzen zum Einsatz. Da bestehende Lern- und Approximations-Algorithmen für Probleme dieser Größe und Komplexität nicht gut geeignet sind, wurden neuartige Modifikationen entwickelt, durch die Lernen in komplexen Spielumgebungen erleichtert wird. In dieser Arbeit werden dabei zum ersten Mal Model Trees für die Approximation der Value-Function in einem anspruchsvollen Computerspiel verwendet. Weiterhin wird demonstriert, wie die Lernaufgabe durch Vorwissen über das Spiel vereinfacht werden kann, um damit sowohl die Lerndauer zu verkürzen, als auch die Leistung der gelernten Strategien zu verbessern. Das Design der künstlichen Agenten für das Spiel konnte dabei absichtlich einfach gehalten werden, da Maschinelles Lernen für Komponenten anwandt wurde, für die es sonst sehr schwierig gewesen wäre, geeignete Verfahren zu entwickeln. In einem Vergleich verschiedener Lernansätze zeigt sich, dass eine Kombination aus Lernen und eingebautem Wissen, trotz der Einfachheit der Architektur, Strategien liefert, die auch in einem komplexen Spiel wie diesem gute menschliche Spieler fordern und sogar schlagen können.

Stichwörter: Computerspiele, Maschinelles Lernen, Reinforcement Learning, Siedler von Catan, Funktions Approximation, Model Trees.

List of Figures

4.1 The board in Settlers of Catan 42

6.1 Performance of Feudal vs. Random Players 64
6.2 Performance of Feudal vs. Human Players 65
6.3 Performance of Module-Based vs. Feudal Players 67
6.4 Performance of Module-Based vs. Human Players 68
6.5 Performance of Heuristic Players vs. Heuristic Random Players . . . 70
6.6 Performance of Heuristic vs. Random, Feudal and Module-based Players 71
6.7 Performance of Heuristic vs. Human Players 72
6.8 Performance of Heuristically Guided Players 74
6.9 Performance of Heuristically Guided vs. Human Players 75
6.10 Performance of all Learned Policies vs. Human Players 77

Chapter 1

Machine Learning in Computer Games

Games have always been an intellectual challenge to humans, and so it is not surprising that computer games have become one of the favourite testbeds for **artificial intelligence** (*AI*) research. Computers that can play games at a level similar to or even higher than that of human experts, are for many people the ultimate proof that "real intelligence" can be recreated by computers.

In order to master a game, humans undoubtedly need *experience* from which to *learn*. Artificial intelligence in games however usually uses a different approach: instead of letting the abilities of the virtual player develop, programmers try to use their knowledge to solve every part of the game. Whether or not this is "artificial" intelligence remains a question that most people - including myself - would answer with "no". Another problem with this approach is that the developer has to foresee every possible situation in the game, which is often not feasible. Intelligent beings are able to develop their own solutions to problems, they can learn e.g. from mistakes, successes or observation of others. **Machine learning** (*ML*) is a branch of AI that is concerned with using biological and mathematical learning concepts on computers. A small number of very successful applications of ML in computer games have shown that it is possible to let virtual game players increase their abilities by learning from experience.

Besides their importance in AI research, computer games have also become a major industry. This however opened up a big gap between the methods studied in the academic world, and those used in commercially successful games. Machine learning e.g. is a field that is almost completely ignored in modern games, even though many people recognize its potential.
Few people from the universities or the software industry have tried to close this gap (see [Lai03] for an exception). This survey should therefore be seen as another attempt to bring both sides closer together. Interesting ideas and projects from the academic and the commercial point of view are presented, and suggestions are made how new techniques can be combined with proven methods to contribute to the success of future games.

1.1 Artificial Intelligence in Games

Almost all computer games use some sort of artificial intelligence. Three rough categories of game AI can be identified:

1. **Opponent AI:** The game controls one or more opposing characters. This involves making high-level strategic decisions and low-level tactical decisions. The ultimate goal is to create the illusion of playing against another human.
2. **Partners:** Many games include some sort of support for the player. This can be friendly characters that cooperate with the player, or a system to give hints and advice.
3. **Support Characters:** Some game worlds are populated by characters that neither directly support nor harm the player. The player can interact with them, but their main purpose is to make the virtual world more realistic.

To fulfill these purposes, several techniques from the broad field of artificial intelligence have found their way into games. The following section describes the most frequently used methods. A distinction is made between methods from "classical" AI, and machine learning methods.

1.1.1 Artificial Intelligence Methods

The methods described in this section are those that have been used in computer games for the last years. In contrast to the machine learning methods described in 1.1.2, these techniques are not able to develop new solutions to occurring problems, but must rely on the rules that were built into them.

Cheating

Even though this method cannot be considered part of artificial intelligence, it must be discussed, because it is very frequently used in commercial games. To overcome problems with weak game AI, opponents in games simply *cheat*. This means they have information or resources that are not available to the human player. This results in creatures that have a 360 field of view, see through walls, move with higher speed or react in a fraction of a second. Strategy games often hide enemy units from the human player but not from the computer.
Cheating is good up to the point where the player notices it. Most players would prefer a cheating AI to an otherwise boring game. Nevertheless cheating is something that most players hate, and therefore game companies try to build better AI that need not rely on cheating.

Game Theory

Mathematical game theory has provided tools to find optimal strategies for some simple deterministic games. Combinatorial games like Nim, Tic-Tac-Toe or Connect-4 were solved, which means that for one of the players an algorithm is known that never loses. It is however practically impossible to solve more complex games like chess, even if theoretical guarantees for the existence of optimal strategies exist in some cases.

Game Trees and Search Methods

One of the oldest principles in game playing is that of **game trees**. A game tree represents the possible states of a game in its nodes and leaves, starting from the root node which corresponds to the initial state. The children of a node represent all states that can be reached with a legal move from the current state. Leaves correspond to terminal states, i.e. states in which the game is over. Game trees are usually used only in two-player games, where players take turns until the game is over. One level of the tree therefore corresponds to a half-move or *ply* by one

player.
A **utility** or **payoff function** assigns to every terminal node a numerical value for the outcome of the game, e.g. +1 for a win, -1 for a loss and 0 for a draw. We assume that higher values are good for the first player and lower values are positive for his opponent. In the simplest possible game, where only one player could make a move, the player would try to maximize its payoff by choosing the move that leads to the highest-valued terminal node. On the other hand, if the second player could react, he would choose the move that leads to the leaf with the lowest utility. This means the first player tries to *maximize* his payoff, while his opponent tries to *minimize* it. This is called the **Minimax**-principle .
If we had the whole game tree, we could apply the Minimax-principle to find optimal strategies for both players. First we would evaluate every terminal node, and then use a bottom-up method to assign a minimax-value to every node in the tree. The minimax-value of a node for the maximizing player is the maximum of the payoffs in all its successors, while minimizing nodes are assigned the minimum of their children's utilities. Using this calculation both players would know which move to make to optimize their payoff, assuming that their opponent plays perfectly.

The problem with the Minimax-algorithm is that the complete game tree for most games is way too large to search all the paths until the end. Instead, the search must be cut off earlier, and a heuristic **evaluation function** must be applied to the resulting nodes. Evaluation functions estimate the winning chances of each player by some numerical value, which must agree with the values of the utility function on the terminal nodes. These functions are often guided by heuristics that humans have developed (e.g. for chess), which are based on some features of the game state. In the last years however successful attempts have been made to *learn* an evaluation function with machine learning methods (see 1.1.2 for details).

In order to find an optimal strategy, the most straightforward approach is to search the game tree from the current position up to a fixed search depth and cut off the tree at this level. Then we apply the evaluation function to the terminal nodes, and use minimax to assign values to all intermediate nodes. Clearly the strength of an algorithm increases with its search depth, but at the same time the computational complexity increases exponentially. The art in applying this algorithm is therefore to restrict the search to only the most interesting regions. Several clever improvements, the most important being **alpha-beta pruning** and **quiescence search** have been successfully used to optimize the performance of minimax-algorithms. See [Rus95] for a discussion of these methods.
Game trees can also be used to some extent in games of chance. The successors of a node are then all states that can be reached from the current node, for *any* outcome of the element of chance (e.g. a die). The minimax-value of a node is then the *expected value* of the minimum or maximum of its successors over all possible random outcomes.
Game playing methods based on game trees are mainly used in "classical" games like chess, checkers or backgammon. Their strength relies mainly on huge computational power to search ever larger parts of the game tree. The famous *Deep Blue* chess computer that beat world champion Garri Kasparov in 1997, relied on 256 parallel processors that could calculate 60 billion moves within three minutes [IBM97]. However, even the developers of Deep Blue say that this is *not* artificial "intelligence", but only the exploitation of the advantages that computers have over humans.

Path Planning

Path planning seemed to be one the major problems in commercial game development in the past. The task is to move a virtual character (e.g. the character representing the player character) from point A to point B without hitting obstacles or getting stuck in dead-ends. The path should be as short as possible, but should look *believable* to the human player. This means, sharp turns and unrealistic paths (e.g. swimming through a river, when a bridge is nearby) should be avoided. The widely accepted solution to this problem is the **A***-algorithm (which dates back to 1968 [Har68]) and its refinements. A^* operates on undirected, weighted graphs, where each node represents some state (e.g. a position) and the weight of an edge is the distance between two nodes of the graph. It starts from the current state and builds a search tree by examining the neighbouring states, until it finds the shortest path to the goal. A^* keeps track of the shortest distance between the start and each visited node, and uses an estimation of the distance to the goal to direct its search to the most interesting states. See e.g. [DeL00] or [Rus95] for a detailed description of the algorithm. A^* always finds a path if one exists, and this path is always optimal if the estimation of the distance to the goal is an underestimation. The main advantage of A^* however is its efficiency in finding the optimal path, compared to other search techniques.
Several heuristics can be applied in order to speed up A^* or produce smoother paths [DeL00]. The graph-representation of the state-space, which is often continuous, plays a major role in the running time of the algorithm. Nowadays A^* is the de-facto standard for 2D path-planning in games. The next step is path-planning in 3D space, which has become increasingly important, due to the recent popularity of 3D games.
Again, path planning, like game-tree search, is no evidence of real intelligence. It is merely the use of a clever algorithm that can fulfill one task - getting from A to B.

Rule Based Methods

The easiest, and most widely used method of simulating intelligent behaviour in older games, is creating a huge set of *IF ... THEN* rules. For the programmer this means that he has to have a lot of experience with the game, because the virtual agents will exactly behave according to these rules. For some parts of the game, where an optimal strategy is obvious, it is however a good idea to use these heuristic rules, because it is simply the most efficient approach.
In recent years, several games have made these *scripts* of rules accessible to the public. Players can then define how their opponents or friendly agents should react, and make the game more interesting for themselves. Of course these scripts can be shared via the Internet.
Not to be mistaken with this simple kind of game AI are *rule-based systems* in artificial intelligence, which actually *reason* about the world. They use a-priori knowledge and percepts to act logically [Rus95]. This kind of AI is clearly different from the simple approach from above, but it is too complicated and computationally expensive to be used outside of toy games.

Finite State Machines

The next best thing to chains of *IF ... THEN* statements is using **finite state machines** (*FSM*). FSMs have a finite number of states, and a transition function that outputs the next state, given the current state and some sensory input.
State machines allow a better structuring of the game AI implementation. The programmer only needs to design behaviours for the prototype states of the FSM, and then think about how certain events require a state transition. Finite state

machines can be used in every part of the game, from e.g. doors that can only be opened or closed, to a full opponent AI.
FSM can be combined with *fuzzy logic* to create **fuzzy state machines** (*FuSM*) . Fuzzy logic is an extension of boolean logic, that can handle the partial membership in logical sets. While in finite state machines the machine is always in exactly one current state, fuzzy machines define the current state through the degree of membership in all possible states. Similarly the transition function changes only these membership levels. The action that is carried out in the end, is a combination of all actions that would be performed in the discrete states, weighted by the degree of membership to these states.
FuSMs remove the need to have only a discrete, finite set of states and actions, because all kinds of intermediate states are possible. Fuzzy machines usually create a more realistic behaviour, without having the designer create rules for all those intermediate situations.
Finite and fuzzy state machines are very popular in commercial computer games and are frequently used. The main reasons are their simplicity, which makes them easy to debug, as well as as the extensive experience with this method in the industry.

Group Behaviour

Many strategy games involve groups of virtual agents that form one unit (e.g. armies). Even though each member of that group is individually controlled, the actions of all team members must be coordinated. Consider e.g. a medieval army consisting of hundreds of virtual soldiers that move in formation between two points. If every individual simply moved along its shortest path to the goal, all the formation would be lost, and the soldiers would block each other.
The most successful algorithm for such herd behaviour is called **flocking**. It is inspired by the behaviour of flocks of birds, schools of fish or swarms of bees [DeL00]. Flocking consists of three simple rules: *Separation* deals with maintaining a minimum distance to the other members of the group. *Alignment* means that agents align themselves with other agents in their neighbourhood. *Cohesion* is the ability to group together with other agents nearby. A fourth rule, which is an extension to the original flocking algorithm, is *avoidance*, which means that local obstacles are avoided by individuals.
The beauty of flocking comes from the fact, that the team members need only information of their local neighbours. The swarm then moves as a whole, needing only one leader that defines the direction of movement. Flocking comes from the field of *artificial life* (*AL*), in which the recreation of animal and human behaviour in the computer is studied.

Hierarchical AI

The artificial intelligence in complex strategy or sports games is often divided into *hierarchical layers*. One layer makes high-level strategic decisions that define the overall goals and plans of the virtual agents as a whole. The second level of hierarchy controls sub-groups that in some sense belong together, e.g. a group of soldiers that together form one army. The third layer directly controls the individuals. Of course architectures with more or less layers are also in use.
This division in the AI makes it easier to execute plans, because the decision-making on a high level does not need to take every single individual into account. Instead the game AI can operate with larger units and make simpler decisions on the lowest level. Hierarchical decomposition of the task also makes it possible to use different AI methods in different layers. The development of a high-level plan can be performed with some rule-based approach or sophisticated learning or planning

techniques, while things like path-planning can be done on the lowest level.

1.1.2 Machine Learning Methods

The above methods have in common that none of them can learn new solutions to the problem. In contrast this section describes techniques that can improve their performance through experience.

Pretending to Learn

Many games that create the illusion of a learning opponent in fact only pretend to do so. A simple way to simulate learning is by introducing random errors into the behaviour of the agent, and reducing the error probability over time. Other approaches simulate learning by including more states in a finite state machine, that become available only after some time has passed or an event occurred.

Neural Networks

Artificial neural networks (*ANNs*) are abstract models of the neurons that form the human nervous system. One neuron receives several input signals, and computes the output as a possibly non-linear transformation of the weighted sum of the inputs. The weights corresponding to the inputs can be learned using a set of labelled training examples. Several neurons can be combined, usually in two or more layers, to form a *neural network*. Given enough training examples and a suitable network architecture, neural networks yield very good approximations of arbitrary functions. Neural networks were very successfully used for approximating evaluation functions (see 1.1.1), e.g. in backgammon [Tes95] or Othello. Neural networks in games can also be used to fine-tune game parameters [DeL01], recognize patterns and make decisions. ANNs can also be used in combination with other methods like genetic algorithms and fuzzy logic to improve their performance.
Neural networks are frequently used in scientific attempts to master games. The game industry however seems a bit sceptical. ANNs require some practice, and often yield results that were not expected. While they could be used for on-line learning and adaption to the player, most commercial games that use neural networks freeze the weights before the game is shipped.

Genetic Algorithms

The idea of **genetic algorithms** (*GAs*) comes from Darwin's theory of evolution. GAs represent their knowledge in a string of parameters, similar to genes, that are interpreted by some program. A *population* of such parameter strings is maintained and permanently updated. In analogy to *selection* in biological evolution, only the *fittest* members of the population survive and reproduce themselves. Fitness in this case is defined by some numerical function that measures the success of the individuals. Virtual *mutation* and *crossover* operators are randomly applied to some members of the population, by altering or exchanging some of the genes. If the application of these genetic operators results in a fitter individual, it will become more successful in the evolutionary selection process. Thus the average fitness of the population increases over time, until finally an optimum is found.
The difficulty with genetic algorithms lies in the definition of the genes and the fitness function. One parameter string can e.g. be interpreted as the weights of a neural network, or weights assigned to features in an evaluation function. The fitness function can be measured as some numerical error, or e.g. as the percentage

of games that the agent encoded by the parameter string wins against some benchmark opponent.
Since GAs obviously require some experience, the game industry has not made much use of them. Some designers use them simply as another optimization algorithm to fine tune game parameters. One field where genetic algorithms have been successfully used is **artificial life** (*AL*). AL studies the recreation of more or less realistic creatures in the computer, and evolution is an important component. Most artificial life games have made use of genetic algorithms.

Data Mining

One way how to learn to play a game is to learn from a human expert. Over the years, huge databases have been created, where recordings of several thousands of matches between human players are collected. The majority of these databases deal with for the game of chess, but there are also some Go, Checkers or Backgammon resources. **Data mining** is an area of machine learning that is concerned with the discovery of knowledge that lies within these huge databases.
Data mining in game databases can construct classifiers that predict whether a state is a winning or losing state, it can reveal patterns that have great impact on the outcome of the game, or it can help recognize and evaluate opening moves, to mention only a few applications. The goal is always to reduce the large amount of raw data to a small number of *concepts* that describe the data.
It seems as if data-mining techniques have almost exclusively been used in classical board games, because there the largest collection of data exists.

Opponent Modelling

Recognizing the plan of your opponent and adapting your style of play often makes the difference between winning or losing when humans play against each other. Computer games however usually only react according to some pre-defined strategy, and so weaknesses in their play can always be exploited by clever human opponents. **Opponent modelling** is concerned with creating a model of the human opponent that shows its strengths and weaknesses. Having this information the strategy of the virtual agent can be changed in order to perform better against this specific opponent. To discover the preferred strategies of the opponent, all kinds of learning algorithms from *supervised learning*, i.e. learning from labelled training examples (like ANNs, decision trees, ...) can be used to predict his actions. *Bayesian networks*, an AI tool to reason under uncertainty, can also be used to model the preferences of a player.
Opponent modelling can also mean giving the virtual adversary its own personality. Sports games e.g. often try to copy the favourite movements of famous stars, which of course could also be learned.
In spite of its potential in creating more human-like agents, opponent modelling is seldom used in computer games. Poker games are the exception, because there the modelling of the opponent is one of the main components of the game.

Reinforcement Learning

Reinforcement learning (*RL*) is concerned with learning strategies through trial and error and delayed reward or punishment. Instead of being given labelled training examples, reinforcement learning explores the world, tries out all possible actions and receives a numerical *reward* for the action. The learning algorithm remembers which actions in which states yielded the greatest reward, and finds its optimal strategy by trying to gain as much cumulative reward as possible in its whole

lifetime. Reinforcement learning is explained in detail in chapter 2.
The big advantage of reinforcement learning is that the artificial agents learn by themselves, e.g. by playing against each other. The AI designer only needs to fix the rewards, which is very easy compared to implementing the whole strategy for a complex game. This makes reinforcement learning very well-suited for problems for which a heuristic solution is difficult to find. The disadvantage of RL is that it requires a large number of training games until the optimal strategy is found. Designers can however use their background knowledge to simplify the learning task and speed up the whole process.
Reinforcement learning was used with huge success in Backgammon [Tes95] and other classical board-game domains. Commercial games have more or less ignored RL until now. Why this is so remains an open question. It may be that game AI designers simply do not know about the potential of reinforcement learning, or they may regard it as too difficult to use.
This thesis tries to demonstrate that reinforcement learning can be a solution to many of the problems encountered in game AI. Reinforcement learning reduces the need for explicit background knowledge from the designer, but we show that simple well-known heuristics can guide the learning system to a better performance. A game based on RL can improve its level of play and adapt itself to its opponent. It is however true that RL requires some experience and may not be the easiest method to use. Starting in chapter 2 we show the potentials and traps of reinforcement learning in games.

1.2 Research vs. Industry: The Dilemma

Artificial intelligence in computer games must meet several criteria. The importance that is assigned to these criteria differs a lot between the research and business community [Lai03] [Woo03].

Optimality vs. Fun

Scientists measure the success of their work by comparing the performance of their system to known benchmarks. This is a good motivation in order to implement game AI that becomes stronger and stronger. The ultimate goal is to find an **optimal** solution, i.e. one that always wins, or at least never loses.
Playing against an optimal opponent however is often not what people want. Every human amateur playing against a grandmaster-level chess program will soon be frustrated after playing several games without any chance of winning. People buy games because they want to have **fun**. Therefore sub-optimal solutions are not only acceptable for computer games, but necessary. This makes it hard to use scientific methods for commercial games, because fun is something that cannot be measured.

Size and Nature of the Problem

Scientific research on games mainly deals with classical board games like chess or Go. Even though these games involve enormous state spaces, the complexity of modern games is even higher. This is obvious because board games are discrete, while in modern games the world is more or less continuous. The nature of the intelligence involved is also completely different: today's artificial intelligence research is focused on games with two players taking turns. The game industry however deals with real-time games, involving sometimes hundreds or thousands of autonomous agents. Classical methods like game-tree search cannot be applied to such problems without great adaptations.

Development Time

Looking at articles of game developers, one can frequently read sentences like: "We planned to use a new technique, but we had to meet the deadline, so we wrote everything as IF ... THEN statements." Researchers usually do not have as strict deadlines as companies do. This is why they can try out time-consuming methods like genetic algorithms or reinforcement learning. Programmers of commercial games cannot risk to delay the shipping of the game because of trying out a new method. Even if they did succeed, this would be no guarantee that the game would sell more copies because of a new AI. This is why old and uninteresting methods stay alive in computer games.

Integration

Games for research projects are usually only the testbed for a new AI algorithm. Therefore it is easy to integrate the AI into the game. In the game industry however, game AI is usually one of the last tasks of the game to be completed. Graphics, sound and basic game elements are already finished when the game AI is written, and the AI programmer has to work with the program interfaces.

CPU Time

Having been one of the major problems in older games, the importance of CPU time in game AI has decreased, due to the speed of modern computers and the availability of high-speed 3D graphic chips. In earlier days, graphics needed the main part of the CPU cycles, and AI had to see what it could do with the rest. Nevertheless, using sophisticated artificial intelligence algorithm in real time games remains a challenge.

Copyright

The goal of commercial games is to make a profit, therefore their source codes are kept secret and are protected by copyrights. In the academic community most software is available to the public. If the game industry could share some parts of their source-codes with the scientific community, both sides would benefit.

Simplicity

It is not surprising that finite state machines still belong to the favourite tools of the game industry. People know how to use them, and they are good enough to satisfy the players.
Scientists on the other hand try to find new and better algorithms that are becoming more and more complex. Even though many people in the game industry have a strong academic background and are interested in such developments, they cannot use them in their games. Therefore the gap between what is applied and what is theoretically available is growing.

Knowledge Transfer

It seems as if researchers on the one hand, and game developers on the other hand, are following divergent paths. Both sides however could learn from each other: scientists have developed methods that could be used in games, and developers have found creative solutions that could inspire new branches of scientific research. Besides, the game industry has the funds to finance artificial intelligence research. Computer games, not only classical board games, must be recognized as a serious area of research. The industry for their part should increase their cooperation with

universities. In any way, the knowledge transfer between AI research and game industry must be increased. Conferences like the Symposium on AI and Interactive Entertainment of the American Association for Artificial Intelligence (AAAI) are a good start.

1.2.1 Conclusion

There is a big discrepancy between the methods used for game playing in scientific research and those used in commercial games. The reason for this is that scientific research is focused on classical games that are of little interest to the game industry. Commercial games however suffer from the pressure of meeting deadlines, so more sophisticated AI methods are avoided to save time.
Players however expect the opponents in games to become more and more human-like. They expect computer controlled agents to act intelligently, learning from their mistakes and adapting to their opponents. It is no surprise that the only games that offer this kind of intelligence to date, namely multi-player network games in which the opponents actually are humans, have become increasingly popular over the last years.
Machine learning techniques would offer the needed tools, but this is often ignored by the game industry. Research on the other hand shows no sign of making the available methods more suitable for the requirements of computer games. Bringing both sides closer together must be a goal for the next years.

1.3 Machine Learning Applications in Classical Games

This section describes success stories of machine learning applications in classical games, like board or card games. Classical games are favoured by the scientific community because of their simplicity. It simply takes a lot more time to implement a 3D shooter game than a chess interface. Even though many commercial programs for board and card games exist, this survey describes mainly the approaches that are of scientific interest.

1.3.1 Checkers

The first application of machine learning in game playing was Samuel's checkers program [Sam59], which dates back to the 1950's. In this work, which is considered one of the most influential early works in machine learning and game playing, he introduced many of the techniques that have been used up until now. The program was based on Minimax game-tree search, but it used machine learning to learn the evaluation function at the cutoff level.
The first method used is called *rote learning*, which is simply based on simply of storing all encountered board positions together with the calculated minimax values. If these states were later found as terminal nodes in the cut-off level of the game tree, the prediction of the evaluation function would be more accurate.
Secondly, Samuel's checkers program included the first application of *reinforcement learning* to learn the weights of the evaluation function by playing against itself. The weights were updated after the next move to shift the estimation of the minimax value of the former position closer to the value encountered after the move. This is the basic principle of *temporal-difference learning.*
Even if Samuel's checkers program was a breakthrough in the application of machine learning to game playing, it did play well but not at master level. Several years later in 1994, *Chinook* [Scha96], another checkers program, became the first artificial

world champion in any game, and has not lost any game since 1994. Interestingly none of the methods that Samuel proposed in his work are used in Chinook.

1.3.2 Chess

Chess is by far the most popular game in AI research. The best chess programs in the world rely on search techniques, database knowledge and huge computational power. They achieve a level of play that is high above the average, and like in the case of *Deep Blue* vs. Kasparov can even beat the human world champion.
Machine learning approaches to the game of chess were not as successful in beating human experts, but several interesting directions of research have developed [Fue96]. First, *data mining* can be used to extract knowledge from chess databases [Fue97]. This in mainly used for opening moves and end-plays. The purpose is to learn *books* of standard moves, that were successful in the recorded games within the database. Some of these data mining techniques use labelled training examples to build *classifiers* that distinguish between winning and losing positions. A third application of data mining would be to induce *features* that are important for the evaluation of game situations.
To speed up the search in game trees, learning methods can also be used to induce order heuristics of moves or to fine tune search parameters.
A different kind of learning is concerned with approximating the evaluation function of chess positions. The weights of all features that influence the evaluation function must be tuned to reach a more accurate approximation. Reinforcement learning techniques combined with neural networks are mainly used to learn this approximation by letting the program play against itself, human opponents or other strong chess computers. Neurochess [Thr95] e.g. used a neural network to estimate the value of each board position, while a second network predicted the state of the board two plies after the current position. A summary of the methods in use is given in [Fue01]. While approaches using reinforcement learning have worked well for other games (e.g. Backgammon, see below), learning via self-play has not lead to comparable results for the game of chess. The lack of variability in the game leaves a great part of possible game states unexplored. Most researchers believe that this is one of the main reasons why self-learning chess programs until now have not mastered the game.

1.3.3 Go

Even if its rules are quite simple, the complexity of the Japanese board game Go is so high, that the usual search techniques have no chance of achieving a human-like level of play. Today's Go programs are far away from beating a human expert. Most even lose against human beginners.
The apparently most promising approach creates sets of rules for playing the game. The induction of these rules is a chance for machine learning methods. Other attempts have been made to learn an evaluation function of board positions and moves. Another direction researches the learning of opening and endgame books. Pattern recognition also seems to have a great potential in finding features that are important for the outcome of a game. A survey of the machine learning methods used for playing Go is given in [Ram01].

1.3.4 Backgammon

In contrast to the above games, Backgammon involves an element of chance, causing search techniques to become even more expensive. Machine learning methods

on the other hand have had great success in the game of Backgammon, more than in any other game. *Neurogammon* by Tesauro [Tes89] used an artificial neural network to predict the evaluation function of board positions from several thousand expert-rated games. For every position several moves, in particular the best and the worst moves, were rated by a numerical value and used as training examples for the ANN. The problem with this approach was, that it is hard for experts to give an accurate numerical value for the utility of a move. In later versions, Tesauro used a method called *comparison learning* which was trained to distinguish between good and bad moves. Thus the network had to learn only the order of the moves, instead of a global evaluation function. Neurogammon became the first program, based primarily on machine learning, that won in a tournament.
The next step in Backgammon playing was *TD-Gammon*, again by Gerald Tesauro [Tes95]. TD-Gammon is one of the most impressive applications of reinforcement learning to date. Without much a-priori knowledge about the game, the program learned through self-play and reached a level near the strongest players in the world. TD-Gammon learned to approximate the evaluation function of board positions using a neural network. The network was not trained by labelled training examples, but by playing hundreds of thousands of training matches against a copy of itself, and receiving a numerical reward of +1 for a win, or else 0. Thus the function learned gave the probability of winning, starting from the current state. Errors in the prediction were propagated back by *temporal difference* (TD) learning, using the TD(λ) algorithm [Sut88]. The approximation of the evaluation function was used for a two-ply game tree search, taking all possible outcomes of the dice into account. The program could then select the move with the greatest expected improvement in utility.
TD-Gammon in its original version played at a level equal to that of the best Backgammon programs at that time, including Neurogammon. The amazing fact about that was, that TD-Gammon used almost zero knowledge about Backgammon, while its opponents relied on the expertise of humans. Later versions of TD-Gammon included specialized Backgammon features, which made the program significantly better than all its predecessors. Other modifications, like an increased search depth and a more complex network architecture raised the playing strength of TD-Gammon to that of human grandmasters. Today TD-Gammon is ranked among the top-three players in the world.
TD-Gammon even showed signs of creativity. The program played certain opening positions differently than most human grandmasters did at that time. After further analysis, the best players in the world since then have changed their style of play, and now use these moves in their openings. Thus, humans have learned from the computer.
Other authors have tried to copy the self-playing method for other games like chess or Go, but none of the results are as impressive as TD-Gammon's. This indicates that the characteristics of Backgammon make it especially suitable for learning from self-play [Fue01]. One reason for its success may also be the large number of training games that TD-Gammon played (over one million). It may be that the learning algorithm converges very late to the real optimal strategy, thus making it difficult to say whether a self-playing approach yields good results in different games.

1.3.5 Poker

Card games like Bridge or Poker are usually games of *imperfect information*, which means that some parts of the game state are hidden to the players. In contrast, players of board games like chess or Backgammon have complete information about the current state of the game. This makes card games a tougher challenge for AI algorithms, but at the same time makes them more interesting, because the

problems are closer to that faced in the real world.
The absence of perfect knowledge makes it insufficient to rely on brute-force search alone. A model of the current game state, which predicts the unknown cards of the opponents must be established.
In Poker one of the most important things that make the difference between winning or losing, is knowing your opponents. Conservative players will tend to fold their cards early, while others take the risk of bluffing. Therefore *opponent modelling* techniques have a great potential in computer Poker programs. One of the best existing Poker programs is *Loki* [Bil99]. Loki records the actions of its opponents, and uses these observations to build a model of their play. Using this model, the program tries to predict the cards that its opponents are holding. The prediction of hidden information uses the fact, that with some combinations of cards the opponent would very likely fold, call or raise. To guide the actions that Loki takes, simulation techniques are used that explore likely scenarios to determine the best action.
Even though Loki does not play at world-championship level, its interesting use of real-time learning makes it another good example for the application of learning techniques in games.

1.4 Machine Learning Applications in Commercial Games

Much of what is known about artificial intelligence in commercial computer games must rely on marketing statements, or interviews with the designers. Whether this is the truth or not, cannot be verified. The statements in this thesis are based on articles available on the net, especially on [GAI03]. The following case studies show commercial games that have made interesting use of machine learning techniques.

1.4.1 Creatures

Creatures by Millennium Interactive is not really a game, but an artificial life simulator. The human player does not have a particular goal, but he observes the development of his virtual creatures, called *Norns*. The interaction of the player involves rewarding or punishing his Norns according to their behaviour, feeding them, or changing the environment in which the creatures live. Thus instead of directly controlling the Norns, the player can teach the creatures whatever he wants them to do, but the reaction cannot be predicted. The creatures grow up and eventually will lay eggs from which new Norns hatch, which inherit genes from their parents. Eggs can also be shared via the Internet, making it possible to introduce new creatures to one's own game world.
Creatures makes use of a technology called *CyberLife*, which combines artificial neural networks and genetic algorithms. ANNs learn to use objects in the environment and to distinguish good from bad behaviour. The manual reward or punishment from the player resembles reinforcement learning techniques, and is probably the target value in training the neural networks. Genetics are used for passing down abilities of parents to their children.
The computational costs of running online learning algorithms are very high. Creatures defines itself as a life-simulator, not a real-time game, and therefore the speed of the simulation is slow enough to simulate life for every single Norn. Using the same intelligence in a real-time strategy game would probably be too expensive in terms of CPU time.

1.4.2 Black & White

In no other major computer game, the effect of learning becomes so obvious as in *Black & White*, developed by Lionhead Studios. In this game the player is the god of a tribe on a small island. With his god-like control the player has to solve several tasks with the help of his people and a creature that he can train. Depending on the nature of the actions that the god takes, the world changes to reflect the behaviour of the player. The land of a good god becomes a fairy-tale kingdom, while nasty actions result in a dark and evil world. The main attraction of the game however is an animal-like creature that the player can train. This creature initially has no knowledge about the world, but it learns by observing the human player, fulfilling orders and receiving reward and punishment for its actions from the god.
The creatures have finite sets of desires that they must satisfy, e.g. they must eat from time to time. The creature can eat anything that exists in the game world, whether it is food or rocks. When the creature becomes curious and eats a rock, it notices that it does not like it, and it does not satisfy its hunger. Food on the other hand tastes good and so the creature will remember that the next time. Creatures that are trained by their master become increasingly helpful, because they try to understand what the player wants them to do.
Some methods used in Black & White are described in [Eva03]. The game uses techniques related to *opponent modelling*, only in the slightly different sense that the behaviour of the player is monitored in order to influence the *supportive* AI. One of the intentions regarding the creature was to make it psychologically plausible. An architecture was used that included *symbolic* representation of beliefs about individual objects, a *decision tree* structure describing beliefs about categories of objects, and a *neural network* to model the desires of the creature. All the objects in the world have some numerical feedback value for all possible actions that describes their utility in achieving the task associated with the action. Rocks e.g. have a lower value for "taste" than food. The creatures use this feedback to build decision trees from all the experience they have gathered. These trees help them in predicting how useful new objects are in satisfying their demands. Their desires are controlled by a neural network, which apparently uses several desire sources (like e.g. hunger) as input and calculates the strength of a desire.
Again learning from reward and punishment comes from reinforcement learning, but it is unknown if RL algorithms are actually used in the game. Nevertheless the combination of classical game AI and learning methods is impressive, and the game is also fun to play.

1.4.3 Games using Neural Networks

Early games that integrated neural networks, used this term mainly for marketing reasons to give the impression of human-like AI. *Battlecruiser: 3000AD* was probably the first non-classical game on the market that claimed to use ANNs [GAI03]. They used supervised and unsupervised learning algorithms for basic decision making. This involved modelling opponent and partner personalities, and route finding in a vast galaxy. All non-player characters in the game are controlled by small neural networks. Nevertheless, the concept did not work out very well and the AI is considered to be rather weak by most players.
Heavy Gear is a game in which the player controls a combat robot. Neural networks are used to increase the level of support that the robot gives to the player. Robots may become faster in reloading the weapons or repairing damage.
Other games, like *7th Legion* have tried to integrate neural networks, but finally had to replace them because they had to meet their deadlines [GAI03]. Other games have made more conventional use of neural networks, e.g. for fine tuning some game

parameters.
It seems that of all machine learning methods, neural networks have the greatest potential of becoming standard tools in the game industry. Developers have gathered some experience with them, and development kits for ANNs are available.

Chapter 2

Reinforcement Learning

A great part of artificial intelligence in computer games is concerned with implementing good strategies for computer-controlled opponents. The goal is to give the human players the illusion of playing against another human being [Sal99]. For a programmer it takes a lot of experience with the game to write a program that plays similar to humans. Still the implemented strategies may be far from optimal, because most games are too complex to find an optimal solution for every single situation in the game.
Reinforcement learning (RL) is one way to solve problems like this, with a minimum of a-priori knowledge required. RL takes advantage of the fact that it is easier to define the goal of a strategy, than the strategy itself. The artificial opponent can learn its strategy through interaction with the game environment and feedback that tells it the results of its actions. Thus there is neither a need for a human teacher (like in supervised learning), nor a need for the programmer to have a complete understanding of the game.

2.1 Reinforcement Learning Model

The formal model of reinforcement learning [Kae96], [Mit97], [Sut98] consists of :

- A set S of **states**
- A set A of possible **actions** that the agent can take
- A numerical **reward** function
- A **transition function** $\delta : S \times A \to S$

In this model the agent interacts with the environment in the following way: it perceives a description s of the current state and then chooses an action a. The action changes the state of the environment to $s' = \delta(s, a)$ and produces a *reward* or *reinforcement signal* $r(s, a)$ that is passed to the agent. This reinforcement signal indicates the quality of the selected action. The task of the agent is therefore to find a **policy** $\pi : S \to A$ that chooses an action a_t in the state s_t at the time-step $t = 0, 1, 2, \ldots$ so as to maximize the possible **cumulative reward** over time. In general, r and δ may be stochastic functions, as well as π may be a stochastic policy.
A formal definition of the cumulative reward expected at the time-step t is the **return**

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

where $0 \leq \gamma \leq 1$ is the **discount factor** that describes the present value of future rewards [Sut98]. Rewards later in the future have a smaller value than immediate rewards if $\gamma < 1$. If $\gamma < 1$ this also has the effect that the sum R_t is bounded, as long as the r_t are bounded. If $\gamma = 1$ the sum may diverge.
The environment, specified by the functions $r(s,a)$ and $\delta(s,a)$ can be stochastic, but we assume that it has the *Markov Property*. That is the rewards and transitions depend only on the *last* state and the last chosen action, not on the whole sequence of states, actions and rewards since the beginning of an episode. Formally

$$\begin{aligned} &Prob(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) = \\ &= Prob(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t) \end{aligned} \tag{2.2}$$

A reinforcement learning task that satisfies the Markov property is called a *Markov Decision Process* or *MDP* [Sut98]. In the following we assume that all learning problems are MDPs.

2.1.1 Value Functions

In order to estimate the quality of a policy π, the most common way is to estimate how desirable it is to be in a given state, or how useful it is to choose a certain action in that state [Sut98]. We define the **value** of a state s under a policy π as the expected discounted return if we start from state s and follow policy π [Sut98]:

$$V^{\pi}(s) = E_{\pi}\left[R_t | s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right] \tag{2.3}$$

Here E_π defines the expected value, if the agent follows the policy π. In order to evaluate different actions in a state we define the **action value** or **Q-value** $Q^{\pi}(s,a)$ of a state-action pair as the expected discounted reward if we take action a in state s and then follow policy π:

$$Q^{\pi}(s,a) = E_{\pi}\left[R_t | s_t = s, a_t = a\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right] \tag{2.4}$$

For a given policy π the functions V^π and Q^π can be estimated from real or simulated experience (e.g. *Monte-Carlo methods*). This is referred to as **policy evaluation**.

Using value functions, we can introduce a partial order on policies. A policy π is better than or equal to another policy π', if $V^{\pi}(s) \geq V^{\pi'}$ for every state $s \in S$. This definition implies that by following policy π we receive at least as much reward as if we followed policy π'.
An **optimal** policy π^* in this sense is a policy that satisfies $V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s)$ for all $s \in S$. We define this value as $V^*(s) = V^{\pi^*}(s)$. Optimal policies also have optimal action-values, and so $Q^{\pi^*}(s,a) = max_{\pi} Q^{\pi}(s,a)$ for all states $s \in S$ and all actions $a \in A_s$, where A_s is the set of all actions that are possible in state s. Similarly to V^* we define $Q^*(s,a) = Q^{\pi^*}$. From the definition it is clear that

$$\max_{a \in A_s} Q^*(s,a) = V^*(s) \tag{2.5}$$

because $Q^*(s,a)$ is the expected return if we take action a in state s and then follow an optimal policy. So we can express $Q^*(s,a)$ via $V^*(s)$ as:

$$Q^*(s,a) = E\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right] \tag{2.6}$$

If we combine 2.5 and 2.6 we get the recursive equation

$$Q^*(s,a) = E\left[r_{t+1} + \gamma \max_{a' \in A_{s_{t+1}}} Q^*(s_{t+1}, a') \;\middle|\; s_t = s, a_t = a \right] \tag{2.7}$$

2.7 is called the **Bellman optimality equation** for Q^*.

2.1.2 The Learning Task

The task of the agent is to find a policy that maximizes its discounted cumulative reward. With the above definition this is equivalent to a policy π^* such that [Mit97]

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s) \qquad \forall s \in S \tag{2.8}$$

or in other words

$$\pi(s) = \arg\max_{a} [r(s,a) + \gamma V^*(\delta(s,a))] = \arg\max_{a} [Q^*(s,a)] \quad \forall s \in S \tag{2.9}$$

Therefore if we knew the optimal action-value function $Q^*(s,a)$, we could construct an optimal policy by selecting in each state the action that yields the highest Q-value. Knowing Q^* makes it possible to find an optimal policy without knowing anything about the dynamics of the environment. So we have reduced the reinforcement learning problem to learning the optimal Q-function.
If we knew only V^* we could only find an optimal policy if we knew the transition function $\delta(s,a)$ and the reward function $r(s,a)$ in advance, because they are needed to look one step ahead. Note that in some cases, especially in deterministic and simple environments like that of board games, it makes sense only to learn the V-function, because there the model of the environment is perfectly known [Tes95] [Sam59].

2.2 Learning Algorithms

In order to learn the Q^*-function, we must know its value for every single state-action pair (s,a). For small problems with finite S and A, one could simply learn a lookup-table with one entry for each pair. In the following section we assume that the problems are small enough to store the Q^* function in tabular form. We will consider problems with larger (possibly infinite) state and action spaces later in section 2.3.

2.2.1 Policy Iteration

The general idea behind the learning algorithms in reinforcement learning is that of **policy iteration**: starting from an arbitrary policy π we use *policy evaluation* to estimate the Q^{π}-function of this policy. Given this estimation $\hat{Q}^{\pi}$ of the Q-function, we can construct a new policy π' that is greedy with respect to $\hat{Q}^{\pi}$, i.e. it always chooses the action a in state s that maximizes $\hat{Q}^{\pi}(s,a)$. This is called **policy improvement**. The *policy improvement theorem* guarantees that this policy is really an improvement over π, because $\forall s \in S$:

$$V^{\pi'}(s) \geq V^{\pi}(s) \tag{2.10}$$

Given this new improved policy, we can estimate $Q^{\pi'}$ and perform another improvement step. Policy iteration describes this process of sequential evaluation and improvement.

If policy improvement does not change the current policy, then it must already have been optimal , because only optimal policies cannot be improved anymore. For finite MDPs this algorithm is guaranteed to converge to an optimal policy, because there is only a finite number of policies.
Instead of alternating policy evaluation and policy improvement, most learning algorithms interleave both processes. This is called **generalized policy iteration** [Sut98]

2.2.2 Action selection

The problem of constructing a policy from the learned Q-function is handled very similarly by all learning algorithms. They all must solve the dilemma of *Exploitation* versus *Exploration*. Exploitation describes the process of selecting a greedy policy with respect to the current estimate of the Q-function. This estimate however may still be faulty, because not all actions were tried in some states. Exploration is the process of selecting apparently sub-optimal actions to discover new characteristics of the state-action function.
This tradeoff between exploitation and exploration is solved by following the greedy policy with respect to the estimate of Q^*, but selecting random actions from time to time to ensure exploration. The easiest way to do this is by following an ϵ-*greedy* policy. For any probability $0 \leq \epsilon \leq 1$, the greedy action is selected with probability $1 - \epsilon$, and a random action is selected with probability ϵ.
Another way of selecting the actions from an estimate of the state-action values is the *softmax*-rule . Here the probability of selecting action a in state s is weighted according to the current estimate of $Q(s, a)$ by using some probability distribution. The most common method uses the *Boltzmann*-distribution: In state s, action a is selected with probability

$$\pi(s, a) = \frac{\exp(Q(s, a)/\tau)}{\sum_{a' \in A_s} \exp(Q(s, a')/\tau)} \tag{2.11}$$

where $\tau > 0$ is a parameter called the *learning temperature* . Higher temperatures make the selection more random, while for $\tau \to 0$ the selection becomes the greedy action selection.
Both methods have their advantages: ϵ-greedy selection is simpler to implement, and for most people it is easier to set the probability ϵ than the temperature. Softmax however has the advantage, that actions with higher Q-values have a higher probability of being selected, while ϵ-greedy assigns a uniform probability to all actions. Softmax is therefore especially well suited for stochastic environments, where it makes sense to select different actions at different times in one and the same state.

2.2.3 Temporal Difference Learning

Temporal Difference (*TD*) learning is the most widely used method in reinforcement learning. TD methods learn the Q-function directly from experience without needing a model of the environment. They try to reduce the differences between estimates of the state-action values at different times. The update of Q-values is based in part on other estimates that were previously learned. This process is called **bootstrapping**. TD-methods have the advantage that they are easy to implement and can be used for *on-line learning*.
TD-learning algorithms differ mainly in the way they estimate the state-action value function. The two most important algorithms, *Q-learning* and *SARSA* , as well as their extensions $Q(\lambda)$ and $SARSA(\lambda)$ are described below.

Q - Learning

Watkins [Wat89] introduced this learning algorithm that approximates Q^* by calculating a fixed point in the Bellman equation 2.7. The update rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right) \quad (2.12)$$

Here α is a constant step-size parameter, sometimes called the *learning rate* . Note that the update is only based on the immediate reward and the value estimation for the successor state. The algorithm directly approximates Q^*, independent of the policy that is being followed [Sut98], since the max operator always computes the value for the greedy policy. This kind of learning is called *off-policy.* If every state-action pair is visited infinitely often, and the learning rate α is decreased over time, Q-learning is guaranteed to converge to Q^* [Wat92].
The complete learning and control algorithm is given in Algorithm 1:

Algorithm 1 Q - Learning [Sut98]

```
Initialize Q(s,a) arbitrarily
for A number of episodes do
  Set s to the initial state
  repeat
    Choose a in s, using ε-greedy or softmax action-selection
    Take action a and observe reward r and new state s'
    Q(s,a) := Q(s,a) + α (r + γ max_a' Q(s',a') − Q(s,a))
    s := s'
  until s is a terminal state
```

SARSA

While Q-learning directly approximates Q^*, SARSA [Rum94] tries to estimate Q^π for a given policy π and simultaneously changes this policy towards a greedy policy with respect to Q^π. Therefore the approximation of the optimal state-value function goes along with an improvement of the policy being followed. Methods like that are called *on-policy learning* . The update rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \ (r_{t+1} + \gamma \ Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (2.13)$$

Here a_{t+1} is chosen in s_{t+1} according to the current policy, which depends on the current estimate of the Q-function. Because of its dependence on the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ this algorithm is called *SARSA*. The convergence of the algorithm is guaranteed if every state-action pair is visited infinitely often, and the followed policy gradually changes towards a greedy policy with respect to $Q(s,a)$ [Sin00]. This can be reached e.g. by decreasing ϵ in ϵ-greedy policies, or decreasing the learning temperature τ for Softmax.

Q(λ) and SARSA(λ)

Both Q-learning and SARSA base their update of $Q(s,a)$ only on the immediate reward and the estimate of the direct successor state. The real Q-value however depends on all successors, until the end of the episode. Sutton [Sut88] introduced a mechanism called **eligibility traces** to speed up the convergence of the above *one-step* learning algorithms. The eligibility trace of a state-action pair measures the influence of a future TD update. They are used to propagate the TD error, i.e. the amount by which a Q-value is changed in an update step, back to the predecessor

states.
Eligibility traces for a state action pair are defined as

$$e_t(s,a) = \begin{cases} \gamma\ \lambda\ e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\ \lambda\ e_{t-1}(s,a) & \text{otherwise} \end{cases} \quad \forall\ s,a \tag{2.14}$$

$0 \leq \lambda \leq 1$ is the so called *trace-decay parameter*, because the eligibility traces of all states that are not visited in one step decay by the factor $\gamma\lambda$. The higher λ, the more influence future updates have on recently visited states.

By combining Q-learning or SARSA with the mechanism of eligibility traces, we get two new learning algorithms, $\mathbf{Q(\lambda)}$ and $\mathbf{SARSA(\lambda)}$. $SARSA(\lambda)$ [Rum94] is a straightforward extension of the original SARSA algorithm. Algorithm 2 shows the complete algorithm:

Algorithm 2 $SARSA(\lambda)$ [Sut98]

```
Initialize Q(s,a) arbitrarily
Set e(s,a) = 0 for all s ∈ S, a ∈ A
for a number of episodes do
  Set s to the initial state
  Choose the initial action a for state s
  repeat
    Take action a and observe reward r and new state s'
    Choose action a' in s', using ε-greedy or softmax action-selection
    δ := r + γ Q(s',a') − Q(s,a)
    e(s,a) := e(s,a) + 1
    for all s ∈ S, a ∈ A do
      Q(s,a) := Q(s,a) + α δ e(s,a)
      e(s,a) := γ λ e(s,a)
    s := s'
    a := a'
  until s is a terminal state
```

The new update rule is similar to the old SARSA-rule 2.13, but here the TD error $\delta = r + \gamma\ Q(s',a') - Q(s,a)$ is multiplied by the eligibility trace $e(s,a)$. So the $SARSA(\lambda)$ update rule becomes

$$Q(s,a) \leftarrow Q(s,a) + \alpha\ e(s,a)\,(r_{t+1} + \gamma\ Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{2.15}$$

Note that this update is performed for *all* state action pairs (s,a), not only the last encountered pair (s_t,a_t).

The $Q(\lambda)$-algorithm by Watkins [Wat89] is similar to $SARSA(\lambda)$, but special care must be taken of the eligibility traces if an exploratory (i.e. non-greedy) action is selected. Q-learning learns about the greedy policy, while it is following a policy that may improve pure random moves [Sut98]. TD errors caused by exploratory actions must not be propagated back to preceding greedy moves. Therefore the eligibility traces must be cut off, i.e. reset to zero, whenever a non-greedy action is selected. The update rule for $Q(\lambda)$ is

$$Q(s,a) \leftarrow Q(s,a) + \alpha\ e(s,a)\left(r_{t+1} + \gamma\ \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right) \tag{2.16}$$

By setting $\lambda = 0$, we get the original Q-learning or SARSA algorithms. The advantages of algorithms that use eligibility traces (with $\lambda > 0$) become evident

when dealing with problems of long-delayed rewards. In order to propagate the reward back to all state-action pairs that were visited before the reward was received, one-step methods would need to revisit every single state along that possibly very long chain. $Q(\lambda)$ and $SARSA(\lambda)$ on the other hand would update all state-action pairs that lead to a reward, at the moment it is received. Especially for on-line learning tasks this compensates the computational effort.

2.3 Generalization and Function Approximation

The learning algorithms from section 2.2.3 learn the state-action value function Q by learning one value $Q(s,a)$ for every state-action pair (s,a) and storing the values in a lookup-table. Often however, the size of the state or the action space is too large to keep a table of the size $|S| \cdot |A|$ in memory. It is even possible that the problem has infinitely many states or actions. On the other hand, even if there were no memory limitations, the state-action space may be so large that it would take way too much time to learn accurate state-action values for every single pair (s,a). What is needed is a representation of the Q-function that depends only on a finite set of parameters. These parameters can be learned from a set of training examples, and should be able to predict accurate values even for those state-action pairs that were not used for training. Fortunately these topics have already been extensively studied in **supervised learning** , which is the most important sub-field of machine learning. The general task in supervised learning is to learn an approximation of a function that depends on several attributes. The input of the learning algorithm is a set of *labelled* training examples, i.e. some *teacher* must tell the algorithm the desired output. The result of the learning process is a representation of the function that predicts values for all possible examples, just from their attributes. In our case the attributes are observations that define the state, called **features**, and the output is the value of the Q-function. Making use of the results from supervised learning, we can choose from a great number of existing algorithms, and take the one that best fits the problem.

2.3.1 Function Approximation

Instead of representing the Q-function as a table for every state-action pair, we approximate it as a function $\hat{Q}$ of a finite-dimensional *parameter vector* $\vec{\theta}$:

$$Q(s,a) \approx \hat{Q}(s,a,\vec{\theta}) \tag{2.17}$$

$\vec{\theta}$ represents all parameters of the underlying approximator, e.g. the coefficients of a linear or polynomial function, the weights of an artificial neural network, or the splitting and regression information of a regression or model tree.
The parameter s indicates the state, but this does not mean that every state must have its unique label. Instead it is now possible to represent the state too as a vector of *features* $s = (s_1, \ldots, s_n)$ and train the approximator with these features as the attributes. This is often a much more natural definition than assigning distinct labels to states. Think e.g. of a two-dimensional grid-world, where the state of an agent consists of the (x,y)-position and orientation ϕ. The natural definition of the current state would be $s = (x, y, \phi)$, but in the tabular case we would need to assign a label to every triple.

2.3.2 Performance of the Approximator

The performance of an approximator is usually measured by the **Mean Squared Error** (*MSE*) of the approximation [Sut98]

$$MSE(\vec{\theta}) = \sum_{s \in S} P(s) \left[Q^{\pi}(s,a) - \hat{Q}(s,a,\vec{\theta}) \right]^2 \tag{2.18}$$

Here Q^{π} is the state-action value function of the policy to be evaluated, and $P(s)$ is some distribution weighting the errors of different states. Usually $P(s)$ is the distribution of states at which backups are done, e.g. the frequency with which states are encountered during learning. The task in function approximation is to find a *global minimum* in the MSE, i.e. a parameter vector $\vec{\theta}^*$ for which $MSE(\vec{theta}^*) \leq MSE(\vec{\theta})$ for all $\vec{\theta}$. Most of the times finding this vector is not feasible, and the best we can get is a *local* minimum, i.e. a parameter vector which is optimal in a certain neighbourhood. Depending on the nature of the approximation, different methods from mathematical optimization can be used to converge to this optimal parameter vector $\vec{\theta}^*$.

2.3.3 Gradient Descent Methods

One popular method for function approximation is **gradient descent**. Gradient descent can be used if the approximation $\hat{Q}(s,a,\vec{\theta})$ is a smooth differentiable function of $\vec{\theta}$. For optimizing $MSE(\vec{\theta})$ we utilize the fact, that the **gradient** of a function always points in the direction of steepest ascent. Thus the negative gradient gives the direction of steepest descent. If we shift a given parameter vector $\vec{\theta_t}$ a little bit to the direction of the gradient of $MSE(\vec{\theta_t})$ over the training examples, we expect the MSE to decrease. If we continue this procedure of decreasing the error via gradient descent, the parameter vector approaches a local optimum. One update step is given by

$$\vec{\theta}_{t+1} \leftarrow \vec{\theta_t} + \alpha \, \left(Q(s_t,a_t) - \hat{Q}(s_t,a_t,\vec{\theta_t}) \right) \, \nabla_{\vec{\theta}_t} \hat{Q}(s_t,a_t,\vec{\theta_t}) \tag{2.19}$$

where α is a small step-size parameter, $Q(s_t,a_t)$ is the current estimate of the state-action value at time t, and $\nabla_{\vec{\theta}_t} \hat{Q}(s_t,a_t,\vec{\theta_t})$ is the gradient of the approximator with respect to $\vec{\theta_t}$.
Gradient descent techniques are well-suited for online-learning, because the update-step can be made immediately after the visit of the next state.

Linear Methods

One special case of gradient descent learning is that, in which the Q-function is a weighted linear combination of features that define the state. Each state s is represented by a feature vector $\vec{\phi_s} = (\phi_s(1), \ldots, \phi_s(n))^T$, and the action-value function is calculated as the linear combination

$$\hat{Q}(s,a,\vec{\theta}) = \sum_{i=1}^{n} \theta(i)\phi_s(i) \tag{2.20}$$

The gradient, which is needed for the update according to equation 2.19 is particularly simple:

$$\nabla_{\vec{\theta}} \, \hat{Q}(s,a,\vec{\theta}) = \vec{\phi_s} \tag{2.21}$$

This simple form makes the mathematical analysis a lot easier, and therefore this is one of the few function approximation algorithms in reinforcement learning,

for which convergence results exist [Sut88] [Tsi97]. The disadvantage of linear approximators is that they are not able to take combinations of features into account. Therefore it is up to the designer of the learning algorithm to select an appropriate way of coding states with features. (See [Sut98] for an overview of several coding techniques)

Artificial Neural Networks

Another popular approach is approximating the state-action value function by an artificial neural network (ANN), typically by a multi-layer ANN. In this case $\vec{\theta}$ represents the weights of the connections in the neural network. The ANN is then trained by the *Backpropagation* algorithm, which is explained in detail e.g. in [Bis95].
Although the convergence of reinforcement learning with function approximation using ANNs is not guaranteed, there are a number of very good results that have been achieved with this method (e.g. [Tes95] [Zha96] [Cri96]).

2.3.4 Function Approximation in Games

In his pioneering work Samuel [Sam59] used a linear approximator to learn the value of board positions in the game of checkers . Linear approximators however are often not expressive enough to represent the very complex functions often encountered in all but the simplest games.
The success of Tesauro's TD-Gammon [Tes95] makes the neural-network-approach very attractive. Nevertheless, ANNs require a large number of training games, and yet may not be able to discover certain features of the action-value function. Especially, neural networks tend to produce a rather smooth approximation of a function, while in some games the real value-functions have discontinuities. If these discontinuities depend on only a small set of features, a very clever design of the neural network, possibly with many (pre-trained) layers is necessary to represent this in the approximation.
One way to overcome this problem is to first split the state-space into disjoint regions, and then learn a local approximation of the value-function in these regions. Variables with great impact may then be used as splitting criteria, to represent their influence on the Q-function. In the regions, the approximation can be very simple, e.g. constants or small linear functions. In this thesis we present an algorithm that has these favourable characteristics, using *regression* or *model trees* for the approximation. Chapter 3 covers this learning algorithm in detail.

2.4 Hierarchical Reinforcement Learning

When a problem becomes too large to solve it as a whole, the typical approach is to divide it into smaller sub-problems, and solve them sequentially. Robots e.g. typically have some set of basic actions like "Move from A to B", "Pick up object O", ... and a high-level controller that switches between these low-level actions to reach a desired goal, like "Find the exit from a maze without hitting the walls". Researchers have tried to find ways to incorporate this principle into reinforcement learning.
The need for hierarchical decomposition is obvious: Many learning tasks involve subgoals that must be reached sequentially, before any other task can be completed. A usual AI approach would be to write a program for each subtask and later combine them. In the classical RL framework we would need one large state space that contains all subgoals. The agent would have to solve the whole learning

process before it can improve its policies for reaching the subgoals.
One would also often like to integrate so called *modules* into the learning process. Modules are sub-programs that solve one specific part of the problem. Path finding is one good example: There are many existing algorithms that can find the shortest path between two points very quickly and reliably. So one would like to use them, instead of waiting until the agent has learned to find its way by itself. This makes the integration of a-priori knowledge into the learning process possible, but modules that are difficult to implement can also be learned. The use of these modules is mainly to decrease the size of the action space. If e.g. an agent in an economic game has chosen to buy stocks, one could use a learned module that determines which stocks, and how many of them should be bought. The alternative would be to have one action for every stock and every quantity, which would result in a huge action-space.

Algorithms for hierarchical reinforcement learning try to learn policies efficiently in terms of training time, while tolerating a slight sub-optimality in performance [Kae96]. It is up to the designer to choose a hierarchy that is well-suited to find near-optimal policies. In this section some of the most important hierarchical RL methods are presented, without claiming to be exhaustive.

2.4.1 Module-Based Reinforcement Learning

One common approach in the hierarchical decomposition of reinforcement learning tasks is to have a collection of *modules* or *behaviours*[1] and learn a *switching* or *gating* function that activates the modules. The decision which behaviour to choose is based on the current state of the environment. Low-level actions are performed only in the modules, though it is possible to simply define one primitive action as an own module.
The most intensively studied approach is to fix the behaviours in advance and learn the gating function through reinforcement learning. The construction of modules of course requires a-priori knowledge by the designer. This method is well-suited for robotics tasks (see e.g. [Mae90]), where the behaviour-based approach has already been used very successfully. A more recent publication by Kalmár et.al. [Kal98] introduced the term **module-based reinforcement learning** for this kind of learning algorithms. The high-level controller uses the modules instead of primitive actions. He decides from the current state and the learned switching function, which module to invoke, and waits until the termination of the module. Then a new behaviour is selected.
This method is well suited for tasks, where the definition of modules is easy. Sometimes however, the design of the behaviours is more critical than the gating function. In this case we can use the dual approach to the above method. The switching function is fixed, and only the modules are trained [Mah91]. For games this approach is very useful in situations where only certain components of the AI should be learned, and otherwise good heuristic rules exist.
The third alternative is to learn both the high-level switching and the behaviours from reinforcement (see e.g. [Dor94]). This is the most complex method, and requires a lot of training time. First the modules must be trained, before a reasonable switching function can be learned. Later both hierarchical levels can improve their performance simultaneously.

[1] Other terms in use are *skills*, *partial policies* or *options*

2.4.2 Options

A formal framework for using *macro-actions*, i.e. sequences of actions, in reinforcement learning tasks is given in [Sut99]. An **option** is a macro-action that consists of a triple (I, π, β). I is the set of all states in which the option becomes available. π is the policy that is executed by this option. β is the termination condition which for every state gives the probability that the option terminates in that state. The policy of an option can select any other option. Primitive actions are always included as one-step options. Therefore the options framework allows the use of extended activities to reach certain subgoals, while at the same time permits the learning at finer grain wherever necessary.
The necessary adaptations in the learning algorithms are given in [Sut99]. The key is to take the duration of multi-step options into account when calculating the cumulative discounted reward.

2.4.3 MAXQ

Dietterich [Die00] developed the MAXQ Value Function Decomposition, in which the core problem is decomposed into a hierarchy of subtasks. For the solution of one subtask, primitive actions can be executed, or policies to solve lower-level subtasks can be invoked. In contrast to the above options formalism in 2.4.2, MAXQ includes the *calling stack* in the state information. This stack gives the names and parameters of the hierarchy of higher-level tasks that were called in order to activate the current sub-task.
A hierarchical policy in this framework must assign one policy for every subtask. The value function for a hierarchical policy can then be decomposed into values for low-level actions and so called *completion functions* that give the expected return if a policy for a subtask is executed until completion. This is the basis of the learning algorithm described in [Die00]. The policies for sub-tasks can also be trained, by using so called *pseudo-reward functions.*

2.4.4 Feudal Q-Learning

The idea of *feudal reinforcement learning* is to have a hierarchy of learning modules or *managers* that are organized in a master-slave architecture. One module can select among several sub-managers, to which it can assign tasks. The sub-managers can then set tasks for their sub-sub-managers and so on. Only the lowest-level modules perform primitive actions. The reinforcement signal for one module comes from its master, only the highest level receives rewards from the environment. Managers must reward their sub-managers for doing what they asked them to do, whether or not this satisfies their own goals. Therefore the lower levels can learn what is expected from them quite rapidly, even if the high-level goal is never reached.
Feudal Q-Learning was introduced in [Day93], where it was used to divide the state space into regions, for which sub-managers that achieve certain sub-goals can be trained efficiently. The simultaneous learning of high- and lower-level modules implies that in early stages no sensible policies can be learned in the higher levels, because the actions are more or less random. Later however feudal Q-learning is quick to converge towards a good strategy.

2.4.5 Discovery of Subgoals

The above methods all assume that a-priori knowledge is available to specify subgoals in advance. Other authors have studied to discover important subgoals of a task automatically. A recent approach by McGovern [McG01] identifies subgoals

as so called *bottleneck states*, which are regions in the state space that are visited frequently on successful, but not on unsuccessful episodes. Creating options that achieve these goals and learning to combine these options on a higher level can result in a significant speed-up of the learning algorithm.

2.4.6 Other Approaches

There are several approaches to hierarchical RL that have not been mentioned in the above sections. Singh's [Sin92] *compositional reinforcement learning* trains behaviours that can achieve certain conditions and combines them to sequentially achieve sub-goals.
Kaelbling's [Kae93] *hierarchical distance to goal* approach partitions the state space into regions of which the centers are called landmarks. If a goal lies within a region, low-level actions are used to move to the goal. Otherwise the high-level moves to the next landmark on the shortest path to the goal-region. This approach is especially useful for problems in which the goals are dynamically input to the learner.
Parr [Par98] introduced the method of *hierarchies of abstract machines.* This approach restricts the room of possible policies by replacing low-level actions with predefined programs. The programs are represented as stochastic finite state machines that use their internal state as well as the state of the learning process to choose the next action. These machines have four types of states which can execute low-level actions, call other machines, terminate the execution or non-deterministically choose the next machine-state. In these *choice states* reinforcement learning can be applied to learn a policy for the hierarchy of abstract machines.
A survey of modern hierarchical approaches to reinforcement learning is presented in [Bar03].

2.5 Reinforcement Learning in Games

So far we have assumed that learning occurs while the agent runs through a number of training episodes and interacts with its environment. In games however, there are typically more than one acting agents, and the performance of a strategy depends not only on the reaction of the static or dynamic environment, but also on the behaviour of all other agents involved. But how can we play training episodes, when the behaviour of *all* agents is undefined?
The solution is, that agents can actually learn by playing against themselves. In the beginning, all agents perform only random actions, but through the rewards received during the game, the agents learn and improve their abilities to play the game. Since all agents learn simultaneously, they have to compete against ever stronger opponents. Thus they learn to adapt to many different kinds of opponent strategies.
This learning approach has been used very successfully for game playing. The two most well-known examples are Samuel's checkers program [Sam59] and Tesauro's TD-Gammon [Tes95].

Multiagent Reinforcement Learning

In a game with two or more players, an agent must take the states of his opponents into account when choosing its next action. If we simply define the opponents as part of the environments dynamics, the Markov property of the learning process and with it the theoretical foundation of most learning algorithms is lost. Results however have shown, that in spite of this violation of the Markov property, the algorithms still work quite well.

A theoretically sound implementation of reinforcement learning in multi-agent environments must take the reactions of the opponents into account. Littman [Lit94] introduced the *Minimax-Q*-learning algorithm, in which methods from mathematical game-theory are used to find an optimal policy for two-player problems. The Q-function $Q(s, a, o)$ now depends on two actions: a is the next action of the agent, and o is the following action of the opponent. The optimal action that the agent can take in state s now is given by

$$\pi^*(s) = \arg\max_a \min_o Q(s, a, o) \tag{2.22}$$

That is, an action is optimal if it yields the maximum reward, assuming that the opponents plays optimally. This is the same *minimax*-principle, that is used frequently in AI approaches to game-playing (see chapter 1.1.1).

While in theory the Minimax-Q algorithm yields better results than accepting the opponents as part of the environment, it is in practice often infeasible to implement. The reasons are the high memory-requirements and the problem of generalization, because for convergence, every state-action-action triple must be visited.

Chapter 3

Regression and Model Trees

In chapter 2.3 it was shown that for large reinforcement learning tasks it is necessary to approximate the Q-function, using some method from supervised learning. For games in particular we saw that gradient descent methods may sometimes fail to discover important characteristics of the value-function, because of their tendency towards smoothing the approximation. For most approximators it is also very difficult to learn strong local dependencies on a small set of variables, which often occurs in games.

Example 3.1: Consider a chess-playing program that learns to evaluate board positions. The position of one pawn is usually not so important in the beginning of the game. So the feature that represents the pawn's position will have only a small influence on the evaluation function. If however in the endgame this is our last pawn and it was free to change into a queen on the next turn, this feature would definitely have a great impact on the value of the position.

Being able to weight the importance of certain features differently in distinct situations is therefore a desirable quality of function approximation schemes for complex games.

3.1 Decision Trees

For classification tasks in supervised learning **decision trees** have shown great ability to deal with problems like this. Decision trees recursively partition the attribute-space into regions by choosing one attribute as a *split criterion*, and split the training examples according to the value of the split-attribute. The splitting criteria are chosen so as to increase the homogeneity in the target variable, i.e. the *class*. Splitting stops if the regions meet some homogeneity criterion, like the class entropy. The region is turned into a leaf of the decision tree, and a class value is assigned to the whole region. To overcome the problem of *overfitting*, i.e. the tendency to perfectly predict the training data, but failing on other data, the decision tree is **pruned**: subtrees are removed and replaced by leaves that predict one class value for the whole region. There are many different learning-algorithms for decision trees that differ especially in the way they are pruning (see e.g. [Qui86], [Qui93]).
The resulting decision tree predicts the class of an instance only by looking at the splitting criteria along the path in the tree, and choosing the class in the leaf at the end of the path. Therefore the learning algorithm chooses only the most relevant attributes for class prediction and ignores irrelevant features. The recursive

partitioning however makes it possible to represent local dependencies in different branches of the tree.
One big advantage of decision trees is, that the resulting model is easy to interpret, and can easily be transformed into a set of logical rules. The disadvantage is, that most decision tree algorithms work only offline, i.e. all training examples must be known in advance. Existing trees therefore cannot be further improved by additional training examples, without including the old training examples in the training set.

3.2 Regression Trees

The problem in function approximation is not predicting some discrete class, but a real valued function. Breiman [Bre84] introduced the CART (*Classification and Regression Trees*) system, which not only was one of the earliest works on decision trees, but also outlined an extension of the concept that can predict numerical values. A **regression tree** uses attributes for splits in the tree nodes just like in decision trees, but predicts a numerical value in the leaves, instead of a class. The algorithms for building regression trees are very similar to decision-tree learning algorithms. The difference lies in the determination of the split criterion, and the assignment of values in the leaves.
To choose the split attribute, we need a new measure of homogeneity, because we have to deal with continuous values instead of discrete classes. One natural measure is the *variance* of the target variable. The split criterion is thus the attribute which causes the greatest reduction of variance in the resulting subtrees, compared to the variance of the whole set.
The prediction of the target variable is given by constant values in the leaves. The constant assigned to one leaf is simply the mean of the target values of all training examples that fall into that leaf.
After the full tree has been grown, pruning is again used to improve the prediction on unseen examples. The pruning methods are straightforward extensions of the methods for decision trees.

3.3 Model Trees

Regression trees predict only one constant value for all examples that fall in to one region. For complex, high-dimensional functions, regression trees therefore need to have a lot of leaves in order to approximate the function with sufficient accuracy. Quinlan [Qui92] therefore proposed an algorithm called **M5** which has *linear models* in the leaves instead of constant values. Trees like this are called **model trees**. Model trees do not necessarily have linear models in the leaves. Torgo [Tor97] e.g. used kernel regressors as leaf-models, and in principle every function approximator can be used. Most algorithms however use linear models because of their efficiency and the experience with this method.

3.3.1 Multivariate Linear Regression

Before we define an algorithm to build a complete model tree, we give an algorithm to estimate a linear model, given a set of training examples.
Linear regression is the estimation of a linear function $f(x) = bx + a$ from a training set $T = ((x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N))$, which best predicts the given training points. The usual method is to reduce the *mean squared error* (MSE)

between the training points and the function. The estimated parameters a and b are then given by the equations

$$b = \frac{\sum_{i=1}^{N}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{N}(x_i-\bar{x})^2} \tag{3.1}$$

$$a = \bar{y} - b\bar{x} \tag{3.2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$ and $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ are the means of the attribute x and the target variable y respectively.

If the approximation should be linear in multiple variables, i.e.

$$y = f(x_1, \ldots, x_n) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{3.3}$$

The fitting of the coefficients w_i such that the error over a training set $T = ((x_{11}, \ldots, x_{1n}, y_1), \ldots, (x_{N1}, \ldots, x_{Nn}, y_N))$ is minimized is called **multivariate linear regression**. We define

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{Nn} \end{pmatrix}, \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \vec{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \tag{3.4}$$

where X is the $N \times n$ matrix containing the attribute values and the constant 1 in the rows. $\vec{y}$ is the vector of all N target values and $\vec{w}$ is the vector of $n + 1$ coefficients. Then we can rewrite equation 3.3 for all training examples in T as

$$X \cdot \vec{w} = \vec{y} \tag{3.5}$$

The task is now to find a coefficient-vector w such that the mean squared error between the actual target values y_i and the predictions $\tilde{y}_i = w_0 + \sum_{j=1}^{n} w_j x_{ij} = \vec{w}^T \cdot \vec{x_i}$ for all training examples is minimized. This is equivalent to minimizing $(X\vec{w} - \vec{y})^T \cdot (X\vec{w} - \vec{y})$, which yields the result

$$\vec{w} = (X^T X)^{-1} X^T \cdot \vec{y} \tag{3.6}$$

This result is the so called **least squares solution** of the minimization problem. The solution 3.6 applies only, if $X^T X$ is invertible, which is e.g. the case if the columns of X are linear-independent. For the rest of the section we assume that this is the case. Otherwise more sophisticated solution techniques like *singular-value decomposition* (see [Schw97]) need to be applied.
The easiest (though sometimes numerically unstable) method to solve equation 3.6 uses a *Cholesky-decomposition* $X^T X = LL^T$, where L is a left-triangular matrix (i.e. $l_{ij} = 0$ for $j > i$). The whole algorithm is then given below:

Algorithm 3 Multivariate Linear Regression [Schw97]

1: Decompose $X^T X = LL^T$ using Cholesky-decomposition
2: *Forward Substitution*: Solve the equation system $L \cdot \vec{z} = X^T \vec{y}$
3: *Backward Substitution*: Solve the equation system $L^T \vec{w} = \vec{z}$
4: $\vec{w}$ is the least squares solution

The left-triangular form of L makes the solution of the linear equation systems in steps 2 and 3 of the algorithm very efficient. Therefore this algorithm is more efficient than a direct matrix inversion like in equation 3.6. The Cholesky-decomposition can also be computed very efficiently and is a standard algorithm in most linear-algebra packages (see e.g. [Schw97] for a detailed description and algorithm). The total computational complexity of the algorithm is $O(n^3 + n^2 N)$.

3.4 Quinlan's M5 Algorithm

Quinlan [Qui92] introduced **M5**, the first algorithm to build a model tree with linear functions in the leaves. The greatest part of scientific work on model trees (e.g. [Kar92], [Cha94], [Ale96], [Tor97]) are variants or improvements of this algorithm. A detailed summary of Quinlan's algorithm, combined with a discussion of weaknesses and possible improvements follows.

The Learning Task

Initially we are given a training set $T = ((x_{11}, \ldots, x_{1n}, y_1), \ldots, (x_{N1}, \ldots, x_{Nn}, y_N))$ of N training examples. The task is to learn a function $f(X_1, \ldots, X_n) = Y$ that predicts the target value Y from the attributes $(X_1, \ldots, X_n)$ with minimal error. The function is constructed only by relating the attributes of the training examples to the corresponding target values. The error of the learned model however is in general measured by the accuracy on unseen examples.

Growing the Tree

Growing the model tree involves selecting split criteria in the nodes, and deciding when to stop splitting. M5 uses the *standard deviation* of the target variable as the measure of homogeneity. We define $\sigma(T)$ as the standard deviation of the target values in the training set T. Let $y(t)$ be the target value of the training example t, and let $\bar{Y}$ be the mean of all target values in the training set T, then the standard deviation can be estimated as

$$\sigma(T) = \sqrt{\frac{1}{|T|-1} \sum_{t_i \in T} (y(t_i) - \bar{Y})^2} \tag{3.7}$$

To find the optimal split criterion, we have to evaluate all possible tests that split the training set into subsets by examining only one attribute X_i. Usually only binary trees are constructed, i.e. a *split-value* or *threshold* s_i for attribute X_i is chosen and the result of the split is two sets $T_0 = \{t \in T \mid x_i(t) < s_i\}$ and $T_1 = \{t \in T \mid x_i(t) \geq s_i\}$.
The chosen test is the one that maximizes the error reduction

$$\Delta_{error} = \sigma(T) - \sum_{i \in \{0,1\}} \frac{|T_i|}{|T|} \sigma(T_i) \tag{3.8}$$

The problem with this estimation of purity in the leaves is that by using 3.8 as an error estimator, one pretends that actually a regression tree is built, which approximates the target values in the leaves only by the mean. The actual error may be much lower, because later a linear regression is calculated. In [Mal01] an example is shown, where M5 fails to find the right split-criterion because of this weakness. RETIS [Kar92] overcomes this problem by calculating a linear model for every possible split. This increases the computational costs significantly, and therefore many attempts have been made to combine the efficiency of M5 with the accuracy of RETIS (e.g. the SMOTI (*Stepwise Model Tree Induction*) algorithm in [Mal01]).
Instead of using only one attribute for the split, one can also use splitting *planes*. In [WanX99] e.g. the splitting plane is chosen among the planes that lie mid-way between two training examples.

After the split criterion has been selected, splitting proceeds recursively on the subsets. The growing of the tree stops, if the number of training examples in a node is lower than some constant, or the standard deviation falls below a certain threshold.

Building the Linear Models

Multivariate linear regression functions are calculated in all the nodes of the tree with an algorithm like Algorithm 3 in 3.3.1. M5 however does not use all the attributes but restricts itself to all attributes that are used as splitting criteria or in other linear models in the subtree at the current node. Initially this means, that only the mean target values are stored in leaves. This changes however, when the tree is pruned.
RETIS [Kar92] calculates linear models using all attributes. Torgo [Tor97] proposed even more complex models like kernel estimators. On the contrary Alexander et. al. [Ale96] proposed using only one attribute variable in the linear models all the time. The tradeoff in all these algorithms is between accuracy and small tree-size on the one hand, and efficiency on the other hand.

Pruning the Tree

When the tree has been fully grown, it usually *overfits* the training data, which means the prediction is too specialized and will fail on many unseen examples. Therefore, like in decision and regression trees, the model tree has to be pruned.
Pruning model trees means that the linear model in a node is chosen instead of the subtree below the node. To estimate the error on unseen examples, M5 uses a heuristic weighting of errors according to the number of parameters used in models. The idea is that models that were constructed from a small number of examples, but include a large number of parameters, are very likely to overfit. Therefore the error estimates are multiplied by the factor

$$\frac{n+v}{n-v} \tag{3.9}$$

where n is the number of training examples and v is the number of parameters in the model. To find the optimal model, M5 simplifies the linear models by removing the parameters with the smallest contribution to the model. Even though this increases the error on the training data, the multiplicative factor 3.9 decreases, and so the total estimation of the error may decrease. In some cases M5 removes all the variables, and leaves only a constant value. One must decide if the removed variables from models below a node should be used for later regressions or not. In general including all variables produces smaller trees [WanY97], but increases the running-time of the algorithm.
If the error of the simplified linear model in a node, multiplied by the factor 3.9, is smaller than the error of the subtree, the node is converted into a leaf.
Other pruning techniques, used e.g. by CART [Bre84] use a separate *pruning set* and cross-validation, to prune the tree during the growing phase. The freely available version of M5 in the WEKA-package[1], which is based on Wang and Witten's algorithm *M5'* [WanY97], a variant of M5, multiplies the number of parameters v in the numerator of 3.9 by a constant ψ called *pruning factor* . A larger constant will therefore increase the level of pruning and produce smaller trees. The default value in the WEKA-package is $\psi = 2$.

[1] available at www.cs.waikato.ac.nz/ml

Smoothing the Prediction Function

Smoothing is used after the construction of the tree has finished, and values should be predicted by the tree. The predicted values at the nodes along the path from the root to the leaf are used to adjust the output value of the model in the leaf. This is a desired effect for the approximation of smooth functions, and when the models in the leaves were constructed from only a small number of examples. M5 uses a backup from the leaf to the root [Qui92]:

- The predicted value at a leaf is the predicted value of the model in the leaf.
- The prediction p' at a node is

$$p' = \frac{np + kq}{n + k} \tag{3.10}$$

  where p is the prediction of the subtree below, n is the number of training examples in that subtree, q is the prediction of the model in the current node and k is a smoothing constant. Quinlan's suggestion was $k = 15$.

Instead of smoothing the values at every prediction, the linear models in the leaves can be pre-computed to reflect the effects of smoothing. This can be done by combining the linear models along the path from the root to a leaf, and adjusting the coefficients in the leaf models. [Fra98]
In many cases, smoothing is known to enhance the performance of model trees [Qui92] [Fra98] [WanY97]. In some cases however, especially when the function to be approximated is known to have discontinuities, it may be better not to smooth the predictions.

3.5 Reinforcement Learning with Regression and Model Trees

Model trees meet all the requirements for function approximation in reinforcement learning. Several aspects make model trees very appealing for function approximation in games:

1. Model trees can handle discontinuities in the Q-function that frequently occur in complex environments like that of computer games.
2. Model trees can handle both discrete and continuous attributes.
3. Model trees are easy to interpret. The splitting criteria define logical rules that can easily be checked for their soundness.
4. Model trees can handle problems of very high dimensionality.
5. Model trees can build local models for regions of the state-space, ignoring irrelevant attributes.
6. Model trees can take combinations of attributes into account.

There are however some disadvantages that model trees have compared to e.g. neural networks:

1. Model trees cannot be used for on-line learning.
2. To refine the predictions of a model tree, we must rebuild it from scratch, using all the training examples that were used for the initial model tree.

It is however possible to refine the linear models in the leaves by simply using an gradient descent algorithm like that in chapter 2.3.3. Even further splittings of leaves are possible. The most difficult thing in order to develop an on-line algorithm for model tree induction would be to change the splitting criteria in interior tree nodes. Until now no algorithm is known to achieve this effect.

3.5.1 Earlier Results

One of the first successful applications of regression trees in reinforcement learning came from Sridharan and Tesauro in [Sri00]. They used regression trees to directly approximate the Q-function in different multi-agent economic environments. In all the tasks, two agents were taking turns in setting prices for their products that were bought by several consumer agents. The consumers were restricted to buy at most from one seller, thus the competitors had to adjust their prices to maximize their long-term profit. The reward came from selling products with the set prices at discrete time-steps. The state-action space was sufficiently small to use a lookup-table representation for comparison. To be more precise, the (discretized) price of the opponent defined the current state and the (discretized) price of the agent itself defined the action.
In an earlier work [Tes99a] tabular methods were shown to converge quickly. In [Tes99b] however, approximation with neural networks needed excessively long training times to find a reasonably good policy. This prevents their use in online-scenarios, where quick learning is necessary to stay profitable. Regression trees were thus studied to find another way of approximation that needed less time and training examples, while generating policies of equal quality.
The algorithm that was used for the regression tree was quite simple: variance was used as the split criterion, and no pruning or smoothing was performed. Only one tree had to be built, because the action (i.e. the price set by the own agent) was included in the attributes, together with the price of the other agent. A third attribute was calculated, which took on the value of 1 if the learner's price was less than the opponent's price and zero otherwise. This was supposed to help the algorithm discover important discontinuities in the learned function. The target value was of course the Q-value of the state-action pair.
Initially, random training examples were created by calculating random prices and the resulting profit, which yielded the reward as the target value. These examples were used to construct a first regression tree. Then repeated sweeps through the training examples were performed, and the target values were updated by the Q-learning rule (see equation 2.12 in 2.2.3). The maximum needed for the Q-learning backup was taken from the current tree. After each sweep, a new tree was built with the updated Q-values, and after a fixed number of sweeps the algorithm terminated. The learning rate α was held constant and high ($\alpha \sim 1$), which gave good results, even though convergence theorems require a decreasing α.
The results of this procedure were very promising: instead of hours or even days of training time for the neural networks, learning a regression tree took only nearly one minute. The resulting policy is comparable to the policy that was learned using a lookup-table, and is superior to the ANN policies. The only advantage of neural networks was found in the approximation of flat and smooth parts of the Q-function. These areas however are more or less irrelevant. The important parts are near discontinuities, and here regression trees perform much better.
It was found that larger training sets yielded better policies, but performance rose very quickly in the beginning. The tree size also grew with the size of the training set, but was substantially smaller than the size of the full lookup-table. Raising the minimum number of examples in a leaf produced smaller trees with slightly inferior

performance. Usually the tree included 300 to 600 nodes, with the minimum number of examples per leaf set to 5.
The authors of [Sri00] found that regression trees offer a much better scaling for problems with large spaces, like those occurring in multi-agent environments. They propose an improvement of their algorithm that uses more sophisticated splitting criteria, or linear models in the leaves, which leads to using *model trees*. The other open question is how to adapt this batch learning algorithm to online learning.

Another pioneering work on the use of regression trees in reinforcement learning comes from Wang and Dietterich [WanX99]. They studied value function approximation using model trees in the job-shop scheduling domain described by Zhang and Dietterich in [Zha96]. This is a combinatorial task where the system learns to adapt schedules to satisfy time and resource constraints. There are several notable differences between the tree-construction method they used and e.g. M5. First, the splitting criteria were not using only one attribute, but splitting planes in the whole state space were constructed. The candidate planes were calculated as the planes that lie mid-way between randomly selected training instances.
Second, the fitting of linear models in the leaves made use of a sophisticated composite error. This error estimation included three terms: first the *mean squared error* between the approximation and a good a-priori estimation of the value. The second term was the *Bellman-error*, which is the difference between the approximated value and the value predicted by the recursive definition of the value function. The third error term is the *advantage error*, which measures the negative influence of sub-optimal decisions in the policy. The three terms were combined to a weighted sum, and the linear models were fitted to minimize this composite error.
Training examples were generated and one *batch* training procedure was started to learn the regression tree. The performance of the learning algorithm was very similar to that of a highly-tuned neural network in [Zha96]. Training the ANN however took at least 50 times longer than training the model tree. This is a strong indication that for tasks like this, regression trees are much more efficient function approximators, while yielding almost the same performance as highly-tuned neural networks.
The authors admitted that the nature of the learning problem is likely to be very well-suited for tree-based regression. Incremental learning cannot be achieved by this method, so they propose an alternation of exploratory search and batch learning for tasks where this is required. This idea is integrated in the learning algorithm given in the next section.

A simple tree-based approximation scheme for online learning was proposed in [Pye98]. This system collected the values of the TD errors in the leaves of a decision tree, and performed a split when a threshold in standard deviation of the data in a leaf was exceeded. The prediction in a leaf was the mean of all Q-values in that leaf. Some examples were given in [Pye98], where the algorithm performed better than neural networks. Unfortunately, no other results are available, and so it remains an open question, if this algorithm can be successfully used in complex domains such as games. Obviously bad splits near the root of the tree are never removed, and no pruning is done. This appears to be a serious limitation for the ability to generalize. It is also probable that these trees can grow extremely large in complex domains.

These papers have demonstrated the qualities of tree-based regression in certain environments. This makes the use of regression and model trees instead of linear or neural approximators an interesting research topic for function approximation in reinforcement learning. This thesis is (to the best of our knowledge) the first publication to investigate the use of tree-based function approximation for reinforcement

learning in a complex computer game environment.

3.5.2 A Q-Learning Algorithm using Model Trees

Summarizing the ideas of [Sri00], we can give an abstract description of a Q-learning algorithm that uses model trees for approximation:

Algorithm 4 Q-Learning with Model Trees

1: Create a training set T of random training examples
2: Set the target value of the examples in T to the immediate reward
3: Construct a model tree from the training set T
4: **repeat**
5: Update all training examples, using the Q-learning rule with the maximum Q-value taken from the current model tree
6: Construct a new model tree from the updated training examples
7: **until** a fixed number of sweeps is reached
8: The final model tree gives the approximation of the Q-function

This algorithm is suitable for smaller problems, where constructing a model tree does not take too long. In games however, we have to deal with extremely large state-action spaces. The example of TD-Gammon [Tes95] has also shown, that a large number of training matches (from 300.000 to 1.5 million) was required to learn a game like Backgammon. For games that are even more complex, we must expect to need at least as many training examples to construct a virtual agent that plays at an expert level. This increases the running time of the tree-construction algorithm significantly, and makes it impossible to build a new tree after each sweep through the training data. Another aspect is that we cannot rely only on random games to learn an optimal behaviour. If the opponents follow more elaborate policies, we must expect the success of certain actions that worked well in a random game to decrease. In [Sri00], trees in the multi-agent case were reconstructed after calculating the immediate reward that would result if everybody followed their learned policies. In [WanX99] it is proposed to alternate exploratory search following the current policy and batch learning of a new regression tree. Thus we need new training examples that reflect the change in the agents' policies.
All this makes the use of a reinforcement learning algorithm like Algorithm 4 not very well-suited for learning tasks in games. A modification that is more practical for RL in games is given below:

The above algorithm of course is only a heuristics, and no convergence theorems for it are available. This was however the algorithm that was used in this thesis to learn a complex game, and the results are quite promising (see chapter 5). The algorithm has several properties that make it attractive for learning in games:

- It limits the number of computationally expensive tree constructions, which saves a lot of CPU time.
- The performance of the learner can be increased by including more training games.
- Although the algorithm runs offline, performance increases steadily with every new tree, because the followed policy converges towards the optimal policy.
- The algorithm is well suited for multi-agent learning.
- The granularity of the learning steps can be controlled by altering the number of training games between learning sweeps.

Algorithm 5 RL for Games with Model Trees

1: Let the computer play a number of random training matches against itself and log the states, actions and immediate rewards
2: Initialize the target values to the immediate rewards
3: **repeat**
4:   **repeat**
5:     Sweep through the training data and update the target values by an *on-policy* update rule like SARSA (see 2.13 in 2.2.3), using only the *direct successor* state and its current Q-value for the update
6:   **until** the Q-values converge
7:   Construct a model tree from the updated data. Possibly build one tree for every action.
8:   Play a number of new training matches, using a policy derived from the current approximation. Add the loggings of states, actions and rewards to the training set.
9: **until** a fixed number of training games are played
10: The final model tree gives the approximation of the Q-function

- The number of learning parameters that need to be tuned is small, compared to e.g. artificial neural networks.

The above algorithm leaves room for several variants and alternatives. First, it is possible not to update the older training data and run the SARSA algorithm only on the most recent data. Or one could update the older training data with an off-policy algorithm like Q-learning, and use on-policy learning only for the new examples which were obtained by following the improved policy. This would in some way correspond to re-playing the old training matches and choose in each occurring state the best action according to the current approximation.
Second, parts of the older training examples could be removed to save CPU time. This would increase the influence of the new training data, which were obtained by following an improved policy. It is however necessary to include training data with low Q-values in the training set, because otherwise all predictions will be too high. In order to reduce the size of the trees, the parameters for the model tree algorithm should be adapted, when the size of the training set grows. Larger trees usually tend to overfit the training data, and fail to generalize on unseen examples. Especially the minimum number of instances in a leaf, and pruning constants are important parameters to reduce the tree size.

3.5.3 Conclusion

Although not many publications about the use of regression and model trees for function approximation in reinforcement learning exist, the results appear promising. Tree-based regression seems especially well-suited for games, because it is - in contrast to neural networks and linear approximators - capable of perfectly modelling discontinuities and local dependencies in the value function. Tree-based approximators have to be trained offline, which makes it necessary to develop new algorithms for generalization in RL. An algorithm suitable for games is proposed here, but we expect to see new developments in this area in the next years.
The next chapter describes an interesting game for artificial intelligence research. Then we describe the approaches that were chosen to let a virtual agent learn how to play this game, using a model tree for function approximation.

Chapter 4

Settlers of Catan

Klaus Teuber's *The Settlers of Catan* (*SoC*) or *Die Siedler von Catan* [1] is probably the most popular modern board game in the German-speaking area. It was elected game of the year in Germany in 1995 and in the US in 1996. Settlers was translated into several languages and is now played worldwide.
The well-balanced rules make SoC both fun to play and a challenge for serious players. Strategic planning, negotiation skills and a bit of luck are all necessary to win the game.

4.1 Rules of the game

Goal of the Game

Four settlers are colonizing an island called Catan by building roads, settlements, cities and other developments. Their colonies produce resources, which can be used to expand the colony. The players are awarded victory points for their buildings and other special achievements. The first player to reach ten victory points wins the game.

The Board

The island of Catan consists of 19 hexagonal fields of different types. There are four forests, four fields, four meadows, three hills, three mountains and a desert. Each land produces a characteristic resource:

- Forests produce wood.
- Fields produce wheat.
- Meadows "produce" sheep (or wool).
- Hills produce clay.
- Mountains produce ore.
- Deserts do not produce any resource.

The island is surrounded by water fields, of which nine are ports. Ports are explained later in section 4.1 about trading. The hexagonal fields are arranged like in figure 4.1. The players can build settlements and cities on the corners, and roads on the edges of the land fields. Players cannot build roads or settlements on water, but it is allowed to build on the coast.

[1] ©1995 by Kosmos Verlag

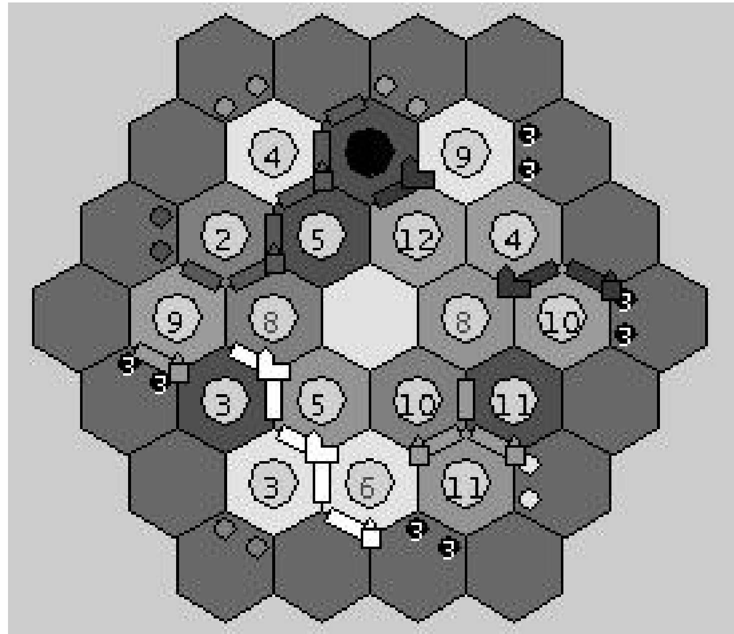


Figure 4.1: A typical situation in a Settlers of Catan game

Every land field, except for the desert, has a number between 2 and 12 associated with it. This number represents the production probability, as is explained in the next section.

Production of Resources

The goal of the game is to colonize the island by building roads, settlements and cities. To do this, the players need resources that enable them to build new objects. The resources are produced by existing settlements and cities. Whenever a new player begins his next turn, two dice are rolled. The sum of the dice is compared with the numbers assigned to the land fields. Every field of which the number equals the rolled number produces resources. A player who has a settlement on the corner of one of those fields receives one piece of the produced resource. A city even doubles the production. The distribution of the dice has mean 7, therefore it is easy to see that fields with numbers close to 7 will produce more resources in the course of the game than e.g. fields with the number 2 or 12. It is therefore favourable to build your settlements adjacent to fields with a high production probability. When the sum of the two dice equals 7, no resources are produced, but the *robber* comes into play. Initially he is placed in the desert, but later the player who has thrown the seven can put him on any land field on the board, except for the desert and the field where he was standing before. The robber blocks the production of the field on which he is standing. Even when the number of the field is thrown, the field does not produce any resources. The player who has placed the robber can steal one resource from any of the players that have settlements or cities next to the field on which the robber was placed. Another effect of a sum of 7 is that all players who have eight or more resources must discard half of them. Therefore it is a good strategy to use the resources as soon as possible, because collecting too many resources increases the risk of losing them on the next seven.

Building

At the beginning of the game each player builds two settlements and two roads, which have to be adjacent to the settlements. Later the players can build more roads to expand their colonies, build new settlements to increase their production, or build new cities to upgrade their existing settlements. They can also buy development cards that can have miscellaneous effects (described in section 4.1). The costs of building must be paid to the bank, i.e. they are lost for the player. Table 4.1 shows the costs of building:

Object to build	Costs
Road	1 Wood, 1 Clay
Settlement	1 Wood, 1 Wheat, 1 Sheep, 1 Clay
City	2 Wheat, 3 Ore
Development Card	1 Wheat, 1 Sheep, 1 Ore

Table 4.1: Building costs in Settlers of Catan

Roads can be built along any free edge of a land field that is adjacent to an existing road, settlement or city of the same player. Settlements can be built on any free corner of a land field that is connected to a road of the same player and is at least two corners away from the next settlement or city. When a player builds a city, it replaces one of the existing settlements of the same player. Each player has only 15 roads, 5 settlements and 4 cities that he can build. However, when a city replaces a settlement, this settlement becomes available again, and can be built somewhere else (after the necessary resources are paid).

Development Cards

For 1 Wheat, 1 Sheep and 1 Ore a player can buy a development card. There are 5 types of cards available:

- **Knights** drive the robber away. Having the largest number of knights also yields important *victory points* (see 4.1).
- By playing out a **Monopoly Card** a player receives all resources of one type from all his opponents.
- By using an **Invention Card** the player can immediately produce two resources of any kind.
- With a **Road Construction Card** the player can immediately build two roads.
- **Victory Point Cards** yield 1 victory point (see 4.1).

If the player buys one card, he will receive randomly one of these cards. Only one card per round can be played out, and a card cannot be played out in the same round that it was bought, except for victory point cards. The development cards that the players buy are hidden from their opponents.

Trading

Normally one settlement is adjacent to three production fields (except for settlements next to water or the desert) and therefore can produce up to three different resources. Nevertheless it is very rare that one player possesses at one time all the

resources necessary for building new objects, because only certain combinations of resources are needed. This is why trading is one of the most important factors of the game. Players can trade with their opponents, or with the bank. No limitations are made on trades between players, even though trades where each of the trading partners exchanges only one resource (1:1 trades) are the rule, and trades with more resources (like 2:1 or 3:1 trades) are the exception. When trading with the bank, each player may exchange four pieces of the *same* resource for one other resource. Better trades can only be made when a player has a **port**. There are two different kinds of ports: For every resource there is a **2:1 - port**, where two pieces of the corresponding resource can be exchanged for one piece of any other resource (e.g. with a wood port, you can exchange 2 pieces of wood for 1 clay). The other four ports are **3:1 ports**, where 3 pieces of the same resource can be exchanged for one piece of any other resource. A port is obtained by building a settlement on one of the coast fields that are marked with the port signs. Building a city there does not improve the port.

End of the Game

The player who first reaches 10 or more **victory points** and has his turn, wins the game. Players collect victory points by building settlements and cities. For every settlement the player is awarded one victory point, for every city two. Also, all victory-point development cards yield one point each. Then there are two special awards that amount to two victory points each: the first award is for the player with the largest army, i.e. the largest number of knights. The first player to play three knight cards receives this award. Later this award can be lost if another player plays at least one more knight.
The second award is for the player with the longest road. The first player to build a continuous chain of roads of length five or more (not counting any branches) wins the award. The award is lost if another player builds a chain of roads that is at least one road longer. A chain of roads is broken by opponent settlements that are built between two roads of the chain. It is therefore possible to destroy an existing chain of an opponent by building a settlement in a "hole" of his longest-road chain.

4.2 Tactics of the Game

A typical Settlers game is divided into three phases [Teu00]: The *starting phase*, the *intermediate phase* and the *endgame*. To be successful, all three phases require a different playing style.
Since Settlers of Catan has not yet been extensively studied in game AI research, we give a detailed explanation of the basic tactics for the game. The goal of this section is to give the reader an idea of the complexity of the decisions that SoC players must make.

4.2.1 Starting Phase

In the first phase, the players place their first two settlements and roads. The general rule is that the first settlements should be next to at least one of the most valuable 6- and 8-fields, or at the intersection of other good production sources. The mix of resources is the next important thing. [Teu00] suggests that the first two settlements should border on all five resources. Other sources like [McP03] suggest a more sophisticated mix of strategies, depending on the availability of certain resources. The goal in all starting strategies is to have an early advantage, because the production of resources increases with the number of settlements. This "exponential growth" is responsible for the fact that early investments in settlements and

cities usually pay off in a huge advantage in the intermediate and end-game.
If a resource is particularly rare, one should try to have at least one of the better production fields, or a high price will later have to be paid to get this resource. This is especially important for ore and clay, because there are only three production fields.
On the other hand, a player may try to get a lot of one common resource. The chance is high that the opponents will rate this resource as relatively cheap and will give them away easily during trades. The player may then exchange 2, 3 or 4 cards of this type for the resource he needs. Playing this strategy it is crucial to secure a port later in the game, especially the port belonging to the common resource.
A slightly different strategy tries to secure a de-facto monopoly on one resource, by occupying two good production places for one resource. Other players will have to pay a high price in trades with the producer or the bank later in the game.
Another basic strategy in the game is to occupy good clay and wood fields in the beginning, because then roads can be built early in the game. This has the advantage that good settlement positions in the neighbourhood can be secured before the opponents can interfere. Later in the game however, one has to build a settlement that produces enough ore to be able to win the game. Building roads early also gives the chance to disturb other players' expansions, and to secure the award for the longest road. This is however risky, because one usually cannot win the game without good production.
The contrary strategy relies on wheat and ore, to build cities as soon as possible. Building cities early increases the production significantly, because usually the production of the first two settlements is better than of all remaining settlements. Having an early advantage in resources, one usually has enough resources to trade, in order to build roads and settlements. If a constant production of sheep is available, buying development cards also becomes part of the strategy.
It is not a good tactic to rely on one or two extremely good fields, like an ore-6 or a clay-8, because these fields usually attract the robber. The worst thing is to place both settlements next to the same good field, making this the favourite robber target for all other players.
Other placement criteria concern the distance between the first two settlements, and the distance to ports and opponents. If one decides that he wants to build the longest road (e.g. because the first settlement produces a lot of clay and wood), the second settlement should be built in the neighbourhood of the first. This has the effect that later in the game the road network attached to each settlement can be closed, resulting in a longer road. It is however usually not a good strategy to connect the first roads, because then two roads have to be built before a new building place is reached. If building the longest road is not a goal, then the distance between the first settlements is of no importance.
Building the first or second settlement on a port field is usually not a good idea, because at least one of the neighbouring fields is water. Water does not produce anything, but production is the most important thing in the beginning and dominates the utility of a port. The third or fourth settlement however should be a port, and so it is important that the first settlements are close, i.e. at most 3 roads away from useful ports. Of course one should always build towards ports where resources can be exchanged, of which the player has a high production.
The distance to the opponents is the third important thing to be considered in placing the initial settlements and roads. If one builds in the center of the map, there is a risk of being trapped by the other players. In the worst case all possible building places are lost. Therefore all players should try to have more than one possible way of building towards new settlements. The standard procedure is to build the first roads towards the coast, because usually nobody builds his initial settlement next to water. Therefore several escape routes are open for the expansion of the road

and settlement network.
Which of the tactics for the initial building of settlements and roads can be applied depends also on the placement order. The first player can build his first settlement on the best position on the map, but he is last to build his second settlement. Therefore the selection of the building place depends heavily on the choices of his opponents. The fourth player however can build both of his settlements at the same time, which allows him to coordinate the placement. Even if his first settlement may not be his favourite, this usually is an advantage for the game. All other players should not base their strategy on the hope that a particular position will be left free for them.

4.2.2 Intermediate Phase

The intermediate phase starts with the first roll of the dice and ends when one of the players reaches 8 or more victory points. Therefore the intermediate phase makes up the largest part of the game.
The goal in this phase is to collect more victory points than the opponents. There are five overall strategies to fulfill this task:

1. **Expansion:** Build new settlements to increase the production of resources. Ports should also be secured to have better trading possibilities. This is the most important strategy in the game.

2. **Conversion to cities:** Settlements at good positions should be converted into cities to increase the production and have more settlements available for expansion.

3. **Longest road:** Two extra victory points are awarded to the player with the longest road. It therefore pays off to build roads without the immediate goal of building new settlements.

4. **Development cards:** Buying development cards has several positive effects. Road construction, monopoly and invention cards allow to save valuable resources. Knights drive the robber away, and a large army has the chance of winning the two extra victory points for the largest army. And of course there are direct victory point cards which are extremely valuable.

5. **Confrontation:** One must have a look at the opponents. Sometimes it pays off to build roads or settlements that block the expansion of an opponent, or even destroy the longest road.

A combination of these five strategies is necessary for a good intermediate play. Which of them is chosen depends on the resources available, and this again depends on the dice. Usually players try to expand their colony in the first phase of the game. City conversion becomes crucial when all settlements are used up. The construction of the longest road can either be combined with expansion, or be a strategy of its own, especially when the player has a lot of clay and wood. The purchase of development cards is usually a strategy for the second half of the intermediate phase. In most games there is no direct confrontation of players, apart from a battle for certain building places or the longest road.
The key element in the intermediate phase is trading, because no player has all resources necessary at all times. Players should always be aware what they plan to do next and what their opponents have in mind when they propose a trade. The value of certain resources changes in the different stages of the game. Clay e.g. is needed for roads and settlements and is usually the most valuable resource in the early stages, while it is rather useless in the endgame. Ore on the other hand is

needed for development cards and cities in the end of the game, but not in the early stages. Players should be greedy and rather reject a trade if it is not ideal for them. Every trade is useful for the person that makes the offer, so one should only make the deals for which the expected own profit is larger than the opponent's. On the other hand, if the recipients of a trade offer want too much, it is often better to make port or even 4:1 bank trades. Although being expensive, these trades at least do not help the opponents.
The construction of roads depends on whether the player tries to build the longest road or not. If he does not, a minimum number of roads should be built. It is e.g. possible to connect three settlements with three roads in a star-shaped pattern. Otherwise the player trying to build the longest road should try to connect the road networks of his two initial settlements as early as possible. Being the first to have the longest road makes it easier to keep these extra points or at least makes it harder for the opponents to take over this award. Branches do not contribute to the length of a road, so they should be avoided. Even if it sometimes seems attractive, one should try to avoid building only roads without settlements in between. Another player could destroy this road by building a settlement into one of the holes. The exception to this rule are roads along the coast, where no opponent interference is possible.
When the road connections have been made, settlements should be built as soon as possible. Otherwise opponents could build at the same place. The location of the next settlement must be chosen before the connecting roads are built. Later in the game there are not many choices because of the distance rule, but in the early stages one should always look for the best possible location. High production probability, a much needed type of resource or the presence of a port are the criteria for choosing the place to build. One should however avoid building too many settlements around one good field, because this is very likely to be attacked by the robber.
Ports are often a key element in a winning strategy. They reduce the dependence on trades with opponents. The best port is one where the resource with the highest production probability can be exchanged. Often however they are too far away from the own settlements. In this case one of the 3:1 ports should be secured. 3:1 ports are also useful if there is no single most frequent resource.
For cities there is only one rule: build it at the best possible production place. Again one should avoid to rely on only one good field because of the robber.
The advantages of development cards are often underestimated. Victory point cards e.g. are the cheapest victory points available [Teu00]. Victory point cards need not be revealed until the end of the game. This makes surprise wins possible, if a player has several hidden victory points.
Most of the development cards are knights, so buying many development cards is highly related to having the largest army. Again, having the largest army first is a great advantage, because all other players have to obtain at least one knight more. Knights can also solve problems with the robber. Cards can be played out before the dice are rolled, thus a knight can be used to free an own production field and block someone else's. The stolen resource card is another nice bonus, and it often destroys the plans of an opponent.
Road construction cards are very useful to win a race for a great production place or port against an opponent. The invention card is extremely useful to get the missing resources for a city or settlement, or it can be seen as one free road. Monopolies are great to disturb players that rely on one resource. It should always be used for the most common resource, even if that is not the resource one is looking for. The stolen resources can be traded with the bank, but the negative effect on the opponents is the most important thing.
Only one development card can be played out in a round. Therefore most cards should be played out as soon as possible to make use of the effects of all purchased

cards. If one is not contesting for the largest army, knights should be saved to remove the robber whenever necessary.
The robber can be used very efficiently to disturb the opponents' production. It should always be placed on fields with high production probability (e.g. 6- or 8-fields) that are surrounded by many settlements and / or cities. One must not forget that blocking a city has the same result as blocking two settlements. The robber should never be placed on fields that are adjacent to one's own settlements. Instead it should be tried to block either the most dangerous player (i.e. the player with the most victory points), or a very valuable resource field to create a short-time monopoly.
The production of resources grows in the course of the game. This increases the risk of losing half of the resources when a 7 is thrown. Therefore one should try to use the resources as soon as possible. Sometimes it may even be wise to trade with the bank in order to reduce the number of resources on the hand.

4.2.3 Endgame

The endgame begins when the first player reaches 8 or more victory points. The rules for this player then change drastically, because all other players will collaborate to prevent him from winning. Usually trade with this player is boycotted, and the robber is always used against the leader. The opponents may also try to extend their roads or armies to steal the two extra victory points. These measures are used against *all* players with 8 or more points, not only the actual leader.
In order to win the game, a player must rely on having the needed infrastructure (production and ports) to collect the last victory points alone [Teu00]. Most games end in a very tight endgame, so luck definitely is an important factor.
The best strategy for the endgame is ***disguise***. Victory points can be hidden in several ways, which makes a player appear weaker than he really is. This prevents him from being the primary target of his opponents. Victory point cards are an obvious means of doing so, because they can be held until the end of the game and then played out all at once. Therefore the other players must assume that every development card an opponent holds is probably a victory point card, because all other cards are usually played out as soon as possible. Another way to hide victory points is keeping knight cards until the end of the game, and then use them to build the largest army, which is worth two victory points. Sometimes it pays off not to build one connecting road between two road networks, and then gain the two extra points for the longest road by closing that gap in the endgame. This is a risky strategy, since other players may build their roads into the gap.

4.3 Artificial Intelligence for Settlers of Catan

Settlers of Catan (*SoC*) is a very interesting testbed for artificial intelligence in games. In a sense, SoC lies between classical board games and commercial strategy games. First, Settlers is more complex than the classical board games that are usually studied in AI research. On the other hand, a Settlers program can be implemented without needing to integrate 3D graphics, real-time interaction or other difficult programming tasks that have nothing to do with AI. Because of its popularity as a board game, a strong SoC program is also of commercial interest. Doing AI research for games like Settlers of Catan is therefore one step that could lead to closing the gap between the game industry and the academic world.

Several aspects make SoC an attractive research environment:

1. Settlers involves several *elements of chance*. Luck can make a difference, but in most games only good strategies win.

2. SoC is a strategic game where immediate actions must be coordinated in order to achieve the goals of a *long-term strategy*. Thus *hierarchical AI* is very well suited for this task.

3. There is no single winning strategy. Players must make modifications to their plans according to the dice and their opponents' actions.

4. Settlers usually involves four players (variants with three, five or six player exist), in contrast to classical board games which usually involve only two players. This is an interesting area for research in *adversarial multi-agent environments*.

5. *Opponent modelling* is necessary to take the likely actions of the opponents into account.

6. The virtual agents must develop *negotiation skills* for the resource trades. These skills are also needed in economic simulations.

For Settlers of Catan one commercial program in German (*Catan - Die erste Insel*, ©2002 Navigo), and several freeware (like e.g. *Gnocatan*) and online implementations exist. SoC was also used for scientific research in [Tho02]. The program *JavaSettlers*[2] uses Settlers as a test environment for multi-agent negotiation. The program plays quite strong and is a challenge even for experienced players.
German and world championships in Settlers of Catan were held during the last five years, which shows that SoC is not only a family game, but also involves serious strategic thinking. Weaker players may still be lucky with the dice and even win a few games, but in the end superior strategies and good negotiation skills almost always pay off.

[2] available online at settlers.cs.northwestern.edu

Chapter 5

Learning to Play Settlers of Catan

To demonstrate the potential of machine learning techniques in complex computer games, an agent was implemented that learned to play Settlers of Catan on one standard map using reinforcement learning. Model trees were used to approximate the Q-function in the huge state space of this game. To speed up the learning task, some explicit background knowledge was integrated. A detailed description of the learning environment follows.

5.1 Program Architecture

A Settlers of Catan program was implemented in Java, providing an interface for human and artificial players. Human players can interact via a Java-Swing graphical interface. The virtual agents run in threads and interact with a supervisor-module. This module controls the board, the resources and the development cards. It is also responsible for scheduling the players and the transfer of resources in trades.
The program is strictly modular and allows different agent implementations to participate. The graphical user interface is also exchangeable, which was used for the self-play to save CPU time.

5.2 Modular Agent Architecture

The artificial intelligence of an agent was structured in three layers. The first layer makes strategic decisions about the desired *high-level behaviour*. The behaviours correspond to eigth strategies similar to those in chapter 4.2. All these strategies have a particular goal, which is described below.

5.2.1 Behaviours

1. **Expansion:** Expand to build new settlements.
2. **Longest Road:** Extend the longest road to get the two extra victory-points.
3. **Largest Army:** Build the largest army to receive the two extra victory-points.
4. **City Conversion:** Convert your settlements to cities.
5. **Ports:** Secure a useful port.

6. **Disturb Opponent Roads:** Build roads and settlements to disturb an opponent's longest road.

7. **Disturb Opponent Expansion:** Block the opponents' expansion.

8. **Development Cards:** Collect development cards.

These behaviours were chosen after playing several training matches against human and computer opponents. The strategies of successful players were carefully studied, and these behaviours seemed to be the most frequently used. (In the first experiments a ninth behaviour was used, which was responsible for collecting resources. It was later removed, because the other modules learned to do the same thing.) Of course a successful player cannot rely solely on one of these tactics, but must mix these behaviours carefully.

5.2.2 Actions

After the high-level behaviour is chosen, a *module* corresponding to the behaviour is invoked. This module has 12 low-level actions available that it can execute:

1. **Pass:** Finish the current move.

2. **Build Settlement:** Build a new settlement at the best available position.

3. **Build City:** Build a new city at the best available location.

4. **Extend Longest Road:** The currently longest road in the own network is extended by one piece.

5. **Build Road towards Goal:** A goal (e.g. a new building place for a settlement) is chosen corresponding to the current behaviour, and the shortest path to this goal is calculated. One road of this path is then built.

6. **Buy Development Card:** A new development card is bought.

7. **Play Knight Card:** Play out a knight card, put the robber on a new field and steal a resource from one player.

8. **Play Invention Card:** Play out an invention card and choose two resources that are gained immediately.

9. **Play Monopoly Card:** Play out a monopoly card and select the resource for the monopoly.

10. **Play Road Construction Card:** Play out the road-construction card, which makes the construction of two roads possible.

11. **Bank Trade:** Trade with the bank, using any available ports.

12. **Opponent Trade:** Trade with one of the opponents. Make an offer, wait for the reactions and choose your trading partner or cancel the trade.

Of course not all actions are available at all times. Playing out a monopoly card e.g. is only possible when you possess an active monopoly card that can be played out in this round. The actions correspond to all available actions during the game, but several details are abstracted. It is e.g. not necessary to specify the exact road to build, when action 3 or 4 are chosen. This lowest-level decision is made in a third level.

5.2.3 Primitive Actions

The difference between actions in the second level of the hierarchy and primitive actions is, that the former do not need to specify all the exact parameters of the action. The second-level action "Build a settlement" e.g. leaves the exact position of the settlement open. The corresponding primitive action would be "Build a settlement at position X".
The third level is where most of the a-priori knowledge from the designer is applied. A combination of deterministic algorithms and algorithms that are derived from the learned Q-functions is used. A short description of all algorithms can be found in appendix A.
From the machine learning point of view, the algorithms that develop during the learning process are most interesting. The most important component is the trading module, which is explained in detail in the next section.

5.2.4 Trading

Only players with good negotiation skills can be competitive in the game of Settlers of Catan. Collecting victory points by building settlements and cities requires very specific combinations of resources, and no player can rely on having exactly these resources available when he needs them. Instead he may possess more than necessary of other resources. In situations like this a player should either trade with the bank or with his opponents to obtain the resources necessary to fulfill his plans.
The usefulness of a trade is mainly determined by the resources that are exchanged. Trading in SoC is like a little market game of its own, where the values of the resources are determined by supply and demand. The supply of a resource depends on the production capacities of the players, while demand is regulated by the current board and resource situation.
When trading with his opponents, a player must be aware that they will only make deals that help them reach their own goals. Giving away too much for a heavily needed resource may help the opponents more than the agent himself. On the other hand both players can equally benefit from a trade, since this is the cheapest way for them to obtain the resources they need. A player must therefore carefully estimate the consequences of a trade for both partners.
Trading with the bank is safe, because no other player is involved, but it is expensive. Ports reduce the costs of trading drastically, and make bank trades more attractive, especially later in the endgame, when the opponents boycott all trades. Trading with the bank is like a planning process, since the outcome of the trade is known in advance. The player only has to decide which resources he wants to give away and which one to obtain.
A trading-algorithm for SoC needs to take all these considerations into account. It should be obvious that this is a very complex task, and a lot of background knowledge would be needed to transform all of this into a set of rules. We therefore decided to use learning techniques for this task and thereby demonstrate the power of learning techniques for tasks for which heuristic solutions are difficult to find.
The trading algorithm uses the low-level Q-function to estimate the value of a trade. The value is the improvement in the Q-function of the currently active module in case of executing this exchange. At the same time, for every possible trading partner the change in his high-level value function is calculated. We use the high-level function here, since we do not know anything about the current behaviour of an opponent. No other economic model is necessary, since all the information about the value of the trade is given by the Q-functions. This is a great simplification, because the construction of a market model would require huge background knowledge.

For bank trades, the player can simply execute the most lucrative trade. If however he initiates a trade with his opponents, the agent first proposes the best trade for himself. If no other player accepts, he then selects the next best trade that is slightly better for the opponents. Unsuccessful offers are remembered and not offered again. To limit the number of possible trades, the agent considers only 1:1 and 2:1 trades. The agent is restricted to offer only a limited number of trades, before he must choose another action. The agent also stops trading if only trades remain that would result in a state that is worse than the current situation.
Players that receive an offer from one of their opponents evaluate the trade in the same way. They only agree to trade if the exchange of resources results in a better state for them, and the improvement for the offering player is not too high. The thresholds for the improvements were set after playing several test games.
This very simple trading-algorithm has shown a very good performance, especially for the (deterministic) bank trades. The main advantage is, that no complicated economic models needed to be developed and all the necessary information can be obtained from the learned value functions. This is an good demonstration of the capabilities of learning algorithms for single components inside a larger problem.

5.2.5 Passive Actions

Some actions are not mentioned above, because they do not fit into the reinforcement learning context. The player *must* make these actions in certain situations, without any alternative. Here they are all implemented with some deterministic algorithm:

1. **Throwing Dice:** The player can only choose between rolling immediately, or first playing out a development card. Only knight cards make sense here, because they can drive the robber away from one's own settlements.

2. **Discarding Resources:** When a player has more than seven resources, and a 7 is thrown, then the player must discard half of his resources. The algorithm selects the resources that result in the smallest decrease in the value function.

3. **Responding to Trades:** If an agent cannot afford a trade, he rejects it. Otherwise the agent uses the trading algorithm described in 5.2.4 to evaluate the trade. If the trade is good for himself and not too good for the opponent, he accepts. Otherwise he calculates possible counter offers that the other player could be interested in and offers them. If no good alternative is available, he rejects the trade.

4. **Placing the Robber:** First the player chooses a target for the robber, which usually is the leading player. Then the player evaluates all fields that produce for the target player and selects the field, where the greatest damage can be done.

5.2.6 Initial Buildings

The selection of the first two settlements and roads is also an exception to the modular architecture. The agents evaluate all possible building places using the high-level value function. Then they choose the location that yields the highest possible value. The same thing is done with the first two roads.
Since the placement of the first two settlements is extremely important, a-priori knowledge was used to restrict the set of possible starting positions. Only settlements with either three producers or two producing fields and a port are considered. Locations that have a very low total production probability are also avoided.

5.3 Reinforcement Learning Framework

5.3.1 State Space

Since Settlers of Catan has far too many possible states to store values for each of them, a parametric description of states was designed. One state is represented by 216 *features*, which describe the current board situation, the state of the learner himself and of all his opponents. Most of the features were calculated as high-level concepts that describe the actual situation. E.g. no information about individual roads was used, but features such as the number of connection components or the length of the longest path were included. This was supposed to speed up the learning, because otherwise the algorithm would have had to discover these important features from experience. A more detailed description of the features can be found in appendix B.

There were three kinds of features included in the model: binary, discrete and real valued features. Since it is known that feature normalization can further facilitate learning, all features were transformed into the interval [0, 1], and one feature was transformed into a set of several variables. This procedure, which was used in a similar way in the famous TD-Gammon example [Tes95], resembles *tile-coding* for linear approximators (see [Sut98]) and makes it easier to discover non-linear dependencies between variables.

To give an example, consider the feature *possession of the resource wheat.* Coding this feature with only one discrete value in the range from 0 to infinity would make it difficult to discover when exactly the possession of wheat is important. If a player e.g. has 9 victory points and possesses three ore and one wheat, he would need exactly one more wheat to build a city and win the game. Therefore we coded the possession of wheat with three features: the first one is zero when no wheat is available, and 1 otherwise. The second feature takes on the value 1 if there are at least two pieces of wheat available. The third feature is greater than zero if more than three pieces of wheat are available. The value is linearly interpolated between 3 and 5 using the formula $v = \frac{n-2}{3}$, where n is the number of wheat available. Therefore having four pieces of wheat would result in a feature vector of $(1, 1, 2/3)$. (Feature values above 1.0 were permitted, e.g. if a player has more than 5 wheat this would yield a feature vector of $(1, 1, 4/3)$. An upper bound was however set at 2.0.) These multiple resolutions in the coding of features are guided by background knowledge about the influence of certain feature-values. For other features, between two and five "quasi-binary" variables were used. Binary features of course were not transformed.

5.3.2 Hierarchical Learning

Two kinds of Q-functions were learned: one for the high-level behaviour selector, and one for the second-level action selector. In every step, the learner first chooses the behaviour, and then the action is suggested by the invoked module. Later we used a slightly different approach: after selecting a behaviour, the corresponding module is executed until either the sub-goal is reached or the agent passes. In this approach only the low-level actions need to be selected at every time step.

Rewards were calculated independently. For the high-level layer, the rewards are defined as

$$r_t^{HL}(s_t, a_t) = \begin{cases} 1 & \text{if the game is won} \\ \frac{VP_{own} - VP_{min}}{VP_{max} - VP_{min}} & \text{if another player wins the game} \end{cases} \tag{5.1}$$

where VP_{own} is one's own number of victory points, and VP_{min} and VP_{max} are the minimum and maximum numbers of victory points among all players. This

interpolated reward seemed more appropriate for this task than giving zero reward to all non-winning players. The endgame is often very tight, and luck instead of strategy may be the decisive factor. Therefore players should not be punished too much for finishing second or third.

Rewards for the lower level were defined as 1 if the task of the module was achieved, and zero otherwise. For some modules, reaching intermediate goals was rewarded with 0.5. The definition of reaching a module's goal can easily be understood from the descriptions of the behaviours. The goal of the city-conversion behaviour e.g. is building a new city. In later experiments the passing action was punished by a numerical reward of -0.1. The intention behind this was that players usually only pass when they have no other option, that is if they cannot build anything and no trades are possible. Still the punishment is small enough to recognize the value of passing, e.g. for the production of new resources.
It is important to see that the reward for the lower-level modules is not influenced in any way by the reward for the higher-level behaviour selection. Therefore the individual modules can be trained to achieve only their specific task, without long-term thinking about winning the game. This is the responsibility of the high-level layer, which instead only learns to switch between the modules.
One episode for a low-level module starts when it is invoked, and ends when either the low-level sub-goal is achieved, or a different module is selected by the high-level component. Therefore the modules are trained to achieve their goal as quickly as possible, assuming that the module stays active until this moment.

Learning state-action values (Q-values) was preferred here to learning only state values (V-values). Of course this dramatically increases the size of the problem, because values for every state and each of the 96 behaviour-action combinations must be learned. However, if we only have state values, we must be able to predict the outcome of an action in a state, and this is impossible here for the pass action. After passing, the three opponents may perform their actions, and there are too many possibilities of moves and dice results to consider to calculate some expected minimax-value. TD-Gammon [Tes95] e.g. used this approach successfully, but in Backgammon the branching factor of the game tree is much smaller than in Settlers of Catan.

5.3.3 References to Related Work

The modular architecture of the agent described above was inspired by hierarchical AI techniques. The behaviour selection is the strategical long-term planner, while the second layer makes tactical decisions and the third layer is responsible for the execution of primitive actions. The division also made it easier to devise heuristic algorithms for the lowest level, because only short-term goals needed to be taken into account.

In the language of hierarchical reinforcement learning, this architecture resembles **module-based RL** and **feudal RL**.
Rewarding the modules only for completing the requested task, independently of reaching a higher-level goal is an idea from feudal learning. This architecture enables the learner to simultaneously develop high- and low-level policies. Of course, learning at the higher level can only start when the lower level modules have recognized how they can achieve their sub-goals.
The high-level module learns the *switching functions* between the different behaviours, similar to the module-based approach. Modules used in robotics usually run until completion before another module is activated. We compared this method

to another approach, which makes high-level decisions after every single low-level action.
The reason for trying out this approach was, that in Settlers of Catan, immediate changes in the state space can occur after one single action. This often makes it necessary to re-think the selection of the high-level behaviour after only one move. The classical example is the *pass* action. When a player passes, he usually plans to receive new resources which he needs for a building or the purchase of a card. After passing however, all the opponents can make their moves, which often dramatically changes the situation. Building places, or possible road extensions can suddenly be blocked. Therefore the player needs to rethink his strategy after executing one pass-move. Even for the strict module-based approach, in which modules always run until completion, we decided to choose a new high-level behaviour after every passing action.

Learning to play games through self-play was first studied by Samuel [Sam59]. TD-Gammon by Tesauro [Tes95] is a more recent example where this method was used very successfully. These papers have clearly inspired this thesis. Nevertheless there are great differences between the games studied in these papers and Settlers of Catan. Most important are the number of players, the dimension of the state space and the number of different actions that are available in every situation.

5.3.4 Function Approximation

Due to the huge state space in Settlers of Catan, function approximation techniques have to be applied. Model trees seemed to be the method of choice, even though this is a completely novel approach for learning via self-play in complex game domains. Earlier results for other scenarios (see [Sri00], [WanX99]) have shown that model trees are well-suited for large state-space problems. It was also expected that model trees need less training time than other approximators. One of their most important properties, which makes them superior to e.g. artificial neural networks for this task, is that they can handle the discontinuities in the value function, which often only depend on a small set of variables.

Example 5.1: Consider a situation where two players A and B both have three knights and the other players have none. Player A was the first to build his third knight, therefore he has two victory points due to the largest army. Assume that both players have 8 victory points. If instead B had built the third knight before, then he would have the largest army and B would have won the game with 10 victory points. So he has to build at least one knight more. Therefore only one binary variable (having the largest army or not) makes a big difference in the value function.

Example 5.2: Now consider another situation at the beginning of a completely different game. Player A has not yet managed to build roads, settlements or cities, but has bought three development cards which all were knights. He therefore has the largest army and 4 victory points. In the meantime, the other players have built new settlements or cities, such that all players now have four victory points. Player A is however in the worst situation of them all, because he has less production. Having the largest army is irrelevant.

These two examples show, that e.g. linear approximators which always assign the same weight to a feature can never express all characteristics of the game. In neural networks, one feature usually does not have enough influence to cause such fluctuations. Model trees however could use the attribute as a splitting criteria or in the linear model for those situations where it is important and ignore it in other cases.

5.4 Training the Model Trees

The state-action value-function was learned through self-play of virtual agents against each other. The occurring states, the selected behaviours and low-level actions, and the high- and low-level rewards were recorded in a log-file. This file was used for offline learning of the Q-function for both the high-level module and the individual behaviours.

High-Level Learning

One episode consists of the recordings (state, behaviour and reward) for one player in one game. All rewards are zero, except for the last state in which a reward according to the achieved victory points is received. This reward is propagated to all predecessor state-action pairs, using a offline SARSA(λ)-update with a discount factor of $\gamma = 0.95$ and $\lambda = 0.7$. Therefore the value of a state-action pair decreases exponentially the further it is away from the final state, where the only positive reward is received. This procedure is repeated until convergence is reached.
The training data is split into different training sets for every behaviour. For older training data, one update sweep is performed, that changes the Q-values for every state-behaviour pair with one Q-learning update. The maximum of all state-behaviour values in the successor state is determined using the last learned model tree. This update partially removes the effect of sub-optimal policies in older training runs. On the other hand, it allows older training data to be used for the construction of new trees. This is important, since older training examples usually have lower Q-values because they were obtained from a sub-optimal policy. Without these negative training examples, the predictions of the model tree would be too high for sub-optimal state-action pairs.
The combination of old and new training sets was used to construct a new model tree, using the algorithm described below.

Low-Level Learning

The individual modules were trained exactly like the high-level switching function. The difference lies in the definition of an episode: a new low-level episode begins with the activation of the module, and terminates either when the sub-goal of the module is achieved, or the switching function decides to activate another module.
The SARSA(λ) algorithm is used (with the same parameters) to train within the episodes. If an episode ends prematurely with the invocation of another module, the last state of that episode needs a special update rule. Let m be the module that is active at time t, $Q_m(s,a)$ be the state-action value for module m of action a in state s, and $r_m(s,a)$ be the low-level reward for executing action a in state s. Then the update rule for the terminal state of a module m is

$$Q_m(s_t, a_t) \leftarrow (1-\alpha)\ Q_m(s_t, a_t) + \alpha \left(r_m(s_t, a_t) + \gamma\ \max_a Q_m(s_{t+1}, a) \right) \tag{5.2}$$

The maximum is calculated using the last learned model tree. Thus the last state of the episode is updated as if the currently best action of the same module was executed in the successor state.

Passing actions cannot be evaluated before control returns to the player. Therefore the next state after passing in an episode is the state when the player can

throw the dice. The reward for the passing action is also calculated at that time. The states when the opponents make their moves are not included in the episode, because the player has no possibility to act. The same applies to the evaluation of building the first two roads, because after the construction the player must immediately pass.

Resampling of Old Training Sets

The training games produced a huge number of training examples (approx. 1 GB for 1000 games). Since we want the policy to improve from the gathered experience but cannot use online-learning because of the nature of model-trees, we need to include the old training examples even for later training runs. If we only used the latest training results, we would rate negative examples, which had been obtained by following a bad policy in the early training runs, too high.
For some actions and behaviours however, this yielded training sets that were much too large to enable efficient training. This made the running time of the model-tree construction algorithms unacceptably long. It also produced trees that were much too large, and tended to overfit the training data. The third effect was that the influence of new and more accurate training examples decreased, when there were too many old examples with inaccurate predictions.
Therefore the old training data was reduced before training. We used a simple *resampling* technique, that randomly picked a certain percentage of training cases from the available data. The resampling percentage depended on the size of the training set. Some actions like e.g. the *pass*-action, were selected far too often in the early training games, simply because they are almost always available. Other actions, like *play-monopoly-card* are rarely used, because they can only be executed in a small number of states. Therefore the amount of data for frequent actions was reduced significantly (by up to 90 %), while for other sets only a slight reduction (less than 50 %) was performed. We used different resampling rates, depending on the age of the data. Older training sets were reduced more drastically than more recent results.

5.4.1 Construction of the Model Trees

For this learning task, a modification of Quinlan's **M5** algorithm [Qui92] was used. We used the Java-implementation of M5' [WanY97] in the WEKA[1] machine learning package, and modified it according to our needs.
Since the training games yielded a huge amount of data, not all the training examples could be stored in memory. Therefore the algorithm was modified to sequentially read the training examples from disk, and store only the indices of the training examples in the swap-file that belong to one set. This is a reduction of memory requirements by a factor of 500. The main disadvantage of course is the required time for disk IO, because the training examples always have to be re-read from disk. The problem could be overcome by storing a cache of the 20,000 least-recently used training examples in memory. This turned out to be a great improvement, since in the construction of model tree branches we always operate on the same training examples until the branch is finished. Therefore data must only be re-read when a new large branch is constructed.

Splitting

For splitting it was only necessary to incrementally calculate the splitting-information for all possible features to determine the best splitting criterion. Since we used

[1] available at www.cs.waikato.ac.nz/ ml

"quasi-binary" features, the splitting value was always chosen to separate examples that had feature value 0 and those that had a value greater zero. In a second sweep through the training data, the indices of the training examples for the two sets resulting from the split were calculated. This procedure was recursively repeated until only leaves were remaining. The prediction of leaves was initialized to the mean target value of all contained training examples.
The minimum number of examples in a split-node was set according to the size of the training set. We found that the size of the grown tree is very sensitive to this parameter. The values were chosen after experimentation. Unfortunately there is no formula available to set good values for this parameter.

Pruning

Pruning was started after the tree was fully grown by the splitting procedure. A sub-tree is pruned when the RMS error of a linear model in the node, multiplied by a pruning-factor is lower than the error of the sub-tree. In contrast to the original calculation of the pruning factor in M5 (see equation 3.9 in 3.4), the WEKA algorithm introduced a pruning constant ψ to control the level of pruning. The new pruning factor is calculated as

$$\frac{n + \psi\, v}{n - v} \tag{5.3}$$

where n is the number of training examples in the node, and v is the number of parameters in the linear model. The higher ψ, the more sub-trees are pruned. The default value in WEKA is $\psi = 2.0$, but for this task we experimented to set higher pruning-constants for problems with huge training sets (more than 100,000 training examples). To further reduce the size of the resulting trees, we experimented with a *squared* pruning factor. This modification had a strong effect on the size of the trees and yielded very promising results.

Calculation of Linear Models

The variables for the calculation of the multivariate linear regression model in the node are the splitting attribute in the node, the splitting attributes of the two children, and the variables in the linear models of the children. The model is later simplified by sequentially removing the least influential variable from the model. Even if this increases the prediction error, it reduces the pruning factor 5.3 by reducing v. The removed variables are **not** used for later models in higher nodes. This yields a huge advantage in running time that outweighs the possibly improved prediction accuracy of models with more variables. To save further running time, pruning was skipped for nodes that included a large number of training examples (i.e. more than 20,000). Subtrees of these nodes are hardly ever pruned, because the number of training examples is so high that the number of parameters becomes irrelevant.

Smoothing

We decided not to smooth the prediction of the model tree, because the ability to represent sharp discontinuities in the value function was one of the main reasons to choose model trees. Smoothing the function could destroy this effect and was therefore not used for this task.

5.5 Training Procedure

5.5.1 General Training Procedure

As a first step, 1000 games in random self-play were played. The states, actions and rewards were recorded. Since the high-level behaviour selection was pure random, the low-level actions were evaluated for all available behaviours in order to gain more training examples. For the high-level module the first data was only used to calculate a *state value* (V-value) instead of a state-action value (Q-value), because there was no difference between the modules in random play. The V-value could be used for goal selection and opponent modelling.
20 sweeps of SARSA(λ) learning were made to update the Q-values to their final values. The obtained data was then used to build 96 single model trees for every action in every module.
The resulting model trees were then used to play the next 1000 training matches following a policy derived from the trees. Again, 20 sweeps were made through the new data, and a Q-learning update was performed for the old training data to make them more consistent with the followed policy. The data was combined to build new model trees.
This sequence of playing training games and learning model trees was then repeated. To speed up learning and produce smaller trees, the oldest training data were re-sampled to use only a smaller percentage of the data for new training sets. This was especially useful for actions that are often available like passing or trading, and which were therefore used too often in random play. Also, the learning temperature was slightly lowered to decrease the random factor in the policies.

5.5.2 Alternative Architectures

The above training procedure was used for four different architectures:

1. The first approach tried to learn both the high-level switching function and the behaviours via reinforcement learning. The agent could change its high-level behaviour after every low-level action. Passing was not assigned a negative reward.

2. The second approach also learned in both levels of hierarchy. In contrast to the first method, new high-level behaviours are only selected after the termination of the current module or after passing. We also introduced two new high-level rewards of 0.1 for reaching 3 or 8 victory points respectively. Passing was punished by a reward of -0.1.

3. The third approach is based upon the second method, but uses a heuristic strategy for high-level decisions. The modules in the second level still learn their policies from experience. The agent is also forced to execute actions that immediately reach the sub-goal of the module.

4. In the fourth approach we use the heuristic strategy and the modules that were learned by the third method to learn a high-level strategy. The heuristic strategy guides the agent in its training games and thereby creates better training examples.

The second and third approach were designed to eliminate the weaknesses of their predecessors. The intention behind the fourth approach was to investigate, whether the agent can learn to imitate the heuristic high-level strategy.
We decided to use heuristics only for the high-level switching function, because

the low-level Q-function is needed for the trading algorithm, which is one of the most interesting results of this paper. The strategies that were used are derived from those explained in chapter 4.2. The behaviour-selection algorithm is a simple combination of IF-THEN-rules which take only the features that define the current state as input for the decisions.
The results of all four approaches are presented in the next chapter.

Chapter 6

Results

In this chapter we describe the process of training an agent that learns to play Settlers of Catan. We describe the problems that arose with certain approaches, and how these problems could be overcome. The policies that were learned at different stages of the learning process are evaluated. We find that in spite of the complexity of the problem, an agent that uses a combination of a-priori knowledge and learning from experience can reach a level close to that of human players.

6.1 Evaluation of the Policies

The learning algorithm uses model trees, which have to be trained offline. Therefore it was necessary to first play a number of training games and then construct new approximations for the Q-functions. After every learning stage, the learned policies were evaluated by playing against their predecessors, against a random player and against a human opponent.

The evaluation of policies at different learning stages shows whether learning is responsible for an improvement or not. We hoped to find that the performance of policies increases with the number of training examples. For the evaluation games, two players of each level played against each other. We measured the number of victories for both sides and the average number of victory points. Victory points are a good measurement for the performance of a player. Good players may lose some games due to bad luck, but still have a high number of victory points.

We also measured the average points of the best and worst players of each level. The average points of the less successful player indicate, whether the policy is able to perform relatively well, even in case of bad luck or superior strategies of the opponents. Experience has shown that in games between humans the player who finishes last usually scores between 3 and 6 victory points. The average points of the better player indicate whether at least one of the agents of the same level was able to compete for victory. If the best player of one level scores 8 or more points on average, we can say that he was close to winning in almost all games. Games with human opponents have shown that this is probably the most important performance measure, because usually at most two players in a game can compete for victory.

To evaluate the real playing strength of the policies we played several test games against human players. In these evaluation games we let three agents of the same level play against one human. We measured the number of victories, the average number of victory points and the average points of the best and worst player. It was not possible to play a large number of test games, because one game lasts between 15 and 20 minutes. We played 5 games against weaker opponents, and 10 against the strongest. Still it was possible to notice big differences in the playing strengths

of different agents.

In the next sections we describe the development process of the agents' architectures. We started with a very flexible approach that relied solely on learning and improved the performance steadily by making changes and adaptations in the learning architecture.

6.2 Feudal Approach

The first method that was tried out was inspired by *feudal learning* [Day93] and learned at both levels of hierarchy simultaneously. The agent can choose his high-level behaviour at every time state and does not need to wait until the module has finished. The rewards for the high-level were set to zero for all non-terminal actions. There are no negative rewards for passing in the second level.

Evaluation

We first evaluated the learned policies in games against a random player. We hoped that the percentage of victories and the average number of victory points would rise with an increasing number of training games. Unfortunately we found the contrary result, namely a decrease in performance for the first 5 policies that were learned from the experience of up to 5000 training games. This was a clear indication of some error in the learning algorithm. One of the main reasons was that the model trees grew far too large (more than 4000 leaves), which lead to strong *overfitting*. We therefore increased the pruning constants (see 5.4.1) and the minimum number of training examples in the tree nodes for the next learning process. The result was a policy that played better than its direct predecessors, but still was no real improvement over the first learned policy. We also found that overfitting occurred again, when we added new training examples, even if the pruning constants were further increased. Figures 6.1(a) and 6.1(b) show the results of the evaluation.

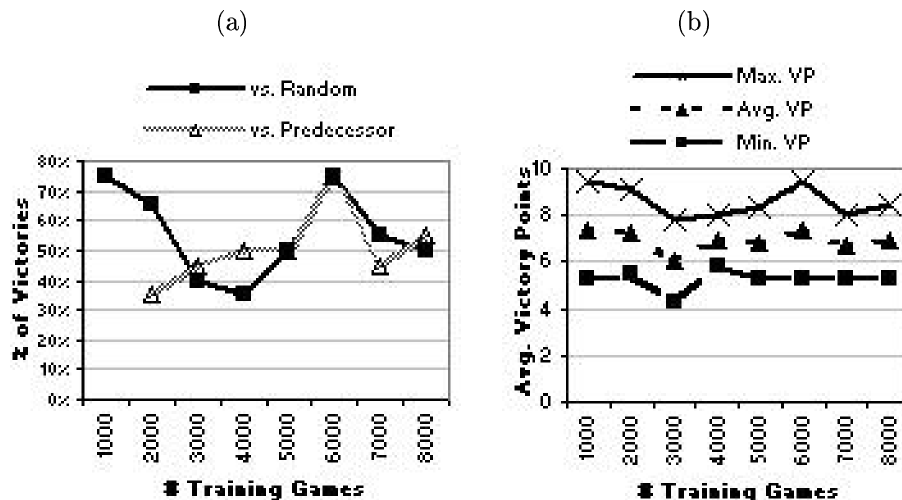


Figure 6.1: (a) The percentage of victories of the feudal policies in games against a random player and their predecessor policies. (b) Average victory points in games against random players.

These figures reveal the decreasing performance for the first 5000 training games. The increased pruning factors are responsible for the improvement of the policy at 6000 training games.

A closer look at the policies that were followed in the training games revealed another big problem: there were far too many switches between the modules in the high-level. Often e.g. a trade was executed that made the reaching of a sub-goal possible, but instead another module was invoked which could not do anything with the available resources. We found that the approximated high-level Q-values of different behaviours were very similar, so the high-level strategy was almost random. The agents did not discover that it makes sense to execute one behaviour until termination. The inaccuracy in the high-level function also made it difficult to discover good starting positions, because their selection depends on the high-level value function.
The third problem was that the agents passed far too often. They were even passing when they had enough resources to build settlements or cities. Possessing too many resources, especially more than 7 resources, is dangerous. The probability that a 7 is thrown before their next turn is larger than 50 %, and so it is likely that they will lose half of their resources. The reason for this behaviour was that the passing action was overrated, because all the opponents in the training matches were also passing too often. In a game between strong players passing is usually the worst possible option, because it gives the opponents the chance to improve their situation.
We found that with the exception of the passing action, the approximation of the state-action value function in the modules was quite good. Even though there were some significant inaccuracies in the approximation in the tree leaves, most model trees used sensible splitting criteria in the nodes.
To test the playing strength of some of the policies trained with this method, we played several test games against a human player. We tested the performance of the random policy, and the policies that were calculated after playing 1000, 3000, 6000 and 8000 training matches respectively. These policies were selected because they were particularly strong or weak in the games against other learned policies. Figure 6.2 shows the results of the evaluation.

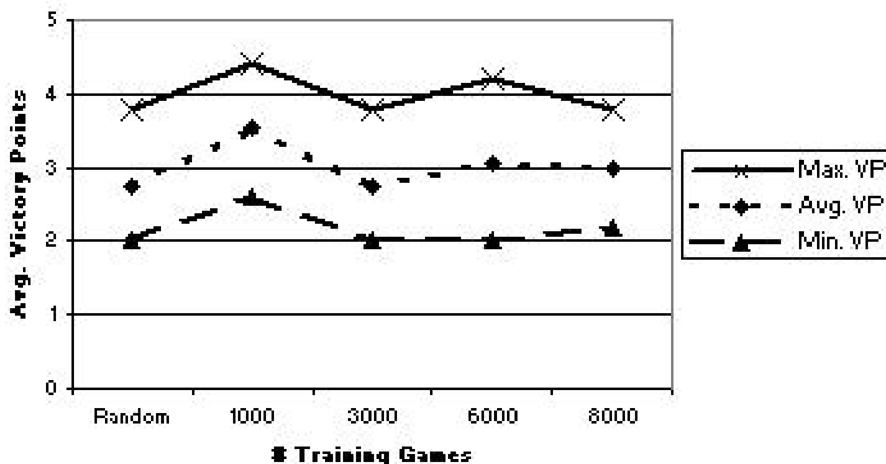


Figure 6.2: Average victory points of the feudal policies scored against human opponents.

The virtual agents never came close to winning one of the games, and often did not even score one additional victory point. No policy scored more than 6 victory points in any of the evaluation games. Frequent passing and uncoordinated actions because of the almost random high-level strategy selection were the main reasons for this rather weak performance. The trading behaviour of the agents was also not very clever, which can be related to the frequent changes in the high-level strategy. A positive sign is that on average the learned policies play stronger than a random player. We can also see that policies that played particularly strong or weak against other learned policies show the same behaviour against a human player. The strongest policies seem to be those that were obtained after playing 1000 or 6000 training games respectively. We call these policies *Feudal 0* and *Feudal 5* respectively and use them later for an evaluation of policies that were constructed by following a different approach.

Conclusion

The results of this approach were not satisfactory. Problems with overfitting and too frequent passing destroyed a possible learning progress. The main problem however seemed to be the selection of the high-level strategy in every time step. This makes it very hard for the agent to discover that it is better to follow one strategy, once it has been selected. In the next section we describe an approach that tries to eliminate all of these problems.

6.3 Module-Based Approach

One of the biggest problems of the feudal approach was that the agent switched too often between high-level behaviours. Therefore we decided to use a method that is similar to *module-based reinforcement learning*, [Kal98] in which modules are always executed until they reach their sub-goal. High-level decisions are also made after passing, because then the situation is usually completely different from the situation encountered in the last turn.
The second big problem of the first approach was that the agents were passing far too often. We therefore decided to punish passing by assigning a reward of -0.1 to every passing action.
The third modification concerned the high-level rewards. The feudal method was not able to discover good locations for the initial settlements, which is one of the most important decisions of the whole game. We therefore introduced two new rewards for reaching 3 or 8 victory points for the first time. The former is responsible for making quick expansion more attractive, which is only possible with good starting positions. The second reward was intended to make the agents discover the importance of entering the endgame. Once a player has 8 or more victory points he can always win the game by either building the largest army or the longest road.
We also decided to remove the ninth behaviour "collect resources", because it turned out that this behaviour was not really necessary.
To avoid the problems with overfitting of the first approach, a *squared* pruning factor was used for the construction of the model trees. This reduced the size of the trees drastically, and yielded much better results.

Evaluation

3000 training games were played, and three different policies were constructed and evaluated. We first played test matches against random players, against agents following the feudal approach, and against the module-based players at different

learning stages.
Table 6.1 shows the percentage of victories of the different module-based players against the random player and against each other:

	Player B			
Player A	Random	Module 0	Module 1	Module 2
Random	-	45%	35%	53%
Module 0 (1000 games)	55%	-	35%	67%
Module 1 (2000 games)	65%	65%	-	76%
Module 2 (3000 games)	47%	33%	24%	-

Table 6.1: Performance of module-based agents. The entries give the percentage of victories of player A (row) against player B (column).

We found that the *Module 1* player, which was trained in 2000 games, performed particularly well. We also found that the size of the model trees was much smaller than before, which was the effect of the squared pruning factor. Unfortunately, the parameters were still not optimal, which caused severe overfitting in the *Module 2* policies. The number of leaves in these trees is almost twice as larger as in the *Module 1* trees. We believe that this is the main reason for the very poor performance of the *Module 2* policy.

The next step was the evaluation of the module-based agents in games against players that used the feudal approach. The *Feudal 0* and *Feudal 5* policies were used to test the playing strength of the module-based agents. Figure 6.3 shows the results of this evaluation.

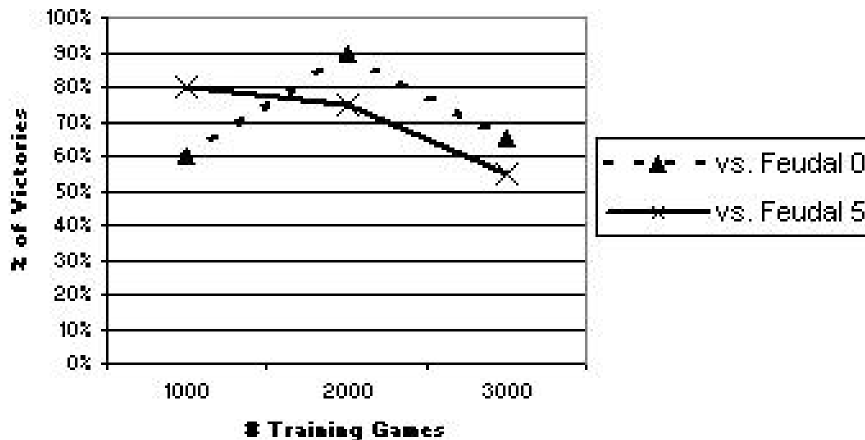


Figure 6.3: Percentage of victories of the module-based policies in games against the *Feudal 0* and *Feudal 5* policies.

We found that the first two module-based policies win between 60 and 90 percent of all games against the strongest players of the former approach. Even the rather poor *Module 2* policy, which suffered from overfitting, was able to win more than 50% of all test games on average. This indicates that the modifications in the learning architecture have successfully raised the performance of the system.
The evaluation of the policies in games against humans was supposed to reveal the real playing strength and possible weaknesses of this approach. Figure 6.4 shows

the average victory points of the three policies and, for comparison, of the random and feudal policies, in the evaluation games.

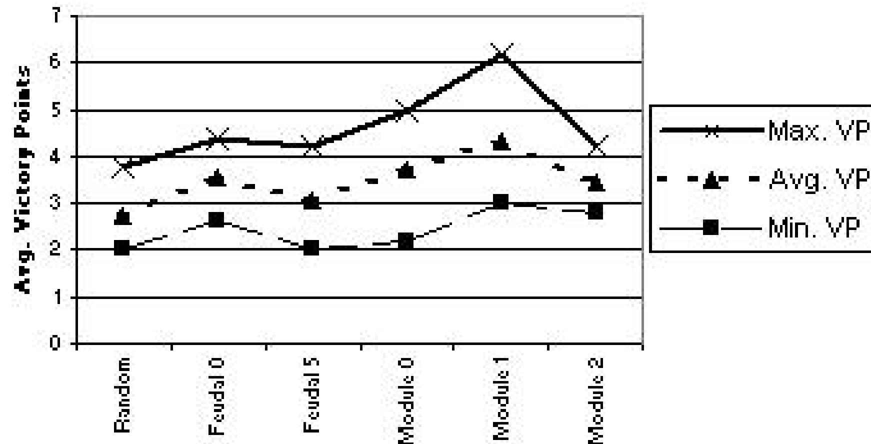


Figure 6.4: Average victory points of the feudal and module-based policies scored against human opponents.

Still none of the virtual players were able to win one of the test games, but an improvement over the *Feudal 0* and *Feudal 5* policies could be noticed. The *Module 1* policy was particularly strong, which was expected from the results of the games against other learned policies. There were several games in which the agents reached 7 or even 8 victory points, and the general impression was that they behaved more cleverly than their predecessors. The agents were able to reach more of their sub-goals, like building the longest road or the largest army, which is an effect of the module-based approach. They also proposed better trades and slightly reduced their frequency of passing.
Unfortunately we found that the high-level switching function was still more or less random. Therefore modules were often invoked that had no chance of reaching their sub-goals and thus had to pass. On the other hand the approximation of the Q-functions in the modules was found to be quite good. These functions are responsible for the improved negotiation skills, which were obvious in the games against human players.
There was however a problem with some inaccuracies in the approximations, which lead to very high coefficients in some linear models that were constructed from only a few training examples. As a result there were often Q-values higher than 1 (which is the maximum reward for reaching a sub-goal), even if the corresponding action did not lead to the sub-goal. Thus the agents often made clever trades to gain all the resources necessary for reaching a sub-goal, but then suddenly passed or made other trades.

Conclusion

The module-based approach yielded policies that were clearly better than the policies constructed with the feudal approach, but still not competitive in games against humans. Their strength came from the improved negotiation skills and their improved ability to actually reach the sub-goals of the modules. The low-level policies that were followed after a module had been invoked were pretty good, but the high-level strategy was still poor. The other major problems were once again overfitting

and too frequent passing. The effect of the improved policies for the modules was also disturbed by errors in some of the linear models. This was the reason for wrong decisions in situations where the choice of the action is absolutely clear for humans. To eliminate these errors, we tried to use more background knowledge for those parts that did not work well, and use the learned policies for the modules, where the results were quite promising.

6.4 Heuristic High-Level Strategies

Even though the modifications of the module-based method increased the performance of the policies significantly, the results against human players showed that the learned policy was good, but far from optimal. An interpretation of the model trees that were constructed from the training experience showed, that the policies of the modules were reasonable, but the high-level switching function still made almost random decisions.
We therefore developed a heuristic high-level strategy, which was derived from the tactics described in chapter 4.2. The usefulness of each behaviour in a specific situation was classified into one of five categories of usefulness by simple IF-THEN - rules. The algorithm then selected the highest-rated behaviour. Always using the same high-level strategy was supposed to make the effect of learning in the low-level modules visible. The most important thing about learning the modules is that the negotiation skills highly depend on the quality of the low-level Q-values. Therefore even a perfect heuristic high-level switching function would not be able to play competently without good learning results for the second level of the hierarchy.
The second big problem of the module-based approach was that small inaccuracies in the approximation sometimes rated unsuited actions too high. This problem was particularly annoying at the end of a module, when the sub-goal could be reached with one more action. We therefore devised a deterministic algorithm that selected low-level actions in those situations where a sub-goal can be reached in one more step. In all other situations, the actions were still selected according to the Q-function.
We hoped that the guidance of the heuristic strategy would produce better training examples, because the modules are only invoked in situations where their use makes sense. We also tried to overcome the severe overfitting problems by increasing the pruning factors once more, using the experience values from the first two approaches.

Evaluation

We let the computer play 6000 training games and evaluated 5 policies and a heuristically guided random player in games against the agents described in the previous sections. The new random player (called *Heuristic Random*) uses the fixed high-level strategy and the deterministic selection of actions in obvious situations, but otherwise makes random low-level actions and trades. This agent is therefore stronger than a pure random player, and the performance of policies against this heuristic random player directly reveals the influence of good low-level policies. If a heuristic agent plays significantly better than the heuristic random player, this is the effect of improved low-level modules.
We therefore first evaluated the policies in games against the heuristic random player and against their direct predecessors. Figures 6.5(a) and 6.5(b) show the results.

We found that these policies (we will later call them *Heuristic 0 - 4*) win between 60 and 90 percent of all games against the heuristic random player. This

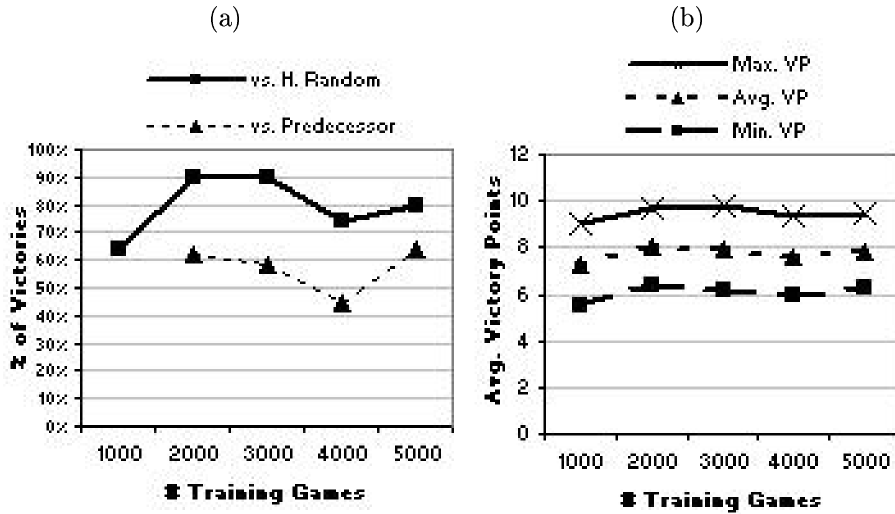


Figure 6.5: (a) The percentage of victories of the heuristic policies in games against a heuristic random player and their predecessor policies. (b) Average victory points in games against heuristic random players.

result clearly shows the importance of good low-level strategies, since all players follow the same high-level policy. All policies, with the exception of the fourth, are improvements over their direct predecessors and win approximately 60% of all games against them. The reason for the weaker performance of the policy *Heuristic 3* (using 4000 training games) apparently was again an increased size of the model trees, due to a bad choice of pruning parameters. The parameters were therefore increased for the following training procedure, which yielded a much better policy (called *Heuristic 4*). This policy (using 5000 training games) was also an improvement over the *Heuristic 2* policy (3000 training games) and won 68% of all games against it.
The evaluation of the average victory points revealed that the heuristic players score at at least 6 victory points on average, which is a very high value. This shows that they are always competitive against the heuristic random player, even in case of bad luck.

The next step was an evaluation of these strategies against players that used the feudal and the module-based approach. We tested the performance of the heuristic policies and the heuristic random player in games against a pure random player, the *Feudal 0, Feudal 5* and *Module 1* policies, which were the strongest players of the previous approaches. Figure 6.6 shows the percentage of victories of the heuristic policies.

We can see that even the heuristic random player is stronger than the *Feudal 0* and *Feudal 5* policies, due to the built in a-priori knowledge for the high-level behaviour and the execution of obvious actions. It is also clear that the heuristic players who use learning in the low-level modules are even better. These policies win nearly 100% of their matches against the pure random player and between 90 and 100% against the players using the feudal approach. The *Module 1* policy is also inferior to the heuristic policies, but at least is better than the heuristic random player and the feudal policies.
This shows that the combination of background knowledge for the high-level behaviour and reinforcement learning for the low-level yields policies that can out-

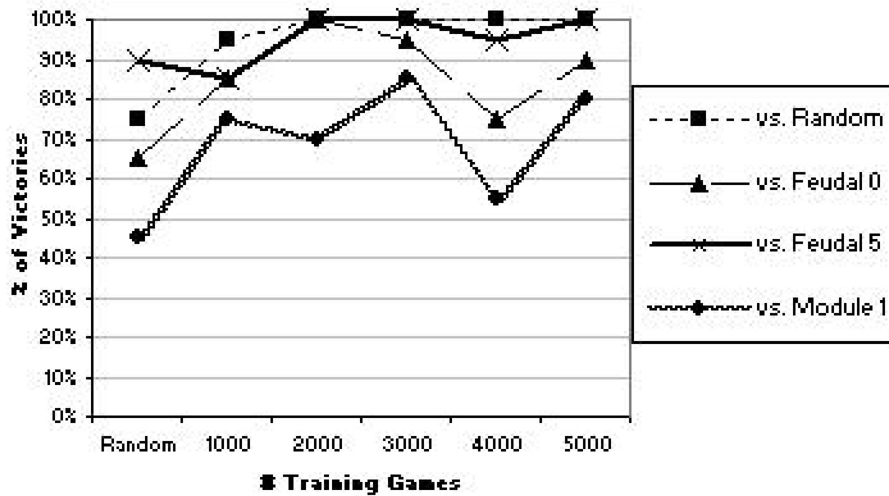


Figure 6.6: Percentage of victories of the heuristic policies in games against pure random players and the *Feudal 0*, *Feudal 5* and *Module 1* policies.

perform all other policies that were obtained from pure learning approaches. Using clever heuristics for the high-level strategy is however not enough. Learning is still a very important part of the system, which becomes obvious when the differences in performance between the heuristic random player and the players who use learning for their low-level modules are investigated. Against the good *Module 1* policy, the heuristic random player loses more than 55% of its games, but the other heuristic policies win more than 70%.
The average victory point statistics of the evaluation games was almost identical for all four opponents. The better heuristic player scored almost 10 victory points on average, and the worse scored 6.

These results are very promising and we expected to see a good performance of these policies in games against human opponents. 10 evaluation games for each of the six policies (*Heuristic Random* and *Heuristic 0 - 4*) were played, and the average victory points were measured. Figure 6.7 shows the results of this evaluation.

The figure shows an increasing playing strength, with a sudden drop at 3000 training examples. We are not sure about the reasons for this drop, since this policy performed very well against other learned policies. In general however, these results are much better than those of the previous approaches. The best policy (*Heuristic 4*, 5000 training games) on average scored between 4 and 8 points, which is comparable to the playing strength of the "Easy" or "Medium" levels of current commercial and non-commercial Settlers of Catan implementations.
All of the heuristic policies (including the heuristic random player) managed to win at least one game against the human player. The *Heuristic 3* and *Heuristic 4* policies even won two out of ten games. Most of the games ended in a tight endgame between the human and one of the virtual agents. Often one of the agents reached 9 victory points, but then dropped to 7, when the human player managed to steal one of the awards for the longest road or the largest army to win the game.
The heuristic agents also showed very good trading skills, and in general they were aware of the resources that they needed to fulfill their goals. Especially their abilities to use bank-trades, invention or monopoly cards to gain the right resources was impressive. In trades with their opponents the agents learned to improve their

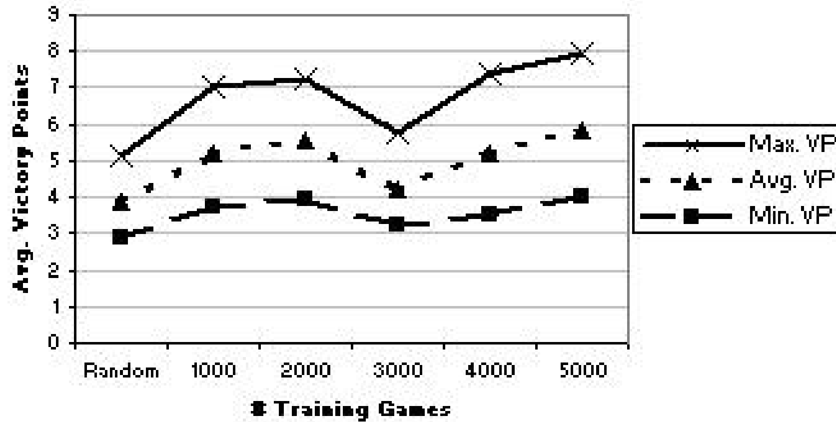


Figure 6.7: Average victory points of the heuristic policies scored against human opponents.

offers once a trade was rejected by the others. This is related to the improved approximation of the low-level Q-function.
We found that the major problems of the previous techniques, namely frequent passing and wrong decisions before reaching a sub-goal were eliminated. Frequent switching between different behaviours could not occur, because of the heuristic high-level strategy.
The most important remaining problem is the selection of starting positions. This selection is guided by the high-level value function, which was still learned, even though it is not used for behaviour selection. The agents did select good locations if they came first or second in the starting order, but usually the placement of the second settlement appeared to be rather random. The placement of settlements later in the game suffered from the same problem. Using more background knowledge here could lead to a much better performance.
The second big problem was that the agents tended to build too many roads before building a new settlement. Often four or more roads were built before the agents decided to save resources for a settlement. Experienced human players would only build the minimum number of roads necessary, namely one or two. The reason for this is that building a settlement requires a very specific combination of resources which is hard to obtain. If the opponents reject to trade, the agents may decide that it is better to build another road instead of passing, which would cause a small punishment. The policies did not learn that building another road actually leads to a worse situation than passing, because then two more resources (clay and wood) are needed.

Conclusion

The results of this approach were very encouraging. The combination of a simple heuristic high-level strategy and behaviours that were trained through reinforcement learning yielded policies that were even able to beat human players in several games. The derived policies also played significantly better than those that were constructed following the feudal or module-based approach. We could also see the importance of learning in the modules, which improves the performance of the system significantly. One should also not forget that without the learned value functions it would be impossible to use such a simple trading algorithm. Instead we would have had to

use a sophisticated economic model, which would have required much more detailed a-priori knowledge.

6.5 Heuristically Guided Learning

The third approach produced very successful policies that showed almost human-like playing strength. For a last short experiment we wanted to investigate whether it was possible to find a better high-level policy by following the heuristic high-level strategy during the training games. The heuristic strategy in this case served as a teacher for learning in the high-level. The low-level modules were reused from the previous experiments with the heuristic policies. The approximation for the high-level Q-function was also available, since it was used for goal-selection and trading. This time however the model trees were also used for high-level decision making.

Evaluation

It was expected that the performance of this method would be inferior to a pure heuristical behaviour-selection. The reason for this is that the learning system has to discover all the implicit knowledge that is hidden in heuristic rules, like dependencies on symmetries or combinations of features. We hoped however that guided learning would be an improvement over the first two approaches that relied solely on learning.

Figure 6.8(a) shows the performance of the new policies (we will later call them *Guided 0 - 4*) against a pure random player and the *Feudal 0* and *Module 1* strategies. The guided strategies appear to be clearly better than the pure random and the *Feudal 0* policy. The playing strength of the *Module 1* and the guided policies seems to be almost equal. The last policy *Guided 4* (5000 training games) suffered from bad luck: it won only 35% of its games against *Module 1*, even though it scored more victory points on average than its opponent. The weak performance of the *Guided 3* policy (4000 training games) had to be expected, since it used the same modules like the *Heuristic 3* policy, which showed a poor playing strength due to overfitting.
We also let the agents using the *Guided* policies play against the heuristic random player and against players that used the same modules in the low-level and the heuristic strategy in the high-level. The third opponent was the *Heuristic 0* policy, one of the weakest heuristic policies, which used only 1000 games for training. This was supposed to show whether improvements in the modules and some learning progress in the high-level function can reach the strength of the heuristic high-level strategy. The results of this evaluation are shown in figure 6.8(b).

We found that the *Guided 0-4* policies play significantly stronger than the heuristic random player. The importance of the high-level strategy is however indicated by the other diagram, which shows that the guided players could only win 20 to 30% of their games against the heuristic agents. The *Heuristic 0* policy outperformed most of the *Guided* policies, but there is a visible learning effect, which can be attributed to the improved modules and some progress in the high-level strategy. The *Guided 4* policy even managed to play at an equal level and win more than 50% of its test games.

The last step was the evaluation of some of the policies in games against human players. We decided to evaluate only the *Heuristic 1, 2* and *4* policies, which appeared to be particularly strong. Figure 6.9 shows the average victory points of the guided strategies scored against human players.

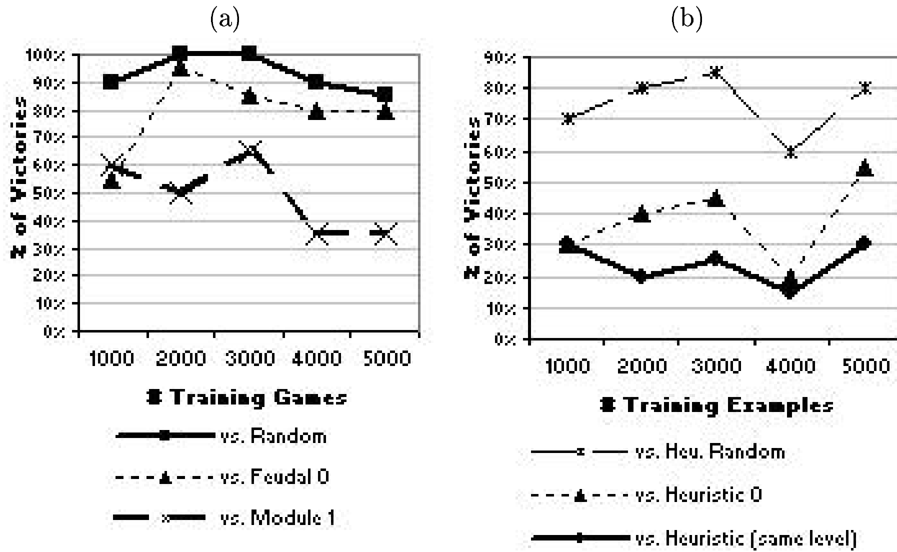


Figure 6.8: (a) Percentage of victories of the guided policies in games against a pure random player and the *Feudal 0* and *Module 1* policies. (b) Percentage of victories of the guided policies in games against the heuristic random player, the *Heuristic 0* policy and the *Heuristic* policy of the same level as the *Guided* policy.

The average victory points of these policies did not reach the level of the heuristic policies, but the results were not bad either. No agent was able to win one of the games, but on average one of them scored between 6 and 7 points. The performance of the best strategy (*Guided 4*) is slightly better than that of the best module-based policy (*Module 1*). We also found that the agents were better in obtaining either the longest road or the largest army, which made winning more difficult for the human player. The trading skills are identical to those of the heuristic agents.
Obviously the sub-optimal high-level policy is responsible for the weaker performance, compared to the agents that used a heuristical behaviour selection. One of the reasons may be that there are not enough negative training examples for the high level. The heuristics (almost) always select behaviours that make sense in this situation. Reinforcement learning on the other hand can only rate actions as bad if it tries them out and finds that it leads into a worse situation. A possible solution to overcome this problem is to use the heuristic policy only for one or two of the agents and let the other players learn strategies against it.

Conclusion

This approach was merely an experiment. We did not expect to find an improvement over the heuristic strategies. The *Guided* policies were not much better than those of the module-based approach. We think that guided learning works better if there are more negative training examples. Otherwise the learning algorithm is not able to distinguish good from bad moves, because it has only seen good ones.

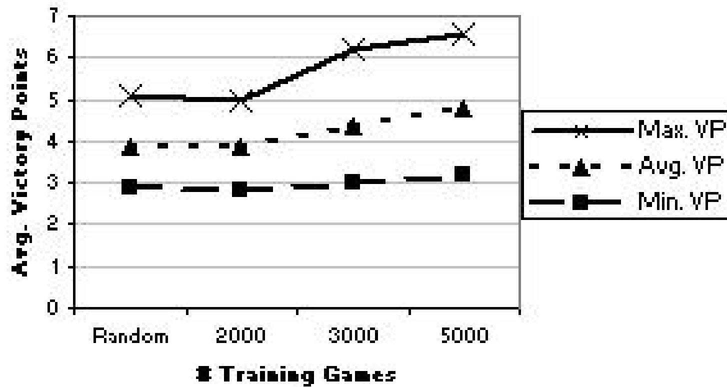


Figure 6.9: Average victory points of the guided policies scored against human opponents.

6.6 Strengths and Weaknesses of the Approaches

Combining Heuristics and Learning

The main result of this thesis is to show that artificial intelligence for games can benefit from using machine learning methods. The designers can and should use all their background knowledge and solve some parts of the game with proven methods, like heuristic rules or finite state machines. Learning can be used for those parts, where good heuristics are difficult to find. The trading algorithm for Settlers of Catan presented here is a good example: if we did not use learning, we would have to develop a whole economic model. This would probably be more time consuming and require the knowledge and experience of an expert.
The results of the learning process however made it obvious, that learning alone is not the ultimate solution. Games like Settlers of Catan are too complex to apply *tabula-rasa* learning, i.e. learning without any background knowledge. It is possible that an increased number of training examples would have yielded a policy of equal or better playing strength than the policies obtained with the heuristic high-level strategy, but this would have required far more learning time. Tesauro [Tes95] e.g. needed between 300,000 and 1,5 millions of training games before TD-Gammon reached its ultimate playing strength. Taking into account that one training run for 1000 SoC games takes between 1 and 2 days on two 1500 MHz CPUs, we would have needed more than one year to play as many training games. Obviously this is an approach that is completely unsuitable for commercial products. On the other hand, the *Heuristic 4* policy needed only less than one week of training time and still managed to be a challenge for human players. This is not much, compared to the usual development time for AI in complex environments like this.

Hierarchical Learning

The hierarchical decomposition of the learning task made it easier to apply background knowledge wherever possible. We found out that learning the low-level policies for the individual behaviours worked very well. Once invoked they often managed to execute good trades and actions to reach their subgoals. It seemed however that the system was not able to learn good high-level strategies. The sudden improvement of the policies once a heuristic high-level strategy was used added

to this impression.
The three layer architecture seemed to be well-suited for this game, because otherwise the number of possible actions would have been too large. The clear design of the modules also made it easier to understand the learned policies. This on the other hand facilitated the implementation of a heuristic high-level strategy.

Module-Based and Guided Learning

The module-based approach, in which modules are always executed until termination, showed better results than the first approach, in which high-level decisions are made after every low-level action. The reason for this is the weak high-level strategy, which makes rather random decision about the next module to be invoked. The modules however can often only fulfill their sub-tasks if their execution lasts longer than one step. Following a heuristic high-level strategy therefore resulted in a much better performance, because then the real strength of the modules became visible.
Using the heuristic strategy to guide the exploration resulted in policies that played slightly better than the best module-based policy. We believe that these policies could have been further improved, if more exploratory moves were made. The absence of negative training examples probably disturbed the success of this approach.

Offline Learning

The nature of the model tree learning algorithms required to use offline learning. Even if most other approaches for reinforcement learning in games use online methods like gradient descent, this method has several advantages. First, the granularity of learning can be controlled by the number of training games played between the learning stages. A small number of training matches before the next learning step means that the policies are only slightly different, but may yield better results, since newly discovered characteristics of the learning task can earlier be exploited. A large number of training games on the other hand is more efficient, because the learning algorithm does not need to be invoked all too often. Another advantage of offline tree-learning is that one can start with a relatively good first policy, if many random games are played before the first learning process is started.
The main disadvantage of this approach is that the memory requirements are very high. The results of all training games need to be stored, and the learning algorithm needs to deal with a huge amount of data.

Model Tree Learning

It is not quite clear how bad choices for the pruning parameters influenced the performance of the different approaches, especially for the first experiments, where we did not have any experience values.
We believe that this dependency on the pruning parameters of the model trees is the main disadvantage of tree-based function approximation. Badly chosen values can easily lead to severe overfitting and therefore to a decrease in performance. We hope that good rules of thumb for the choice of these parameters will be published in the next years, since this is a relatively new area in machine learning research. Despite of this problem, we find that model trees were very well suited for this task. The chosen splitting criteria make sense, and apparently can very well distinguish between important and irrelevant features. Time did not permit to try out

other function approximation techniques for comparison, but we believe that they would have had huge problems in an environment like that of Settlers of Catan. Linear approximators would require a very sophisticated coding scheme to take all the dependencies on combinations of attributes into account. This would however drastically increase the size of the state space. A clever and highly tuned neural network architecture with several hidden layers probably would have been able to solve this task, but the published results ([Sri00], [WanX99]) indicate that they require much more learning time than model trees. And of course fine tuning the parameters of the neural network requires at least as much experience as choosing good model tree parameters does.

Ranking of Policies

The following figure 6.10 shows a ranking of the different policies according to their performance in games against human opponents.

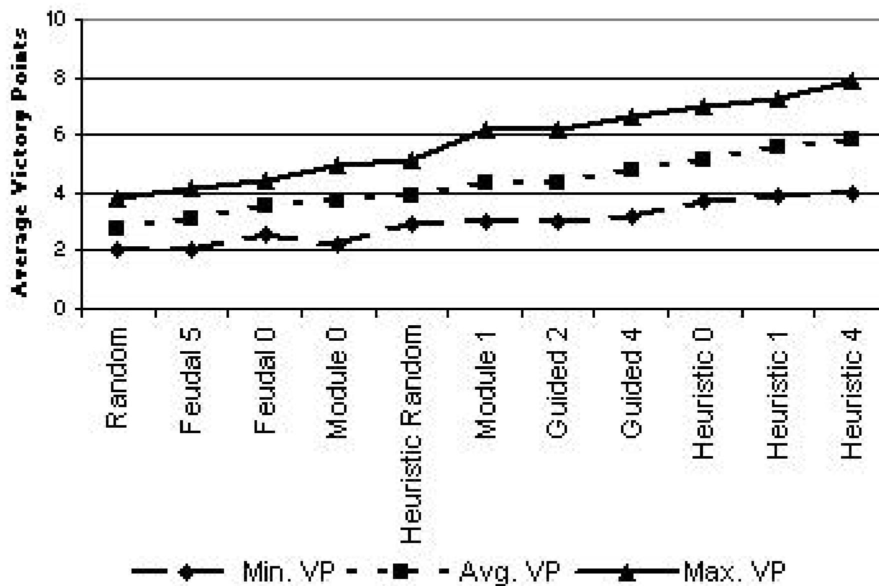


Figure 6.10: Average victory points of the best learned policies scored against human players.

One can see in figure 6.10 that the use of a heuristic high-level strategy and some background knowledge for the modules yields much better policies than the approaches that rely solely on learning.
Taking the complexity of the game and the simplicity of the architecture into account, the success of the policies is surprising and promising. Winning against the best policies like e.g. *Heuristic 4* is not easy, even for experienced players. Unfortunately there is no real benchmark player for Settlers of Catan, but for the future it would be interesting to test the obtained policies in games against other computer programs.

6.7 Possible Applications in Computer Games

We believe that several of the results of this thesis can be applied in computer games, either for the scientific or the commercial sector, to improve the performance of current game AIs.

Using Reinforcement Learning for Commercial Games

Even though reinforcement learning was responsible for huge successes in scientific approaches to game playing, there are almost no examples of commercial games that make use of this technique. We believe that more work is necessary to study and ultimately eliminate the "real-world" problems of RL in complex game environments. In the end this could make the use of machine learning techniques in commercial games more attractive. We hope that this thesis is one small step into the right direction.

Learning Components

The trading algorithm used in this paper for SoC has shown that learning can actually make the design of a game AI easier. The components often are small enough too require only little training time. This can help saving development time and costs (e.g. for expert knowledge) for commercial computer games.

Learning in Combination with other AI Techniques

If some hierarchical approach is used, it is not necessary, and often not advisable, to use learning for all components of the game. Instead the developers can use their favourite methods, like heuristic rules, search techniques or fuzzy state machines for all those tasks, where enough background knowledge is available. For the other parts that are not so well understood, learning algorithms can be applied.

Tree-Based Generalization

Model trees can very efficiently be used for large, discrete state spaces with many irrelevant variables. The state spaces of most games meet these criteria, which makes the use of tree-based regression for value-function approximation in these tasks very interesting. To the best of our knowledge however, there are no published research results about using model or regression trees for learning in game environments. Instead scientists in the field of game AI rely mainly on neural and linear approximation techniques. Since earlier results ([Sri00], [WanX99]) have shown that for some similar environments the use of regression trees is advantageous, it would be interesting to see applications of tree-based generalization techniques in games like chess, Go or Backgammon.

Settlers of Catan

We hope to have shown that Settlers of Catan is an interesting game for AI research. The rules for this game are more complex than those of games like chess or Go, but creating an implementation for this game is not as difficult as for most commercial games. To master the game it is necessary to develop many different skills, like path planning, long-term planning, resource scheduling, or trading abilities. This thesis has shown that well-known deterministic algorithms can be used for some of the components, while for other parts the use of machine learning methods is an interesting challenge.

6.8 Future Work

The results of combining learning and rule-based components for an agent in this complex game are promising. Still there is room for many modifications and directions of research which should be investigated:

- **More training examples:** The long duration of training matches and the huge memory requirements prevent the gathering of further training data. It is possible that more training examples would yield a much better performance.
- **Comparison with other function approximation techniques:** We found that model trees are very well-suited for this task, but we could not compare their performance to other approximation techniques like linear approximators or neural nets.
- **Comparison with other Settlers of Catan programs:** There is no single benchmark program for Settlers of Catan, and the existing programs are not designed to play against another computer program. Evaluating this program against other commercial and non-commercial products would be interesting.
- **Learning from human players or other programs:** Training the program by playing against human opponents or strong computer programs could help the agents adapt to stronger strategies.
- **Learning low-level actions:** Instead of using a-priori knowledge to translate actions from the second level into primitive actions, a third learning layer could be used.
- **Exploiting symmetries in the value functions:** A different coding scheme could be used that considers symmetric relations of features, e.g. ports distances or features that describe the state of the opponents.
- **Random maps:** The program could be trained to play on any random Settlers map.
- **Other games:** The results for Settlers of Catan were quite promising, but it would be interesting to see the performance of the developed methods in other, possibly even more complex games.

Chapter 7

Conclusion

In this thesis we studied the use of machine learning methods for a game that is different from the classical board games like chess, Go or Backgammon which are usually studied in scientific research about game playing. Our goal was to show that sophisticated learning techniques can be interesting for the game industry, which still relies on the simplest possible approaches for artificial intelligence in games. Using methods like reinforcement learning could instead open up unknown possibilities for creating more human-like intelligence, which is something that customers increasingly demand. The scientific community on the other hand has failed to create methods that are simple and efficient enough to be used in modern computer games. Closing this gap between research and industry could be the beginning of a fruitful collaboration, from which both sides could benefit.

We presented Settlers of Catan as an interesting new game for artificial intelligence research. This game has a lot in common with modern commercial strategy games, and requires the virtual agents to exhibit several very different skills. This paper has concentrated on demonstrating the capabilities of a combined approach, which used a-priori knowledge or heuristic strategies wherever possible, and reinforcement learning for all other components. That way it was possible to solve very difficult problems through the use of learning. This kept the design simple and reduced the development time of the whole system.

A novel offline method for self-training in games was developed, which made use of model trees for the approximation of the value function. This algorithm is well suited for problems with discrete state spaces and lots of irrelevant variables. In contrast to methods like linear or neural-network approximation, tree-based regression is also capable of representing sharp discontinuities in the value functions. All these properties make the learning method especially well-suited for game environments.

Several learning approaches were tried, and it was found out that learning without a-priori knowledge is not able to learn good strategies in reasonable time. Instead the combination of heuristics and learning from experience created policies that could compete with human players and was even able to beat them from time to time. The central result was, that one of the most important components of the agents, namely the trading algorithm, could be almost completely derived from learning. A great part of the improved policy against strong opponents could be attributed to this trading module.

These are very promising results, which indicate that reinforcement learning can be

successfully used to learn strategies for very complex computer games. We hope to see more results for other games of the same or even higher complexity in the next years.

Appendix A

Heuristics for Low-Level Actions

A.1 Low-Level Actions

The third layer in the agent architecture consists of several heuristic algorithms. A short description of each module is given here.

Action 1 and 6: Pass and Buy Development Card

These actions involve no decision.

Action 2 and 3: Build Settlement or City

If there is a free building place at the current goal, a settlement is build there. Otherwise the algorithm evaluates all possible locations using the high-level value function. For settlements, all places that are connected to roads and are at least 2 roads away from all other settlements are possible building places. Cities can be built to replace of existing settlements.

Action 4: Extend Longest Road

This is a heuristic depth-first search algorithm. Starting from the endpoints of the currently longest road, all possible extensions are calculated, and a heuristic numerical value is assigned to every road. This value takes several factors into account: the maximum length of the resulting extension is the most important factor, because extensions that have a high potential for the future should be favoured. If existing roads can be used, e.g. by connecting two separate road networks, this is also a positive factor. If the path uses roads that are adjacent to the opponents' existing roads, this is a negative factor. The later such contact is made the better, because then the building of the whole path is more likely to be actually executed as planned. Contacts however can be ignored if the player has the resources or a construction card to build all the roads up to the contact depth at once. For all paths the factors are combined into a weighted sum which describes their value of extending the longest road. The starting points of the paths are then evaluated by calculating the mean value for all paths that start there, and the best road is built immediately.

Action 5: Build Road towards Goal

The selection of the next goal is described in the next section. Assuming that a goal is available, a breadth-first algorithm finds all the shortest paths to that goal. All paths are evaluated by a weighted sum, using fixed hand-coded weights. The most important factor is the length of the path, but like in the above algorithm, opponent contact must also be considered. The third important criterion is whether the longest road can be extended by this road. All these factors are calculated, and the best road is built.

Action 7: Play Knight Card

Playing out the card itself requires no intelligence. The placement of the robber uses the same standard placement algorithm that is used whenever a seven is thrown. The leading player is usually the target, and his most valuable production field is selected as the new location for the robber.

Action 8 and 9: Playing Invention or Monopoly Cards

The values of all possible inventions or monopolies are calculated, like in the trade heuristics described below. The best one is then chosen.

Action 10: Play Road Construction Card

The card is played out and then the player must invoke one of the road building algorithms described above.

Action 11: Bank Trade

First, a list of all possible bank trades is created. For each of these possibilities the change in the current module's value function in case of executing this trade is calculated, and the best one is chosen.

Action 12: Opponent Trade

The agents make only 1:1 or 2:1 trades, which means they never demand more than one resource, and never give away more than two. For each of these possibilities the change in the current module's value function in case of executing this exchange is calculated. At the same time, for every opponent the change in his high-level value function is calculated. First, the agent proposes the best trade for himself. If any opponent accepts, the initiator decides whether this trade is not too good for the trading partner, i.e. his estimated improvement in the value function is below a threshold. If a counter offer is placed, this trade is evaluated and compared to the original trade. The counter offer is only accepted, if it is nearly as valuable as the original offer, and is not too much better for the other player. If all opponents reject the trade or make bad counter offers, the agent proposes another trade. He then selects the next best trade that is slightly better for any of the opponents. The agents remember all unsuccessful offers, and never propose the same trade twice. If only trades remain that would result in a state that is worse than the current state, or a fixed number of offers was proposed, the agent chooses another action within the active module.
One could also use the high-level value function for the evaluation of a trade, but using the low-level values has yielded better results.

A.2 Goal Selection

For building settlements and roads, the agents have to figure out a target position that they want to reach. This can be a port, a building place, or a point where the longest road or the expansion of an opponent can be disturbed. A general rule is that goals in the vicinity of the existing road network should be favoured over positions that are farther away. Not only does this save resources for the roads, but it also reduces the risk of being blocked by other players. The goal-finding procedures for the different behaviours are described below.

Behaviour 5: Ports

The goal is the position of the best port nearby. The value of a port is determined by the production probability of the corresponding resource, where 3:1 ports are assigned 2/3 of the probability of the most frequent resource. This heuristic value is then divided by the number of roads necessary to connect to this location. The port with the maximum value is selected as the goal.

Behaviour 6: Disturb Opponent Roads

If an opponent has holes in his longest road, and these holes are within 2 roads distance, this is the primary target. Otherwise the opponents are sorted by their victory points, and leading players or players that have the longest road are chosen for their roads to be disturbed. The goal position is then one of the end-points of the opponent's longest road. This should prevent them from extending their longest road.

Behaviour 7: Disturb Opponent Expansion

In this behaviour we want to block an opponent's expansion by reducing his number of possible building places. For every possible road and settlement in 2 roads distance we calculate the reduction of building places for every opponent. It is more valuable to block locations where the opponent could otherwise build immediately or after building only 1 additional road. This can be achieved by building a settlement, which destroys other building places through the distance rule. The reduction of building places that are farther away from the opponent's current network is a good indication whether it is possible to trap him in only a small part of the board. This seriously damages the winning chances of that player. The heuristic value is a weighted sum of the reductions, weighted by the distance to the opponent's roads. The own effort (number of roads and settlements) is also taken into account, when the best location is selected as the next goal.

All other Behaviours: Expansion

All other behaviours select their goal to expand the own colony. Therefore all building places within 5 roads distance are evaluated, taking the shortest path to the goal and the new settlement into account. The heuristic value of the goal is the improvement in the high-level value function divided by the distance, in order to favour nearby locations. The chosen goal is the one with the highest heuristic value.

Appendix B

Definition of Features

The following is a list of the features that were used to define the state space in Settlers of Catan. Low-level features that directly describe the state of every single road or settlement location on the board were avoided, and the state was encoded by more sophisticated high-level features. Altogether there are 216 features, but the list only describes the important groups.

- **Counters:** The number of built roads, settlements and cities, as well as the victory points, development cards, resources and knights.
- **Likelihoods:** The overall likelihood of resources, defined by the number of fields and the assigned dice values.
- **Production:** The production likelihood for each resource and player.
- **Resources:** The currently held resources of all players, including the opponents. This is not cheating, because one can simply count the produced and consumed resources of each type.
- **Possible Actions:** Actions for which the resources are available. Involves also the exchange of resources with the bank.
- **Exchange Habits:** Which resources were already exchanged by opponents with the bank.
- **Available Buildings:** Number of roads, settlements and cities that are left to be built.
- **Robber:** Who, and which resource is blocked by the robber.
- **Ports:** Possession of ports for all players. Distance to the closest ports of every type for the own player. Minimum distance of any opponent to the same ports.
- **Development Cards:** Possession of active and passive development cards of each type (only for learner himself).
- **Awards:** Owner of the awards for longest road and largest army.
- **Road Features:** High-level features defining the graph structure of the road networks: number of connection components, branches and holes in the roads.
- **Longest Road Extension:** Is an extension of the longest road possible.

- **Expansion Distances:** Number of building places within 0 to 5 roads distance for all players. Number of building places for the opponents that lie within 1 field distance of the own network (targets for disturbing actions).
- **Winning Indicator:** Indicates whether any player has 8 or more victory points, and who is the leader.
- **Last Behaviour:** What was the last active behaviour.

Appendix C

List of Abbreviations

AI	*Artificial Intelligence*
AL	*Artificial Life*
ANN	*Artificial Neural Network*
Avg	*Average*
CART	*Classification and Regression Trees*
FSM	*Finite State Machine*
FuSM	*Fuzzy State Machine*
GA	*Genetic Algorithm*
GB	*Gigabyte*
IO	*Input - Ouput*
Max	*Maximum*
MDP	*Markov Decision Process*
Min	*Minimum*
ML	*Machine Learning*
MSE	*Mean Squared Error*
Q	*Action Value, State-Action Value*
RL	*Reinforcement Learning*
RMS	*Root Mean Squared*
SoC	*Settlers of Catan*
TD	*Temporal Difference*
V	*Value, State Value*
VP	*Victory Point(s)*

Appendix D

Summary of Notation

A	Set of all actions
A_s	Set of all actions available in state s
a, a_t	Action (at time t)
$E[X]$	Expected value of the random variable X
$E_\pi[X]$	Expected value of random variable X, if the agent follows policy π
$e(s, a)$	Eligibility trace of a state-action pair
max	Maximum
min	Minimum
$O(\ldots)$	Order of computational complexity
$Prob(X = x)$	Probability that the random variable X takes on the value x
$Prob(X = x \| Y = y)$	Conditional probability that $X = x$, if $Y = y$
$Q^\pi(s, a)$	Action value for taking action a in state s and then following policy π
$Q^*(s, a)$	Action value of a in state s if we thereafter follow an optimal policy
r, r_t	Reward (at time t)
S	Set of all states
s, s_t	State (at time t)
t	Time step
$V^\pi(s)$	Value of the state s if we follow policy π
$V^*(s)$	Value of the state s if we follow an optimal policy
X^T	Transposed of the matrix X
$\bar{x}$	Mean of the variable x
$y(t)$	Target value of the example t
γ	Discount factor
δ	Transition function
$\vec{\theta}$	Parameter vector for function approximation
λ	Trace-decay parameter
π	Policy, mapping of states to actions
$\pi(s)$	Selected action in state s by deterministic policy π
$\pi(s, a)$	Probability of choosing action a in state s
$\sigma(T)$	Standard deviation of the target variable in the set T
$\nabla_{\vec{w}} F(\vec{w})$	Gradient of the function F with respect to the vector $\vec{w}$

Bibliography

[Ale96] W.P. Alexander, S.D. Grimshaw. *Treed Regression.* in: Journal of Computational and Graphical Statistics 5, 1996

[Bar03] Andrew G. Barto, Sridhar Mahadevan. *Recent Advances in Hierarchical Reinforcement Learning.* to appear in: Discrete-Event Systems Journal, 2003

[Bil99] Darse Billings, Lourdes Peña, Jonathan Schaeffer, Duane Szafron. *Using probabilistic Knowledge and Simulation to play Poker.* in: Proceedings of the 16th National Conference on Artificial Intelligence, 1999

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, United Kingdom, 1995

[Bre84] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont CA, 1984

[Cha94] P. Chaudhuri, M.C. Huang, W.Y. Loh, R. Yao. *Piecewise-polynomial Regression Trees.* in: Statistica Sinica 4, 1994

[Cri96] R. Crites, A. Barto. *Improving Elevator Performance using Reinforcement Learning.* in: D.S. Touretzky, M.C. Mozer, M.E. Hasselmo (Editors): *Advances in neural information processing systems*, 8, 1996

[Day93] P. Dayan, G.E. Hinton. *Feudal Reinforcement Learning.* in: S.J. Hanson, J.D. Gowan, C.L. Giles (Editors). *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, San Mateo CA, 1993

[DeL00] Mark A. DeLoura (Editor). *Game Programming Gems.* Charles River Media, Rockland MA, 2000

[DeL01] Mark A. DeLoura (Editor). *Game Programming Gems 2.* Charles River Media, Hingham MA, 2001

[Die00] Thomas G. Dietterich. *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition.* in: Journal of Artificial Intelligence Research 13, 2000

[Dor94] M. Dorigo, M. Colombetti. *Robot Shaping: Devloping Autonomous Agents through Learning.* in: Artificial Intelligence 71(2), 1994

[Eva03] Richard Evans. *AI in Games: A personal View.* available online at: www.gameai.com/blackandwhite.html, 26th April 2003

[Fra98] Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, Ian H. Witten. *Using Model Trees for Classification.* in: Technical Note, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand, 1998

[Fue96] Johannes Fuernkranz. *Machine Learning in Computer Chess: The next generation.* in: International Computer Chess Association Journal 19(3), 1996

[Fue97] Johannes Fuernkranz. *Knowledge Discovery in Chess Databases: A Research Proposal.* Technical Report, Austrian Research Institute for Artificial Intelligence, 1997

[Fue01] Johannes Fuernkranz. *Machine Learning in Games: A Survey.* in: J. Fuernkranz, M. Kubar. *Machines that Learn to Play Games.* Nova Scientific Publishers, Huntington NY, 2001

[GAI03] *Games Making Interesting Use of Artificial Intelligence Techniques.* available online at: www.gameai.com/games.html, 26th April 2003

[Har68] P.E. Hart, N.J. Nilsson, B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* in: IEEE Transactions on Systems Science and Cybernetics, SSC-4(2), 1968

[IBM97] IBM Research. *How Deep Blue Works.* available online at: www.research.ibm.com/deepblue/meet/html/d.3.2.html, 26th April 2003

[Kae93] Leslie Pack Kaelbling. *Hierarchical Learning in Stochastic Domains: Preliminary Results.* in: Proceedings of the 10th International Conference on Machine Learning, Amherst MA, 1993

[Kae96] Leslie Pack Kaelbling, Michael L. Littman, Andrew W. Moore. *Reinforcement Learning: A Survey.* in: Journal of Artificial Intelligence Research 4 (1996)

[Kal98] Zsolt Kalmár, Csaba Szepeszvári, András Lörincz. *Module-Based Reinforcement Learning: Experiments with a Real Robot.* in: Machine Learning 31, 1998

[Kar92] A. Karalic. *Linear Regression in Regression Tree Leaves.* in: International School for Synthesis of Expert Knowledge, Bled, Slovenia, 1992.

[Lai00] John E. Laird, Michael van Lent. *Human-level AI's Killer Application: Interactive Computer Games.* Invited Talk at AAAI 2000, 2000

[Lai03] John E. Laird. *Bridging the Gap between Developers and Researchers.* available online at: www.gamasutra.com/features/20001108/laird_03.htm, 26th April 2003

[Lit94] Michael L. Littman. *Markov Games as a Framework for Multi-Agent Reinforcement Learning.* in: Machine Learning 11, 1994

[Mae90] Pattie Maes, Rodney A. Brooks. *Learning to Coordinate Behaviours.* in: Proceedings of AAAI-90, Boston, MA, 1990

[Mah91] S. Mahadevan, J. Connell. *Scaling Reinforcement Learning to Robotics by Exploiting the Subsumption Architecture.* in: Proceeding of the 8th International Workshop on Machine Learning, 1991

[McG01] Amy McGovern, Andrew G. Barto. *Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density.* in: C. Brodley, A. Danyluk (Editors): *Proceedings of the 18th International Conference on Machine Learning*, Morgan Kaufmann, San Francisco CA, 2001

[McP03] Scott MacPherson. *Settlers of Catan Strategy and Tactics Guide.* available online at: www.geocities.com/TimesSquare/Metro/5303/catanstrat.html, 28th March 2003

[Mal01] Donato Malerba, Annalisa Appice, Antonia Bellino, Michelangelo Ceci, Domenico Pallotta. *Stepwise Induction of Model Trees.* in: F. Esposito (Editor), AI*IA 2001: Advances in Artificial Intelligence, Lecture Notes in Artificial Intelligence, 2175, Springer, Berlin, Germany, 2001

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill International Editions, Singapore, 1997

[Par98] Ronald E. Parr. *Hierarchical Control and Learning for Markov Decision Processes.* PhD Thesis, University of California, Berkeley CA, 1998

[Pye98] Larry D. Pyeatt, Adele E. Howe. *Decision Tree Function Approximation in Reinforcement Learning.* in: Technical Report CS-98-112, Colorado State University, 1998

[Qui86] J.R. Quinlan. *Induction of Decision Trees.* in: Machine Learning 1(1), 1986

[Qui92] J.R. Quinlan. *Learning with Continuous Classes.* in: Proceedings Australian Joint Conference on Artificial Intelligence, World Scientific, Singapore, 1992

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo CA, 1993

[Ram01] Jan Ramon, Hendrik Blockeel. *A Survey of the Application of Machine Learning to the Game of Go.* in: Proceedings of the First International Conference on Baduk, 2001

[Rum94] G.A. Rummery, M. Niranjan. *On-line Q-learning using Connectionist Systems.* Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, United Kingdom, 1994

[Rus95] Stuart J. Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River NJ, 1995

[Sal99] Marc Saltzman. *Game-Design: Die Geheimnisse der Profis.* X-Games, Markt&Technik Verlag, Munich, 1999

[Sam59] A. L. Samuel. *Some Studies in Machine Learning using the Game of Checkers.* in: IBM Journal on Research and Development 3, 1959

[Scha96] J. Schaeffer, R. Lake, P. Lu, M. Bryant. *Chinook: The World Man-Machine Checkers Champion.* in: AI Magazine 17(1), 1996

[Schw97] Hans Rudolf Schwarz. *Numerische Mathematik* B.G. Teubner, Stuttgart, 1997

[Sin92] S.P. Singh. *Transfer of Learning by Composing Solutions of Elemental Sequential Tasks.* in: Machine Learning 8(3), 1992

[Sin00] S.P. Singh, T. Jaakkola, M.L. Littman, C. Szepesvari. *Convergence Results for Single-Step On-policy Reinforcement-Learning Algorithms.* in: Machine Learning 38, 2000

[Sri00] Manu Sridharan, Gerald Tesauro. *Multi-agent Q-learning and Regression Trees for Automated Pricing Decisions.* in: Proceeding of the 17th Conference on Machine Learning, Morgan Kaufmann, San Francisco CA, 2000

[Sut88] Richard S. Sutton. *Learning to Predict by the Methods of Temporal Differences.* in: Machine Learning 3, 1988

[Sut98] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge MA, 1998

[Sut99] Richard S. Sutton, Doina Precup, Satinder Singh. *Between MDPs and Semi-MDPs.* in: Artificial Intelligence 112, 1999

[Tes89] Gerald J. Tesauro. *Neurogammon: A Neural-Network Backgammon Program.* in: Proceedings of the International Joint Conference on Neural Networks, Volume III, 1989

[Tes95] Gerald J. Tesauro. *Temporal Difference Learning and TD-Gammon.* in: Communications of the ACM 38, 1995

[Tes99a] Gerald J. Tesauro, Jeffrey O. Kephart. *Pricing in Multi-Agent Economies using Multi-Agent Q-Learning.* in: Proceedings of the Workshop on Decision Theoretic and Game Theoretic Agents, London, United Kingdom, 1999

[Tes99b] Gerald J. Tesauro. *Pricing in Agent Economies using Neural Networks and Multi-Agent Q-Learning.* in: Proceedings of the IJCAI-99 Workshop on Learning About, From and With Other Agents, Stockholm, Sweden, 1999

[Teu95] Klaus Teuber. *Die Siedler von Catan: Regelheft.* Kosmos Verlag, Stuttgart, 1995

[Teu00] Klaus Teuber (Editor). *Die Siedler von Catan: Das Buch zum Spielen.* Kosmos Verlag, Stuttgart, 2000

[Tho02] Robert Thomas, Kristian Hammond. *Java Settlers: A Research Environment for Studying Multi-Agent Negotiation.* in: Proceedings of Intelligent User Interfaces (IUI-2002), 2002

[Thr95] Sebastian Thrun. *Learning to Play the Game of Chess.* in: G. Tesauro, D. Touretzky, T. Leen (Editors). *Advances in Neural Information Processing Systems 7.* The MIT Press, Cambridge MA, 1995

[Tor97] Luís Torgo. *Kernel Regression Trees.* in: European Conference on Machine Learning, Poster Paper, 1997

[Tsi97] J. N. Tsitsiklis, B. Van Roy. *An Analysis of Temporal-Difference Learning.* in: IEEE Transactions on Automatic Control 42, 1997

[WanY97] Yong Wang, Ian H. Witten. *Induction of Model Trees for Predicting Continuous Classes.* in: Proceedings of the poster papers of the European Conference on Machine Learning, Prague, 1997

[WanX99] Xin Wang, Thomas G. Dietterich. *Efficient Value Function Approximation Using Regression Trees.* in: Proceeding of the IJCAI-99 Workshop on Statistical Machine Learning for Large-Scale Optimization, Stockholm, Sweden, 1999

[Wat89] C. Watkins. *Learning from Delayed Rewards.* Ph.D. dissertation, King's College, Cambridge, United Kingdom, 1989

[Wat92] C. Watkins, P. Dayan. *Q-Learning* in: Machine Learning 8, 1992

[Woo03] Steven Woodcock. *Game AI: The State of the Industry.* available online at: http://www.gamasutra.com/features/19990820/game_ai_01.htm, 26th April 2003

[Zha96] W. Zhang, T.G. Dietterich. *High-Performance Job-Shop Scheduling with a Time-Delay TD(λ) Network.* in: D.S. Touretzky, M.C. Mozer, M.E. Hasselmo (Editors): *Advances in neural information processing systems*, 8, 1996