Aalborg University
Esbjerg Institute of Technology

Master of Science Thesis

Efficient Supervised Reinforcement Learning in Backgammon

by

Boris Søborg Jensen

Supervisor: associate professor Daniel Ortiz

Esbjerg, June, 2009

Title

Efficient Supervised Reinforcement Learning in Backgammon

Project period

P10

February 2, 2009 - June 10, 2009

Projectgroup

F10s-1

Group members

Boris S. Jensen

Supervisor

Daniel Ortiz-Arroyo

Total number of pages: 60

Abstract

Reinforcement learning is a powerful model of learning, but has the practical problems of requiring a large amount of experience, and bad initial performance of the learner. This master degree thesis presents a possible solution to the second of these problems in the domain of backgammon, where a supervisor is employed to ensure adequate initial performance, while retaining the ability of the learning agent to discover new strategies, enabling it to surpass the supervisor. There is no standard measure for this kind of improvement, so an ad-hoc measure is proposed, which is used to guide the experimentation. Using this measure shows that supervised reinforcement learning can be used to solve the problems posed by ordinary reinforcement learning in the backgammon domain.

Preface

This master degree thesis is written by group F10s-1-F08, at Aalborg University Esbjerg. The project theme is Supervised Reinforcement Learning. Part of the code used in this project, namely the neural network implementation has been provided by the NeuronDotNet project [Neu08]. The Pubeval implementation has been adapted from [Tes08].

Reference to other parts of the report will be marked as follows:

- References will be marked like this: Figure 2.1.
- Citations will be marked like this: [Tes95].

The folders on accompanying cd-rom contains the following:

- **Articles**. Articles cited in the report.
- **Thesis**. This thesis itself in pdf-format and LaTeXformat.
- **Source**. Source code for the supervised reinforcement learning backgammon controller.
- **Results**. Excel files containing the results of the tests described in the report.

The author would like to thank associate professor Daniel Ortiz for great patience and optimism.

Contents

Chapter 1

Introduction

Machine learning is often based on the same principles as human learning. Imagine for instance a boy, trying to throw a ball through a ring. One way to for the boy to learn this, is to just throw the ball and observe what happens. At first the results will seem random, but little by little, the aim and throw will get better, as the boy learns by experience, what does and does not work. This is the basic principle behind the field of machine learning known as reinforcement learning: Learning comes from interacting with the environment.

This simple principle gives rise to the reinforcement learning model, in which an agent interacts with the environment by performing actions dependent on the state of the environment. The environment responds by changing its state and by rewarding the agent with a scalar reward signal. The learning occurs, as the agent tries to maximize the cumulative reward over time. The simplicity of this model makes it possible to apply reinforcement learning to a wide variety of situations. Examples of domains, where reinforcement learning has been applied succesfully, include dynamic channel allocation for cellphones [SB97], elevator dispatching [CB96] and backgammon [Tes95]. In the case of the backgammon domain, Gerald Tesauro created the TDGammon player, which is based on a reinforcement learning technique called temporal-difference learning. TDGammon eventually reached a playing strength that rivals or perhaps even surpasses that of human world class backgammon master players, and some of its tactics have even been adopted by these players. One of the surprising lessons learned from TDGammon was, that it could achieve this high level of performance, even though it had started out knowing nothing about the game, and had only learned through self-play, i.e. playing both sides of the board.

As powerful as the reinforcement learning principle may be, there are still some drawbacks. They are related to the fact that, starting from scratch, it can take a long time to reach an acceptable level of performance. This can be annoying in learning from simulated experience, where the amount of simulations required can prohibit extensive testing of variations - the version of TDGammon, that reached master level, played 6.000,000 games against itself[Tes02] - but the real problems arise when the actions are performed in a real environment. Here, it may be very difficult to obtain the required amount of experience, and the consequences of not performing at an adequate level may be severe, depending on the domain.

What can be done to avoid these drawbacks? An answer may be found by considering, that the reinforcement learning principle is not the only mode of learning. Humans can also learn from teachers, either by mimicking them, or by receiving advice or evaluations from them. In the example of the ball-throwing boy, imagine the situation when a parent or teacher is nearby to demonstrate the throw, or give praise, when the boy has the correct posture. In such a situation, the boy might learn faster, and the first shots will likely not be as bad, since the teacher will guide them. With such a setup, it is important, that the teacher at some time, when the boy has become proficient, becomes less involved, leaving the boy to discover by experience more advanced techniques and skills, that may even surpass those of his teacher. Such a mode of learning could be called supervised reinforcement learning.

Fortunately, in many cases it is possible to formulate simple heuristics that may act as a teacher to a computer program trying to learn by reinforcement. The purpose of this report is to explore how supervised reinforcement learning can be applied to improve the results of TDGammon. Chapter 2 describes the game of backgammon, while chapter 3 describes how reinforcement learning can be treated mathematically. The classic reinforcement theoretical corpus is concerned with state spaces, that can comfortably be represented in a table, but real world problems, including that of learning to play backgammon, often operate in state spaces that are much too large to practically fit in any table, so a later section of the chapter adresses the practical problem of how to learn anything in a state space as vast as that of backgammon. Chapter 4 describes the other work done, and the challenges faced in the field of supervised reinforcement, as well as how the task of improving TDGammon compares to other tasks that supervised reinforcement learning has been used to solve. This comparison shows that there is a need for a custom testbed to measure the improvements achieved, so chapter 5 provides one such measure. Since the measure is custom made, it cannot be used for comparing with previous results in related work, but it can be used as a guide for experiments within this report. Chapter 6 uses the previous knowledge to produce three models of supervised reinforcement learning for the backgammon domain. These models only differ in the method by which the teacher gradually withdraws, as the learning agent becomes more knowledgeable. The models are tested in chapter 7, and finally chapter 8 sums up the feasibility of using supervised reinforcement learning to improve the results of TDGammon.

Chapter 2

The Backgammon Domain

2.1 Introduction

This chapter introduces the rules of the game of backgammon. This is done in section 2.2 and provides the basic understanding required of the next two sections. Section 2.3 discusses the complexity involved in playing well, and section 2.4 shows how the game mechanics can be used to specify how the backgammon agents should interact. In the following, the terms player and agent will be used interchangeably.

2.2 Backgammon

Backgammon is a boardgame for two players. Each player has 15 checkers, which start on the board in prespecified positions, and the goal for each player is to move all his checkers off the board. The first player to do so, wins the game. Figure 2.1 shows the initial placement of checkers for each player. Each point has a number from 1 to 24. These numbers are relative to the current player. The home table of the current player always contains the points 1-6, while the opponent home table always contains 19-24. The players move their checkers in opposite directions, trying to get all their checkers into their home table.

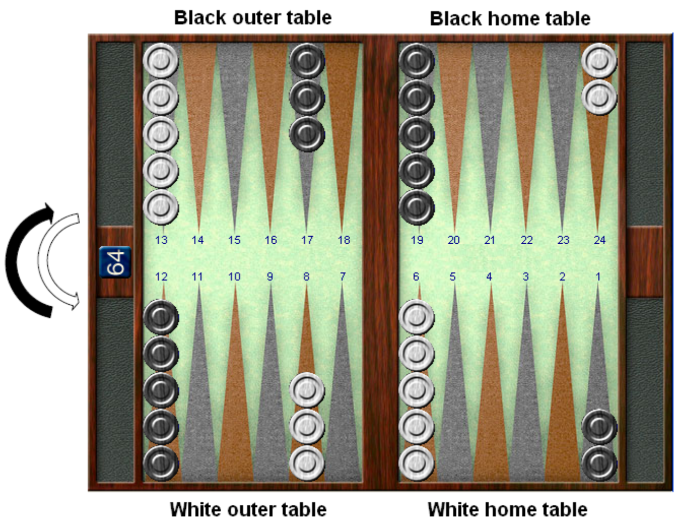

Figure 2.1: The backgammon board initial setup[Hei04]

2.2.1 Moving

The players take turns to move their checkers. Each turn, a player throws 2 dice. Unless a double is thrown, two moves are allowed, one for each die. If a double is thrown, then 4 moves are allowed, each move using the number of pips on one of the dice, e.g. if a double 5 is thrown, then the player has 4 moves of 5 points each. These moves can be combined freely, i.e. the moves can be divided on different checkers, or a single checker can use more than one of the available moves. Multiple moves are carried out consecutively, e.g. using 3 moves of 5 doesn't count as making a single move of 15. A player is not allowed to pass on his moves, as many moves as possible must be made each turn. If a point is occupied by two or more checkers, then they are safe, and no move by the opposing player may land at that point. If a point is occupied by a single checker, that is called a blot, and the checker is vulnerable to a hit by the opposing player. Checkers are hit if the opposing player moves a checker onto the blot checker point. Checkers, that are hit, are moved to the bar, that divides the outer table from the home table, and from here they must enter the home table of the opposing player. Entering the opposing players home table counts as a normal move, e.g. using a move of 1 lands the checker on the 24 point. While a player still has checkers on the bar, he may not move any other checkers.

2.2.2 Bearing Off

Once a player has all his checkers in his own home table, he may start to remove them from the game. This is known as bearing off. Using a move of 1 will bear off a checker from the 1 point, using a move of 2 will remove a checker from the 2 point and so on. A player doesn't have to bear off, he may also spend available moves moving checkers inside his home table. If the player has an available move, that is higher than the highest point, at which the player still has checkers, then he may use that move to bear off a checker from the that point. If at any time the opposing player captures a checker, then the player cannot continue to bear off, before all his checkers are again in his home table.

2.2.3 Winning

The first player to bear off all his checkers wins the game and the current stake. The normal stake to be won is 1 point. This stake can be modified in two ways: By winning very impressively or by using a special die called the doubling cube. If the opposing player has not yet borne off any of his checkers, then the player has won a "‘gammon"’, which doubles the stake. If the opposing player still has checkers on the bar, or in the players home table, then the player has won a "‘backgammon"’, which triples the stake instead. The doubling cube is a die, whose side shows the numbers 2, 4, 8, 16, 32 and 64, representing the current value of the stake. When it is his turn, a player may offer to double the stake, before he throws the dice. If this offer is accepted by the opposing player, then current stake is doubled, and the doubling cube is used to indicate this. If the opposing player rejects the offer, then the player wins the current stake before the doubling. Initially, any player can make the doubling offer, but once the stake has been doubled, only the opposing player is allowed the next doubling offer.

2.3 Complexity

Strategically, backgammon can be divided into two distinct phases: The contact phase and the race phase. The game starts in the contact phase, and the race phase then occurs, when the last checkers of each player have passed each other, removing any possibility for the players to hit the opponent's checkers, or block their path. The strategy of the race phase thus reduces to setting up the checkers in the home table, such that all moves can be fully utilized when bearing off. The contact phase on the other hand requires the weighing of several subgoals. For instance, it is often desirable to hit an opponents checker, but if the player has many single checkers in his home table, it may not be such a good idea, since the opponent in his next turn may be able to hit those checkers in return. This could be a bad trade, since the player usually has spent a number of moves on checkers in his home table. Another consideration is building blockades, or primes, which are consecutive positions, where the player has two or more checkers. Long primes impedes the opponents checkers, since the prime must be traversed in a single move. It is therefore important to have a flexible position, that allows using the future dice rolls to build a prime. Obtaining this flexible position, however, can mean exposing single checkers to being hit by the opponent, leading to a conflict between the goals of building primes and avoiding blots. To obtain good performance, such concerns must be addressed appropriately for each of the board states, that are encountered during a game. The number of possible states in backgammon is estimated to be in the order of 10^{20}[Tes95]. A lot of these states have an extremely small likelihood of occuring during a normal game, so in reality the number of states, for which an agent needs to have good judgement is much smaller, but the number of states, that can reasonably be expected to occur in a normal game is still very high.

Another source of complexity is the branching factor. One way to evaluate the consequences of a given move is to imagine the possible scenarios that might occur after the move. This is called searching the game tree. The nodes of a game tree are board states, and the children of a node are all the states that might occur as the result of a player taking a turn from that state. The feasibility of game tree search is bounded by the branching factor, which is the average number of children of a node in the tree. The average number of moves in backgammon, given a certain dice combination, is around 20 [Tes95], and since the number of effectively different dice combinations is 21, backgammon has a branching factor of around 400, which in practice limits the depth, to which the game tree can be searched, to 2 or 3.

2.4 A Framework for Backgammon Agents

Backgammon can be seen as a system with partial knowledge of the dynamics: The action of making a move leads with certainty to a new state, called the afterstate, at which point the opponent takes his turn by rolling the dice, and making his move. The opponents turn can be seen as a probabilistic system response. The important thing to realize here is, that this system response is dependent only on the afterstate. Stated another way, the expected system response is the same for two board state/dice roll combinations, that lead to the same afterstate. This means that the utility of an afterstate will be the same as that of all the state/dice combinations that lead to such an afterstate. Therefore it makes sense to only evaluate the afterstates, since that reduces the size of the total state space by a factor of 21.

A way to implement this would be to have a central backgammon playing framework, that rolls the dice for each player, and generates the possible board states, that can be reached from the current board with the new dice. The job of an agent is then only to evaluate the board states that are presented to it. This is the way, that TDGammon was implemented [SB98]. Using these evaluations, the framework can then decide which move to make for the player. The obvious choice would be to choose the move with the highest evaluation, but in some situations it may be advantageous to make another move. This is due to the dilemma between exploitation and exploration. Exploitation refers to choosing the move, that the player thinks is best, while exploration tries to expand the knowledge of the player by choosing a move, that may lead to some states which the player is not familiar with. Exploitation will maximize the expected reward in the short run, while a certain amount of exploration will be beneficial in the long run, since the player may discover that some moves were better, than previously expected. However, too much exploitation will lead to the player consistently choosing sub-optimal moves, leading to decreased long-term performance.

2.5 Conclusion

Backgammon is a game with simple rules, but with some game mechanics, that make it a very complex game. This complexity arises from the task of managing conflicting goals such as hitting an opponent and/or building a flexible position versus avoiding blots. The high branching factor also contributes to the complexity by making it difficult to consider all the possible scenarios more than two or three steps into the future. Another factor is the dimensionality of the problem. A backgammon board can be fully characterized by an array of 24 integers, encoding the number of checker on each board location, plus information about checkers on the bar. 24 is already a considerable number of dimensions, but this relatively compact encoding is highly non-linear. For instance, there is a very large difference in the tactical value between positions with a single checker, and positions with two checkers, while the difference is much less between four checkers and five. As will be shown in chapter 5, TDGammon overcomes this problem by specifically encoding, whether there is a blot, but this only serves to increase the dimensionality of the problem.

The fact, that backgammon is a game with partial knowledge of the dynamics makes it possible to reduce some of this complexity. Specifically, it is not necessary for the agent to have any knowledge of dice rolls; by presenting only afterstates to the agents, move decisions can be made without any loss of performance. Another decision, that can be made easier through knowledge of the game mechanics, is that of exploration versus exploitation. Backgammon already provides a sort of exploration mechanism through the dice rolls. The dice ensure, that it is very unlikely for the same two players to experience the exact same sequence of board states twice, even though their policies have not changed. Therefore, although the optimal degree of exploration is difficult to determine, a practical rule is for the framework to disregard any further exploration, and just choose the exploiting move, i.e. the move that has received the highest score from the agent.

Chapter 3

Reinforcement Learning

3.1 Introduction

Many machine learning techniques require examples of correct behaviour, that are to be imitated and extrapolated. Reinforcement learning[SB98] is a set of learning methods, that do not require such examples. Instead, the idea behind reinforcement learning is to learn by interaction with an environment, using feedback to adjust the choice of actions. Section 3.2 will explain the basic reinforcement learning problem. Section 3.3 describes temporal difference learning, which is a very succesful reinforcement learning method, and finally section 3.4 describes how function approximators such as artificial neural networks can be used in reinforcement learning to cope with large state spaces.

3.2 The reinforcement learning problem

In reinforcement learning, the focus is on learning through interaction. The learner and decisionmaker is called the agent, and it interacts with the environment, which comprises everything outside the agent. The interaction is cyclic: The environment is in a certain state, leading the agent to take an action, which changes the state of the environment, leading to new actions and so on. The environment also responds to the actions of an agent with a scalar reward signal, according to the action taken. A full specification of the environment with reward and state transition functions is called a reinforcement learning task.

A natural formulation of the reward function in the domain of backgammon is to award zero points for moves, that do not either win or lose, 1, 2 or 3 points for moves, that lead directly to winning the game, and -1, -2 or -3 points for moves after which the opponent directly wins the game. Once an opponent has been specified, the state transitition function is also easily defined: The opponent acts as the environment, making changes to the board according to his own policy.

The agent and the environment can interact at each of a sequence of discrete time steps $t = 0, 1, 2, ...$. At each time step t, the agent receives some representation of the environment state, $s_t \in S$, where S is the set of states of the environment. In response to the received state representation, the agent takes an action $a_t \in A$, where A is the set of actions available to the agent. At time $t + 1$ the agent then receives a scalar reward r_{t+1}, and finds itself in a new state s_{t+1}. r_{t+1} and s_{t+1} are both partially dependent on s_t and a_t. This interaction is shown on figure 3.1. At each time step t, the agent implements a function $\pi_t(a, s) = Pr(a_t = a | s_t = s)$.

This function is called the agents policy.

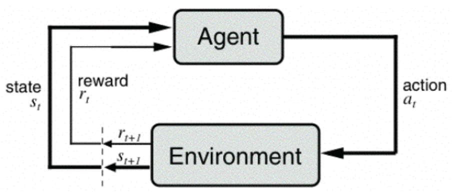

Figure 3.1: The interaction between an agent and its environment[SB98]

3.2.1 The Return

Reinforcement learning is concerned with changing the policy, such that the expected value of some aggregation of the reward sequence, starting at time t, is maximized. This aggregation is called the return, and is denoted R_t. There are a number of ways to define the return. A natural way could be to define the return as

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T = \sum_{k=1}^{T} r_{t+k}, \tag{3.1}$$

where T is a final time step. This makes sense in tasks that have a natural notion of a final time step, such as games, where winning or loosing the game causes a transition to a terminal state s_T. Such tasks are called episodic tasks. For tasks, that have no terminal state, such as an ongoing control process, simply adding the received reward could cause the return to grow unbounded. Therefore, the return for non-episodic tasks are often defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{3.2}$$

This is called the discounted return, and the parameter $\gamma \in [0,1]$ is called the discounting rate. Adjusting the γ parameter corresponds to adjusting how the agent weighs the different rewards in the sequence. If γ is 0, then the agent considers only the reward from the next action. As γ increases, more weight is put on rewards farther in the future. When γ is 1, the agent considers all rewards equally, but this requires, that the sum of rewards is bounded. The return for the episodic task can also be represented as a special case of the return for the non-episodic task, using a γ of 1, and with the convention, that terminal states transition to themselves, yielding rewards of 0, as exemplified in figure 3.2.

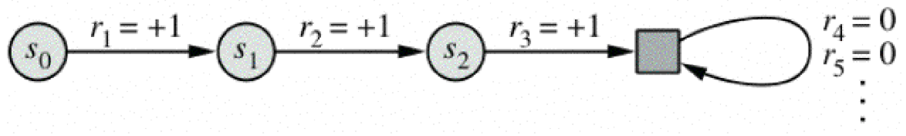

Figure 3.2: Episodic task with terminal state transitions[SB98]

The backgammon task is a good example of an episodic task. The game has a clear ending, after which point no more rewards can be earned, and the reward is only non-zero just when the game ends. Therefore the total reward is always bounded, and there is no need for a discount. In fact, using a discount would not be good, since the first states encountered would receive more reward from winning a short game, than a long game, although the only goal is to win the game, no matter the lenght.

3.2.2 Markov Decision Processes

Reinforcement learning theory presupposes, that the environment has the markov property. This property can be stated as the requirement that the probability of entering a state s and receiving reward r at time $t+1$ is only dependent on the state and action at time t, or

$$Pr(s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \cdots, r_1, s_0, a_0) = Pr(s_{t+1} = s, r_{t+1} = r | s_t, a_t). \tag{3.3}$$

The markov property is required, since the policy is a function of only the current state. If the environment does not have the markov property, then the policy can not be strictly trusted upon as an instrument to achieve the maximum return. This means that the state signal must include all the information necessary to reliably predict the next state and reward. An example could be the case of a backgammon game, where a representation of the board would provide all relevant information about the state of the game. A learning task, to which the markov property applies is called a markov decision process, or MDP. If the set of states and the set of actions are finite, then the task is called a finite MDP. Finite MDPs are defined by the state and action sets, and the single step dynamics. These include the transition probability

$$P^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a), \tag{3.4}$$

and the expected reward

$$\Re^a_{ss'} = E\{r_{t+1} | s_{t+1} = s', s_t = s, a_t = a\}. \tag{3.5}$$

3.2.3 Value Functions

Reinforcement learning algorithms changes the policy of an agent by estimating value functions. There are two kinds of value functions of interest: The state-value function, and the action-value function. The state-value function V^π for policy π, indicates the expected return when starting in a given state and following policy π

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, \tag{3.6}$$

where E_π is taken to mean the expected value when following policy π. The action-value function for policy π, Q^π indicates the expected value when starting in a given state, taking a given action and then following policy π

$$Q^\pi(s,a) = E_\pi\{R_t | s_t = s, a_t = a\} = E_\pi \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a. \tag{3.7}$$

Such value functions can be approximated through experience. As an example, in an episodic task, the value of a state s for a given policy π can be approximated as the average return gained from starting a number of times in s and following policy π. This form of reinforcement learning is called a Monte Carlo method. For most tasks, it makes more sense to estimate the action-value function, since that makes it easier to find the best action to take in a given state. However, for the backgammon domain, it is possible to take advantage of the after-states, discussed in chapter 2, to find the best move using only a state-value function. This move can be defined as $\arg\max_{s \in S_L} V^{\pi}(s)$, where S_L is the set of all legal moves.

Value functions impose a partial ordering on the set of possible policies. A policy π is better than another policy π', if $V^{\pi}(s) > V^{\pi'}(s)$ for all $s \in S$. There is always at least one policy that is better than or equal to all other policies. This can be seen by considering that two policies π' and π'', for which there is no ordering relation, can be combined to create a new policy π''', that is better than both, by letting $\pi(s,a)''' = \arg\max_{\pi \in \{\pi', \pi''\}} V^{\pi}, \forall s$. A policy, that is better than or equal to all other policies is called an optimal policy, and is denoted π^*. Such an optimal policy in turn gives rise to an optimal state-value function V^*

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{3.8}$$

and an optimal action-value function Q^*

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{3.9}$$

3.2.4 Generalized policy Iteration

As previously stated, the value function for a given policy can be estimated through experience. At the same time, each value function determines an optimal policy with regard to that value function. This suggests, that reinforcement learning can be seen as two concurrent processes. In the first process, the agent has a value function V, and a policy π, that is optimal with regard to V. Through experience, V is updated towards the true value function for policy π, V^{π}. Updating the value function may lead to the policy π no longer being the optimal policy with regard to V. Thus, the second process updates π towards the optimal policy for the updated V. If the first process causes π to be suboptimal with regard to V, then the second process will strictly improve π, bringing it closer to an optimal policy π^*. If the second process causes π to improve, then the first process will drive V towards V^*. Therefore, the only stable point for these processes are, when the value function is the optimal value function, and the policy is the optimal policy. The situation is shown in figure 3.3.

This concept is called the generalized policy iteration, and is used in almost every reinforcement learning method. The methods can differ in how long they allow each process to run, before changing to the other process. In the reinforcement learning method known as dynamic programming, which requires knowledge of $P^a_{ss'}$ and of $\Re^a_{ss'}$, V is updated fully towards V^* before the policy is updated. In other methods, the two processes may be much more intermingled, with each update of V being followed by an update of π.

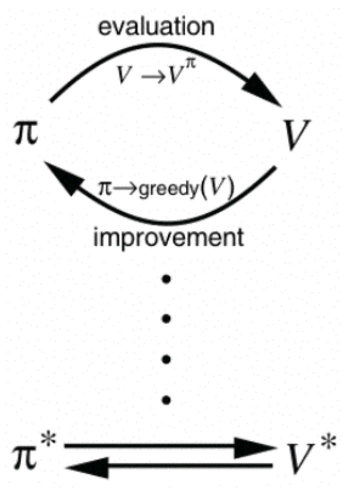

Figure 3.3: Generalized policy iteration: policy and value function interact, until they are consistent at the optimal value function and policy[SB98]

3.2.5 Action Selection

It is important to note, that generalized policy iteration can only converge to the optimal policy and value functions, if every state/action pair is visited repeatedly, until the functions reach optimality. This raises precisely the question of exploration versus exploitation, already discussed in chapter 2. A natural choice in the backgammon domain would be to disregard any exploration other than what is already imposed by the random dice rolls, but there are other options, that make even more explorative moves. These are known as soft-max policy functions. Such a function gives higher probability to the best actions, but gives a non-zero probability to even the worst action. A common soft-max policy, that uses an action-value function is the Boltzmann distribution:

$$\pi_{Boltzmann}(s, a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a)/\tau}} \tag{3.10}$$

Here, τ is a positive constant called the temperature, which influences the amount of exploration. As $\tau \longrightarrow \infty$, the actions become almost equiprobable, causing greater exploration. As $\tau \longrightarrow 0$, the policy goes toward the greedy policy, causing greater exploitation. It can be difficult to judge the amount of exploitation at a given temperature. Therefore, another soft-max policy commonly used is the simpler ϵ-greedy policy, which almost always chooses the greedy action, but with a small percentage ϵ chooses evenly among all the available actions. This makes it simpler to control the amount of exploration being performed, but has the downside, that exploration is just as likely to choose the worst action as the second-best action, which, depending on the environment, can be undesirable.

3.3 Temporal Difference Learning

This section describes temporal difference learning methods, which is a subset of reinforcement learning methods. Temporal difference methods also use the generalized policy iteration, and rely on the fact that the value function V^π of a policy π can be expressed recursively. Using the definition of the state-value function given in equation 3.6 yields:

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} && (3.11)\\
&= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} && (3.12)\\
&= E_{\pi}\{r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} && (3.13)\\
&= E_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\} && (3.14)
\end{aligned}$$

Temporal difference methods use this recursive relationship to make updates to the value function, according to the following update formula called the TD(0) algorithm

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{3.15}$$

where $\alpha \in]0, 1]$ is called the stepsize, or the learning rate. The formula can be intuitively understood by realizing that $V(s_t)$ and $r_{t+1} + \gamma V(s_{t+1})$ are actually both estimates of the value function at time t. The only difference is, that while $V(s_t)$ is based on an estimate of all the future rewards, $r_{t+1} + \gamma V(s_{t+1})$ includes knowledge of the first of these rewards. It therefore makes sense to trust the latter estimate more than the first, since they are otherwise the same. This trust is expressed by updating the value function by an amount (positively) proportional to the difference $(r_{t+1} + \gamma V(s_{t+1})) - V(s_t)$, which is called the temporal difference error. The value of V at $s = s_t$ is adjusted in the direction of $r_{t+1} + \gamma V(s_{t+1})$, which is therefore called the target of the update.

Note however, that the target should not be trusted too much. After all, in general the reward is only a probabilistic function of s_t, a_t and s_{t+1}, and s_{t+1} itself is a probabilistic function of s_t and a_t, so the reward just received could be very atypical. This is the reason, that the stepsize should not be 1. In fact, updating according to this rule will cause V to converge to v^{π} in the mean only if the stepsize is sufficiently small, and with probability 1 only if the stepsize α decreases with time[SB98].

3.3.1 On-policy and Off-policy Algorithms

Using TD(0), it is possible to define a complete reinforcement learning control algorithm, based on temporal difference methods and action value functions:

```
• Initialize $Q(s, a)$ arbitrarily
• Repeat for each episode
    – Initialize $s$
    – Choose $a$ from $s$, using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    – Repeat for each step of episode
        ∗ take action $a$, observe reward $r$ and next state $s'$
        ∗ Choose action $a'$ from $s'$, using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        ∗ $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        ∗ $s \leftarrow s'$, $a \leftarrow a'$
```

This algorithm is called sarsa, because it uses all the elements $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ that define a transition from one state to the next. Sarsa will converge to the optimal value function on two conditions. First, every state/action pair must be visited recurringly. This can be achieved with a softmax policy such as ϵ-greedy. The second convergence criterium is that the policy must converge in the limit to the greedy policy. This can be achieved for the ϵ-greedy policy by letting $\epsilon = \frac{1}{t}$. Sarsa is an on-policy algorithm, because the action suggested by the policy is used both to transition to another state and to update the action-value function. This is in contrast to an off-policy algorithm such as the Q-learning algorithm:

```
• Initialize $Q(s,a)$ arbitrarily
• Repeat for each episode
    – Initialize $s$
    – Repeat for each step of episode
        ∗ Choose $a$ from $s$, using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        ∗ take action $a$, observe reward $r$ and next state $s'$
        ∗ $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        ∗ $s \leftarrow s'$
```

Here, the policy is responsible for which state/action pairs are updated, but not for the value with which they are updated. The only criterium for convergence of Q towards Q^* is therefore that every state/action pair is visited recurringly.

The difference between an on-policy and an off-policy algorithm is best shown by example. The example task is walking on the edge of a cliff, getting to the other end as fast as possible. The task is episodic, stopping when the other end of the cliff has been reached. To encourage getting to the end as fast as possible, the reward for each step is -1, while the reward for falling off the cliff is -100. The state set is a grid as shown in figure 3.4, and the actions are up, down, right and left. Every action leads deterministically to the next state in the indicated direction, except for except for any action, that goes over the cliff edge, which leads back to the start state S. The state set contains a single terminating state, marked with G. Figure 3.5 shows the performance of sarsa and Q-learning algorithms over the task, using ϵ-greedy algorithms with a fixed ϵ of 0,1. The sarsa algorithm learns the safe path, yielding higher results than the Q-learning algorithm, which actually learns the optimal path, but has lower performance than sarsa due to the softmax policy. Of course, if ϵ converged towards 0, then the performance of both algorithms would approach the optimum.

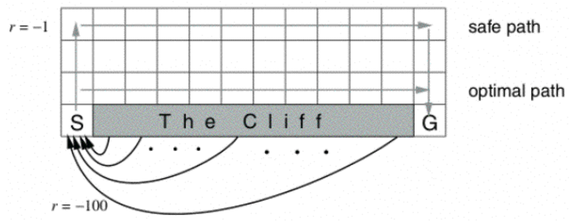

Figure 3.4: A cliff walking task[SB98]

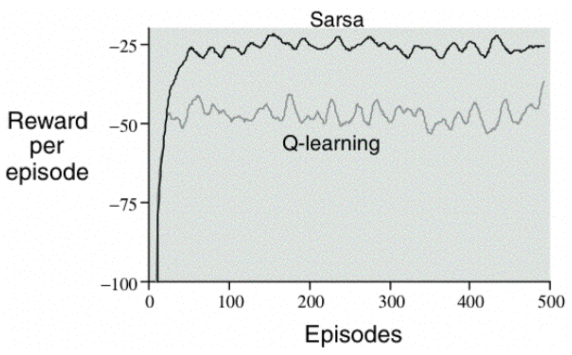

Figure 3.5: Performance of sarsa and Q-learning on the cliff walking task[SB98]

When estimating a state-value function by using afterstates, instead of an action-value function, such as is the case in the backgammon domain, the reinforcement learning policy is always on-policy, since there are no actions to maximize over, as in the Q-learning algorithm. Instead, the algorithm for learning a state-value function is a variation of the sarsa algorithm:

```
• Initialize $V(s)$ arbitrarily
• Repeat for each episode
    – Initialize $s$
    – Repeat for each step of episode
        * Choose an afterstate $s_{after}$ from $s$, using policy derived from $V$ (e.g. $\epsilon$-greedy)
        * take any action leading to $s_{after}$, observe reward $r$ and next state $s'$
        * $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
        * $s \leftarrow s'$
```

3.3.2 The TD(λ) Algorithm

This section describes an abstraction of the update rule given in equation 3.15. In this update rule, recall that the quantity $r_{t+1} + \gamma V(s_{t+1})$ is called the target of the update, and in this case it is called a 1-step target, denoted R_t^1, since it is the expected return R_t evaluated at time $t+1$. However, it is possible to update according to other targets as well. The recursive definition of V^π given in equation 3.11 can be expanded as

$$V^\pi(s) = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s\} = E_\pi\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^\pi(s_{t+2})|s_t = s\} \quad (3.16)$$

leading to another update formula

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma r_{t+1}\gamma^2 V(s_{t+2} - V(s_t))) \quad (3.17)$$

which uses the 2-step target $R_t^2 = r_{t+1} + \gamma r_{t+1}\gamma^2 V(s_{t+2})$. In general, update rules can use any n-step target R_t^n, for $n \in 1, 2, \cdots$. It is also possible have a combined target, that is the weighted sum of several different targets. The only requirement is that the weights of the different targets must sum to 1. For instance, a target could be a combination of the

1-step and 2-step targets, such that the update rule would be $V(s_t) = V(s_t) + \alpha(R_t^{combined} - V(s_t)))$, where $R_t^{combined} = \frac{1}{2}R_t^1 + \frac{1}{2}R_t^2$. This idea of combining targets give rise to the TD(λ) algorithm, for $\lambda \in [0, 1]$, which updates according to the target R_t^λ which is a weighted mixture of infinitely many targets:

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^n \tag{3.18}$$

The $(1-\lambda)$ factor is prepended to the infinite sum to make the weights sum to 1. Updates made according to the R_t^λ target take into account a weighted sum of all the different estimates of the return received after time t. The function of the λ parameter is to control how these estimated returns are weighted. The TD(0) method, shown in equation 3.15 only uses the R_t^1 target, but for $\lambda > 0$ the other return estimates gain more weight, and as λ goes to 1, the returns are given almost equal weight.

The purpose of updating according to an infinite mixture of targets is to learn more efficiently. In principle, the idea may have some merit. But in practice, it may seem quite inefficient: The target can only be fully evaluated in the context of an episodic task, only when the task is finished, and until then it is necessary to remember all the received rewards. However, equation 3.18 can be transformed to an incremental update rule, using a mechanism known as eligibility traces. The idea that visiting a state makes it eligible to receive updates from all the future rewards. The weight of each of these rewards in the R_t^λ target can be calculated, so when a reward is received, the value function can be incrementally updated. The important thing is to remember how much reinforcement a state should receive from each new reward. This weight can also be calculated incrementally, and is called the the eligibility trace of the state. The full TD(λ) algorithm is as follows:

```
• Initialize $V(s)$ arbitrarily
• Set $e(s) = 0 \forall s$
• Repeat for each episode:
    – Initialize $s$
    – Repeat for each step of episode:
        * Choose an afterstate $s_{after}$ from $s$, using policy derived from $V$ (e.g. $\epsilon$-greedy)
        * take any action leading to $s_{after}$, observe reward $r$ and next state $s'$
        * $\delta \leftarrow r + \gamma V(s') - V(s)$
        * $e(s) = e(s) + 1$
        * For all $s$:
            · $V(s) \leftarrow V(s) + \alpha\delta e(s)$
            · $e(s) = \gamma\lambda e(s)$
        * $s \leftarrow s'$
```

The best value of λ is not known, and is probably different for different domains. In backgammon, [Tes02] finds that values between 0.0 and 0.7 yield roughly the same performance, while values above 0.7 decrease the performance, and therefore chooses $\lambda = 0$, since that reduces the TD(λ) algorithm to the simpler TD(0) algorithm of section 3.3.1.

3.4 Function Approximation

In many realistic settings the set of states is infinite. Even in the case of a finite state set, the number of states may be very large, such as the set of possible configurations of chess pieces on a chess board. This makes it impossible or impractical to update the value of every state or state/action pair even once, which violates the assumption in the generalized policy iteration. The only way to learn on these tasks is to generalize from previously experienced states to ones that have never been seen. Learning an unknown function, such as a state-value function, using only examples of this function is called supervised learning, and fortunately, there are already a number of supervised learning techniques that can be used.

The update rules so far have all been about adjusting an estimate of a value function towards a given target. This target, together with the corresponding state or state/action pair can be used as examples for the function approximator. The chosen function approximator must be able to handle non-stationary target functions. This is because in generalized policy iteration seeks to learn a value function V^{π} for a policy π, while the policy changes. This also means, that the function approximator must be able to handle non-static example sets. The rest of this section presents three different function approximation techniques: Neural networks, tile coding and Kanerva coding, all of which are examples of gradient descent methods.

3.4.1 Gradient Descent Methods

The basic idea behind gradient descent methods is to represent the value function V_t as a parameterized function $V_{\vec{\theta}_t}$, where $\vec{\theta}_t$ is the parameter vector at time t. The purpose of the gradient descent is to adjust the parameter vector, so that the overall error between $V_{\vec{\theta}_t}$ and V^{π}, the true value function of the current policy, is made as small as possible. More concretely, the purpose of gradient descent is to minimize an error function defined as mean squared error between the true value function V^{π} and the approximation V_t, over some distribution P of the inputs

$$MSE(\vec{\theta}_t) = \frac{1}{2} \sum_{s \in S} P(s)(V^{\pi}(s) - V_t(s))^2 \tag{3.19}$$

P is important, because function approximators are generally not flexible enough to reduce the errors to 0 for all states. P specifies a priority of error correction for the function approximator. P is often also the distribution, from which the states of the training examples are drawn, since that makes the function approximator train on the same distribution of examples as the one for which it tries to minimize the error. If the distribution describes the frequency with which states are encountered while the agent interacts with the environment, then it is called the on-policy distribution. On-policy algorithms such as sarsa train on states from an on-policy distribution, while off-policy algorithms do not. The distinction is important, because some environments can cause a function approximator, trained on an off-policy distribution and using gradient descent, to diverge, causing the error to grow without bound. Convergence for function approximators is still largely unexplored, but it seems to be safer to use on-policy algorithms in connection with function approximators.

Assuming that P is an on-policy distribution, a good strategy is to minimize the error over the observed examples. The mean squared error is decreased by adjusting the parameter vector in the direction of the steepest descent of the error. This leads to the following update rule for the parameter vector

$$\vec{\theta_{t+1}} = \vec{\theta_t} - \alpha \nabla_{\vec{\theta_t}} MSE(\vec{\theta_t}) \tag{3.20}$$

$$= \vec{\theta_t} - \alpha \nabla_{\vec{\theta_t}} (\frac{1}{2}(V^{\pi}(s_t) - V_t(s_t))^2) \tag{3.21}$$

$$= \vec{\theta_t} + \alpha (V^{\pi}(s_t) - V_t(s_t)) \nabla_{\vec{\theta_t}} V_t(s_t), \tag{3.22}$$

where $\alpha \in]0, 1]$ is the learning rate. Since the true value of V^{π} is unknown, it is necessary to use an approximation, e.g. the 1-step target described in section 3.3:

$$\vec{\theta_{t+1}} = \vec{\theta_t} - \alpha \nabla_{\vec{\theta_t}} (\frac{1}{2}(R_t^1 - V_t(s_t))^2) \tag{3.23}$$

$$= \vec{\theta_t} + \alpha (r_{t+1} + V(s_{t+1}) - V_t(s_t)) \nabla_{\vec{\theta_t}} V_t(s_t). \tag{3.24}$$

Generally, it is not a good idea to have a learning rate of 1. Although it would reduce the error on the current example, it would increase the error on many other examples, since the parameter space is in general not large enough to reduce the error to 0 for all examples.

3.4.1.1 Neural Networks

Artificial neural networks, or ANNs, are a type of parameterized functions, that can be trained using gradient descent. A neural network is a kind of computational structure, where simple computational units feed their results into other computational units. In neural networks, the basic unit is called a neuron, or node. The neuron takes a vector $\overline{x}$ of inputs, and computes a weighted sum net of these inputs, using a weight vector $\overline{w}$, which is part of $\vec{\theta}$, and which belongs to that neuron, so that $net = \overline{x} \bullet \overline{w}$. The neuron then produces an output o by applying an activation function to net. This is shown in figure 3.6

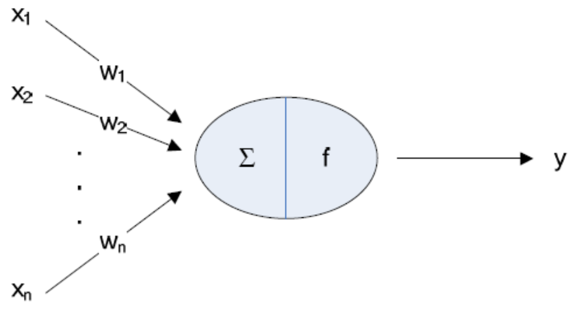

Figure 3.6: The artificial neuron[Nie06]

The characteristics of the artificial neuron are very much dependent on the choice of activation function. There are several options, but the most popular is the sigmoid function

$$o = \frac{1}{1 + e^{-net}} \tag{3.25}$$

The graph for the sigmoid function is shown in figure 3.7. This function has many desirable properties such as being differentiable, being non-linear, mapping to a bounded range, monotonicity and having an especially nice derivative.

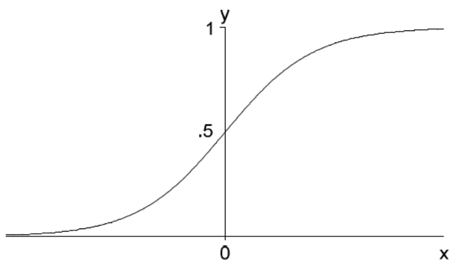

Figure 3.7: The sigmoid function[Nie06]

The nodes of an ANN are arranged in a network. There are many ways to arrange nodes, but a simple and widely used configuration is to arrange them in a layered structure, such that a node in a given layer only receives input from all the nodes in a previous layer, as shown in figure 3.8. Thus information flows forward from the input layer to output layer. This is the forward propagation of input in an ANN.

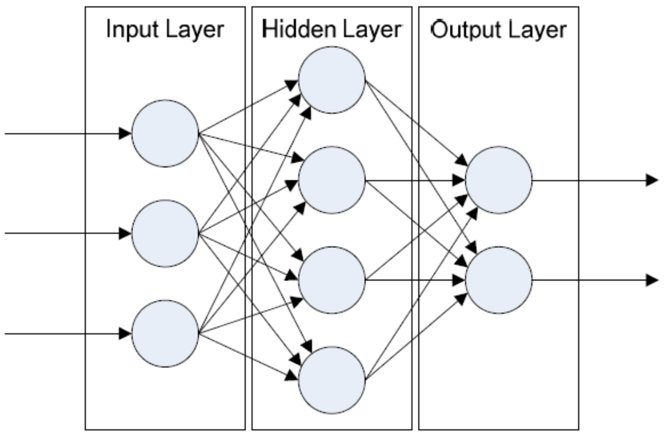

Figure 3.8: Typical structure of a neural network[Nie06]

It is common to include in the input to a node the output of an "'always-on"', or bias, node, which always has the output 1. This effectively allows the node to shift the graph of its activation right or left, according to the weight on the input of the bias node. In typical networks, the nodes of the input layer do not apply the activation function, instead they simply propagate the unchanged input values forward to the nodes in the hidden layer. In this light, the network in figure 3.8 shows a network with only two computing layers. This is not uncommon, as neural networks are able to represent a large number of functions using a very limited number of layers.

3.4.1.2 Error Backpropagation

In the case of neural networks, the error is taken over all the output units:

$$E(\vec{\theta}_t) = \frac{1}{2} \sum_{k \in outputs} (t_{k,d} - o_{k,d})^2 \tag{3.26}$$

where outputs is the set of output nodes, and $t_{k,d}$ and $o_{k,d}$ are the target and actual outputs of output layer node k respectively, for the input d. The weight updates are similar to those just described, but for neural networks it makes sense to look closer at the update formula for each individual component $w_{i,t}$ of $\vec{\theta}_t$:

$$w_i = w_i - \alpha \frac{\partial E}{\partial w_i}. \tag{3.27}$$

The first thing to notice is that a weight can only affect the error through the output of the corresponding node. Thus, using the chain rule:

$$\frac{\partial E_d}{\partial w_{i,n}} = \frac{\partial o_{n,d}}{\partial w_{i,n}} \frac{\partial E_d}{\partial o_{n,d}} \tag{3.28}$$

where $w_{i,n}$ is the i'th parameter of node n, and $o_{n,d}$ is the output of node n, using the input of example d. The chain rule allows further decomposition, by noting that net can only affect the output through the activation function:

$$\frac{\partial o_{n,d}}{\partial w_{i,n}} = \frac{\partial o_{n,d}}{\partial net} \frac{\partial net}{\partial w_{i,n}} = \frac{\partial o_{n,d}}{\partial net} x_{i,n,d} \tag{3.29}$$

where $x_{i,n,d}$ is the i'th input to node n, assuming example d. If the network uses the sigmoid activation function σ, the $\frac{\partial o_{n,d}}{\partial net}$ becomes simple, since $\frac{\partial \sigma(y)}{\partial y} = y(1 - \sigma(y))$:

$$\frac{\partial o_{n,d}}{\partial net} = \frac{\partial \sigma(net)}{\partial net} = o_{n,d}(1 - o_{n,d}) \tag{3.30}$$

Thus the $\frac{\partial o_{n,d}}{\partial w_{i,n}}$ term of equation 3.28 becomes:

$$\frac{\partial o_{n,d}}{\partial w_{i,n}} = x_{i,n,d} o_{n,d}(1 - o_{n,d}) \tag{3.31}$$

assuming sigmoid activation functions. The $\frac{\partial E_d}{\partial o_{n,d}}$ part of equation 3.28 varies depending on whether node n is in the output layer or not. If it is in the output layer, the calculation is straightforward, using the definitions of the error:

$$\frac{\partial E_d}{\partial o_{n,d}} == \frac{\partial \frac{1}{2} \sum_{k \in outputs} (t_{k,d} - o_{k,d})^2}{\partial o_{n,d}} = \frac{\partial \frac{1}{2} (t_{n,d} - o_{n,d})^2}{\partial o_{n,d}} = -(t_{n,d} - o_{n,d}) \tag{3.32}$$

which gives the following value of $\Delta w_{i,n}$ for a parameter in a node in the output layer:

$$\Delta w_{i,n} = -\eta \frac{\partial E_d}{\partial w_{i,n}} = -\eta \frac{\partial o_{n,d}}{\partial w_{i,n}} \frac{\partial E_d}{\partial o_{n,d}} \tag{3.33}$$

$$= \eta x_{i,n,d} o_{n,d}(1 - o_{n,d})(t_{n,d} - o_{n,d}) \tag{3.34}$$

For a node that is not in the output layer, the output of the node can only affect the error through the output of the nodes of the next layer, thus:

$$\begin{aligned}\frac{\partial E_d}{\partial o_{n,d}} &= \sum_{k \in downstream(n)} \frac{\partial o_{k,d}}{\partial o_{n,d}} \frac{\partial E_d}{\partial o_{k,d}} = \sum_{k \in downstream(n)} \frac{\partial net_{k,d}}{\partial o_{n,d}} \frac{\partial o_{k,d}}{\partial net_{k,d}} \frac{\partial E_d}{\partial o_{k,d}} \\ &= \sum_{k \in downstream(n)} w_{n,k} \frac{\partial o_{k,d}}{\partial net_{k,d}} \frac{\partial E_d}{\partial o_{k,d}}\end{aligned} \tag{3.35}$$

where $downstream(n)$ is the set of nodes that receive input directly from node n and $w_{n,k}$ is the weight from node n to node k. This gives the following value of $\Delta w_{i,n}$ for a parameter in a node in a hidden layer:

$$\Delta w_{i,n} = -\eta \frac{\partial E_d}{\partial w_{i,n}} = -\eta \frac{\partial o_{n,d}}{\partial w_{i,n}} \frac{\partial E_d}{\partial o_{n,d}} \tag{3.36}$$

$$= -\eta x_{i,n,d} o_{n,d}(1 - o_{n,d}) \sum_{k \in downstream(n)} w_{n,k} \frac{\partial o_{k,d}}{\partial net_{k,d}} \frac{\partial E_d}{\partial o_{k,d}} \tag{3.37}$$

This recursive definition relies on the error gradients of the following layer, and is what gives the backpropagation algorithm its name, since the error signal is propagated backwards from the output layer.

3.4.2 Tile Coding

The neural networks described above are effective and widely used, partly because of their flexibility: Neural networks are able to approximate any bounded continuous function to within an arbitrarily small error margin, using only single hidden layer[Mit97]. But this flexibility comes at a price. Although in practice they have worked very well, there is no general convergence proof for sigmoidal neural networks. Linear methods, on the other hand, do have convergence proofs, and can be very efficient both in terms of data representation and computation. These features make them very attractive. In linear methods, a state s is represented using an n-dimensional feature vector $\vec{\phi}_s$. The value function is parameterized by a n-dimensional weight vector $\vec{\theta}_t$, and the state value estimate is computed as

$$V_t(s) = \vec{\theta}_t \vec{\phi}_s = \sum_{i=1}^{n} \vec{\theta}_t(i) \vec{\phi}_s(i) \tag{3.38}$$

where the arguments for the vectors denote their individual components. A central issue in linear methods is how the features of the state should be encoded. It is of course important, that the encoding capture all the features that are needed to generalize properly. This requirement is the same as with neural networks. However, the linear nature of the computation means that it is impossible to take into account interactions between the features, such as feature a only being good in the absence of feature b. Therefore, for linear methods, it is necessary that the feature vector also represents all relevant combinations of features.

One very efficient form of feature encoding is known as CMAC[1] or tile coding[SB98]. In tile coding, the input space is exhaustively partitioned into tiles. Each tile corresponds to a feature of the state vector. As an example, take a state space, where each state is represented

[1]CMAC: Cerebellar model articulator controller. The model was originally used to model the human brain[RB04]

by two bounded real numbers. Then the state space can be divided into the 16 tiles shown on figure 3.9.

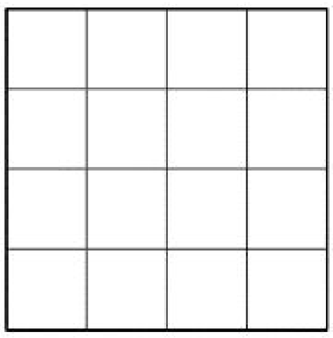

Figure 3.9: A simple tiling of a bounded 2-dimensional state space[SB98]

The tiling defines a binary encoding, such that when a state falls within a tile, the component of the feature vector is set to 1, while the other components are set to 0. One of the advantages of tile encoding is its simplicity. Given the values of the two state dimensions, and a rectangular tiling such as this, it is easy to find the single feature vector component, that is set to 1, and the output of the the value function is just the corresponding component of the weight vector. The resolution of the approximation can be improved by adding new tilings, as shown on figure 3.10. These tilings are offset from each other and from the state space. The feature vector is the combination of the feature vectors due to each of tilings, and the value function output is just the sum of the weight the two relevant weight components. Broader generalization can be achieved by adding new tilings with larger tiles.

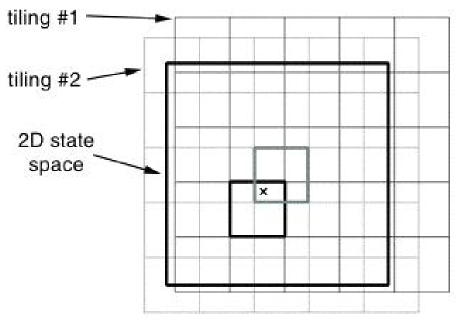

Figure 3.10: Multiple tilings increase resolution[SB98]

Although tile coding is computationally inexpensive on a "‘per tile"’ basis, tile coding suffers from the requirement to encode every relevant feature combination. This means that in the worst case, the number of tiles necessary for adequate performance grows exponentially with the number of dimensions in the pure state signal. Tile codings have been used successfully in experiments[RB04][Sut96], encoding up to 14 dimensions, but state signals with over a few tens of dimensions are usually too large to be handled by tile coding.

3.4.3 Kanerva Coding

Kanerva coding[Kan93] is an attempt to expand some of the general ideas of linear methods to very large state spaces. Consider a large state space, spanning hundreds of dimensions. In the worst case, the complexity of the target function grows exponentially with the number of dimensions, and no linear method will be able to represent the target function accurately. However, the complexity of the target function does not necessarily grow exponentially. Consider for instance a small state space, where the target function is modelled accurately. Adding new sensors will increase the dimension of the state space, but it will not increase the complexity of the problem. The problem with CMAC is, that the complexity of the tiling does increase exponentially with the state space, while the complexity of the function remains the same.

Kanerva coding solves the problem by considering features, that are unaffected by the dimensionality of the state space. These binary features correspond to particular prototype states. These prototype states are a set of states, randomly chosen from the entire state space. A prototype state activates, when an observed state is in relative proximity of the state, using some metric. For instance, in a binary state space, the metric could be hamming distance between the states. The output of the value function is then the sum of the components of the weight vector $\vec{\theta}_t$, corresponding to the active states.

Given the fact, that Kanerva coding uses random features, it is perhaps surprising that it has worked quite well in a number of applications[KH01][SW93]. Kanerva coding does not reduce the essential complexity of complex problems, but it can remove some accidental complexity of simpler problems.

3.5 Conclusion

Reinforcement learning is a set of very general methods for learning from interaction with an environment. Backgammon is a good candidate to learn using reinforcement learning, since it is a Markov process. Learning backgammon can be stated as an episodic, undiscounted reinforcement learning task, with rewards being awarded only when the game ends. Although it is usually the best idea to estimate an action-value function, the backgammon game mechanics makes it natural to estimate the state-value function instead, using a variation of the sarsa algorithm. Another effect of the backgammon game mechanics, is that there arguably is no need to induce any exploration by using a softmax policy, since the dice rolls themselves already provide exploration. TD(0) algorithms update the value function based on the 1-step target. The TD(λ) algorithm provides a generalization, but in the backgammon domain a value of $\lambda = 0$ seems not to reduce performance. In many application, the set of states is often so large, that the value functions must be implemented by function approximators, using the update target as training example. Of these, neural networks provide the ability to accurately model any continuous, bounded function, while linear methods such as CMAC and Kanerva coding come with convergence guarantees for on-policy algorithms.

Chapter 4

Related Work

4.1 Introduction

At this point, the basic framework for the backgammon agent has been decided upon, and the field of reinforcement learning has been introduced, but many questions still remain in the area of supervised reinforcement learning: What should be the relationship between the learner and the teacher, in what ways can the teacher influence the learner, are the two kinds of learning even compatible, or will they work against each other? In order to answer such questions, this chapter presents some of the previous work done in the field of supervised reinforcement learning. Section 4.2 shows, that the two types of learning are indeed theoretically compatible, and shows some measures that can be taken, for this still to be the case in practice. Section 4.3 presents the RATLE framework that allows composite advice to be given in a stylized, human-readable language, while section 4.4 has a good overview of the ways in which the teacher can influence the actions of the learner, and presents a flexible way to interpolate between the actions of the learner and the supervisor. Section 4.5 shows how testing strategies differ between several previous papers on supervised Learning. Finally, section 4.6 describes how the task of avoiding the initial period of bad performance in TDGammon by using supervised reinforcement learning can be classified in relation to the previous work.

4.2 Two Kinds of Training Information

The of supervised reinforcement learning has two separate roots, that have later joined. The first root concerns the notion of a machine learner, that may change its behavior based on advice given during execution. The idea of a program learning from external advice was first proposed [MSK96] in 1959 by John McCarthy [McC59], the inventor of Lisp. The other root concerns that of reinforcement learning, which originated [Sut88] with the Checker program by Samuel [Sam95] also in 1959. However, the fusion between the two fields is much younger. It came about in 1991, when Utgoff and Clouse published "‘Two Kinds of Training Information for Evaluation Function Learning"’[UC91][1].

Utgoff and Clouse make the observation, that there exist two fundamental sources of training information: Future payoff achieved by taking actions according to a given policy from a given state, and the advice from an expert regarding which action to take next. Training methods relying on the future payoff are called temporal difference methods, while methods relying on

[1]Benbrahim [Ben96] also combined the two ideas at the same time, independent of Utgoff and Clouse, but Utgoff and Clouse provided a clearer exposition

expert advice are called state preference methods. State preference methods are so called, because it the goal of the learner is not necessarily to mimic the exact values assigned by the expert to a state. Instead, the goal for the learner is to have the same preference as the expert, when presented with a set of possible states. Thus, the only thing that matters, is that the sign of the slope of the learners evaluation function between two states be the same as that of the experts. This means, that there is generally infinitely many functions, that produce the same control decisions as the expert evaluation function, making state preference methods very flexible with regard to incorporating other types of learning information.

Utgoff and Clouse also make the key observation is that temporal difference methods and state preference methods are orthogonal. Using the general model of an evaluation function as a parameterized function of the state, state preference methods attempt to change the parameters of the evaluation function so as to obtain the same slope as the expert between the values of the possible next states, while temporal difference methods are concerned with obtaining the correct value for the sequence of actions experienced by following the current policy. In other words, training information in state preference methods is horizontal in the game tree, while it training information for temporal difference methods. And since the state preference methods are so flexible, it is possible to change the parameters of the evaluation function to incorporate both types of information.

In practice, however, the compatibility between the two classes of methods may be less than perfect. A state preference and a temporal difference method will be in conflict to the degree, that the expert is fallible. There is therefore a need for a rule to determine how much to trust the expert. Another concern could be, that the expert may not always be available. This could be the case for instance with a human expert. Utgoff and Clouse therefore implement a heuristic, after which the learner should only ask for teacher input, when the state is poorly modeled, as indicated by a large update to the parameter of the utility function resulting from the temporal difference method.

4.3 The RATLE Framework

In 1996, Maclin and Shawlik create the RATLE[2] framework[MSK96]. The RATLE framework is focused on having a human teacher, or supervisor. Such a teacher may not be available at all times, and in any case, would probably not be able to answer questions about which actions to take, at the same pace that these questions could be posed by the learner. One of the design decisions of RATLE is therefore, that it is the teacher, and not the learner, that decides when advice must be given. In addition, it is possible to give general tactical advice, instead of suggestions of the right action to take in a specific situation. This maximizes the effectiveness of the teacher, who will most probably think in terms of these tactics anyway. Finally, the advice is given in a stylized, but human-readable language, which uses domain-specific terms, logical connectives and vague quantifiers such as "‘big"’ or "‘near"’ to capture the intent of the teacher. Examples of such a piece of advice can be seen in figure 4.1, which shows from left to right the stylized advice, an english interpretation and a figurative explanation of the strategy implied.
RATLE uses neural networks[3] to implement the state evaluation function, and implement the new advice by first transforming the advice to a set of hidden nodes, and then integrating

[2]the Reinforcement and Advice Taking Learning Environment
[3]Neural networks will be explained in section 3.4

```
WHEN Surrounded ∧
       OKtoPushEast ∧
       An Enemy IS Near
REPEAT
       MULTIACTION
          PushEast
          MoveEast
       END
UNTIL ¬ OKtoPushEast ∨
       ¬ Surrounded
END;
```

When the agent is surrounded, pushing east is possible, and an enemy is near, then keep pushing (moving the obstacle out of the way) and moving east until there is nothing more to push or the agent is no longer surrounded.

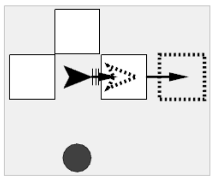

Figure 4.1: Composite advice in the RATLE framework[MSK96]

them into the already existing network, as shown in figure 4.2.

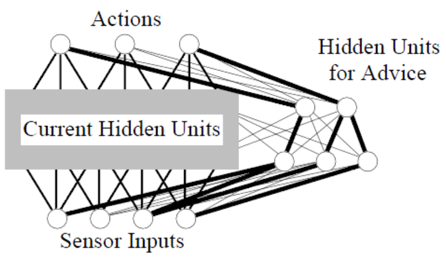

Figure 4.2: Incorporating advice into the evaluation function[MSK96]

4.4 Supervised Actor-Critic Reinforcement Learning

In 2004, Rosenstein and Barto[RB04] adapted the actor-critic model[SB98] for supervised reinforcement learning. Briefly, the actor-critic model is a general model for reinforcement learning, where an actor makes decisions about which actions to take, while the critic learns a utility function of the states by reinforcement learning, and critizes the actor on the actions it chooses, based on this utility function, causing the actor to update its policy. One of the advantages of the actor-critic model, is that it separates decisions of which action to take, from the task of construction the correct utility function. Based on this separation, Rosenstein and Barto identified 3 ways, in which a supervisor can influence the actions of a reinforcement learner. Figure 4.3 shows the 3 ways, which are value function shaping for the critic, exploratory advice for the actor and direct control, in which the supervisor supremely chooses the actions.

One of the contributions of Rosenstein and Barto is a framework, that flexibly interpolates between the actor and the supervisor. The general structure is shown in figure 4.4. The output of both the actor and supervisor is assumed to be a one-dimensional scalar variable. These are fed into the gain scheduler unit, which computes the final output of the composite actor as a weighted average of the actor and supervisor outputs. The model does not specify, how the weights are obtained. For instance, the interpolation can be controlled by the actor, letting it seek explorative advice from the supervisor, or it can be controlled by the supervisor, which can use it to ensure a minimum level of performance.

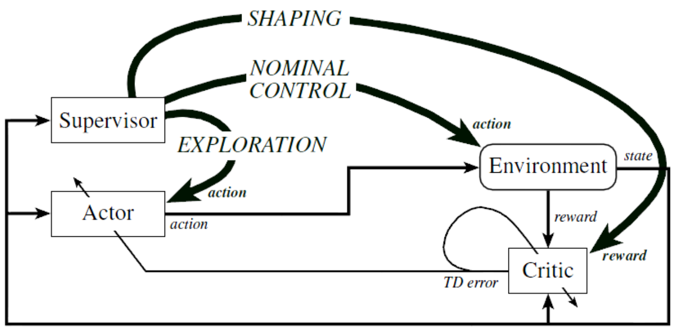

Figure 4.3: Supervised reinforcement learning using an actor-critic architecture[RB04]

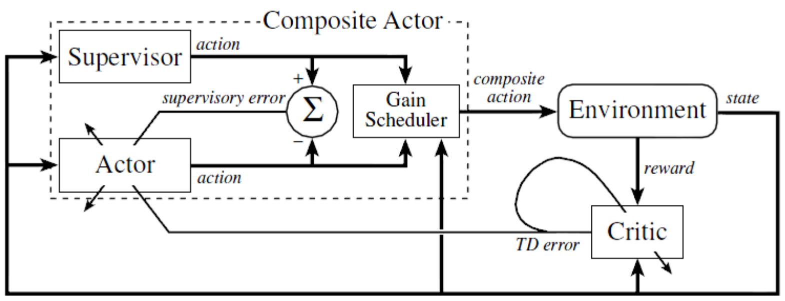

Figure 4.4: Action interpolation between actor and supervisor[RB04]

4.5 Testing

Testing the effect of combining reinforcement learning with online supervision is obviously important in determining its usefulness. The majority of the papers, that deal with supervised reinforcement learning have a section devoted to testing, and there are some important similarities, as well as some distinct differences in the their testing approaches.

- Baseline presence: Most of the tests report their progress using a baseline result, that must be improved upon, however some tests in [RB04] are reported as-is, without any baseline, with the only measure of success being, that a task was accomplished at all.
- Type of baseline: For the tests, that use a baseline result, the baseline is typically an agent trained by reinforcement learning without any supervision[Cet08][Cet06]. In other cases, however, the supervisor acts as the baseline[RB04].
- Purpose: The purpose of some tests is to arrive at a higher level of performance, than that achieved by the baseline[MSK96], while other tests aim to reduce the cumulative error during the entire training period, compared to some perfect performance[RB04].
- Type of testbed: Most of the tests are done on a custom testbed, developed specifically for the method being presented. This is natural, since the many of the methods are aimed at solving specific problems, which may not appear on a standard testbed. However the lack of a standard testbed also prevents the comparison of many methods, that could naturally be compared, and so there are some attempts at creating a standard testing environment. Most notably, the RoboCup environment[KAK$^+$95], which specifies several tasks in connection with creating simulated soccer players, seems to be used as a standard test reference for a number of papers[Cet08][KH01].

4.6 Conclusion

This chapter has presented several axes, along which a supervised reinforcement learning task may vary, and it is now possible to place the task of improving TDGammon performance along these axes.

- Presence of baseline: Since the purpose is to improve the results of TDGammon, it is natural to use TDGammon as a baseline.
- Type of baseline: TDGammon was trained using pure reinforcement learning.
- Purpose: Since the initial problem is to improve upon the early period of bad results in the training of TDGammon, the purpose of the test will be to maximize cumulative improvement over the baseline, rather than to increase the final level of improvement. Given that the final version of TDGammon reached a status, where it has become one of the top players in the world, it is doubtful, that any known supervisory help could increase the final level of performance by any significant degree.
- Learner/supervisor relationship: The supervisor will be a computer program, that has a reasonable level of performance. Since it is not a human, there is no practical restriction on when the supervisor can provide advice.
- Supervisor influence: The supervisor can influence the learner through an action interpolation mechanism similar to the one discussed in section 4.4. That model was developed for actors and supervisors with an output range of the real numbers, while the output of TDGammon is a move choice in the set of legal backgammon states, which is discrete and unordered, so the model of section 4.4 will have to be adapted. This adaptation is the subject of sections 6.2 and 6.3.
- Type of testbed: The test must be domain-specific, since the purpose is to test performance improvement against the pure reinforcement learning of TDGammon.

In the domain of backgammon, there are some semistandard testbeds, that can be adapted to the testing required in this report. This adaptation is the topic of the following chapter.

Chapter 5

Test Definition

5.1 Introduction

In the last chapter it was established that there was a need for a custom testbed, using the performance of TDGammon after various numbers of training episodes as the baseline. This definition leaves the questions of what is the nature of the performance, and how much can these performance measures be trusted. It is the purpose of this chapter is to answer these questions. There are several versions of TDGammon, and so the first priority, addressed in section 5.2, is to establish which version is to be used. A natural performance measure of TDGammon can be as the rate of victories against a benchmark opponent. Chapter 5.3 discusses the choice of this benchmark opponent, chooses a mid-level backgammon player called Pubeval, and shows the Pubeval evaluation strategy, which is very simple. Section 5.4 discusses, how many games should be played, before a reasonable degree of confidence in the performance is obtained.

5.2 TD-Gammon

TD-Gammon[Tes02] is a backgammon program created by Gerry Tesauro. There are several versions, which were developed during the early to mid-nineties. The first version, TDGammon version 1.0, was developed as an experiment to test the ability of reinforcement learning to create a backgammon player of reasonable playing strength. Tesauro had previously created the Neurogammon player, whose input included both raw information about the board, as well as various handcrafted expert features, and it was implemented as a neural network using backpropagation, trained on a set of recorded expert move decisions. Neurogammon had previously won the 1989 computer olympiad [Tes92], but TDGammon 1.0 was able to achieve roughly the same playing strength, using only a very basic encoding of the board as input. The following versions of TDGammon had modifications such as a larger network, longer training times, 3-ply[1] search, and an input, that was expanded to include the handcrafted features of Neurogammon. Using these modifications, TDGammon 3.1 achieved a playing strength that makes it arguably the strongest backgammon player in the world.

Unfortunately, the handcrafted features of the later versions of TDGammon are not publicly available, so these versions cannot be used for the testing baseline. This leaves only version 1.0. TDGammon 1.0 was implemented using a single neural network. The encoding used

[1]A ply is a single turn by one player

players checkers on point	encoding
0	0000
1	0001
2	0010
3	0100
4 or more	1000

Table 5.1: TDGammon input encoding

for the input mapped each of the 24 points of the board to 4 contiguous units for each player[Tes92]. The encoding is shown in table 5.1.
Using this encoding, the checkers on each point for both players are encoded in 8 units. The clear separation in the encoding between a blot and a blocking point makes it easier to take into account the non-linear influence that the number of checkers on a point has on the tactical assesment of that point, as discussed in section 2.5. I addition to the raw board encoding, 2 units encode the number of checkers on the bar for each player, and 2 units units encode the men off board, for a total of 196 input units.
The output format of TDGammon is an array of 4 doubles, encoding the probability of a normal win, a gammon win, a normal loss and a gammon loss respectively[2]. Due to the rarity of the occurence, backgammon wins are not considered. The utility of a board is then evaluated as the sum of the points received due to the different outcomes, weighted by the estimated probabilities of these outcomes. The network was trained using the TD(λ) algorithm, using a λ of 0.7 and a constant learning rate of 0.1.

5.3 The Benchmark Opponent

This section describes the opponent that will be used to test the performance of both the baseline player TDGammon, as well as the players trained using supervised reinforcement learning. Some desirable traits of such a player are the following:

- availability: The player should be reasonably easy to come by, also for others, enabling them to repeat the experiment.
- modifiability: It should be easy to modify the player to fit into the testing framework.
- universality: The player should be a standard opponent used in other papers. This allows for better comparison of results.
- speed: With higher speed more tests can be performed.

There are several open-source backgammon players, that are very available on the internet. Of these, the Pubeval player[Tes08] is highly modifiable, very fast and has been used as a benchmark opponent in [PB98], [SALL00] and [Hei04]. Therefore, the pubeval player is chosen as the fixed test opponent.

Pubeval is a very simple backgammon program created by Gerry Tesauro. It plays at a medium level, and takes as input a raw encoding of the board state, which is very similar to TDGammon. Pubeval includes two weight vectors, both as long as the input vector, for the

[2]No attempt is made to make the estimated probabilities sum to 1.

contact and race phase. It computes the utility of a board by computing the dot product of the input vector with the appropriate weight vector. Unfortunately, Pubeval does not include the use of the doubling cube, so the test can only measure performance where none of the players are able to double the stakes.

5.4 Measuring Performance

At this point, the baseline player and the test opponent has been found. The structure of the test is to let Pubeval play a number of games against the baseline and the backgammon players being tested, at various stages of their training. Such a number of games can be called a trial, and the strength of a player at a given stage is indicated by the percentage of games won in the trial at that stage. It still remains to be determined what these stages of training are, and how many games should be played at each trial. Beginning at the end, there are two conflicting goals that must be addressed. Having a lower number of games per trial allows for finer granularity of the test, and for more tests to be run. Having a higher number of games per trial allows for greater confidence in the performance of the player measured by each trial.

The trial length used in other papers, that use pubeval as a benchmark, varies greatly, depending on the context. [PB98] use a trial length of 200 games in a hill-climbing strategy, where several players are maintained simultaneously. [SALL00] uses a trial length of 5000 games to track the performance of a single player during training, and [Tes98] uses 10000 games to evaluate the final performance achieved by a player. So it seems, that a length of 5000 games is reasonable, but a length of 10000 should be chosen, if the performance of a single player is particularly important. This seems to be consistent with figure 5.1, which shows the results of 30 different trials of the baseline TDGammon implementation, each having a length of 5000 games. As can be seen, the performance is reasonably stable, but nonetheless there is a difference of about 3 points between the best and the worst measured performance.

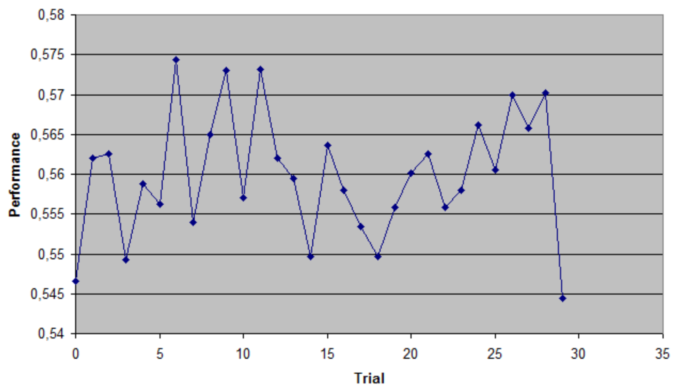

Figure 5.1: Baseline performance measured in 30 trials

With the trial length decided, it was easier to decide the test granularity, and the length of training. Measuring performance every 1000 games and stopping training after 100000 games meant a total of 101 trials, which took an average of 15 hours. Using these parameters, figure 5.2 shows the performance of the baseline. Note, that TDGammon implementations are usually trained for more than 100000 episodes, but in this case 100000 training episodes were deemed to be enough for two reasons: The need to measure the performance at various stages of training, and the fact that one of the primary purposes of this thesis is to improve the initial performance of TDGammon. However, bear in mind that the performance has likely not settled yet after 100000 training episodes. The shape of the performance graph shows the problem clearly. There is an initial period of poor performance, which lasts for approximately 3000 training episodes, at which point the performance starts to rise. Sharply at first, but then with steadily decreasing slope. At certain points, the performance even seems to decrease, for instance just around 90000 training episodes, although that could also just be a particularly bad measurement, since the two measurements on either side are very similar.

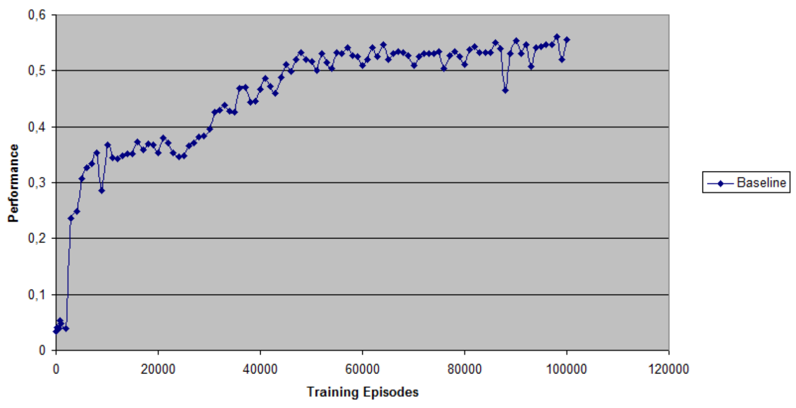

Figure 5.2: The baseline test result

5.5 Conclusion

This chapter has defined a testbed, that can be used to measure the performance of adding supervision to the reinforcement learning process. The testbed has been defined by using relatively common components, e.g. the baseline and the benchmark opponent, and by setting parameters specific to the purpose, e.g. the relatively short training time.

Chapter 6

Adapting the Supervised Actor-Critic Learning Model

6.1 Introduction

The time has finally come to design and implement an agent that learns to play backgammon using supervised reinforcement learning. During the course of the project, other things have been implemented as well, most notably the backgammon framework outlined in chapter 2, that makes it possible to train and test such a player, but these are only tangentially related to the topic of supervised reinforcement learning. Appendix A details the implementation of an n-ply-search functionality, but other than that, the framework will not be mentioned further.

Before going into the design, it is useful to sum up the knowledge gained over the previous chapters. Supervised reinforcement learning requires a supervisor, and a supervisee. The supervisee will be called the agent. Quite a lot is known about the agent:

- The agent will be trained using reinforcement learning as described in chapter 3
- In order to have the best comparison against the performance of TDGammon, which is the test baseline, the agent will receive the same type of input, and must return the same type of output, as TDGammon. The input and output format of TDGammon is described in chapter 5.
- Basically, the agent will be a copy of TDGammon, and will therefore also be implemented using neural networks.

Many details of the supervisor are as yet unknown. The supervisor should not learn anything while the agent is being trained, only provide a reasonable level of play. This level must not be too low, since then there would no point in having a supervisor, and not too high, because then it might be better to just let the supervisor play, instead of learning anything. One thing, that is known from chapter 4 about the supervisor, is that the interpolation between the supervisor and agent actions, and later the learning arising from those actions, should be similar to that presented in [RB04], and recapped in section 4.4, so section 6.2 takes a closer look at that model. The model has to be adapted, in order to fit the backgammon domain. This adaptation is the topic of section 6.3. One of most important points of [RB04] is that the interpolation is decided by a state dependent variable k, and so, having defined the supervisor in section 6.3, section 6.4 defines 3 models of supervised reinforcement learning for backgammon, based on 3 different ways to compute k.

6.2 The Supervised Actor-Critic Reinforcement Learning Model

The model of learning in [RB04] is called supervised actor-critic reinforcement learning, since it uses an actor-critic model to implement the reinforcement learning. The structure was presented in section 4.4, but for convenience the model is also presented here, in figure 6.1.

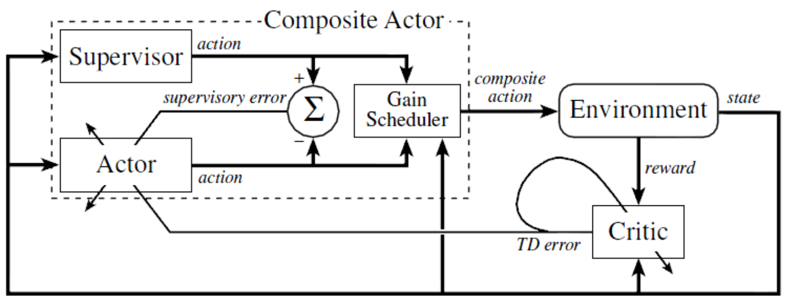

Figure 6.1: The structure of the supervised actor-critic[RB04]

The critic implements an ordinary state-value function, using TD(0). The critic computes the TD error $\delta = r_{t+1} + \gamma V(s_{t+1} - v(s_t)$, which it uses to update both its own state-value estimates, as well as the actor policy.

The gain scheduler combines the actions of the actor and the supervisor according to an interpolation parameter k. Since actions are assumed to be scalars, the composite action is just a weighted sum of the two actions:

$$a = ka^E + (1-k)a^S \tag{6.1}$$

Where a^E is the actors exploratory action, and a^S is the supervisor action, according to policies π^E and π^S respectively. The actor also has a greedy policy π^A, and the exploratory policy is really just the greedy policy with an added gaussian noise with zero mean. Since the k value plays an important role in choosing the action, it is only natural, that it also plays a similar role in adjusting the policy of the actor according to the reward received. Assuming that π^A is a parameterized function with parameter vector w, the equations for updating the actor policy are:

$$w \leftarrow w + k\Delta w^R L + (1-k)\Delta w^S L \tag{6.2}$$
$$\Delta w^R L = \alpha\delta(a^E - a^A)\nabla_w \pi^A(s) \tag{6.3}$$
$$\Delta w^S L = \alpha(a^S - a^A)\nabla_w \pi^A(s) \tag{6.4}$$

Where α is a learning rate, and δ is the TD error from the critic. The effect of equation 6.3is to move $\pi^A(s)$ closer to $\pi^E(s)$, when the reward for the exploratory action was higher than expected, leading to a positive TD error, and further away from it, when the TD error is negative. Equation 6.4 is very similar to the supervised updates discussed in section 3.4. The effect is to move $\pi^A(s)$ closer to $\pi^S(s)$, regardless of the reward received.

6.3 Adapting the Model to Backgammon

The model described in the following chapter is very nice, but there are some aspects of it, that make it difficult import the model into the backgammon domain. It all has to do with the nature of actions in backgammon. Actions in backgammon are more or less just new backgammon states, whereas the model assumes scalar actions. Backgammon states are discrete and there is no natural ordering of the states. This makes it impossible for the combined state of the gain scheduler to be a weighted sum of the supervisor and agent actions, and it makes it impossible to subtract actions from each other or to create an exploratory policy, that is a noisy version of the greedy policy.

Rather than try to copy the exact equations, it is useful to try to find action interpolation and policy update rules, that capture the intentions of the model. The action interpolation is the easiest, since a solution for discrete actions is suggested in [RB04]. That solution is to interpret the k value as the probability for the gain scheduler to choose the agent action, instead of the supervisor action. This is a reasonable discrete approximation of the smooth mixing of the two actions. For the equations 6.3 and 6.4, their intent is to change the policy according to experience and according to the advice from the supervisor. The policy of a reinforcement learning backgammon agent is implemented by the state-value function, so the above intent translates to moving the value function in the direction of the received reward, and moving the value function in the direction of the supervisor value function.

This puts quite a restriction on the supervisor: Instead of just emitting actions based on an unknown policy, implemented in an unknown way, the supervisor is now required to expose a state evaluation format, that is the same as that of the agent - a vector of 4 doubles representing the probabilities of the different outcomes of the game. Under such strict restrictions, there are not many choices for a supervisor implementation. A fitting supervisor is therefore a version of TDGammon, which stopped training early. The chosen supervisor was trained for 34500 games, after which it won approximately 34% of the games against pubeval.

The agent implements the state-value function as a neural network, so the update rule of 6.2 is not the most natural form. Although the two forms are functionally the same, it is more natural to define a target output value for the state, and let the backpropagation take care of the rest. By using the above interpretations of the intent of the supervised actor-critic model updates, it is possible to specify a combined target for the value function update as

$$\begin{aligned} combinedtarget_t &= k(s_t)(r_{t+1} + V(s_{t+1})) + (1 - k(s_t))(output_{supervisor}(s_t)) && (6.5)\\ &= k(s_t)V(s_{t+1}) + (1 - k(s_t))(output_{supervisor}(s_t)) && (6.6) \end{aligned}$$

In the backgammon domain, the reward is zero for all state transitions except the last. This is modeled by removing the reward term, and letting V_{T+1}, where T is the time of the final state seen by the agent, equal one of the four reward vectors 1000, 0100, 0010 or 0001, depending on whether the agent won normally, won a gammon, lost normally or lost a gammon. For faster learning, the training only started, when a game had been completed, and updates to the estimate of the value of a state were done starting with the last state. In this way, some of the reward for a game propagated down to the first state seen already during the training immediately after that game.

6.4 K-values

The supervised actor-critic model was put through a number of different tests[RB04]. These tests required an implementation of a function that would provide a state-dependent value of k, to be used for the actor policy update, and the gain scheduler action interpolation. This function had two requirements:

- Visiting a state should raise the value of k for that state.
- Not visiting a state for some time, should cause the k-value for that state to slowly drop.

The first requirement is due to the fact, that visiting a state will increase the knowledge of the agent about that state, so it can be trusted more in its decisions regarding that state. The second requirement is due to the fact, that the actor policy was implemented as a parameterized function. Since parameterized functions have to use the freedoms provided by the parameter space to ensure good performance over the observed samples, not visiting a state for some time will generally have the effect of increasing the expected error of the policy, when compared to an optimal policy. Therefore, the supervisor should be trusted more in these cases.

These requirements were fulfilled using a variation of tile encoding, which was presented in section 3.4. The test used 25 tilings over a 2-dimensional input space, but the weights associated with each tile were not updated according to any gradient descent method. Instead, the weights of visited tiles were increased by a small amount, and after each episode, all weights were multiplied by a factor of 0.999.

Unfortunately, the state space of backgammon is so large, that tile coding schemes become very impractical. The state signal for a backgammon board, used by both TDGammon and the agent is a 196-dimensional vector. Although most of these dimensions are binary, the state space simply too large. In addition, tile coding schemes do not work very well with binary dimensions. They are better suited to dividing up continuous dimensions into a number of smaller areas. Therefore, the tile coding scheme cannot be used for backgammon, and so some other way of providing a state dependent k-value, which fulfills the requirements must be found.

6.4.1 Kanerva Coding

The reason, why tile coding of the k function worked well for the supervised actor-critic model was, that there was a very simple relationship between the weights and the output. That made it easy implement a gradually decreasing k-value for states, that had not been visited for some time. As shown in section 3.4, the same simple relationship between weights and output is also present in the Kanerva coding method, so a version of the agent was implemented, which used a k function based on Kanerva coding.

Designing the Kanerva-based k-function required answering questions such as: What is the nature of the prototype space, how should the prototypes be generated, and how many prototypes are required. An obvious answer to the first question is to use a 196-dimensional prototype space, corresponding to the input space for the agent, and using the hamming distance as metric. However, the input for the agent was designed to make it easy for the

encoding	interpretation
000	0 checkers
001	1 opponent checker
010	2 opponent checkers
011	3 or more opponent checkers
100	1 agent checker
101	2 agent checkers
110	3 agent checkers
111	4 or more agent checkers

Table 6.1: Kanerva prototype encoding

agent to recognize the non-linear effect of the input on the output. If the goal is simply to be able to distinguish the different states from each other, then a more compact representation of the state, which encodes every point in only 3 bits, can be used. This is shown in table 6.1. Since a lot of the work in a Kanerva coding scheme goes into finding the distance between state represention, a more compact encoding will result in a performance speedup.

Generating prototypes is easy, when the different dimensions are independent: For each dimension in the prototype, generate a random value within the set of legal values for that dimension. This approach cannot be used in backgammon, because the sets of legal values are not independent of each other. For instance, within the 24 dimensions corresponding to the points of the backgammon board, there cannot be 4 values of 111, since that would indicate a board with 16 or more agent checkers, which would be illegal. To overcome this problem, another method of generating the prototypes is used: A series of games is set up with two players, that choose moves randomly. Any sample state from any one of these games will be a legal board, that can be translated into a legal prototype. It is a desirable property of the set of prototypes, that they cover a lot of different states, i.e. the prototypes should not be too similar. This means that the samples should not be from early parts of the game, where the states will be similar for all games, samples from the same game should not come from positions in the game sequence, that are too close, and since random games tend toward the same type of positions, with most of the checkers in the opponents home table, samples should not come from too late in the game. Under these guidelines, every eigth of the first 64 board states in a game were used to produce prototypes.

To answer the question of how many prototypes to generate, it is necessary to weigh the increased resolution achieved by larger numbers of prototypes against the reduced computational load from smaller numbers of prototypes. The problem is, that the time spent computing a k-value increases linearly with the number of prototypes, so a large number of prototypes would make a single game take very long. In the end, samples were taken from 1000 games, since that resulted in a reasonable game speed. That yields a rather small number of prototypes compared to the size of the prototype space, so each prototype must generalize to many other states in order to be useful. In order to achieve a high degree of generalization, the set of prototypes activated by a state s is assumed to be the top 5% most similar prototypes, where similarity is the number of dimensions with the same encoding.

6.4.2 Neural Networks

Another way to implement the k function could be to use neural networks. After all, they have proved to be valuable in TDGammon, which also computes a function over backgammon boards, and a simple way to raise the value of visited states could be to provide a target of 1, and then do backpropagation. The problem arises with the requirement, that the state value should decrease for states, that have not been visited for a while. The naive way to implement this would be to go over all non-visited states, training them with a target of 0. However, the number of unvisited states in a simple game is extremely large, so that is not a feasible solution.

What is the effect of not doing any effort to fulfill the requirement to gradually decrease values? The frequently visited states will be trained to full autonomy for the agent, while the effect on the unvisited states will be more difficult to predict. This may not be a problem, if none of the infrequently visited states are encountered during a normal game, which by definition is unlikely to happen. Therefore, it seems reasonable to implement the k-value by initializing a neural network to produce a value close to 0 for all inputs, and then training the output for visited states on a target of 1. The network can be initialized to produce (close to) 0 for all states by setting all connection weights to 0, and all neuron biases to -20.

6.4.3 State Independent Interpolation

When the requirement, that the k-value of infrequently visited states should decrease, is dropped, the values for all states will rise toward 1 with varying speeds. For the set of states frequently visited states during a normal game, those speeds may be roughly similar. This suggests a further simplification: To simply disregard the specific state, and only vary the k-value as a non-decreasing function of the number of training episodes. A simple example could be a trapezoid function, that is zero for all training episodes below some threshold, rising linearly to 1 between that threshold and another, and 1 for training episodes above the second threshold, as shown in figure 6.2.

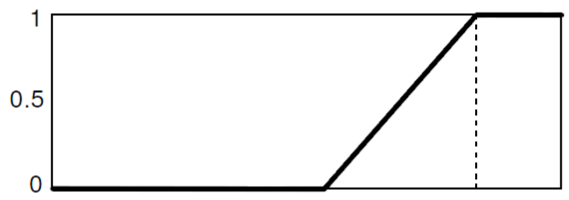

Figure 6.2: A trapezoid function

Such a function raises an interesting question: What should be the positions of the left and right thresholds? Looking at the performance of the baseline shown in chapter 5, it seems clear, that the agent should be in full control, when the performance of the agent trained without supervision reaches that of the supervisor. Otherwise, the supervisor may actually hurt the combined performance. In this case, the performance of the baseline, which is just the agent trained without supervision, shows, that the right threshold should be at roughly at 5000 training episodes. However, in some situations, there may not be an estimate of the unsupervised agent performance. For the left threshold, it is not completely clear, whether it is better to leave the supervisor in total control for a while, or whether control should start to transfer to the agent immediately.

6.5 Conclusion

In summary of this chapter:

- The agent is implemented as a neural network structure.
- The combined action is either the agent or the supervisor action, partially dependent on k.
- The network is trained on a weighted sum of the reinforcement target and the supervisor target, with the weights dependent on k.
- 3 different k function have been proposed: Kanerva based, neural network based and trapezoid.

It remains to choose a few parameters:

- Value of λ in TD(λ): 0. [Tes92] states, that performance was similar for values in the range [0.0 , 0.7], and a value of 0 is computationally simpler.
- Number of hidden units: 80. This is the same as some versions of TDGammon.
- Learning rate: 0.1. This is the same as in TDGammon[Tes92].

There are now 3 slightly different designs of supervised reinforcement learning in backgammon ready to be tested.

Chapter 7

Test

7.1 Introduction

The testing process was defined in chapter 5. Briefly, each agent is to be trained for 100000 training episodes[1]. The performance of the agent is then to be measured every 1000 training episodes, and compared to the baseline.

7.2 Test Results

The results for the 3 designs are shown in figures 7.1, 7.2 and 7.3. It seems clear, that the agents using interpolation based on either Kanerva coding or neural networks do not improve upon the performance of the supervisor. In fact, the results seem to indicate, that the interpolation for both these agents must be heavily favoring the supervisor throughout the training. There are a couple of reasons, why this might be so. For the neural network interpolation, the network was initialized to a value of nearly 0 for all states by setting all weights to 0, and the bias to -20. It is not normal to initialize any weights or biases in a neural network with such a numerically large value, since the gradient for the sigmoid activation function, evaluated at -20 is very small, and thus during training, where the weight update is dependent on the gradient, the weight increments were too small to have any effect during the course of the 100000 training episodes.

For the Kanerva-based interpolation, a could be a problem, that the number of prototypes, which was approximately 8000, were not sufficient to capture the complexity of the state space. Therefore, with the decision of always activating the closest 5% of the prototypes, prototypes would be randomly activated. With the random activation, the weight increases due to state visits would spread evenly among all the protypes. Since none of the prototypes would consistently receive weight increases, the weight decreasing factor of 0.999, that was applied to all weights, might have been enough to keep all of the prototype weights at a low enough level, that the supervisor could dominate the interpolation.

In contrast to the both the Kanerva-based and neural network-based interpolation, the simple trapezoid interpolation worked reasonably well. It was able to avoid the initial bad performance of the baseline, although the performance did fall a bit before the learning really started to work. Another problem is, that although the performance was initially better than the baseline,

[1]interpreted here as equivalent to 50000 self-play games, since the agent will be trained on both the winning and losing board sequence.

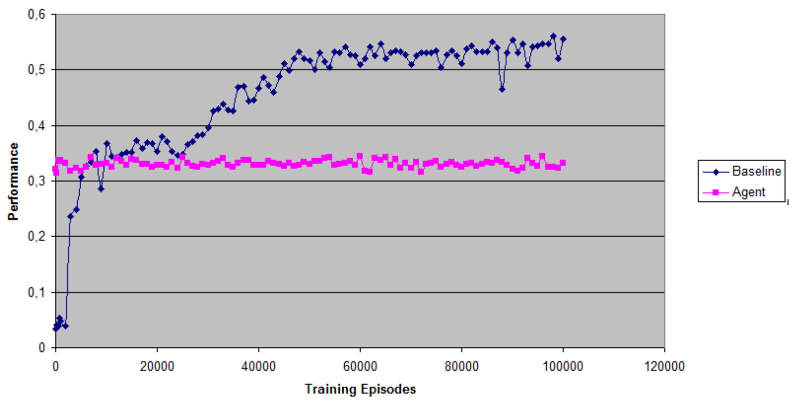

Figure 7.1: Performance of supervised reinforcement learning using Kanerva-based interpolation

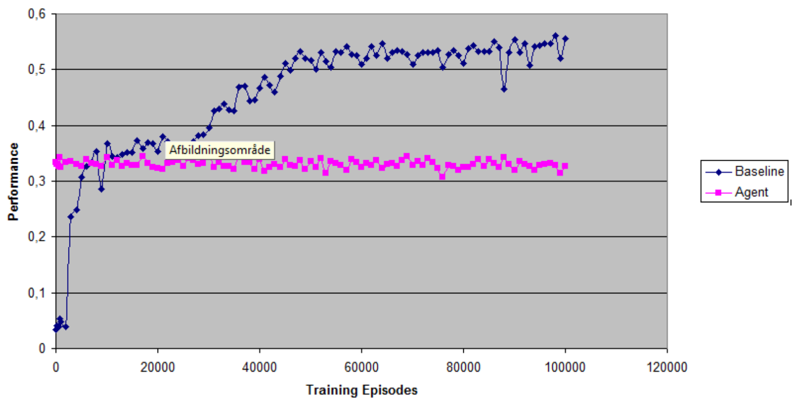

Figure 7.2: Performance of supervised reinforcement learning using neural network-based interpolation

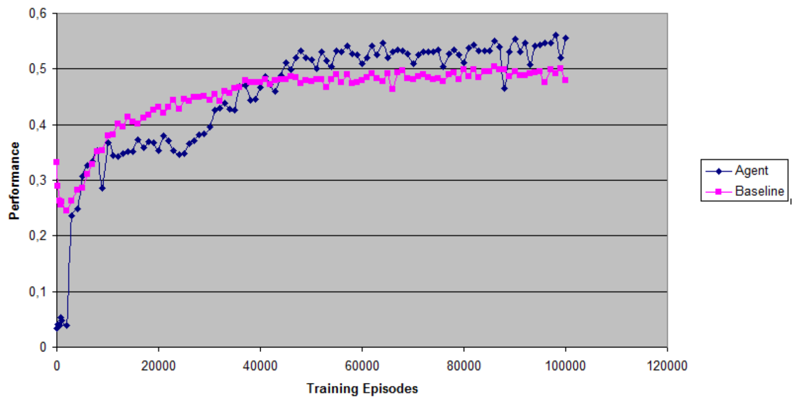

Figure 7.3: Performance of supervised reinforcement learning using trapezoid interpolation, left threshold 0, right threshold 5000

the baseline performance overtook the agent performance at around 45000 training episodes. This is not easily explainable, since at that point the agent should be fully autonomous, and therefore use the same learning mechanism as the baseline. A final problem with the state-independent trapezoid interpolation is, that it is not clear how to choose the shape of the interpolation function. The effect of choosing a very different interpolation function is shown on figure 7.4, which also uses a trapezoid interpolation, but with the right threshold moved to 50000 instead of 5000 training episodes. This interpolation ensures, that the initial drop in performance seen in the shorter trapezoid interpolation does not occur. The price paid for this is that the baseline overtakes the agent much earlier, and that the agent stays at the supervisor performance much longer.

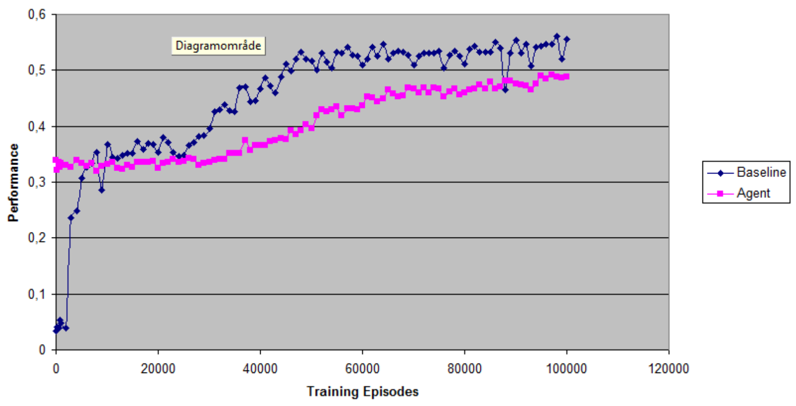

Figure 7.4: Performance of supervised reinforcement learning using trapezoid interpolation, left threshold 0, right threshold 5000

7.3 Conclusion

This chapter has presented the results of testing the agents designed previously. It has shown, that it is possible to apply supervision to the reinforcement learning process in order to avoid the initial period of bad performance. In addition, it has shown the importance of increasing agent autonomy at the right pace. If the agent has control too soon, the performance may drop initially, since the supervisor has not had enough time to teach. If the supervisor remains in control for too long, performance will suffer in the longer term, since the agent is not allowed to learn from experience, and thereby surpass the performance of its supervisor.

Some possible improvements for the agents suggest themselves:

- The neural network interpolation could probably be improved by also setting the bias to 0. That would initialize the interpolation to 0.5 for all states. Then output in the range [0.5 - 1] could be mapped to [0 - 1]. In this way the network would be much more responsive to training.

- If another supervisor was chosen, with higher performance, it may be that a trapezoid interpolation function could be found, that allowed the agent enough time to train to avoid the initial performance drop, and at the same time released control to the agent quickly enough, that the baseline did not overtake it.

Chapter 8

Conclusion

The backgammon domain is very well suited to pure reinforcement learning: It has a natural formulation as a reinforcement learning task, it has a built-in exploration mechanism, and can take advantage of the presence of afterstates to estimate a state-value function instead of the more complicated action-value function. As a result, several very good backgammon players, such as TDGammon have been implemented using reinforcement learning.

Reinforcement learning has the drawback, that it may take a long time to reach an good level of performance, and that when the training starts, the performance may be unacceptably low. This report has developed a framework for supervised reinforcement learning in the backgammon domain, based on an interpolation of the agent and supervisor, in order to reduce those drawbacks.

The backgammon domain is not as perfect a fit for supervised reinforcement learning using the interpolation model, as it is for pure reinforcement learning. This is primarily due to two aspects of the domain:

1. The input has a high dimensionality. This has meant, that the interpolation function needed to be adapted.
2. The actions are discrete and unordered. This has meant that the learning function needed to be adapted.

Adapting the interpolation function was made easier by the realization, that in the domain of backgammon, there is a relatively small part of the state space, that can reasonably be expected to occur in a normal game. For these states, the agent should be steadily more autonomous, since these are the states, that the agent gets to know well. Using a kind of circular logic, since these are the states, that are likely to be seen in a normal game, the agent should be granted more autonomy for all states it encounters. This simplification allowed the implementation of 3 different interpolation functions. Both the function based on Kanerva coding, and the function based on neural networks did not do so well. Their performance remained at the level of the supervisor, and performed worse than the baseline for most of the test. This is probably for different reasons. The Kanerva function only had around 8000 prototypes, and given the complexity of the input space, this was probably not enough. The neural network function may have been initialized so that it required more training than included in the test to reach autonomy for the agent. In contrast, the trapezoid interpolation function removed the initial period of bad performance, while still being able to improve as a result of the reinforcement learning. However, a negative effect of the supervision seems to be a slower overall learning rate, and a bad choice of thresholds of the interpolation function can reduce performance.

Adapting the learning function resulted in the requirement that the supervisor should have the same type of output as the learner. This has also worked well, but is a quite harsh restriction on the supervisor. The framework presented here would be much more useful, if an arbitrary supervisor could be used.

Going forward from here, there are some simple modifications, that might increase performance:

- Using a supervisor that plays at a higher level may yield a consistently better performance than the baseline, until they both settle, instead of the baseline just catching up, when it starts to improve.
- Using a constant learning rate, as TDGammon does is not recommended in the theory. Using a diminishing learning rate might allow the performance to stabilize at a higher level, although the performance might rise slower.

These modifications are aimed at increasing performance, which is very nice, but the most benefit would come from increasing the scope of the framework. To achieve this goal, it is necessary to address one or both of the following challenges.

- Find some way to remove the supervisor output restriction.
- Find a heuristic to determine the best shape of the stateless interpolation function, or implement a good state-based interpolation function.

Appendices

Appendix A

N-ply Search

This chapter describes the n-ply search, that has been implemented in the backgammon framework. N-ply search can be thought of as mentally playing ahead of the game, instead of just looking at the current board. A ply is a single turn for one of the players, and the 'n' in n-ply refers to how many plies the search should look ahead. The branching factor for backgammon is quite high, so it is usually not feasible to look ahead more than 2 or 3 plies. The search gives rise to a tree of different boards: A single board expands to more boards, and each of these boards expand to even more boards. The 'n' of n-ply can therefore also be thought of as the depth of this search tree.

As an example, consider the expansion to depth 1 associated with 1-ply search, in a game between players A and B. In order for player A to obtain the score of one of the possible moves, recall that such a move represents the state of the backgammon game, after it has been player A has finished his turn, if he chooses to make that move. So now it is player B's turn to move. Since player A doesn't know, which dice B will roll, he has to try out all combinations of them. There are 21 different combinations (since permutations give rise to the same possible moves for the opponent), and each of these combinations present B with a choice of which move to make, using those dice. player A doesn't know how B chooses moves, so he has to try to guess. The guess is, that B will choose the move, that A would choose, without looking ahead, only with the sides reversed: A guesses that B chooses the move, that A thinks is the worst move. So now A has for each dice combination a move, that he thinks, that B will make using those dice, and has assigned a value to each of those moves. Thus, the original move is associated with 21 different scores, which can be aggregated into the single 1-ply search score. However, these 21 boards are not all equally likely to occur. Specifically, dice combinations (and their associated scores) that are not doubles, are twice as likely to occur as dice combinations, that are doubles. Therefore, the 1-ply search score of the original board is a weighted average of the expanded board scores.

Deeper expansion is easy. For instance, boards can be expanded to depth 2 by further expansion of each of the 21 boards selected by the opponent, for a total of 441 weighted scores. The weights are modified, so that scores obtained from two successive double rolls are given weight 1, scores obtained from a combination of double and non-double rolls are given weight 2, and scores obtained from two successive non-double rolls are given weight 4. It is worth noting, that the value function used for choosing boards at each step of the expansion is the same, regardless of search depth or current expansion step. This has the effect that the original moves are evaluated using an n-depth search, but the expansion mechanism itself uses only zero-depth search, for each expansion step. . Obviously, it is not possible to use n-depth search for all expansion steps, because then the search would never finish (or at least not until the end of the game had been reached, which would take an exceedingly long time, considering that each ply adds a branching factor of 21), but it is conceivable that expansion step 1 used (n-1)-ply search, expansion step 2 used (n-2)-ply search and so on. This is not implemented for performance reasons.

Finally, once each of the possible original moves of A have a score, it is time to choose the best move. This is dependent on the search depth. The score for each move represents the probable value of choosing that move as the next move, as perceived by the player, who, at the n'th expansion step, supplied the weighted scores, which have been aggregated to form the final move score. Therefore, if the depth is even, the current player has supplied the scores, and the best move is the move with the highest score, and if the score is odd, the opponent has supplied the score, and therefore the best move is the move with the lowest score.

This n-ply search scheme was implemented after a design by Gerald Tesauro, outlined in [Tes02]. Note, that the semantics of this report differ from those of Tesauro, in that what he describes as 1-ply search, is equivalent to 0-ply search in the context of this thesis. Using n-ply search is a very good way to improve the performance of an agent, after it has been trained. It is not good to evaluate states by any other search than 0-ply search, since using just 1-ply search will cause an agent to spend roughly 400 times more time on choosing a move. However, once an agent has been trained, its performance can be increased quite dramatically by using 1-ply search. As an example, the test baseline evaluated after 100000 training episodes, using 0-ply search won 55.7 percent of the games in a 5000 game trial, while the same baseline using 1-ply search won 65,8 percent of the games in a 400 game trial. Using 2-ply search is usually only practical for games versus humans, since just making a single move may take on the order of 10 seconds, according to [Tes02]. The tests in this thesis were conducted using only 0-ply search.

Bibliography

[Ben96] Hamid Benbrahim. *Biped dynamic walking using reinforcement learning*. PhD thesis, Durham, NH, USA, 1996. Director-Miller,III, W. Thomas.

[CB96] Robert Crites and Andrew Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1017–1023. MIT Press, 1996.

[Cet06] Victor Uc Cetina. A multiagent architecture for concurrent reinforcement learning. In *ESANN*, pages 107–112, 2006.

[Cet08] Victor Uc Cetina. Autonomous agent learning using an actor-critic algorithm and behavior models. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1353–1356, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[Hei04] Mikael Heinze. Intelligent game agents, developing an adaptive fuzzy controlled backgammon agent. Master's thesis, Aalborg University Esbjerg, Esbjerg, Denmark, 2004.

[KAK$^+$95] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative, 1995.

[Kan93] Pentti Kanerva. Sparse distributed memory and related models. In *Associative Neural Memories*, pages 50–76. Oxford University Press, 1993.

[KH01] Kostas Kostiadis and Huosheng Hu. Kabage-rl: Kanerva-based generalisation and reinforcement learning for possession football. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS*, 2001.

[McC59] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[MSK96] Richard Maclin, Jude W. Shavlik, and Pack Kaelbling. Creating advice-taking reinforcement learners. In *Machine Learning*, pages 251–281, 1996.

[Neu08] Neurondotnet, 2008.

[Nie06] Niels Nygaard Nielsen. Intelligent game agents, improving a fuzzy controller based backgammon player. Master's thesis, Aalborg University Esbjerg, Esbjerg, Denmark, 2006.

[PB98] Jordan B. Pollack and Alan D. Blair. Co-evolution in the successful learning of backgammon strategy. In *Machine Learning*, pages 225–240, 1998.

[RB04] Michael T. Rosenstein and Andrew G. Barto. Supervised actor-critic reinforcement learning. In *Learning and Approximate Dynamic Programming: Scaling Up to the Real World*. John Wiley & Sons, 2004.

[SALL00] Scott Sanner Sanner, John R. Anderson, Christian Lebiere, and Marsha Lovett. Achieving efficient and cognitively plausible learning in backgammon. In *In Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000*, pages 823–830. Morgan Kaufmann, 2000.

[Sam95] A. L. Samuel. Some studies in machine learning using the game of checkers. pages 71–105, 1995.

[SB97] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 974. The MIT Press, 1997.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[Sut88] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[Sut96] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, volume 8, pages 1038–1044, 1996.

[SW93] Richard S. Sutton and Steven D. Whitehead. Online learning with random representations. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 314–321. Morgan Kaufmann, 1993.

[Tes92] Gerald Tesauro. Practical issues in temporal difference learning. In *Machine Learning*, pages 257–277, 1992.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[Tes98] Gerald Tesauro. Comments on “co-evolution in the successful learning of backgammon strategy”. *Mach. Learn.*, 32(3):241–243, 1998.

[Tes02] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artif. Intell.*, 134(1-2):181–199, 2002.

[Tes08] Gerry Tesauro. Benchmark player, January 2008.

[UC91] Paul E. Utgoff and Jeffery A. Clouse. Two kinds of training information for evaluation function learning. In *In Proceedings of the Ninth Annual Conference on Artificial Intelligence*, pages 596–600. Morgan Kaufmann, 1991.