Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-Based Systems Group
Prof. G. Lakemeyer, Ph.D.

Seminar Foundations of AI

summer term 2009

Reinforcement Learning

Christian Kalla

last build: May 6, 2009

Contents

1 Introduction

Reinforcement Learning can be regarded as one of the most natural ways of learning by interacting with the environment.
In contrast to other methods of machine learning, an agent only learns by receiving feedback from its environment and trying out which actions yield the highest return in a certain situation. It does not seem astonishing at all that one of the roots of this method of learning can be found in psychology. An illustrative example is comparing this method to the way infants learn when making their first steps in an unknown environment. For example crying will often lead to the result that the parents get something to drink or to eat for the child, which can be regarded as a reward. So crying would be learned as a good action when trying to reach the goal of getting food. Another analogy is the situation when learning to drive (also referred to in [Mar04]). Neglecting the driving instructor who could turn this example into one for supervised learning, driving can also be learned by paying attention to the reactions of the environment (e.g. the car and other road users). One goal could be to get no bad reaction of the other drivers, so honking could be a bad signal from the environment and remind you to drive faster for example. More of those analogies can be found in [SB98].
One important fact is that the aim of an agent is always to maximize the sum of rewards on the way to its goal and not to focus on single rewards. This stresses the need of a so called value function that tells the agent how good certain states are with respect to the expected sum of rewards.
The first section of this paper will deal with methods and algorithms to compute this value function and point out the occuring difficulties and will conclude with an overview of some interesting applications. In this context Markov Decision Processes (see [BSW90],[How60]) play an important role which assume a stationary environment that does not contain other adaptive agents. The arising problem from this is that cooperation of agents that would be important for many multiplayer games cannot be considered as it is not supported by the mathematical framework.
This problem is regarded in the second section of this paper mainly considering the work of [Lit94]. Here Markov Games which explicitely support multi-agent environments are used to adapt the Q-Learning algorithm presented in the first section to develop an agent for a simple version of soccer. The corresponding algorithm will be presented and the results against other agents will be shown and discussed.

2 Reinforcement learning in general

2.1 The main idea behind reinforcement learning

The general idea of reinforcement learning is visualized in figure 1. An agent starts at time t in a given situation s_t and chooses an action a_t. This action takes him to a new state s_{t+1} and gives a reward r_{t+1} as feedback. The new state and the reward depend on the dynamics of the environment and may be deterministic or stoachastic. This means that the rewards and successor states are either clearly defined (e.g. in games like chess) or occur with a certain

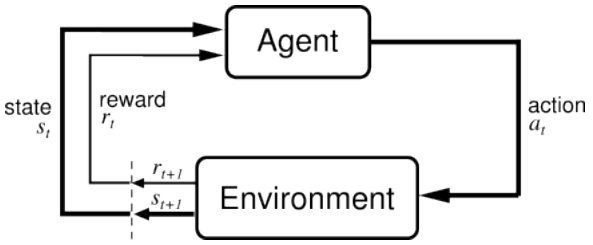

Figure 1: The model of the agent and its environment for reinforcement learning (taken from [SB98])

probability (this could be the case in real-time environments). In many scenarios the reward is often not given directly, but when reaching the goal. This often leads to the problem that an agent does not know how good his actions were before having reached the goal state for the first time.

2.2 Important problems

The control problem

Problem of exploration and exploitation A difficult task for an agent in an environment is to find the right balance between using the information gained by already visited states (e.g. the values of states in the reinforcement learning case) and trying to explore the environment by taking actions leading to unknown states. One problem that can occur during reinforcement learning is that once the agent has found a way to the goal, it will always use the same way again, without knowing if there is another (probably better) one. We all know this problem when driving to work for example being annoyed about traffic jams, for example. Of course one way found by our initial policy (e.g. the motorway) definitely leads to the goal, but having driven the same way for many times, we try out something new the next day and "explore" the state space in some way. Of course this may also lead to worse results than before, but there also might be faster ways (analogous to better policies) leading to the goal. ϵ-greedy policies are an example for policies that in general choose the action leading to the state with the highest value, but sometimes (with probability ϵ) take random actions.

The prediction problem

Partial observability problem

Curse of Dimensionality difficulty to determine optimal state and action sets

Credit Structuring Problem

2.3 Value function and Policies

As mentioned before, the main goal of an agent is to find an optimal policy π^*, i.e. a roadmap that enables the agent to choose the right action in each state in order to reach its goal as fast as possible. Or more formally:

Definition 1 (Policy) *Let S be a set of states and A be a set of actions an agent can choose in its environment.*
A mapping $\pi : S \rightarrow A \; \forall s \in S$ is called a policy .

Definition 2 (Optimal Policy) *A policy $\pi^* \in \Pi$ (Π denotes the set of all possible policies) that maximizes the quantity*

$$R = r_0 + r_1 + ... + r_n \tag{1}$$

for Markov decesion processes with a terminal state or the quantity

$$R = \sum_t \gamma^t r_t \tag{2}$$

for MDPs without a terminal state ($0 \leq \gamma \leq 1$ denotes the discounting factor) is called optimal policy.

In order to find optimal policies, a function that defines the value of a state itself (V-function) or the value for choosing an action in a given state (Q-function) is needed. As the goodness of a state always depends on the expected future reward, it depends also on the future actions taken by the agent. That is why value functions always have to be considered with respect to a certain policy (denoted by the index π). More formally:

Definition 3 (State-value function) *The function*

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \tag{3}$$

is called the state-value function for policy π. E_π denotes the expected value given the fact that the agent follows the policy π with an arbitrary time step t.

The action-value function $Q^\pi(s, a)$ that assigns values to a state-action pair is defined similarly:

Definition 4 (Action-value function) *The function*

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{4}$$

is called the action-value function for policy π.

As the goal of reinforcement learning is to find an optimal policy from the initial state to the goal, there have to be methods to compute an optimal policy efficiently. A naive way to do so would be trying out all possible policies which means taking every possible action in every state and receiving the rewards from the environment. Of course the complexity of this approach would be too high, because the set of states and the set of actions are very large in general, so exploring the whole search space is inefficient. There are many methods to approximate optimal value functions and therefore also give an approximation for an optimal policy. The first two methods that will be presented use dynamic programming and perform policy or value iteration. This means to start with an arbitrary policy for example, determine the resulting value function for this policy and use this value function to get an improved policy. After this the same procedure is iterated several times until the optimal policy is found (or at least a good approximation). A significant disadvantage of those dynamic programming methods is that they require a complete model of the environment and so they are not applicable if the environment is unknown to the agent.
In contrast Monte-Carlo methods do not require this model. Here the value function is learned by averaging the rewards of many episodes. Because the transition probabilities of the environment are not known (in contrast to DP methods), they have to be determined by many iterations.
Another important kind of reinforcement learning algorithms do not require this environment model and use Temporal Difference learning. This means that they compute the difference between the two values of a value function at different time points (e.g. $Q(s,a) - Q(s',a')$). The presented algorithms Q-learning and SARSA are two examples for this kind of learning method.

2.4 Separation from other machine learning approaches

This section gives a brief overview of the placement of reinforcement learning in machine learning. Machine learning in general distinguishes between three learning approaches: Supervised learning, unsupervised learning and reinforcement learning. The main task of all those approaches is to use the knowledge gained from previous experience to solve new tasks using this knowledge.
In supervised learning the agent is given a set of examples containing input and the correct output. After that the agent adapts the parameters of its model (e.g. the parameters of a probability distribution) in such a way that the seen data is represented best. This is often done by the maximum likelihood method. One example for this kind of learning could be the recognition of handwritten letters or digits, where the agents gets some example images with the correct letter or digit. After that the model is used to classify some unknown input text.
In unsupervised learning the agent is just given the data itself, but not the correct class, so there is no "teacher" as in supervised learning. In order to learn from the data and derive a model, clustering methods and the EM algorithm play an important role.
Reinforcement learning differs from both previous approaches, because the agent has to discover its environemnt on its own, without having a supervisor that tells him the correct actions. The only feedback if the chosen action was good or bad can be extracted from the reward or punishment the action brings.
But which advantages does this approach have? In [SB98] an example of a Tic-

Tac-Toe playing agent is given. Everybody knows that this game always ends up with a draw when both players make the best moves, but it could also be the case that the opponent agent shows weaknesses in certain positions, although it might play well otherwise. A standard approach to those kinds of zero sum games is applying the Minimax algorithm with a simple heuristic that guarantees to gain the best result assuming that the opponent always chooses the best move. One disadvantage here is that always optimal play of the opponent is assumed, although this might be wrong. Applying reinforcement learning algorithms to this kind of game would result in an agent that discovers the weaknesses of its opponent and exploits, whereas a minimax approach would just be able to reach a draw for example. A quite strong agent for the game of backgammon using reinforcement learning techniques is presented at the end of the first section.

2.5 The Markov Property and Markov Decision Processes

The goal of reinforcement learning is to maximize an agents reward and to find an optimal policy leading to a goal state. But how do environments behave, if an agent chooses an action in a state at a given time? The first thing we have to consider when answering this question is to imagine different environments and think about their characteristics. At this point we have to distinguish between deterministic and stochastic environments. Playing chess is an example for a deterministic environment. There is a specified choice of actions in each state and each of those actions leads to a clearly defined next state. The environment we live in does not necessarily lead to one defined state when taking an action. Instead an action could lead to several states and each of those successor states can occur with a given probability. For example the state "Driving drunk" with the action "Driving in an erratic manner" could lead to the state "Getting stopped by the police", but also to the state "Arriving safe at home". Another important question is if the probability of the successor states depends on the actions taken before, or if just the current state counts for future decisions. This is exactly what the Markov Property demands. For the chess example this question is easy to answer: The agent has to make his decision based on the current position (i.e. the current state) and the environment also does not care about what happened in the past. For other kinds of environment this conditional independence from earlier states cannot be guaranteed in many cases. One thing that might be confusing in this context is the definition of a state itself. The definition of a state takes information about the agents' sensors as input, so a state can depend from sensations made somewhen earlier. The Markov expression can formally be expressed as follows:

$$\begin{aligned} &Pr\left\{s_{t+1}=s', r_{t+1}=r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\right\} \\ =&Pr\left\{s_{t+1}=s', r_{t+1}=r | s_t, a_t\right\} \end{aligned} \tag{5}$$

Every reinforcement learning task whose states have the Markov Property is called a Markov Decision Process (MDP). If the state and action space are finite we talk about finite MDPs. MDPs are formally defined in [How60] and consist of the following components:

- a set of states $\mathcal{S}$

$$\pi_0 \overset{evaluation}{\longrightarrow} V^{\pi_0} \overset{improvement}{\longrightarrow} \pi_1 \overset{evaluation}{\longrightarrow} V^{\pi_1} \ldots \overset{improvement}{\longrightarrow} \pi_* \overset{evaluation}{\longrightarrow} V^{\pi_*}$$

Figure 2: The principle of policy iteration

- a set of actions $\mathcal{A}$
- a set of rewards $\mathfrak{R}$
- a transition function $T : \mathcal{S} \times \mathcal{A} \to PD(\mathcal{S})$ where $PD(\mathcal{S})$ denotes the set of probability distribution over $\mathcal{S}$
- a reward function $R : \mathcal{S} \times \mathcal{A} \to \mathfrak{R}$

In stochastic environments the successor state is not only defined by an action, but also by a probability distribution defining the transition probabilities:

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{6}$$

Accordingly the expected value of the next reward can be defined as

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \tag{7}$$

2.6 How to find optimal policies?

Gives an overview of algorithms to compute optimal policies.

2.6.1 Dynamic Programming

The most obvious way of finding an optimal policy is using dynamic programming techniques. One way is to start with an arbitrary policy, derive the value function and improve the policy thereafter (policy iteration). Another way is to start with an initial value for all states and adapt this value function iteratively (value iteration).

Value iteration

Policy Iteration An illustration of the principle of policy iteration can be found in the following figure taken from [SB98]. The algorithm in pseudocode is given in 1. The example in figure **??** underlines how the presented algorithm works. The task of the agent is to get from its starting state (the green square) to the goal (the red square) by moving up, down, left and right. The agent only receives a reward if the goal state is reached (+100), otherwise it does not gain anything (0). The initial policy is deterministic and seems to be quite stupid, but according to the algorithm the agent improves this policy quite fast. The discount factor γ is set to 0.9.

Algorithm 1 Policy Iteration

1. **Initialization**
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$
2. **Policy Evaluation**
   repeat
   $\delta \leftarrow 0$
   for $s \in \mathcal{S}$ **do**
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)}(\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s'))$
   $\delta \leftarrow \max(\delta, |v - V(s)|)$
   end for
   until $\delta < \theta$ (a small threshold to be reached)
3. **Policy Improvement**
   $policy - stable \leftarrow true$
   for $s \in \mathcal{S}$ **do**
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a}(\mathcal{R}_{ss'}^{a} + \gamma V(s'))$
   if $b \neq \pi(s)$ **then**
   $policy - stable \leftarrow false$
   end if
   end for
   if $policy - stable$ **then**
   STOP
   else
   GOTO 2.
   end if

2.6.2 Monte Carlo Methods

The previously discussed dynamic programming methods had the clear disadvantage that they required a model of the dynamics of the environments, i.e. that the transition probabilities of each state-action pair had to be known in advance. Monte Carlo methods do not need this kind of knowledge. Instead they learn the value function of a state by computing the average return after the occurrence of a state generating a large number of episodes. If the number of generated episodes is high enough, the computed state value or state-action value will converge to the real one. Algorithm 2 shows a simple way of computing an optimal policy by generating an episode first and adapting the state-action value and the policy afterwards.
One important fact about Monte Carlo algorithms is that they do not make use of bootstrapping, i.e. they do not update the value of a state based on the values of successor states (also known as bootstrapping), but they compute the accumulated reward for each episode after the first occurrence of the state (or the state-action pair). This also makes them less "vulnerable" against environments that do not fulfill the Markov property.

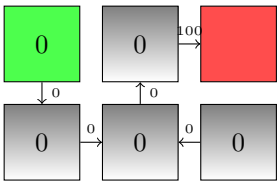

Figure 3: Initial situation of the grid world with rewards for the actions

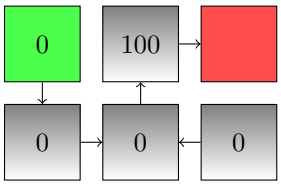

Figure 4: situation after one value update

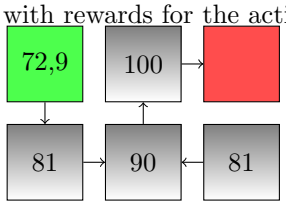

Figure 5: situation after all value updates

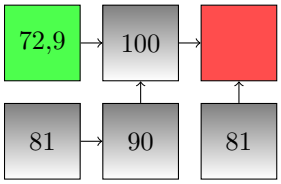

Figure 6: situation after policy change

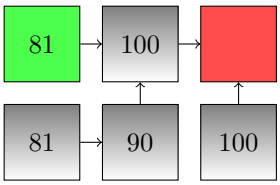

Figure 7: algorithm terminates after final value update

2.6.3 Temporal Difference Learning

Algorithms that make use of temporal difference learning perform bootstrapping (as dynamic programming methods do), but do not need a model of the environment (a property that also applies to Monte Carlo methods). The term "temporal difference" just means that the value update of a state depends on the value of another state at a different timepoint (namely the successor state). In connection to this kind of learning methods the terms "on policy" and "off policy" are often used. "On policy" means that the policy that is used to generate an episode (the behaviour policy) is the same as the policy that is evaluated and improved (the estimation policy). In "off policy" methods those two policies can be independent. The advantage of this is that the state space could be explored better than in an "on policy" method, because episodes generated by a random policy ensure that not only similar episodes are generated (as it could be the case if a greedy policy is used both for estimation and behaviour control). Today TD-learning algorithms are very often applied for reinforcement learning problems.

The Q-Learning algorithm The Q-Learning algorithm is an off-policy control algorithm introduced in 1989 by C.J.C.H. Watkins and was quite a breakthrough at this time. It approximates the optimal state-action function Q^* independent of the policy being followed. the Q-learning algorithm (in pseudocode)

Algorithm 2 Monte Carlo Algorithm for finding an optimal policy

Initialize $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow arbitrary$
$\pi(s) \leftarrow arbitrary$
$Returns(s,a) \leftarrow emptylist$
repeat
  Generate an episode using policy π and a random starting state
  for all pairs (s,a) appearing in the generated episode **do**
    $R \leftarrow$ return following the first occurence of (s,a)
    Append R to $Returns(s,a)$
    $Q(s,a) \leftarrow average(Returns(s,a))$
  end for
  for all states s in the episode **do**
    $\pi(s) = \arg\max_a Q(s,a)$
  end for
until final number of episodes is reached

Algorithm 3 Q-Learning algorithm

Initialize $Q(s,a)$ arbitrarily
for each episode **do**
  Initialize s
  repeat
    Choose a from s using policy derived from Q
    Take action a and observe a, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
    $s \leftarrow s'$
  until s is terminal
end for

can be found in algorithm 2.6.3. The basic idea is to generate a state-action pair by the behaviour policy and to update the value of the old state-action pair by observing the reward and the value of the highest state-action pair of the folowing state after executing the one generated by the policy. Of course again the discount factor γ and the learning rate, denoted by α play a role. As already mentioned, the Q-Learning algorithms plays an important role in reinforcement learning, because it is widely used and part of many applications. One example is the backgammon-playing agent by Tesauro.

Sarsa This algorithm was invented by Rummery and Niranjan in 1994 and is also regarded as a popular algorithm for reinforcement learning. In contrast to the Q-Learning algorithm it is an on policy approach and makes use of the following update rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t) \tag{8}$$

The basic idea of this update rule is to start with an inital state, choose an action according to the policy (note that the policy that chooses actions and the policy that is improved are identical), execute this action and receive the

reward. Afterwards the best action according to the policy (a_{t+1}) is chosen and the Q-value of the old state action pair $Q(s_t, a_t)$ is updated. Then the procedure is repeated with the new state (s_{t+1}) and the new action (a_{t+1}) until a terminal state is reached. The complete algorithm is omitted at this point, but with the update rule as its central component it should not be hard to imagine how it should look like. The name of the algorithm "Sarsa" can be formed from the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. A pseudocode example can be found in [SB98].

TD(λ)

2.6.4 Eigibility traces as an improvement of reinforcement learning algorithms

2.7 Applications of reinforcement learning

Gives an overview of some interesting applications of reinforcement learning. Perhaps some other interesting stuff could be added here.

2.7.1 TD-Gammon

3 Multi-agent reinforcement learning

As we have seen in the previous section, the term reinforcement learning is strongly related to Markov decision processes. Unfortunately the underlying mathematical framework for MDPs only supports environments with one agent. But how to apply the algorithms for MDPs to multiagent environments? Littman's paper ([Lit94]) on multi-agent reinforcement learning addresses this problem and presents a variation of the Q-Learning algorithm, called "Minimax Q-Learning" based on Markov games.

3.1 The framework of Markov Games

Markov games are also called stochastic games and are formally defined as follows (also see [Owe95]):

Definition 5 *A Markov game is defined by a set of states $\mathcal{S}$ and a collection of action sets $A_1, ..., A_k$ for each agent in the environment. A transition function T maps a tuple of the current state and actions taken by each agent to a new state with a certain probability:*

$$T : S \times A_1 \times ... \times A_k \mapsto PD(\mathcal{S}) \tag{9}$$

Furthermore there are reward functions R_i for each agent i:

$$R_i : S \times A_1 \times ... \times A_k \mapsto \Re \tag{10}$$

The aim of each agent i is to maximize the expected sum of discounted rewards:

$$E\left\{\sum_{j=0}^{\infty}\gamma^j r_{i,t+j}\right\} \tag{11}$$

As the mathematics of those multiplayer games can get very complicated, Littman simplifies it considering only two agents with opposed goals playing against each other. This assumption makes it possible to use a single rewards function that is maximized by the first agent and minimized by the second one. Nevertheless cooperation between agents can no longer be considered. This simplified game can be seen as a "Zero-sum markov game", because the sum of rewards of the agents is zero. One important thing to realize is that Markov games are a generalization of MDPs, in which the number of the opponents's actions (denoted by $|O|$) is 1.

3.1.1 Finding optimal policies in Markov games

There are some important differences between optimal policies in markov games and markov decision processes. The aim of an agent in an MDP is to maximize the sum of discounted rewards. The same applies to Markov games, but here it can not be assured that there is an undominated policy. That means for every policy it could be the case that there is a state from which another policy would return a higher sum of discounted rewards than the current one. So in contrast to Markov games, each MDP has an undominated optimal deterministic policy which is also stationary (i.e. it does not depend on the time). As optimality of a policy in Markov games always depends on the behaviour of the opponent, an optimal policy is defined as a policy that assures the highest reward assuming optimal play of the opponent. Taking this definition, also every Markov game has at least one optimal (and even stationary) policy. But the deciding difference between optimal policies in MDPs and those in Markov games is that the policies in Markov games may be stochastic, whereas it is guaranteed that there is an optimal deterministic policy in an MDP.
An easy example is the game of "rock-paper-scissors". It is not hard to realize that choosing one of the three symbols at random is a better policy than always taking the same symbol so that it could happen to be second guessed. Also the simplified version of soccer presented later on requires a stochastic policy in certain situations.
But how to find the optimal policy in Markov games?
Similar to MDPs we can define $V(s)$ as the expected reward for the optimal policy in state s and $Q(s,a,o)$ as the expected reward when taking action a in state o knowing that the opponent has chosen action o. According to MDPs the value of a state can be expressed as

$$V(s) = \max_{\pi\in PD(A)} \min_{o\in O} \sum_{a\in A} Q(s,a,o)\pi_a \tag{12}$$

and the value of a state-action pair against the opponent action o as

$$Q(s,a,o) = R(s,a,o) + \gamma \sum_{s'} T(s,a,o,s')V(s') \tag{13}$$

Algorithm 4 The Minimax-Q algorithm (adapted from [**?**])

1. **Initialization**
   $\forall s \in \mathcal{S}, a \in \mathcal{A}, o \in \mathcal{O}$
   $Q(s, a, o) \leftarrow 1$
   $\forall s \in \mathcal{S}$
   $V(s) \leftarrow 1$
   $\forall s \in \mathcal{S}, a \in \mathcal{A}$
   $\pi(s, a) \leftarrow 1/|A|$
   $\alpha \leftarrow 1.0$

2. **Choice of the action**
   return a random action with probability *explor*
   Ohterwise:
   if current state is s **then**
   return an action a with probability $\pi(s, a)$
   end if

3. **Learning phase**
   Receive reward $r(s, a, o, s')$ for moving from state s to s'
   $Q(s, a, o) \leftarrow (1 - \alpha) \cdot Q(s, a, o) + \alpha \cdot (rew + \gamma \cdot V(s'))$
   Use linear programming to find $\pi(s, .)$ such that
   $\pi(s, .) \leftarrow \arg\max_{\pi'(s,.)} \{\min_{o'} \{\sum_{a'} \pi(s, a') \cdot Q(s, a', o')\}\}$
   $V(s) \leftarrow min_{o'} \{\sum_{a'} \pi(s, a') \cdot Q(s, a', o')\}$
   $\alpha \leftarrow \alpha \cdot decay$

Those equations look very similar to the equations for MDPs with the min and max operators and the opponent action as third tuple element as the only differences. The value function of a state s can be computed by solving a so called "Matrix game" (a markov game with just one state and a matrix of rewards such as rock-paper-scissors) using linear programming techniques. Another way of getting optimal Q-values and therefore an optimal policy is transforming the Q-Learning algorithm mentioned in the first section of this paper to a similar algorithm for Markov games. The only change that has to be made to get this new algorithm is replacing the max operator in the update step of the Q-Learning algorithm with a minimax operator.

3.2 The Minimax-Q learning algorithm

The algorithm defined in Michael Littmans's paper ([Lit94])

3.3 Soccer as an application of the Minimax-Q algorithm

3.3.1 Game description and rules

The game condidered in Littman's paper is a very simplified version of soccer. It is played between two agents A and B in a 4×5 gridworld. Figure 8 gives

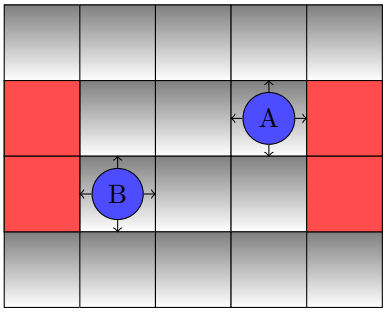

Figure 8: Initial situation

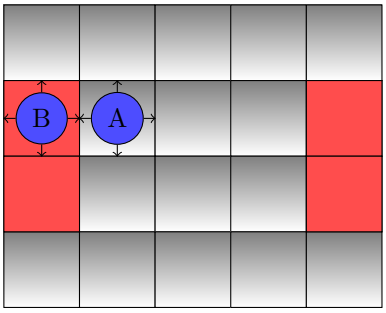

Figure 9: Situation that requires a probabilistic policy

an overview of the initial setup of the grid. The red squares mark the goals and the aim of each agent is to step into one of those squares in order to score a goal. After a goal has been scored the game is reset to the initial state. Ball possession is assigned randomly to an agent at the beginning of the game. Unlike to normal soccer there is no real possibility to shoot, but you can just move with the ball or stand. So in each state an agent can choose between the actions $\mathcal{A} = \{N, S, E, W, stand\}$ where the first four actions stand for moving into a direction (north, south, east or west) and the last action (stand) keeps the situation unchanged. If an agent chooses an action that takes him to a square occupied by the other player, the player that did not move will get the ball. The game consists of rounds and the players select their moves at the same time without knowing the move of their opponent in advance. After the selection of the moves, they are executed in a random order.
The discount factor γ is set to 0.9 which assures that scoring goals earlier is better than scoring them later.
Figure 9 shows a situation that requires a probabilistic policy in order to win. Just suppose A always chooses to go left in this situation, then B would always choose the action stand and get the ball after both moves are executed. If A chooses a probabilistic policy, things look different. Now B does not know if a chooses "stand" or "south", because those actions are selected by a probabilistic policy and A could probably get a good opportunity to score (ore walk into) the opponent's goal.

3.3.2 Training and Testing

3.3.3 Results

4 Related Work

5 Conclusion and Outlook

Conclude with a summary and give an outlook on future stuff.

Table 1: Results of games of the agents trained with different policies (taken from [Lit94])

	MR		MM		QR		QQ	
	% won	games	% won	games	% won	games	% won	games
vs. random	99.3	6500	99.3	7200	99.4	11300	99.5	8600
vs. hand-built	48.1	4300	53.7	5300	26.1	14300	76.3	3300
vs. MR-challenger	35.0	4300						
vs. MM-challenger			37.5	4400				
vs. QR-challenger					0.0	5500		
vs. QQ-challenger							0.0	1200

References

[ASP00] Sachiyo Arai, Katia Sycara, and Terry R. Payne. Multi-agent reinforcement learning for planning and scheduling multiple goals. In *In Proceedings of the Fourth International Conference on MultiAgent Systems*, pages 359–360, 2000.

[BSW90] Andrew G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. In *Learning and Computational Neuroscience*, pages 539–602. MIT Press, 1990.

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

[How60] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The MIT Press, Cambridge, Massachusetts, 1960.

[Lit94] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[Lit01] Michael L. Littman. Value-function reinforcement learning in markov games. *Cognitive Systems Research*, 2(1):55 – 66, 2001.

[Mar04] Irene Markelic. Reinforcement Learning als Methode zur Entscheidungsfindung beim simulierten Roboterfußball. Technical report, Universität Koblenz-Landau, 2004.

[Owe95] Guillermo Owen. *Game Theory: Third Edition.* Academic Press, 1995.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. of Res. Develop.*, 3:211–229, July 1959. (Reprinted in *Computers and Thought*, (eds. E. A. Feigenbaum and J. Feldman), McGraw-Hill, 1963, pages 39–70).

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

[Tan93] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[TCR+05] Karl Tuyls, Tom Croonenborghs, Jan Ramon, Robby Goetschalckx, and Maurice Bruynooghe. Multi-agent relational reinforcement learning. In *Proceedings of the First International Workshop on Learning and Adaptation in Multi Agent Systems*, pages 123–132, 2005.

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

[Tes02] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.