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 21, 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 Foundations of Reinforcement Learning

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 state s_t and chooses an action a_t ($a_t \in \mathcal{A}(s_t)$). 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

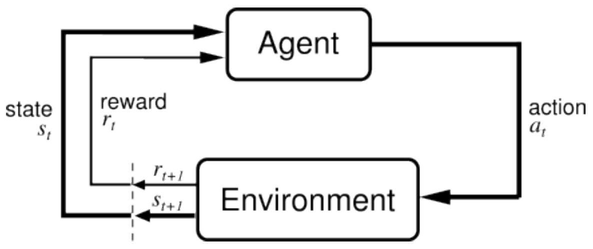

Figure 1: The model of the agent and its environment for reinforcement learning (taken from [SB98])

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

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 and the control problem To reach our main goal in reinforcement learning, namely to maximize the long-term reward and reach a goal state (at least in a continuous environment), it is necessary to define values for every state that determine how good a state is in terms of maximizing the future rewards. The task to find optimal values predicting the expected reward is referred to as "Prediction problem" in literature. A related problem making use of the solution for the prediction problem is the control problem. A solution to this problem is a plan (an optimal policy) that tells us in every situation which action to take in order to receive the highest reward in a long-term view. So a solution for the prediction problem would give us directly a solution for the control problem.

Partial observability problem Real world environments are often not fully observable, that means that the agent does not exactly know in which state it will end up after performing an action. This could occur if the values of the sensors cannot clearly indicate if the agent is in state A or state B, because the measurements are characteristic for both states. In the case of partial observability "standard" reinforcement learning algorithms are not applicable, but there are other solution methods based on POMDPs (partial observable Markov Decision Processes).

Curse of Dimensionality In many environments the state and the action space are to huge to be explored completely. Due to this facts there have to be methods to reduce the dimension of those spaces without losing important information. This could be done by means of state-space abstraction.

Credit Structuring Problem As the agent learns by being rewarded or punished, it is also an important task to structure the rewards in such a way that the agent learns fastest. Taking the game of chess as an example, it would make no sense to give a reward for winning material in general, because this could tempt our agent into winning material instead of checkmating the opponent which has to be seen as the main goal. One disadvantage of delaying the reward to the goal state is that it may take a very long time until the agent reaches this state, because there is no indicator in form of rewards that lead the agent to the final state. This is why learning could take very long, because the state sequence to the goal state in games like chess is very long, also because of the fact that the agent just makes random moves at the beginning, because it has not developed a policy yet. This is why assigning rewards to states is not an easy task at all.

Non-stationary environments This problem addresses the fact that the environment sometimes changes during learning. So a value function learned for a high number of episodes could be wrong, because the dynamics of the envirionment have already changed in the meantime. If the change of the environment is too fast, learning is no longer possible. Nevertheless, not only reinforcement learning methods, but learning in general is affected by this 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 $\pi : S \rightarrow A$ for all $s \in S$ is called a policy .

Definition 2 (Optimal Policy) *A policy π^* that maximizes the quantity*

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

for Markov decesion processes (MDPs) 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 an optimal policy.

In order to find optimal policies, a value 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 value 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*

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

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*

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

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.

2.3.1 ϵ-greedy policy

In order to derive a policy from the value function, it makes sense to choose the action that maximizes the value of the successor state. But in order to guarantee an optimal exploration of the state space, an ϵ-greedy policy chooses a random action with probability ϵ instead of trying to choose the "best" action in terms of the highest value of the successor states. Those kind of policies are often used in reinforcement learning algorithms.

2.4 Separation from other machine learning approaches

This section gives a comparison of reinforcement learning and other machine learning techniques. 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 sequence 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.

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

$$\begin{tikzcd}\pi_0 \arrow[r, "evaluation"] & V^{\pi_0} \arrow[r, "improvement"] & \pi_1 \arrow[r, "evaluation"] & V^{\pi_1}\ldots \arrow[r, "improvement"] & \pi_* \arrow[r, "evaluation"] & V^{\pi_*}\end{tikzcd}$$

Figure 2: The principle of policy iteration

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

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

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

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

2.6 How to find optimal policies?

This section 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).

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 shown in the figures 3-7 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.

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

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
```

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, 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.

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

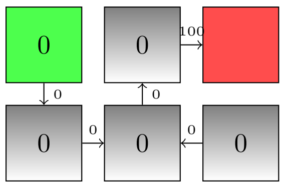

Figure 3: Initial situation of the grid world with rewards for the actions

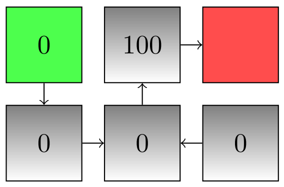

Figure 4: situation after one value update

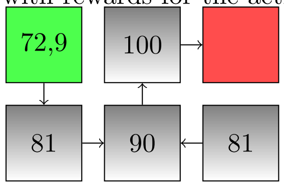

Figure 5: situation after all value updates

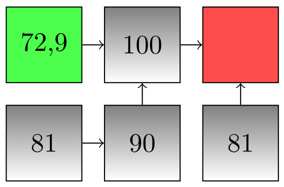

Figure 6: situation after policy change

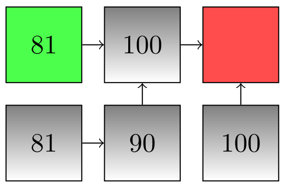

Figure 7: algorithm terminates after final value update

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 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) can be found in Algorithm 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 following state after executing the one generated by the policy. Of course again the discount factor γ and the learning rate α play a role. As already mentioned, the Q-Learning algorithm is crucial for reinforcement learning and it is widely used and part of many applications. One example is the backgammon-playing agent by Tesauro.

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 $\pi$ 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
```

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 what 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].

2.6.4 Eligibility traces as an improvement of reinforcement learning algorithms

In the introductory part we faced the problem that reaching a first reward may last a very long time. An additional problem occurs when using standard TD algorithms in this scenario: Only one state value (or state-action value) $V(s)_t$ ($Q(s,a)_t$) is updated based on the successor value $V(s)_{t+1}$ ($Q(s,a)_{t+1}$). During the next episode we are given no information how to behave until we have reached a recently updated state. So it would be nice if the previously visited state values $V(s)_{t-1}, V(s)_{t-2}, ...$ or state-action values $(Q(s,a)_{t-1}, Q(s,a)_{t-2}, ...)$ would also be updated with a specified decay factor such that earlier actions are affected less than recent actions. This is exactly what eligibility traces do. In principle they store an additional value $e(s)$ ($e(s,a)$) for each state or each state-action pair that describes how much influence a state has on the whole state sequence followed by the agent. So if a state value is updated, all other values of states visited before are also updated (with the factor $\gamma\lambda$ (λ denotes the decay factor and γ the already known discount factor)). This technique can be applied to all known TD-learning algorithms. Algorithm 4 shows an extended implementation of SARSA making use of eigibility traces. Temporal difference

Algorithm 4 SARSA(λ)

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0 \forall (s,a)$
repeat
  initialize $s, a$
  repeat
    execute action $a$ and receive reward $r$ and successor state $s'$
    choose $a'$ from $s'$ according to the policy derived from $Q$
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all state action pairs $(s,a)$ do
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    end for
    $s \leftarrow s'$
    $a \leftarrow a'$
  until $s$ is terminal
until maximal number of episodes reached
```

learning algorithms using a parameter λ for eligibility traces are often abbreviated as TD(λ). Figure **??** shows a backup diagram of TD(λ). There are two special cases: If $\lambda = 0$ then the scenario is equal to one-step TD methods (like discussed before). If $\lambda = 1$, then the algorithm turns to a Monte-Carlo algorithm, because the value of a state depends on the whole set of successor states until the final state. But how to choose the parameter λ appropriately? There is no clear answer to this question, but in tasks with long delayed rewards it makes sense to use quite a high value in order to accelerate convergence. Nevertheless the computational effort of eligibility traces is higher than using one-step prediction methods.

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. As there has been a lot of research on this topic during the last years, there are lots of papers describing different approaches. A good overview of the research can be found in [BBD06] and [**?**]. In fact there are two main approaches: Using methods from game theory to extend the single agent view of MDPs to Markov Games or Stochastic Games (the Minimax-Q and the Nash-Q algorithm are an example for this category) and extending standard reinforcement learning algorithms by using distributed value (or state-value) functions (the Distributed-Q algorithm serves as an example for this class). The goal of this section is to address the problems evolving when regarding a multiplayer scenario instead of a single player one and to give an overview of solution approaches.

3.1 Problems of multi-agent reinforcement learning

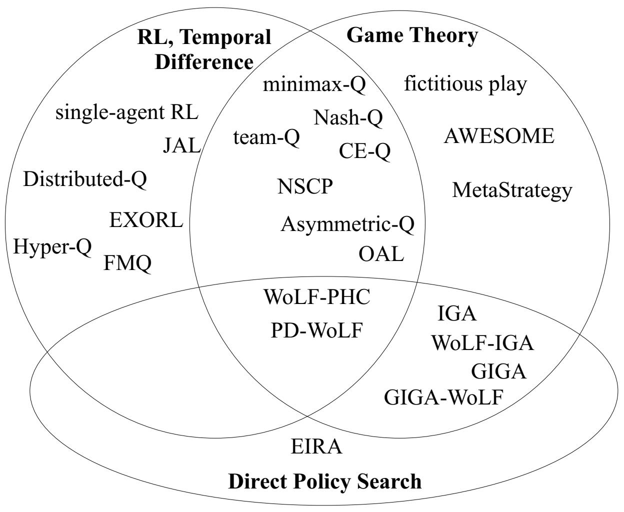

Figure 8: Categorization of algorithms

3.2 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 its 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.

3.2.1 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 s 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

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

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.

3.3 The Minimax-Q learning algorithm

The Minimax-Q learning algorithm ([Lit94]) is a modification of the Q-Learning algorithm and makes it possible to compute an optimal stochastic policy for an agent acting in a two-player Markov game. The important point here is that the resulting policies are no longer deterministic (as in the original algorithm), but stochastic and assume optimal play of the opponent (as this is usual for Minimax). The update function is identical to the Q-Learning case, except the fact that the state-value function $Q(s, a)$ is extended to $(Q(s, a, o)$, also considering the opponent's action o. The probabilities for taking an action in each state are computed using linear programming and maximize the own reward assuming the opponent tries to minimize the $Q(s, a', o')$ value.

3.4 Soccer as an application of the Minimax-Q algorithm

3.4.1 Game description and rules

The game considered 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 9 gives 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

Algorithm 5 The Minimax-Q algorithm (adapted from [Lit94])

```
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
     $\pi(s, .) \leftarrow \arg\max_{\pi'(s,.)} \{\min_{o'} \{\sum_{a'} \pi(s, a') \cdot Q(s, a', o')\}\}$
     $V(s) \leftarrow min_{o'} \{\sum_{a'} \pi(s, a') \cdot Q(s, a', o')\}$
     $\alpha \leftarrow \alpha \cdot decay$
```

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 10 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.

3.4.2 Training and Testing

This section describes the setup of the games the agents played against each other.
There were four different policies trained, two using the standard Q-learning

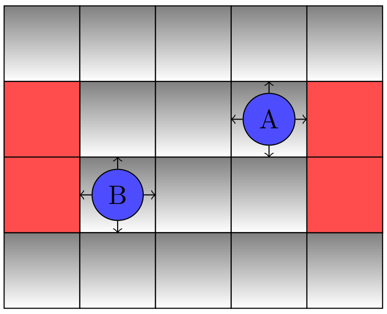

Figure 9: Initial situation

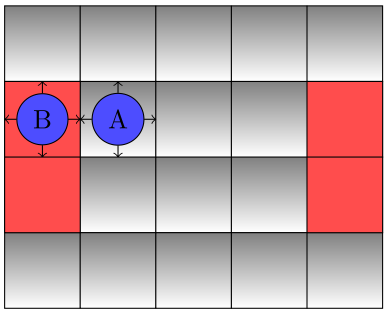

Figure 10: Situation that requires a probabilistic policy

algorithm and two using the Minimax-Q learning algorithm presented in this section. The policies were always trained against an opponent using a random policy and against an opponent using the same policy as the original agent. So the resulting policies were QR (Q-Learning trained agains random) and QQ (Q-Learning trained against Q-Learning), MR and MM (analogously). The parameters of the Minimax-Q learning algorithm were set to $explor = 0,2$, $decay = 10^{\log 0.01/10^6} = 0,9999954$ (in order to read a learning rate of $0,1$ at the end of the run). The standard Q-Learning algorithm was initialized with the same parameters as the Minimax-Q variation. There were three ways of evaluating the policies:

1. each policy vs. random policy for 100000 steps (probability of $0,1$ for a state being declared a draw)
2. each policy vs. hand-built policy (deterministic policy with rules for scoring and blocking that won 99,5% of 5600 games against a random opponent)
3. a challenger opponent for each of the policies MR, MM, QR and QQ was trained using Q-Learning. The champion policy was always fixed and the challenger was trained against this policy. After that the resulting policies were evaluated against their champions.

3.4.3 Results

An overview of the results can be found in Littman's paper.
The good performance of the four policies (MR,MM,QR,QQ) versus the random policy become obvious. In the case of QR this does not seem astonishing, because the policy was trained to beat a random opponent. Nevertheless the good performance of MR is remarkable, because the policy always chooses the actions with respect to the optimal behaviour of the opponent (the Minimax principle).
The performance of the different policies against the hand-built opponent were quite surprising. Both MR and MM won about 50% of the games. The small

difference of about 5 % seems strange, because normally the performance of an agent using the Minimax-Q algorithm does not depend on the opponent it was trained against (due to the fact that it always assumes optimal play). This might be an indicator for the fact that the policy has not yet converged to the optimal one. The performance of QR and QQ against the hand-built opponent show bigger differences. QQ found an optimal policy (probably by luck) that guaranteed a win in about 75 % of the games whereas QR got stuck in an inferior policy that only won about 25 % of the games. This also shows the difference between using standard Q-Learning that does not use a minimax approach and tries to adapt optimally to the opponent and tries to exploit its weaknesses and Minimax-Q. Nevertheless it the success of the Q-Learning algorithm versus the hand-built policy seems surprising, because QQ performed much better than the Minimax-Q approaches. Futhermore training two Q-Learning agents against each other is not mathematically justified, as Littman also mentions in his paper. In some cases a local maximum is reached and both agents stop learning, but in other cases this kind of policy training seems to work really well as in this case (see also [Tes92]).
In the third part of the experiment, the agents had to play against a challenger policy (i.e. a policy that was trained to beat the original policy, after it had been trained). Again MR and MM performed similarly and won a bit more than one third of the games. The fact that they won games at all is connected to the point that Minimax-Q always tries to find an optimal probabilistic policy, as this is demanded in Markov Games. So the opponent sometimes does not know how the agent will behave and loses games due to this uncertainty. In contrast Q-Learning just finds optimal deterministic policies and this is why both Q-Learning policies used in the soccer scenario completely lost all of their games against their challengers.

4 Related Work

Since reinforcement learning has been a topic of interest in artificial intelligence research for quite a long time, there is lots of information to be found in books, papers or on the internet. The most fundamental one which is also available online is the book "Reinforcement Learning- An Introduction" ([SB98]) written by Sutton and Barto. It explains the basics of reinforcement learning illustrated by intuitive examples. The thesis by Irene Markelic gives even a more intuitive understanding of the topic with less theoretical background. She applied the SARSA(λ) algorithm to a robot soccer task and also explained the principles of reinforcement learning. Littman's paper ([Lit94]) extends the single agent scenario to a multi-agent one, limiting the world to two agents. Nevertheless important multi-agent phenomena such as cooperation are also not considered there.

5 Conclusion and Outlook

This paper gave an overview of reinforcement learning in general separating it from other machine learning approaches. We saw that reinforcement learning is

closely related to solving Markov Decision Processes (MDPs) and presented several methods for this task with different characteristics. Dynamic programming methods need full world knowledge, that means they have to know the state set and the tranistion probabilities of the world before. With this information methods like policy iteration can be used. For this purpose an initial policy is chosen which is evaluated by computing the value function of each state according to the policy. After that the policy is improved by choosing the action that leads to the state with the highest value function. Those two steps (policy evaluation and improvement) are iterated until the policy does not change any more.
Monte Carlo methods do not require full world knowledge and learn by episodes. At first a random episode (a state sequence from the initial state to the goal state) is computed and the rewards that occured after visiting a state are stored in a table. The value of the state is computed as the average of the accumulated rewards after visiting this state. So there is no update of the value function within episodes, only after completing an episode. Furthermore the value function is not computed based on successor states, i.e. no bootstrapping is performed.
The next important class of algorithms are temporal difference learning algorithms that do bootstrap but do not require a model of the environment. They make their value updates during an episode based on other estimates. Often so called eligibility traces are used here to "transfer" the knowledge gained by a reward also to the states visited during the timesteps before. Those approaches have to be separated into on-policy and off-policy methods. On-policy methods like SARSA use the same evaluation and control policy, whereas those policies can be different in off-policy methods like Q-Learning. TD-algorithms are the most widely used reinforcement learning algorithms today.
Because the framework of MDPs is not applicable to multi-agent tasks, the framework of Markov Games was used to be able to cope with this tasks. Here only a two player environment of soccer-playing agents was considered in order to simplify the mathematics. In this context a new Minimax-Q learning algorithm was developed that is identical to the "standard" Q-learning algorithm expect of replacing the max operator with a minimax operator and considering actions taken by the opponent. An important point here is that Q-Learning learns deterministic policies, whereas Minimax-Q learns stochastic policies which explains the outcome of the games against the challenger opponent. Furthermore it became obvious that the Minimax-Q algorithm did not perform that well, because it assumes optimal play of the opponent. Of course, this could also be an advantage when playing against an opponent that pretends to be a bad player and tries to force the agent to learn a bad policy in order to exploit it afterwards. Q-Learning certainly would lose against such an opponent, because it always adapts its play to the opponent's moves, but identifying those kind of agents to improve the Q-Learning performance could be an interesting topic for future research.

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.

[BBD06] L. Buşoniu, R. Babuška, and B. De Schutter. Multi-agent reinforcement learning: A survey. In *Proceedings of the 9th International Conference on Control, Automation, Robotics and Vision (ICARCV 2006)*, pages 527–532, Singapore, December 2006.

[BBDS08] L. Buşoniu, R. Babuška, and B. De Schutter. A comprehensive survey of multi-agent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, March 2008.

[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.

[lea] http://2009.rl-competition.org.

[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.