Reinforcement Learning applied to Keepaway, a RoboCup-Soccer Subtask

Student:
Robin Soetens
r.p.t.soetens@student.tue.nl

Supervisor:
Ir. Rob Janssen
r.j.m.janssen@tue.nl

August 30, 2010
Abstract

This Bachelor Final Project aims to be a demonstration of the power and usefulness of reinforcement learning, especially for RoboCup-Soccer. In the first part general theories behind reinforcement learning are described. Different kinds of basic solving methods are compared; Value iteration, Policy iteration, Monte Carlo methods, Sarsa and Q-Learning. Eligibility traces are added to the basic methods and a brief introduction to function approximation (linear tile coding) is given. The second part demonstrates the use of reinforcement learning. Keepaway, a RoboCup subtask, provides a platform to develop a homemade Sarsa algorithm with tile coding function approximation. This algorithm is used to run experiments focused on the influence of the exploring rate ($\epsilon$). Eventually a routine is added to decrease $\epsilon$ during the simulation. Learned policies are found to significantly outperform policies who behave in a random way.

Keywords: Reinforcement Learning, RoboCup-Soccer Subtask, Keepaway, Sarsa, Linear Tile-Coding, epsilon-greedy, Artificial-Intelligence

A compact-disc with keepaway software is attached in the back of this report.
Contents

1 Introduction 3

2 Reinforcement Learning Framework 5
   2.1 Two difficulties ................................................................. 6
   2.2 Three building blocks ....................................................... 6
      2.2.1 Rewards ................................................................. 7
      2.2.2 Actions ................................................................. 8
      2.2.3 States ................................................................. 8
   2.3 Explore versus exploit ..................................................... 10
      2.3.1 $\epsilon$ - greedy .................................................... 10
      2.3.2 Softmax action selection ........................................... 11
      2.3.3 Alternative methods ............................................... 11

3 Solution methods 13
   3.1 Dynamic programming .................................................. 13
   3.2 Monte Carlo methods .................................................... 14
   3.3 Temporal-Difference learning ......................................... 16
   3.4 Eligibility traces ......................................................... 17

4 Scaling up to practical applications 20
   4.1 Linear gradient-decent ................................................. 21
   4.2 Coarse coding ............................................................ 21
   4.3 Extension of solving methods ........................................ 22

5 Keepaway, a RoboCup-Soccer Subtask 25
5.1 The game of Keepaway ............................................. 25
  5.1.1 Keepaway as a reinforcement learning problem ............. 25
  5.1.2 Previous results ........................................... 26
5.2 Writing a learning code ........................................... 27
  5.2.1 Implementing Sarsa with linear tile coding ................. 28
  5.2.2 Experimental results ....................................... 28
5.3 Future work ...................................................... 31

6 Conclusion .......................................................... 32

Bibliography .......................................................... 33

A Pseudo code of various solution methods .......................... 34

B MATLAB example Monte Carlo method .............................. 38

C Installation RoboCup simulator with Keepaway players (tutorial) .................................................. 40

D Array structures in LearningAgent .................................. 43

E Learning agent code .................................................. 45
  E.1 LearningAgent.cc ............................................... 45
  E.2 LearningAgent.h .................................................. 50
Chapter 1

Introduction

Reinforcement learning has its roots in three different areas of science; Machine Learning, Neural Networks and Artificial Intelligence. In those three areas scientists reached a point at which traditional learning algorithms (supervised learning, pattern classification, adaptive control) did not fulfill their needs anymore. Problems became more sophisticated and thereby the need for a system that actually 'wants' something was urging. These systems take a hedonistic approach on learning. Algorithms based on reinforcement learning use punishment and reward as guiding parameters to maximize a certain reward value.

This concept could be considered a major breakthrough in developing automatic processes. Reinforcement learning allows scholars to impose a goal on an agent without specifying the exact actions necessary to reach that goal. The agent investigates which choices need to be made; It is able to learn by trial and error.

In 1950 *Mind Magazine* published an article written by Alan Turing (code-breaker of the Enigma and one of the founding fathers of computer technology). In this article Turing wrote:

'I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.'

Development of reinforcement learning algorithms has been one of the reasons his prediction was right.

The remainder of this report will describe the technical details that come with an application of reinforcement learning. We start with some general theoretics mostly based on a book by Sutton and Barto [1] and on Stanford University courses on Machine Learning [2]. Because most of the information in the first part of the report is based on these sources there is not a reference included at every separate remark. If a different source of information is used, references are included of course. In the second part of this report an attempt is made to demonstrate the usefulness of reinforcement learning for RoboCup-Soccer.

More specifically; A simulator made for the RoboCup simulation league is used to apply learning to a RoboCup sub-task named 'keepaway'. Artificial teams playing keepaway try to maximize the time their team possesses the ball while at the same time an opponent tries to steal it from them. Learning is applied to high level decisions like whether the player in possession should pass or hold the ball at a certain point in time. Keepaway is an interesting field of research for teams participating in RoboCup-Soccer tournaments, especially because algorithms first developed for keepaway have been extended to learn successful half field offense policies [5].

These RoboCup subtasks (both keepaway and half-field offense) provide a stage for impressive demonstrations
of the full potential of reinforcement learning methods. Hopefully my report will make a slight contribution in showing the usefulness of reinforcement learning for high level decision making in RoboCup-Soccer. Short movies demonstrating keepaway can be found at a website of the University of Texas at Austin [6], similar movies exist for half field offense [7].

Below a link to a YouTube video is provided that shows what learning algorithms in general are capable of doing. It involves a remote controlled helicopter that performs all kinds of maniac stunts. Researchers of Stanford University used much more learning methods than reinforcement learning alone (also supervised learning methods were implemented) but still the movie is impressive.

http://www.youtube.com/watch?v=VCdxqn0fcnE
Chapter 2

Reinforcement Learning Framework

Not every problem is suited to be solved by a reinforcement learning algorithm. An important property that distinguishes reinforcement learning from supervised learning is that the latter always requires an example, it always needs information about how the job should be done. Reinforcement learning does not require an example, it only needs to know whether its activity resulted in success or not. A problem could be optimized using reinforcement learning if agents:

- Should be able to act despite uncertainties about their environment.
- Constantly interact with their (possibly changing) environment.
- Have to involve implications that occur in the far future in choices made in the present.

Problems who meet these criteria can be divided in multiple categories. If the environment does not change, the problem is called stationary. In such an occasion the learning process is finished at a certain moment in time and all that is left is following the result of the learning process (an optimal policy). If the environment does change, the problem is called non stationary. In this case the learned values constantly have to be adapted in order to deal with the changes in the environment. A second division is one between 'episodic tasks' and 'continuing tasks'. An episodic task has a clear point in time at which the process is finished. In order to optimize the values a new episode should be started. Maze running is a clear example of an episodic task, after the agent escaped from the maze the job is done. An continuing task does not have a clear point in time at which the process is finished, in fact the process can go on forever. Typical examples of these kind of problems are pole balancing or learning how to walk. Often these kind of problems also face a non stationary environment. The last division we are going to make is one between 'associative' and 'non-associative' tasks. If an agent has to be able to distinguish which situation it faces before it is able to choose an action, the problem is called 'associated'. A situation is not the same as a state. The state tells the agent at which point in a domain he is while the situation tells the agent at which type of domain he is into. Basically an agent is keeping track of a separate set of values for each situation. Applied to maze running an associated problem would be 'there are three different mazes, escape as fast as possible from the maze you are facing' (of course the agent is provided with information about the maze it is facing at the beginning of an episode). Running the exact same maze until the learning values are optimal would be non associated maze running.

Within reinforcement learning methods learning always takes place by evaluation instead of instruction, no matter which type of the above mentioned problems we are facing. Therefore reinforcement learning is always about reaching a goal solitary by learning from past interactions with the environment. Publications dealing with this subject often use terms like 'agent, environment and action', instead of the terms 'controller,
controlled system (plant) and control signal’ most mechanical engineers are familiar with. This is because (as mentioned in the introduction) reinforcement learning is a field at which multiple disciplines come together. The first set of terms is generally accepted in artificial intelligence and neural networks, the second set of terms is common in machine learning.

2.1 Two difficulties

In order to solve a reinforcement learning problem there are two typical difficulties that need to be dealt with. First of all, we are often facing the problem of ‘delayed reward’. Often the agent has to go through a huge amount of states (which on itself do not yield any reward) in order to achieve a final state that yields a maximal reward. Therefore we need more information than direct reward only. Apart from the reward an agent receives for reaching a state, also a value should be determined for every state. This value should be a reflection of how much reward is expected in the future after passing this state. A nice example of the need for such values is learning to drive a car. Usually, right before an accident the driver hits the brake as hard as possible. Reinforcement learning algorithms that maximize only direct reward would notice a pattern of hitting the brake and then crashing. This shortsighted algorithm would conclude the accident is caused by hitting the brake, which is obviously wrong. In section 2.2.3 this difficulty is described in more detail.

A second typical difficulty is the trade-off between exploration (trying new opportunities) and exploitation (making use of knowledge from the past in order to choose the action that seems best at that moment in time). Humans face this trade-off as well, for example when choosing which dinner we are going to prepare. One option is to use the best recipe of your life until then, but it could also be smart to try something new. People who prepared many dinners before should choose to make their best recipe ever (this is called ‘choosing the greedy option’), while someone who did not cook before might be better off in the long run choosing something new. A second criterion in this decision is how many times cooking are left. If you know beforehand you will have to prepare your last dinner ever, it is always smart (also in the long run) to choose the best recipe you are familiar with. In section 2.3 the trade-off between exploration and exploitation is dealt further with.

2.2 Three building blocks

Every reinforcement learning problem can be reduced to three different channels of communication between agent and environment. These three building blocks are Rewards, States and Actions (depicted in figure 2.1. At every discrete step in time, the agent receives information about the state it is in and the reward that comes with it. At the same time the agent emits information about which actions it is willing to take. Because this is all communication needed, it immediately shows a tremendous advantage of reinforcement learning relative to other learning methods; No complete environmental model is needed. Such a model could be used but is not required. Multiple reinforcement learning methods are able to solve problems without making use of a model of the environment.

The concept of an agent sending and receiving information also implies there must be a boundary between agent and environment somewhere. This boundary is placed in a way in which everything that is not under direct control of the agent is part of the environment. In most cases this does not match with the boundary between robot and environment because the agent is also confronted with actuators and a mechanical system which both possess properties not controlled by the agent. The agent-environment boundary is usually placed somewhere within the robot, comparable with the place at which nerves are attached to muscles within human beings.
2.2.1 Rewards

The reward signal is the sole tool we have, to tell an agent whether his behavior is correct or not. The computation of how much reward the agent receives for being into his current state, is not controlled by the agent and therefore is part of the environment. While determining a reward signal we should always keep in mind to only pass the goal in its most basic form. In reinforcement learning, finding a way in how the goal should be achieved is done by the solving method itself. This should not be imposed by the developer of the learning algorithm. The solving methods (described in chapter 3) will look for a policy to maximize long or average term reward, this policy coincides with the best way to reach the goal. Often ‘the goal in its most basic form’ is not the same as only putting a large reward value on a terminal state. Applied to maze running again, the agent would have no reason at all to finish his task fast as possible if there is only a reward released for reaching the terminal state. In maze running a fast route to the terminal state is definitely desired so this should be included in the reward signal. This could be done by delivering a negative reward (a punishment) for each step the agent takes. The reward for reaching the terminal state then becomes superfluous and could be replaced by a reward equal to zero. Not receiving punishment could also be interpreted as a reward.

In solving episodic tasks, the agent chooses to maximize the total reward \( R \) it expects to receive before it reaches the terminal state. This decision is made at every current point in time \( t \).

\[
R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T
\]  

(2.1)

Equation 2.1 is useless in solving continuing tasks. Continuing tasks could theoretically go on forever and therefore the equation for total reward could become infinite. An infinite amount of reward makes it difficult to compare one path to another. Furthermore a reward that is expected to be cashed in the near future should be given more importance than a reward expected in the far future because a lot of unexpected events can take place in the meantime. A discount rate \( \gamma \) could solve this issue and is implemented in the equation below. Only discount rates between zero and one are useful. Choosing \( \gamma = 0 \) would imply the agent maximizes only immediate reward (the agent is myopic). Choosing \( \gamma = 1 \) would make the agent ultimately foresighted and would change equation 2.2 into equation 2.1 again. A rate like \( \gamma = 0.85 \) is optimal in most cases, such a rate maximizes reward in the medium term.

\[
R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\]  

(2.2)

Adding a constant to every reward would make no difference for continuing tasks, only the spread between the rewards matters. For episodic tasks this is not always true, adding a constant could in these cases change
punishment into reward. Thinking of the maze running example, the punishment for every step taken by the agent could turn into a positive reward which would change the agents behavior entirely.

2.2.2 Actions

There is only one form of information that comes out of the agent. By choosing the appropriate actions a well learned agent can solve a problem in an optimal fashion. Actions are the outcome of a policy function which maps states to actions. In most cases the policy function does not choose directly which action is suited for the state it faces. Instead it consists of a probability distribution, every possible action in a certain state is accompanied of a likeliness of choosing that action. A common way of defining a policy is 'choose action \( a_1 \) 95% of the times you are facing this state and choose action \( a_2 \) in the remaining cases. Action \( a_1 \) and \( a_2 \) are part of the total action space \( (a_i \in A) \).

The policy function is commonly denoted using the \( \pi \)-symbol and an optimal policy (a problem could have multiple optimal policies) is denoted as \( \pi^* \). Because values represent the amount of reward that is expected in the future, the value function depends on policy choices. A change in policy would result in different choices in the same states and therefore would influence the value of states. The policy dependency of values is denoted by adding \( \pi \) as a superscript to the value function.

2.2.3 States

Our final building block is part of the information that goes into the agent, it is the basis on which the agent decides which action it should take. The state signal carries the coordinates of the agent’s position within the domain, but often more than that. It should contain exactly the amount of information necessary to determine the next state \( s' \) out of the current state \( s \) and the choice of an action \( a \). A state signal that meets this condition is said to possess the Markov property and a reinforcement learning framework with a state signal that possesses the Markov property is called a Markov Decision Process (MDP). The game of chess is a nice example of a Markov decision process. In chess, the next action can be chosen on the bases of the current configuration of the pieces. All information about how the pieces got into this configuration can easily be left out without consequences.

Mostly \( s' \) can not be predicted out of \( s \) and \( a \) with 100% accuracy. The signal is still said to meet the Markov property if the probability of reaching \( s' \) depends on \( s \) and \( a \) only. Equation 2.3 shows the Markov property using mathematical notation.

\[
Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \tag{2.3}
\]

As said in section 2.1, every state should have a value attached to it that represents how much reward is expected in the future after passing that state. The value is a representation of how desirable it is to be in a state and should equal the mathematical expectation (or expectation value) of reward receivable. Equation 2.4 is called the state value function.

\[
V^\pi(s) = E_\pi(R_t \mid s_t = s) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{2.4}
\]

The same principle could also be applied to a state-action pair. By attaching a value to a state-action pair instead of a state alone, an algorithm needs less computational effort to find the action that leads to the state with the highest value. The price of this improvement is an increased computational effort to determine the
values in the first place. In terms of state-action pairs the expectational value could be described as below. Equation 2.5 is called the action value function.

\[
Q^\pi(s, a) = E_\pi(R_t \mid s_t = s, a_t = a) = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\}
\]  
(2.5)

Equations 2.4 and 2.5 are not very practical to deal with. A solution method should have to take itself through all future time steps to determine the value of a state every time. This would cause a huge amount of unnecessary computations because the rewards after the next state (s') are already summarized in the value of the next state. An equation shaped like equation 2.6 would save us a lot of computing.

\[
\text{Value of current state (s) = Direct reward for reaching s' + Rewards after s'} \quad \frac{\text{Equal to value of s'}}{2.6}
\]

The above mentioned way to write the value function is possible and is shown in equation 2.7 for state values and in 2.8 for action values. These equations are referred to as a ‘Bellman equations’ (\(P_{sa}^a\) reflects the chance of reaching s' after choosing a and \(R_{sa}^a\) reflects the expected direct reward for reaching s').

\[
V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{sa}^a \left[ R_{sa}^a + \gamma V^\pi(s') \right]
\]  
(2.7)

\[
Q^\pi(s, a) = \sum_{s'} P_{sa}^a \left[ R_{sa}^a + \sum_{a'} \pi(s', a') \gamma Q^\pi(s', a') \right]
\]  
(2.8)

These Bellman equations are better understandable by visualizing one discrete step in time using a tree of states and actions (a backup diagram). Figure 2.2 shows a backup diagram for \(V^\pi\) (a) and \(Q^\pi\) (b). Open circles represent a state, filled ones represent a state-action pair. Both diagrams show one discrete step in time ending in the bottom of the diagram.

![Backup diagram](image)

Figure 2.2: Backup diagram for \(V^\pi\) (a) and \(Q^\pi\) (b), Source: Sutton en Barto [1] p70.

The values of all possible next states are used by the Bellman equation in order to determine the value of the current state. This means, in order to start an optimization method, an initial guess is needed. To shorten the learning process it could be smart to choose the initial guess very close to the final solution. In some cases however it can be smart to deliberately choose initial values that are way too high. These optimistic initial values force the agent to explore every possible action at least once because the policy will choose the action that leads to the highest value more frequently. Optimistic initial values are not very useful when dealing with non stationary problems because any environmental changes will not be explored after the initial value has been adjusted.
Solving a reinforcement learning problem is a quest for $\pi^*$, a policy that yields the most reward in the long run. Because of the policy dependency of value functions, an optimal policy will also lead to an optimal value function (equation 2.9 should be true for every single state). This concept is valid for action value functions as well.

$$V^*(s) = \max_\pi V^\pi(s)$$  \hspace{1cm} (2.9)

If we are dealing with a finite problem that fits the Markov property, it is possible to determine an analytical solution. By doing so, the Bellman equations transform into Bellman optimum equations (as shown in equation 2.10 for state value functions and in 2.11 for action value functions).

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a \left[ R_{ss'}^a + \gamma V^*(s') \right]$$  \hspace{1cm} (2.10)

$$Q^*(s,a) = \sum_{s'} P_{ss'}^a \left[ R_{ss'}^a + \max_{a'} \gamma Q^*(s',a') \right]$$  \hspace{1cm} (2.11)

By computing $V^*(s)$ basically what we are doing is reducing the difficulty of delayed reward to a simple choice on a local level. The greedy choice locally, always yields the most reward also in the long run as long as we are using the optimal value function. Unfortunately in most cases it is not that easy to determine the Bellman optimum equation. Often we are dealing with huge amounts of states while facing a limited computational and memory capacity. Furthermore in most cases the Markov property is not perfectly in place which also makes it more difficult to solve the Bellman optimal equation. The combination of these two shortcomings makes us look for an approximation instead of an exact solution, chapter 3 describes a variety of solution methods that can provide us with an approximation of the Bellman optimal equation.

### 2.3 Explore versus exploit

In section 2.2.3 value functions were introduced to solve the difficulty we had dealing with delayed rewards. The other difficulty we were facing was the trade off between moves that exploit knowledge we have and moves that explore new ground in order to increase our knowledge. Multiple methods exist to deal with this issue. All of them include one or multiple parameters that allow the user to tune the amount of greedy actions versus the amount of non greedy actions. Many applications profit from a shift in these parameters towards a more greedy policy while learning takes place. For stationary problems it probably is smart to fully stop taking exploring steps after the agent has been able to learn for a while. Non stationary applications will always need exploring actions (also after a long period of learning) because an action that has been optimal in the past might not be optimal anymore after a change in the environment.

#### 2.3.1 $\epsilon$ - greedy

The most common and most straightforward method is known as $\epsilon$-greedy. It does nothing more than choosing the greedy action with probability $1 - \epsilon$ and choosing random out of the non greedy (exploring) actions with probability $\epsilon$. When more than one action carries the highest value, it chooses random out of the multiple greedy actions. If the $\epsilon$ value is too low (as shown in figure 2.3), the solution will converge extremely slow towards the optimal solution and might even get stuck within a sub-optimal solution. Choosing $\epsilon$ too high also provides us with negative consequences, the solution will converge fast towards an optimal solution.
but takes an unnecessary large amount of exploring steps. By gradually decreasing the value of $\epsilon$ after each episode we could use best of both worlds (fast learning in the beginning and no unnecessary exploring while the optimal action is already known). There is no clear rule of thumb about what value of $\epsilon$ is best or how fast it should decrease. A lot of testing (plotting learning curves) and fine tuning is needed to find the optimal $\epsilon$ value. Such an optimization process for a maze running problem is described in [8] by Vivek Mehta and Rohit Kelkar.

2.3.2 Softmax action selection

Instead of $\epsilon$-greedy we could also use a more advanced method of choosing between explore and exploit. By using a Gibbs/Boltzmann distribution we could create a probability distribution depending on the values of the actions we could choose. When $Q_t(a)$ denotes the value attached to action $a$, the chance of choosing this action could be written as:

$$P(\text{action } a) = \frac{e^{Q_t(a)}}{\sum e^{Q_t(a)}}$$

In equation 2.12 the parameter $\tau$ is called the temperature of the distribution, it is a parameter to adjust the spread between different action possibilities. A high temperature would cause a flat distribution (almost equal chances for every action), a low temperature would cause the likeliness to choose the greedy action to increase. Again, in most applications it is clever to slowly decrease this parameter while more cycles have passed.

2.3.3 Alternative methods

The previous two methods each keep track of a value attached to an action and determine (using $\epsilon$-greedy or softmax) a distribution of preferences using these values. Alternatively we could also choose to only keep track of the preferences instead of the action values. In order to update the preference towards the best option, we need some sort of a reference of how good the reward we are receiving is (without values it is difficult to determine whether a certain reward is good or not so good at all). As a reference we could use the average of every reward received thus far $\bar{r}$. This method is called 'reinforcement comparison' and could be used as an alternative to the two action-value methods described in the previous sections. A reinforcement comparison update is described by equation 2.13 for the average of rewards and by equation 2.14 for the
update of preferences ($\alpha$ and $\beta$ both are learning rates between 0 and 1 and $p_t(a)$ is the preference for action $a$ before the update).

$$\bar{r}_{t+1} + \alpha[r_t - \bar{r}_t] \quad (2.13)$$

$$p_{t+1}(a_t) = p_t(a_t) + \beta[r_t - \bar{r}_t] \quad (2.14)$$

A second alternative is to keep track of action values and preference distributions. These methods are called pursuit methods because the preferences constantly pursue the greedy action (the action with the highest action value). Every time the state is passed the preference of the greedy action is incremented towards one, all other preferences are incremented towards zero. At the same time also action values are updated so the next time the state is passed it could be possible that the greedy position taken over by a different action.

Which of the above mentioned methods works best will depend on the problem you are facing. In order to choose one of the methods they could be compared using learning curves.
Chapter 3

Solution methods

Section 2.2.3 introduced the Bellman optimal equation which is the basis of a reinforcement learning problem. The solution of this equation yields a function that can be used to determine a path through states that produces the most reward and thereby solves the learning problem. This chapter describes three fundamental methods that provide us with an approximated solution to the desired value function (this is called ‘solving the prediction problem’). Every method has its advantages and disadvantages so in the last part of the chapter combinations of the three fundamental methods will be studied.

The basic methods can be sorted by characteristics like the use of bootstrapping, the use of a model of the environment and whether it includes on-policy or off-policy learning (a survey is shown in figure 3.1). The exact meaning of these characteristics will be explained later on in this chapter.

3.1 Dynamic programming

In order to solve a reinforcement learning problem, one could simply take equation 2.7 or 2.8 (the Bellman equations) and turn this equation into an update rule within an iterative process. Each of the iterative steps should contain a full sweep through state space at which for every possible state the current policy is evaluated and its value is updated according to expectations based on the values of successor states. The

Figure 3.1: Survey of basic reinforcement learning methods
previous sentence immediately shows dynamical programming uses bootstrapping. A method is said to use bootstrapping if it bases a prediction (value of current state) upon other predictions (values of successor states). Furthermore because of the use of Bellman equations dynamical programming requires a model of the environment. Both \( R_{ss'} \) (the expected reward for reaching \( s' \)) and \( P_{ss'}^a \) (the likeliness of reaching \( s' \) after choosing \( a \)) are parameters that can only be calculated using a model. This requirement automatically makes dynamical programming inappropriate for many reinforcement learning applications. Pure dynamic programming could even be considered not to be a reinforcement learning method in the first place because it learns out of sweeps over the state space, not out of real experience. Still it is a method worth studying because many thoughts out of dynamic programming are implemented in other methods and basic dynamical programming could be adapted into ‘asynchronous dynamical programming’ which does not not require full sweeps over the state space. Still, the algorithm in its most basic form is only useful to solve problems for which the agent is provided with a model.

Literally executing the above mentioned iteration is called ‘iterative policy evaluation’, this idea can be used to find the value function corresponding to a certain policy (solving the prediction problem). Policy evaluation solves only half the problem because after a policy evaluation also the policy itself should be updated in order to shift towards an optimal policy. A complete dynamical programming method therefore consists of both policy evaluation and policy improvement. The general concept of policy evaluation interacting with policy improvement processes is called ‘generalized policy iteration’ and most reinforcement learning methods can be brought down to this idea.

Adding policy improvement to the algorithm can be done in two different ways. One could use the Bellman equation to update the value function once, then update the policy once, then use the Bellman equation again etc. Eventually this iteration (referred to as ‘policy iteration’ pseudo code shown in appendix A) would yield an optimal policy with a corresponding optimal value function. One could also choose to slightly adapt the Bellman equation and instead use 3.1 as an update rule for policy iteration.

\[
V(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')] \tag{3.1}
\]

Updating values according to the action with maximum expected reward makes the policy evaluation converge towards an optimal value function instead of the value function corresponding to the current policy (as with policy iteration). The adaptation of the Bellman equation allows us to first completely determine the optimal values and afterwards turn to the policy improvement part, only to state that the optimal policy is the greedy policy. This method is called ‘value iteration’, pseudo code is shown in appendix A.e towards an optimal value function instead of the value function corresponding to the current policy (as with policy iteration). The adaptation of the Bellman equation allows us to first completely determine the optimal values and afterwards turn to the policy improvement part, only to state that the optimal policy is the greedy policy. This method is called ‘value iteration’, pseudo code is shown in appendix A.

Both policy and value iteration will mostly provide the same optimal policy with the same optimal values. Depending on the problem we are facing the one method can be a little faster than the other. Mehta and Kelkar [8] suggest using the ratio of action space size to state space size as an indication of which method converges faster. A high ratio favors policy iteration.

### 3.2 Monte Carlo methods

Unlike dynamical programming, Monte Carlo methods learn from experience only. In many ways it could be considered the most straightforward method of reinforcement learning. A Monte Carlo method determines the value of a state by simply passing it multiple times and averaging the total reward received after the
state has been passed. Because the total reward received is unknown before the episode has elapsed, Monte Carlo methods are only applicable to episodic tasks.Updating only after moving through state space has finished is called off-line updating (as opposed to updating while still moving through state space which is called on-line updating). A second important observation is that the values are based on true experience, not on values of successor states. This means the method does not use bootstrapping. Methods who do not bootstrap suffer less from violations of the Markov property.

Perhaps the biggest advantage of Monte Carlo methods over dynamical programming is the absence of calculations that require a model of the environment. Each backup is based on a truly experienced sum of rewards so averaging over many returns makes it unnecessary to know what the likeliness of the specific return was. In appendix B a simple example of a Monte Carlo method is demonstrated. It involves a soccer player that has to take penalty kicks and, by doing so, has to discover which out of four corners the goalie tends to cover least. The problem consists of only one state, four actions (a shot in one out of four corners) and a positive reward is distributed for every score. Even though it is a very easy single-state problem, it possesses many of the interesting parameters one can use to tune a Monte Carlo solution method. It also shows us that the amount of required episodes (shots) before the agent finds the optimal solution can become massive. When using only four actions (four corners), the algorithm converges within approximately 50 kicks. Using ten instead of four possible corners, the algorithm already requires hundreds of kicks in order to converge.

The values determined in the MATLAB example are action values which is common when using Monte Carlo methods because the absence of a model would make it difficult to determine which action is the greedy one using only state values. The policy function does need to know this in order to choose an action so choosing action values instead of state values avoids this obstacle.

An important question that arises is: What should we do with states that are visited more than once within the same episode? We could either choose to only use the total reward after the first visit or we could choose to average total rewards for each time the state is visited. Both 'first visit Monte Carlo' and 'every visit Monte Carlo' methods will converge. There is no clear evidence that one of these converges faster.

We could choose either to use an on-policy or an off-policy Monte Carlo method. The on-policy version uses the same policy that is updated and improved also as a guide while building up experience to learn from. After each cycle the policy is incremented towards the greedy choice (policy improvement) so after a while it becomes difficult to maintain enough exploration. This difficulty can be solved by using a random state as a starting point every episode (exploring starts). If we are only interested in the values of a subset of total state space, the choice of starting point could also be useful. By deliberately choosing only states within the subsection we are interested in, we can determine these values much faster than we would get them if the values for the entire collection of states had to be determined. In other words: Monte Carlo methods are easy to focus.

In some applications it is difficult in practice to choose a starting state (think of pole balancing for example). For these kind of situations off-policy control could be a solution to the problem of maintaining exploration. Off-policy control takes place if one policy is used to move through state space while a separate policy is updated after each cycle. The policy used to move through state space often is a soft policy. A policy is said to be soft if every action possible in a certain state has a non-zero likeliness of occurring. In order to determine a value corresponding to policy $\pi$ out of experience generated by policy $\pi'$ we have to weight the experienced sum of returns according to the relative probability of occurring (equation 3.2).

$$V^\pi(s) = \frac{\sum_{i=1}^{n_1} P_i(s) R_i(s)}{\sum_{i=1}^{n_1} P_i(s)}$$

In this equation $p_i(s)$ represents the probability of reaching $s$ using policy $\pi$ during episode $i$. This probability can be calculated out of policy choices in all states that could bring the agent towards state $s$. The ratio of
probabilities depends only on the two policies itself (not on the environment). Appendix A contains pseudo
code both for on-policy as for off-policy Monte Carlo control.

3.3 Temporal-Difference learning

The previous methods both own serious weaknesses, Monte Carlo is unable to solve continuing tasks and dy-
namic programming requires a model of the environment. Temporal-difference learning uses sample backups
(like Monte Carlo) and bootstrapping (like dynamic programming) at the same time and therefore combines
only the good parts of the previous methods. Instead of updating values only at the end of an episode
(using the sum of rewards), temporal-difference methods update values on-line. They use reward received for
reaching s’ and the value of s’ to compute a temporal-difference error that is multiplied by a learning rate
before the update is executed. An update using state values would look like equation 3.3 (an update using
action values would look analogous), \( \alpha \) represents a learning rate (typically around 0.1).

\[
V(s_t) = V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]  \tag{3.3}
\]

Same as with Monte Carlo, temporal-difference can be executed on-policy or off-policy. The on-policy version
is almost always based on action values for the same reason mentioned in section 3.2. During one backup
the agent chronologically goes through a ‘State, Action, Reward, next State, next Action’ cycle. Therefore
the on-policy temporal-difference method is called ‘Sarsa’. It is based on the action value version of equation
3.3, pseudo code of a Sarsa algorithm is depicted in appendix A.

Off-policy temporal-difference learning is called ‘Q-learning’ and uses equation 3.4 instead of equation 3.3 to
update its value function.

\[
Q(s,a) = Q(s,a) + \alpha [r + \max_{a'} \gamma Q(s',a') - Q(s,a)]  \tag{3.4}
\]

Including the max statement means backups are executed according to greedy choices only. Even on exploring
steps, the Q-learning algorithm backs up a temporal-difference error based on what would have been the
temporal-difference error in case of an exploiting step. In terms of convergence rate this is advantageous
because exploring moves could cause unnecessary disturbances. When the optimal action has been correctly
recognized already, an exploring step causes a wrong update of the previous value. These disturbances are
not there for Q-learning methods but this goes at the expense of losing information. For example if the agent
is into a state from which it will move towards a next ‘normal’ state 90 percent of the time. The remaining
10 percent the agent is forced (either by exploring steps or by noise) to move towards a terrible state that is
accompanied by a huge punishment. Q-learning would not include the consequences of the 10 percent risk
into the value of the current state, while Sarsa would. A well known example is one of a walking robot that
has to determine a shortest path from one point to another while walking next to a cliff. Every step taken by
the walking robot yields a small punishment but falling into the cliff yields very large punishment. Applying
a Q-learning algorithm would make the robot walk the absolute shortest path taking the risk of a small slip
that would make him fall into the cliff. Sarsa would always implement a buffer zone between cliff and path
even if that means the walking robot does not follow the absolute shortest path anymore. It depends on the
kind of problem we are facing whether Q-learning or Sarsa is optimal. Appendix A includes pseudo code for
a plain, one-step Q-learning method.

Not requiring a model and allowing continuing tasks makes temporal-difference the most widely applied
learning method. In the text above, temporal-difference is presented in its most elementary form. It includes
only temporal-difference errors over one discrete step in time and therefore this method is called ‘one-step
temporal-difference’. In the following section the temporal-difference concept is expanded by using eligibility
traces.
3.4 Eligibility traces

Section 3.2 introduced a method that creates value functions by making backups after the entire episode has finished (Monte Carlo), section 3.3 introduced a method that backs up information after every step (temporal-difference learning). Would it not be more effective to use an algorithm that makes use of temporal errors also to inform states that have been visited more than one step back in time? That is: Would it not be smart to update values also after n-steps, and somehow weigh the temporal errors according to their distance to the value that is going to be updated?

Eligibility traces provide us with a tool to arrange this. The use of an eligibility trace is indicated by adding $\lambda$ to the name of the learning method. For example: $TD(\lambda)$ stands for 'temporal-difference method with eligibility trace'.

In order to record whether a state is recently visited or not, we could attach an eligibility trace ($e$) to each state. This value should be increased when the state is visited and gradually decrease for every step the agent takes after visiting that state. How fast the eligibility trace decreases towards zero is indicated by $\lambda$ with $0 < \lambda < 1$. An eligibility trace then can be written as:

$$e_t(s) = \begin{cases} 
\gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\
\gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad (3.5)$$

Equation 3.5 represents an 'accumulating eligibility trace' because every time a state is visited, its eligibility trace is increased by one. Accumulating visits can cause problems, especially when using action values. Consider a situation for which taking the wrong action brings the agent back to the state he was already into. The agent takes the wrong action multiple times before choosing the right action which yields a reward. In such an occasion the eligibility trace of the wrong action might exceed the eligibility trace of the right one, which means the value of the wrong action is increased more than the value of the right one! To prevent this we could use 'replacing eligibility traces', shown in equation 3.6.

$$e_t(s) = \begin{cases} 
\gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\
1 & \text{if } s = s_t \end{cases} \quad (3.6)$$

The Sarsa method is well suited to implement eligibility traces. After every step the following procedure needs to be executed: (i) Assign eligibility traces to state-action pairs (ii) Calculate current temporal-difference error $\delta_t$ (iii) Update all values using $\alpha \cdot \delta_t \cdot e_t(s)$ as a state-specific error. Figure 3.3 shows the backup diagram for Sarsa($\lambda$). It consists of temporal-difference errors for every state-action pair that has been visited and the relative weight of each backup is indicated by the factor underneath each path.

The backup diagram makes clear that eligibility traces are able to combine Monte Carlo with the 1-step temporal-difference method derived in sections 3.2 and 3.3 respectively. Choosing $\lambda = 0$ provides us with the same backup diagram of 1-step temporal-difference and choosing $\lambda = 1$ coincides with Monte Carlo. More precisely, first-visit Monte Carlo corresponds to off-line $TD(\lambda = 1)$ with replacing eligibility traces and every-visit Monte Carlo corresponds to off-line $TD(\lambda = 1)$ with accumulating eligibility traces. Pseudo code of a Sarsa($\lambda$) algorithm is included in appendix A.

Because of the off-policy methodology, eligibility traces are difficult to implement in Q-learning methods. In section 3.3, Q-learning has been described as a method that backs up information out of exploiting steps only. This was easy in the previous section because the algorithms were backing up only one step, we only
Figure 3.2: Survey of backup diagrams of basic learning methods
had to add ‘maxₐ Q(s’, a’)’ in calculating the temporal-difference error. Q(λ) backs up more than one step so any backup that passes an exploring step does not obey the off-policy definition of Q-learning. We can deal with this issue in two different ways.

The fist one is called Watkin’s Q(λ). It simply sets all eligibility traces to zero after an exploring move has been made. Of course this is not very efficient because lots of information is thrown out of the window. Peng’s Q(λ) is different. It works like Sarsa(λ) in a sense because it backs up also when exploring moves have been made. The difference with Sarsa (λ) is that local temporal-difference errors are still calculated using a greedy policy. So locally the errors are calculated as if the policy always chooses the greedy option but these errors are backed up following a trace that could contain non-greedy choices as well. Peng’s Q(λ) is sometimes called a hybrid method because it converges neither to Qπ nor to Q∗ but to a hybrid of these. This phenomenon is caused by the contradiction in learning off-policy locally but backing up following an on-policy trace.

Although most research finds Peng’s Q(λ) to perform better than Watkin’s Q(λ), they both perform worse than Sarsa(λ). Therefore Sarsa(λ) is used in most applications.

In general eligibility traces are mainly useful when more information should be collected out of less data, especially when that data is hard (or slow) to gather. The extra updates that come with eligibility traces demand much computational expense so when data is relatively easy to gather it might be easier to simply collect more data instead of learning more out of the given amount of data.
Chapter 4

Scaling up to practical applications

The methods to solve a reinforcement learning problem as described in chapter 3 all have been presented in ‘tabular form’. Every solving method is presented as if there is a table that contains all possible states with their corresponding value. This will work in applications for which state space is small, but most practical problems consist of a huge amount of states. Contemporary computer technology still can not fill these enormous tables fast enough. Furthermore it might not be possible to visit every state at least once during the time available for training. This is especially true when we are facing a continuous state variable, for example the distance between two moving points in continuous space. Continuous state variables will yield training examples that only provide information about a single point, this information should be generalized to all points nearby. Therefore, in order to solve practical problems, somehow a function should be plotted through all state values that generalizes also to states that have never been visited. If this function consists of much less parameters than the total amount of states, the solving method will stay within limitations of computer capacity as well.

Function approximation, for the sake of generalization and fewer computational expense, is the next topic we are going to discuss. The \( \vec{\theta} \) symbol is used to represent a column of parameters that builds up the function that approximates the real value function. First thing we need is an equation that provides us with a measurement tool to see how well the actual values are approximated by the function. Equation 4.1 does exactly this, the Mean Squared Error (MSE) resembles the correctness of the approximation. \( P(s) \) is used to denote a weighting factor which grants higher importance to states that are visited more frequently.

\[
MSE(\vec{\theta}_t) = \sum_{s \in S} P(s)[V^\pi(s) - V_t(s)]^2
\]  

In order to make our algorithm learn accurately, equation 4.1 should be minimized. A set of parameters \( \vec{\theta}^* \) for which \( MSE(\vec{\theta}^*) \leq MSE(\vec{\theta}) \) is true for every state and every possible \( \vec{\theta} \) would be optimal but unfortunately this will only be possible for simple function approximators. More complex ones will only allow us to reach for a local optimum instead of a global one. This means an increase in accuracy at one place will always be at the expense of a decrease in accuracy somewhere else. Think of this as a water bed; Push it down somewhere and it comes up somewhere else. Because of this restriction in accuracy the weighting factor P should represent the distribution for which the mean squared error should be minimized (the distribution that represents the way states are actually appearing).

Function approximation is a topic not specifically bound to reinforcement learning. Many methods have been developed in other fields of science, some well-known methods are: (Linear) gradient-decent, memory-based and decision-tree methods. A thorough description of all function approximation methods goes beyond
the scope of this report so in the next section only linear gradient-decent (which is the method most easy to implement) will be discussed. Santamaria, Sutton and Ram wrote an article [9] that provides more information about other kinds of function approximation suitable for continuous space reinforcement learning problems. Any method of function approximation should be capable of dealing with a non stationary target function in order to be applicable to reinforcement learning. The target function is the true value function which keeps on changing during a learning process because of policy changes and because of possible changes in the environment, therefore this function is non stationary.

4.1 Linear gradient-decent

To make gradient-decent possible, \( \vec{\theta} \) should consist of a column with a fixed number of real valued components. Every state or action value can be written as a combination of these components. In linear gradient-decent this combination should be a linear one, \( V_t(s) = \theta(2) + \theta(5) \) could be an example. A general gradient-decent parameter update would look like equation 4.2. In this equation \( \alpha \) is a step-size parameter which plays a similar role as the learning rate in the algorithms shown in chapter 3. The \( V^\pi(s_t) \) parameter is unknown so instead we will have to use \( \upsilon \), a sample return.

\[
\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2} \alpha \nabla_{\vec{\theta}_t} [V^\pi(s_t) - V_t(s_t)]^2 = \vec{\theta}_t + \alpha [V^\pi(s_t) - V_t(s_t)] \nabla_{\vec{\theta}_t} V_t(s_t) \tag{4.2}
\]

Most interesting part of equation 4.2 is the nabla operator which takes the gradient of \( V^\pi(s_t) \) with respect to \( \vec{\theta}_t \). This operation makes sure the update is executed in the direction for which the error falls most rapidly. Because we are only discussing the linear form of gradient-decent, the gradient of a state or action value can only consist of a column of constant values (\( \vec{\phi}_s \) in equations 4.3 and 4.4).

\[
\nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}_s \tag{4.3}
\]

\[
V_t(s) = \vec{\theta}_t^T \cdot \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i) \phi_s(i) \tag{4.4}
\]

We can interpret \( \vec{\phi}_s \) as a state-specific vector that contains information about which parameters out of \( \vec{\theta} \) should be summed up in order to obtain the value of the current state. The practical meaning of this will become more clear in the following section. For now we have to remember that \( \vec{\theta} \) and \( \vec{\phi}_s \) should be of equal lengths (otherwise equation 4.4 would make no sense).

4.2 Coarse coding

Linear gradient-decent provides a tool to bring problems with enormous amount of states (or a continuous state space) within the range of computer capacity, it does not take care of generalization. In order to generalize experience from a state towards other states that are nearby, the concept of feature use must be introduced.

Imagine a one dimensional state space consisting of nothing more than ten meters of straight line. This continuous state space could be cut into ten parts (features). Referring to section 4.1, we could then construct a vector \( \vec{\phi} \) that possesses a parameter for every separate feature. If our agent is in the second feature, the second feature is said to be present and this is recorded by setting its value to one in \( \vec{\phi} \). Applying equation
4.4 we would find the value for each state equal to one of the parameters of $\theta$. This is a kind of rough way to generalize, particularly near the edges of features. A more smooth generalization would be obtained by using multiple compositions of features (tilings), each with a specific offset, within the same domain. Figure 4.1 visualizes this concept. The gray squares represent features who are present, the white ones all are absent. By using multiple tilings on the same domain, the generalization will be carried out more smooth and it allows adjacent states to share learning experience. The value of the state the agent currently is into can be determined by adding up the three parameters in $\theta$ corresponding to the present (gray) features. How much generalization occurs between two points in state space is determined by how much features they have in common. The idea of representing states with overlapping features is being referred to as 'coarse coding'.

Interesting to mention is that it is not the shape or size of features that determines the accuracy of generalization. The size of features can make generalization more narrow or broad and the shape of features could create asymmetric generalization but in the end its the total amount of features that determines the resolution.

Sometimes also generalization between two or more separate state parameter is desired (the process described above deals only with generalization within one state parameter). A state parameter with value A could, for example, only be desirable if another state parameter has value B. This kind of generalization is also possible using coarse coding, figure 4.2 shows a two dimensional example. One state parameter is plotted horizontally, the other one vertically. The gray scale shows how much generalization with state x will occur, state y has only one feature in common with x so there will be only slight generalization. Features in figure 4.2 are circular but this could be any arbitrary shape. Mostly plain grids (consisting of squares) are used, this type of coarse coding is called 'tile coding' or 'CMAC'. In figure 4.3 tile coding is depicted for a two dimensional state space. Richard Sutton describes tile coding routines for C++ on one of his websites [10].

### 4.3 Extension of solving methods

Adding linear gradient-decent function approximation to any of the learning methods derived in chapter 3 requires three algorithm changes. Using linear gradient-decent Sarsa as an example: (i) A feature vector should be build, indicating which features are present. This vector plays a similar role as $\phi_s$ in section 4.1 but is also sometimes written as $F_a$, to indicate the use of action values. (ii) The value of the current state should be calculated using equation 4.4. (iii) Eligibility traces should be attached to features, not to states. This causes updates to occur on function approximation parameters instead of state values. Pseudo code of linear gradient-decent Sarsa is provided in appendix A.

Bootstrapping methods are more difficult to combine with function approximation than non bootstrapping
Figure 4.2: Coarse coding two dimensional, source: Sutton en Barto [1] p203

Figure 4.3: Tile coding using two gridtilings, source: Sutton en Barto [1] p206
ones. In general, a bootstrapping method will generate a much higher mean squared error (MSE, equation 4.1). This is not necessarily problematic because the difference between state values creates behavior, not its absolute value. Adding 10 to every value would increase the MSE but would not change our agents behavior. For off-policy methods this can still be problematic. Combinations of linear gradient-decent function approximation with off-policy learning methods sometimes lead to divergence. Why this occurs is not always well understood. Baird’s counterexample is a well known theoretical problem in which the combination of dynamical programming and linear function approximation leads to a diverging iteration (at least for a certain range of initial values).
Chapter 5

Keepaway, a RoboCup-Soccer Subtask

In 2001, Richard Sutton and Peter Stone of The University of Texas at Austin proposed keepaway as a RoboCup-Soccer Subtask [3]. The keepaway players they designed were based on programming code originally written by Jelle Kok. His UvA Trilearn team [11] was very successful in the RoboCup simulation league. Gregory Kuhlmann, at that time a PhD student at The University of Texas at Austin, published a paper together with Peter Stone in 2003 [12]. Most of their previous empirical results were put together and published in a more extensive paper in 2005 [4]. Interesting progress was made by Kalyanakrishnan, Liu and Stone who published an article in which the keepaway code was extended towards half-field offense [5]. Currently Shivaram Kalyanakrishnan is most actively working on keepaway. He is associated to The University of Texas at Austin as a PhD student and maintains a website [13].

Source code that allows anyone to implement his or her own learning code on the keepaway players is available [6], appendix C can be useful during installation.

5.1 The game of Keepaway

Keepaway is a passing game involving keepers and takers within a certain bounded area. The keepers try to maintain possession of the ball while the takers are trying to steal it from them. It is not the amount of passes that has to be maximized by the keepers, they should try to increase total episode duration. An episode can end by either a taker gaining possession of the ball or by the ball going out of bounce. At every high level point of decision, the keeper in possession can decide whether he passes or holds the ball. He is not allowed to walk with the ball and the position on the court of his teammates is arranged deterministically. Taker behavior is also deterministic so learning is applied only on the pass decision. In our case the game involved three keepers playing against two takers on a square court of 20 x 20 length units.

5.1.1 Keepaway as a reinforcement learning problem

Deciding whether to pass or hold the ball is a high level decision. This means that agents who decide to call the 'PassBall' routine trigger a sequence of lower level routines. The time needed to execute these actions may vary. Therefore the system can not be fitted in a normal Markov decision problem. Instead the problem is called a SMDP (Semi Markov Decision Problem). This is a sequence of decisions who obey the Markov property but with unequal time intervals in between. Pass routines vary in time because of server misses and because sometimes the agent needs to execute some extra actions in order to find out his own position.
As described in section 2.2.1, the reward signal should only pass the goal in its most basic form. Since episode duration has to be maximized, the reward provided to the agent is chosen equal to the time difference between two consecutive SMDP steps. This reward is provided to the learning agent by passing the amount of server steps that have gone by. One server step equals 100 milliseconds.

In our case (3 keepers vs. 2 takers) the keeper in possession has to choose between 'hold', 'pass to teammate 1' and 'pass to teammate 2'. This is a relatively easy to handle, non-continuous action space. The state space is a little more difficult. With five agents and a ball, we have six elements who can change their position in a continuous way. It is obvious that this leads to millions of different state compositions. Therefore a smart selection of state variables is required. In the keepaway framework, these variable are already available. Eleven distances and two angles are provided to the learning agent in an array format further described in appendix D. All state variables are formulated with respect to the keeper in possession of the ball. This makes sense because this agent is the only one learning. When he passes the ball to a teammate the state variables are provided with respect to this next player. In figure 5.1 keeper 1 (K1) is in possession, the state variables are shown with blue lines/arcs.

The three building blocks (reward, state and action) now are defined so the keepaway problem has been mapped to a reinforcement learning problem. Next thing we need is some benchmark policies to compare our learning algorithm with. For this sake a random policy, a policy that always holds the ball and a hand coded policy is implemented in the keepaway player code. These methods provide an average episode duration which has to be beaten within few as possible episodes by the learning policy.

### 5.1.2 Previous results

Already in the first keepaway publication [3] the learned players were able to significantly outperform all of the benchmark policies in 3 vs. 2 keepaway. Learning curves were flattening around 12 seconds episode duration while the best performing benchmark policy averaged a little less than 6 seconds episode duration (shown in figure 5.2). The hand coded policy used in this publication performed only a little better than a random policy because it had not been tuned for this specific court size. Later on, tuned hand coded policies were used who performed just as good as the learned policies (but still not better). The difference in
performance between hand coded policies and learned ones seemed to increase when court size was decreasing. This implies reinforcement learning is especially useful in cases demanding more complex policies. Facing a smaller court, it can be more difficult for developers to create a set of rules that make up a hand coded policy.

Stone and Sutton used a Sarsa($\lambda$) algorithm (described in sections 3.3 and 3.4). Because the $\lambda$-value was set to zero the eligibility trace was not used, forcing the method to behave like plain Sarsa (without eligibility trace). Tile coding was used to deal with the continuous state space (as described in section 4.2). Generalization between different state variables was not considered desirable so tile coding was implemented one dimensional (similar to figure 4.1). Each state variable was covered by 32 different tilings, each with an offset equal to 1/32 of the feature size. For distances the feature size was 3.0 length units, for angles 10.0 degrees.

In the second keepaway publication [12], much of the simplifications in the previous paper were proven to be superfluous. Learning turned out to perform better than hand coded policies, also when noise was added to state info and when world info was limited because of narrow vision. Also many of the 13 state variables turned out to be redundant. Using only five state variables, performance was just as good. According to this paper, being a multi agent learning task is the real difficulty of keepaway. All three keepers are learning separately so three separate sets of values are being optimized. When only one keeper was learning while playing with teammates who used already optimized weights, he was able to reach the optimal policy much faster. Like humans, keepaway players learn faster in the presence of better teammates.

The third keepaway paper repeats the previous conclusions with extra proof but also adds an interesting comparison with Q-learning (described in section 3.3). Eventually both Sarsa and Q-learning seem to converge towards an optimal policy but Q-learning does this much slower and with a higher variability between runs (as shown in figure 5.3).

5.2 Writing a learning code

The keepaway framework is written in object-oriented C++. Tutorials provided by cplusplus.com [14] have been really useful in improving my programming skills and learning the object-oriented way of thinking.

LearningAgent.cc and LearningAgent.h are the files where one can implement self-made learning algorithms. They are provided on the keepaway site, but without the learning code. The empty file only contains a declaration of the LearningAgent class, who inherits four public methods from the SMDPAgent class. These are: startEpisode, step, endEpisode and setParams. In main.cc a LearningAgent object is created and send to KeepawayPlayer. The startup script (keepaway.sh) calls the 'keepaway_player' executable once for every player, so every player is running from its own executable and can learn its own set of values. This makes it difficult to implement inter-agent communication in the code.
5.2.1 Implementing Sarsa with linear tile coding

Sarsa was chosen as a solving method in keepaway because of multiple reasons. First of all it does not require a model of the environment (as policy or value iteration does). Such a model is not available because it would have to involve opponent behavior, which is unknown. The second reason is Sarsa being an on-line learning method, which means updates can be carried out immediately. An off-line method (like Monte Carlo) would have to keep track of all features who have been present during an episode. Our code uses 13 state variables and 32 tilings per variable, so for every SMDP step 416 integers would have to be stored in order to execute a Monte Carlo update. We are trying to maximize episode duration so the more successful we are, the more data needs to be stored to update values off-line. Because the keepaway algorithm in the future has to be useful for bigger problems (more players, bigger court etc.) the storing of features who have been present might become problematic. The last reason to choose Sarsa is that it is on-policy, which is known to converge faster than Q-learning [4]. Q-learning is the off-policy version of temporal-difference learning, it probably converges slower because it has more trouble correcting its mistakes. Once a wrong action is recognized as greedy in a certain state, it takes much more time to correct this when backups are carried out only according to the greedy policy.

Figure 5.4 is an overview of the most important private methods implemented within the virtual boundary of the agent. These methods are written according to standard Sarsa and one dimensional tile coding, described in sections 3.3 and 4.2. Changing parameters like tile width, learning rate and exploring rate can be done in the upper part of LearningAgent.h using ‘#Define’ statements.

A routine has been written to load and save weights automatically. In the weights folder a readme file is included that describes these functions. The structure of two important two dimensional arrays (weights and features_present) is explained in appendix D.

The full code of LearningAgent.cc and LearningAgent.h is copy-pasted in appendix E. To make this code work in the standard keepaway framework some small changes should be made in main.cc and keepaway.sh (the keeper player number has to be passed to the LearningAgent class). These changes are already in place in the files on the attached CD.

5.2.2 Experimental results

All simulations described below were carried out on 3 vs 2 games of keepaway, learning rates were equal to 0.125 and 13 state variables were used.
In the first simulation we tested the random policy. After 39 hours of learning the average episode duration turned out to be 5.443 seconds. This is equal to the value provided by the developers of the keepaway code. Learning time is not equal to simulation time because during simulations the server runs in sync mode which speeds up things five or six times.

After the benchmark policy was tested, a couple of simulations were carried out that all showed diverging, unstable behavior. Code 1, 2 and 3 (on the attached CD) each contained small bugs.

Code 4 was the first one to demonstrate a more or less stable behavior. In code 4 we used only one tiling per state variable (only a simple discretization). Distances were divided using tiles of 1.0 length unit, angles were divided in tiles of 5.0 degrees. A lack of generalization caused this algorithm to not even beat the random policy.

Next thing we investigated was the influence of $\varepsilon$. Code 5, 6 and 8 all use 32 tilings per state variable (3.0 length units tile width for distances, 10.0 degrees for angles). The learning curves for the highest exploring rate ($\varepsilon = 0.1$) was more steep in the beginning but could not reach the same optimal policy as the algorithm with $\varepsilon = 0.03$. Figure 5.5 shows learning curves for the three simulations.

Because the influence of $\varepsilon$ appeared to be strong, an extra function was added to the LearningAgent code to make $\varepsilon$ vary during the simulation (figure 5.6). Until 10000 episodes were executed, $\varepsilon$ was chosen to be stable at 0.1, it would than gradually decrease towards 0.01. Although indeed the learning curve was steep in the beginning, it did not reach a higher policy than policies using a steady $\varepsilon$. For future research it might be interesting to develop a code that increases $\varepsilon$ when the learning curve is flattening and decreases it when the curve is rising.
Figure 5.5: Learning curves for different $\epsilon$ rates

Figure 5.6: Learning curve of decreasing epsilon algorithm ($\epsilon$ in red)
5.3 Future work

Ultimately, the goal of developing keepaway algorithms is not only finding good policies to play keepaway. Subtasks like keepaway should improve our understanding of reinforcement learning theories, which in the end should lead to a learning algorithm that can optimize policies that are much more difficult to find than optimal keepaway policies. In RoboCup-Soccer this is especially challenging for the following reasons:

1. RoboCup involves a distributed, multi-agent domain (both teammates and opponents).
2. Sensors and actuators are noisy.
3. There is hidden state because of limited vision.
4. It involves only limited possibilities for communication.
5. Decisions have to be made in real time.

In simulated environments only the first one of these difficulties is always present. The other ones can be simulated but the ability to shut these down makes the simulation league particularly suited to develop new learning algorithms.

Future challenges include finding a way to deal with 'the curse of dimensionality'. Adding a player to the keepaway court approximately doubles the time required to find an optimal solution. A second challenge is finding methods of function approximation that never cause learning algorithms to become unstable. Also more sophisticated means of parameter optimization should be developed because today this is done mainly by trial and error.

It could also be interesting to create learning algorithms that learn more out of less data. This way it could become possible to search for optimal policies in real time (during a single game) depending on the opponents behavior. The first thing that can be done to learn more out of less data is expanding the Sarsa algorithm in LearningAgent.cc with eligibility traces. This would steepen the learning curve because more backups are made out of the same data. It would also increase computational cost on the other hand.
Chapter 6

Conclusion

The theoretical part of this report has shown reinforcement learning as a broad area of science. Solving methods can be organized according to different characteristics like whether or not they use a model of the environment, whether or not they use bootstrapping and whether learning occurs on-policy or off-policy.

A wide range of problems can be solved using reinforcement learning. As long as the system obeys the Markov property and can be described in terms of the three building blocks (rewards, states and actions) the problem is suited. This applies to maze running, playing chess, shooting penalty kicks and many other problems. Function approximation has even extended the range of possible applications, making also continuous state spaces suitable for reinforcement learning.

RoboCup-Soccer related possibilities to apply reinforcement learning are widely available. Keepaway is an example of learning applied to high level policies, the same could also be done for low level policies like optimizing passing or shooting routines. Developing a self made learning algorithm proved reinforcement learning to be possible also with limited programming experience.

The simulations in section 5.2.2 demonstrated the influence of the exploring rate. A higher exploring rate steepens the learning curve initially but often causes the learning agent to behave suboptimal hereafter.

Future applications of reinforcement learning should especially be searched for in areas where deterministic (hand coded) policy rules are not that obvious. Learning algorithms can come up with smart solutions as long as they are provided with enough training episodes.

Currently, the biggest bottleneck of the algorithms described in the previous chapters is the difficulty to scale them to larger domains with more agents. This is a challenging, but rewarding, task for the future.
Bibliography


Appendix A

Pseudo code of various solution methods

1. Initialization
\[ V(s) \in \mathbb{R} \text{ and } \pi(s) \in \mathcal{A}(s) \text{ arbitrarily for all } s \in \mathcal{S} \]

2. Policy Evaluation
   Repeat
     \[ \Delta \leftarrow 0 \]
     For each \( s \in \mathcal{S} \):
     \[ v \leftarrow V(s) \]
     \[ V(s) \leftarrow \sum_{a'} \mathcal{P}_{s,a'} \left[ \mathcal{R}_{s,a'} + \gamma V(s') \right] \]
     \[ \Delta \leftarrow \max(\Delta, |v - V(s)|) \]
   until \( \Delta < \delta \) (a small positive number)

3. Policy Improvement
   policy-stable \leftarrow true
   For each \( s \in \mathcal{S} \):
   \[ b \leftarrow \pi(s) \]
   \[ \pi(s) \leftarrow \arg \max_a \sum_{a'} \mathcal{P}_{s,a'} \left[ \mathcal{R}_{s,a'} + \gamma V(s') \right] \]
   \[ \text{If } b \neq \pi(s), \text{ then } \text{policy-stable} \leftarrow false \]
   \[ \text{If } \text{policy-stable}, \text{ then stop; else go to 2} \]

Figure A.1: Dynamic Programming (Policy Iteration), source: Sutton en Barto [1] p98
Initialize $V$ arbitrarily, e.g., $V(s) = 0$, for all $s \in S^+$

Repeat
\[
\Delta \leftarrow 0
\]
For each $s \in S$:
\[
v \leftarrow V(s)
\]
\[
V(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]
\]
\[
\Delta \leftarrow \max(\Delta, |v - V(s)|)
\]
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
\[
\pi(s) = \arg \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]
\]

Figure A.2: Dynamic Programming (Value Iteration), source: Sutton en Barto [1] p102

---

Initialize, for all $s \in S$, $a \in A(s)$:

- $Q(s, a) \leftarrow$ arbitrary
- $\text{Returns}(s, a) \leftarrow$ empty list
- $\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Generate an episode using $\pi$
(b) For each pair $s, a$ appearing in the episode:
\[
R_e \leftarrow \text{return following the first occurrence of } s, a
\]
Append $R_e$ to $\text{Returns}(s, a)$
(c) For each $s$ in the episode:
\[
a^* \leftarrow \arg \max_a Q(s, a)
\]
For all $a \in A(s)$:
\[
\pi(s, a) \leftarrow \begin{cases} 
1 - \frac{\epsilon}{|A(s)|} & \text{if } a = a^* \\
\frac{\epsilon}{|A(s)|} & \text{if } a \neq a^*
\end{cases}
\]

Figure A.3: Monte Carlo (on-policy), source: Sutton en Barto [1] p125

---

Initialize, for all $s \in S$, $a \in A(s)$:

- $Q(s, a) \leftarrow$ arbitrary
- $N(s, a) \leftarrow 0$ \quad ; Numerator and
- $D(s, a) \leftarrow 0$ \quad ; Denominator of $Q(s, a)$
- $\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Select a policy $\pi'$ and use it to generate an episode:
\[
s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{t-1}, a_{t-1}, r_t, s_t
\]
(b) $\tau$ \quad latest time at which $a_t \neq \pi(s_t)$
(c) For each pair $s, a$ appearing in the episode after $\tau$:
\[
t \leftarrow \text{the time of first occurrence (after } \tau \text{) of } s, a
\]
\[
w \leftarrow \prod_{k=t+1}^{T} P_{ss'}^{a_k} \prod_{k=t+1}^{T} [1 - \pi(s_k, a_k)]
\]
\[
N(s, a) \leftarrow N(s, a) + w
\]
\[
D(s, a) \leftarrow D(s, a) + w
\]
\[
Q(s, a) \leftarrow \frac{N(s, a)}{D(s, a)}
\]
(d) For each $s \in S$:
\[
\pi(s) \leftarrow \arg \max_a Q(s, a)
\]

Figure A.4: Monte Carlo (off-policy), source: Sutton en Barto [1] p127
Figure A.5: Temporal-difference (Sarsa, 1-step), source: Sutton en Barto [1] p146

Figure A.6: Temporal-difference (Q-learning, 1-step), source: Sutton en Barto [1] p149

Figure A.7: Temporal-difference, Sarsa($\lambda$), source: Sutton en Barto [1] p181
Figure A.8: Linear, gradient-decent Sarsa(\(\lambda\)), source: Sutton en Barto [1] errata
Appendix B

MATLAB example Monte Carlo method

```matlab
%% Robin Soetens – juni 2010
1
2 - clsc
3 - clsc
4 - close
5 - clc
6 - figure
7
8 % This m-file is a straightforward example of reinforcement learning
9 % applied to a soccer player who is able to aim at multiple corners while
10 % taking penalty kicks. After a certain amount of shots, the player learns to avoid
11 % the corners the goalie tends to choose. The problem consists of only one state and
12 % an action space equal to the amount of corners. It distributes adjustable rewards
13 % for goals who are stopped or not stopped.
14
15 % Declaration of parameters
16
17 - corners = 4; % Number of corners possible (action space)
18 - covered = 3; % Number of corners the goalie tends to choose (it chooses random which one of these to cover in each episode)
19 - delay = 0.25; % Delay between episodes (in order to visualize)
20 - epsilon = 0.2; % Fraction of exploring steps, gradually decreases towards zero
21 - kicks = 100; % Number of episodes
22 - stopped = 0; % Reward on a kick stopped by the goalie
23 - made = 1; % Reward on a kick not stopped by the goalie
24 - cavorall = 1; % Choose 1 to occasionally also cover a non-covered corner
25 - initial = 2; % Initial values (could be chosen optimistic)
26
27 % Initialisation
28
29 - for i = 1 : corners
30 - Q(i) = initial;
31 - end
32
33 % Penalty kicks
34
35 - for i = 0 : kicks
36    % Goalie uses uniform random distribution to choose out of covered corners
37    goalie = round( rand * covered + 0.5 );
38    if (goalie == 1)
39       - random = rand;
40       - uncovered = corners - covered;
41       - fraction = uncovered / corners | | 5 * uncovered ;
```

Figure B.1:

38
if random < fraction
    % One out of 20 kicks the goalie chooses a non covered corner randomly
    noncovered = corners - covered;
    goalie = round ((rand * noncovered + 0.5) + covered);
end

% Player chooses corner using epsilon-greedy
random = rand;
epsilon = epsilon * exp (0.97 * 1) / exp (1);
if random < epsilon
    % Choose random out of all corners
    player = round ((rand * corners + 0.5));
else
    % Choose greedy option, in case of multiple options, first one found
    [a,player] = max(a);
end

% Plotting current episode
plot (goalie, 0, 'o', 'MarkerSize', 13)
hold on
plot (player, 0, 'b', 'MarkerSize', 13)
axis ([0 corners + 1] [-1 1])
legend('Goalie', 'Player')
hold off

% Determining reward
if player == goalie
    reward = stopped;
else
    reward = make;
end

% Update values (average of reward received)
if Q(player) == initial
    % In case of a first visit
    Q(player) = reward;
else
    % Use incremental implementation to calculate average
    Q(player) = Q(player) + 1/d * (reward - Q(player))
end

% Delay (in order to visualize)
sleep (delay)
end

Note: The required function file ‘myWait.m’ can be downloaded at: www.mathworks.com/matlabcentral/...
...fileexchange/4194-delay

Figure B.2:
Appendix C

Installation RoboCup simulator with Keepaway players (tutorial)

This appendix describes the installation procedure for a RoboCup simulation league server and compatible keepaway players provided by the University of Texas at Austin. After installation a framework suitable to implement and test self made learning codes is available. The basic tutorial found on the following url has been of great use.

http://userweb.cs.utexas.edu/~AustinVilla/sim/keepaway/

While solving errors during installation, the keepaway mailing list (keepaway@utlists.utexas.edu) helped me out. Special thanks to Sam Devlin of The University of York for his useful assistance.

If you are not familiar using the Ubuntu terminal commands read this page first:


Below I have put an adapted version of the first part of the basic tutorial making it better suitable to install the framework on recent Ubuntu distributions.

I carried out the following procedure running Ubuntu 8.04 - Hardy Heron, without any updates. This Ubuntu distribution features GCC and g++ version 4.2.3. I also tried to install keepaway on Ubuntu 10.04 - Lucid Lynx, but this requires a lot of changes to be made in files. At the end of this tutorial a few of these are mentioned, just to get you started.

Download and extract rcssserver, rcssbase and rcssmonitor (each version 11.1.0). They are probably still available on Sourceforge

http://sourceforge.net/projects/sserver/files/

If not, you will find them here:

http://ftp.heanet.ie/disk1/sourceforge/s/project/ss/ssserver/OldFiles/

Also download and extract the keepaway players (version 0.6), you will find 'keepaway-0.6.tar.gz' on the before mentioned keepaway website.

Make a directory which is going to contain the installation files, probably something like (replace 'username' with your own username):

[40]
In order to allow the software to find this directory, two environment variables should be declared. By adding this declaration of variables to a script that is automatically executed during Ubuntu’s startup procedure, the RoboCup software will be able to find the installation directory permanently. Open the file ‘profile’ found in root directory `/etc` and add the following lines:

```bash
export RCSSBASEDIR=/home/username/rcss
export RCSSBASE=$RCSSBASEDIR
export PATH=$RCSSBASEDIR/bin:$PATH
```

Before proceeding with the actual compiling, a couple of packages are needed. Open the Synaptic Package Manager and install the following packages:

- g++
- libboost-dev
- libboost-filesystem-dev
- bison
- flex
- libx11-dev
- libxpm-dev
- xutils-dev
- csh
- gnuplot
- gnuplot-nox
- gv

The last four are only required to compile and use the additional tools provided with the keepaway players. These are tools useful for manipulating and plotting data output.

Now navigate to the folder rcssbase-11.1.0 and execute the following terminal commands:

```bash
./configure --prefix=$RCSSBASEDIR --with-boost-filesystem=boost_filesystem-mt
make
make install
```

Navigate to the folder rcssserver-11.1.0 and compile using the same procedure, hereafter run the same commands also for rcssmonitor-11.1.0. By now the RoboCup simulator is installed. Check this by executing ‘rcssserver’ in a terminal and ‘rcssmonitor’ in a new terminal. The server should recognize a monitor connecting.

To install the keepaway players navigate to keepaway-0.6 and open the ‘player’ folder. Open ‘Makefile’ and change ‘makedepend’ into ‘gecmakedep’. Now compile the keepaway players by executing the following commands in the player folder:

```bash
make depend
make
```

Install the additional tools by navigating to the tools directory and execute ‘make’ there as well.
Copy the binaries you have created (hist / killserver / kunzip / kwyzipper / kzip / monitor / winsum) to a directory in your path. For example:

/usr/local/bin

To start a keepaway simulation, the script file ‘keepaway.sh’ must be executed. Open this script first and change #!/bin/sh into #!/bin/bash to force Ubuntu to use the Bourne Again Shell (bash).

Your computer is now ready to play its first round of keepaway! Execute the terminal command ./keepaway.sh in the keepaway folder and then execute the monitor script by typing ‘monitor’ and you will see a 3 vs. 2 game of keepaway.

In order to plot learning curves, use the commands provided in the basic tutorial (on the keepaway website). You will use winsum to create a ’1.out’ file, this file can be plotted using gnuplot. Navigate to the directory containing ’1.out’ and execute:

```bash
gnuplot
plot "1.out"
```

The Learning Agent Tutorial (also on the keepaway website) provides additional information to add LearningAgent.cc and LearningAgent.h to the keepaway framework. Your own learning code should be implemented in these two files. Adding them to the keepaway player framework should produce no additional errors.

Installing the keepaway framework on today’s most recent Ubuntu version (Ubuntu 10.04 - Lucid Lynx) requires lots of changes to be made in code files. Just to get you started;

The robocup soccer simulator version 11.1.0 is difficult to install. Instead try rcssserver-14.0.3 (available on Sourceforge), rcssbase is already integrated in this server version. Use rcssmonitor version 11.1.0 or one of the newer ’classic versions’.

In order to run ’make depend’ in the keepaway player folder, install ’build-essential’ through the package manager and change makedepend into gccmakedep in the makefile.

Newer versions of gcc require some additional header file specification. In the player folder, add:

```c
#include <cstring>
```

To WorldModelHighLevel.cc / WorldModelUpdate.cc / Acthandler.h / Logger.cc and LoggerDraw.cc. Furthermore, add:

```c
#include <stdlib.h>
```

To WorldModelUpdate.cc / HandCodedAgent.cc and in the tools folder to kwyzipper.cc.
Appendix D

Array structures in LearningAgent

state[ . ]

The following array of state variables is provided to methods startEpisode, step and endEpisode. All variables are defined with respect to the player in possession of the ball.

\[
\begin{align*}
\text{state}[0] &= \text{Distance to center of court.} \\
\text{state}[1] &= \text{Distance to nearest teammate.} \\
\text{state}[2] &= \text{Distance to other teammate.} \\
\text{state}[3] &= \text{Distance to nearest opponent.} \\
\text{state}[4] &= \text{Distance to other opponent.} \\
\text{state}[5] &= \text{Distance nearest teammate to center of court.} \\
\text{state}[6] &= \text{Distance other teammate to center of court.} \\
\text{state}[7] &= \text{Distance nearest taker to center of court.} \\
\text{state}[8] &= \text{Distance other taker to center of court.} \\
\text{state}[9] &= \text{Distance nearest teammate to his nearest opponent.} \\
\text{state}[10] &= \text{Distance other teammate to his nearest opponent.} \\
\text{state}[11] &= \text{Angle made by nearest teammate with his nearest opponent.} \\
\text{state}[12] &= \text{Angle made by other teammate with his nearest opponent.}
\end{align*}
\]

features_present[ . ][ . ]

The elements in the features_present matrix indicate a certain feature that the agent is currently into. When 3 state variables each with 2 tilings of 10 features are used, the total amount of features available is \(3 \times 2 \times 10 = 60\). This means a value in features_present never is above 59 (not 60 because 0 is also included). In this example \(3 \times 2 = 6\) features will be present and \(3 \times 2 \times 9 = 54\) features will not be present. Features_present is
build two dimensional, with the first index indicating the state variable and the second one indicating which
tiling within that state variable. In other words: The feature who is present in the first tiling of the second
state variable can be found at features_present[1][0].

weights[ . ][ . ]

Tile coding requires a weight for every feature, in the above mentioned example this would imply 60 weights.
Because we are using action values a separate weight is needed for every possible action. Using 3 vs 2
keepaway as an example, there are three action possibilities so we need 60 * 3 = 180 weights. The ‘weights’
array in LearningAgent is a two dimensional one with the first index representing the feature that is present
and the second one the action that is chosen. For example, the weights corresponding to the three actions
possible when the 7th feature is present can be found at weights[6][0], weights[6][1] and weights[6][2].
Appendix E

Learning agent code

E.1 LearningAgent.cc

#include "LearningAgent.h"
#include <math.h> // For power raising in getEpsilon

*****************************************************************************/
* Bachelor Thesis Robin Soetens (TU/e mechanical engineering)
* June, July, August 2010
* Original file downloaded at:
* http://userweb.cs.utexas.edu/~AustinVilla/sim/Keepaway/
* ***************************************************************************/

// Constructor (numFeatures and numActions are required)
换句话来说，（numFeatures和numActions是必填的）

LearningAgent::LearningAgent( int numFeatures, int numActions, int keeper_number):
    SMDPAgent( numFeatures, numActions )
{
    printf("Keeper number %d is being constructed.\n", keeper_number);
    strcpy ( weights_dir, "/home/s070476/keepaway-0.6/weights/");
    // MANUALLY ADJUST THIS, ALSO IN SAVEWEIGHTS !!!!
    my_number = keeper_number;
    loadWeights( weights_dir, weights, keeper_number );
    episode_count = 0;
    epsilon = getEpsilon( episode_count );

    if(keeper_number == 1)
    { printf("Learning rate = %f\n\tExploring rate = %f\n\tTilings per variable: %d\n", LEARNING_RATE, epsilon, TILINGS_PER_VAR); }
}
Public Methods

// Public Methods
//-------------------------------------------------------------------------------------

int LearningAgent::startEpisode( double state[] )
{
    epsilon = getEpsilon( episode_count );
    getFeaturesPresent( state, last_features_present );
    getCurrentActionValues( last_features_present, current_action_values, weights );
    last_action = selectAction( current_action_values, epsilon );
    last_action_value = current_action_values[last_action];
    return last_action;
}

//-------------------------------------------------------------------------------------

int LearningAgent::step( double reward, double state[] )
{
    if( rand() %200 == 0 )
    {
        printf("reward step: %f\n", reward );
    }
    epsilon = getEpsilon( episode_count );
    if( rand() %200 == 0 && DECREASE_EPSILON == 1 )
    {
        printf("episode: %d epsilon %f\n", episode_count, epsilon );
    }
    getFeaturesPresent( state, features_present );
    getCurrentActionValues( features_present, current_action_values, weights );
    current_action = selectAction( current_action_values, epsilon );
    current_action_value = current_action_values[current_action];
    if( rand() %200 == 0 )
    {
        printf("current action values: %f %f %f\n", current_action_values[0],
                current_action_values[1], current_action_values[2]);
    }
    temporal_error = current_action_value + reward - last_action_value;
    update( last_features_present, temporal_error, last_action, weights );
    last_action = current_action;
    memcpy( last_features_present, features_present, TILINGS_PER_VAR * STATE_VARIABLES
                * sizeof( int ) );
    last_action_value = current_action_value;
    return last_action;
}

//-------------------------------------------------------------------------------------

void LearningAgent::endEpisode( double reward )
{
    temporal_error = 0.0; // Reward in endEpisode set to zero!
    update( last_features_present, temporal_error, last_action, weights );
    if ( episode_count % 500 == 0 )
    {
        saveWeights( weights_dir, my_number, episode_count, weights );
    }
    episode_count++;
}

46
void LearningAgent::setParams(int iCutoffEpisodes, int iStopLearningEpisodes)
{
}

// Private Methods

void LearningAgent::loadWeights ( char* weights_dir, double weights[][ACTION_POSSIBILITIES],
                                        int keeper_number )
{
    //Look for initial values file
    strcpy ( filename , weights_dir);
    strncat( filename , "weights_file.log" , 60);
    weights_file = fopen( filename, "r" );

    //If initial values file does not exist, check for previous saved weights
    if( weights_file == NULL )
    {
        strcpy ( filename , weights_dir);
        sprintf( buffer , "saved_player_%d.log", keeper_number);
        strncat( filename , buffer , 60);
        weights_file = fopen( filename, "r" );
        if( weights_file != 0)
        {
            strcpy ( filename , filename_saved_weights );
            printf("Saved weights player %d are found.\n", keeper_number);
        }
    }

    //If initial values and saved values don't exist, create initial values file
    //(initial weights/values are zero!)
    if( weights_file == NULL )
    {
        weights_file = fopen( filename, "w" );
        for( j = 0 ; j < TOTAL_FEATURES ; j++ )
        {
            for( i = 0 ; i < ACTION_POSSIBILITIES ; i++ )
            {
                fwrite("0.0\t", 4, 1, weights_file);
            }
            fwrite("\n", 1, 1, weights_file);
        }
        printf("Weights file with initial values is created.\n");
    }

    if( weights_file == NULL )
    { printf("\nFailed to open/create save weights file in method 'loadWeights'\n");}

    //Copy file to working memory if file exists
    if( weights_file != NULL )
    {
        fflush( weights_file );
        fclose( weights_file );
    }
}
weights_file = fopen( filename, "r" );
rewind( weights_file );
for( i = 0 ; i < TOTAL_FEATURES ; i++ )
{
    fgets( data_str_line, 150, weights_file);
    j = 0;
data_str = strtok( data_str_line , "\t" );
while( data_str != NULL )
{
    weights[i][j] = atof( data_str );
data_str = strtok( NULL , "\t" );
j++;
}
fclose(weights_file);
printf("Weights file is found and loaded.\n");
}

void LearningAgent::saveWeights ( char *filename , int my_number , int episode_count , double weights[][ACTION_POSSIBILITIES] )
{
    //Create file name
    strcpy ( filename , "/home/s070476/keepaway-0.6/weights/");
k = sprintf( buffer , "episode_%d_player_%d.log", episode_count, my_number);
if ( k < 0 )
{
    printf("\nFilename exceeds buffer size in method 'saveWeights'.\n");
}
strncat( filename , buffer , 60);
//Write data to file
weights_file = fopen( filename, "w");
close(weights_file);
weights_file = fopen( filename, "a");
for( j = 0 ; j < TOTAL_FEATURES ; j++ )
{
    // Does not automatically adjust for amount of action possibilities!!
    fprintf( weights_file, "%e\t%e\t%e\n", weights[j][0], weights[j][1], weights[j][2]);
}
close( weights_file );
printf("Weights data player %d saved, episode %d.\n", my_number, episode_count);
}

void LearningAgent::update ( int last_features_present[][TILINGS_PER_VAR], double temporal_error, int last_action, double weights[][ACTION_POSSIBILITIES] )
{
    for( i = 0 ; i < STATE_VARIABLES ; i++ )
    {
        for( k = 0 ; k < TILINGS_PER_VAR ; k++ )
        {
            weights[ last_features_present[i][k] ][last_action] += LEARNING_RATE *
temporal_error / (float) STATE_VARIABLES / (float) TILINGS_PER_VAR;
        }
    }
}
int LearningAgent::selectAction( double current_action_values[], double epsilon )
{
    //Find greedy actions
    greedy_action_value = current_action_values[0];
    greedy_actions[0] = 0;
    amount_greedy_actions = 1;
    for( i = 1 ; i < ACTION_POSSIBILITIES ; i++ )
    {
        if( current_action_values[i] > greedy_action_value )
        {
            greedy_action_value = current_action_values[i];
            greedy_actions[0] = i;
            amount_greedy_actions = 1;
        }
        if( current_action_values[i] == greedy_action_value )
        {
            greedy_actions[ amount_greedy_actions ] = i;
            amount_greedy_actions++;
        }
    }
    //Choose action using epsilon-greedy method
    random_number = 0.01 * ( rand() % 100 );
    if( random_number < epsilon )
        return rand() % ACTION_POSSIBILITIES;
    else
        return greedy_actions[ rand() % amount_greedy_actions ];
}

void LearningAgent::getFeaturesPresent( double state[], int features_present[][TILINGS_PER_VAR] )
{
    for( i = 0 ; i < STATE_VARIABLES ; i++ )
    {
        for( k = 0 ; k < TILINGS_PER_VAR ; k++ )
        {
            j = - 1;
            state_current = state[i];
            if( i < 11 )
                state_current += FEATURE_SIZE_LINE * (float) k / TILINGS_PER_VAR;
            else if( i > 10 )
                state_current += FEATURE_SIZE_ANGLE * (float) k / TILINGS_PER_VAR;
            while( state_current >= 0 )
            {
                j++;
                if( i < 11 )
                    state_current -= FEATURE_SIZE_LINE;
                else if( i > 10 )
                    state_current -= FEATURE_SIZE_ANGLE;
            }
        }
    }
}
features_present[i][k] = j + i * FEATURES_PER_TILING * TILINGS_PER_VAR + k * FEATURES_PER_TILING;

void LearningAgent::getCurrentActionValues( int features_present[][TILINGS_PER_VAR],
                                            double current_action_values[],
                                            double weights[][ACTION_POSSIBILITIES] )
{
    for( j = 0 ; j < ACTION_POSSIBILITIES ; j++ )
    {
        current_action_values[j] = 0.0;
        for( i = 0 ; i < STATE_VARIABLES ; i++ )
        {
            for( k = 0 ; k < TILINGS_PER_VAR ; k++ )
            {
                current_action_values[j] += weights[ features_present[i][k ]][j];
            }
        }
    }
}

double LearningAgent::getEpsilon( int episode_count )
{
    epsilon = EPSILON_FRACTION;
    if ( DECREASE_EPSILON == 1 && episode_count > 10000 )
    {
        epsilon = 0.2 * pow( 0.5 , (double) episode_count / 10000.0 ) + 0.01;
    }
    return epsilon;
}

E.2 LearningAgent.h

#ifndef LEARNING_AGENT
#define LEARNING_AGENT

#include "SMDPAgent.h"

#define STATE_VARIABLES 13
#define FEATURES_PER_TILING 20
#define ACTION_POSSIBILITIES 3
#define FEATURE_SIZE_LINE 3.0
#define FEATURE_SIZE_ANGLE 10.0
#define EPSILON_FRACTION 0.03
#define LEARNING_RATE 0.125
#define TILINGS_PER_VAR 32
#define TOTAL_FEATURES FEATURES_PER_TILING * TILINGS_PER_VAR * STATE_VARIABLES

50
#define DECREASE_EPSILON 0

// Note: Do not forget to manually reset episode_count
// after restarting a simulation with decreasing epsilon

/******************************************************************************
 *
 * Bachelor Thesis Robin Soetens (TU/e mechanical engineering)
 * June, July, August 2010
 * Original file downloaded at:
 * http://userweb.cs.utexas.edu/~AustinVilla/sim/Keepaway/
 *
******************************************************************************/

#include <string.h>  // for memcpy and strcpy
#include <stdlib.h>  // for rand
#include <stdio.h>   // for testing

class LearningAgent:public SMDPAgent
{
    // Private methods and variables
    double epsilon;
    double last_state[MAX_STATE_VARS];
    int last_action;
    int i, j, k;
    double state_current;
    int episode_count;
    int features_present[STATE_VARIABLES][TILINGS_PER_VAR];
    int last_features_present[ STATE_VARIABLES ][TILINGS_PER_VAR];
    double weights[TOTAL_FEATURES][ACTION_POSSIBILITIES];
    double current_action_values[ACTION_POSSIBILITIES];
    char data_str_line[150];
    char* data_str;
    int greedy_actions[ ACTION_POSSIBILITIES ];
    int current_action;
    double greedy_action_value;
    double last_action_value;
    double current_action_value;
    int amount_greedy_actions;
    double random_number;
    double temporal_error;
    int my_number;
    char filename[100];
    char filename_saved_weights[100];
    char weights_dir[120];
    char buffer[160];
    FILE* weights_file;

    void loadWeights ( char* weights_dir, double weights[][ACTION_POSSIBILITIES],
                      int keeper_number );
    void saveWeights ( char* filedirectory, int my_number, int episode_count ,
                       double weights[][ACTION_POSSIBILITIES] );
    void update ( int last_features_present[][TILINGS_PER_VAR],
                   51

51
double temporal_error, int last_action,
    double weights[][ACTION_POSSIBILITIES] );

int selectAction ( double current_action_values[], double epsilon );
void getFeaturesPresent ( double state[], int last_features_present[][TILINGS_PER_VAR] );
void getCurrentActionValues ( int last_features_present[][TILINGS_PER_VAR],
                          double current_action_values[],
                          double weights[][ACTION_POSSIBILITIES] );
double getEpsilon ( int episode_count );

public:

LearningAgent ( int numFeatures, int numActions, int keeper_number );

int startEpisode( double state[] );
int step( double reward, double state[] );
void endEpisode( double reward );
void setParams(int iCutoffEpisodes, int iStopLearningEpisodes);

};
#endif