Reinforcement Learning in Keepaway Framework for RoboCup Simulation League

Author: Wei Li
wli09001@student.mdh.se
Schoole of Innovation, Design and Engineering(IDT)
Mälardalen University
Västerås, Sweden

Supervisor: Lars Asplund
lars.asplund@mdh.se

August 2011
Abstract

This thesis aims to apply the reinforcement learning into soccer robot and show the great power of reinforcement learning for the RoboCup. In the first part, the background of reinforcement learning is briefly introduced before showing the previous work on it. Therefore the difficulty in implementing reinforcement learning is proposed. The second section demonstrates basic concepts in reinforcement learning, including three fundamental elements, state, action and reward respectively, and three classical approaches, dynamic programming, monte carlo methods and temporal-difference learning respectively. When it comes to keepaway framework, more explanations are given to further combine keepaway with reinforcement learning. After the suggestion about sarsa algorithm with two function approximation, artificial neural network and tile coding, it is implemented successfully during the simulations. The results show it significantly improves the performance of soccer robot.

Keywords: Reinforcement learning, Sarsa, Keepaway framework, Tile coding
Acknowledgements

I would like to thank my supervisor Lars Asplund who helped me a lot during the thesis. I am very thankful for all interesting discussions in our meetings. He provided me with several interesting papers and always had amazing ideas about solving my difficulty during the thesis. I am really grateful for all his comments and feedbacks on my work.

In addition, I would like to express my gratitude to Damir Isovic from Mälardalen University and Wang Huifeng from East China University of Science and Technology(ECUST) who both provided me with such precise opportunity to live in Sweden for two years during my master study.

Finally, I am grateful to my girlfriend Qiu Honglei for her love and my classmates from Mälardalen University as well as from ECUST. Without all your help, I cannot finish my thesis. Thanks for all your support.
## Contents

1. Introduction ................................................................................................................. 7  
   1.1 Background ........................................................................................................ 7  
   1.2 Previous work ..................................................................................................... 8  
   1.3 Difficulty in reinforcement learning ................................................................. 9  
2. Reinforcement Learning .............................................................................................. 10  
   2.1 Three Fundamental elements ............................................................................. 10  
      2.1.1 State ........................................................................................................... 10  
      2.1.2 Action ....................................................................................................... 11  
      2.1.3 Reward ..................................................................................................... 11  
   2.2 Three Classical Approaches .............................................................................. 12  
      2.2.1 Dynamic programming ............................................................................. 12  
      2.2.2 Monte Carlo methods ............................................................................... 13  
      2.2.3 Temporal-Difference learning ................................................................. 13  
3. Keepaway Framework ................................................................................................. 14  
   3.1 State .................................................................................................................. 14  
   3.2 Action ............................................................................................................... 16  
   3.3 Reward ............................................................................................................. 17  
4. Applied learning algorithm in Keepaway Framework ............................................. 18  
   4.1 Sarsa algorithm .................................................................................................. 18  
   4.2 Function approximation ...................................................................................... 19  
      4.2.1 Artificial neural network ........................................................................... 19  
      4.2.2 Tile coding ................................................................................................. 20  
   4.3 Experimental Result .......................................................................................... 22  
5. Conclusions and future work .................................................................................... 26  
Reference ...................................................................................................................... 28  
Appendix ....................................................................................................................... 30
List of Figures

2.1 The work flow of reinforcement learning.........................................................9
3.1 The state variables used for learning with three keepers and two takers.........14
3.2 The policy space for keepers.................................................................15
4.1 The architecture of artificial neural network...........................................18
4.2 The learning result with exploring rate $\epsilon = 0.01$.................................23
4.3 The learning result with exploring rate $\epsilon = 0.02$.................................24
4.4 The learning result with exploring rate $\epsilon = 0.1$.................................24
4.5 The learning result with exploring rate decreasing from $\epsilon = 0.1$ to $\epsilon = 0.01$...25
List of Tables

4.1 The general sarsa algorithm ................................................................. 18
4.2 The sarsa algorithm combined with tile coding .................................... 20
Chapter 1

Introduction

During the last several decades, there is an interesting phenomenon that animals get their optimal behavior after interacting with the environment for evolving thousands of years. This attracts many scientists' attention and the concept of reinforcement learning is introduced from that[1].

1.1 Background

Reinforcement Learning is typically one of the famous machine learning approaches designed to improve the behavior of robot agent. The general idea of reinforcement learning is by interacting with the environment, as soon as the soccer agent execute one action, it will get a reward for a good action, otherwise a punishment for a bad behavior. Through tens of thousands of tries, probably even more tries for more complex system, the robot will know which action is better and which action is worse in each state by itself. This is very useful for autonomous soccer robot since the robot can learn without any help except trying different actions in limited states within a couple of hours. Therefore reinforcement learning has achieved a great success on soccer robot field [2] as well as many other domains[3][4] [5].

On the other hand, RoboCup simulated soccer has been getting more and more popular during the past several years since it contains a great challenge about the control and cooperation among multi agents[6][7]. It exactly simulates the real environment of soccer robot competition such as the soccer robot only have partial observation of the environment instead of the full view of environment and the inaccurate information resulted from noisy sensors and actuators[2].

This thesis tries to combine one on-line reinforcement learning algorithm Sarsa with two function approximation techniques separately[] artificial neural network and tile coding. Therefore it is applied into keepaway framework, which is a part of soccer robot in RoboCup simulation league. The motivation of this thesis project is to test the performance of Sarsa algorithm with tile coding as a way of studying reinforcement learning and decide whether it is suitable for implementing this similar algorithm in
soccer robot from RoboCup Middle Sized League. Furthermore, it also tries to examine the influence of different factors within the algorithm Sarsa, especially the exploring factor.

1.2 Previous work

During the period of this thesis, most previous work information comes from the RoboSoccer KeepAway website itself, where several papers about Reinforcement learning are available. Thanks to this great website, lots of details about the use of reinforcement learning are explained in detail.

In paper [2] Sarsa method is employed as a reinforcement learning method. Sarsa has a lot points in common with Q learning, which is also another popular accepted reinforcement learning method. They both have the characteristic of on-line learning, which means the reward is immediate without any delay. However, the slight difference is Q learning updates the value function based on current action value whenever the state is[8], while Sarsa updates the value function based on next action value and next state.

For the function approximation, usually there is two approaches, artificial neural network and tile coding. The good point from artificial neural network is the construction is very transparent. It is quite easy to see and learn. However, the defect is a simple network never leads to a good result while a complex network takes too long time for training. When it comes to tile coding although it is not easy to see the result is quite good from [2][13]. Tile coding focuses on the active tiling for each tile and only updates the valid tiling from active ones. So the update rule in tile coding is much better than artificial neural network, since the weight update rule in the latter one is quite time consuming. Besides, the state in tile coding is discrete and there is no limit on the state space. So it is safe to implement tile coding for a complex artificial system.

According to the previous work, sarsa with tile coding approach has already had some success. However there should still exist some improvements in it.
1.3 Difficulty in Reinforcement Learning

To get a better learning result, in general two aspects which is hard to implement in reinforcement learning, are as follows:

- The selection of reinforcement learning algorithm
- The selection of function approximation

Since different algorithm has different benefits and drawbacks, it is very difficult to pick the right algorithm out during the preparation from the start. Considering that keepaway is running in a continuous environment, Sarsa which is already known as a good solution to handle continuous state as discrete state by means of Semi-Markov Decision Procedure(SMDP) is selected in this thesis.

As is mentioned above, due to the fact that in reality most state is continuous while the state is all discrete in the study of reinforcement learning, it is very important to deal with the mapping from continuous state and action values to discrete state and action values. Therefore this thesis applies tile coding to represent the state and action values as a value function.

Furthermore in detail, several factors in the algorithm have significantly influence on the testing result. It is worth to carefully figure out which factor has the biggest influence on the result as well.
Chapter 2

Reinforcement learning

From the definition of Markov Decision Process (MDP)[10], many reinforcement learning problems can be fully represented by Markov Decision Process, which mainly consists of:

1. A finite set of states $S$, which contains all possible states.
2. A finite set of actions $A$, which contains all possible actions.
3. A reward function $r(s)$, which works by calculating the reward that the agent gets updated when the agent is in a given state.
4. A state transition function, which works as a transition from one state to another state by executing an action[9].

The following figure shows the work flow of reinforcement learning.

![Fig 2.1 The work flow of reinforcement learning](image)

2.1 Three Fundamental elements

2.1.1 State

Basically the environment in reinforcement learning is represented by several states. In the robotic area, coordinates of robot agent’s position and angles of the agent heading are both mostly used as representatives of the state of robot. Based on the states, the robot agent decides which action it should take to get a maximum reward.
As the system gets more and more complex, to achieve a better performance, the state variable tends to have more and more information to fully describe what the exactly environment is. This could be also very helpful for determining the best next state according to the current state and the best action.

The relationship among state, action and reward is described as follows:

$$P_r \{ S_{t+1} = S', R_{t+1} = R | S_t, A_t \}$$

For every state, there is an attached value to describe the estimate of the reward right after this given state. The value function is described as follows:

$$V^\pi(s) = E_x(R_t | S_t = S) = E_x \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = S \right\}$$

### 2.1.2 Action

In each state, there are several actions to choose. Similar to the state value, there is an also attached value for each action in every scenario. In most cases, the concept of action here is quite close to policy. Actually the key point in reinforcement learning is the soccer robot has the capability of always finding the optimal policy in every state. So the question switches to how to find an optimal policy in some specific state.

Taking Q learning as an example, Q learning updates action value (Q value) with the following rule:

$$Q(s, a) = r(s, a) + \gamma \max_a (Q(s', a'))$$

The action that is selected among several choices is given by

$$\pi(s) = \arg \max_a Q(s, a)$$

### 2.1.3 Reward

Similar to the state value, there is also an attached value for each pair of state and action. The reward function is described as follows:
\[ R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (2-5) \]

If the loop of experimentation is endless and the reward is infinite, the algorithm will never converge. In case of this situation, discounting factor \( \gamma \) is very helpful for encourage the algorithm to converge as fast as possible. In most cases, the discounting factor \( \gamma \) is set between 0 and 1.

### 2.2 Three Classical Approaches

#### 2.2.1 Dynamic programming

In 1957 Bellman [11] published an very impressive paper from which the term dynamic programming (DP) is firstly introduced. The most important thing is it introduces a serious of algorithms by which an optimal policy can be calculated in the manner of MDP. However, dynamic programming algorithms still need a perfect model of the environment whenever they are working as they are supposed to be [9]:

\[
V^\pi(s) = E_\pi \left\{ R_t \mid s_t = s \right\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} = E_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s \right\} \\
= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s \right\}] = \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \\
(2-6)
\]

From (2-6), \( P_{ss'}^a \) is the probability of transition from state \( s \) to state \( s' \) when action \( a \) is taking in \( s \), which evaluates how much influences that \( R_{ss'}^a + \gamma V^\pi(s') \) has on the value of state \( s \).

From Bellman[11], the biggest problem with dynamic programming is the calculation will get much more work to do as the explosively increase of state variables. In other words, the more state variable there are, the more calculation needs to be done. Obviously this is the biggest defect for dynamic programming when applying it to big systems.
2.2.2 Monte Carlo methods

As is discussed above, the dynamic programming approach describe the whole
environment straightforward with reward function and transitions function. However,
in reality it is impossible to gather all the necessary information. Monte Carlo (MC)
methods is introduced to solve this problem. The biggest improvement in Monte
Carlo method is that agent interacts with the environment and then take notes of the
samples. These standard samples contain all the necessary information about all the
states, actions and rewards. Most importantly, it is through the interacting with the
environment that the information is obtained instead of having all the necessary
information in advance[9].

As is stated above, the way how the value function is getting updated is given by

\[ V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)] \] (2-7)

Obviously \( V(s_t) \) is a sample from the trying process.

2.2.3 Temporal-Difference learning

Temporal-Difference learning (TD learning) has become also a popular reinforcement
learning technique in recent years after TD methods have had a great number of
success from many experiments by applying function approximations to generalize
learning to large systems with large numbers of state variables space.

What’s more important, temporal difference learning updates the value function based
on the difference between the real value function and the value function estimate. In
most cases, the way the update works is given by

\[ V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \] (2-7)

One of the key issues for the temporal difference learning algorithms is how to easily
achieve an balance between exploration and exploitation, which means the current
optimal policy has the possibility of achieving a worse performance in the long run.
So sometimes temporary policy except optimal one is necessary to be executed with a
small probability[9].
Chapter 3

Keepaway Framework

Keepaway is a classical subproblem of the robot soccer domain especially for the simulation league. The simplifications that are modified from simulation competition to keepaway are as follows:

- Fewer players are running in the environment, typically 3 keepers against 2 takers. This is to decrease possible policy space;
- The football training field is a bit smaller, typically only 20cm * 20cm. This is to simplify possible state space and make the function approximation easier;
- The players only need to concentrate on high-level strategy goal. The soccer player will not take the competition situation into consideration and instead only execute the best action in a given state.

In article [12], they only put an emphasis on the learning result of the keepers when 3 keepers playing against 2 takers with random policies. However, the long-term purpose of keepaway studies is to apply keepaway skills into standard RoboCup simulation league as well as practical soccer robot, such as Middle Sized League.

3.1 State

One key feature in reinforcement learning is it does not need to know the exact model of environment but only keep trying out through interacting with the environment. And the environment is represented by several state variables. In keepaway, keepers used state variables to generate a value function which is helpful to determine the best policy under some circumstance.

In [2][9][13], the state variables for the keepers and takers is computed depending on the positions of them.

- The keepers $K_1 - K_n$, where $n$ is the number of the keepers. The number is ordered by the increasing distance between the keepers;
- The takers $T_1 - T_m$, where $m$ is the number of takers. The number is ordered as well as the takers;
\text{C} \text{ means the center of the playing region.} \\
Assume two definition as follows:

\text{dist}(a,b) \text{ means the distance between soccer agent } a \text{ and soccer agent } b \\
\text{ang}(a,b,c) \text{ means the angle between } a \text{ and } c \text{ with vertex at b[2][13].}

As Figure 3.1 below shows, the following 13 state variables are used to fully represent the simulation environment when 3 keepers are playing against 2 takers.

\text{dist}(K_1,C) ; \text{dist}(K_2,C) ; \text{dist}(K_3,C); \\
\text{dist}(T_1,C) ; \text{dist}(T_2,C); \\
\text{dist}(K_1,K_2) ; \text{dist}(K_1,K_3); \\
\text{dist}(K_1,T_1) ; \text{dist}(K_1,T_2); \\
\text{Min}(\text{dist}(K_2,T_1),\text{dist}(K_2,T_2)); \\
\text{Min}(\text{dist}(K_3,T_1),\text{dist}(K_3,T_2)); \\
\text{Min}(\text{ang}(K_2,K_1,T_1),\text{ang}(K_2,K_1,T_2)) \\
\text{Min}(\text{ang}(K_3,K_1,T_1),\text{ang}(K_3,K_1,T_2))

Figure 3.1 The state variables used for learning with three keepers and two takers.
3.2 Action

As is shown in last section, state space is shown specially for 3 keepers against 2 takers. Correspondingly, this section will give out policy space, sometimes also meaning available actions, specially for 3 keepers against 2 takers as well.

First of all, since the learning process only takes place when the player has the ball, the player has the highest priority to receive the ball from other teammates or fetch the ball by itself whenever the ball is not in possession.

As soon as the keeper has the ball in possession, on the other hand, it has multiple choices to take. The keeper could only hold the ball to prevent from taking away by opponents. Otherwise, it could choose to pass the ball to other teammates in case of emergence. Taking the competition between 3 keepers and 2 takers, the general policy space for keepers is \( \{ \text{HoldBall, PassK_2 Then Receive, PassK_3 Then Receive} \} \).

![Figure 3.2 The policy space for keepers](image)

3.3 Reward

One different thing in the keepaway framework is that the reward is set according to the time during which keepers has the ball. In other words, the longer the keeper possess the ball, the bigger the reward they will get. The direct learning task is to
increase the reward throughout the whole learning process.

More in detail, the reward in the keepaway framework is calculated by the rule:

\[ reward = \text{CurrentActionTime} - \text{LastActionTime} \quad (3-1) \]
Chapter 4
Applied learning algorithm in Keepaway Framework

4.1 Sarsa algorithm

As is known to all that Sarsa is an on-line learning method, the pair of action and state value is estimated completely depending on the currently being taken action. At the same time, the policy is continually updated relying on the estimates of action value. In one word, the value action is getting updated again and again. The learning procedure in Sarsa is as follows:

<table>
<thead>
<tr>
<th>Initialize $Q(S,a)$ arbitrarily</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat (for each episode):</td>
</tr>
</tbody>
</table>
| Initialize $e(s,a) = 0$ for all $s,a$
| Initialize $s$
| Choose $a$ from $s$ using policy derived from $Q$
| Repeat (for each step of episode): |
| Take action $a$, observe rewards $r,s'$
| Choose $a'$ from $s'$ using policy derived from $Q$
| $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
| $e(s,a) \leftarrow e(s,a) + 1$
| For all $s,a$
| $Q(s,a) \leftarrow Q(s,a) + \alpha \delta e(s,a)$
| $e(s,a) \leftarrow \gamma e(s,a)$
| $s \leftarrow s'$, $a \leftarrow a'$
| Until $s$ is terminal |

Table 4.1 The general sarsa algorithm

From the table 4.1, $\alpha$ is a learning rate parameter which is always set between 0 and 1; $\gamma$ is a discount factor which is responsible for updating the weight and set between
0 and 1 as well; the eligibility traces $\epsilon(s, a)$ keep track of last action and represent the influence from current reward to last action; $\lambda$ means how much eligibility traces needs to be delivered back and kept for next episode[2][13] and it is also usually set between 0 and 1.

4.2 Function approximation

The main idea of sarsa algorithm is to get action and state values as much and accurate as possible. Besides, each action is supposed to have unlimited trials in each state. The most important question is how to get large number of states generalized to adapt for large state space. Until now basically there are two methods for function approximation, artificial neural network and tile coding respectively.

4.2.1 Artificial neural network

ANN is a machine learning algorithm which is vastly used in artificial intelligence systems. The architecture of artificial neural network is shown in figure 4.1. It contains three layers, input layer, hidden layer and output layer. After the calculation, the output is obtained based on the input data.

![Fig 4.1 The architecture of artificial neural network](image)
The typical sequential version of backpropagation algorithm which is popular and used in artificial neural network works is as follow:

1. Initialize weights with a small random value
2. Present Inputs to the network and compute the outputs
3. Compute the error terms in the output layer
4. Compute the error terms in the hidden layer
5. Update the weights for all units according to
6. Check if overall error $E$ becomes acceptably low.

In this thesis project, there are 13 state variables totally as inputs and 3 available actions as output. The output action value is treated as the main standard for selecting the actions. In other words, the action with biggest output value is selected as the optimal policy all the time.

### 4.2.2 Tile coding

Tile coding method generally can arbitrarily take dozens of continuous state variables at one time[2]. For tile coding, each tile only contains the current state in each tiling and several tiles constitute a feature set $F_a$, with each action $a$ describing the different tilings[13]. In keepaway domain, the number of tilings is limited since state variables are limited while the number of tiles is unlimited.

Taking Sarsa with tile coding as an example, the following three points need to be clearly known.

1. A feature vector $F_a$ is used to indicate which features are present in use.

2. The current state value is calculated completely relying on the features $F_a$.

3. Eligibility traces $\varepsilon(s,a)$ is attached to features since feature set $F_a$ represents all the states and environment [9].

The learning procedure in Sarsa combined with tile coding is as follows:

```
RL startEpisode:
For all $a \in A^s$
```
\[ F_a \leftarrow \text{set of tiles for } a, s \]
\[ Q_a \leftarrow \sum_{i \in F_a} \theta(i) \]
\[ \text{LastAction} \leftarrow \begin{cases} \arg \max_a Q_a & \text{w/ prob } 1 - \varepsilon, \\ \text{randomaction} & \text{w/ prob } \varepsilon \end{cases} \]
\[ \text{LastActionTime} \leftarrow \text{CurrentTime} \]
\[ e(i) \leftarrow 0 \]
For all \( i \in F_{\text{LastAction}} \)
\[ e(i) \leftarrow 1 \]

RL step:
\[ r \leftarrow \text{CurrentTime} - \text{LastActionTime} \]
\[ \delta \leftarrow r - Q_{\text{LastAction}} \]
For all \( a \in A^s \)
\[ F_a \leftarrow \text{set of tiles for } a, s \]
\[ Q_a \leftarrow \sum_{i \in F_a} \theta(i) \]
\[ \text{LastAction} \leftarrow \begin{cases} \arg \max_a Q_a & \text{w/ prob } 1 - \varepsilon, \\ \text{randomaction} & \text{w/ prob } \varepsilon \end{cases} \]
\[ \text{LastActionTime} \leftarrow \text{CurrentTime} \]
\[ \delta \leftarrow \delta + Q_{\text{LastAction}} \]
\[ \theta \leftarrow \theta + \alpha \delta e \]
\[ Q_{\text{LastAction}} \leftarrow \sum_{i \in F_{\text{LastAction}}} \theta(i) \]
\[ e \leftarrow \lambda e \]
If Player acting in state \( s \):
For all \( a \in A^s \), \( a \neq \text{LastAction} \):
For all \( i \in F_a \):
\[ e(i) \leftarrow 0 \]
As the table above, the whole learning process is divided into three sub-procedures, RLstartEpisode, RLstep and RLendEpisode. In RLstartEpisode, it starts to find all the available actions by activating the current state in each tile. The weights of the tiles in $F_a$ is summed up as the action value. The action with the biggest action value will be selected as the current action. At the final of this stage, the eligibility traces are initialized. The soccer agent will go to the RLstep as long as it still has the ball. The difference is it calculates the error between the value action estimate and last action value as well as reward. Then the error and reward are used to update action values to get a new current value again. The Last part is RLendEpisode which has quite similar theory meaning as RLstartEpisode and RLstep.

4.3 Experimental Result

In this thesis, we implement sarsa learning algorithm with tile coding to get soccer agent trained in a couple of hours. The reason why sarsa is selected here is as follows. First of all there is not enough information about the exact environment. While sarsa does not rely on the model of exact environment, sarsa will be very beneficial here. Therefore if soccer agent can learn on-line through sarsa, there is no delay reward. In other words, the value function can be updated immediately without any delay.

On the other hand, the algorithm with on-line learning is capable of converge fast. From the perspective of function approximation, artificial neural network is used at first. However, the result shows that the learning process is really very slow and in some special cases it does not converge at the end of learning. The slow learning process is understandable because a huge number of weights are always updated during the learning if a very large and complex network is built. This means the learning result has a very close relationship with the construction of neural network.

\begin{table}
\centering
\begin{tabular}{c}
\hline
For all $i \in F_{\text{LastAction}}$
\begin{align*}
e(i) &\leftarrow 1 \\
\text{RLendEpisode:} &
\begin{array}{l}
\ r \leftarrow \text{CurrentTime} - \text{LastActionTime} \\
\delta \leftarrow r - Q_{\text{LastAction}}
\end{array}
\end{align*}
\end{tabular}
\caption{The sarsa algorithm combined with tile coding}
\end{table}
Since neural network does not work properly at the beginning, tile coding approach is implemented in order to get a better simulation result.

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. The tiling parameters are set as one tiling per state variable and 32 tilings totally per state variable.

Fig 4.2 The learning result with exploring rate $\epsilon = 0.01$
Fig 4.3 The learning result with exploring rate $\epsilon = 0.02$

Fig 4.4 The learning result with exploring rate $\epsilon = 0.1$
Fig 4.5 The learning result with exploring rate decreasing from $\epsilon = 0.1$ to $\epsilon = 0.01$

From the figures 4.2-4.4 above, we can see that when $\epsilon$ is smaller, 0.01 and 0.02 respectively, the algorithm takes approximately 40 training hour to converge and the peak point is less than 6 seconds. However, when the $\epsilon$ is set to 0.1, it converges much faster from the start and it can arrive at higher than 6 seconds. The only drawback is it is not as stable as the scenarios of 0.01 and 0.02.

Since exploring rate $\epsilon$ plays an important role in the learning result, an $\epsilon$-related function was designed to improve the simulation result. In the manner of $\epsilon$-related function, exploring rate stays stable at $\epsilon = 0.1$ until the 10000th episodes were executed. To make the learning curve more stable, exploring rate afterwards will gradually decrease until it is equal to 0.01. The simulation result shows it can achieve 7 seconds without decreasing the converge speed.

As a conclusion, adjusting the value of exploring factor along the experimentation can effectively improve the learning result for sarsa algorithm with tile coding.
Chapter 5
Conclusions and future work

Reinforcement learning is a very good solution for a wide range of reality system problems, as long as the system satisfy the Markov property and has three basic characteristics such as states, actions and rewards. During the past two decades, it is already applied into chess playing, mazing problem and so on.

While most states theoretically is discrete in MDP, most reality system all has the continuous states. Function approximation is introduced to solve this problem and extend the use of reinforcement learning as well.

Keepaway soccer robot is a very good example for both reinforcement learning and function approximation. Learning algorithm sarsa is implemented in this thesis and the simulation result shows it works perfectly.

Due to the limited time for my thesis, there is still a lot of interesting issues left to be taken into consideration in the next step:

- Since the construction of artificial neural network has a significant influence on the learning result, further work need to study the relationship between the construction of neural network and sarsa learning algorithm. Probably the number of state variables and available actions (different policies) need to be take into consideration as well, because the input of network is the state variables and the output is the action (policy).
- The simulation shows the value of exploring rate is very important. A higher exploring rate could cause the fast converge and get a better performance. However, there should be a critical point that maybe either slow the converge speed down or weaken the performance. So it is important to find the critical point for exploring rate.
- Now all the simulations are running within a environment of 3 keepers and 2 takers. The field is smaller as well, only 20 cm * 20 cm. Next step is to enlarge the field as big as possible and apply more keepers and takers in the field. Since more state is introduced, it could be interesting to see what the simulation result is.
Although the learning algorithm works now, it still takes too much time to achieve a good result. There could be some improvements in the learning algorithm itself. Therefore, currently the soccer agent is trained based on too much data. It is possible to see what it will behave with less training data. In other words, finding an algorithm or continue improving sarsa is still necessary to make the algorithm converge faster.

Until now the soccer agent is running in a world without any noise. However, the real soccer robot will meet multiple of noise which could possibly reduce the intelligent performance. Since it is possible to add some noise in the keepaway framework, it is interesting to see what the result is.

Until now the soccer agent has a whole view of the field, including both view front and back. Apparently it is not possible in reality. Since it is possible to change this condition in the keepaway framework, it is interesting to see what the result is.

The simulation aims to design an algorithm which can be completely applied to reality. However, when it is applied in the real soccer robot, even the algorithm with the best performance in the simulation, the real soccer robot will probably not behave as imagined. This is always true after so many theroical experiment and practical test. The most important issue is that there is some uncertain relationship between them. The future work finally is to see what the real robot behave with the learning algorithm and it is definitely necessary to find the relationship out.
Reference


Appendix

Source for installation instructions

http://www.cs.utexas.edu/~AustinVilla/sim/Keepaway/tutorial.html
https://utlists.utexas.edu/sympa/arc/keepaway/2010-07/msg00004.html

General information:
Version 11.1.0 of the 'rcssbase', 'rcssserver' and 'rcssmonitor' are used, newer versions fail to compile.
Version 0.6 of the 'keepaway_player' is used.
The framework was installed successfully on a clean Ubuntu 8.04 LTS installation in VMWare Workstation.
Installation on newer versions of Ubuntu and other distributions have failed.

Install prerequisite packages:
sudo apt-get install g++ libboost-dev libboost-filesystem-dev bison flex libx11-dev
libxpm-dev csh gnuplot-nox gv
The 'csh', 'gnuplot-nox' and 'gv' packages are only required for standalone tools such as drawing graphs.

Choose the installation directory:
First, you must choose where you want to install the soccer server files. For system-wide installation, you will probably want to choose a directory such as /usr/local. If you do not have root access, you will probably want to choose something like a subdirectory of your home directory. For instance, /home/username/rcss. You need to set the RCSSBASEDIR environment variable to this directory:

export RCSSBASEDIR=/home/username/rcss

Now you need, to add the bin sub-directory to your PATH. For system-wide installation, you will probably not need to do this because the sub-directory (e.g. /usr/local/bin) is already in your PATH. For local installation:

export PATH=$RCSSBASEDIR/bin:$PATH
For this change to take effect permanently, you will need to add this line to a file that is sourced when you open a shell, such as ~/.profile.

Install the base files:

```
tar xjvf rcssbase-11.1.0.tar.bz2
pushd rcssbase-11.1.0
./configure --prefix=$RCSSBASEDIR --with-boost-filesystem=boost_filesystem-mt
make
make install
popd
```

Install the server:

```
tar xjvf rcssserver-11.1.0.tar.bz2
pushd rcssserver-11.1.0
```

The following three steps are necessary for local installation only: (Note that it is unnecessary to set these variables permanently as they are used only during installation.)

```
export LDFLAGS="-L$RCSSBASEDIR/lib"
export CXXFLAGS="-I$RCSSBASEDIR/include"
export LLIMPORTER_PATH="$RCSSBASEDIR/bin"
```

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

Install the monitor:

```
tar xjvf rcssmonitor-11.1.0.tar.bz2
pushd rcssmonitor-11.1.0
./configure --prefix=$RCSSBASEDIR --with-boost-filesystem=boost_filesystem-mt
```
make
make install
popd

2) Downloading and Compiling the Players
First, download the keepaway framework package:

* Download Version 0.6

Next, open the package in the directory in which you would like it installed:

tar xzvf keepaway-0.6.tar.gz

Now compile the players:

cd keepaway-0.6/player
Open the player Makefile, change 'makedepend' to 'gccmakedep' under the 'depend'
build rule.
make depend
make

Next, compile and install the tools:

cd ./tools
make

Copy the binaries to a directory in your PATH. For example:

cp hist killserver kunzip kwyzipper kzip monitor winsum /usr/local/bin

3) Running the players
To start the simulation:

(Open 'keepaway.sh' and change the sha-bang from '#!/bin/sh' to '#!/bin/bash')
The startup script will finish after about 10 seconds. The simulation is now running. To watch the players, execute the monitor script:

monitor

To end the simulation:

killserver