MULTI-AGENT REINFORCEMENT LEARNING USING ROLES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERKİN ÇİLDEN

116240

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

116240

SEPTEMBER 2001

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Faruk Polat
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Göktürk Üçoluk (Chairman)

Assoc. Prof. Dr. Faruk Polat

Asst. Prof. Dr. Halit Oğuztüzün

Asst. Prof. Dr. Ahmet Coşar

Asst. Prof. Dr. Bilge Say

ABSTRACT

MULTI-AGENT REINFORCEMENT LEARNING USING ROLES

Çilden, Erkin

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Faruk Polat

September 2001, 71 pages

Reinforcement learning (RL) is known to be an promising machine learning technique and is extensively used in agent theory. Multi-agent reinforcement learning (MARL) deals with how to scale up RL to multi-agent domains. Coordination among agents can be achieved without explicit information sharing. Decomposition of task may be necessary to realize learning in some problems. In this thesis, roles are defined for an agent as an alternative approach to task decomposition for problems made up of sequential sub-tasks. The approach is experimented and discussed on two domains of different characteristics.

Keywords: Reinforcement learning, multi-agent learning, multi-agent coordination, agent roles

ÖZ

ROLLER KULLANARAK ÇOKLU-ETMEN TAKVİYE ÖĞRENME

Çilden, Erkin

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Faruk Polat

Eylül 2001, 71 sayfa

Takviye öğrenme, önemi artan bir makine öğrenme tekniği olarak bilinmektedir ve etmen teorisinde yaygın olarak kullanılmaktadır. Çoklu-etmen takviye öğrenme ise takviye öğrenme tekniklerinin çoklu-etmen ortamlara genişletilerek uyarlanması ile ilgilenir. Etmenler arası koordinasyon açıkça bilgi paylaşımı olmadan edinilebilir. Kimi problemlerde öğrenmenin gerçekleşmesi için görev bölümlendirmesi gerekebilir. Bu tezde görev bölümlendirmesine alternatif bir yaklaşım olarak, alt görevlerinden oluşan problemlerin çözümü için etmen rolleri tanımlanmıştır. Yaklaşım farklı özellikteki iki ortamda denenmiş ve tartışılmıştır.

Anahtar Kelimeler: Takviye öğrenme, çoklu-etmen öğrenme, çoklu-etmen koordinasyon, etmen rolleri

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor Dr. Faruk Polat. He had been so patient and encouraging throughout the development of this thesis.

TABLE OF CONTENTS

ABSTRACT iii

ÖZ iv

ACKNOWLEDGMENTS v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

CHAPTER

I INTRODUCTION 1

II MULTI-AGENT REINFORCEMENT LEARNING 5

II.1 Multi-Agent Systems 6

II.2 Reinforcement Learning 8

II.2.1 Markov Decision Processes 9

II.2.2 Dynamic Programming 11

II.2.3 Monte Carlo Methods 13

II.2.4 Temporal Difference Learning 15

II.2.4.1 TD(0) Learning 15

II.2.4.2 Q-Learning 16

II.2.4.3 Using Eligibility Traces 17

II.3 Multi-Agent Reinforcement Learning 20

III SIMULATED SOCCER DOMAIN 23

III.1 Overview 23

III.2 RoboCup Soccer Server 25

III.2.1 General Description 25

III.2.2 Simulation of Soccer Field 26

III.2.3 Player Simulation 28

III.2.3.1 Commands to Server 28
III.2.3.2 Sensor Information 30
III.2.4 The Coach Client 36
IV LEARNING ROLE DEPENDENT BEHAVIORS 37
IV.1 Roles and Coordination 38
IV.2 Adversarial Carry-track Domain 41
IV.3 2 vs. 1 Man Passing in Simulated Soccer Domain 47
IV.3.1 Role Based Design 48
IV.3.2 Task Based Design 55
V CONCLUSION 65
REFERENCES 68

LIST OF FIGURES

II.1 Agent-environment interaction in RL. 9
II.2 First-visit MC method for estimating V^π. 14
II.3 TD(0) algorithm for V^π estimation. 16
II.4 One step Q-learning. 17
II.5 TD(λ) algorithm. 19
II.6 Watkin's Q(λ) algorithm. 19

III.1 A snapshot from soccer server 28
III.2 Flags and lines in soccer server 32
III.3 Visible range of an agent in the soccer server. 35

IV.1 A sample initial setting in carry-track world. 41
IV.2 A possible losing state (left) and one of the winning states (right). 42
IV.3 Distances sensed by agent located at center. x is the same location with agent while n indicates *near*. Any other location is sensed as *far*. 44
IV.4 Directions sensed by agent located at the grid. An object is sensed to be in one of those four directions. 45
IV.5 Enemy avoidance of an agent by a *leave* action. 47
IV.6 A typical coordination strategy learned. 48
IV.7 A sample passing instance. 49
IV.8 Partitioning of agent's view cone by distance (small numbers) and direction (big numbers). 50
IV.9 Initial positions of ball and goal-keeper. 52
IV.10 Test field. 53
IV.11 Discretization of the calculated distance of object to ball. 57
IV.12 Distances to ball as observed by an agent for task 1. 57
IV.13 Tabular view of state definition for task 1. 58
IV.14 An initial placement for task 1. 59
IV.15 Examples of positions causing a positive (left) and a negative (right) reward. 59
IV.16 Dominant strategies when initial positions are apparently different. 60
IV.17 Dominant strategies when initial positions of both agents are near the ball. 60
IV.18 Dominant strategies when initial positions of both agents are far away from the ball. 61
IV.19 Distances to ball as observed by an agent for task 2. 62

IV.20 Tabular view of state definition based on distances for task 2. . . . 62
IV.21 A possible initial position setting for task 2. 63

CHAPTER I

INTRODUCTION

For the last few decades, problem solving using autonomous software entities, called *agents*, has become very attractive for artificial intelligence (AI) researchers. In AI point of view, the idea behind agent technology is the existence (execution) of a computer program, possibly running a robot, within a model of the problem, trying to optimize some environmental or internal parameters, for the solution of all or a part of the problem. This can either be done by an external supervision, or by only calculations built on an initial knowledge. In either case, interaction with the problem environment is a must and adaptation is a consequence.

Since the emergence of the idea, adaptation of an agent to its environment has become one of the main problems to be solved. Many known techniques for improving automatically with experience, called *machine learning*, has been applied and enhanced [Mitchell 97], and new techniques are proposed for the agent technology specifically after then. A philosophical aspect of learning machines had also been discussed since the early times of computing machinery [Turing 50].

Like adaptive agency, social agency might be a very effective method leading us to the solution. Imitating the idea of collective behavior for a certain task, as many living creatures always perform throughout their lives, gives rise to a different perspective in agent theory, *multi-agent systems.* When this behavior is learned using machine learning techniques, we say that the agents perform *multi-agent learning.* In this paradigm, the key idea is that by individual learning efforts, a collective learning on a common aim can be realized.

Most of the time, agent learning requires an on-line adaptive learning agent architecture. In many real applications, like robotics, distributed computing, Internet software, this property comes up to be a necessity. *Reinforcement learning* techniques are model-free learning methods suitable for this purpose. Reinforcement learning methods are effective in solving *temporal credit assignment* problem, which can be summarized to be the late consequences of actions performed. Moreover, these methods can easily be scaled up to multi-agent learning [Tan 93].

Many problems can be modeled in such a way that when they are split into sub-problems, solution of the problem by learning is simplified. In agent learning, these kind of problems have been attractive for the last few years. A trivial approach is to assign each of these sub-problems to agents in a predefined manner. This approach is called *task decomposition*, and is excessively used in multi-agent systems also.

In this thesis, we propose that, *role decomposition*, decomposing state and action sets of agents, can also lead to effective multi-agent learning in some multi-task domains. In many cases, this decomposition is inevitable by definition of the problem, or by sensing and acting capabilities of agent, in others, it is a

design issue. In either case, different than separately learning sub-tasks of a problem and then combining them to be used as a solution to overall task, with role decomposition, agent might learn the task as a whole, and through a single learning effort.

We experimented two multi-agent learning tasks using roles: Adversarial carry-track domain as a grid-world problem and 2 vs. 1 man passing problem in simulated soccer domain. Both problems are suitable to be solved using task decomposition and learning of sub-tasks independently. We attacked these two problems using role decomposition as an alternative approach, expecting a coordination among the agents for a common goal. An observation on soccer domain characteristics lead us to experiment the task decomposition based approach on man passing problem. We implemented and comparatively discussed the results.

In most of our experiments, we used a popular reinforcement learning method that makes effective use of temporal differences, named Q(λ) learning, in order to show that role decomposition can be effective. The results of experimentations and the effectiveness of role decomposition in multi-agent reinforcement learning is discussed.

Organization of thesis

Chapter 1 introduces the agent learning concepts and is a brief summary on attacked problem. Chapter 2 is a step-by-step introductory material for multi-agent reinforcement learning. Chapter 3 presents simulated soccer domain, with emphasis on its properties used in this thesis. Chapter 4 includes the necessary definitions of rule-decomposition, description of two problem domains and in-

volves the experiments done on these domains. In chapter 5, a discussion on the work done is made, and some open problems are indicated.

CHAPTER II

MULTI-AGENT REINFORCEMENT LEARNING

Recently, a powerful trend has emerged in problem solving in computing systems which is based on modularity, adaptivity, and autonomy. This approach depends on the idea that whenever the software entity starts to *live* (possibly with an initial knowledge), its motivation is just to adapt to its environment through interactions if necessary, and take action decisions in order to maximize an internal *success measure*. This form of software is called *agent*. An *agent*, in most general sense, is anything that perceives and acts in an environment [Russell and Norvig 95].

Autonomy of an agent usually refers to the freedom an agent has while executing in an environment. Of course, an agent is autonomous in this sense up to the level specified by its design.

A typical domain where autonomy is important is distributed computing systems. In this area, agent theory has gained more importance recently, since mobile

software is the most suitable solution for many tasks. When we talk about mobility of a program, autonomy is necessary. After the agent leaves the system it is created in, all its action decisions need to be made by itself.

In real life, on the other hand, autonomy is a must, but still not enough. The agent is surrounded by many distracting or unnecessary information. Thus, the agent must develop an adaptation strategy. For example, a mobile robot trying to find a path to the door must *learn* that an obstacle on the way to door is something bad for its purpose, and in order to save power, it should drive not directly to the obstacle, but around.

In many problem domains, more than one agent helps each other for achievement of a goal. This may be just to speed up the solution, or a necessity for the nature of the problem. In either way, the resulting system is said to be *multi-agent*.

II.1 Multi-Agent Systems

If more than one agent is involved in solution of a problem task, we talk about a *multi-agent system* (MAS). The nature of a MAS is designated by the agent architecture, goal to be achieved and the need for coordination among the agents. MASs are known to be effective in areas like open, dynamic, and complex systems, and widely used in the industry (process control, air traffic control), in commercial applications (information management, electronic commerce) and in entertainment (games, simulations).

In many problem domains, an agent capable of doing everything for the solution is not a realistic approach. There are many constraints like space required

to hold perceptual information, memory and decision parameters; limited time to solve the problem, huge amount of environmental data to be explored etc. A very straightforward solution is to divide the overall task into subtasks, assign each subtask to an agent, and design a coordination among agents, such that a probably better solution is reached faster, using tolerable amount of computational space. This is the key idea behind MASs.

Agents in a MAS may be identical in terms of their design. If similar agents try to achieve a common goal, the system is *homogeneous.* On the other hand, in *heterogeneous* systems, each agent or group of agents is donated with different capabilities and has a local *sub-goal.*

By the nature of MASs, the observed world is *not stationary* in at least one agent's point of view. Moreover, agent's perception mechanisms are usually not sufficient to have a complete information about its environment, which is known as *partial observability.* Although there are MASs in which a superior being (like human or a global controller) helps the agents to overcome this difficulty, this is not a desirable property, since it is a violation of autonomy property of an agent. Thus, many AI techniques are used to make the agent adaptive.

The problem has gained importance recently, so that a new AI research direction, *distributed artificial intelligence* (DAI) has emerged. DAI is a relatively new and hot topic of AI that aims to provide models of agent operation and interaction in complex, dynamic and unpredictable environments for problem solving. In the context of DAI, almost all known AI techniques are extended to the multi-agent case, and new methods are being developed.

AI has many solution candidates, such as artificial neural networks, deci-

sion trees, genetic algorithms, reinforcement learning etc. An observation of the need for adaptivity, non-stationary nature of the domain and observability limits of multi-agent environments force us to eliminate those techniques that require complete knowledge of environment, need for a supervisor's help, and static environmental rules. Among the remaining learning algorithms, artificial neural networks and reinforcement learning techniques dominate. However, reinforcement learning seems to be the most suitable candidate for agent design for a MAS, since it is based on the interaction between agent and its environment, and can easily be implemented to perform on-line. Note that since other agents are also part of the environment in one of the agent's point of view, the method scales the effort up to a multi-agent learning solution.

II.2 Reinforcement Learning

Reinforcement learning (RL) stands for a family of machine learning methods that originates from *dynamic programming* (DP). DP is a field of mathematics that deals with solution of optimization and control problems. On the other hand, RL has a different aspect of background based on psychology of animal learning by trial-and-error [Sutton and Barto 99].

RL is based on learning a behavior through trial-and-error interactions with the environment (Figure II.1). More precisely, it is a technique to estimate the utility of taking actions in states of the world, where learner (agent) is supposed to adapt to [Kaelbling *et al.* 96].

In RL approaches, the learner (agent) takes actions in the environment, and for some actions taken in certain states of the world, it receives "punishment"

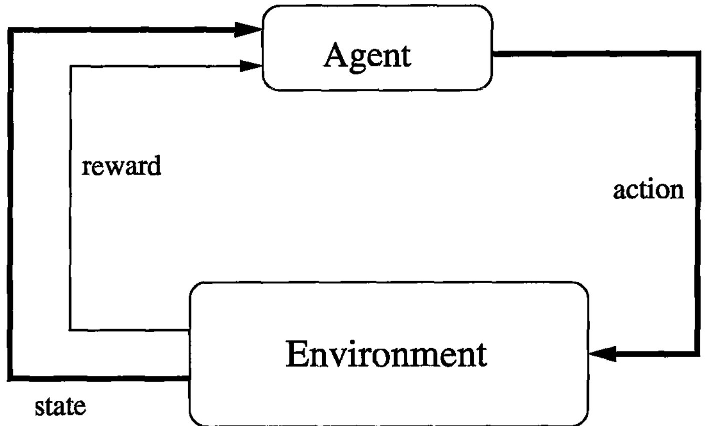


Figure II.1: Agent-environment interaction in RL.

or "reward" signals, called *reinforcement signals.* Usually, learning is an *episodic task* in RL domains, meaning that certain state(s) of the world has (have) no possible next state, so that when one of these states is reached, a new learning episode is started.

II.2.1 Markov Decision Processes

RL is defined on environments that can be modeled as *Markov Decision Process* (MDP). Formally, a MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \tau, \mathcal{R}, \tau_0 \rangle$ such that

- $\mathcal{S}$ is the set of states in the environment,
- $\mathcal{A}$ is the set of possible actions (some actions might be unavailable in some states),
- $\tau : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the probability distribution function by which the transition from state s to state s' occurs when action $a \in \mathcal{A}$ is taken. This is denoted by $\tau(s'|s, a)$, and called *transition probability,*
- $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$ is the real valued *reward function* received when action

$a \in \mathcal{A}$ if performed which causes the transition from s to s', and

- τ_0 is the starting state distribution.

Given a MDP, a policy, π, is defined to be a function of the form $\pi : \mathcal{S} \rightarrow \mathcal{A}$.

A RL algorithm seeks for the answer to the question "how good is taking an action for a given state?" for all states of the environment. This can be achieved in two ways:

- Learning *state-value function* for policy π: Value of state s under a policy π, denoted $V^{\pi}(s)$ is the expected return when starting in s and following π thereafter. For MDPs, $V^{\pi}(s)$ is defined as

$$V^{\pi}(s) = E_{\pi}\left\{ret_{t\infty}|s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\} \quad \text{(II.1)}$$

- Learning *action-value function* for policy π: Value of action a in state s under a policy π, denoted $Q^{\pi}(s,a)$, is the expected return starting from s, taking the action a, and thereafter following policy π. $Q^{\pi}(s)$ is defined as

$$Q^{\pi}(s,a) = E_{\pi}\left\{ret_{t\infty}|s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\right\} \quad \text{(II.2)}$$

For the equations II.1 and II.2, $E_{\pi}\{\}$ denotes the expected value given that the agent follows the policy π, γ is the *discount rate*, r_t is the reward received at time step t. The aim of both approaches is to maximize the expected *discounted return* $ret_{t\infty}$ where

$$ret_{tT} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^T r_{t+T+1} \quad \text{(II.3)}$$

$$ret_{t\infty} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \quad \text{(II.4)}$$

The discount rate determines how "farsighted" the learning is; in other words, is $\gamma = 0$, only immediate rewards are to be maximized, if $0 < \gamma < 1$, a reward to be taken in the future has a certain limited (depending on whether γ is near 0 or 1) influence to expected return compared to its immediate application, if $\gamma = 1$, future rewards are equally taken into calculation [Sutton and Barto 99].

II.2.2 Dynamic Programming

From the equation II.1, one can derive the following sequence of equations

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left\{r_t \gamma \sum_{k=0}^{\infty} r_{t+k+2} | s_t = s\right\} \\ &= \sum_a \pi(s|a) \sum_{s'} \tau(s'|s,a) \left[R(s'|s,a) + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1}\right\}\right] \\ &= \sum_a \pi(s|a) \sum_{s'} \tau(s'|s,a) \left[R(s'|s,a) + \gamma V^{\pi}(s')\right] \qquad \text{(II.5)} \\ &= R(s'|s,\pi(s)) + \gamma \sum_{s'} \tau(s'|s,\pi(s)) V^{\pi}(s') \qquad \text{(II.6)} \end{aligned}$$

where the equation II.5 is known as the *Bellman equation* [Bellman 57], which gives possibility to solution of V^{π} values via DP. The equation II.6 is a simple rearrangement of II.5, and is also a Bellman equation.

Simply re-writing equation II.6 in terms of action-value function, we obtain

$$Q^{\pi}(s,a) = R(s'|s,a) + \gamma \sum_{s'} \tau(s'|s,a) Q^{\pi}(s',\pi(s')) \qquad \text{(II.7)}$$

The existence and uniqueness of V^{π} is guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π. If the environment's dynamics are completely known, then II.5 is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $V^{\pi}(s), s \in \mathcal{S}$). Iterative

solution methods are suitable for this task. Consider the rearranged form of II.1:

$$V_{i+1}(s) = E_\pi \{r_{t+1} + \gamma V_i(s_{t+1}) | s_t = s\} \tag{II.8}$$

Given an arbitrary initial approximation V_0 other than the terminal state, each successive approximation of V^π can be calculated iteratively using equation II.8. The sequence $\{V_i\}$ can be shown in general to converge to V^π as $i \to \infty$ under the same conditions that guarantee the existence of V^π. Computation of V^π for an arbitrary policy π is called *policy evaluation* [Sutton and Barto 99].

Finding Optimal Policies

An optimal state-value function V^* is the value function that simultaneously maximizes the expected cumulative reward in all states $s \in \mathcal{S}$, and called *Bellman optimality equation*:

$$V^*(s) = max_a \left[R(s'|s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^*(s') \right] \tag{II.9}$$

Like optimal state-value function, an optimal action-value function is denoted $Q^*(s,a)$ and satisfies the equation

$$Q^*(s,a) = R(s'|s,a) + \gamma \sum_{s'} \tau(s'|s,a) \max_{a'} Q^*(s', \pi(s')) \tag{II.10}$$

A policy π is called *greedy* and denoted π^* iff

$$R(s'|s,\pi(s)) + \gamma \sum_{s'} \tau(s'|s,\pi(s)) V(s') \geq R(s'|s,a) + \gamma \sum_{s'} \tau(s'|s,a) V(s') \tag{II.11}$$

for $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$.

Following the inequality II.11, given the optimal value function V^*, optimal policy π^* can be found by

$$\pi^*(s) = arg\ max_{a \in \mathcal{A}} \{R(s'|s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^*(s')\} \tag{II.12}$$

In terms of action-value function,

$$\pi^*(s) = arg\ max_a Q(s, a) \tag{II.13}$$

Once the value function is approximated for a given policy, the next task is to find the best policy (or policies) among all possibilities. This is called *policy improvement.*

Policy improvement can be achieved in two ways:

- *Policy iteration*: After each policy evaluation, an *improvement* step is executed on the current policy, until the best policy is found.
- *Value iteration*: Similar to policy iteration, but without waiting for policy evaluation convergence, the evaluation is truncated in several ways without losing the power of policy iteration.

Note that in either way, value estimate for current state is calculated based on successor states' value estimates. This is called *bootstrapping* in DP literature.

II.2.3 Monte Carlo Methods

DP methods can effectively approximate the value function given the complete knowledge of the environment. However, in many problem domains, full environmental information is not available, or there is no *model* of the environment. For such a case, environment must be observed through *experience*. *Monte Carlo* (MC) methods provide some means of making this possible. All MC methods assume the learning task is episodic.

In order to estimate $V^\pi(s)$, the value of state s under policy π, given a set of episodes obtained by following π and passing through s, we first define each

1. Initialize
2. $\pi \leftarrow$ policy to be evaluated
3. $V \leftarrow$ an arbitrary state-value function
4. $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
5. Repeat forever:
6. (a) Generate an episode using π
7. (b) For each state s appearing in the episode:
8. $R \leftarrow$ return following the first occurence of s
9. Append R to $Returns(s)$
10. $V(s) \leftarrow$ average($Returns(s)$)

Figure II.2: First-visit MC method for estimating V^{π}.

occurrence of state s in an episode to be a *visit* to s. Then, *every-visit MC method* estimates $V^{\pi}(s)$ as the average of the returns following all the visits to s in a set of episodes. Another approach, called *first-visit MC method* (Figure II.2) averages just the returns following first visits to s. Both first-visit MC and every-visit MC converge to $V^{\pi}(s)$ as the number of visits (or first-visits) to s goes to infinity [Sutton and Barto 99].

If there is no model available, using action-values is better since simple one step look-aheads might not give sufficient information about the best next state. However, using action-values to estimate $Q^{\pi}(s, a)$, some of the action-values might never be visited, since following a deterministic policy π may cause observation of just one action outcome per state. The solution is to specify starting state of each episode by a state-action *pair*, so every state-action pair will be visited in infinite number of episodes. This approach is called *exploring starts.* Another solution to this problem is always using stochastic policies to select actions.

After calculating the estimation of V^{π} or Q^{π}, following a policy improvement technique similar to that of DP, optimal policy can be found by iterative appli-

cation of this cycle forever.

II.2.4 Temporal Difference Learning

Temporal Difference (TD) learning [Sutton 88] is a combination of MC and DP methods. In other words, TD methods learn through experience even if no model is available, using bootstrapping.

For the every-visit MC method, the incremental implementation of value update rule is

$$V(s_t) \leftarrow V(s_t) + \alpha[ret_{0t} - V(s_t)] \tag{II.14}$$

where α is the step-size parameter.

The following subsections will be introducing some well known TD techniques.

II.2.4.1 TD(0) Learning

Using the rule II.14, one waits until the end of the episode to determine the $V(s_t)$ increment, because value of ret_{0t} is available only at the end of the episode. However, TD methods use a slightly different increment rule:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{II.15}$$

The rule II.15, known as *TD(0) update rule*, performs the $V(s_t)$ increment using the immediate observed reward r_{t+1} and existing $V(s_{t+1})$ estimate, thus, this method does not require waiting for the end of episode [Sutton and Barto 99]. Bootstrapping is used since the update is based on an existing estimation, and the rule is model-free since it is derived from an existing MC update.

1. Initialize $V(s)$ arbitrarily, π to the policy to be evaluated
2. Repeat (for each episode):
3. Initialize s
4. Repeat (for each step of episode):
5. $a \leftarrow$ action given by π for s
6. Take action a; observe reward, r, and next state, s'
7. $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
8. $s \leftarrow s'$;
9. until s is terminal

Figure II.3: TD(0) algorithm for V^{π} estimation.

The complete TD(0) algorithm is given in Figure II.3. TD(0) is proven to converge for small α values.

II.2.4.2 Q-Learning

[Watkins 92] developed a TD algorithm based on TD(0), but performing updates on action-values rather than state-values. Different than TD(0), this algorithm converges to the optimal action-value function independent of the policy followed. The algorithm, called *Q-learning*, uses the following modified TD(0) update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{II.16}$$

Under the usual assumption of update requirement for all state-action pairs, and some stochastic approximation conditions on the step-size parameters, Q_t is proved to converge with probability 1 to Q^*. Procedural form of Q-learning is shown in Figure II.4.

In RL literature, Q-learning has a significant importance in the sense that it provides a way to learning of optimal policy through approximation of optimal action-value instead of policy iteration. This approach is called *off-policy* learning.

1. Initialize $Q(s,a)$ arbitrarily
2. Repeat (for each episode):
3. Initialize s
4. Repeat (for each step of episode):
5. Choose a from s using policy derived from Q (e.g. ϵ-greedy)
6. Take action a, observe r, s'
7. $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma\ max_{a'} Q(s',a') - Q(s,a)]$
8. $s \leftarrow s'$;
9. until s is terminal

Figure II.4: One step Q-learning.

In this point of view, Q-learning is an *off-policy TD control algorithm.*

On-policy version of Q-learning, named SARSA [1] in short, was explored by [Rummery and Niranjan 94]. The update rule of SARSA is

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\,[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \qquad \text{(II.17)}$$

The main difference of SARSA from Q-learning is this update rule. Unlike Q-learning, SARSA seeks for an estimate of $Q^{\pi}(s,a)$, for a given policy π.

II.2.4.3 Using Eligibility Traces

Both MC and TD methods have their own advantages and disadvantages. There is a way of combining MC value-backup strategy with almost all TD learning algorithms, resulting in a family of algorithms that converge faster and can be applied on a broader range of domains, including those that satisfy MDP characteristics only partially.

Rule II.14 requires the discounted return be available for each step of incremental update of $V(s_t)$. In rule II.15, single-step reward mechanism is sufficient.

[1] The name SARSA was introduced by Sutton in 1996, as an acronym for State-Action-Reward-State-Action. The name used by Rummery was *modified Q-learning.*

In the combined approach, the discounted return is *backed-up* to an intermediate point, say n^{th} value after current, and evaluated. This approach is called *n-step backup*, and TD methods where temporal difference extends over n steps instead of 1 are called *n-step TD methods.* Evaluation of *n-step return* for a TD algorithm can be realized by holding an associated memory variable for each state s, called *eligibility trace*, and denoted by $e_t(s)$, defined by

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & , if \; s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1, & if \; s = s_t \end{cases} \tag{II.18}$$

for all $s \in \mathcal{S}$, where γ is the discount rate and $\lambda(0 \leq \lambda \leq 1)$ is the *trace decay parameter.* The traces are said to indicate the degree to which each state is "eligible" for undergoing learning changes, should a reinforcing event occur. A reinforcing event at time t, δ_t, is the one-step TD error which was also used in TD(0):

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{II.19}$$

[Sutton and Barto 99]

The complete algorithm is shown in figure II.5, and is called $TD(\lambda)$. It can easily be observed that when $\lambda = 0$ or $\lambda = 1$, algorithm in figure II.5 becomes one-step TD learning and MC learning, respectively.

As one might expect, the version of the eligibility trace approach using action-values is available. However, the application of eligibility traces to Q-learning or SARSA is not trivial, since TD-errors are not added up correctly unless greedy action is taken always (which means, in fact, application of *current* policy). [Watkins 89] was the first to observe that, in order to apply eligibility traces

```
1. Initialize V(s) arbitrarily, for all s ∈ S
2. Repeat(for each episode):
3.     Initialize e(s) = 0, for all s
4.     Initialize s
5.     Repeat (for each step of episode):
6.         a ← action given by π for s
7.         Take action a, observe r, and next state s'
8.         Choose a' from s' using policy derived from Q (e.g. ε-greedy)
9.         δ ← r + γQ(s') − V(s)
10.        e(s) ← e(s) + 1
11.        For all s:
12.            V(s) ← V(s) + αδe(s)
14.            e(s, a) ← γλe(s)
15.        s ← s'
16.    until s is terminal
```

Figure II.5: TD(λ) algorithm.

```
1. Initialize Q(s, a) arbitrarily, for all s, a
2. Repeat(for each episode):
3.     Initialize e(s, a) = 0, for all s, a
4.     Initialize s, a
5.     Repeat (for each step of episode):
6.         Take action a, observe r, s'
7.         Choose a' from s' using policy derived from Q (e.g. ε-greedy)
8.         a* ← arg max_b Q(s', b) (if a' ties for the max, then a* ← a')
9.         δ ← r + γQ(s', a*) − Q(s, a)
10.        e(s, a) ← e(s, a) + 1
11.        For all s, a:
12.                Q(s, a) ← Q(s, a) + αγe(s, a)
13.                If a' = a*, then e(s, a) ← γλe(s, a)
14.                          else e(s, a) ← 0
15.        s ← s'; a ← a'
16.    until s is terminal
```

Figure II.6: Watkin's Q(λ) algorithm.

to Q-learning, all eligibility values should be reset unless a greedy action is selected. This version of algorithm [2], denoted by $Q(\lambda)$, is shown in Figure II.6 $SARSA(\lambda)$, on-policy version of Q(λ) was explored as a control method by [Rummery and Niranjan 94].

Holding eligibility traces is effective especially when

- reward is delayed [Watkins 89]
- environment has non-Markovian characteristics [Peng and Williams 96]

There are several convergence proofs for TD methods with eligibility traces, however, although there are many experimentally good results, convergence of Q(λ) and SARSA(λ) has not been proved for $0 < \lambda < 1$.

II.3 Multi-Agent Reinforcement Learning

From the perspective of a single agent, existance of other agents in the environment makes no significant difference in terms of individual motivation on benefit maximization. In a broader sense, however, the group of agents acting together, each trying to maximize its expected reward, may perform a series of independent actions resulting in a collective behavior better than that in a single agent case. The field of DAI that deals with scaling up RL to MASs this way is called *multi-agent reinforcement learning* (MARL).

Although coordination among agents to achieve a goal more efficiently can be realized by cooperation through information sharing [Tan 93], much work relies

[2] In RL literature, there are two frequently used Q(λ) algorithms. One of them is due to [Watkins 89], and the other is due to [Peng and Williams 96].In this thesis, Watkins' algorithm is meant whenever Q(λ) is referred.

on an carefully designed mechanism only. For example, a good distribution of rewards among agents is an important factor.

In their study, [Claus and Boutilier 98] argued the conditions under which RL can be usefully applied to MASs directly. They discussed dynamics of RL in MASs, in terms of learning *joint* actions, convergence settings and system structure, by focusing on the influence of partial action observability and exploration strategies.

[Sen *et al.*] discussed how RL can be used by multiple agents to learn coordination strategies without having to rely on shared information. They experimented block-pushing problem with two agents. In their work, each agent was independently optimizing its own environmental reward, but a global coordination behavoiur emerged. The learning system, thus, is robust and general-purpose.

[Abul 99] experimented MARL without explicit cooperation on an adversarial grid-world domain. He tested various RL algorithms together with function approximation techniques for generalization of large state-space, in order to achieve coordination among agents.

Because of the dynamic nature of MASs, direct MARL applications are usually limited or heuristic by nature. There are studies where RL methods are equipped with methods that improve multi-agent learning.

[Schmidhuber 96] added a stack-based backtracking procedure to RL called "environment-independent reinforcement acceleration" (EIRA), by which each agent holds a history of significant performance accelerations, using an acceleration criterion. This way, agents are enforced to learn cooperation in situations where all agents try to speed up the same reinforcement signals, but no agent can

speed up reinforcement intake immediately by itself.

[Weiß 93] described two algorithms for learning of appropriate sequences of action sets in MASs with RL, which he called ACE and AGE (standing for "ACtion Estimation" and "Action Group Estimation" respectively). Both algorithms are based on learning an estimation of goal relevance for actions, using coordination. To do so, he introduced a *bidding* procedure among agents where each agent competes for the right to take an action and he tested his algorithms on block world domain.

CHAPTER III

SIMULATED SOCCER DOMAIN

III.1 Overview

Soccer game is one of the most challenging domains for AI studies, due to its extremely dynamic and complex nature. Any close-to-real soccer match is a *multi-agent system*, where each agent has only a partial view of its surrounding (*hidden state*) and must decide its action immediately according to its current belief, using the limited information it gathers from the environment (*real-time decision*). Teammates must *cooperate* to defeat their opponents, because the soccer domain is *adversarial*. Moreover, each agent must cope with a huge *continuous state* and *action space*.

The frequently used grid world domain is not suitable to reflect the main characteristics of soccer domain. [Uther and Veloso 97] modified the traditional grid world to be composed of hexagonal grids, in order to achieve a closer imitation of soccer game, and also to make effective comparisons among learning algorithms

giving results best for grid worlds. However, the new grid domain was far from being similar to a soccer domain, in the sense that continuous state and action space was still missing.

In their study, [Slustowicz *et al.* 97] used a soccer simulator of their own design, as a test-bed for evolving soccer strategies. The enhancement was the existence of continuous 2-dimensional Cartesian soccer field, and continuous action space resulted in a much better soccer simulation. However, capturing the ball was done by occupying a neighborhood of the ball, even if it were already captured by another player. By this assumption, behavior of ball becomes unrealistic.

In the meantime, proposal of the robotic soccer cup, named *RoboCup*, was made by [Kitano *et al.* 97], in order to provide a common soccer platform to compare the performance of systems developed by many researchers and robotic soccer hobbiests. Following a preRoboCup in 1996, the first official RoboCup event occurred in 1997[1]. Since then, the competition is held once every year.

RoboCup consists of 3 sections of games [Noda 95]:

- a real-robot section
- a simulation section
- a special-skill section

For competitions in the simulation section, simulated *Soccer Server* was announced to be the official simulation tool [Kitano *et al.* 97]. The server was based on the previous work of Itsuki Noda et. al., in September of 1993. While Noda's system was built as a library module for demonstration of a Prolog-like

[1] http://www.robocup.org

programming language, the first version was written in LISP, before its official announcement. Beginning from its first official version, it has been developed in C++ [Noda *et al.* 99].

Since it is first used in preRoboCup'96, the soccer server system is being updated frequently for bug-fixes, and rarely for additional features.

III.2 RoboCup Soccer Server

This section paraphrases a similar section in the soccer server user manual [Noda *et al.* 99].

III.2.1 General Description

RoboCup Soccer Server is a 2-dimensional soccer simulator, written in C++. It enables autonomous agents consisting of programs written in various languages to play a match of soccer against each other. It provides a virtual platform to which each agent program is connected via UDP/IP socket connection, each representing a soccer player; therefore any system having UDP/IP support can be thought of as a potential soccer agent development platform.

Official RoboCup Soccer Server consists of two programs. The actual server is `soccerserver`. It is a program that simulates ball and player movements in a virtual soccer field, communicates with player client programs (agents), simulates some environmental noise (wind factor, player stamina etc.) in order to make the simulation more realistic. `soccerserver` also has a built-in referee to control the rules of the match. The other program, `soccermonitor` is a graphical front-end for `soccerserver` which uses X window graphical system [Noda *et al.* 99].

`soccerserver` has an internal clock, and changes in status of server occur by a new clock tick, which is called a *simulation step.* Using the UDP/IP connection, an agent program sends action commands to `soccerserver`. These commands are interpreted by the `soccerserver` and the response is the corresponding change in state of the virtual environment according to the action taken. The action model is discrete, in other words, all action requests within one time step occurs at the end of the step. `soccerserver` keeps track of sensory status of each player and sends the current sensory information to each agent in certain time intervals (possibly a different step size than simulation step). Whenever an agent needs this information, it simply listens to the `soccerserver` socket port connection. Following a historical convention, information is passed among `soccerserver` and the agents using strings containing *S-expressions.*

III.2.2 Simulation of Soccer Field

The game field of `soccerserver` is a 2-dimensional continuous coordinate system in which center of field is accepted to be the (0, 0) point, and all stable objects are indicated with *flags* and *lines*, and located accordingly.

The possible moving objects are the ball and the players. Maximum number of agents that can be connected to the server for one team is 11, which adds up to a total of 22 players. Additionally, a coach agent can be connected, but is not represented as a moving entity in the field.

> "The players and the ball are treated as circles. Front side of a player is represented by unshading the half-circle of corresponding direction. All distances and angles used are to the centers of the circles.
>
> In each simulation step, movement of each object is calculated as:

$$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) \quad : \text{accelerate} \qquad \text{(III.1)}$$

$$
\begin{aligned}
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1}) \quad \text{: move} \\
(v_x^{t+1}, v_y^{t+1}) &= decay \times (u_x^{t+1}, u_y^{t+1}) \quad \text{: decay speed} \\
(a_x^{t+1}, a_y^{t+1}) &= (0, 0) \quad \text{: reset acceleration}
\end{aligned}
$$

where, (p_x^t, p_y^t), and (v_x^t, v_y^t) are respectively position and velocity of the object in timestep t. *decay* is a decay parameter specified by `ball_decay` or `player_decay`. (a_x^t, a_y^t) is acceleration of object, which is derived from *Power* parameter in `dash` (in the case the object is a player) or `kick` (in the case of a ball) commands in the following manner:

$$
(a_x^t, a_y^t) = Power \times power_rate \times (cos(\theta^t), sin(\theta^t))
$$

where θ^t is the direction of the object in timestep t and *power_rate* is `dash_power_rate` or is calculated from `kick_power_rate`. In the case of a player, this is just the direction the player is facing. In the case of a ball, its direction is given by the following manner:

$$
\theta_{ball}^t = \theta_{kicker}^t + Direction
$$

where θ_{ball}^t and θ_{kicker}^t are directions of ball and kicking player respectively, and *Direction* is the second parameter of a `kick` command.

If at the end of the simulation step, two objects overlap, then the objects are moved back until they do not overlap. Then the velocities are multiplied by -0.1. It is possible for the ball to go through a player as long they do not overlap at the end of cycle.

In order to reflect unexpected movements of objects in real world, `soccerserver` adds noise to the movement of objects and parameters of commands.

Concerned with movements, noise is added into Equation III.1 as follows:

$$
(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}_{r_{max}}, \tilde{r}_{r_{max}})
$$

where $\tilde{r}_{r_{max}}$ is a random number whose distribution is uniform over the range $[-r_{max}, r_{max}]$. r_{max} is a parameter that depends on amount of velocity of the object as follows:

$$
r_{max} = rand \cdot |(v_x^t, v_x^t)|
$$

where *rand* is a parameter specified by `player_rand` or `ball_rand`.

Noise is added also into the *Power* and *Moment* arguments of a command as follows:

$$
argument = (1 + \tilde{r}_{rand}) \cdot argument
$$

[Noda *et al.* 99]”

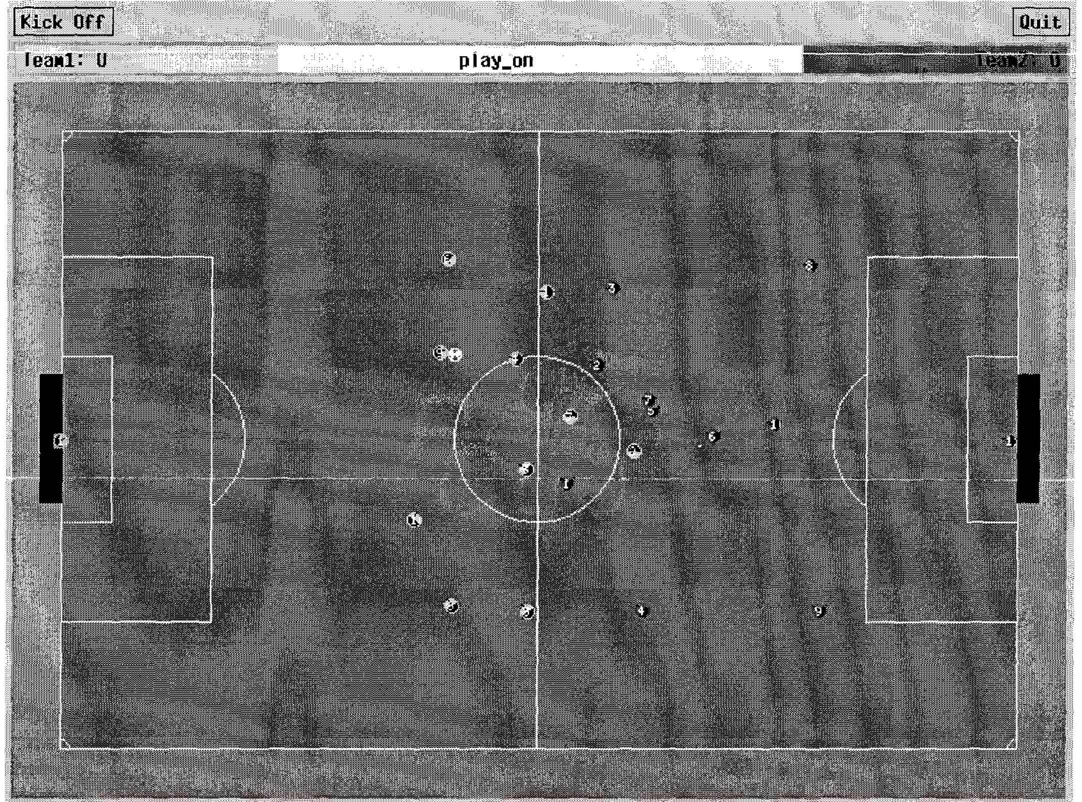


Figure III.1: A snapshot from soccer server

III.2.3 Player Simulation

III.2.3.1 Commands to Server

An agent program can send eight types of commands to `soccerserver`. These commands are `turn`, `dash`, `kick`, `turn-neck`, `move`, `catch`, `say`, `change_view`. First half of the commands are *basic action commands.* In this thesis, first three basic action commands and the `move` command are used:

- `(turn` *Moment*`)`

  Change the direction of the player according to *Moment*. *Moment* should be between `minmoment` and `maxmoment` (default is [-180, 180]). The actual change of direction is reduced when the player is moving quickly. Specifi-

cally, the actual angle the player is turned is as follows:

$$actual_angle = moment/(1.0 + inertia_moment \times player_speed)$$

[Noda *et al.* 99]

- `(dash` *Power*`)`

  Increases the velocity of the player in the direction it is facing by *Power* $\times$ `dash_power_rate`. *Power* should be between `minpower` and `maxpower` (default: [-30, 100]). If power is negative, then the player is effectively dashing backwards. [Noda *et al.* 99]

- `(kick` *Power Direction*`)`

  Kick the ball with *Power* in *Direction* if the ball is near enough (the distance to the ball is less than `kickable_area` (which is equal to the $kickable_margin + ball_size + player_size.$). *Power* should be between `minpower` and `maxpower` (default is [-30, 100]). *Direction* should be between `minmoment` and `maxmoment` (default is [-180, 180]). Most powerful kick can be done when the ball is directly in front of the player and very close to him, and drops off as both distance and angle increase. [Noda *et al.* 99]

- `(move` *X Y*`)`

  Move the player to the position *(X Y)*. The origin is the center mark, and the X-axis and Y-axis are toward the opponent's goal and the right touch-line respectively. Thus, X is usually negative to locate a player in its own side of the field. This command is available only in the `before_kick_off` mode, and for the goalie immediately after catching the ball (`catch` com-

mand). [Noda *et al.* 99]

III.2.3.2 Sensor Information

Three types of messages arrive from `soccerserver` to clients: visual, auditory, physical (related to self condition). In this thesis, there is massive use of visual information of agents. No passing of auditory or sensory type message is involved.

Visual Information

> "Visual information arrives from the server in the following basic format:
> (*ObjName Distance Direction DistChng DirChng BodyDir HeadDir*)

```
ObjName ::= (player Teamname UniformNumber)
            |(goal [l|r])
            |(ball)
            |(flag c)
            |(flag [l|c|r] [t|b])
            |(flag p  [l|r] [t|c|b])
            |(flag g [l|r] [t|b])
            |(flag [l|r|t|b] 0)
            |(flag [t|b] [l|r] [10|20|30|40|50])
            |(flag [l|r] [t|b] [10|20|30])
            |(flag [l|r|t|b])
```

> *Distance, Direction, DistChng* and *DirChng* are calculated in the following way:

$$
\begin{aligned}
p_{rx} &= p_{xt} - p_{x0} \\
p_{ry} &= p_{yt} - p_{y0} \\
v_{rx} &= v_{xt} - v_{x0} \\
v_{ry} &= v_{yt} - v_{y0} \\
Distance &= \sqrt{p_{rx}^2 + p_{ry}^2} \\
Direction &= \arctan p_{ry}/p_{rx} - a_0 \\
e_{rx} &= p_{rx}/Distance \\
e_{ry} &= p_{ry}/Distance \\
DistChng &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) \\
DirChng &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/Distance] * (180/\pi)
\end{aligned}
$$

where (p_{xt}, p_{yt}) is the absolute position of the target object, $(p_{x0}, py0)$ is the absolute position of the sensing player, (v_{xt}, v_{yt}) is the absolute velocity of the target object, (v_{x0}, v_{y0}) is the absolute velocity of the sensing player, and a_0 is the absolute direction the sensing player is facing. The absolute facing direction of a player is the sum of the *BodyDir* and the *HeadDir* of that player. In addition to it, (p_{rx}, p_{ry}) and (v_{rx}, v_{ry}) are respectively the relative position and the relative velocity of the target, and (e_{rx}, e_{ry}) is the unit vector that is parallel to the vector of the relative position. *BodyDir* and *HeadDir* are only included if the observed object is a player, and is the body and head directions of the observed player relative to the body and head directions of the observing player. Thus, if both players have their bodies turned in the same direction, then *BodyDir* would be 0. The same goes for *HeadDir*.

The `(goal r)` object is interpreted as the center of the right hand side goalline. `(flag c)` is a virtual flag at the center of the field. `(flag l b)` is the flag at the lower left of the field. `(flag p l b)` is a virtual flag at the lower right corner of the penalty box on the left side of the field. `(flag g l b)` is a virtual flag marking the right goalpost on the left goal. The remaining types of flags are all located 5 meters outside the playing field. For example, `(flag t l 20)` is 5 meters from the top sideline and 20 meters left from the center line. In the same way, `(flag r b 10)` is 5 meters right of the right sideline and 10 meters below the center of the right goal. Also, `(flag b 0)` is 5 meters below the midpoint of the bottom sideline.

In the case of `(line ...)`, *Distance* is the distance to the point where the center line of the player's view crosses the line, and *Direction* is the direction of the line.

All the flags and lines are shown in Figure III.2. [Noda *et al.* 99]"

Range of view of the player is limited and it depends on several factors. The server parameters `send_step` and `visible_angle` and player parameters `view_quality` $\in \{high, low\}$, `view_width` $\in \{narrow, normal, wide\}$, and the floating point number `view_angle`. Default values of `view_quality`, `view_width` and `view_angle` are $high, normal$ and `visible_angle`. In that case the agent receives visual information every `send_step` milliseconds [Stone 98]. To calculate the `view_frequency` and `view_angle` of the agent, the following equations are

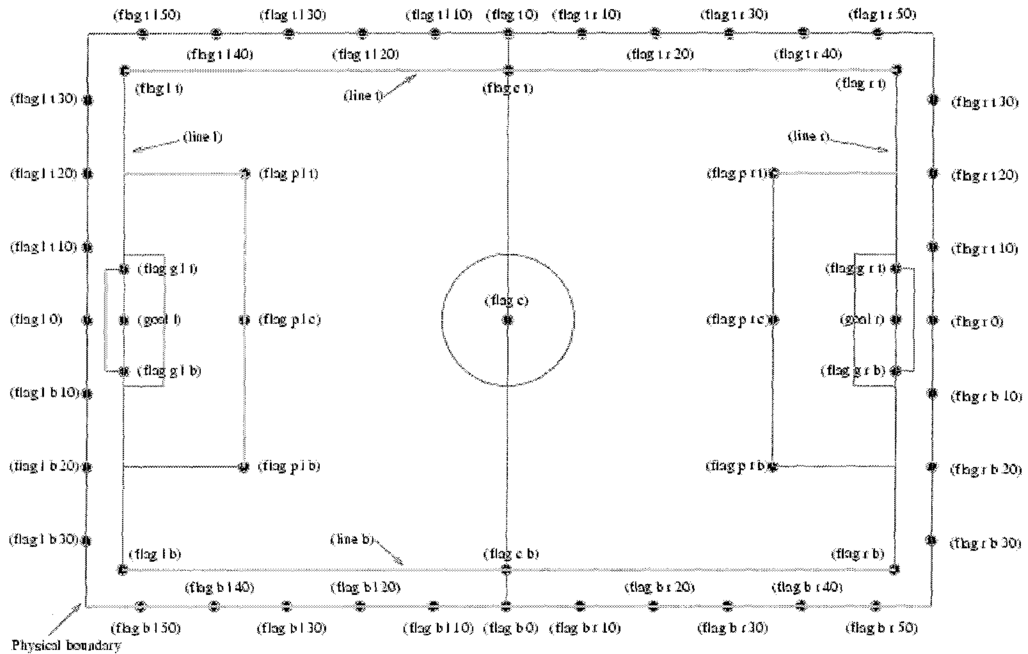


Figure III.2: Flags and lines in soccer server

used:

$$\texttt{view_frequency} = \texttt{sense_step} * \texttt{view_quality_factor} * \texttt{view_width_factor} \tag{III.2}$$

where view_quality_factor is 1 iff view_quality is *high* and 0.5 iff view_quality is *low*, view_width_factor is 2 iff view_width is *narrow*, 1 iff view_width is *normal*, and 0.5 iff view_width is *wide*.

$$\texttt{view_angle} = \texttt{visible_angle} * \texttt{view_width_factor} \tag{III.3}$$

where view_width_factor is 0.5 iff view_width is *narrow*, 1 iff view_width is *normal*, and 2 iff view_width is *wide* [Noda *et al.* 99].

A player can directly adjust its view quality and and its frequency on-line by modifying the player view parameters view_quality and view_width, together

called `view_mode`. However, there is a trade-off between frequent view information versus high quality visual data by the calculations in equations III.2 and III.3.

Another parameter defining the view cone is `visible_distance`:

> "As illustrated in Figure III.3, the amount of information describing a player varies with how far the player is. For nearby players, both the team and the uniform number of the player are reported. However, as distance increases, first the likelihood that the uniform number is visible decreases, and then even the team name may not be visible. It is assumed in the server that `unum_far_length` $\leq$ `unum_too_far_length` $\leq$ `team_far_length` $\leq$ `team_too_far_length`. Let the player's distance be *dist*. Then
>
> - If *dist* $\leq$ `unum_far_length`, then both uniform number and team name are visible.
> - If `unum_far_length` $<$ *dist* $<$ `unum_too_far_legth`, then the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as *dist* increases.
> - If *dist* $\geq$ `unum_too_far_length`, then the uniform number is not visible.
> - If *dist* $\leq$ `team_far_length`, then the team name is visible.
> - If `team_far_length` $<$ *dist* $<$ `team_too_length`, then the probability that the team name is visible decreases linearly from 1 to 0 as *dist* increases.
> - If *dist* $\geq$ `team_too_far_length`, then neither the team name nor the uniform number is visible.
>
> For example, in Figure III.3, assume that all the labeled black circles are players. Then player c would be identified by both team name and uniform number; player d by team name, and with about 50% chance, uniform number; player e with about a 25% chance, just by team name, otherwise with neither; and player f would be identified simply as an anonymous player.
>
> [Stone 98]"

If an object in the field is not in the view cone of the agent, there are two possibilities:

- The object is not visible (for example, objects b and g in Figure III.3).

- The object may be in `visible_distance` meters far, then it is visible independent of face direction (for example, object a in Figure III.3).

The information about objects seen becomes more unreliable if the object is distant. As written in the manual:

> "In the case that an object in sight is a ball or a player, the value of distance to the object is quantized in the following manner:
>
> $$d' = Quantize(exp(Quantize(log(d), 0.1)), 0.1) \quad \text{(III.4)}$$
>
> where d and d' are the exact distance and quantized distance respectively, and
>
> $$Quantize(V, Q) = ceiling(V/Q) \cdot Q \quad \text{(III.5)}$$
>
> This means that players can not know exact positions of very far objects. For example, when distance is about 100.0 the maximum noise is about 10.0, while then distance is less than 10.0 the noise is less than 1.0.
>
> In the case of flags and lines, the distance value is quantized in the following manner:
>
> $$d' = Quantize(exp(Quantize(lod(d), 0.01)), 0.1) \quad \text{(III.6)}$$
>
> [Noda *et al.* 99]"

Auditory Information

`soccerserver` does not provide a tool for direct agent-to-agent communication. Information sharing can only be done by (in a way) "shouting" to the counterpart. A player can hear only 1 message from each team during 2 simulation cycles. An auditory information is passed to server by a `(say` *Message*`)` command, where *Message* is a character string. An agent can transmit this information by seeking for `(hear` *Time Direction Message*`)` type of message from the server. Communication range is limited to 50 meters from a player.

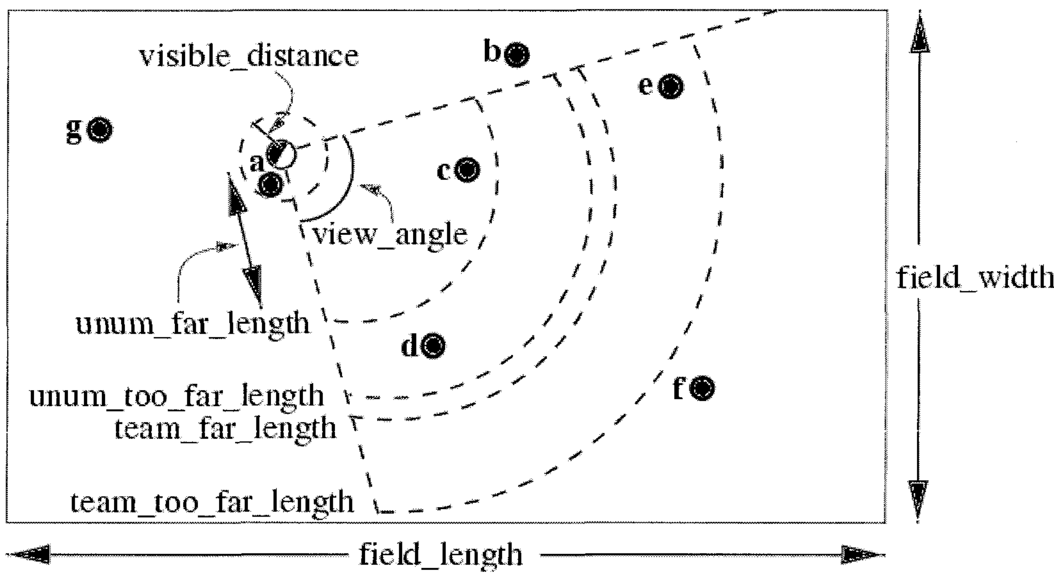


Figure III.3: Visible range of an agent in the soccer server.

Online coach and referee can also send auditory information to players and have no range limitations. These broadcast messages mostly concerning the current status of the match reach to all players.

Physical Information

`soccerserver` periodically (every `sense_step` msec.) sends information to each player about physical status of the player. The information consists of the agent's current

- `view_mode` pair,
- `stamina`, `effort`, and `recovery` values,
- `speed`, `head_angle` values, and
- count of `dash`, `kick`, `turn`, `say` and `turn_neck` actions.

III.2.4 The Coach Client

A special *coach* client can be connected to `soccerserver`. Coach client uses a dedicated port and can rule the game just like a referee. Unlike players, it has the full control over the game (changing play mode, moving players and objects to different locations any time etc.) and full sight of the field. In this thesis, there is effective use of coach client to arrange episodic training sessions, to place players to certain field locations when necessary and to keep track of scores.

CHAPTER IV

LEARNING ROLE DEPENDENT BEHAVIORS

In real life, human-environment or human-human interactions heavily rely on an implicit *role* mechanism. One is temporarily assigned a role until he performs whatever is necessary for that role; after that, he turns back to his initial role or is assigned a new role, and this loop continues. Each role is defined by the task that is to be completed.

Consider the very simple task of carrying a remote object from one place to another. First period of the task involves walking to the object and taking it. This period can be seen as a "take object" task. The second period is walking to the target, and upon reaching the correct place, leaving the object. This procedure can be defined to be a "carry object to target" task.

Learning an obviously composite task is a difficult and open problem in AI. A very new and effective RL solution to this problem is *hierarchical reinforcement*

[illegible library stamp]

learning [Dietterich 00], in which, given a value-function hierarchy, agent tries to get and carry a passenger in a grid-world task, named "taxi domain". This task is composed of subtasks "get passenger", "refuel", and "leave passenger"; and each subtask is composed of sub-subtasks etc. Given this value-function hierarchy, the agent is donated with an initial clue on the optimal hierarchical policy. Learning is said to occur when the order of execution of each subtask at the same level is learned. Later, [Makar *et al.* 01] carried the hierarchical reinforcement learning to multi-agent platform.

In task decomposition, the overall task is divided into subtasks, and whenever one subtask ends, another starts; so the distinction among subtasks is realized by the nature of the problem. Unlike task decomposition, role distribution focuses on the agent's state and action sets. From agent theory point of view, our critical observation is that task decomposition may be a consequence of role decomposition in many cases, especially in multi-agent platforms.

With a more careful observation on the object carrying task defined above, we can see that the overall task is performed based on two distinct roles of the agent respectively: "without object" and "with object".

IV.1 Roles and Coordination

We define a *role* to be a tuple $\langle \mathcal{S}_{role_i}, \mathcal{A}_{role_i} \rangle$, where $\mathcal{S}_{role_i}$ and $\mathcal{A}_{role_i}$ are state and action sets respectively for $role_i$. The current role of the agent is determined *internally* by being in a *role discriminating state* or not. This discrimination may also be realized by taking a special action within a role that takes the agent in or out of the discriminating state.

The subscript $role_i$ ensures the uniqueness of the state and action sets for the ith role. In other words, even if the state set of a role $role_1$ and another role $role_2$ are identical out of the role context, by the role definitions, they become disjoint. Therefore, we can easily integrate the role concept to MDPs.

If a number of roles are observed within a problem domain that can be defined as a MDP, then

$$\mathcal{S} = \bigcup_i \mathcal{S}_{role_i} \tag{IV.1}$$

$$\mathcal{A} = \bigcup_i \mathcal{A}_{role_i} \tag{IV.2}$$

where S and A are as defined in section II.2.1, assures that the domain is still Markovian.

Moreover, given the existence of role discriminating states and provided that actions that carry the agent from or to those states exist, the state transition function τ (see section II.2.1) includes necessary transitions among state set of roles, so that every state in $\mathcal{S}$ is a potential next state of another state. In other words, every state in $\mathcal{S}$ is reachable. This property is a promise for the most important requirement of RL algorithms: need to visit all states infinitely many times.

There are problem domains, in which role definitions are not only obvious, but inevitable as well. A very striking observation is that, in the object carrying task example given at the beginning of this chapter, existence of object location dimension in state space when "without object" role is active is obvious; however, when "with object" role is active, this dimension is useless, because there is no alternative in this role other than carrying the object.

Suppose, more interestingly, that object to be carried is a tool that increases visual sensing capabilities. For instance, when "with object" role is active, the agent's visual depth increases. Obviously, a generic state space description ignoring such a distinction will give unsatisfactory results.

In the multi-agent case, role distinction becomes more meaningful than single agent case. Many problems are, by nature, best solved using complementary role distributions among agents. What we mean by a complementary role is a role that helps the other agent(s) perform its task. Usually, when one agent dedicates itself to a role, the best thing that the other agent(s) can do is to select a -possibly complementary- role that maximizes the overall success.

Coordination of agents in a homogeneous multi-agent system through role distinction can be achieved by a careful design of role state descriptions and reward functions. In many cases, reward is delayed, because after many trials, one of the roles take the reward winning action some time, and this reward is usually distributed among agents.

In this thesis, learning through roles is implemented using Q(λ) (Figure II.6). We chose Q(λ) because of the need for eligibility traces to reduce the non-Markovian effects in the environments. Moreover, the rewards are delayed. A reward is usually received after a long exploration by switching the roles.

For an agent, implementation of roles using look-up table approach in Q-learning requires existence of more than one Q-table, one for each role. This way, the distinction between two same sensory mappings of different roles is achieved. The learning procedure triggers among the tables as the role changes. However, there is no difference between table of one role and the other, as far

as Q-learning is concerned; learning occurs as if there is a single Q-table. In our experiments, we follow this convention, where each agent is equipped with two Q-tables representing two roles.

In general, role distinction is obviously a good solution candidate for multi-agent tasks that deal with "carrying something to somewhere" type of problems.

IV.2 Adversarial Carry-track Domain

We define an adversarial grid-world domain, named carry-track world. In this 10×10 grid-world, there is an object located at one of the sides. The aim of the game is to carry this object to the opposite side. There are two agents that try to carry the object and an enemy that tries to prevent them to achieve this goal.

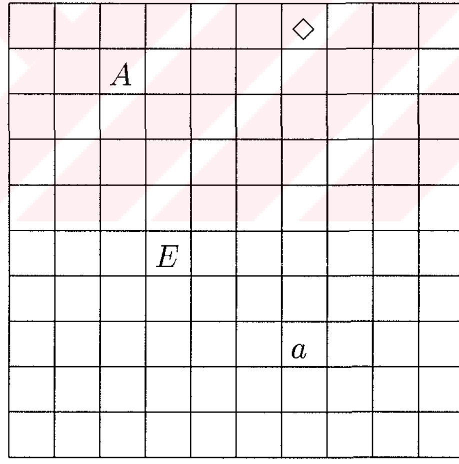
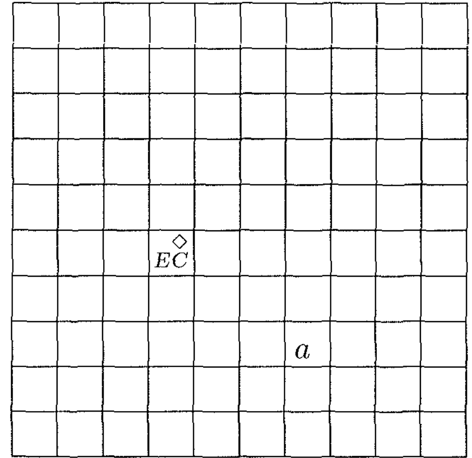


Figure IV.1: A sample initial setting in carry-track world.

Learning agents denoted by a and A as in Figure IV.1, or a and C as in Figure IV.2. The object is denoted by a $\diamond$ and the enemy by E. The general rules of the game are as follows:



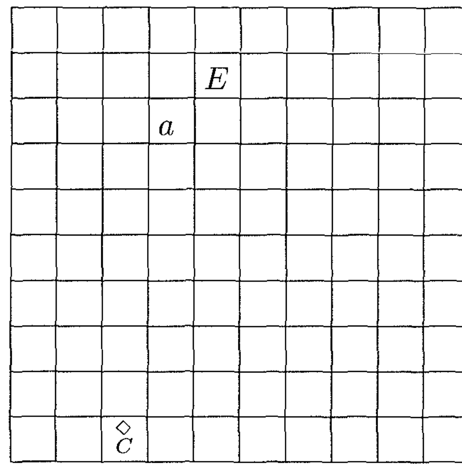


Figure IV.2: A possible losing state (left) and one of the winning states (right).

- more than one object in the world may occupy the same grid:
  - if an agent occupies the same grid as the object, it becomes the carrying agent, provided that it did not perform a *leave* action in previous step (see below); and is denoted by C; if both agents occupy the same grid as the object, one of them is randomly assigned to be the carrying agent, unless one of them is already the carrying agent, in which case the configuration does not change.
  - if the enemy occupies the same grid as an agent and if the agent is the carrying agent, the episode ends: enemy wins (Figure IV.2, left).
  - if the object, being carried by C, occupies one of grids on the target line, the episode ends: agents win (Figure IV.2, right).
- time is discrete, and actions are taken simultaneously for each time step:
  - possible moves for non-carrying agent(s) are $\{north, south, east, west\}$; agent takes one grid further per time step for each move. A move to

a non-existing grid (on the edges) causes the agent stay still.

  - possible moves for enemy are $\{north, south, east, west, jump\}$ where first four moves are as in the non-carrying agent case. The special *jump* move causes the enemy jump to the grid the agent A or C occupies in the next time step, provided that they are near by a certain Manhattan length (2 in our case).
  - possible moves for a carrying agent are $\{north, south, east, west, leave\}$ where first four moves are as in the non-carrying agent case. The special move *leave* causes the carrying agent leave the object in place. In other words, *leave* action results in a C to A transformation. In order for the agent to grasp the object again, it must leave the object's grid, and re-enter.
  - when an agent becomes a carrying agent, the object is attached to it, until it performs a *leave* action.

- each agent has the capability of sensing whether it is the agent who last touched the object; such an agent is denoted by a capital letter (C or A).
- enemy has the capability of sensing which agent last touched the object.
- the only donation of enemy is the necessary policy to attack to an agent denoted by a C or an A, and try to occupy the same cell with it. Note that whenever it occupies the same cell with a C agent, it wins the episode. Moreover, if the enemy catches an agent, it keeps track of it after then, until other agent touches the object.

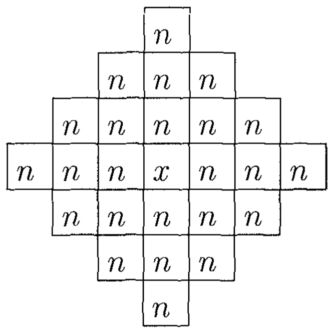


Figure IV.3: Distances sensed by agent located at center. x is the same location with agent while n indicates *near*. Any other location is sensed as far.

Apparently, there are two roles in this task, which are "with-object" and "without-object". We denote these two roles as $role_w$ and $role_{w/o}$ respectively. The definitions of these roles are:

$$role_w = \langle \mathcal{S}_{role_w}, \{north, south, east, west, leave\} \rangle \tag{IV.3}$$

$$role_{w/o} = \langle \mathcal{S}_{role_{w/o}}, \{north, south, east, west\} \rangle \tag{IV.4}$$

$\mathcal{S}_{role_w}$ and $\mathcal{S}_{role_{w/o}}$ are state descriptions defined for that role.

An agent's state description depends on sensation of the surrounding objects by their

- *distance* : an object can be far from, *near* to or at the *same* location with the sensing agent, in terms of Manhattan distance. In our settings, far limit is 3 steps (Figure IV.3).
- *direction* : an object can be at four different directions (Figure IV.4).

An agent can fully observe the environment by its *distance* and *direction* sensors.

Given the two parameters for state description, each object other than agent itself, defines a state dimension. In other words, an object can be in one of the

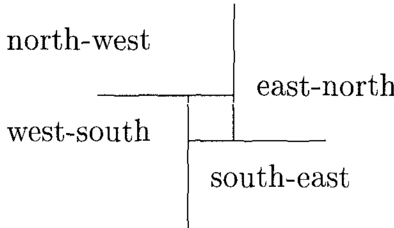


Figure IV.4: Directions sensed by agent located at the grid. An object is sensed to be in one of those four directions.

states in $\{n, f, x\} \times \{east - north, north - west, west - south, south - east\}$. So, cardinality of resulting state definition of an agent in terms of the number of objects sensed in the environment is

$$|\mathcal{S}_{obj}| = (3 \times 4)^{number\ of\ objects\ sensed} \tag{IV.5}$$

We define $\mathcal{S}_{w/o}$ in terms of 3 objects in the environment (other agent, enemy, object), and a boolean flag for sensing whether the agent itself is the last who touched the object. Thus,

$$|\mathcal{S}_{w/o}| = 2 \times (3 \times 4)^3$$

To define $\mathcal{S}_w$, since object is already carried by the agent, we ignore object and boolean flag dimensions in state definition. Moreover, since the motivation of this role is more related to avoiding the object from the enemy, we ignore the other agent dimension also. Thus,

$$|\mathcal{S}_w| = 3 \times 4$$

Using these definitions of roles, we implemented Q(λ) in order for agents to learn a coordination against the enemy. In our settings, we used $\alpha = 0.2, \gamma = 0.95,$

$\lambda = 0.9$. As the action selection strategy, *Boltzmann exploration* function is used:

$$\mathcal{P}(a_i|s) = \frac{e^{Q(s,a_i)/T}}{\sum_{j \in \mathcal{A}} e^{Q(s,a_j)/T}} \tag{IV.6}$$

where $\mathcal{P}(a_i|s)$ is the probability of choosing action a_i when agent is in state s, T is a positive parameter called the *temperature*. High temperatures cause the action selection to be nearly equiprobable, wherever as $T \to 0$, action selection becomes greedy. We used $T = 1$.

Agents are implemented to be identical. Obviously, two distinct tables are needed for each agent to hold Q-values of two different roles. If the enemy wins the episode, agents are punished by each receiving -1 as reward. In case agents win the episode, a positive reward of 10 is given to each agent. Moreover, C, the agent carrying the object, receives a reward of 1 for each action toward the target line. All other situations cause a 0 reward.

After execution of 10000 episodes, we observed that in 8998 of the episodes, agents succeeded in carrying the object to the target edge without being caught by enemy.

For $role_w$, since cardinality of state space $\mathcal{S}_w$ is small, and highest immediate reward is gained during exploration, enemy avoidance is learned by each agent quickly after start of training episodes, namely, approximately in 100 episodes (see Figure IV.5).

A typical strategy learned by the agents is shown in Figure IV.6. While agent a attacks to the object, the other agent A, occupying the same grid as enemy, tries to take the enemy away from the object. This scenario occurred so frequently and in so many different combinations, so that we believe it might be accepted

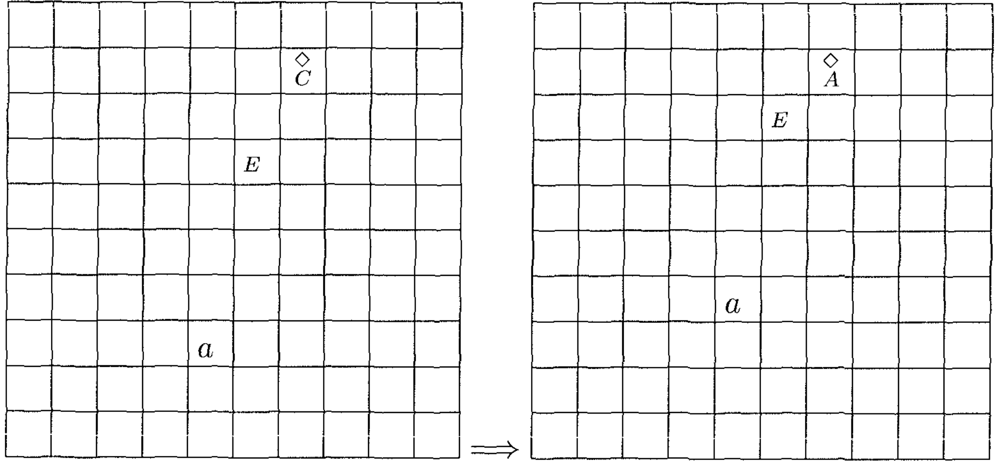


Figure IV.5: Enemy avoidance of an agent by a *leave* action.

as a form of *coordination using role learning.*

IV.3 2 vs. 1 Man Passing in Simulated Soccer Domain

A typical problem in soccer is carrying the ball behind the opponent. A tactic used in soccer games involves coordination of two (or more) players in order to move the ball behind the opponent by using short passes among themselves.

We focus on the 2 vs. 1 man passing problem on simulated soccer domain (chapter III) and discuss solution of the problem using RL techniques (chapter II). More specifically, the task is to develop necessary passing coordination by RL after many trials in the environment, so that, given a randomly placed ball initially put somewhere in front of the goal, an opponent goal-keeper and two randomly placed learning agents, the agents perform the necessary actions the coordination requires (Figure IV.7).

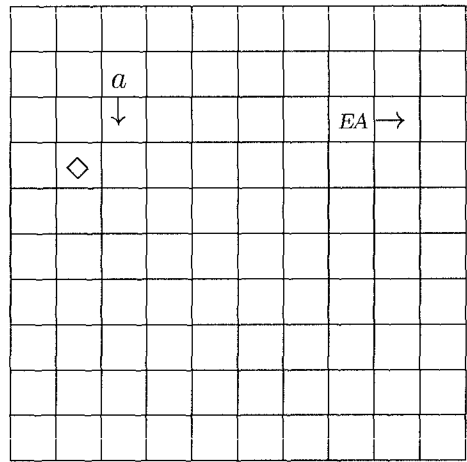


Figure IV.6: A typical coordination strategy learned.

IV.3.1 Role Based Design

In the two player passing coordination problem, two different roles can be defined: "being the passing player (with ball)", "being the player to be passed (without ball)". Let us denote them by $role_{w-b}$ and $role_{w/o-b}$.

Since almost all necessary characteristics of a real world play exists in simulated soccer domain, a number of *generalizations* have to be made in state and action descriptions.

Visual information coming from the server are coarsely discretized. This discretization has two parameters:

- *distance*: Visible area is divided into 5 horizontal slices, with respect to the agent's view axis. Self position of agent being the starting point, first 4 portions 0, 1, 2, 3 are 5 meters long each and the farthest (5th) slice has no visual depth limit. Invisible area (out of view cone) is accepted as the 6th slice.

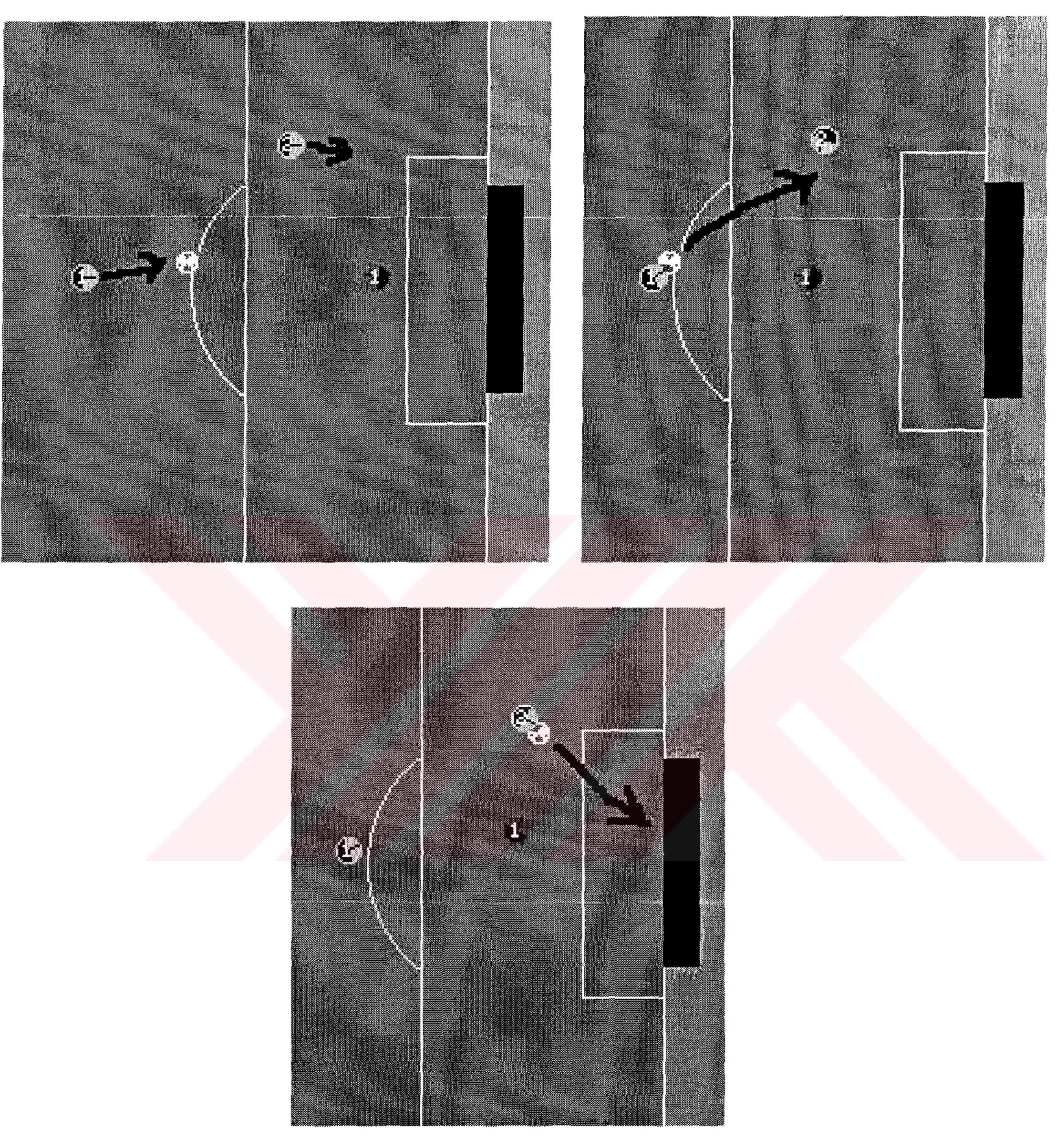


Figure IV.7: A sample passing instance.

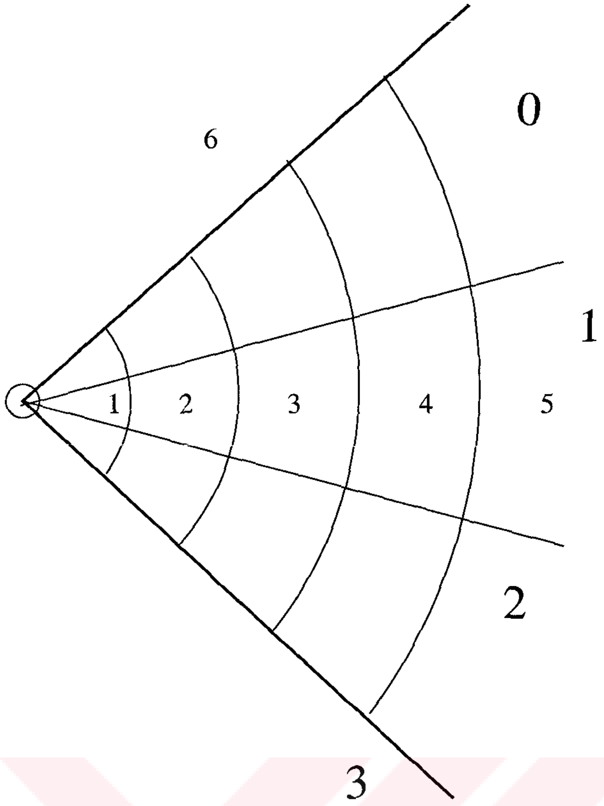


Figure IV.8: Partitioning of agent's view cone by distance (small numbers) and direction (big numbers).

- *direction*: Visible area is divided into 3 vertical slices with respect to the agent's view angle. Similarly, invisible area, forms the 4th slice.

Discretization is illustrated in Figure IV.8, with a combined view. Note that invisible slices related to distance and direction together define a single *invisible* slice and need not be taken into consideration otherwise. With this simplification, combined view provides $3 \times 5 + 1 = 16$ entries for a dimension of state space description, coarsely covering all possible locations of the seen object. State space of an agent, in general, may have 3 dimensions: *ball state, teammate state, opponent state.*

Learning a high level behavior such as coordination directly from input sensor data and primitive actions is intractable [Stone 98]. As an abstraction, we defined *meta-level actions* using primitive actions with parameters we experimentally found appropriate:

- *captureBall(n)*: turn to ball, and dash forward until ball is within kickable area, or n dashes are done.

- *runToOpen(n)*: go to a certain distance away from the ball to the goal, keeping an approximate distance from the virtual line between ball and goal. Direction of the move depends on whether the initial position of the agent is above or below the line.

- *pass()*: turn to teammate, and kick to ball forward with such a power that target of ball is experimentally found to be near the teammate.

- *shoot()*: turn to goal, and kick forward with a power such that a goal can be scored if the goal is approximately 10 meters away.

- *dribble()*: turn to goal, and kick forward with a very low power, so that the ball takes a few meters away.

Supporting our observations, the action definitions also imply the role definitions as follows:

$$role_{w-b} = \langle \mathcal{S}_{role_{w-b}}, \{pass, shoot, dribble\} \rangle \tag{IV.7}$$

$$role_{w/o-b} = \langle \mathcal{S}_{role_{w/o-b}}, \{captureBall, runToOpen\} \rangle \tag{IV.8}$$

where we define $\mathcal{S}_{role_{w-b}}$ by *opponent state* dimension only, and $\mathcal{S}_{role_{w/o-b}}$ by *ball state*, *teammate state* and *opponent state* dimensions. Thus, cardinalities of state

Figure IV.9: Initial positions of ball and goal-keeper.

spaces for the roles become

$$|\mathcal{S}_{w-b}| = 16 \tag{IV.9}$$

$$|\mathcal{S}_{w/o-b}| = 16^3 \tag{IV.10}$$

We developed two identical agents having the sensing and acting capabilities as defined above, and a goal-keeper client. Goal-keeper attacks to capture the ball whenever he detects the ball is closer than 4 meters. Additionally, we designed a coach client to determine episode starts and ends. At the beginning of each episode, coach client places the ball to a constant position as seen in Figure IV.9. Goal-keeper takes its place just in front of the goal and begins keeping track of the ball. Coach client assigns each agent a random position within 20 meters of the ball. Coach also keeps track of position of the ball in the field. The test play occurs within a $30m. \times 40m.$ rectangular test field as shown in Figure IV.10. After these initializations, the episode starts.

An episode is finished by the coach when

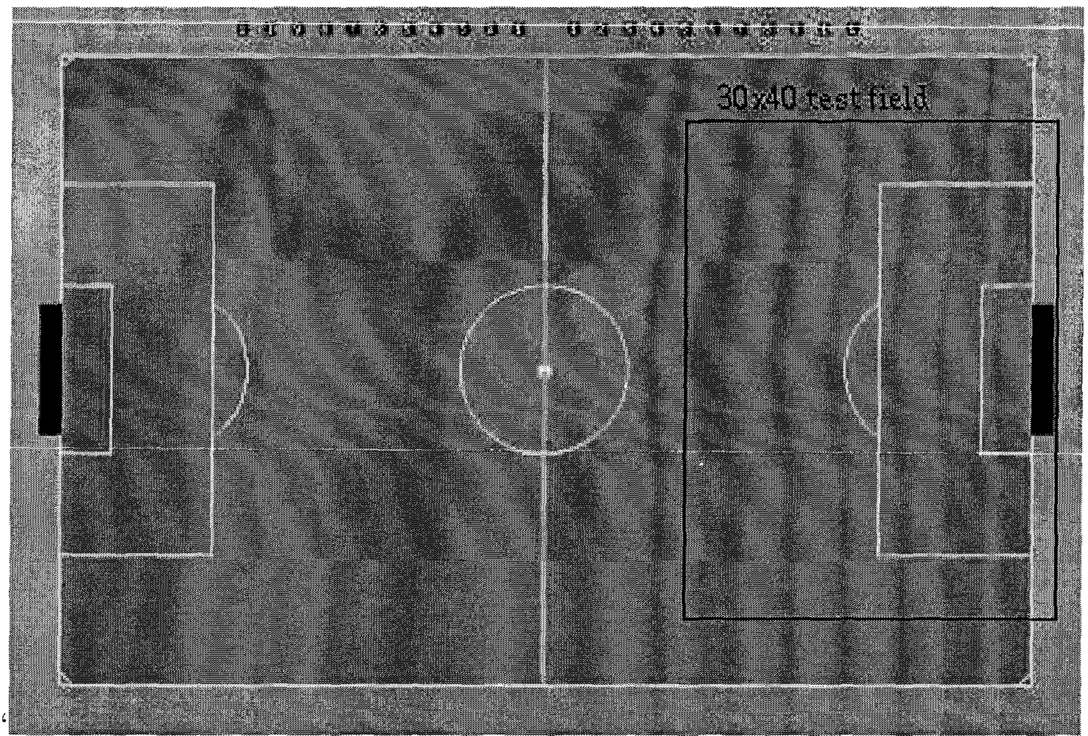


Figure IV.10: Test field.

- ball is captured by the goal-keeper,
- a goal is scored,
- ball goes out of the test field.

We used Q(λ) for the learning agents (Figure II.6), with the settings $\alpha = 0.2$, $\gamma = 0.95$ and $\lambda = 0.9$. Action selection is done via Boltzmann exploration (equation IV.6) with $T = 1$.

Each agent has an internal reward mechanism. Because of this, rewards strongly depend on current visual information of the agent. When goal is scored, agents that observe the goal receive a reward of 10. If the goal-keeper catches the ball, the observing agent receives a -1 as reward. Moreover, if an agent is very

close to ball, and the other agent is more than 10 meters away from the ball, an agent who observes this situation takes a reward of 0.1. Any other situation is a cause of 0 reward.

Note that if an agent takes the *captureBall* action, it is about to switch its role from $role_{w/o-b}$ to $role_{w-b}$, if the action succeeds. On the other way, if an agent has already captured the ball, any action it will perform will cause its role to change from $role_{w-b}$ to $role_{w/o-b}$. However, actions are not always successful, because of the non-deterministic and noisy nature of the simulated soccer domain.

Assessment of the performance on soccer domain is not clear, and there is no evident way of measurement in the literature. [Stone *et al.* 01] suggest to compare best approximation of V^{π} with test sample return values. However, the success is rated using time spent, due to the nature of the problem giving credit to longer time measurements. In our case a rating strategy, such as number of goals, does not seem to be reliable, since there are many continuous parameters (noise in sensory input, wind factor etc.) that complicate even the very simple task of scoring a goal by capturing the ball and shooting appropriately. In this study, all of these parameters are empirically assigned constant values. Instead of measurements we made observations on a number of sessions, testing whether a coordination behavior is repeatedly invoked by agents or not.

We had performed many training sessions, each of which was more than 5000 episodes long. Although the agents effectively learned to take correct action when ball is captured, if both are away from the ball, they could not get prepared for a "passing coordination" by deciding which one of them is to capture the ball, and which one is to get to open. In most of the cases, an agent had the tendency to

attack the ball at the same time with the other agent, whatever its distance to the ball is. A man passing situation rarely occurred compared to dribbling actions and direct shoots to the goal, and seemed to be achieved by chance. Moreover, when both agents attack to ball together, taking an action becomes more difficult, since visual sensors and target locations of the ball are blocked by teammate in an agent's point of view. This causes an inconsistency in actions and latency in achievement of reward.

In general, we were unable to observe an apparent learned coordination among agents through role definitions.

IV.3.2 Task Based Design

In the role based design of the 2 vs. 1 man passing problem, when both agents are dedicated to "without ball" role, a timing problem exists for switching of one of them to the other role: if one agent passes the ball *before* the other agent reaches to the correct receiving position, the pass becomes useless!

This observation suggests that the 2 vs. 1 man passing problem is more suitable to be solved by task decomposition rather than role assignment.

Two main subtasks can be pointed for an agent constructing the overall task: "prepare for a potential pass" (task 1), "take action with ball" (task 2). Note that, the first subtask necessarily results in a situation where ball is captured.

We implemented these two sequential subtasks. Actions sets are the same as in the role based case. For the first subtask, the actions are {*captureBall, runToOpen*}, for the second subtask, the actions are {*passToTeammate, shoot, dribble*}.

This time, state description with distance parameter is constructed using distances of objects to the ball. Upon seeing a flag in the field, an agent can determine its current coordinates approximately. Using this approximation, distance and direction information relative to agent position and face direction as it senses can be used to calculate the distances of any object in the field relative to any other. In our new state description, the state entry for an object is calculated by the observing agent as follows:

- if the object is self, turn to ball and use information from visual sensors to grasp ball distance
- otherwise, do
  1. calculate self coordinates using a field flag around
  2. turn to object, calculate object coordinates using result of 1 and information from visual sensors
  3. turn to ball, calculate ball coordinates using result of 1 and information from visual sensors
  4. calculate distance between ball and object using results of 2 and 3

Discretization of the distance is similar to the distance discretization defined for the role based case. This time, however, we have experimentally found that 7 slices of equal length and 1 slice as an unseen entry provides sufficient generalization and resolution over the continuous state space (Figure IV.11).

For task 1, we defined two state dimensions. One of them is over distance of the observing agent itself to the ball, and the other is over distance of the

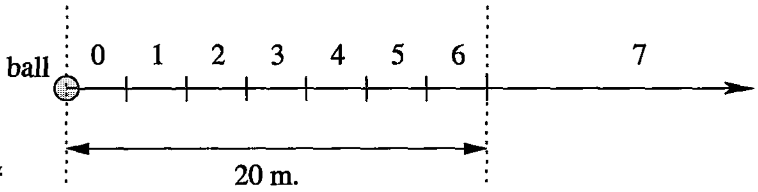


Figure IV.11: Discretization of the calculated distance of object to ball.

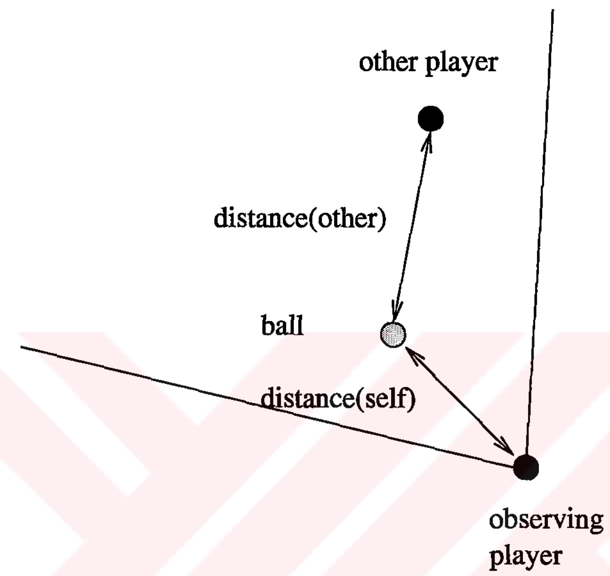


Figure IV.12: Distances to ball as observed by an agent for task 1.

teammate to the ball. A possible situation is shown in Figure IV.12. The resulting state definition can be illustrated by two dimensional table as in Figure IV.13. Thus, $|\mathcal{S}_{task_1}| = 64$.

In the experimental setup for task 1, agents learn through Q(λ). The constants α, γ, λ are set to 0.2, 0.95 and 0.9 respectively. At the beginning of each episode, the ball is set to its initial position by the coach client as shown in Figure IV.9. Again, two agents are placed at random positions within 20 meters away from the ball. Boltzmann exploration (equation IV.6) is used for action selection.

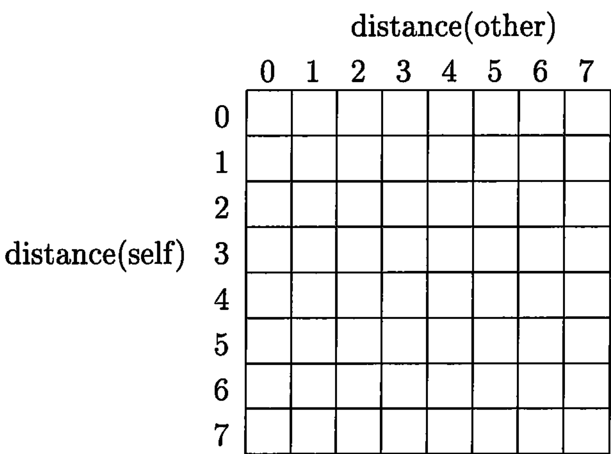


Figure IV.13: Tabular view of state definition for task 1.

At the beginning of an episode, by checking initial position of the agents, the agent near to ball is detected to be used later in reward mechanism during the episode (Figure IV.14). An episode ends when an agent captures the ball.

Rewards are internally assigned. If an agent is the nearest agent to the ball at the beginning of the episode, it is marked. Whenever it is near the ball ($distance(self) = 0$) and the other agent is acceptably far from ball so that it is ready to receive a pass ($distance(other) > 2$), then *both agents* receive a reward of 10 (see Figure IV.15, left). If both agents are near the ball ($distance(self) = distance(other) = 0$), then both agents receive a reward of -1 (see Figure IV.15, right).

After running 1000 episodes, the agents mostly succeeded in taking positions necessary for a passing coordination, getting ready for task 2. When both agents observes different state entries ($distance(self) \neq distance(other)$) at the beginning of the episode, the nearest agent captures the ball while the other gets to open (Figure IV.16).

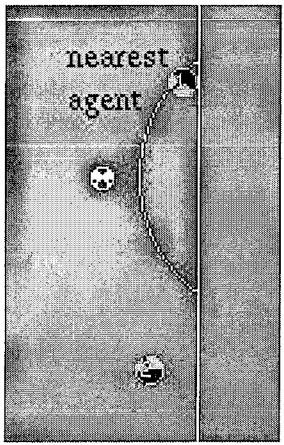
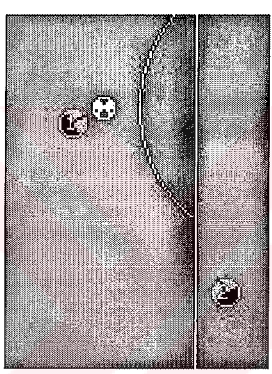


Figure IV.14: An initial placement for task 1.



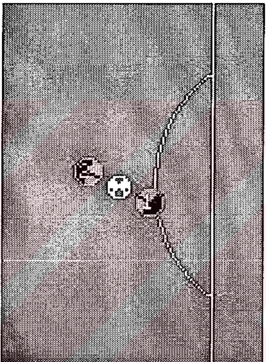


Figure IV.15: Examples of positions causing a positive (left) and a negative (right) reward.

When both of the agents are close to ball at the beginning, they both tend to get away from the ball (i.e., $distance(self), distance(other) \in \{0, 1\}$).

If both agents start the episode with state entries that are *close* to each other ($distance(self) \approx distance(other)$) but not so far from the ball, then one of the agents, as a learned policy, attacks to ball while the other gets to open (Figure IV.18).

In general, agent which is closer to the ball (independent of initial position)

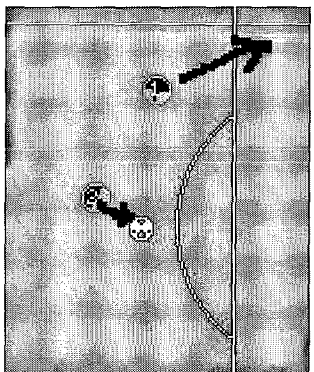


Figure IV.16: Dominant strategies when initial positions are apparently different.

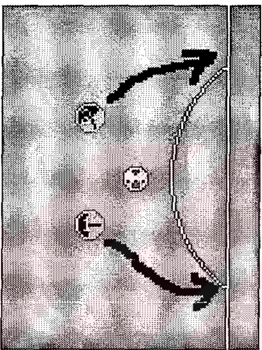


Figure IV.17: Dominant strategies when initial positions of both agents are near the ball.

seems to attack to ball even if there is a conflicting similar position with the other agent. However, an oscillating behavior occurs when agents are very near to ball, because of the repelling effect of negative reward. After the episode ends, the ball is captured, and the agents are ready for task 2.

We define the start of task 2 to be the time an agent captures the ball. When an agent observes this situation (on itself or teammate), it decides that task 1 has ended. Since task 1 and task 2 are sequential tasks, we discuss task 2 independent of task 1.

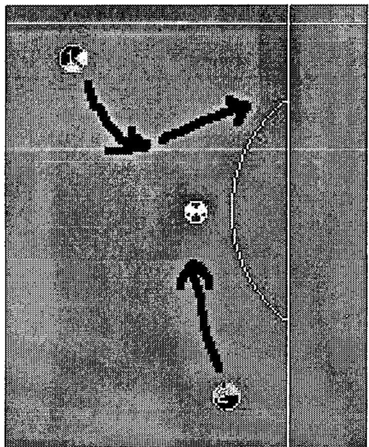


Figure IV.18: Dominant strategies when initial positions of both agents are far away from the ball.

Task 2 is a decision task for a single agent, and is similar to the "with ball" role in the role based design. For a conformity in design with task 1, new state definitions are introduced. This time, in addition to 2 distance based state descriptions, a direction based state description is introduced with the same slicing strategy as shown with large fonts in illustration in Figure IV.8. Goal-ball distance constitutes one distance based state description while opponent-ball distance is calculated for the other (Figure IV.19). The direction based state is calculated using the direction of opponent (goal-keeper). Note that, in task 2, since ball is very close to agent, all distances and directions can be directly grasped from visual sensory input.

The two distance based dimensions, like in task 1, build up a state table of 64 entries. Adding the opponent direction dimension, with cardinality 4, the state space grows up to $|\mathcal{S}_{task_2}| = 256$.

In the experimental setup for task 2, we had one learning agent, a non-moving

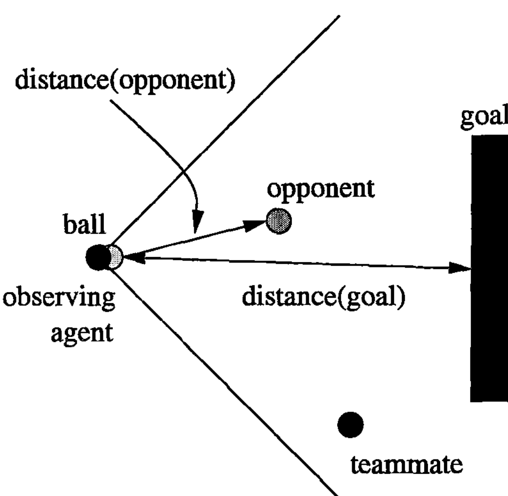


Figure IV.19: Distances to ball as observed by an agent for task 2.

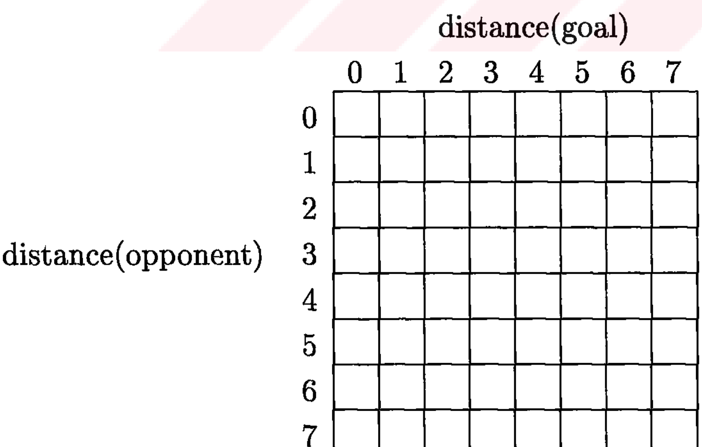


Figure IV.20: Tabular view of state definition based on distances for task 2.

Figure IV.21: A possible initial position setting for task 2.

teammate, and the goal-keeper. For the learning agent, we invoked Q-learning (Figure II.4) where $\alpha = 0.2$, $\gamma = 0.95$ and action selection is done via Boltzmann exploration strategy (equation IV.6); since the task requires action selection only, and the reward is immediate. Note that Q-learning is a special form of Q(λ) where $\lambda = 0$.

At the beginning of each episode, the coach client moves the ball at a random point 10 to 20 meters far from the goal. The learning agent is moved just behind the ball so that ball is within kickable area of the agent. The other agent is located randomly at a position within 20 meters from the ball. Initial location of the goal-keeper is the same as in Figure IV.9. A possible episode start is seen in Figure IV.21.

The agent is assigned a reward of 10 if a goal is scored. It receives -1 as reward if goal-keeper catches the ball. Any other situation causes the agent receive 1 as reward.

After more than 5000 episodes, the learning agent seemed to adopt to the reward mechanism of the task. For example, when goal-keeper is far from the ball, a dribble action was the most preferred, and if goal-keeper is at left or right slice of visible angle, a shoot or pass action was more likely to occur depending on the distance to goal.

However, since each episode is a single step update task, the learning process heavily relies on visiting each possible state by infinitely many random initializations. Thus, even around episode 5000, exploration continues.

Potentially, a recurring application of task 1 and task 2 makes up the overall task of 2 vs. 1 man passing.

CHAPTER V

CONCLUSION

If a problem is composed of sub-problems, reinforcement learning methods can provide a solution usually assuming explicit knowledge of the sub-tasks, application on each of them alone and then combining them. However, philosophy behind agent theory encourages the designer of the agent to focus on adaptivity and autonomy. A previous knowledge of tasks conflicts with autonomy.

We proposed that, as an approach to overcome this conflict, we can use role definitions in agent design instead of providing the agent with more than one learning task. This way, the learning process still remains modular and no extra knowledge is necessary for the agent other than the state and action sets for each role, and their activation requisites.

We focused on problems where the overall task is carrying an obstacle to its initial location to somewhere else with two agents avoiding an enemy. Nature of this kind of problems are suitable to be easily fragmented into two sub-tasks.

In all cases of our study, state space of an agent is coarsely coded using distance

and direction partitions with respect to sensing capabilities of that agent. As a reinforcement learning method, Watkins' Q(λ) algorithm is used, to reduce the negative effects of coarse coding and non-Markovian nature of one of the domains. Moreover, RL methods using eligibility traces usually result in faster convergence when lookup tables are used.

A hostile grid-world environment, named adversarial carry-track domain is defined. Role definitions make learning in this two-task domain easier to design. After learning, it is observed that the agents succeeded in learning this two-task problem effectively, and good coordination instances are observed.

Using the existing platform RoboCup Soccer Server, a 2 vs. 1 man passing problem is defined. Soccer environment is popular with its close-to-real characteristics, and is known to be non-Markovian. We applied multi-agent learning with roles on this problem and had poor results, in terms of coordination action sequences.

There are some critical issues to be pointed out concerning the difference in success of the method in these two domains:

- In grid world domain, states and actions are discrete while in soccer server provides continuous state and action spaces. Obviously, this makes a significant difference in learning since the definition of learning method used requires a Markovian property while the soccer domain defines a typical non-Markovian environment.

- State space in soccer domain is coarsely coded. Coarse coding is a trivial method for discretization, however, obviously means loss of information by

generalization.

- In soccer domain, construction of meta-level actions using primitive actions is a problem on its own. There is no generally accepted way of parameter selection in action design, and there are infinitely many possibilities. In robotic soccer and simulated soccer studies, much of the effort focuses on learning of performing an action only.

- In grid world domain, time is discrete, while soccer domain has almost continuously passing time counter. In another sense, an agent takes a composite action while a certain amount of time passes, and the amount of time taken is not known in advance.

- Soccer domain donates each agent with very a limited amount of information related to the environment through its visual sensors, and causes the emergence of hidden state problem. However, in grid world domain, environment is fully observable.

We observed that 2 vs. 1 man passing problem has such a complex nature that prevents coordination learning over long-term rewards as in carry-track domain, and concluded that the problem can best be solved using task decomposition rather than role decomposition. We tested the 2 vs. 1 man passing problem using task decomposition in order for a justification of our conclusion. The results were much better compared to the role based case.

In general, multi-agent reinforcement learning using roles seems to be a good alternative for multi-task problems of simple nature, but extensions must be made

for more complex problems. Function approximation methods may be effective to improve learning in continuous state and action space domains.

Although we gave a general definition of role, we experimented the idea on problems defined by two roles. Another future direction might be more solution of more complex problems in terms of number of roles in the domain.

In our study, role definitions are initially available by agents. Decision of relevant state dimensions, or elimination of unnecessary actions during training may be seen as another learning problem. Thus, defining roles dynamically by online changing of current role definitions in an adaptive manner may be valuable.

REFERENCES

[Abul 99] Abul, O., *Multi-Agent Reinforcement Learning with Function Approximation*, M.S. Thesis, Department of Computer Engineering, Middle East Technical University, Turkey, 1999.

[Abul *et al.* 00] Abul, O., Polat F., Alhajj R., *Multi-agent reinforcement learning using function approximation*, IEEE Transaction on Systems, Man and Cybernetics, Vol.30,No.4, pp.485-497, Nov.2000.

[Bellman 57] Bellman, R. E., *Dynamic Programming*, Princeton University Press, Princeton, 1957.

[Claus and Boutilier 98] Claus, C., Boutilier, C., *The dynamics of reinforcement learning in cooperative multiagent systems*, AAAI/IAAI, 746-752, 1998.

[Dietterich 00] Dietterich, T. G., *Hierarchical reinforcement learning with the MAXQ value function decomposition*, Journal of Artificial Intelligence Research, volume 13, 227-303, 2000.

[Kaelbling *et al.* 96] Kaelbling, L. P., Littman, M. L., Moore, A. W., *Reinforcement Learning: A Survey*, Journal of Artificial Intelligence Research 4, pages 237-285, 1996.

[Kitano *et al.* 97] Kitano, H., Asada, M., Kuniyoshi, Y., Noda I., Osawa, E., *RoboCup: The Robot World Cup Initiative*, Proceedings of the First International Conference on Autonomous Agents (Agents'97), ACM Press, ISBN 0-89791-877-0, pages 340-347, 1997.

[Kuter and Polat 00] Kuter, U., Polat F., *Learning better in dynamic, partially observable environments*, Proceedings of the Fourteenth Biennial European Conference on Aritificial Intelligence (ECAI), Workshop on Modeling Artificial Societies and Hybrid Organizations, Berlin, Germany, pp.50-68, 2000.

[Makar *et al.* 01] Makar, R., Mahadevan, S., Ghavamzadeh, M., *Hierarchical multiagent reinforcement learning*, Proceedings of the Fifth International Conference on Autonomous Agents, 2001.

[Mitchell 97] Mitchell, T. M., *Machine Learning*, McGraw-Hill Companies, Inc., 1997.

[Noda 95] Noda, I., *Soccer Server : a Simulator of RoboCup*, In Proceedings of AI symposium '95, pages 29-34. Japanese Society for Artificial Intelligence, December 1995.

[Noda *et al.* 99] Noda, I. et al., *Soccerserver Manual*, Ver. 5 Rev. 00 beta (for Soccerserver Ver.5.00 and later), 1999.

[Peng and Williams 96] Peng, J., Williams, R. J., *Incremental multi-step Q-learning*, Machine Learning, 22:283-290, 1996.

[Rummery and Niranjan 94] Rummery, G. A., Niranjan, M., *On-line Q-learning using connectionist systems*, Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, 1994.

[Russell and Norvig 95] Russell, S. J., Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice-Hall International, Inc., 1995.

[Schmidhuber 96] Schmidhuber, J., *A general method for multi-agent reinforcement learning in unrestricted environments*, Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems, Sandip Sen (ed.), 84-87, Stanford University, CA, 1996.

[Sen *et al.*] Sen, S., Sekaran, M., Hale, J., *Learning to coordinate without sharing information*, Proceedings of the Twelfth National Conference on Artificial Intelligence, 426–431, 1994.

[Slustowicz *et al.* 97] Salustowicz, R., Wiering, M., Schmidhuber, J., *Learning Team Strategies With Multiple Policy-Sharing Agents: A Soccer Case Study*, Technical Report, IDSIA-29-97, 1997.

[Stone 98] Stone, P., *Layered Learning in Multi-Agent Systems*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, CMU-CS-98-187, 1998.

[Stone *et al.* 01] Stone, P., Sutton, R. S., Singh, S., *Reinforcement learning for 3 vs. 2 keepaway*, RoboCup-2000: Robot Soccer World Cup IV, P. Stone, T. Balch, G. Kreatzschmarr (eds.), Springer Verlag, Berlin, 2001.

[Sutton 88] Sutton, R. S., *Learning to predict by the method of temporal differences*, Machine Learning, 3:9-44, 1988.

[Sutton and Barto 99] Sutton, R. S., Barto A. G., *Reinforcement Learning: An Introduction*, MIT Press, 1999.

[Tan 93] Tan, M., *Multi-Agent reinforcement learning: independent vs. cooperative Agents*, Proceedings of the Tenth International Conference on Machine Learning, 330-337, 1993.

[Turing 50] Turing, A. M., *Computing machinery and intelligence*, Mind, 95:433-460, 1950.

[Uther and Veloso 97] Uther, W., and Veloso, M. *Generalizing Adversarial Reinforcement Learning*, AAAI Fall Symposium on Model Directed Autonomous Systems, 1997.

[Watkins 89] Watkins, C. J. C. H., *Learning from Delayed Rewards*, Ph.D. thesis, Cambridge University, 1989.

[Watkins 92] Watkins, C. J. C. H., Dayan, P., *Q-learning*, Machine Learning, 8:279-292, 1992.

[Weiß 93] Weiß, G., *Learning to coordinate actions in multi-agent systems*, Proceedings of the International Joint Conference on Artificial Intelligence, 311-316, 1993.