INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
MONTERREY CAMPUS

TECNOLÓGICO DE MONTERREY

GENERATION OF MOTION AND DECISION-MAKING POLICIES APPLYING MULTIAGENT REINFORCEMENT LEARNING IN SIMULATED ROBOTIC SOCCER

THESIS
SUBMITTED TO THE GRADUATE PROGRAM IN DIVISION OF MECHATRONICS AND INFORMATION TECHNOLOGIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN INTELLIGENT SYSTEMS

BY:
DAVID ALEJANDRO GARCIA ORTEGA

DECEMBER 2010
GENERATION OF MOTION AND DECISION-MAKING POLICIES APPLYING MULTIAGENT
REINFORCEMENT LEARNING IN SIMULATED ROBOTIC SOCCER

THESIS
SUBMITTED TO THE GRADUATE PROGRAM IN
DIVISION OF MECHATRONICS AND INFORMATION
TECHNOLOGIES IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE
IN
INTELLIGENT SYSTEMS

BY:
DAVID ALEJANDRO GARCIA ORTEGA

DECEMBER 2010
Generation of Motion and Decision-Making Policies Applying Multiagent Reinforcement Learning in Simulated Robotic Soccer

by

David Alejandro García Ortega

Thesis
Submitted to the Graduate Program in Division of Mechatronics and Information Technologies in partial fulfillment of the requirements for the degree of

Master of Science in

Intelligent Systems

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

December 2010
Thesis committee members recommend the present thesis of David Alejandro García Ortega to be accepted in partial fulfillment of the requirements for the degree of Master of Science, in:

Intelligent Systems

Thesis committee:

Ph.D. Leonardo Hernando Luna
Thesis advisor

Ph.D. Ramón Brena Pinero
Examiner

Ph.D. Rogelio Soto Rodríguez
Examiner

Ph.D. Ramón Brena Pinero
Graduate Program Director

December 2010
Generation of Motion and Decision-Making Policies Applying Multiagent Reinforcement Learning in Simulated Robotic Soccer

David Alejandro García Ortega, M.Sc.
Instituto Tecnológico y de Estudios Superiores de Monterrey, 2010

Thesis advisor: Ph.D. Leonardo Garrido Luna

When designing intelligent agent systems, it is impossible to implement all the potential situations an agent may encounter and specify an optimally behavior in advance. An intelligent agent must be capable of using its past experience to learn how its actions affect the environment in which it is situated. Reinforcement learning (RL) is a machine learning technique where an agent learns from its own environment while the agent is in it. With this technique, goal directed agents learn what to map states to actions in order to maximize their utility. RL techniques have addressed the problem of learning for a single agent acting in a stationary environment and for many agents acting in a dynamic environment. Multiagent environments are inherently dynamic since the agents may change the environment simultaneously, as all of them are learning and adapting at the same time.

This thesis aims to implement both single agent and multiagent RL approaches in the international “RobotStadium” simulated robotic soccer competition in order to generate motion and decision-making policies. The “RobotStadium” competition is a simulated soccer league based on the “Nao” robot (from Aldebaran robotics) and the “Webots” simulator (from Cyberbotics), using the same rules of the Standard Platform League (SPL) of the international “RoboCup” competition. The “RobotStadium” platform offers an excellent experimental framework for multiagent systems under a realistic, uncertain and highly dynamic environment.

The classical design of motion and decision-making policies is a laborious effort, requiring many hours of experimental work, designing, testing, and redesigning the control program until the desired behavior is achieved. In this thesis, a RL approach is implemented for automatic generation of motion and decision-making policies through the experience of the agent in its environment. However, automatic generation of motion and decision-making policies is a difficult task mainly due to the large and continuous states spaces that the “RobotStadium” environment represents. In problems with large state spaces, the agent can generalize from...
limited experience by grouping together similar states. This process is called generalization and provides the agent the ability to learn without the necessity of exploring all the possible states of the environment.

Specifically, this thesis uses the Connectionist Q-Learning Framework (CQLF) an open source Java library for developing learning systems. CQLF is based on a backpropagation neural network that is trained using the CQL algorithm by Sutton [Sutton, 1987]. This framework uses a neural network instead of the Q-Table of classical Q-Learning algorithm. The neural network allows state generalization in order to avoid the continuous large state space problem.

First, the CQLF is implemented to generate motion policies, where the goal is to learn how to move a set of servos in order to achieve a motion behavior. This problem is first addressed in a single agent learning approach were the agent learns to move all the servos in order to generate the motion and then in a multiagent learning approach were every servo is a learning agent and the resulting policies conform the overall motion behavior. According to experimental results, it was concluded that multiagent approach together with the CQLF is a good approach to tackle the problem of generating motion policies. In experiments the multiagent approach overcame the single agent approach. Also, it was noticed that the multiagent approach is a more suitable alternative for the problem given its nature of parallelism. With the multiagent approach agents can be modeled as a group of simple interacting agents with single state spaces as well as single action spaces.

Then, the framework is implemented to generate decision-making policies with the objective of generate single and multiagent strategies. The decision-making problem involves the selection of high-level actions in contrast with the motion problem where the selected actions are low level. In the decision-making problem the selected actions correspond to motion behaviors. During experiments, it was found that at least the learned policies are as good as custom decision-making policies. Moreover learned policies also have the advantage of being flexible to noisy perceptions, because the agent learns from experience in the noisy environment. Custom policies on the other hand are rigid and prone to failure by noise.

In this thesis, it is showed that it is possible to automatically generate motion and decision-making policies in continuous and dynamic environments through RL as long as there is a method of generalization. Furthermore, it is showed that learning motion policies can be simplified by a multiagent approach, in which each servo or actuator is an agent learning at same time as the others. Moreover it is described how generated decision-making policies are capable to work in noisy environments and perform very well in comparison with custom policies. Finally, is important to stress that these results are part of our ongoing research for generating even more complex motion policies and with more precise control. On the other hand, regarding the high-level decision-making problem, there are confidence that this approach can generate even more complex high-level strategies with the involvement of a greater number of agents.
Contents

Abstract iv

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1

1.1 Motivation 2
1.2 Problem Definition 2
1.3 Objectives 3
1.4 Hypotheses 4
1.5 Research Questions 4
1.6 Organization 5

Chapter 2 The RobotStadium Simulated Robot Soccer Competition 6

2.1 Motivation for RobotStadium 6
2.2 Rules 7
2.3 Webots Simulator 8
2.4 Simulated 3D Environment 8
2.5 Nao Robot and Simulated Nao Robot 9
2.6 Programming 11
2.7 Summary 13

Chapter 3 Solution method: Reinforcement learning plus generalization 14

3.1 Reinforcement Learning 14
3.2 Q-Learning 16
3.3 Eligibility Traces 17
3.4 Generalization and Function Approximation 17
3.5 Connectionist Q-Learning Framework 18
3.6 Related Work 21
3.7 Summary 24

Chapter 4 Motion Policies Experiments 25

4.1 The Problem of Generating Motions Policies 25
4.2 Empirical Scenarios 26
4.3 "Kicking the Ball" Experimental Scenario ........................................ 28
   4.3.1 Design of experiment ...................................................... 28
   4.3.2 Single agent approach ................................................... 30
   4.3.3 Multiagent approach ..................................................... 32
   4.3.4 Results and discussion .................................................. 35
4.4 "Jumping to Block the Ball" Experimental Scenario ................................ 36
   4.4.1 Design of experiment .................................................... 36
   4.4.2 Single agent scenario .................................................... 38
   4.4.3 Multiagent scenario ..................................................... 39
   4.4.4 Results and discussion .................................................. 41
4.5 Summary ................................................................................. 43

Chapter 5 Decision-Making Policies Experiments ........................................ 44
5.1 Decision Making Problem ............................................................ 44
5.2 Empirical Scenarios .................................................................... 46
5.3 Chasing the Ball ........................................................................ 46
   5.3.1 Single Agent Approach .................................................... 47
   5.3.2 Design of Experiment ..................................................... 47
   5.3.3 Results ............................................................................ 49
5.4 Blocking Penalties ...................................................................... 50
   5.4.1 Single Agent Approach .................................................... 51
   5.4.2 Design of Experiment ..................................................... 52
   5.4.3 Results ............................................................................ 54
5.5 Shooting Penalties ...................................................................... 56
   5.5.1 Single Agent Approach .................................................... 57
   5.5.2 Multiagent Approach ....................................................... 57
   5.5.3 Design of Experiment ..................................................... 58
   5.5.4 Results ............................................................................ 59
5.6 Coordination for Chasing the Ball .................................................. 61
   5.6.1 Empirical Scenario .......................................................... 63
   5.6.2 Design of Experiment ..................................................... 64
   5.6.3 Results ............................................................................ 65
5.7 Summary ................................................................................. 67

Chapter 6 Conclusions ......................................................................... 69
6.1 Contributions ............................................................................. 69
6.2 Future Work .............................................................................. 70

Appendix A Example of Java code implementing the CQLF .............................. 72

Bibliography ..................................................................................... 75
List of Tables

2.1 Technical specifications of Nao robot ........................................ 10
2.2 Servo Motors of NaoV3R Model ............................................... 11
2.3 Example of a motion table ....................................................... 13

4.1 CQLF configuration for the single agent approach for “kicking the ball” experiment ....................................................... 31
4.2 CQLF configuration for the multiagent approach for “kicking the ball” experiment ....................................................... 33
4.3 CQLF configuration for the single agent approach for “jumping to block the ball” experiment ....................................................... 38
4.4 CQLF configuration for the multiagent approach for “jumping to block the ball” experiment ....................................................... 40

5.1 CQLF configuration for the “chasing the ball” experiment ............... 49
5.2 CQLF configuration for the “blocking penalties” experiment ............... 54
5.3 CQLF configuration for the single agent scenario in “shooting penalties” experiment ....................................................... 60
5.4 CQLF configuration for the multiagent scenario in “shooting penalties” experiment ....................................................... 60
5.5 CQLF configuration for the “coordinated chasing the ball” experiment ....................................................... 65
**List of Figures**

2.1 Webots simulator running RobotStadium contest ........................................ 8
2.2 RobotStadium field dimensions and lines (mm) ........................................... 9
2.3 Main features of Nao robot ................................................................. 10

3.1 The agent-environment interaction in RL ............................................... 15
3.2 Accumulative eligibility traces .............................................................. 17
3.3 Connectionist Q-Learning method .......................................................... 18
3.4 Connectionist Q-Learning Framework ..................................................... 19
3.5 Sigmoid function ..................................................................................... 20

4.1 Single agent empirical scenario for learning motion problem .................... 27
4.2 Multiagent empirical scenario for learning motion problem ....................... 28
4.3 Environment for kicking the ball experiment ........................................... 29
4.4 Single agent approach for learning motion problem .................................. 30
4.5 Variation to single agent approach for learning motion problem ............... 31
4.6 Results of single agent approach for “kicking the ball” experiment .......... 32
4.7 Multiagent approach for learning motion problem .................................... 33
4.8 First variation to multiagent approach for learning motion problem ........ 34
4.9 Second variation to multiagent approach for learning motion problem .... 34
4.10 Results of multiagent approach for “kicking the ball” experiment ............ 35
4.11 Comparison between single and multiagent approach for “kicking the ball” experiment ................................................................. 36
4.12 Environment for jumping to block the ball experiment ............................ 37
4.13 Results of single agent approach for “jumping to block the ball” experiment 39
4.14 Results of multiagent approach for “jumping to block the ball” experiment 40
4.15 Comparison between single and multiagent approach for “jumping to block the ball” experiment ................................................................. 41

5.1 Empirical scenario for chasing the ball experiment ................................... 47
5.2 RobotStadium environment for chasing the ball experiment .................... 48
5.3 Learning curve for “chasing the ball” experiment ...................................... 50
5.4 Comparison between learned and custom policies for “chasing the ball” experiment ................................................................. 51
5.5 Empirical scenario for blocking penalties experiment .............................. 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>RobotStadium environment for blocking penalties experiment.</td>
<td>53</td>
</tr>
<tr>
<td>5.7</td>
<td>Learning curve for “blocking penalties” experiment.</td>
<td>54</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison between learned and custom policies for “chasing the ball” experi-</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>ment.</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>Single agent empirical scenario for shooting penalties experiment.</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>Multiagent empirical scenario for shooting penalties experiment.</td>
<td>58</td>
</tr>
<tr>
<td>5.11</td>
<td>RobotStadium environment for shooting penalties experiment.</td>
<td>58</td>
</tr>
<tr>
<td>5.12</td>
<td>Learning curves for single and multiagent scenarios in “shooting penalties”</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>experiment.</td>
<td></td>
</tr>
<tr>
<td>5.13</td>
<td>Comparison between learned and custom policies for “shooting penalties” experi-</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>ment.</td>
<td></td>
</tr>
<tr>
<td>5.14</td>
<td>Empirical scenario for “coordinated chasing the ball” experiment.</td>
<td>63</td>
</tr>
<tr>
<td>5.15</td>
<td>RobotStadium environment for “coordinated chasing the ball” experiment.</td>
<td>64</td>
</tr>
<tr>
<td>5.16</td>
<td>Learning curve for “coordinated chasing the ball” experiment.</td>
<td>66</td>
</tr>
<tr>
<td>5.17</td>
<td>Comparison between learned and custom policies for “coordinated chasing the</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>ball” experiment.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

According to [Russell and Norvig, 1995], Artificial Intelligence (AI) studies and designs intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions which maximize its utility. Exist many definitions of AI, however its main goal is to create an artificial man which exhibit traits as reasoning, knowledge, planning, learning, communication and perception.

Machine learning is one of the most ambitious goals of AI. One of the main features of an intelligent agent is that it can adapt to the unknown environment. Learning from interaction with the environment is a foundational idea underlying nearly all theories of learning and intelligence. Reinforcement Learning (RL) is a learning type that allows that an agent acquires knowledge through the continuous interaction with its environment.

This thesis is part of the Intelligent Autonomous Agents research group from the Computer Science Department at Tecnológico de Monterrey, Campus Monterrey University, which objective is to develop innovative technology oriented towards distributed knowledge handling by researching about agent technologies and multiagent systems.

This thesis implements Reinforcement Learning (RL) and Multiagent Reinforcement Learning (MRL) in the RobotStadium simulated robot soccer competition in order to generate motion and decision making policies. Specifically, this thesis implements the connectionist reinforcement learning framework which uses a backpropagation neural network as a method of generalization in order to avoid the large state space problem. First, the framework is implemented to generate motion policies, where the goal is to learn how to move a group of motors in order to achieve a motion behavior like a kick. This problem is first addressed in a single agent learning approach were the agent learns to move all the motors in order to generate the motion and then in a multiagent learning approach were every motor is a learning agent and the resulting policies conform the motion behavior. Then the framework is implemented to generate decision making policies with the objective of generate single agent strategies, like the strategy for shooting penalties. The decision-making problem involves high level action selection in contrast with the motion problem where the selected actions are low level. Actually, in the decision making problem the selected actions correspond to motion behaviors for example the turn or kick motions. Finally, the framework is implemented to generate decision making policies for multiple agent strategies like the kick off strategy which requires agent cooperation and coordination.
1.1 Motivation

RoboCup is an international initiative that fosters research and education in robotics and artificial intelligence, on multi-robot systems in particular, through competitions of RoboCup Soccer, RoboCup Rescue, RoboCup Home and RoboCup Junior. The ultimate goal of the RoboCup soccer project is by 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer. RoboCup chose to use soccer game as a central topic of research, aiming at innovations to be applied for socially significant problems and industries.

RoboCup currently includes a number of different robot soccer leagues that focus on different research challenges. One of these leagues is the RoboCup Standard Platform League (SPL), which all teams compete with identical robots, the Nao robot. In addition to the SPL, RobotStadium is a parallel simulated league, where contestants can battle simulated Naos against each other. RobotStadium league is based on the RoboCup SPL and uses the Webots simulator by Cyberbotics.

RobotStadium offers an excellent multi agent system platform under a realistic, uncertain and highly dynamic environment. It provides an interesting domain to do research in artificial intelligence and multiagent systems. Participating teams do not need to worry about hardware, they can focus on software development corresponding to the controller of the robot. However, the development of controllers for RobotStadium soccer competition offers a great variety of problems from low level capabilities like basic agent motion to high level capabilities like multiagent cooperation.

Intelligent Autonomous Agents research group is interested in coordination and cooperation among autonomous agents in the RoboCup domain. Research team already has an extensive experience in the simulation leagues of RoboCup, it has presented some research works related to the RoboCup 3D simulation league. A fuzzy bayesian approach for decision making in RoboCup Simulation 3D was presented in the RoboCup Symposium [Bustamante et al., 2006a]. Later, in [Bustamante et al., 2006b] as published a comparison between fuzzy bayesian classifiers and gaussian bayes classifiers. Afterwards, a hybrid Monte-Carlo localization with Kalman filter sensor fusion approach was used for diminishing the effect of noise and uncertainty in the agent self localization process, and was published in [Bustamante and Garrido, 2007].

Moreover, RobotStadium offers the possibility to work with the real Nao robots in the RoboCup SPL reusing the controller developed in RobotStadium simulated competition due to the high compatibility implemented in the RobotStadium platform.

1.2 Problem Definition

The main objective in a simulated robotic soccer environment consists in design intelligent agents that jointly play soccer. The main problem is that it is not possible to implement all the potential situations agents may encounter and specify an optimally behavior in advance. Agents have to learn from, and adapt to their environment, especially in a highly dynamic multiagent environment.
One of the main problems in the development of an intelligent agent that plays soccer is the design of motion behaviors. These motions behaviors allow the robot to perform basic movements to play soccer like walk, turn, shoot, stand up, etc. Implementing these motions is a hard task mainly due to the large and continuous state space and the parallelism in the movement of the motors.

Motion behaviors imply the movement of servo motors corresponding to the robot joints. The state space consists of the positions of servo motors. The position of a servo motor is a continuous angle value delimited by a range (minPosition, maxPosition). Every motion behavior requires the movement of various servo motors, thus the state space becomes very large. Moreover, a motion behavior imply the movement of the servo motors at the same time, thus there is parallelism that makes harder to manually generate motions.

Other important problem is the implementation of rules for decision-making. This is a hard task because it is not possible to program the agents with all possible actions for all possible states. The state space is a huge table of states * actions which needs huge memory for storing and a huge amount of time to converge for classical RL techniques. This huge table gives a problem that could become intractable.

Finally, it is necessary to achieve coordination and cooperation between agents to jointly play soccer and achieve team goals. The implementation of rules for decision-making in single agents is hard but it is harder in multi agent systems in which the agent learning depends on the actions of the others.

1.3 Objectives

The main objective of this thesis is to implement RL in the international “RobotStadium” simulated robotic soccer competition in order to generate motion and decision-making policies. This thesis aims to automatically generate motion and decision-making policies through the experience of the agent in its environment. With this approach the objective is that the designer specifies the desired behavior of the agent trough rewards, rather than the control program that produces the desired behavior.

From this overall objective can be described the following specific objectives:

- Implement a RL technique for automatic generation of motion and decision making policies.
- Implement a generalization technique together with RL in order to being capable to learn policies involving continuous state and action spaces.
- Implement the Connectionist Q-Learning algorithm through the Connectionist Q-Learning Framework in order to learn policies by RL plus generalization.
- Implement and compare two empirical scenarios for learning motion and decision-making policies: single agent approach and multiagent approach.
- Describe a framework for addressing problems of generating motion and decision-making policies using single and multiagent approaches.
• Experiment with simple but common motion and decision-making tasks in RobotStadium.

• Establish the foundation for future work to address problems involving more complex policies (i.e. harder to learn) as well as problems involving a larger number of interacting agents.

Finally it is expected that the results of this thesis provide new skills to the Borrregos-Nao team in the RobotStadium competition.

1.4 Hypotheses

According to the objectives outlined in the previous section, this thesis aims to test the following hypotheses:

• Through the implementation of RL techniques is possible to automatically generate motion and decision-making policies.

• Through generalization is possible to apply RL techniques to problems with large and continuous state and action spaces.

• Complex environments can be modeled as multiagent learning systems by a group of simple interacting agents with single state spaces as well as single action spaces.

• Communication among individual agents is not necessary in order to learn together a policy.

• Multiagent approach is a more suitable alternative for generating motion policies given its nature of parallelism.

• The single agent approach is able to simulate the parallelism required in motion policies.

• The Connectionist Q-Learning algorithm is able to work in noisy environments.

• Learned decision-making policies can be at least as good as custom policies.

1.5 Research Questions

To achieve the proposed objectives as well as to verify the hypotheses, the following research questions are addressed in this thesis:

• How can RL automatically generate motion and decision-making policies?

• What are the advantages and disadvantages of the single and multiagent approaches for implementing reinforcement learning to problems of generating motion and decision-making policies?
• Which approach, single or multiagent, is more effective to solve problems of generating motion and decision-making policies?

• Why to use generalization instead discretization to apply RL techniques in the Robot-Stadium environment?

• How effective are automatic generated decision-making policies compared to custom hand-crafted policies?

1.6 Organization

The thesis is divided in six chapters. The current chapter presents an introduction and shows the motivation, problem definition, objectives and hypotheses. Chapter 2 describes the task environment. Chapter 3 contains the theoretical framework about Reinforcement Learning and also contains some related work. Chapter 4 is about the implementation of the Connectionist Q Learning framework to the learning motion policies problem. Chapter 5 is about the implementation of the Connectionist Q Learning framework to the learning decision making policies problem. Finally Chapter 6 shows obtained conclusions, contributions and future work.
Chapter 2

The RobotStadium Simulated Robot Soccer Competition

RobotStadium [RobotStadium, 2010] is an on-line simulated robotic soccer competition based on the Nao robot and the Webots simulator. RobotStadium simulation uses the rules of RoboCup Standard Platform League (SPL), and actually, is a very accurate simulation of the RoboCup SPL. In the on-line RobotStadium competition a round is run every business day. During the round, every team plays at least one match against another team. If \( N \) teams are participating, \( N - 1 \) matches will be performed during one round. The first match will oppose the bottom team \( \#N \) to the one just above it \( \#N - 1 \). If controller \( \#N \) wins, then the positions of these two teams are swapped. The second match will oppose the new \( \#N - 1 \) team to team \( \#N - 2 \). If team \( \#N - 1 \) wins, they will swap their position. The same procedure is repeated with the new \( \#N - 2 \) team and team \( \#N - 3 \) and so on until team \( \#2 \) plays against team \( \#1 \). This process is similar to a bubble sort algorithm.

The Hall of Fame is a table displayed on the web site of the contest, showing the current ranking of all the teams in competition. New teams are added at the bottom of the Hall of Fame. Due to round mechanics, a team can climb to the top of hall of fame in just one round. By contrast a team just can lose one position in one round.

A match on the contest server is made of two halves of 10 minutes each. During the half-time break, the team colors and defended goals are swapped. The match is automatically supervised by a Supervisor program which counts the goals scored by each team and places the ball and the players at initial positions after a goal was scored. In case of a tied game (equal scores) at the end of the second half, a penalty kick shoot-out will take place. The penalty kick shoot-out does always concludes with a winner and a looser.

2.1 Motivation for RobotStadium

Since 2008, RobotStadium has been a passionating on-line competition open to everyone and free of charge. RobotStadium warmly welcome RoboCup teams and robotics students from all around the world and is one of the main testbeds for the the RoboCup initiative in terms of AI techniques like reasoning, planning, learning, coordination, communication and opponent modeling.

Participating teams just need to develop a controller and upload it to the competition server. Then a round will be executed and results can be viewed on the RobotStadium website, where also videos for all games are available. Therefore developers can test their
algorithms almost immediately and retrieve feedback to improve them and test again in the next round.

RobotStadium is very accessible competition, it has examples of controllers for different programming languages. Moreover RobotStadium has a fully featured website that allow participants to communicate between them and also with the organizers. It contains a simple and complete documentation and also provides support trough a forum.

Starting in 2009, RobotStadium competition awarded the first three places. The competitor ranked #1 in the hall of fame at the termination of the competition received a cash prize of 1000 Swiss Francs and also a Webots PRO box (DVD, Documentation and License). Competitor ranked #2 received a Webots EDU box and competitor ranked #3 received a Webots EDU license. In 2010 RobotStadium competition awarded only the first place with a Webots PRO box.

In 2009 the Borregos-Nao team from Tecnológico de Monterrey, Campus Monterrey finished in fourth place in the RobotStadium competition and in 2010 climbed to the second position.

2.2 Rules

In order to participate in RobotStadium, developers must be at least 18 years old and have technical programming education, experience and knowledge to program a simulated autonomous robot. It is requested that the strategies developed by the competing teams respect the standard soccer rules (i.e., do not pick up the ball with the hands and rush to the goal). Moreover, each robot should have a fair play behavior. Furthermore, any robot controller submitted to the contest, should respect the following rules:

- No graphical user interface (GUI). not try to read any input from the keyboard or any other input device.
- No networking.
- No sound.
- No file I/O outside the controller's directory.
- Do not attempt to access or modify the process of your RobotStadium opponent or any other process of the contest host.
- Do not attempt to affect the integrity or the security of the computer that hosts the contest.

The RobotStadium simulation is inspired from the official RoboCup rules for the Nao SPL. However it is designed to run without human intervention in order to allow developers to run fully automatic and unattended matches that allow them to optimize their strategy.
2.3 Webots Simulator

The Webots mobile robotics simulation software developed by [Cyberbotics, 2010], provides a rapid prototyping environment for modeling, programming and simulating mobile robots. With Webots it is possible to design complex robotic setups, with one or several robots, in a shared environment. There are many simulated sensors and actuators available to equip each robot. Moreover, the robot behavior can be tested in physically realistic worlds.

Webots simulator can model and simulate any mobile robot either wheeled, legged or flying. Moreover robot controllers can be programmed in C, C++, Java and third party software like URBI and Matlab. Webots uses OpenGL for robots and world 3D modeling and also uses ODE (Open Dynamics Engine) library for accurate physics simulation. Webots also allows to simulate multiagent systems with communication facilities. Finally it contains documentation and many examples with controller source code.

There are two versions of the Webots simulator, the Webots Pro for researchers and the Webots Edu for educational proposes. However, the Demo version of Webots allow participating teams to work with the RobotStadium simulation. Figure 2.1 shows the Demo version of the Webots simulator running the RobotStadium contest environment.

2.4 Simulated 3D Environment

In Webots distribution, the RobotStadium contest environment is provided in the file “robotstadium.wbt”. This is a text file which contains the description of every object within the environment (soccer field, walls, goals, robots, ball, lights and supervisor). The RobotStadium environment file also contains the controller information for robots and supervisor. The environment file can be edited with a text editor or with the “Scene tree” editor of Webots.
The RobotStadium environment is inspired from the official RoboCup SPL, thus the simulated soccer field is very accurate to the real field. Field is built on a total carpet area of length 7.4 m and width 5.4 m, however the soccer field area is of length 6 m and with 4 m. All lines on the soccer field (side lines, end lines, halfway line, center circle, corner arcs, and the lines surrounding the penalty areas) are 50 mm in width. The center circle has an outside diameter of 1250 mm. Figure 2.2 shows the field dimensions. Goals have a width of 1.4 m and a height of 0.8 m with a depth of .4 m.

Every simulated object has properties like color, mass, dimension, translation and rotation that makes the simulated environment really accurate. In this way, teams can develop controllers that works similarly in the real robots as well as the simulated robots.

2.5 Nao Robot and Simulated Nao Robot

In RobotStadium environment simulated robots correspond to the Nao robot which is an autonomous medium-sized humanoid robot developed by the French company Aldebaran Robotics [Aldebaran-Robotics, 2010]. Nao robot was designed for entertainment purposes, is able to interact with its owner with behaviors and functionalities only limited by the imagination. Figure 2.3 shows a picture of a real Nao robot and its main features and table 2.1 shows the technical specifications of Nao robot.

RobotStadium uses a Webots model of the Aldebaran “NaoV3 RoboCup Edition” to simulate the Nao robot [RobotStadium, 2010]. This model is called “NaoV3R”, where the R means for RoboCup. In Webots the model is represented by a proto file called “NaoV3R.proto”. The robot model contains 22 servos (Head (2) + Shoulder (4) + Elbow (4) + Hip (6) + Knee (2) + Ankle (4)), a camera web, an accelerometer, a gyro, 4 ultrasounds and 4 Force Sensitive Resistors (FSRs) in each foot. It also contains radio emitter and receiver and many leds (Eyes, chest, ears, foot).
Table 2.1: Technical specifications of Nao robot.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>58 cm</td>
</tr>
<tr>
<td>Weight</td>
<td>4.3 kg</td>
</tr>
<tr>
<td>DOF</td>
<td>21 - 25</td>
</tr>
<tr>
<td>CPU</td>
<td>x86 AMD GEODE 500 MHz</td>
</tr>
<tr>
<td>Memory</td>
<td>256 MB SDRAM / 2 GB flash memory</td>
</tr>
<tr>
<td>Built-in OS</td>
<td>Embedded Linux (32 bit x86 ELF)</td>
</tr>
<tr>
<td>Compatible OS</td>
<td>Windows, MacOS, Linux</td>
</tr>
<tr>
<td>Programming languages</td>
<td>C++, C, Python, Urbi</td>
</tr>
<tr>
<td>Vision</td>
<td>CMOS 640 x 480 cameras[4]</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Ethernet and Wi-Fi</td>
</tr>
</tbody>
</table>

Figure 2.3: Main features of Nao robot.
<table>
<thead>
<tr>
<th>Name</th>
<th>minPosition</th>
<th>maxPosition</th>
<th>maxVelocity</th>
<th>maxForce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(rad)</td>
<td>(rad)</td>
<td>(rad/s)</td>
<td>(Nm)</td>
</tr>
<tr>
<td>LKneePitch</td>
<td>0.0</td>
<td>2.2689</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>RKneePitch</td>
<td>0.0</td>
<td>2.2689</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>LAnkleRoll</td>
<td>-0.7854</td>
<td>0.7854</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>RAnkleRoll</td>
<td>-0.7854</td>
<td>0.7854</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>HeadYaw</td>
<td>-1.5708</td>
<td>1.5708</td>
<td>8.27</td>
<td>2.27</td>
</tr>
<tr>
<td>HeadPitch</td>
<td>-0.7854</td>
<td>0.7854</td>
<td>7.19</td>
<td>2.61</td>
</tr>
<tr>
<td>LShoulderPitch</td>
<td>-2.0944</td>
<td>2.0944</td>
<td>8.27</td>
<td>2.27</td>
</tr>
<tr>
<td>RShoulderPitch</td>
<td>-2.0944</td>
<td>2.0944</td>
<td>8.27</td>
<td>2.27</td>
</tr>
<tr>
<td>LElbowYaw</td>
<td>-2.0944</td>
<td>2.0944</td>
<td>8.27</td>
<td>2.27</td>
</tr>
<tr>
<td>RElbowYaw</td>
<td>-2.0944</td>
<td>2.0944</td>
<td>8.27</td>
<td>2.27</td>
</tr>
<tr>
<td>LHipYawPitch</td>
<td>-0.9425</td>
<td>0.6807</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>RHipYawPitch</td>
<td>-0.9425</td>
<td>0.6807</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>LHipRoll</td>
<td>-0.4363</td>
<td>0.7854</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>RHipRoll</td>
<td>-0.7854</td>
<td>0.4363</td>
<td>4.16</td>
<td>11.97</td>
</tr>
<tr>
<td>LAnklePitch</td>
<td>-1.2217</td>
<td>0.7854</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>RAnklePitch</td>
<td>-1.2217</td>
<td>0.7854</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>LShoulderRoll</td>
<td>0.0</td>
<td>1.658</td>
<td>7.19</td>
<td>2.61</td>
</tr>
<tr>
<td>RShoulderRoll</td>
<td>-1.658</td>
<td>0.0</td>
<td>7.19</td>
<td>2.61</td>
</tr>
<tr>
<td>LHipPitch</td>
<td>-1.5708</td>
<td>0.5236</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>RHipPitch</td>
<td>-1.5708</td>
<td>0.5236</td>
<td>6.4</td>
<td>7.78</td>
</tr>
<tr>
<td>LElbowRoll</td>
<td>-1.658</td>
<td>0.0</td>
<td>7.19</td>
<td>2.61</td>
</tr>
<tr>
<td>RElbowRoll</td>
<td>0.0</td>
<td>1.658</td>
<td>7.19</td>
<td>2.61</td>
</tr>
</tbody>
</table>

Table 2.2: Servo Motors of NaoV3R Model.

In the Nao model the simulated Servo motors have the same orientation and range of motion as in the real Nao robot. Therefore motion sequences designed for the real Nao should play more or less similarly on the simulated NaoV3R model. The simulated motors will also allow approximately the maximal torque and velocity of the real motors. Table 2.2 shows the available simulated Servo motors and their main properties.

### 2.6 Programming

In the current Webots version, a robot controller can be developed with one of the following programming languages: C, Java, URBI, Python and Matlab. The controller can access the devices of the simulated Nao robot through a special library provided by the Webots simulator. Every robot in the RobotStadium environment must be associated with one of the two available robot controllers: the *nao_soccer_player_red* (used by the red Nao robots) and *nao_soccer_player_blue* (used by the blue Nao robots). The source code for these controllers must be located in one of the two folders according to the team color. The controller code must be able to play as both a red or a blue robot, because during the contest matches it will
play one half time with each color.

Every controller must be developed taking into account the basic “Time Step” of the simulation. The basic “Time Step” specified in the RobotStadium environment corresponds to the physics integration step. This is a virtual time duration that indicates how often the forces are recomputed and applied to the simulate rigid bodies. A sensor measurement and a motor force actuation can only happen between two basic time steps. Controllers must specify the control step by calling the method “robot_step()” from the library. By default, in RobotStadium, time step is defined as 40ms, which means that controllers can read sensors and actuate motors at maximal virtual frequency of 25 Hz. In Webots, the controllers can be synchronous or asynchronous. In synchronous mode Webots waits endlessly for the robot to call “robot_step()”. The synchronous mode is the default and is used for most simulations however not for RobotStadium in order to ensure the CPU fairness of the game. In synchronous mode a player could use several times more CPU than its opponent and that would be unfair.

Developed controllers for RobotStadium must implement the following cycle:

1: readSensors();
2: think();
3: actuateSensors();
4: robot_step();

First the controller reads the sensors and updates its world model. Then the controllers deliberate about the next actions to perform taking into account the state of the world model. According to the actions chosen in the previous step, controller must set the actions in the actuators. Finally calling the “robot_step()” method the controller gets synchronized with the simulation, sensors are updated with new values and specified actions are performed on the actuators. The simulator might carry out several simulation steps before the controller comes with a new actuator command. If the 3 functions “readSensor()”, “think()” and “actuateMotors()” take more time than Webots needs to carry out the simulation step, then Webots does not wait. Which means that it will just continue the simulation while using the previous motor target position of the previous “robot_step()”. So the quicker is the calculation in the first 3 methods of the cycle, the quicker are the robot reaction and motion.

Design of motion behaviors is one of the most challenging problems in the development of a robot controller. A motion is a text file in comma separated value format which specify motion sequences that usually involve several servo motors playing simultaneously. Every participating team must design motion behaviors for basic abilities like walking, turning and standing up. These motion files are accessed by the robot controller and can be played, stopped, resumed and reversed. A motion file consist of a table in which there are one column for every motor and other for the simulation time. Every row specify the servo positions (in radians) for the given time. The rows are sorted by the time column in ascending order. Time must be specified in multiples of basic “Time Step” (40ms). Table 2.3 shows a simple motion example in which there are moved three servos during 800 ms taking four different poses.
<table>
<thead>
<tr>
<th>#Webots_Motion</th>
<th>V1.0</th>
<th>RShoulderPitch</th>
<th>LShoulderPitch</th>
<th>RShoulderRoll</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:200</td>
<td>Pose1</td>
<td>1.57</td>
<td>1.57</td>
<td>0</td>
</tr>
<tr>
<td>00:00:400</td>
<td>Pose2</td>
<td>1.37</td>
<td>1.37</td>
<td>0.1</td>
</tr>
<tr>
<td>00:00:600</td>
<td>Pose3</td>
<td>1.27</td>
<td>1.27</td>
<td>0.2</td>
</tr>
<tr>
<td>00:00:800</td>
<td>Pose4</td>
<td>1.07</td>
<td>1.07</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 2.3: Example of a motion table.

2.7 Summary

This chapter presents a brief description about the testbed using in this thesis. Testbed corresponds to the Webots simulator and the RobotStadium competition. First, the characteristics of the Webots simulator were shown. Then, the soccer environment was specified and their main properties were described. Also, the definition and characteristics of the simulated robots, as well as their sensors and actuators were shown. Finally, a brief description of the programming process was described.
Chapter 3

Solution method: Reinforcement learning plus generalization

The type of feedback available for learning is usually the most important factor in determining the nature of a learning problem. According to [Russell and Norvig, 1995], there are three main types of learning: supervised, unsupervised and reinforcement. The problem of supervised learning is learning a function by means of examples of its inputs and outputs. Unsupervised learning is to learn from input patterns which are not specified the values of their outputs. The Reinforcement Learning (RL) problem is the most general of the three learning types. Instead of the agent receive instructions on what to do by a teacher, the agent must learn from reinforcement. RL is focused on goal directed learning from interaction.

This thesis implements a framework that integrates an RL algorithm for the automatic generation of motion policies plus a generalization mechanism allowing to work in continuous environments like RobotStadium. RL algorithm is based on the classical Q-Learning algorithm using a neural network instead of the Q-Table to represent the mapping between states and actions. In CQLF, the neural network works as a function approximator that allows to make the process of generalization. The following sections provide a brief background on RL and generalization, specifically about Q-Learning and generalization with neural networks.

3.1 Reinforcement Learning

According to [Sutton and Barto, 1998], RL is a machine learning technique where an agent learns from the environment while it is in it. With this technique, goal directed agents learn how to map states or situations to actions in order to maximize their utility. The learner is not told which actions to take, instead learners must discover which actions give the highest reward by trying them. Moreover, a key factor in RL is that actions may affect not only the immediate reward but also the future rewards.

One of the main challenges present in RL is the trade-off between “exploration” and “exploitation”. To obtain a lot of reward, a RL agent must prefer actions that it has tried in the past and found to be effective in producing reward (exploitation). But to discover such actions, it has to try actions that it has not selected before (exploration).

[Sutton and Barto, 1998] describe four main elements of a RL system:

Policy Defines the way in which a learning agent behaves at a given time. A policy is a mapping from perceived states of the environment to actions to be taken.
**Reward Function** Defines the goal in a RL problem. It maps each perceived state (or state-action pair) of the environment to a single number, a reward, indicating the desirability of that state.

**Value Function** Specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

**Model of the Environment** Mimics the behavior of the environment. Models are used for planning and might predict the resultant next state and next reward.

The action selection process is based on value function instead of the reward function. The objective is to select actions that result in states of highest value rather than highest reward, because actions look for the highest amount of reward in the long run.

The interaction between a learning agent and its environment is defined by states, actions, and rewards. The agent selects actions and the environment responds to those actions presenting new situations and also giving rewards. [Sutton and Barto, 1998] formally define the interaction between agent and environment as follows: given a sequence of discrete time steps $t = 1, 2, 3...$ at each time step $t$, the agent receives state, $s_t \in S$, where $S$ is the set of possible states, and on that basis selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state $s_t$. One time step later, the agent receives a reward, $r_{t+1} \in \mathbb{R}$, and a new state, $s_{t+1}$. Figure 3.1 shows the agent-environment interaction.

At each time step $t$, the agent maps current state to probabilities of selecting each possible action. This mapping is called the policy, and is denoted $\pi_t$, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$.

In a RL system, agent always learns to maximize its reward. Therefore, reward functions must be designed in such a way that by maximizing them, the agent will also achieve the desired goals. In RL systems reward functions indicate to the learning agent what must be accomplished. Moreover reward functions can be immediate or delayed. Immediate it is when reward is obtained for each action made exactly after their realization and delayed when reward is obtained after completing a sequence of actions.
3.2 Q-Learning

According to [Sutton and Barto, 1998], there are three fundamental classes of methods for solving the RL problem: Dynamic programming, Monte-Carlo methods, and Temporal-Difference learning. Dynamic Programming (DP) methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo (MC) methods do not require a model, but are not suited for step-by-step incremental computation. Finally, Temporal Difference (TD) methods require no model and are fully incremental, but are more complex to analyze. TD learning is a combination of MC and DP ideas. Like MC, TD methods can learn directly from raw experience without a model of the dynamics of the environment. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (bootstrapping).

Almost all RL algorithms are based on estimating value functions. Value functions are functions of state-action pairs that estimate how good it is for the agent to perform a given action in a given state. Temporal Difference (TD) Learning methods are used to estimate value functions. TD methods learn their estimates in part on the basis of other estimates. The state-action values are updated every training step. Expressed formally:

\[ V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \] (3.1)

The parameters used in the update process are:

- \( \alpha \) the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly.

- \( \gamma \) the discount factor, also set between 0 and 1. If \( \gamma = 0 \), the agent is concerned only with immediate rewards. As \( \gamma \) approaches 1, the future rewards are more important.

There are two types of TD methods: On-Policy TD methods that learn the value of the policy that is used to make decisions and Off-Policy methods that learn different policies for behavior and estimation. In Off-Policy methods value functions are updated using results from executing actions determined by some policy. Two common policies are used for executing actions:

- \( \epsilon \)-greedy most of the time the action with the highest estimated reward is chosen. With probability \( \epsilon \) an action is selected at random.

- softmax A random action is selected with regards to the weight associated with each action, meaning the worst actions are unlikely to be chosen. This is a good approach to take where the worst actions are very unfavorable.

Q-Learning is an Off-Policy algorithm for TD learning. It can be proven that given sufficient training using any soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy.
The procedural approach for Q-Learning is as follows:

Algorithm 1 Q-Learning

1: Initialize the Q-values table, \( Q(s,a) \).
2: Observe the current state, \( s \).
3: Choose an action, \( a \), for state \( s \) based on one of the action selection policies (\( \varepsilon \)-greedy or softmax).
4: Take the action, and observe the reward \( r \), as well as the new state, \( s' \).
5: Update the Q-value for \( s \) using \( r \) and the maximum reward possible for \( s' \).
6: Set \( s = s' \), and repeat the process until a terminal state is reached.

Formally the update process is described as follows:

\[
Q(s,a) = Q(s,a) + \alpha [r + \gamma \max_a Q(s',a) - Q(s,a)]
\]  

(3.2)

3.3 Eligibility Traces

Eligibility traces are one of the basic mechanisms of RL. Eligibility traces indicate the degree to which each state is eligible for learning changes every learning step. In order to calculate eligibility traces there is an additional memory variable associated with each state. The eligibility trace for state \( s \) at time \( t \) is denoted \( e_t(s) \in \mathbb{R}^+ \). Every step, the eligibility traces for all states decay by \( \gamma^\lambda \), and the eligibility trace for the one state visited on the step is incremented by 1. Figure 3.2 illustrate the behavior of accumulative eligibility traces.

\[
e_t(s) = \begin{cases} 
\gamma^\lambda e_{t-1}(s) & \text{if } s \neq s_t \\
\gamma^\lambda e_{t-1}(s) + 1 & \text{if } s = s_t
\end{cases}
\]

Eligibility traces are a very useful mechanism when rewards are delayed by many steps.

3.4 Generalization and Function Approximation

Estimates of value functions are represented as a table with one entry for each state or for each state-action pair. This is a clear and instructive case, but it is limited to tasks with small numbers of states and actions. For large tables the problem is not just the memory needed, but the time needed to fill them accurately.
One of the simplest methods of dealing with large state spaces is discretization. When it is performed, a state space is split into small size areas, each being an input of the Q-table. The success directly depends on how well this splitting represents the Q-function. To achieve a greater accuracy, state space must be split into smaller areas resulting in a Q-table of a larger size. On the other hand, splitting into larger areas can result in the impossibility of reaching the optimal control policy.

Several methods exist that help to speed up the process of learning when Q-tables of a large size are used. The objective is to produce a good approximation of a large state space based on a generalization over a limited subset of the state space. Generalization from examples has already been extensively studied in supervised learning field. The kind of generalization required is often called function approximation because it takes examples from a desired function (a value function) and attempts to generalize from them to construct an approximation of the function.

It is well-known that multilayer perceptron is a good function approximator. The Kolmogorov neural network mapping theorem [Hubbard and Ilyashenko, 2003] states that feedforward neural networks with three layers (input, layer, and output layers) can accurately represent any continuous function. The multilayer artificial neural networks using the error backpropagation algorithm is a function approximation that have been used widely together with RL.

3.5 Connectionist Q-Learning Framework

When connectionist approach is employed in the Q-learning algorithm, the tabular representation of Q-function is replaced by a neural network. States are forwarded to the inputs of the network whereas estimates of Q-values serve as output data. The Connectionists Q-Learning (CQL) method consists in applying a separate neural network for each action as shown in figure 3.3.

During each iteration of the algorithm, the current state of the system is forwarded to the inputs of each neural network, but the weights are only updated for the network whose action was selected. Formally, the network weight correction error using CQL is:

\[ e_t = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \] (3.3)

Sutton [Sutton, 1987] has described in detail the application of the TD algorithm with neural networks. The algorithm is called on-line update and is based on using vectors of
eligibility traces with which the weights of the neural network are provided. The use of eligibility traces makes possible to take into account the error at the previous steps as they store the weighted sum of output gradients. The algorithm is as follows:

**Algorithm 2 Connectionist Q-Learning**

1. Set eligibility traces equal to zero, \( e_0 = 0 \).
2. Set \( t = 0 \).
3. Select action \( a_t \).
4. If \( t > 0 \), then make weight correction \( w_t = w_{t-1} + \alpha (r_{t-1} + \gamma Q_t - Q_{t-1}) e_{t-1} \).
5. Calculate the output gradient \( \nabla_w Q_t \) only for the network the action of which was selected.
6. Set \( e_t = \nabla_w Q_t + \gamma \lambda e_{t-1} \).
7. Execute action \( a \) and take reward \( r_t \).
8. Stop if the absorbing state has been reached, otherwise \( t = t + 1 \) and go to Step 3.

The Free Connectionist Q-learning Framework (CQLF) [Kapusta, 2010] is an Open Source Java library for developing simple or complicated learning systems. It can be used anywhere, where an action can be chosen depending on the environment state, and where executing the action can be rewarded or punished. The framework is small, easy to use and speeds up the development of learning agents.

The basic idea of the framework is based on a neural network, which is trained using the CQL algorithm by Sutton [Sutton, 1987]. Figure 3.4 shows the architecture of the framework. Sensor inputs are collected by the Perception module. One of the inputs is the reward given to the agent for actions performed in previous steps. This information is passed to the Brain module, which learns, by the reward value, whether what the agent did before was good or bad. The output of the network is the Q-function, so the number of output neurons equal to the number of actions.

The framework contains three main Java classes: Perception, Brain and Action. The perception class is an abstract class that implements three main abstract methods:

- **isUnipolar()** Boolean method that determines if the input values are between 0 and 1 (true) or between -1 and 1 (false).

- **updateInputValues()** Void method that contains \( n \) calls to “setNextValue(value)”, where \( n \) is the number of inputs and “setNextValue()” is a method that sets input values for the
Brain class.

**getReward()** Double method that calculate the reward or punish obtained in a given state and the previous action.

This method must be implemented in a custom class extending Perception class. Thus, it is possible to set how the agent will observe the world and how the agent will be punished or rewarded. An important consideration is how the inputs (states and rewards) are normalized in order to be a correct input for the CQLF. Recall that in the CQLF the neural network uses a sigmoid transfer function. A sigmoid curve is produced by a mathematical function having an “S” shape as shown in figure 3.5. Sigmoid function is often defined by the next formula:

\[
P(t) = \frac{1}{1 + e^{-t}}.
\]  

(3.4)

In CQLF sigmoid function can be unipolar if the output of the function is between 0 and 1 or bipolar if the output is between −1 and 1. Equations 3.5 and 3.6 show the formulas for the unipolar and bipolar sigmoid transfer functions.

\[
1.0/(1.0 + \exp(-d))
\]  

(3.5)

\[
2.0/(1.0 + \exp(-d)) - 1.0
\]  

(3.6)

In CQLF the desired transfer function is configured in the Perception class using method “isUnipolar()”. If method returns true, the sigmoid function is unipolar and bipolar if the return is false. Scaling on inputs is another important consideration. For example, using a bipolar sigmoid function, an state input of a servo motor between $-\pi/3$ and $\pi/3$ produce an output range between −0.48 and and 0.48. In this case, the output range of the function does not fully cover the maximum possible range between −1 and 1. If the state input is scaled by a factor of 3 the output range is between −0.91 and 0.91, giving an output that covers almost the entire possible range, making the states better distributed throughout the output range.
The abstract class Action implements an abstract method called execute. This is a void method that contains the behavior of an specific action of the agent. For each action the agent can perform, there must be a custom class extending the Action class.

Finally, the Brain class contains all the necessary stuff to execute the Connectionist Q-Learning algorithm. In the custom Agent implementation there must be an instance of Brain class which must be configured indicating the Perception class, the Action classes and the parameters of the algorithm. The Brain class contains the methods “count()” and “execute-Action()” whose performs a step of the algorithm. After every training epoch, the agent must call the method “reset()” of Brain class.

An important configuration concerning the neural network is the amount of hidden layers (layers between input and output layer) and their number of neurons. This parameter is set at the moment of creating the Brain class. Set the amount of neurons in the hidden layer is really tricky and requires lot of testing and experience in neural networks. This parameter is crucial in determine the capability of the network to learn the policy.

Parameters of CQL algorithm are configured in the Brain class of CQLF by the following methods:

- setAlpha(α), for learning rate.
- setGamma(γ), for discount factor (delayed reward).
- setLambda(λ), for forgetting rate (elegibility trace).

Finally, CQLF uses ε-greedy policy for executing actions during learning. ε-greedy policy choose the action with the highest estimated reward most of the time. With probability ε an action is selected at random. User can configure parameter ε in order to keep balanced the trade-off between exploration and exploitation. ε is configured in CQLF by the “setRandActions()” method which receive as parameter the percentage of time (1-100) a random action will be selected instead the action with the highest reward. Appendix A shows a complete example of a Java implementation of the CQLF.

After finishing the training epochs, the learning policy cab be saved using the “saveQ” method of Brain class. Afterwards the policy can be loaded again using the “loadQ” method of Brain class. As shown in the example, the framework is very intuitive and easy to use and at the same time very powerful and customizable. However, it can be extended with desired functionality. The framework is contained in a single jar file, including the source code, and it is published by [Kapusta, 2010].

3.6 Related Work

One reason that RL is popular is that is serves as a theoretical tool for studying how agents learn to act. However RL is very popular in practice, it has also been used by a number of researchers as a practical computational tool for constructing autonomous systems that improve themselves with experience. These applications have ranged from robotics, to industrial manufacturing, to combinatorial search problems.
Starting in 2008, with the launch of RobotStadium and the appointment of the robot Nao as the new official RoboCup SPL robot, many RoboCup teams have been working with Nao robots in either simulation (RobotStadium and RoboCup 3D simulation league) or real robots (RoboCup SPL). Currently there are few published results concerning reinforcement learning applied to robot control tasks in the robotic soccer field.

In [Saggar et al., 2007] is presented an implementation of the policy gradient machine learning algorithm that searches for a parameterized walk policy while optimizing for both speed and stability. Their main objective was to prevent unsteady camera motions which degrade the robot's visual capabilities by achieving an stable and fast walking policy for the Sony Aibo (4-legged) robot, which was the official robot for the SPL until 2007. In their experimental results they achieved a walking policy reasonably fast and considerably more stable compared to their previous policies. The checked that the achieved stability significantly improved the robot's visual object recognition.

More recently, [Hester et al., 2010] presented a RL algorithm with Decision Trees called RL-TD that uses decision trees to learn the model by generalizing the relative effect of actions across states. The objective of the proposed algorithm is to be sample efficient. This implies to learn competent behavior from very few real-world trials. Experimental work presents a comparison between classical model-free RL techniques against the RL-TD algorithm. This experiments are based on a on an physical Nao humanoid robot scoring goals in a penalty kick scenario. Experimental results demonstrate that RL-DT is a good choice for learning tasks on real robot. The proposed approach allow to learn a reasonable policy quickly without lot of exploration which can be very expensive and time consuming to get on a real robot.

Below are papers related to the proposed solution method of this research work. These papers propose some learning approaches applied to robot control problems that could be applied or combined with the solution method proposed in this chapter to solve the problem of generating motion policies.

The CQLF implemented in this research work is based on the work by [Kuzmin, 2002] which defines the connectionist Q-learning algorithm and also implements it in a 2D robot control task. The main feature of the work is that in the process of learning the system is not shown how to act in a specific situation. Instead, learning develops by trial and error using reward and penalty signals. His base algorithm is Q-learning additionally introducing generalization means using multilayer perceptron (MLP) as a Q-learning table approximator. The studied robot task consisted to reach a goal and avoid collision with random positioned obstacles. His experiments had as purpose to compare the modifications of the Q-learning algorithm. During experiments Kuzmin found that modifications of the Q-learning algorithm had a quicker convergence than the classic Q-learning algorithm. Specifically the connectionist Q-learning algorithm with on-line learning demonstrated the best results.

The work of Kuzmin, as this research work, demonstrate how to successfully implement RL techniques in continuous environments. Kuzmin also experimented using software simulator of a robot functioning in the continuous environment. Unlike this work, Kuzmin used a Boltzmann function to maintain a more active exploration in the early stages and gradually reduce it to have an exploitation policy. In addition, the work of Kuzmin is limited to a single
learning agent and also situated in a deterministic environment with no noise in perceptions or actions. Kuzmin’s work is very important because it clearly demonstrates the advantage of generalization in the learning process in continuous environments.

Another work that successfully used neural networks for generalization in learning of robot control policies is described in [El-Fakdi et al., 2005]. In this paper, it is presented a policy method as an alternative to value methods to solve RL problems. This paper proposes a direct policy search algorithm using a neural network to represent the control policies. In that method the policy is represented by a neural network whose input is a representation of the state, whose output is action selection probabilities, and whose weights are the policy parameters. The method is based on a stochastic gradient descent with respect to the policy parameter space (network weights). The methodology is to approximate directly on the policy parameters (network weights) to find the configuration that maximizes the reward. The methodology is called direct policy search because it works on the parameters of the policy and not on the values of the states.

[H. and de Lope, 2007] presents a distributed approach to RL in multi-link robot control tasks. This approach avoids the combinatorial explosion when multiple states variables and multiple actuators are needed to optimally control a complex agent in a dynamical environment. The experimental results clearly show that it is not necessary that each individual agent perceives the complete state space in order to learn a good global policy but only a reduced state space directly related to its own environmental experience. This work uses SARSA (another TD algorithm) as the basis for the RL algorithm.

Mataric [Mataric, 1994] describes a robotics experiment with a high dimensional state space with many dozens of degrees of freedom. Mataric described some enhancements to the basic Q-learning algorithm including a decentralized control in which each robot learned its own policy independently without explicit communication with the others (approach used in this research work). Instead of using generalization, Mataric brutally quantized state space into a small number of discrete states according to values of a small number of boolean features of the underlying sensors. The performance of the Q-learned policies were almost as good as a simple hand-crafted controller for the job.

A very interesting robot control application is the work of [Smart and Kaelbling, 2002] which introduces a framework for RL on mobile robots. It describes a value-function approximation approach through the use of a general-purpose function approximator called locally weighted regression (LWR). The key feature of this paper is the introduction of prior knowledge into the learning system. The problem is that if there are only a few rewards, and the state-action space is large, the chances of finding a reward by chance are very small. They solved the problem by implementing 2 learning phases:

In the first phase the robot is being controlled by a supplied control policy. This can either be actual control code, or a human directly controlling the robot. During this learning phase, the RL system is passively watching the states, actions and rewards that the supplied policy is generating. It uses these rewards to bootstrap information into its value-function approximation. The key element of this phase is that the supplied control policy exposes the RL system to the “interesting” parts of the state space (parts where the reward is non-zero).
In second phase, the learned policy is in control of the robot, as it would be in a standard RL implementation. By splitting the learning into 2 phases is ensured that, once the second learning phase starts, the robot will be capable of finding reward giving states. This sort of learning by demonstration has become quite popular recently.

3.7 Summary

This chapter contains a theoretical framework about Reinforcement Learning. It defines the RL problem and its components and describes the elementary solution methods specially Temporal Difference methods and particularly the Q-Learning algorithm. Also are explained the concepts of eligibility traces and generalization. Finally is defined the Connectionist Q-Learning algorithm and its implementation in the Connectionist Q-Learning Java Framework.
Chapter 4

Motion Policies Experiments

A motion policy is a function that maps states to motion behaviors. State consists of the simulation time, a continuous variable specified in multiples of basic "Time Step" (40ms). Motion behaviors imply the movement of servo motors to a specific position. The position of a servo motor is a continuous angle in radians delimited by a range (minPosition, maxPosition). In conclusion, a motion policy receives as input the simulation time and it gives the positions of servo motors as outputs.

An example of a motion policy could be a "getting up policy". For getting up many servomotors of nao robot are involved. Thus "getting up policy" has many outputs indicating positions for each servomotor. The policy receives the simulation time as input in an incremental way. For each input, policy sets the positions of the servo motors. However, this does not guarantee that the robot will be in an upright position after finishing with the last time step because there are not feedback available to the policy.

Unlike motion tables, motion policies can be provided with feedback about the positions of the servo motors. Consider an example in which a servo motor needs to go to the zero position (0 radians) and the initial position is set randomly. The available actions are moving the motor towards $\pi$ and towards $-\pi$ radians. The time-based motion policy could fail because completely ignores the actual motor position. However, motion policy can be adapted by setting as input the actual position of the servo motor. In this case the state is the position of the servo motor and the state space is between $-\pi$ and $\pi$ radians. With this state, motion policy have information about the current position of the servo motor and can select a correct action. Nevertheless this state space is also continuous.

4.1 The Problem of Generating Motions Policies

The classical design of motion policies is a laborious effort. It requires many hours of designing, testing, and redesigning the policy until the desired behavior is achieved. Every motion policy generally requires the movement of various servo motors and at the same time, thus the continuous state and action spaces become enormous.

This thesis implements a RL approach for automatic generation of motion policies. Through the experience of the agent in its environment, it will be able to generate the required motion policies. However the problem of enormous state and action spaces is still present for the RL approach.
The first approach trying to deal with big state spaces is discretization. In this way the continuous spaces are split into small size areas, each being an input of the Q-table. For example, for a single servo motor, its state space between $-\pi$ and $\pi$ can be split every 0.087 radians (approximately five degrees). In this case, the Q-table will have 72 rows, one for each state.

For the case of one servo motor, discretization can be applied successfully, but for motion policies involving several servo motors, this is not a good approach. Consider the task of learning to kick a ball. This scenario involves at least moving the servos from one leg. In the Nao robot each leg have five servo motors and supposing each one has a range between $-0.7854$ and $0.7854$ radians (approximately 130 degrees), splitting every five degrees result in 26 areas for each servo motor. Thus combining all servo motors in the Q-table result in $26 \times 26 \times 26 \times 26 \times 26 = 26^5 = 11,881,376$ rows. This is an enormous quantity of possible state combinations. The state space grows exponentially increasing the number of servo motors. Therefore, discretization is not useful in learning of motion policies.

Generalization helps to speed up the process of learning when Q-tables of a large size are used. Generalization is the only way to infer from previously experienced states to ones that have never been seen, a very common situation on many interesting RL tasks. This thesis implements the Connectionist Q-Learning (CQL) algorithm in which a multilayer artificial neural network with the error backpropagation algorithm performs the generalization. With this approach the tabular representation of Q-function is replaced by a neural network which makes possible to work with continuous state spaces.

4.2 Empirical Scenarios

For motion policies the state space is a combination between the simulation time and the position of the servo motors. A servo motor can be set to a specific position between (minPosition, maxPosition) range. This is a continuous action space. In order to implement the CQL framework, the action space needs to be discretized because the framework requires a discrete number of actions (output neurons = number of actions). Moreover, while smaller the number of actions, simpler the learning problem (i.e. the neural net is simpler).

A very simple way to discretize the action space is generating three possible behaviors for each servo motor: move forwards, move backwards and stay in current position. Moving forwards and backwards behaviors modify the position of servo motor adding or subtracting respectively the constant “motor_step” to the current position. The “motor_step” constant is a small step in the position and is set about three to five degrees. This is equivalent to split the range of a servo motor (action space) into small areas of three to five degrees.

For the problem of learning motion policies there were defined two different empirical scenarios using the CQLF: Single learning agent which must learn to control all servo motors and multiple learning agents, one agent per each servo motor.

The single agent approach is a simple way to implement the learning agent, but has an important disadvantage, there are lots of possible combinations of actions. For example, a motion policy involving five servo motors has $3 \times 3 \times 3 \times 3 \times 3 = 3^5 = 243$ possible actions. This
is because every motor can move in two different directions or stay in the current position. At each simulation step, every motor must select one of the three possible movements, thus there is a combinatorial number of possible actions depending on the number of servo motors involved. Figure 4.1 shows a diagram of the single agent empirical scenario for learning motion problem.

In order to avoid the combinatorial number of possible actions, this approach is adjusted to select only one movement per step. For example, for five servo motors there are only $3 \times 5 = 15$ possible actions corresponding to the three possible movements for each servo motor. By doing this, action space is reduced enormously, but also the parallelism is lost. This empirical scenario seeks to test the hypotheses that parallelism can be simulated, choosing a movement of a servo motor per step. This can be compared to a multi task processor in which each process uses the processor for short periods of time, but the end result is almost as if they were running at the same time.

On the other hand, the multiagent approach is a much more natural implementation for the learning agent. Here every servo motor is a learning agent generating its own motion policy. Individual policies are simpler than the general policy, they have only three possible actions. They can retrieve the simulation time and optionally its own position and the position of other servo motors as inputs. This approach implicitly contains parallelism because all servomotors are learning at the same time. Thus every simulation step all servo motor execute one of the three possible actions.

Multiagent empirical scenario aims to prove that complex environments can be modeled as a group of simple interacting agents with single state spaces as well as single action spaces. In multiagent approach all learning agents receive the same global reward. Thus if the reward function returns a negative reward, all agents are punished regardless of whether they have done really well. This implies difficulty to individual learning, however, in the problem of learning motion policies is very difficult to design a reward function for every learning agent (servo motor). This is due to the nature of the problem where the objective (learning task)
depends on the joint actions of all agents. Figure 4.2 shows a diagram of the multiagent empirical scenario for learning motion problem.

Two different experiments were designed for the problem of learning motion policies. These experiments correspond to common tasks in the RobotStadium environment: "kicking the ball" for field players and "jumping to block the ball" for the goal keeper. For this two experiments were implemented both empirical scenarios. Following sections describe the experiments design, implementation and results.

4.3 "Kicking the Ball" Experimental Scenario

Kicking the ball is the most representative behavior in soccer. Soccer is entirely based on hitting the ball with feet. Kicking the ball may seem an easy task but it is not. The kicking motion requires a lot of technique to hit the ball properly. In the RobotStadium environment, kicking motions require lot of balance in robots and generally aim to hit the ball as hard as possible and in a straight direction. For kicking motion generally the servo motors from both legs are used and sometimes also the arms are involved.

4.3.1 Design of experiment

"Kicking the ball" experimental scenario consists of a RobotStadium environment in which there is only one robot on the soccer field. The right foot of the robot is aligned with the ball and separated 15 cm of this. The ball is at the center of the field. Also the environment contains a supervisor code encouraged to place the player and ball at the beginning of each training epoch and sends a message to the agent when the ball is touched. Figure 4.3 shows the RobotStadium environment for this experiment.

The learning task consists in kicking the ball with the right leg. For simplicity, and with the aim to reduce the state space, in this experiment robot do not use arms for kicking. Also the robot must kick the ball moving only three motors from right leg (RHipPitch, RKneePitch,
Figure 4.3: Environment for kicking the ball experiment.

RAnkleRoll) and two motors from left leg (LKneePitch, LAnkleRoll). The less servo motors are involved, the learning task will be less complex. However it must be considered the feasibility of doing the task properly by reducing the number of actuators.

In RL the way to establish what the agent must to learn is through the design of rewards. For this experiment, in order to learn to kick the ball, a positive reward is given when the agent hits the ball. If the robot loses his balance then receives a negative reward. Apparently with these simple rewards the robot could learn to kick the ball without losing balance, nevertheless, it is not desirable that the robot touches the ball but kick as hard as possible, in a straight line and spending a minimum amount of time. Therefore, for both empirical scenarios (single agent and multiagent) the Perception module calculates the reward as follows:

- $-1$, if the robot falls.
- $-1$, if the time exceeds 8000 ms.
- $0.1 - 1$, if the robot receive a message from supervisor indicating that ball was touched. Reward depends on velocity and direction of the ball. The greater the speed and direction in a straight line, the greater the reward.
- $0$ otherwise.

With these rewards the objective is to learn to kick the ball, hitting hard and in straight line in a short period of time. A learning episode ends when the reward is different to 0, thus the absorbing states are when the robot falls, time exceed 800 ms, or robot kick the ball. As can be seen, the rewards different from 0 will be received until the end of the learning epoch, which is why the concept of delayed reward is important in the problem of learning motion policies.

For kicking experiments the Perception module were configured to bipolar transfer functions. Bipolar sigmoid function has a bigger output range and is more adequate for inputs that have negative and positive values like rewards between $-1$ and $1$ and states of servo motors between $-\pi$ and $\pi$. 
Perception module use unipolar (output range from 0 to 1) or bipolar (output range from -1 to 1) sigmoid transfer functions to pass input signals to the Brain. Thus, input signals must be scaled in order to accurately represent the state for the neural network. For servo motors positions is used a scaling factor of 3 giving the output range between —0.91 and 0.91 and for simulation time, is used the following formula:

\[ \frac{t}{25} - 4 \]  

(4.1)

Where \( t \) is the amount of simulation steps. If \( t = 0 \), output will be —4. If \( t = 200 \), the maximum allowed simulation time (200 steps \(*\) 40ms = 8000ms), output will be 4. Passing the output of this scaling function to the transfer function is obtained an output range that is distributed properly in the total range of —1 to 1.

This thesis implemented CQLF using only one hidden layer. The amount of neurons in hidden layer was configured specifically for each approach. For kicking experiments, the amount of random actions is set to 10 percent. Thus 1 of 10 times, agent will choose a random action instead of the best known action for current state. This parameter maintains an equilibrium between exploration and exploitation.

During kicking experiments, parameters of CQL algorithm will be tested with many values looking for those that show the quickest convergence. As a-priori knowledge, learning rate \( \alpha \) will be set between 0.2 and 0.5 for getting a soft learning and being capable to work with noise. Discounting factor \( \gamma \) will be very important because in kicking experiments all reward is delayed, thus this parameter will be very close to 1. Finally the forgetting rate \( \lambda \) will be around 0.5 playing an important role because here is configured how earlier states are given credit for the current TD error (reward).

### 4.3.2 Single agent approach

For the single agent approach, the Perception module pass six input signals to the Brain: simulation time and current position of the five servo motors. There are 11 Action modules,
one for each possible move (backwards and forwards) for the five servo motors and a simple “dummy” function that does nothing. Figure 4.4 shows the overall framework architecture for the single agent scenario solving the learning motion problem experiments.

In a variation to this experiment, the Perception module was modified to retrieve just one input signal corresponding to simulation time and ignoring the position of servo motors. This was with the aim to simplify the state space and thus have a simpler neural network which despite the loss of feedback nevertheless is able to learn the motion policy. Figure 4.5 shows the variation to the single agent framework architecture.

For single agent scenario in the “kicking the ball” experiment there were run many trials for the two configurations. Each trial consisted of 3000 training epochs. During trials, average reward is computed and saved in a text file. The average reward is the total amount of reward received by the agent during training epochs divided by the number of training epochs. This measure is very useful in determining how a learning agent is behaving in its environment. Average reward plots consist of a learning curve that clearly shows if the learning process converges to a solution or if the agent has not learned the desired behavior.

Table 4.1 shows the CQLF configuration parameters for the single agent approach for “kicking the ball” experiment. Configuration is the same for the two variants.

Table 4.1: CQLF configuration for the single agent approach for “kicking the ball” experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single agent approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>12</td>
</tr>
<tr>
<td>ε</td>
<td>0.1</td>
</tr>
<tr>
<td>α</td>
<td>0.2</td>
</tr>
<tr>
<td>γ</td>
<td>0.99</td>
</tr>
<tr>
<td>λ</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Figure 4.6 shows a comparison between the learning curves of the two variants of the single
agent approach for solving the “kicking the ball” experiment. The single agent approach had a regular performance achieving an average reward of 0.26. This approach converges to that value of average reward at which the agent learns to kick the ball, however, the kick is very weak so that the received reward is small. For the variant to the single agent approach, it is observed a much lower performance than the original approach. This variation only achieved a final reward of $-0.93$, which indicates that the agent was unable to learn the required task.

According to this results, it can be noticed that it is not possible to eliminate the feedback about the position of the servo motors. The learning agent requires that states provide information about the current position of the servo motors in order to differentiate between them and learn the mapping to actions properly. When trying to reduce the state space by reducing the number of inputs, it is possible to cause the failure of the agent to learn the required task.

Although the single agent approach learns to kick the ball weakly, it can be concluded that this approach can simulate the behavior of the motors in parallel despite the serial execution is carried out.

### 4.3.3 Multiagent approach

For the multiagent approach, the perception module for each learner is the same that for the single agent approach. Each Perception module receives the simulation time and position of all servo motors as inputs. Finally every learner has three possible actions: move forwards, move backwards and stay in current position. Figure 4.7 shows the overall framework architecture for the multiagent scenario solving the learning motion problem for the kicking
In this empirical scenario there were implemented two alternate configurations. In first variation, the Perception module of each learning agent just retrieve the simulation time as input, ignoring the position of others servo motors, and in the second variation, learning agents also retrieve their own position. This is for the purpose of verifying that the actions of other agents in the environment can be ignored and yet be able to learn an adequate own policy. Figures 4.8 and 4.9 shows the variations to the multiagent framework architecture.

Many trials were run in order to obtain the average reward for the multiagent approach. Each trial consisted of 1500 training epochs unlike the single agent scenario in which were 3000 epochs. This is because in multiagent approach convergence was achieved more quickly as as will be shown later. Table 4.2 shows the CQLF configuration parameters for the multiagent approach for “kicking the ball” experiment and its variations.

Figure 4.10 shows a comparison between the learning curves of the three configurations of the multi agent approach for solving the “kicking the ball” experiment. In the graph can be observed the good performance of multiagent approach in the experiment of “kicking the ball”. This approach achieved a very rapid convergence since that only after 500 training epochs the average reward were close to 0.8, reaching 0.86 after 1500 epochs. In this case, the agent learned to kick the ball with considerable force as the final average reward is quite close to

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single agent approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>12</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.2</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Figure 4.8: First variation to multiagent approach for learning motion problem.

Figure 4.9: Second variation to multiagent approach for learning motion problem.
Just as in the single agent approach, variations to multiagent approach achieved less performance than the original approach. Despite this, these variations also managed to get a good performance converging rapidly and achieved average reward of 0.71 and 0.65 respectively. Variations also learned to kick the ball, but got a kick of lower quality.

Thus it can be concluded that the multiagent approach had a very good performance in the experiment of “kicking the ball”, achieving a fast convergence and obtaining an average reward near to optimal. For the variations can be concluded that by simplifying the state space by removing information from other agents, the agent had more difficulty learning the task required as states contains less important information about the environment. Despite the reduction in state space, these variations were successful in achieving convergence and getting a very acceptable final average reward.

4.3.4 Results and discussion

Finally, to conclude with the “Kicking the Ball” experiments, there were compared both empirical scenarios: single agent and multiagent. Figure 4.11 shows the learning curves of both approaches. It can be noticed that multiagent approach clearly outperforms the single agent approach. Multiagent approach had a quicker convergence and also a higher final average reward. During this experimental scenario, multiagent approach showed a good performance in combination with the CQLF.

Although the single agent approach was overcome by the multiagent approach, it was demonstrated that it is possible to achieve convergence with this approach by simulating a
parallelism in the execution of actions. It might be think that the single agent approach is
simpler than the multiagent as is required only to learn a policy, in spite of learning individual
policies for each motor. However, individual learning problems are much simpler than the
overall learning problem of single agent approach. Furthermore, the multiagent approach is a
more natural implementation given the inherent parallelism in the learning motion problem.

4.4 "Jumping to Block the Ball" Experimental Scenario

Jumping to Block the Ball is the most representative behavior for goal keepers in soccer.
Goal keepers must stop attacks from enemy by blocking the ball to prevent an enemy goal.
Jumping to block the ball is an spectacular and very useful motion and it is crucial for the
defense as the primary objective is that the ball does not enter in the defended goal. This
motion requires a lot of technique, but above all, speed to block the stronger shots from the
opposing team. In the design of this motion generally are involved the servo motors from both
legs and arms.

4.4.1 Design of experiment

"Jumping to block the ball" experiment consist of a RobotStadium environment in which
there is one robot on the soccer field. The robot is positioned like a goal keeper at the center
of the yellow goal and is separated 20 cm from goal line. Robot is positioned in an upright
position and with his right arm raised to reach the ball located beside. The ball is in front

Figure 4.11: Comparison between single and multiagent approach for "kicking the ball" ex-
periment.
of the right post of goal and is separated 20 cm from it. Also the environment contains a supervisor code encouraged to place the player and ball at the beginning of each training episode and sends a message to the agent when the ball is touched. Figure 4.12 shows the RobotStadium environment for this experiment.

The learning task consists in jumping to block the ball with the right hand. The robot must hit the ball with the right hand moving all the servo motors from both legs (HipPitch, HipRoll, KneePitch, AnklePitch, AnkleRoll). For both empirical scenarios (single agent and multiagent) the Perception module calculates the reward as follows:

- $-1$, if the time exceeds 320 ms.
- $1$, if the robot receive a message from supervisor indicating that ball was touched.
- $0.3$ if robot jump to the right side.
- $-0.1$ otherwise, the more time passes more will be punished.

With these rewards the objective is to learn to jump to block the ball by hitting it with the right hand as fast as possible. A learning episode ends when the reward is $-1$ or $1$, thus the absorbing states are when time exceed 320 ms or the robot hit the ball. Unlike the experiments for kicking the ball, here is configured a negative reward of $-0.1$ for nonterminal states. This is in order to make the robot learn to block the ball in the shortest possible time. Also the agent is rewarded when jumps to the right side with the aim to give him more insights about the desired behavior.

Like the experiments for kicking the ball, CQLF is configured to use bipolar sigmoid transfer functions and also is used scaling on inputs. For servo motors positions is used a scaling factor of 3 and for simulation time, is used the following formula:

$$ t/10 - 4 $$

Where $t$ is the amount of simulation steps. For example if $t = 0$, output will be $-4$. If $t = 80$, the maximum allowed simulation time ($80 \text{steps} \times 40 \text{ms} = 320 \text{ms}$), output will be
Table 4.3: CQLF configuration for the single agent approach for “jumping to block the ball” experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single agent approach</th>
<th>Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

4. Passing the output of this scaling function to the transfer function is obtained an output range that is distributed properly in the total range of $-1$ to $1$.

For jumping to block the ball experiments, the amount of random actions is set to 10 percent. Parameters of CQL algorithm will be tested with many values looking for those that show the quickest convergence. Learning rate $\alpha$ will be set between .2 and .5 for getting a soft learning. Discounting factor $\gamma$ will be very close to 1 because there is delayed reward. Finally the forgetting rate $\lambda$ will be around .5.

### 4.4.2 Single agent scenario

For the single agent approach, the Perception module pass 11 input signals to the Brain: simulation time and current position of the 10 servo motors from both legs. There are 21 Action modules, one for each possible move (backwards and forwards) for the 10 servo motors and a simple “dummy” function that does nothing. The framework architecture for the single agent scenario solving the learning motion problem is showed in figure 4.4.

In a variation to this experiment, the Perception module was modified to retrieve just one input signal corresponding to simulation time and ignoring the position of servo motors. This is with the aim to simplify the state space and thus have a simpler neural network which despite the loss of feedback nevertheless is able to learn the motion policy. Figure 4.5 shows the variation to the single agent framework architecture.

Just as in the previous experiment, for single agent scenario in the “jumping to block the ball” experiment, there were run many trials for the 3 configurations in order to obtain the average reward. Each trial consisted of 1500 training epochs. Table 4.3 shows the CQLF configuration parameters for the single agent approach for “jumping to block the ball” experiment and its variant.

Figure 4.13 shows a comparison between the learning curves of the 2 configurations for the single agent approach for solving the “jumping to block the ball” experiment. In the graph can be observed that the variation outperforms the original single agent approach. The variation to single agent approach converges faster retrieving a final average reward of 0.30 against 0.22 of the original approach. With these final average rewards the agent learned to block the ball in a fast way, but the result is far from optimal.

With the results can be confirmed once again that the single agent approach is able to
learn motion policies by simulating an execution of movement actions in parallel. Unlike the experiment of kicking the ball, in “jumping to block the ball” experiment, the variation to the original single agent approach achieves a better performance. This suggests that in the problem of learning to block the ball, states can be easily differentiated only by the simulation time, and for kicking the ball it is necessary to have information about the positions of the servo motors in order to distinguish between different states.

4.4.3 Multiagent scenario

For the multiagent approach, the perception module for each learner is the same that for the single agent approach. Each Perception module receives the simulation time and position of all servo motors as inputs. Every learner has three possible actions: move forwards, move backwards and stay in current position. Figure 4.7 shows the overall framework architecture for the multiagent scenario solving the learning motion problem for the “jumping to block the ball” experiment.

Also in this empirical scenario were implemented two alternate configurations. In first variation, the Perception module of each learning agent just retrieve the simulation time as input, ignoring the position of others servo motors, and in the second variation, learning agents also retrieve their own position. This is for the purpose of verifying that the actions of other agents in the environment can be ignored and yet be able to learn an adequate own policy. Figures 4.8 and 4.9 shows the variations to the multiagent framework architecture.

Here were also run many trials for the three configurations, each trial consisting of 1500 training epochs. Table 4.4 shows the CQLF configuration parameters for the multiagent

![Figure 4.13: Results of single agent approach for “jumping to block the ball” experiment.](image)
Table 4.4: CQLF configuration for the multiagent approach for “jumping to block the ball” experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single agent approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>15</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.3</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Figure 4.14 shows a comparison between the learning curves of the three configurations for the multiagent approach for solving the “jumping to block the ball” experiment. Here can be noticed that the second variation obtained the best performance of 3 configurations retrieving a final average reward of 0.56 converging really quick. The original multiagent approach obtained a very similar performance obtaining a final average reward of 0.52. The first variation obtained a similar performance than the single agent approach with 0.26 of final average reward. With multiagent approach the agent quickly learned to block the ball and in a fast way. It appears that the final value of the average reward is far from optimal but it should be considered the reward function in which the agent receives $-0.01$ in each simulation step. Thus the average reward value decreases every time.
With the results can be noticed again that the multiagent approach, along with CQLF, achieved good results in a learning motion policies task. Multiagent approach achieved a fast convergence and obtained an average reward near to optimal. In “jumping to block the ball” experiments also the variations to the original multiagent approach achieves a good performance in particular the second variation in which is considered simulation time and own servo motor position.

### 4.4.4 Results and discussion

Finally, regarding the experiment of “Jumping to block the ball”, there were compared both empirical scenarios: single agent and multiagent. Figure 4.15 shows the learning curves of best of both approaches. It can be noticed again that multiagent approach outperforms the single agent approach. Multiagent approach had a quicker convergence and also a higher final average reward.

Throughout the experiments in this chapter, it was demonstrated that it is possible to achieve convergence with the single agent approach by simulating a parallelism in the execution of actions, however, results were poor, achieving final average rewards far from optimal, and clearly outperformed by the multiagent approach. Despite the simplicity in the design of single agent approach for learning motion policies, the learning problem is complex because it loses the notion of parallelism and also global representations of states and actions will lead to a complex mapping (complex neural net) more difficult to learn.

On the other hand, for multiagent approach, despite increasing the number of agents
learning at the same time, individual mappings are less complex and have simpler representations of states and actions. It was demonstrated that multiagent approach obtained good results in convergence time and final average rewards close to optimal. It was shown that despite all the learning agents are fed by the same reward function, they manage to converge towards the global goal. This verifies that the CQL algorithm is able to learn even in noisy environments as the learning agent is able to converge towards the target even as their perceptions may be distorted by other agents or by the nature of environment.

Finally speaking of variations to state representations, it can be concluded that depending on the learning task, sometimes is better to include or remove information from the states regarding to information from other agents and even information about itself.

In this chapter, is concluded that RL is an useful technique for the automatic learning of motion policies, and in particular RL with generalization is able to learn properly in complex environments that contain continuous variables. Specifically the CQL algorithm that implements the CQLF is a very powerful tool to attack the problem of learning motion and also very easy to configure and use in a variety of environments where an agent receives information from its environment and can perform actions on it.

Also is concluded that the design of the reward function is essential for success in learning, as it is here where it is specified what is wanted the agent to learn. Also the design of inputs for the agent is essential in RL, thanks to a good specification of the state where the agent is, it can be learned to take the right action for each state. It is important that the input signals are fully capable of distinguishing between the states of the environment.

Surely the most difficult activity in order to learn motion policies with CQLF is the configuration of the parameters of both neural network and the CQL algorithm. It has been found that the algorithms yield good results in a fairly broad range of parameter changes. For the CQL algorithm results has shown the best result at these parameters:

- $\alpha = 0.2$
- $\gamma = 0.99$
- $\lambda = 0.5$
- $\varepsilon = 0.1$

With these values is obtained a soft learning being capable to work with noise($\alpha$), great importance to the delayed reward ($\gamma$) and giving credit to earlier states for the current reward($\lambda$). Also, in order to maintain an active exploration, the CQLF is configured to select a random action instead the best action 1 of 10 times. The most tricky configuration for CQLF are the parameters of neural net. Here, according to Kolmogorov, only is needed one hidden layer, however can be configured more than one hidden layer. The amount of neurons in hidden layer was configured specifically for every experiment, generally the amount of neurons is set slightly larger than the number of input signals.
4.5 Summary

This chapter defined the problem of the generation of motion policies within the Robot-Stadium environment. First the concepts of motion and motion policy were defined. Then the problem of generating motion policies were described and also was shown how motion policies can be automatically generated by RL techniques. Subsequently, it was defined how generalization can tackle learning problems involving continuous state spaces and actions. Moreover, this chapter details the use and configuration of the Connectionist Q-Learning (CQL) algorithm by means of Connectionist Q-Learning Framework (CQLF) for the implementation of RL with generalization in the problem of learning motion policies.

In order to test the hypotheses of this thesis regarding the development of motion policies, there were defined two empirical scenarios: single agent and multiagent. Regarding the first approach, it was designed as a single learning agent that controls all the robots actuators while for the second approach, there were defined a learning agent for each actuator involved in the learning task. For both empirical scenarios there were designed two experiments: “kicking the ball” and “jumping to block the ball”. In the first experiment the learning task consists to generate a motion policy to kick the ball with the right leg of the robot. The second experiment consists to jump to block the ball with the right arm of the robot.

After experiments design, there were showed the results and discussion of the two experiments using the CQLF in both empirical scenarios (a single agent and multiagent). It was concluded that multiagent approach together with the CQLF is a good tool to attack the problem of generating motion policies. In both experiments the multiagent approach overcame the single agent approach. Also it is described how multiagent approach is a more suitable alternative for the problem given its nature of parallelism. This proves the hypotheses of this study in that by implementing RL together with generalization is possible to generate motion policies. Specifically, with the multiagent approach agents can be modeled as a group of simple interacting agents with single state spaces (partial observations of states) as well as single action spaces (individual action selections).

Although single-agent approach is significantly overcame, it was shown how this approach is also able to learn the motion policies through the simulation of a parallel execution performed by a sequential execution of actions from actuators. It was conclude that despite the simplicity in the design of single agent approach, the global mapping between states and actions is much more complex than the individual mappings in multiagent approach. Finally, it was demonstrated how the input signals that contain information about the states, can be customized according to the learning task and sometimes in addition to the time they can also include information about the current state of the actuators in order to facilitate learning.
Chapter 5

Decision-Making Policies Experiments

A decision-making policy is a function $f$ that maps states to actions $S \rightarrow A$. In this case, the state consist of the perceptions of the environment that the agent gets through its sensors. Perceptions in RobotStadium environment include: vision, accelerometer, gyro, ultrasound, servo positions, and force sensitive resistors. As mentioned in chapter 2, in RobotStadium environment, perceptions are represented as continuous values. For example, accelerometer gives a three dimensional vector of float values indicating the gravity force on each axis. Decision-making policies involve high-level actions in contrast with motion policies where the actions are low level (servo positioning). Actually, in decision-making policies the actions correspond to motion behaviors. For example, an action of a decision-making policy is turn to the left. This high-level action implies the execution of many low-level actions. Finally, in decision-making policies, actions can take more than basic time step of 40 ms. In motion policies, every time step, learning agents takes an action based on the current state. In decision-making policies an action is selected until the last selected action has finished its execution.

An example of a decision-making policy could be a “chasing policy”. The learning task of this policy consists of chasing the ball until the robot is behind the ball and facing it. This policy has three possible outputs: turn left, turn right and walk. The policy receives the ball distance and direction calculated from the perceived image by the camera of the robot. This policy maps the state given by the relative ball position to the correct action in order to go for the ball. As can be noticed in decision-making policies, state representation and actions are configured according to the desired task. Nevertheless, the state space is always continuous and is obtained from the sensors and the actions are high-level behaviors (motions).

5.1 Decision Making Problem

In multiagent systems, the basic question that agents want answered is: “what should I do?”. To consider that an agent is intelligent, it must behave as such making the right decisions according to the state of its environment. However, in many real life cases, provide an agent with all the information needed to make the right decisions is very difficult. When designing agents often the specific circumstances that the agents will face at run time are unknown. This is the reason why designers often use learning agents. Agents might learn because they do not know everything about the environment or because they do not know how the other
agents behave.

This is consistent with the environment definition of RobotStadium, in which agents have a limited perception of the environment and they do not know how the other agents are acting. Also RobotStadium is a continuous environment in which perceptions and actions are continuous. This feature makes the mapping between states and actions even more complex and difficult the learning process. In addition, perceptions and actions in RobotStadium environment contains a random noise.

Finally, the problem is even bigger by increasing the number of learning agents in the environment. The agent learning depends on the actions of the others, thus it is complicated to know which agent must be punished or rewarded. These agents often face each other in encounters where the simultaneous actions of a set of agents leads to different utility payoffs for all the participants. There are two major categories of multiagent learning approaches. The first one, called team learning, applies a single learner to search for behaviors for the entire team of agents. Such approaches are more along the lines of traditional machine learning methods, but they may have scalability problems as the number of agents is increased [Alonso et al., a]. A second category of techniques, concurrent learning, uses multiple concurrent learning processes. Rather than learning behaviors for the entire team, concurrent learning typically employ a learner for each team member, in the hope that this reduces the joint space by projecting it into $n$ separate spaces. However, the presence of multiple concurrent learners makes the environment dynamic, which is a violation of the assumptions behind most traditional machine learning techniques [Alonso et al., a].

In multiagent systems, agents are able to act in their environment and they have different spheres of influence, they can have control over different parts of the environment. Sometimes, this spheres of influence may coincide and can give dependencies between the agents. The learning can happen in a cooperative environment where is desired that the agents share their learned knowledge, or in a competitive environment where is desired the best each other.

In cooperative scenarios each agent learns a map of its world and then, all agents share their maps in order to aggregate a global view of the environment and cooperatively decide their actions. The outcome of this auction will result in a utility gain or loss for all the agents. On the other hand, in competitive scenario each selfish agent tries to maximize its own utility by learning the other agents behaviors and weaknesses. The results of their combined actions have direct results in the utilities the agents receive from their actions.

In this thesis is implemented a RL approach for automatic generation of decision-making policies. In order to avoid the continuous state space problem, this thesis implements RL together with generalization. This technique is provided by the CQLF in which a neural network performs the generalization and replaces the tabular representation of Q-Learning. The objective is to generate the required decision-making policies through the experience of the agent in its environment.
5.2 Empirical Scenarios

Four experiments have been designed in order to test the hypotheses concerning the decision-making problem of this thesis. The first experiment consists of a simple learning task in which an agent must chase the ball. In this experiment is introduced a single agent approach for solving the decision-making problem. Then, second experiment consists of a more complex learning task in which a goal keeper agent must stop penalty shoots. In third experiment is introduced a multiagent approach for solving a decision-making policy. In this experiment an agent must learn to kick penalty shoots. Here the desired behavior (policy) is decomposed in two main sub task: positioning and shooting. Both sub-policies are learning at the same time in order to achieve a single global policy for shooting penalties. Finally, the most challenging experiment consist of a multiagent scenario in which a pair of teammates must get coordinated in order to chase the ball.

Unlike the experiments of the previous chapter (chapter 4) where 2 empirical scenarios were defined for all experiments, in this chapter is defined an empirical scenario for each experiment. This is because the generation of motion policies problem is more generic than the problem of generation of decision-making policies in which each specific problem has its peculiarities.

5.3 Chasing the Ball

Chasing the ball is a very representative soccer activity. Every soccer player must be capable of chasing the ball. In this thesis, this activity is classified as a high-level because implies the accomplishment of low-level activities which correspond to motion behaviors (chapter 4). Chasing the ball seems an easy task, but it involves a decision-making process in which an agent can decide to turn, walk, wait, and even search and track the ball. As it can be noticed the decision-making process is based on the information the agent has about the ball.

Chasing the ball activity involves many sub-activities to be performed. First the agent needs to retrieve information about the ball. To do this an agent needs to use its sensors, in this case, the camera sensor. Therefore an agent does not receive specific information about the ball, but it receives an RGB image. This image contains very useful information for the agent, but it must be processed first. From an image an agent can obtain information about its localization, the localization of other agents, and principally information about the ball.

Throughout this chapter, for the definition of experiments, many high-level tasks like image processing will be bypassed. For example for the empirical scenario for chasing the ball, state representation will be given by the location of the ball (distance and direction) and not by an image. Therefore the problem of chasing the ball is bounded only to a mapping from states to actions that are in essence the decision-making problem.

Chasing the ball is an easy but complete decision-making problem. In chasing the ball experiment is introduced a single agent approach for solving the decision-making problem using the CQLF for automatic generation of desired policies. Next section describes the empirical scenario for this experiment.
5.3.1 Single Agent Approach

The empirical scenario for the chasing problem uses a state representation composed by two float values indicating the relative position of the ball with respect of the agent. This two values consist of the ball distance (meters) and direction (radians). An important consideration is that state representation consists of two continuous measures, thus a generalization process must be performed in order to being capable of learn a mapping from states to actions.

For the actions and agent can perform, the empirical scenario has been simplified to three simple activities: turn left, turn right, and walk straight. Here is considered that the agent always have information about the ball (the agent is always seeing the ball). Also an automatic tracking of ball is performed every time step, therefore the ball is always in the center of the image perceived by the agent. This is achieved by setting the servos of head (yaw and pitch) pointing the camera towards the ball. Figure 5.1 shows a diagram explaining the empirical scenario for the chasing the ball experiment.

With this empirical scenario, the decision-making policy must map two inputs representing the state of the environment to three possible actions. The objective of the empirical scenario is to achieve a policy for chasing the ball so that the agent approaches the ball position facing it.

5.3.2 Design of Experiment

In order to experiment with the chasing the ball problem, it was designed a RobotStadium environment, which implements the described empirical scenario. This environment contains an agent that is randomly placed along the middle field line. The robot is facing the east goal in such away that is facing the ball placed at the penalty mark of that goal. With this configuration is ensured that the agent always start seeing the ball. However an agent can turn much and lose sight of the ball. Figure 5.2 shows an image of the RobotStadium environment for the chasing the ball experiment.

This environment also contains a supervisor which is encouraged to place the agent and the ball in their correct positions at the beginning of each training epoch. Also is encouraged to take care about the time. For this experiment, the agent has a minute for going to the ball position. When time expires, begins a new training epoch.

To allow the agent to learn the policy for chasing the ball is necessary to provide rewards
Figure 5.2: RobotStadium environment for chasing the ball experiment.

that tell the agent when it has done something good and punishment (negative reward) to indicate to the agent when it has done something wrong. Therefore for this experiment there are defined the following rewards:

- $-1$ If time expires.
- $-1$ If robot lose sight of ball.
- $-0.3$ If ball direction is greater than 0.25 rad or lower than $-0.25$.
- $1 - \frac{\text{ball distance}}{2}$ for rewarding or punishing according to distance. By reducing the distance to the ball increases the reward and to increase the distance to the ball decreases the reward.
- $1$ When the agent gets closest to the ball and facing it.

With these rewards, the agent is encouraged to be always facing the ball and get as close as possible to it. In addition the agent must hurry because if time is out it receives a negative reward. In addition to the rewards, the perception module of the agent is configured to receive two inputs for the distance and direction to the ball. The perception module is configured to use bipolar input signals as input for the neural network. Thus ball distance and direction are converted to bipolar sigmoidal functions as showed before. In order to properly differentiate between the possible states is necessary to carry out an escalation in the input values. For ball direction is used a scaling factor of 3 giving a sigmoid output range between $-0.91$ and $0.91$ for a maximum of $\pi$ and minimum of $-\pi$ radians. For ball distance, is used the formula $\text{distance} - 1 * 4$, giving a sigmoid output range between $-0.99$ and $0.99$ for a maximum of two and a minimum of zero meters. Recall that scaling is crucial in learning process because it allows to accurately distinguish between different states.

The brain module is configured to use the perception module described in previous paragraph and also to use three motion behaviors (walk forwards, turn left and turn right) as the possible actions to select. The neural network is configured with only one hidden layer.
containing five neurons. The exploration policy is set to $\varepsilon$-greedy with $\varepsilon = 10\%$ meaning that 10 of 100 times a random action selection will be performed in order to maintain an active exploration.

Finally, during experiments, parameters for the CQL algorithm will be tested with many values looking for those that show the quickest convergence.

5.3.3 Results

After many iterations, the values for the parameters of the QLF that showed the best behavior and quickest convergence are showed in table 5.1

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>5</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.99</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.2</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 5.1: CQLF configuration for the “chasing the ball” experiment.

Whit these parameters it can be noticed that in chasing th ball experiment the delayed reward does not play a role as important as in other experiments. This is primarily because every action carries a significant immediate result which is also immediately rewarded. For example if the robot turns and misses the ball or simply increases the direction angle, the agent is punished immediately. Likewise if the agent walks to the ball, it receives immediately reward even if is not fully completed the task. In the same way, forgetting rate ($\lambda$) is of little importance in this experiment.

To obtain the learning curve for this experiment there were run many trials, each one consisting of 200 training epochs. Figure 5.3 shows the average reward per training epoch for chasing the ball experiment.

The learning curve shows that convergence is reached after 120 training epochs. The learning curve converges to a maximum average reward of 0.6. The behavior showed by the plot indicates that the agent learned a policy to chase the ball. It would seem that the average reward obtained in the training (0.6) is far from the optimal average reward value (1). However, behavior showed in learning curve corresponds to training and therefore the policy has a constant component of exploration which sometimes takes a random action.

In order to test how good is the learned chasing policy, this must be configured to exploitation only, setting the exploration rate to 0. Also learned policy is compared to a custom chasing policy programmed in traditional way using if sentences to design the decision-making policy. The custom chasing policy is a simple rule described next:

```python
if(ball_direction > 0.25)
    turn_right
```
else if(ball_direction < -0.25)
    turn_left
else
    walk_forwards

For comparing both policies, it is necessary to obtain average reward for each one. For obtaining average reward for both policies there were run 200 training epochs. A comparison between average reward of both policies is showed in figure 5.4.

First, the plot shows that the exploitation mode of learned policy achieved a higher reward than the learning policy giving a final average reward of about 0.9, a value really close to optimal. Also comparing the learned policy in exploitation mode against the custom policy it can be noticed that both have almost the same behavior. Thus learned policy is as good as custom policy and both are really close to the optimal average reward.

5.4 Blocking Penalties

Blocking penalties is one of the most important high-level activity for goal keepers in soccer. Goal keepers must stop penalty shoots from enemy by blocking the ball to prevent an enemy goal. Blocking penalties is mainly based on jumping to block the ball motion, an spectacular and very useful ability. Also blocking penalties is crucial for the defense as the primary objective is that the ball does not enter in the defended goal.

Blocking penalties is classified as a high-level because implies the accomplishment of low-level activities which correspond to motion behaviors. Blocking penalties task involves
Figure 5.4: Comparison between learned and custom policies for “chasing the ball” experiment.

A decision-making process in which an agent can decide to stay, step to left, step to right, throw left and throw right. The decision-making process for blocking penalties is based on the information the agent has about the ball (direction and distance). As mentioned before, ball information is available to agents by an RGB image. However, for simplification purposes, the inputs to the CQLF learner are two float values indicating the ball direction and distance. With this, it is excluded the problem of image processing from the decision-making problem.

Blocking penalties is a decision-making problem in which goal keeper must decide rapidly the action to perform. In this problem the agent has a little time to decide the action because in penalties ball is positioned at only 1.2 m of the goal and the shooters kick the ball with force to increase their chances of success. In blocking penalties experiment is also used a single agent approach for solving the decision-making problem using the CQLF for automatic generation of policy. Next section describes the empirical scenario for this experiment.

5.4.1 Single Agent Approach

Blocking penalties task is an especial case of the decision-making problem in which is performed a single mapping from state to action in each epoch. This means that the agent only receives one representation of the state and executes one action based on that perception. This is due to the nature of the problem which requires that the agent performs a unique action very rapidly.

The empirical scenario for the chasing problem uses a state representation composed by three float values indicating the ball direction at three different times. The agent is representing the state by three readings of the ball direction in three consecutive time steps. Thus in
120 ms (40 ms per time step) the agent receives three different perceptions of ball direction an with this information the agent must learn what is the best action for blocking the penalty shoot. It is important to mention that the three readings of the ball direction are made immediately after the ball is hit by the penalty shooter. In this way, the state can provide information about the ball trajectory to the agent in order to take an appropriate action.

An important consideration is that state representation consists of three continuous measures, thus a generalization process must be performed in order to being capable of learn a mapping from states to actions. For the actions and agent can perform, the empirical scenario contains five simple activities: stay, step left, step right, throw left and throw right. With these simple actions the agent is able to fully cover its goal. Also is considered that the agent always have information about the ball (the agent is always seeing the ball). Therefore the blocking penalties task is abstracted to the mapping of a single state, represented by the ball trajectory, to a single action for intercepting the ball trajectory. Figure 5.5 shows a diagram explaining the empirical scenario for the blocking penalties experiment.

5.4.2 Design of Experiment

In order to experiment with the blocking penalties problem, it was designed a Robot-Stadium environment, which implements the described empirical scenario. This environment contains a goal keeper agent that is placed in the middle of the blue goal. The agent is facing the ball placed at the penalty mark. For convenience there is no a shooting agent. Instead, the environment simulates a shooter by moving the ball towards the goal with random direction and velocity. This is with the aim of achieving a smoother and faster simulation, avoiding the programming of a shooter agent and also achieving a better control about the direction and speed of shots. Figure 5.6 shows an image of the RobotStadium environment for the blocking penalties experiment.

This environment also contains a supervisor which is encouraged to place the agent and the ball in their correct positions at the beginning of each training epoch. Also is encouraged to take care about the shoots. The supervisor calculates randomly the ball direction and speed in every training epoch. In order to allow the agent to learn the policy for blocking penalties is necessary to provide rewards that tell the agent when it has done something good.
and punishment (negative reward) to indicate to the agent when it has done something wrong. For this experiment, rewards are really simple:

- $-1$ If goal keeper receives a goal.
- $1$ If goal keeper blocks the penalty shoot.

With these simple rewards, the goal keeper agent is simply encouraged to block the ball. In addition to the rewards, the perception module of the agent is configured to receive three inputs for the ball direction. The perception module is configured to use bipolar input signals as input for the neural network. Thus ball direction is converted to bipolar sigmoidal functions as showed before. In order to properly differentiate between the possible states is necessary to carry out an escalation in the input values. For the 3 inputs of ball direction is used a scaling factor of 130 giving a sigmoid output range between $-0.99$ and $0.99$ for a maximum of $0.035$ and minimum of $-0.035$ radians (between $2$ and $-2$ degrees). The scaling factor is big because the ball direction in this experiment has a small range. This is because the perceptions of ball direction are made almost immediately the ball is hit. However, the three perceptions of ball direction are enough to detect the ball trajectory and still have enough time to perform the action to block the ball.

The brain module is configured to use the perception module described in previous paragraph and also to use five motion behaviors (stay, step left, step right, throw left and throw right) as the possible actions to select. The neural network is configured with only one hidden layer containing five neurons. The exploration policy is set to $\varepsilon$-greedy with $\varepsilon = 10\%$ in order to maintain an active exploration.

Finally, during experiments, parameters for the CQL algorithm will be tested with many values looking for those that show the quickest convergence. Due to the nature of the blocking penalties task, it is known that there are no delayed reward and neither is considerable the forgetting rate.
5.4.3 Results

After many tests with blocking the ball experiment, the values for the parameters of the CQLF that showed the best behavior and quickest convergence are showed in table 5.2. Whith these parameters is checked that in blocking penalties experiment the delayed reward does not have any sense since only one action is executed every epoch. In this way, every action is immediately rewarded. In the same way, forgetting rate ($\lambda$) does not have importance in this experiment. To obtain the learning curve for this experiment there were run many trials, each one consisting of 500 training epochs. Figure 5.7 shows the average reward per training epoch for blocking penalties experiment.

The learning curve shows that convergence is reached just after 160 training epochs. Te learning curve converges to a maximum average reward of 0.4. The behavior showed by the plot indicates that the agent learned a policy to block penalties. Apparently average reward is far from the optimal. However, recall that learning curve corresponds to training and therefore

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>5</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>1</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.0</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 5.2: CQLF configuration for the “blocking penalties” experiment.

![Learning curve for blocking penalties experiment](image-url)
the policy has a constant component of exploration. In order to test how good is the learned blocking policy for blocking penalties, this is compared to a custom blocking penalties policy programmed in traditional way using if sentences to design the decision-making policy. The custom blocking penalties policy is a simple rule described next:

```python
if (ball.direction > 0.03)
    throw_right
else if (ball.direction >= 0.005)
    step_right
else if (ball.direction < -0.03)
    throw_left
else if (ball.direction < -0.005)
    step_left
else
    stay
```

For comparing both policies, it is necessary to obtain average reward for each one. Learned policy was configured to exploitation only, setting the exploration rate to 0. In this way, there was obtained a real average reward using the learned policy. For obtaining average reward for both policies there were run 200 epochs. A comparison between average reward of both policies is showed in figure 5.8.

This comparison shows that learned policy in exploitation mode improves its average reward from 0.4 to 0.7. However, average reward in exploitation mode is not too close to
the optimal. Comparing learned policy in exploitation mode against the custom policy can be noticed that both have a similar performance, even the learned policy has slightly higher performance than the custom policy. According to the obtained results, can be noticed that it is difficult to generate near-optimal policies for the problem of blocking penalties.

5.5 Shooting Penalties

As a counterpart to the above experiment, shooting penalties is one of the most important high-level activities for field players in soccer. Shooting penalties is a very important task because players have a great chance to score a goal. However, shooting penalties task requires good technique by the shooter and also involves a decision-making problem. The decision-making process for shooting penalties implies decision about which leg use to kick, the alignment of the player with the ball, the force of the kick, the desired direction, etc.

Many times real life soccer players choose randomly the direction and the force to hit the ball and also players generally kicks only with one leg. Therefore real life soccer players are more concerned about the precision in shooting penalties. They train a lot in order to perform shoots with the desired force and direction. On the other hand, in robotic soccer, players needs to decide their alignment with respect to ball and then which leg to use according to that alignment. Most robotic soccer players are trained for shooting with enemy goal direction. More advanced players also try to make more precise shoots concerning direction. In this thesis, shooting penalties problem will be addressed giving more importance to the positioning and also the shooting decision (right or left leg). Shooting direction is also an important feature which will be covered in this experiment trough the positioning.

Shooting penalties is classified as a high-level because implies the accomplishment of low-level activities which correspond to motion behaviors. Shooting penalties task involves a decision-making process in which an agent can decide to step to left, step to right, shoot with left leg and shoot with right leg. The decision-making process for shooting penalties is also based on the information the agent has about the ball (direction and distance).

In shooting penalties experiment there are implemented two different approaches for solving the decision-making problem. First approach is the single agent approach used in previous experiments. Second approach, consists of an splitting of the global task into subtasks and therefore the global decision-making problem is converted to many decision-making problems. Then, this approach is suitable for a multiagent learning process for solving the individual learning problem and consequently the global decision-making problem.

In the single agent approach, the learning agent is encouraged to learn the complete mapping function from state representation to all possible actions the agent can perform. On the other hand, multiagent approach splits the global task into subtasks where the subtasks implies simpler mappings between states and actions than the global task. Therefore the global decision-making policy is conformed by separate decision-making policies learned by separate learning agents. Next section describes the empirical scenario for these two approaches.
5.5.1 Single Agent Approach

Single agent scenario for shooting penalties uses a state representation composed only by the ball direction. The state representation is simplified by removing the ball distance value, considering that the player is always positioned along a line that is separated 15 cm from the ball. The actions that the agent can perform are: step to left, step to right, shoot with left leg and shoot with right leg. Figure 5.9 shows a diagram explaining this empirical scenario.

With this empirical scenario, the agent is able to get aligned correctly with the ball and also to perform a shoot with the correct leg. Moreover, the agent is capable to implement certain direction to the shoot by alignment with the ball.

5.5.2 Multiagent Approach

Multiagent empirical scenario for shooting penalties uses two separate learning agents. The first agent is encouraged to learn the positioning policy whereas the second agent is encouraged to learn the shooting policy. The positioning policy implies lateral steps for getting aligned with the ball. Thus possible actions for the positioning policy learner are: wait, step to the left and step to the right. The shooting policy implies to wait until the robot is correctly positioned and then decide to shoot with the correct leg. Thus possible actions for the shooting policy learner are: wait, shoot with left leg and shoot with right leg. Both policies receive ball direction as input for state representation. Figure 5.10 shows a diagram explaining this multiagent empirical scenario.

In multiagent empirical scenario both learning agents are training at the same time. Recall that the wait action in both policies is very important to achieve the desired behavior for shooting penalties. By wait action both policies get coordinated. The shooting policy is encouraged to wait until the positioning policy is done whereas positioning policy can wait for the shoot. Also, it is important to recall that both learning agents are running in the same robot (process). Next section (5.6) show an experiment in which two different robots are learning at the same time and running its own learning process each one.

As considerations for both empirical scenarios, the input values for state representation (ball direction) are continuous. Also is considered that the agent always have information about the ball.
5.5.3 Design of Experiment

In order to experiment with the shooting penalties problem, it was designed a RobotStadium environment, which implements the two described empirical scenarios. The environment contains a filed player agent which is random positioned along a line which is separated 15 cm from the penalty mark. The agent is facing the ball placed at the penalty mark. Also this environment contains a goal keeper that is placed in the middle of the blue goal. This dummy goalkeeper does nothing in the environment (i.e. has no controller program). The purpose of this robot is that the shooter agent learns to shoot penalties with direction to the sides of the goalkeeper. Thus the penalty shoot is more likely to become a goal. Put another way, it avoids to learn a policy for shooting penalties through the center of the goal. Figure 5.11 shows an image of the RobotStadium environment for the shooting penalties experiment.

This environment also contains a supervisor which is encouraged to place the shooting agent, the goal keeper robot and the ball in their correct positions at the beginning of each training epoch. Also supervisor determines if the penalty shoot becomes a goal or not and also updates the game score. To allow the shooting agent to learn the policy for shooting penalties is necessary to provide rewards that tell the agent when it has done something good or has done something wrong. For this experiment, following simple rewards are defined:
• -1 If shooter agent misses the goal.
• 1 If shooter agent scores a goal.

With these simple rewards, the shooter agent is encouraged to score a goal in every penalty shoot. Due to the random position at the beginning of training epoch, shooter agent is encouraged to position properly and then shoot the penalty with the correct leg. In this activity positioning is very important because allows to perform the shoot and also it determines the direction towards the goal.

In addition to the rewards, the perception module of the agent is configured to receive the ball direction every time step. The perception module is configured to use bipolar input signals as input for the neural network. Thus ball direction is converted to bipolar sigmoidal functions. In order to properly differentiate between the possible states is necessary to carry out an escalation in the input values. For the ball direction is used a scaling factor of 8 giving a sigmoid output range between $-0.98$ and $0.98$ for a maximum of $0.5$ and minimum of $-0.5$ radians.

The brain module for the single agent empirical scenario is configured to use 4 motion behaviors (step left, step right, shoot with left leg and shoot with right leg) as the possible actions to select. The neural network is configured with only one hidden layer containing 3 neurons. The exploration policy is set to $\varepsilon$-greedy with $\varepsilon = 10\%$ in order to maintain an active exploration.

In multiagent empirical scenario, both learning agents uses the same perception module, thus both policies have the same input (ball direction) and the same rewards as the single agent empirical scenario. The brain module for the positioning policy is configured to use 3 motions behaviors (wait, step left and step right) whereas the shooting policy is configured with three possible actions (wait, shoot with right leg and shoot with left leg). The neural network of both policies is configured with only one hidden layer containing 3 neurons. The exploration policy is set to $\varepsilon$-greedy with $\varepsilon = 10\%$.

Finally, during experiments, parameters for the CQL algorithm will be tested with many values looking for those that show the quickest convergence.

### 5.5.4 Results

Single agent approach proved to be complicated to configure and poor results were obtained with many configurations. After many iterations, the values for the parameters of the QLF that showed the best behavior and quickest convergence in single agent scenario are showed in table 5.3.

As can be seen, the task requires taking much into account delayed reward as only is received after scoring or missing the penalty kick. Whit parameters defined in 5.3 can be noticed that delayed reward and forgetting rate play an important role for the single agent approach. This is primarily because reward is received only after shoot actions. Thus the learning process is based on the terminal actions.

The values for the parameters of the QLF that showed the best behavior and quickest convergence in multiagent scenario are showed in 5.4.
Recall that parameters are configured separately for both subtasks in multiagent scenario. It can be noticed that for positioning subtask, delayed reward is important since lateral steps do not receive immediate reward whereas in shooting task delayed reward is not important since shooting actions receive immediate reward. There were run many trials in order to obtain the learning curve for both scenarios. Every trial consisted of 300 training epochs. Figure 5.12 shows the average reward per training epoch for both scenarios in shooting penalties experiment.

Learning curves for both approaches had a similar performance, Multiagent approach slightly outperformed the single agent approach finishing with an average reward close to -.1 against -.2 of the single agent approach. This final average rewards are really far from optimal, showing that the learning task is complex. However both approaches were improving over the training epochs showing a final convergence. These learning curves shows that the multiagent approach simplified the learning task by splitting it into simpler subtasks.

In order to test how good are the learned policies, these are compared against to a custom “shooting penalties” policy programmed in traditional way using if sentences. The custom chasing policy is a simple rule described next. Recall that ball direction is given in radians.

```python
if (ball_direction >= 0 and ball_direction < 0.35)
    step_left
else if (ball_direction < 0 and ball_direction > -0.35)
    step_right
else if (ball_direction >= 0)
    shoot_right_leg
```
else if (ball_direction < 0 )
    shoot_left_leg

For comparing learned polices against the custom policy, is necessary to obtain average reward for each policy. In case of two learned policies, exploration rate is set to 0 (i.e setting exploitation mode). In this way, there can be obtained a real average reward values using the learned policies. Also there were run 300 training epochs for obtaining average reward. A comparison between average reward of both learned policies and the custom policy is showed in figure 5.13.

This comparison shows that policy learned with multiagent approach is clearly better than the single agent approach policy. Although the learning curves of both approaches had a similar performance, learned policies are totally different especially when they are set to exploitation only. Comparison show that multiagent approach is almost as good as custom policy obtaining an average reward very close to the optimal. However in “shooting penalties” experiment, custom policy achieved a better result than the learned policies. This demonstrates the complexity involved in learning the task.

5.6 Coordination for Chasing the Ball

Chasing the ball in a coordinated way is an essential capability of soccer players in a team. If all the robots of a team chase the ball at the same time would have a lot of interference and collisions between teammates. This problem is greater by increasing the
number of agents in teams. Also is important to be efficient in chasing the ball as the less work is done for accomplish the task. In the simplest scenario, generally is better the closest agent to ball performs the chasing. Therefore coordination for chasing the ball is probably the most important coordinated decision-making task in robotic soccer.

In the decision-making process for chasing the ball an agent can decide to wait or to chase the ball. This decision-making process is based on the information the agent has about the relative ball localization. The wait action is the key for coordination, it prevents interferences and collisions between teammates and is crucial in the efficiency of the task by assigning the chasing task to the closer agent to the ball.

In previous chasing experiment, agent only perceives the distance and direction to the ball, thus agent just can decide which route to take for chasing the ball. However, in order to achieve coordination is necessary that the agents have own information about ball, and also the same information from other agents.

In "coordinated chasing the ball" experiment there is implemented a multiagent approach for solving the decision-making problem. This experiment is an extension to the "chasing the ball" experiment described in section 5.3. Actually players in this experiment are very similar to the individual agent in "chasing the ball" experiment. Next section describes the empirical scenario for the multiagent approach for solving this coordinated decision-making problem.
5.6.1 Empirical Scenario

In "coordinated chasing the ball" experiment agents use emitter and receiver devices for sharing their own ball information. Specifically in RobotStadium environment agents have access to simulated radio emitter and receiver devices. Through these devices, agents can communicate among themselves and also an agent can exchange messages with the supervisor. Every time step an agent can send a broadcast message to all teammates and also an agent can receive many messages from various agents. The message consists of a character string in ASCII format that can contains any information coded.

The empirical scenario for this experiment is different from others since it contains two learning agents, each one running in different robot (i.e. learning agents run in separate processes). Both learning agents control its own robot in order to chase the ball in a coordinated way. This empirical scenario for the "coordinated chasing the ball" experiment uses a state representation composed by two float values indicating the own ball distance and also the ball distance from teammate. These input values consists of continuous measures.

For the actions and agent can perform, the empirical scenario has been simplified to two activities: wait and chase the ball. Here is considered that the agent always have information about the ball and also an automatic tracking of ball is performed every time step. Chase the ball action is a high-level behavior which implies the execution of one of three possible motions every time: walk forwards, turn left and turn right. In this way the policy is just encouraged to map the own and teammate ball information to the two possible actions. Figure 5.14 shows a diagram explaining the empirical scenario for the "coordinated chasing the ball" experiment.

With this empirical scenario, the objective is to achieve two chasing policies that can
work together for achieving coordination in that task. Recall that policies can be different since each one is a different learning process. In this empirical scenario the key factor is to wait if the other is chasing the ball or chase the ball if the other is waiting. Also collisions and interferences should be prevented.

### 5.6.2 Design of Experiment

In order to experiment with the “coordinated chasing the ball” problem, it was designed a RobotStadium environment, which implements the described empirical scenario. This environment contains two robots, each one controlled by a learning agent. Both robots are placed randomly along two parallel lines at middle field. Both robots are facing to the east goal in such away that is facing the ball, placed at the penalty mark of that goal. With this configuration is ensured that the robots always start seeing the ball. Figure 5.15 shows an image of the RobotStadium environment for the chasing the ball experiment. Red lines represent the possible initial locations of the robots.

This environment also contains a supervisor which is encouraged to place the robots and the ball in their correct positions at the beginning of each training epoch. The agent sets both robots in valid positions, thus there are not collisions neither interferences at the beginning of training epoch. Also, supervisor is encouraged to take care about the time, robots has a minute for going to the ball position.

To allow the agents to learn the policy for chasing the ball in a coordinated way is necessary to provide rewards that tell the agents when they has done something good and punishment (negative reward) to indicate when they has done something wrong. Therefore for this experiment there are defined the following rewards:

- $-1$ If time over.
- $-1$ If the distance between the robots is less or equal to 30 cm. There is a collision.
- $-0.6$ If the distance between the robots is less or equal to 60 cm. There is interference between robots.

Figure 5.15: RobotStadium environment for “coordinated chasing the ball” experiment.
Table 5.5: CQLF configuration for the “coordinated chasing the ball” experiment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>neurons in hidden layers</td>
<td>5</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.2</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.5</td>
</tr>
</tbody>
</table>

- 1 If any robot is close to ball and facing it.

With these rewards, agents are encouraged to be always chasing the ball and get as close as possible to it but taking care about interferences and collisions. In addition the agent must hurry because if time is out it receives a negative reward. In addition to the rewards, the perception module of the agent is configured to receive two inputs corresponding to the ball distance information of both agents.

The perception module is configured to use bipolar input signals as input for the neural network. Thus distances are converted to bipolar sigmoidal functions. In order to properly differentiate between the possible states it is necessary to carry out an escalation in the input values. For both input signals is used the following formula for scaling: $ball_{distance} \times 4 - 6$. This scaling formula results in a sigmoid output range between $-0.99$ and $0.99$ approximately. Distance to ball is ranging from 0 to 3 meters. Recall that scaling is crucial in learning process because it allows to accurately distinguish between different states.

The brain module is configured to use the perception module described in previous paragraph and also to use 2 possible actions to select: wait and chase the ball. The neural network is configured with only one hidden layer containing 3 neurons. The exploration policy is set to $\varepsilon$-greedy with $\varepsilon = 10\%$. Perception and Brain modules of both robots will be configured in the same way as described before. However, resulting learned policies may be different.

Finally, during experiments, parameters for the CQL algorithm will be tested with many values looking for those that show the quickest convergence.

5.6.3 Results

After many tests with “coordinated chasing the ball” experiment, the values for the parameters of the CQLF that showed the best behavior and quickest convergence are showed in table 5.5.

With these parameters is checked that in “coordinated chasing the ball” experiment the delayed reward plays an important role since reward different to 0 is not received every time. In this way, many actions are not immediately rewarded. In the same way, forgetting rate ($\lambda$) is important in this experiment. To obtain the learning curve for this experiment there were run many trials, each one consisting of 200 training epochs. Figure 5.16 shows the average reward per training epoch for “coordinated chasing the ball” experiment.
The learning curve shows that convergence is reached really quick just after 100 training epochs. The learning curve converges to a maximum average reward of 0.7. The behavior showed by the plot indicates that both agents learned policies in order to chase the ball in a coordinated way. Learning curve shows that average reward is close to the optimal. However, recall that learning curve corresponds to training and therefore the policy has a constant component of exploration.

In order to test how good are the learned policies, these are compared to a custom policy programmed in traditional way using if sentences to design the decision-making policies for both robots. The custom policy is a simple rule described next:

```python
if (ball_distance < teammate_ball_distance)
    chase the ball
else
    wait
```

For comparing learned policies against custom policy, is necessary to obtain average reward for each one. Learned policies were configured to exploitation only, setting the exploration rate to zero. In this way, there was obtained a real average reward using the learned policies. For obtaining average rewards there were run 200 epochs. A comparison between average reward of learned and custom policies is showed in figure 5.17.

This comparison shows that learned policies in exploitation mode had a better performance than with exploration mode. Average reward in exploitation mode is really close to the optimal. Comparing learned policies in exploitation mode against the custom policy can...
be noticed that both have a similar performance, even the custom policy has slightly higher performance than the custom policy.

5.7 Summary

This chapter defined the problem of the generation of decision-making policies within the RobotStadium environment. First it was defined the concept of decision-making policy and the process of mapping states to actions. Then, it was described the problem of generating decision-making policies and also was shown how these policies can be automatically generated by RL techniques. Subsequently, it was defined how generalization can tackle learning problems involving continuous state spaces. Moreover, this chapter details the use and configuration of the Connectionist Q-Learning (CQL) algorithm by means of Connectionist Q-Learning Framework (CQLF) for the implementation of RL plus generalization in the problem of generating decision-making policies.

In this chapter there were designed four experiments in order to experiment with decision-making policies. The four experiments are: “chasing the ball”, “blocking penalties”, “shooting penalties” and “cooperative chasing the ball”. In the first experiment is introduced a single agent approach for solving the decision-making problem. In second experiment, is also used the single agent approach but this time the required policy is harder and highly prone to failure due to noisy perceptions. In third experiment is introduced the multiagent approach for solving the decision-making problem and is compared to the single agent approach. Here,
the decision-making policy is complex and can be split into subtasks in order to implement multiagent learning. Finally, the fourth experiment is a purely multiagent decision-making problem in which the agents need to get coordinated for chasing the ball. In this experiment, every learning agent runs separately in distinct robots. In order to achieve coordination, agents share localization information through communication.

For each of the four experiments, it was defined the specific decision-making problem, the empirical scenario (single and multiagent), the design of the experiment (RobotStadium environment and configurations) and the obtained results. Results show that single agent approach is able to learn decision-making policies and even behave as well as a custom pre-programmed decision-making policy (if-then-else). Moreover, single agent learning is able to learn decision-making policies in noisy environments. In tasks highly prone to failure by noise, single agent approach outperforms custom decision-making policies.

On the other hand, multiagent approach showed good results solving decision-making problems. It is showed that the multiagent approach is very useful when decision-making policies are very complex and can be divided into simpler parts. Thus the multiagent approach is able to solve the global problem by learning simpler policies. Using this approach, a learning agent can be modeled as a group of simple interacting agents with simpler state spaces as well as simpler action spaces. Moreover, multiagent approach showed that it is able to learn purely multiagent decision-making tasks in which coordination and cooperation are required.
Chapter 6
Conclusions

This thesis has showed that it is possible to automatically generate motion and decision-making policies in continuous and dynamic environments through reinforcement learning as long as there is a method of generalization. Moreover, this research work proposes a framework for addressing problems of generating motion and decision-making policies using single and multiagent approaches in simulated robotic soccer.

For the problem of generating motion policies the multiagent approach overcame the single agent approach. It was noticed that the multiagent approach is a more suitable alternative for the problem given its nature of parallelism. With the multiagent approach agents can be modeled as a group of simple interacting agents with single state spaces (partial observations of states) as well as single action spaces (individual action selections). Despite the simplicity in the design of single agent approach, the global mapping between states and actions is much more complex than the individual mappings in multiagent approach. Although single-agent approach is significantly overcame, it was shown how this approach is also able to learn the motion policies through the simulation of a parallel execution performed by a sequential execution of actions from actuators.

For the problem of generating decision-making policies, it was concluded that at least the learned policies are as good as custom decision-making policies. Moreover learned policies also have the advantage of being flexible to noisy perceptions, because the agent learns from experience in the noisy environment. Custom policies on the other hand are rigid and prone to failure by noise. It was concluded that in tasks highly prone to failure by noise, single agent approach outperforms custom decision-making policies. Multiagent approach showed good results solving decision-making problems. It was showed that the multiagent approach is very useful when decision-making policies are very complex and can be divided into simpler parts. Thus the multiagent approach is able solve the global problem by learning simpler policies. Moreover, multiagent approach showed that it is able to learn purely multiagent decision-making tasks in which coordination and cooperation are required.

6.1 Contributions

The main contribution of this thesis is the proposed framework for addressing problems of generating motion and decision-making policies using single and multiagent approaches in the "RobotStadium" environment. This research work is the first attempt to solve the problem
of generating motion and decision-making policies using a reinforcement learning technique in the “RobotStadium” environment. The proposed framework uses the Connectionist Q-Learning algorithm which combines a reinforcement learning technique (Q-Learning) with a generalization method (backpropagation neural network). However, the key of the contribution are the proposed approaches for implementing the algorithm. This thesis proposed two different approaches: single agent and multiagent.

For generating motion policies, the main contribution is the multiagent approach in which every servo motor learns its own motion policy. In this way, individual policies are simpler than the global policy and by learning them it is possible to achieve the global desired motion. The multiagent approach is a very suitable technique for generating motion policies mainly for their inherent parallelism. Also the multiagent approach is very simple given that individual agents receive the same global rewards and state representation, focusing only on individual actions regarding its own servo motor.

For generating decision-making policies, the single agent approach is an important contribution in which learning agents are able to learn simple strategies resulting in good quality policies able to work in noisy environments. Moreover, this thesis contributes a multiagent approach for generating complex decision-making policies by splitting them into smaller and easy to learn sub-policies. Finally, a very interesting contribution is the multiagent approach for achieving coordination in a decision-making task in which learning agents share information to accomplish the common goal.

The main practical contribution is the improvement of the “Borregos” team from Tecnológico de Monterrey, Campus Monterrey University in the RobotStadium contest as well as the Standard Platform League of RoboCup. This thesis contributed to the generation of basic motion policies like shooting and throwing as well as decision-making policies for simple basic game strategies like blocking and shooting penalties. However, the proposed framework is able to generate even more motion and decision-making policies for the improvement of “Borregos” team.

6.2 Future Work

There are many possibilities for future work, principally addressing problems involving more complex policies and a greater number of interacting agents. In the case of motion policies, future work can address more complex motions like walking and getting up policies which require more precise control and many servos playing at the same time. The proposed solution method can be extended to support continuous action spaces instead a discrete number of actions. In this case, action space could represent the positions of servos by taking the outputs of the neural network as the position of servos. Thus the number of output neurons must be equal to the number of servos needed for playing the motion.

On the other hand, regarding the high-level decision-making problem, the proposed solution method can address more complex high-level strategies with the involvement of a greater number of agents. There are many possibilities for future work due to a great variety of approaches to implement multiagent learning like cooperative, competitive, team learning and
concurrent learning. However, it is important to assess to what extent the proposed solution method is able to tackle more complex problems without the need to use some other multi-agent coordination techniques like negotiation, voting, contract net, etc.

Moreover, this framework can be implemented and tested on the real Nao. Thus, it is possible to generate policies using physical robots or generate policies in simulation for further implementation in real robots. Finally, the framework also can be implemented in different simulated robotic soccer leagues like RoboCup 2D and 3D simulated leagues.
I order to show a fully CQLF implementation consider the following example. An agent is placed in a Grid-World of m rows and n cols. The agent must learn to arrive to a desired cell (finalM, finalN) in the grid. The perception of the agent is conformed by the current values for m and n. If the agent arrive to the final cell, the reward is 1. If the agent moves out of the grid the reward is -1, otherwise reward is 0. The custom Perception class for this agent can be:

```java
public class MyPerception extends Perception{

    private Agent agent;

    public MyPerception(Agent agent) {
        this.agent = agent;
    }

    public boolean isUnipolar() {
        return true;
    }

    public double getReward() {
        double r = 0;
        if(player.m == finalM && player.n == finalN)
            r = 1;
        if(player.m > limitM || player.n > limitN)
            r = -1;
        return r;
    }

    protected void updateInputValues() {
        setValue(player.m);
        setValue(player.n);
    }
}
```
The agent can move to the north, south, east and west. A custom Action class for moving to the north can be:

```java
public class goNorth extends Action{
    private Agent agent;

    public MyPerception(Agent agent) {
        this.agent = agent;
    }

    public int execute() {
        player.m = player.m - 1;
        return 0; // exit code
    }
}
```

Every training epoch finishes when the agent gets a reward different from 0 (get out of grid or success). The class for the agent can be defined as follows:

```java
import pl.gdan.elsy.qconf.Action;
import pl.gdan.elsy.qconf.Brain;
import pl.gdan.elsy.qconf.Perception;

public class Agent{

    private Brain brain;

    public Agent(){
        Action actionArray[] = new Action[4];
        actionArray[0] = new goNorth(this);
        actionArray[1] = new goSouth(this);
        actionArray[2] = new goEast(this);
        actionArray[3] = new goWest(this);

        Perception perception = new MyPerception(this);
        brain = new Brain(perception, actionArray);
    }

    public configure(){
        brain.setAlpha(0.9); // learning rate
        brain.setGamma(0.9); // discount factor
        brain.setLambda(0.2); // eligibility factor
        brain.setRandActions(10); // 1 to 100
    }
}
```
epochs = 30; //training epochs
}

public run()
{
    for( int i = 0 ; i < epochs ; i++ )
    {
        runEpoch();
    }
}

private void runEpoch()
{
    reset();
    perception.perceive();
    while( perception.getReward()==0 )
    {
        brain.count();
        brain.executeAction();
        perception.perceive();
    }
    brain.count();
}

Bibliography


