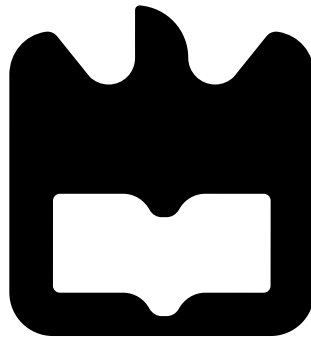




**Rui
Serra**

**Aprendizagem Automática de Comportamentos
para Futebol Robótico**

Automated Behavior Learning for Robotic Soccer





**Rui
Serra**

Aprendizagem Automática de Comportamentos para Futebol Robótico

Automated Behavior Learning for Robotic Soccer

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de Prof. Doutor Nuno Lau e Prof. Doutor Luís Seabra Lopes, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutora Ana Maria Perfeito Tomé

Professora Associada da Universidade de Aveiro (por delegação da Reitoria da Universidade de Aveiro)

vogais / examiners committee

Doutor José Nuno Panelas Nunes Lau

Professor Auxiliar da Universidade de Aveiro (orientador)

Doutor Francisco Saraiva de Melo

Professor Auxiliar da Universidade Técnica de Lisboa - Instituto Superior Técnico

agradecimentos /
acknowledgements

Este trabalho não seria possível sem a ajuda e apoio dos meus orientadores, professores Nuno Lau e Luís Seabra Lopes, e do João Cunha. A eles o meu obrigado. Quero também agradecer aos meus pais, António Luís e Maria do Carmo, pelo apoio incondicional que me sempre me deram durante todos estes anos. Por último, à minha namorada, Zsuzsanna, que sempre teve palavras de motivação, e pela sua paciência incansável.

This work would not be possible without the help and support of my coordinators, professors Nuno Lau and Luís Seabra Lopes, and of João Cunha. My thanks to them. I would also like to thank my parents, António Luís and Maria do Carmo, for their relentless support throughout all these years. Last but not least, to my girlfriend, Zsuzsanna, who always had words of motivation, and for her enduring patience.

Resumo

Um robô futebolista necessita de executar comportamentos variados, desde os mais simples aos mais complexos e difíceis. Programar manualmente a execução destes comportamentos pode tornar-se uma tarefa bastante morosa e complicada. Neste contexto, os métodos de aprendizagem automática tornam-se interessantes, pois permitem a aprendizagem de comportamentos através de uma especificação a muito alto nível da tarefa a aprender, deixando a responsabilidade ao agente autónomo de lidar com os detalhes.

A Aprendizagem por Reforço toma inspiração na natureza e na aprendizagem animal para modelar agentes que interagem com o seu ambiente de forma a escolherem as acções que aumentam a probabilidade de receberem recompensas e evitarem castigos. À medida que os agentes experimentam acções e observam os seus efeitos, ganham experiência e a partir dela derivadam uma política. Isto é feito após cada observação do efeito de uma acção, ou após reunir conjuntos destas observações. Esta última alternativa, também chamada Aprendizagem por Reforço Batch, tem sido usada em aplicações reais com resultados promissores.

Esta tese explora o uso de Aprendizagem por Reforço Batch para a aprendizagem de comportamentos para futebol robótico, tais como driblar a bola e receber um passe. Os resultados presentes neste documento foram obtidos de experiências realizadas com o simulador da equipa CAMBADA, assim como com os seus robôs.

Abstract

A soccer-playing robot must be able to carry out a set of behaviors, whose complexity can vary greatly. Manually programming a robot to accomplish those behaviors may be a difficult and time-consuming process. Automated learning techniques become interesting in this setting, because they allow the learning of behaviors based only on a very high-level description of the task to be completed, leaving the details to be figured out by the learning agent.

Reinforcement Learning takes inspiration from nature and animal learning to model agents that interact with an environment, choosing actions that are more likely to lead them to accumulate rewards and avoid punishment. As agents experience the environment and the effect of their actions, they gain experience which is used to derive a policy. Agents can do this instantaneously after they observe the effect of their last action, or after collecting batches of these observations. The latter alternative, called Batch Reinforcement Learning, has been used in real world applications with very promising results.

This thesis explores the use of Batch Reinforcement Learning for learning robotic soccer behaviors, including dribbling the ball and receiving a pass. Practical experiments were undertaken with the CAMBADA simulator, as well as with the CAMBADA robots.

Contents

Contents	i
List of Figures	iii
List of Tables	vii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Thesis Outline	2
2 Robotic Soccer	3
2.1 Introduction	3
2.2 RoboCup	5
2.3 The CAMBADA Platform	10
2.3.1 Hardware	10
2.3.2 Software	13
2.4 Summary	16
3 Reinforcement Learning	17
3.1 Introduction	17
3.2 Markov Decision Processes	18
3.3 Value Iteration	19
3.4 Q-Learning	22
3.5 Fitted Q-Iteration	23

3.6	Neural Fitted Q-iteration	26
3.7	Q-Batch update rule	27
3.8	Applications in Robotic Soccer	30
4	Learning Behaviors with the CAMBADA robots	33
4.1	Introduction	33
4.2	A Reinforcement Learning framework for CAMBADA	33
4.3	Learning in a simulated environment	35
4.4	Learning in a real environment	36
4.5	Rotate to an absolute position in the field	36
4.5.1	Overview	36
4.5.2	Behavior specification	37
4.5.3	Learning procedure	39
4.5.4	Learning results in a simulated environment	39
4.5.5	Learning results in a real environment	40
4.5.6	Comparison between Q-Learning and Q-Batch update rules	41
4.6	Dribble the ball in a given direction	42
4.6.1	Overview	42
4.6.2	Behavior specification	42
4.6.3	Learning procedure	43
4.6.4	Learning results in a simulated environment	44
4.6.5	Learning results in a real environment	46
4.7	Receive a Pass	48
4.7.1	Overview	48
4.7.2	Behavior specification	48
4.7.3	Learning procedure	50
4.7.4	Learning results in a simulated environment	51
4.7.5	Learning results in a real environment	53
5	Conclusion	57
5.1	Future work	58

List of Figures

1.1	Timeline of CAMBADA’s evolution.	2
2.1	The RoboCup project’s logo, taken from the RoboCup website.	5
2.2	A Middle Size League soccer game situation.	6
2.3	A Small Size League soccer game.	7
2.4	Examples of game situations from both Simulation leagues.	8
2.5	A Standard Platform League game example.	8
2.6	Pictures from games of the three Humanoid leagues.	9
2.7	A CAMBADA robot.	11
2.8	Mechanical drawings of the CAMBADA robots detailing their modular and layered structure, adapted from [2].	11
2.9	Holonomic drive.	12
2.10	The grabber and kicker system.	12
2.11	The omnidirectional vision system.	13
2.12	Harware architecture with functional mapping, adapted from [3].	13
2.13	Biomorphic architecture, adapted from [4].	14
2.14	Layered software architecture, adapted from [5].	14
3.1	Interaction between the agent and the environment in a reinforcement learning setting, adapted from [6].	17
3.2	The Batch Reinforcement Learning framework, depicting the interconnections of the various modules, taken from [10].	25
4.1	Class diagram of the behavior classes.	34
4.2	Class diagram of the task definition classes.	34

4.3	The CAMBADA training field. The highlighted region represents the working area. Black rectangles are the approximate location of desks and other obstacles.	36
4.4	Annotated plot of equation 4.2, adapted from [25].	38
4.5	Visual representation of the action set used to learn the rotation behavior. . .	38
4.6	Comparison of the robot's orientation error when using both the hand-coded behavior and the learned behavior.	40
4.7	Comparison of the robot's absolute orientation error when using both the hand-coded behavior and the learned behavior.	41
4.8	Comparison of the mean cost per cycle when using Q-Learning and Q-Batch update rules.	42
4.9	Visual representation of the action set used to learn the dribble behavior. . .	43
4.10	Comparison of the hand-coded and the learned behavior for a specific test case. Squares are placed on each trajectory every second, allowing for a temporal comparison. For each trial, the robot started in the (0, 0) position. The dribbling behavior kicked in after the robot reached an YY position of less than -2 meters.	45
4.11	Comparison of the hand-coded and the learned behavior for a similar test case. Squares are placed on each trajectory every second, allowing for a temporal comparison. For each trial, the robot started close to the (0, -4) position. It then grabbed the ball and moved forward. The dribbling behavior kicked in after the robot reached an YY position of greater than -2.5 meters.	47
4.12	We can reduce the state space by projecting the robot's coordinates onto relative XX and YY axes. The XX axis is parallel to the ball's direction of movement, while the YY axis is perpendicular to it.	49
4.13	Comparison of the hand-coded and the learned behavior for a specific test case. Squares and circles, representing, respectively, the position of the robot and of the ball, are placed on each trajectory every third of a second, allowing for a temporal comparison. Circles with a strong outline signify the ball is grabbed by the robot. The receiving behavior is enabled shortly after the robot detects that the ball started moving.	52

4.14	Comparison of the hand-coded and the learned behavior for a specific test case.	
	Both the robot's and the ball's trajectories are marked with squares and circles every third of a second. A circle with a strong outline signifies the robot has grabbed the ball. The receiving behavior was enabled shortly after the robot detected that the ball started moving.	54

List of Tables

3.1	Skills learned through the use of neural reinforcement learning methods in the Brainstormers 2D Simulation League team and the Brainstormers Tribots Middle Size League team, from 2000 to 2008 [21]. Filled dots represent skills that were used in competition, while empty circles are skills that, although successfully learned, were not used in competition.	31
-----	---	----

Chapter 1

Introduction

1.1 Motivation

CAMBADA¹, University of Aveiro's Middle Size League robotic soccer team, has established itself as a major competitor at a worldwide level. Founded in 2003 as a research project of the ATRI² group within IEETA³, it followed an impressive evolution since then, achieving first place worldwide in the RoboCup competition in 2008 (Suzhou, China), and third place in 2009 (Austria), 2010 (Singapore), 2011 (Turkey) and 2013 (Netherlands). Furthermore, it was crowned National Champion seven times in a row from 2007 to 2013.

In order to carry on with this track of success, constant improvement of both its hardware and software platforms is required. By implementing more efficient behaviors, we can therefore increase an individual agent's efficiency, and thus that of the whole team. Manually coding and optimizing those behaviors can be both difficult and time-consuming. Automated learning methods provide a solution, by allowing the programmer to specify a high-level description of the behavior to be learned, and leaving the search for a successful and optimized control policy to the autonomous agent.

¹Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture

²Transverse Activity in Intelligent Robotics

³Institute of Electronics and Telematics Engineering of Aveiro

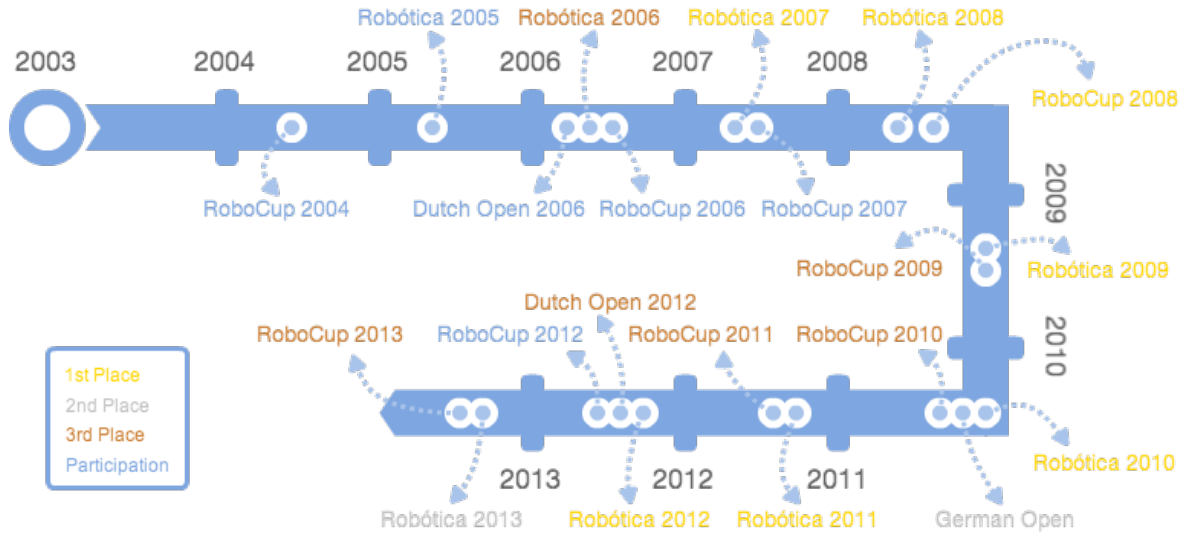


Figure 1.1: Timeline of CAMBADA's evolution.

1.2 Objectives

The focus of this thesis is the application of automated learning techniques, specifically Reinforcement Learning, to the domain of robotic soccer, in order to learn more efficient robot behaviors than the explicitly programmed ones used so far in the CAMBADA team. The goals of this thesis can be summarized as follows:

1. Study the existing body of knowledge on Reinforcement Learning methods;
2. Apply Reinforcement Learning techniques to the context of robotic soccer, specifically the CAMBADA robots;
3. Compare the performance of the learned behaviors with the explicitly programmed ones;

1.3 Thesis Outline

The rest of this thesis is organized as follows: chapter 2 describes robotic soccer, the RoboCup competition and the CAMBADA platform. Chapter 3 covers a background of Reinforcement Learning relevant for this thesis. In chapter 4, the behaviors to be learned are identified, in addition to an analysis of the learning results and comparison with existing explicitly-programmed behaviors.

Chapter 2

Robotic Soccer

2.1 Introduction

Relevant and challenging benchmarks are needed to drive forward scientific and engineering state of the art. In the case of multi-agent robotic systems and artificial intelligence, many benchmarks exist, each catering to a niche of specific needs and problems. One of these benchmarks is robotic soccer.

Soccer is an interesting environment for robotics and artificial intelligence research. Aside from raising challenging technical problems, it is also easily accessible and enjoyable by the general public, thus taking advantage of the actual sport's enormous popularity around the world. More than just presenting scientists and engineers a setting where they can develop, test and compare new technologies, it can also become a source of entertainment for the general public and generate an industry of its own.

In order to successfully play soccer, robots need to possess basic abilities, such as, for example, perceiving their surroundings, namely the field lines, the soccer ball, as well as other robots and (possibly) other obstacles in the field. Furthermore, they need to be able to move around in the field and kick the ball around, if they are to pose a challenge to their opponents. In order to avoid penalties, they need to be able to sense the state of the game and respect the rules of soccer.

While these low-level abilities may be enough to achieve soccer-playing robots, they are not enough to ensure the game is played intelligently. More complex abilities become a necessity, such as, for example, keeping a formation, following a strategy, and adapting to the opposing

team's strengths and weaknesses.

These abilities can be performed by humans with relative ease, but they prove to be challenging for robots.

We can characterize the properties of robotic soccer as an environment for intelligent agents under the framework presented in [1]:

Partially observable: Sensors are imperfect, and as such, it is impossible for an agent to, at all times, perceive all the relevant information with accuracy. On the other hand, teams don't know their opponents' immediate intentions, and so have to work with limited information;

Multi-agent: Robotic soccer is simultaneously cooperative and adversarial: teams are composed of many robots which cooperate among themselves while competing against their opponents;

Stochastic: Taking only into account the currently perceived state and the last action taken, agents cannot perfectly predict the following state. Certain actions can be too complex, or may depend on actuators with a low degree of accuracy, or their outcome can be modified by other players, regardless of whether they are teammates or adversaries;

Sequential: Short-term actions taken within a play or a game can have long-term consequences. For example, failing to accurately pass the ball to a teammate can lead to situations where the opposing team gains control of the ball and attacks, which can impact the outcome of the game;

Dynamic: In game situations, agents cannot pause to think which action to take next will yield the best results, since the other players will not wait for it to be done before carrying on. If an agent takes too long to decide, that is the same as deciding to do nothing;

Continuous: State, actions and time are continuous. For example, the distance between a robot and the ball or to the goal is continuous rather than discrete, and robots can move within the field with a continuous range of speeds.

2.2 RoboCup

RoboCup¹ is the name of an international robotics research initiative, which sponsors an international competition under the same name. Its first edition ran in 1997, and has since then been organized yearly. It aims to drive forward scientific progress in robotic systems and artificial intelligence by presenting a very ambitious long term goal: that by the year 2050, a team of autonomous robots shall play against the most recent World Cup Champion and win, complying with the official FIFA rules.



Figure 2.1: The RoboCup project’s logo, taken from the RoboCup website.

Such high aspirations, however, need to be broken down into smaller subgoals in the meantime. Using modified rules in early stages of the project to reduce complexity allows researchers to focus on simpler and more feasible problems. Then, as solutions to those simpler problems become available, the rules can be incrementally changed to introduce the avoided complexities. For example, until 2010 the color of the ball was pre-established as orange. Nowadays, any standard FIFA ball can be used, provided its main colors are not white, black or green.

RoboCup Soccer is structured in various leagues:

Middle Size League²: This league features teams of up to 5 robots, with each robot having at most 50 centimeters in diameter, 80 centimeters of height and 40 kilograms of weight. Matches take place in a 12 by 18 meters field, and a regular sized FIFA approved soccer ball is used. All sensors must be mounted on the robots. Wireless communication between players is allowed to enable coordination and cooperative behavior. Additionally, robots can also communicate with an external entity, designated as “coach”. The coach is an autonomous program with no sensors of its own, which can use sensor information relayed by players to make decisions and facilitate coordinated behavior among them.

¹Robotic World Cup Initiative website, last visited July 2013: <http://www.robocup.org/>

The rules of the game are adapted from the official FIFA rules, and are enforced by a human referee and one or more assistant referees.



Figure 2.2: A Middle Size League soccer game situation.

Small Size League³: This league is comprised of teams of five robots, each robot having at most 18 centimeters in diameter and 15 centimeters in height. The field is 6.05 meters long and 4.05 meters wide, and an orange golf ball stands in for the soccer ball. Two cameras located 4 meters above the playing field capture images, while a standardized vision system, SSL-Vision, processes them to detect and track the ball and players. Off-field computers communicate with the robots wirelessly to inform them of their position and of referee commands, and, typically, they also perform most of the computation necessary for determining agents' control commands and coordination. This combination of centralized and distributed control in a dynamic and multi-agent environment is one of the focuses of this league.

Simulation League⁴: A league with a strong focus on artificial intelligence and team play, as the players are not robots but simulated agents, playing on a simulated environment. This league is subdivided into the following subleagues:

2D Simulation This subleague is characterized by using only 2 dimensions to represent the virtual world. Teams of eleven agents compete against each other in this environment, which is managed and simulated by a central server, called SoccerServer. The server keeps track of the world state, and each player communicates with it to receive information from their virtual sensors and to give commands to their actuators. Communication is made in cycles of 100 milliseconds, with games



Figure 2.3: A Small Size League soccer game.

lasting 6000 cycles.

3D Simulation: This subleague increases realism by using 3 dimensions to model the world, which in turn increases complexity and brings the simulation closer to real world situations. Additionally, the virtual body of the agents are models of the NAO humanoid robots which are also used in the Standard Platform League. This means researchers can test algorithms in this platform before using them on the real NAO's, while, on the other hand, means that, unlike the 2D Simulation League, there is less focus on high-level behaviors and teamplay, and more on low level abilities humanoid robots need to master to play soccer, such as walking, running, kicking, etc.

Standard Platform League⁵: This league is played by teams made up of identical robots, hence the name “Standard Platform”, with the focus on software, the hardware being the same for all teams. Currently, the standard robot is the humanoid NAO by Aldebaran Robotics ⁶, but previously a “Four-Legged League” existed which used Sony’s AIBO dog robots.

Humanoid League⁷: This league is played by robots with human-like body and sensors. As the objective is to have robots play soccer as humans would, perception and world modelling are not simplified by using non-human-like sensors. The main research issues

⁶Aldebaran Robotics, last visited in July 2013: <http://www.aldebaran-robotics.com/>



(a) 2D Simulation league.



(b) 3D Simulation league.

Figure 2.4: Examples of game situations from both Simulation leagues.



Figure 2.5: A Standard Platform League game example.

in this league are dynamic walking, running and kicking without losing balance, self-localization within the field, visual perception to detect other players, field markings and the ball and teamwork with other teammates.

This league is further divided in three subleagues:

Kid Size: Robots with 30 to 60 centimeters of height play in teams of three robots;

Teen Size: Robots of 100 to 120 centimeters of height compete in teams of two robots;

Adult Size: Robots of 130 centimeters and taller compete against each other individually. The robots play in striker versus goalkeeper situations, exchanging roles after each play.

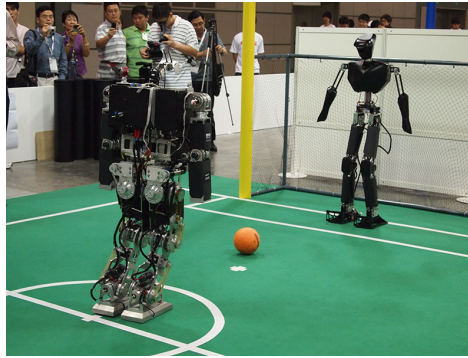
Besides RoboCup Soccer, three other competition domains have been introduced:



(a) Kid size league.



(b) Teen size league.



(c) Adult size league

Figure 2.6: Pictures from games of the three Humanoid leagues.

RoboCup Rescue⁸: Disaster relief is an area that could benefit greatly from the use of robotic systems. RoboCup Rescue fosters research and development in robotics for search and rescue situations. The main areas of research within this competition are multi-agent systems, information systems for collecting, treating, summarizing and disseminating relevant information, decision support systems for planners, reliable simulators and benchmarks to evaluate rescue strategies and integrated robotic systems. This competition is divided in two leagues:

- Robot League
- Simulation League

RoboCup@Home⁹: Domestic situations present a major opportunity for future applications of robotic systems. Robots could be used to provide physical help to elderly or reduced mobility individuals, or just perform service duties by taking over or helping out with everyday chores. The competition focuses on a set of tests and an open challenge

that takes place in a realistic domestic environment. The main research areas include, but are not limited to, Computer Vision, Object Manipulation, Human-Robot Interaction, Localization, Navigation, Mapping, Adaptive Behaviors, Ambient Intelligence and System Integration.

RoboCupJunior¹⁰: A competition with educational goals, organized at local, regional and international levels, with the objective of sparking the interest of young students in robotics. Participants are given the opportunity to learn and gain hands-on experience in robotics, electronics, programming and teamwork, while competing against peers from diverse backgrounds, who they also get to meet. The competition is organized in three leagues:

- Soccer League
- Dance League
- Rescue League

2.3 The CAMBADA Platform

2.3.1 Hardware

The CAMBADA robots follow a modular structure, with three main layers. The lower layer is composed by an aluminum plate and by three motors, batteries and wheels, as well as a kicker and grabber system. The middle layer holds a laptop, which is the “brain” of the robot. The topmost layer extends close to the maximum height of 80 centimeters. It holds a camera and a hyperbolic mirror, as well as an electronic compass. The two lower layers resemble a triangle, with a cut-out section in the front to fit a soccer ball.

Holonomic motion is accomplished through the use of three omniwheels, also called swedish wheels, located at the vertices of an equilateral triangle centered on the center of the robot.

In order to maintain control of a soccer ball, the robot has two small arms, each with a small motor and omniwheel. By spinning the omniwheels against each other, the robot can keep the ball close to its body, allowing it to roll along with the robot, but not to roll away from it. The heights of each arm are adjustable, and they are used to sense whether the ball is under control. This system is called “grabber”.



Figure 2.7: A CAMBADA robot.

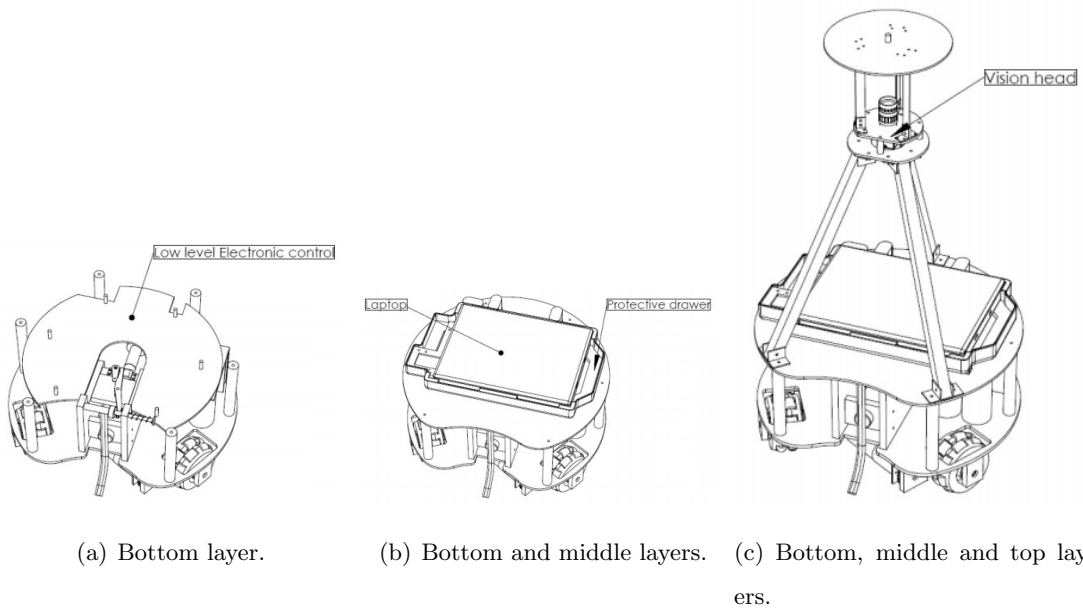
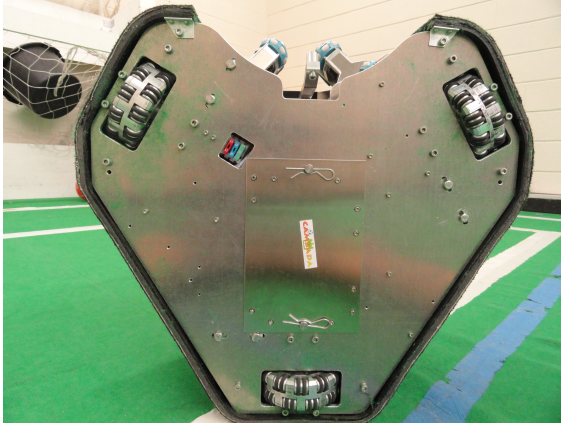


Figure 2.8: Mechanical drawings of the CAMBADA robots detailing their modular and layered structure, adapted from [2].

The kicker system is based on an electromagnetic solenoid actuator and a metal bar. It allows direct or lobbed kicking: for direct kicks the solenoid hits the ball directly, whereas for lob kicks the solenoid pushes the bar, which lifts the ball off the ground.

Omnidirectional vision is achieved using a catadioptric system made of a regular video camera pointed at a hyperbolic mirror.

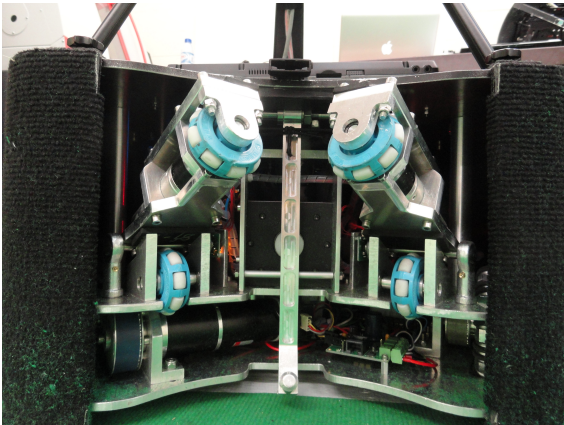


(a) Wheel placement.



(b) Omniwheel detail.

Figure 2.9: Holonomic drive.



(a) Close-up of the grabber and kicker systems.



(b) Grabber system in use.

Figure 2.10: The grabber and kicker system.

The general architecture adopted by CAMBADA was modeled according to a biomorphic approach. It consists of two layers: a coordination layer and a low-level layer. The former includes communication with teammates and the coach, sensors with high bandwidth requirements, such as a camera, and a main processing unit, the “brain”. The low-level layer stands as the “nervous system”, and is used to receive low bandwidth sensing information and to send commands to the actuators. The “brain” is the central node of this layered architecture: it handles external communication, processes sensing information and decides which commands to be applied.

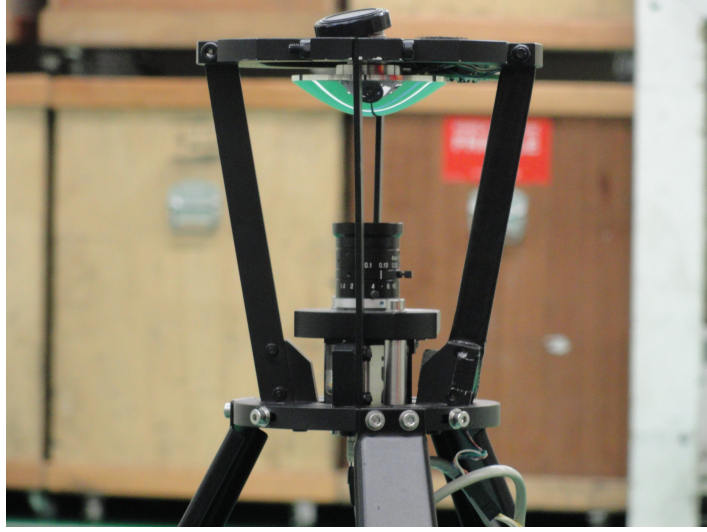


Figure 2.11: The omnidirectional vision system.

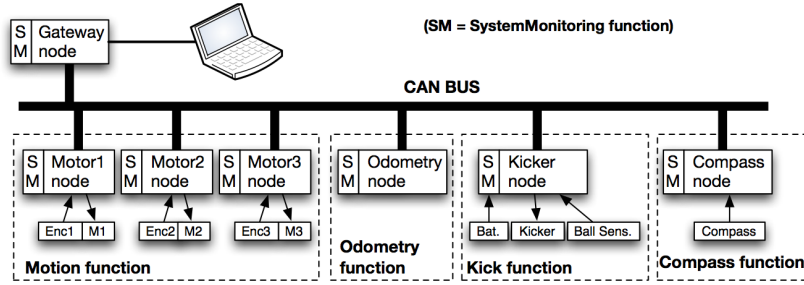


Figure 2.12: Harware architecture with functional mapping, adapted from [3].

The low-level control layer follows a distributed model. It is implemented as a network of micro-controllers, each encapsulating basic functions of the robot. To comply with the real-time requirements of this network, a variant of the Controller Area Network (CAN) is used.

2.3.2 Software

The software architecture follows the biomorphic paradigm mentioned above, which is also depicted in figure 2.13. Figure 2.14 shows the internal components of each layer. While this figure is pretty much self explanatory, some noteworthy details will be presented.

The RTDB, short for Real Time Database, is essential for cooperation and inter-agent communication, as it allows agents to share sensing information among themselves. Information

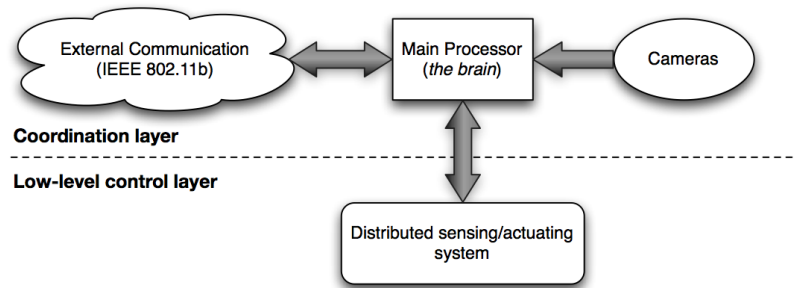


Figure 2.13: Biomorphic architecture, adapted from [4].

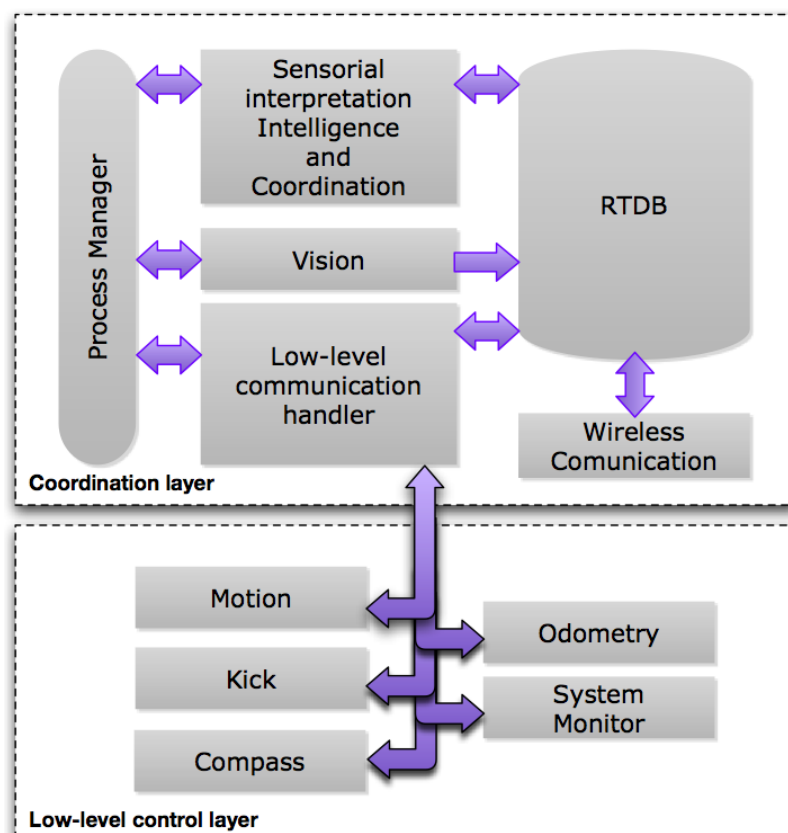


Figure 2.14: Layered software architecture, adapted from [5].

sharing enables higher reliability and allows better decision-making. However, this needs to be done taking into account the tight temporal constraints for information to be of relevance, since players and the soccer ball can move at high speeds. To this end, robots broadcast some of their state data, which is then stored by every agent, following a distributed shared memory model.

The RTDB stores both local and shared information. Local information needs to be transmitted between processes, which the RTDB facilitates. Shared data is disseminated by agents and updated automatically by an autonomous communication system [3], and contains the positions of all players and of the ball, goal areas and corners, as reported by them.

Wireless communication is done through an IEEE 802.11 network. In order to minimize collisions within the CAMBADA team, an adaptive TDMA¹¹ protocol is used [3]. Each agent has a predefined slot to transmit its state information within a round, and the time length of rounds is adapted to fit the channel's status. Individual access slots are separated from each other as much as possible within a round, therefore reducing chances of collisions within the team.

The omnidirectional vision process is able to identify and estimate the positions of various objects, such as line markings, obstacles and a soccer ball, from the frames captured by the catadioptric system. This is done mainly through the use of radial search lines and color analysis [3], taking advantage of the highly structured nature of the MSL soccer environment. The identification of arbitrarily-colored soccer balls is done using an edge detection algorithm and applying the circular Hough transform to its results. Around 30 frames are processed each second.

The low-level communication module acts as a gateway between the RTDB and the dedicated microcontrollers, communicating through the CAN. Software processes access the RTDB to get sensor information and to send actuator commands. The gateway takes care of either updating the RTDB according to the latest sensing values or of sending commands to the actuators.

The process manager is responsible for triggering processes, guaranteeing certain constraints, such as, for example, that precedence among related processes is respected.

CAMBADA robots run an “agent” process which is responsible for decision-making. It uses the gathered state information and interprets the data to reach a current state representation with a high degree of certainty, and communicates with its teammates and with the “coach” entity to achieve coordinated and cooperative behavior. Sensor fusion takes not only the robot's own sensors but also those of its teammates. The coach program is, among other things, responsible for propagating referee commands.

¹¹Time Division Multiple Access

The actions each agent executes are dependant on its “role” and “behavior”. Behaviors are basic skills such as kicking or dribbling the ball, moving to a given location and orientation, etc. Roles embody higher-level behavior expectations such as, for example, acting as a striker, which means the agent needs to dribble the ball to the opponent team’s goal and, when in range, shoot.

Agents follow a formation, which dictates a movement model for each agent [3]. Together with automatic role assignment, this leads to coordinated gameplay.

The basestation is a monitoring application which gives some degree of control over the playing agents. It displays some of the internal state of each agent, such as which role and behavior are being executed, as well as their position in the field and their velocity, and other lower-level information such as battery charge. It is possible to send Start or Stop signals to each agent and to manually assign roles, which is fundamental for testing specific situations. During matches, the basestation is responsible for relaying referee commands, transmitted by the “referee box”, to the agents.

2.4 Summary

After this chapter, one should understand the importance of robotic soccer as a research benchmark and how competitions such as RoboCup help setting up a nurturing environment for this venture to flourish and evolve. Additionally, the anatomy of the CAMBADA robots was discussed, both its hardware and software.

Chapter 3

Reinforcement Learning

3.1 Introduction

Reinforcement learning [6] is the study and application of algorithms which allow an autonomous agent to increase its performance in a given task by building up on experience gained through interaction with its environment. The agent need not be told how to accomplish its task, but only which conditions mean success or failure. It is up to the learning agent to experiment and find out which is the best way to achieve its given goal.

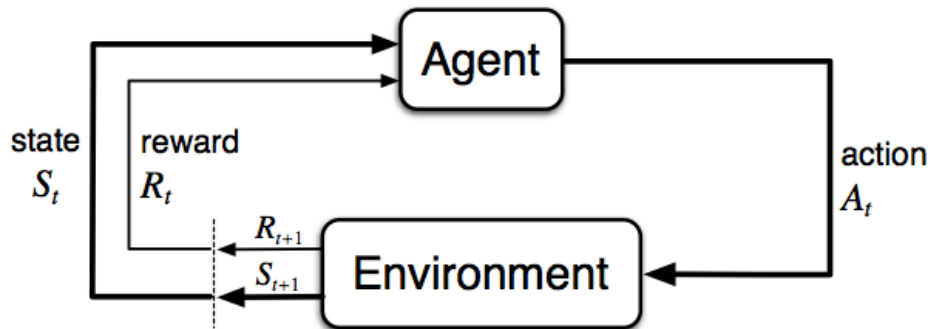


Figure 3.1: Interaction between the agent and the environment in a reinforcement learning setting, adapted from [6].

An agent must be capable of sensing, even if only partially, its state and of modifying it by acting. A reward signal is given to the agent at each moment, allowing it to understand whether the consequences of its actions were positive or negative. The agent's objective is to optimize the accumulated reward collected throughout its lifetime, which should lead to an

optimal control policy.

This kind of learning can be best described as "trial-and-error" learning. The agent must try a wide range of action-state combinations and memorize the reward obtained, so that it can later search through the available options and choose the best action. For many problems, actions can have delayed effect, i.e. the full extent of consequences from a single action may only be apparent much later, thus affecting reward signals since that action was taken.

A permanent dilemma within reinforcement learning is the so called "exploration vs. exploitation" problem. To discover which actions are most appropriate, an agent may need to, at some point, take unfavorable actions, which could, in the future, take the agent to more interesting states. Otherwise, the agent may just fall into a local minimum. Exploring new states and exploiting states previously known to lead to good rewards is an ever present trade-off.

This chapter will provide a background of the relevant theory and algorithms to better understand the work presented in this thesis.

3.2 Markov Decision Processes

Markov Decision Processes (MDP) provide an appropriate mathematical framework to formally describe the interaction between a decision making agent and its environment, as well as to apply optimization methods such as Reinforcement Learning or Dynamic Programming to find optimal policies for said agent.

Specifically, an MDP is a 4-tuple $\langle S, A_s, P(s_{t+1}|s_t, a), R(s_t, a, s_{t+1}) \rangle$:

S is the set of all possible states;

A_s is the set of possible actions for each possible state s ;

$P(s_{t+1}|s_t, a)$ is the transition model of the system. It defines the probability of the system transitioning to state s_{t+1} knowing that action a is taken in state s_t ;

$R(s_t, a, s_{t+1})$ is the reward function used to give feedback to the agent. The reward is determined knowing the agent ended up in state s_{t+1} after taking action a in state s_t . This can sometimes be abbreviated to $R(s_t, a_t)$ or even $R(s_t)$, depending on the specific reward function chosen.

An important property of MDPs is that every state transition depends only on the current state and action taken, i.e. in order to determine the probability of reaching a given state s_{t+1} we only need to know s_t and a_t . This property is formally represented in equation 3.1, and is usually called Markov property.

$$P(s_{t+1}|s_t, a) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \dots) \quad (3.1)$$

Since we have S and A_s we can describe the behavior of the agent as a function of its state s which returns an action a to be taken. Such a function is called a policy, usually denoted as $\pi(s)$. Having defined a reward function $R(s_t, a, s_{t+1})$, we can evaluate policies by analysing their cumulative reward collected over the lifetime of the agent. However, due to the stochastic nature of the task, different runs of the same policy starting in the same state may lead to different outcomes. Therefore, we choose to instead evaluate policies based on the expected value of cumulative reward.

$$J^\pi(s) = E \left[\sum_{t=0}^{\infty} R(s_t, \pi(s_t), s_{t+1}) \right], s_0 = s \quad (3.2)$$

The immediate reward given by the reward function may be considered either as positive or negative, and thus solving a MDP becomes a matter of, respectively, maximizing the accumulated reward or minimizing the accumulated cost, therefore yielding an optimal policy $\pi^*(s)$. For the remainder of this document, the immediate rewards will be considered negative, and can be regarded as transition costs.

$$\begin{aligned} \pi^*(s) &= \arg \min_{\pi} J^\pi(s) \\ &= \arg \min_{\pi} E \left[\sum_{t=0}^{\infty} R(s_t, \pi(s_t), s_{t+1}) \right], s_0 = s \end{aligned} \quad (3.3)$$

3.3 Value Iteration

Let us begin by considering one of the simplest algorithms to compute the optimal policy in an MDP. The Value Iteration algorithm is based on Bellman's Equation [7]. For deterministic environments, where state transitions are given by a function $f(s_t, a_t) = s_{t+1}$, this equation is the following:

$$J^*(s) = \min_{a \in A} \left\{ \sum_{s_{t+1} \in S} R(s_t, a_t, f(s_t, a_t)) + J^*(f(a_t, s_t)) \right\} \quad (3.4)$$

The idea behind equation 3.4 is that the cost of state s is the immediate reward of taking the optimal action in that state plus the cost of the next state, assuming the agent will again take the optimal action. When taking into account stochastic environments, this equation turns into the following:

$$J^*(s) = \min_{a \in A} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) (R(s_t, a_t, s_{t+1}) + J^*(s_{t+1})) \right\} \quad (3.5)$$

The optimal policy $\pi^*(s)$ can then be computed from $J^*(s)$, as we have seen before.

$$\pi^*(s) = \arg \min_{a \in A} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) (R(s_t, a_t, s_{t+1}) + J^*(s_{t+1})) \right\} \quad (3.6)$$

Therefore, we only need to get $J^*(s)$ in order to solve the MDP. To do that, we start from an arbitrary $J_0(s)$ and, for every state, we iteratively calculate $J_k(s)$ using the Bellman update rule [7]:

$$J_{k+1}(s) \leftarrow \min_{a \in A} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) (R(s_t, a_t, s_{t+1}) + J_k(s_{t+1})) \right\} \quad (3.7)$$

$$\lim_{k \rightarrow \infty} J_k(s) = J^*(s) \quad (3.8)$$

In order for equation 3.8 to be true, certain conditions have to be met. On one hand, convergence is guaranteed for stochastic shortest path problems. These are characterized by the following:

- The policy space contains at least one proper policy, that is, a policy with greater than zero probability of reaching a terminal state. This is true when there is a connection from every state represented in the Markov Chain to a terminal state;
- For every improper policy, there is at least one state with infinite path costs, i.e. $\exists s \in S, J(s) = \infty$;
- There is at least one terminal state with zero cost: $\exists s \in S, R(s) = 0$.

On the other hand, we can introduce a discount factor, usually represented as γ , to avoid having $J(s)$ drift to an infinite horizon. In that case, the expected path cost of a state is given by:

$$J^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1}) \right], 0 \leq \gamma < 1 \quad (3.9)$$

If $\gamma = 1$ then equation 3.9 becomes equation 3.2.

The pseudocode for Value Iteration is presented in algorithm 3.3.

Algorithm 1 Value Iteration algorithm

```

1: function VALUE-ITERATION( $S, A, P, R, \epsilon$ )
2:   inputs:
3:      $S$  is the set of possible states
4:      $A$  is the set of possible actions
5:      $P$  is the transition model of the system
6:      $R$  is the reward function
7:      $\epsilon$  is a small positive number
8:   outputs:
9:      $\pi^*(s)$ , the optimal policy
10:     $J^*(s)$ , the cost function

11:   Initialize  $J_0$  arbitrarily
12:    $k \leftarrow 0$ 
13:   repeat
14:      $k \leftarrow k + 1$ 
15:     for each state  $s \in S$  do
16:        $J_k(s) \leftarrow \min_{a \in A} \left\{ \sum_{s_{t+1} \in S} P(s_{t+1} | s_t, a_t) (R(s_t, a_t, s_{t+1}) + \gamma J_{k-1}(s_{t+1})) \right\}$ 
17:   until  $\forall s |J_k(s) - J_{k-1}(s)| < \epsilon$ 
18:    $J^*(s) = J_k(s)$ 
19:   for each state  $s \in S$  do
20:      $\pi^*(s) = \arg \min_{a \in A} \{P(s_{t+1} | s_t, a_t) (R(s_t, a_t, s_{t+1}) + J^*(s_{t+1}))\}$ 
21:   return  $\pi^*(s), J^*(s)$ 

```

The obvious problems of this algorithm is that it requires large amounts of memory space and many computations both to store and update $J(s)$ for each state. Furthermore, it also requires the transition model, that is, $P(s_{t+1}|s_t, a_t)$, to be known. In fact, few interesting problems with real world applications fall into this category.

3.4 Q-Learning

Q-Learning [8] can be used to obtain an optimal policy when the transition model or the reward function is not known, and it is as such called a model-free learning algorithm. The agent needs to interact with the environment and learn from its experience, which is represented as a collection of state transitions of the form $\langle s_t, a_t, s_{t+1}, r_{t+1} \rangle$. Essentially, the agent observes that it took action a_t in state s_t and ended up in state s_{t+1} , and so it can later remember how beneficial that state/action combination was.

We can get the estimated expected reward as specified in equation 3.10.

$$Q^\pi(s_t, a_t) = \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t)(R(s_t, a_t, s_{t+1}) + \min_{a_{t+1} \in A(s_{t+1})} Q^\pi(s_{t+1}, a_{t+1})) \quad (3.10)$$

A policy can be obtained from a given $Q(s, a)$ function by always selecting the action with the least expected cost. As such, to get the optimal policy we need to find $Q^*(s, a)$:

$$\pi^*(s) = \arg \min_{a \in A(s)} Q^*(s, a) \quad (3.11)$$

Similarly to how $J^*(s)$ was obtained in Value Iteration, $Q^*(s, a)$ needs to be calculated iteratively. Equation 3.7 can be adapted to become:

$$Q_{k+1}(s_t, a_t) \leftarrow \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t)(R(s_t, a_t, s_{t+1}) + \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a_{t+1})) \quad (3.12)$$

As it was previously said, in model-free scenarios both the transition model $P(s_{t+1}|s_t, a_t)$ and reward function $R(s_t, a_t, s_{t+1})$ may be unknown, so we need an update rule that does not depend on them. It has also been said that the agent can interact with its environment to collect experience in the form of 4-tuples $\langle s_t, a_t, s_{t+1}, r_{t+1} \rangle$. Because these experience tuples

contain both the state transition and the immediate reward, it is possible to directly calculate $Q_{k+1}(s_t, a_t)$:

$$Q_{k+1}(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a_{t+1})), 0 \leq \alpha \leq 1, \quad (3.13)$$

where α , usually called learning rate, allows to control how much the new estimate relies on the previous estimate and the sampled experience.

In order for $Q_k(s, a)$ to converge to $Q^*(s, a)$, the same conditions as for Value Iteration convergence must be met. Additionally, every state/action pair needs to be visited infinitely often, and the learning rate parameter should decrease after each update.

Algorithm 3.4 presents the pseudocode for Q-Learning.

3.5 Fitted Q-Iteration

Interesting real world problems which are good candidates to being solved by reinforcement learning algorithms usually have very large state and action spaces. The classical Q-learning algorithm uses a discrete table to represent Q-values, i.e. the state space is broken down to individual cells, where the number of cells depends on the level of detail we decide to settle for. In this situation, we face a dilemma: smaller sized cells yield better resolution, such that if we had infinitely small cells we would approach a continuous state space representation; on the other hand, the smaller each cell is, the more cells are necessary to represent the complete state space, thus increasing the memory requirements of our system. For many problems, tabular methods can become too demanding, or cannot represent the Q-function accurately enough to derive a good solution.

To counter this problem, Fitted Q-Iteration methods have been studied [9]. The core idea of these methods is that any regression algorithm can be used to approximate Q-functions, thus taking advantage of their generalization capabilities. This greatly reduces the memory requirements, as we no longer need to represent the Q-function as a discrete table.

In many real world systems, it may be costly to collect experience. In such situations, gathering vast amounts of transition tuples that are needed for Q-learning inspired algorithms may be impractical. For Reinforcement Learning to be feasible in these situations we need data efficient algorithms.

Algorithm 2 Q-Learning algorithm

```
1: function Q-LEARNING()  
2:   outputs:  
3:      $\pi^*(s)$ , the optimal policy  
4:      $Q^*(s, a)$ , the optimal  $Q$ -value function  
  
5:   initialize  $Q_0(s, a)$  arbitrarily  
6:    $k \leftarrow 0$   
7:   repeat  
8:      $t \leftarrow 0$   
9:     repeat  
10:      observe  $s_t$   
11:      select an action  $a_t$  (either greedily or exploratorily)  
12:      execute  $a_t$   
13:      observe  $s_{t+1}, r_{t+1}$   
14:       $Q_{k+1}(s_t, a_t) \leftarrow (1 - \alpha)Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a_{t+1}))$   
15:       $t \leftarrow t + 1$   
16:    until a terminal state is reached  
17:     $k \leftarrow k + 1$   
18:  until convergence criteria is met  
19:  for each state  $s \in S$  do  
20:     $\pi^*(s) = \arg \min_{a \in A(s)} Q^*(s, a)$   
21:  return  $\pi^*(s), Q^*(s, a)$ 
```

Q-function approximators can be computed at each time step or after a set of transitions has been sampled. In the first case, also called online learning, a new function approximator is learned after each transition tuple is sampled, and that tuple is then discarded. Alternatively, in the latter case, called offline or batch learning, the agent cycles between an interaction phase and a learning phase. First, it builds up a set of experience tuples using a fixed policy, and afterwards proceeds to learn a new Q-function approximator over all the experience collected so far. Unlike online learning, experience tuples are not discarded, and are instead kept to be used in further learning phases. Batch learning has shown superior results to online learning, as Q-function approximators are more stable, and show higher data efficiency.

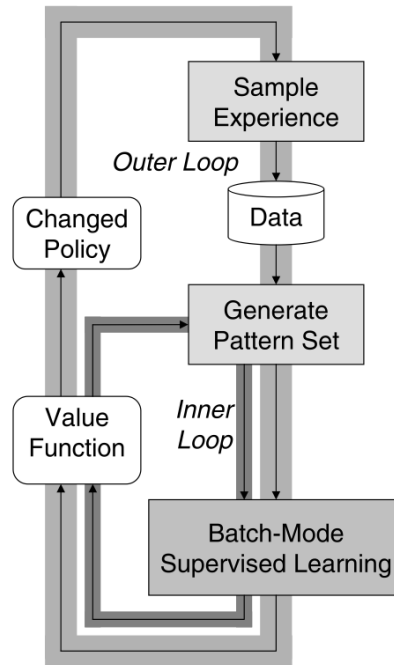


Figure 3.2: The Batch Reinforcement Learning framework, depicting the interconnections of the various modules, taken from [10].

Figure 3.2 shows how the Batch Reinforcement Learning scheme can be decomposed in three different modules. These modules are run sequentially and in loops. First, the agent gathers experience, which is then processed to generate a pattern set. A pattern set consists of a sequence of $\langle \text{input}, \text{target output} \rangle$ tuples, where the input is a tuple that describes the state and action taken, and the target output is the updated Q -value, calculated according to an update rule. A supervised learning algorithm is then trained over the pattern set, yielding a

new approximation of the Q -function. The pattern set generator and the supervised learning modules can be run multiple times without proceeding to collect new sets of experience, until a satisfying policy is learned.

Each of the three modules can be modified and customized to suit specific problems. For example, experience may be gathered using a purely greedy policy, or it can include some degree of exploration. Also, the amount of data that is gathered is a matter of choice. In certain situations, it may be easier to collect larger amounts of experience data before proceeding to a learning phase; whereas when the learning platform can be accessed easily and with little cost, such as a simulator, we may opt for smaller amounts of experience between each learning phase. When it comes to the pattern set generator module, we can choose different update rules [10], which ultimately lead to different learning results. Within the supervised learning module, we can decide to experiment with different algorithms, or just to customize some of its parameters. The Batch Reinforcement Learning is but a framework, which allows for a generous degree of flexibility.

3.6 Neural Fitted Q-iteration

Neural networks are good candidates to be used as Q -function approximators, because of their ability to approximate non-linear functions with accuracy and their generalization properties [11]. However, when used in an online learning scheme, problems of instability arise [12]. This is because individual updates from isolated transition tuples cause weights between neurons to change, but changing those weights can affect the output of different state-action combinations, yielding different results. Intuitively, this means that learning on individual experience tuples can cause the network to “forget” about previous experience. Alternatively, a batch learning scheme can be used, in which the network is trained over all the collected experience tuples sampled so far. This forces the network to “remember” the previous experience when adding new transition tuples.

In this spirit, Neural Fitted Q-iteration is focused on the use of neural networks in batch Fitted Q-iteration. Additionally, Riedmiller advises [13, 14, 10] the use of RPROP [15], a variant of the classical backpropagation algorithm, as it reportedly converges faster and with greater insensitivity to its parameters, thus providing the benefit of not having to fine tune them.

Some precautions need to be taken in order to avoid common problems when approximating value functions with neural networks. Different state dimensions can have different ranges of values. On the other hand, depending on the reward function, outputs may also have a wide range of values. Scaling the pattern set’s inputs and outputs can be beneficial, and even necessary, so that the network can accurately approximate them. This is always possible, because the full pattern set is generated before the training phase.

When applying NFQ to a very wide state-action space, it can be hard for the learning agent to identify the goal state and how to reach it. If the agent does not sample enough transitions to goal states, then the neural network’s output will tend to increase to its maximum value, and will be unable to correctly approximate goal-state transitions. To mitigate this problem, a heuristic called “hint to goal” can be used. It consists of adding artificial transition tuples to the pattern set over which the neural network is trained. These tuples should clearly identify goal states, in order to direct the agent to them. Alternatively, a different heuristic, “Q-min”, may be used. The idea behind this heuristic is that by subtracting all pattern set outputs by the minimum output the pattern set will always contain outputs of value 0. An added advantage is that no additional information needs to be specified to the agent.

The original version of NFQ uses an update rule similar to Q-Learning, with parameter $\alpha = 1.0$ [13, 14]:

$$Q_{k+1}(s_t, a_t) \leftarrow r_t + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a_{t+1}), \quad (3.14)$$

and so the pattern set is built as a collection of tuples of the form:

$$\begin{aligned} \langle input, target \rangle &= \langle \langle s_t, a_t \rangle, Q_{k+1}(s_t, a_t) \rangle \\ &= \langle \langle s_t, a_t \rangle, r_t + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a_{t+1}) \rangle \end{aligned} \quad (3.15)$$

This algorithm has the desirable properties of being very data efficient and model-free[14], and, as such, was chosen to be used in the learning experiments that this thesis describes.

3.7 Q-Batch update rule

In section 3.5 - Fitted Q-Iteration, we mentioned that the batch reinforcement learning framework follows a modular structure, and that each module can be customized according to one’s needs. This section focuses on the pattern generation module, specifically on the use of Q-Batch, a recently developed update rule [16].

Algorithm 3 Neural Fitted Q-iteration algorithm

```
1: function NFQ( $D, N$ )  
2:   inputs:  
3:      $D$  is the experience data  
4:      $N$  is the number of inner iterations of the Batch RL framework  
5:   outputs:  
6:      $Q_N(s, a)$ , the  $Q$ -value function approximator learned after  $N$  iterations  
  
7:   initialize  $Q_0(s, a)$  arbitrarily  
8:    $k \leftarrow 0$   
9:   while  $k < N$  do  
10:    Generate pattern set  $P$  from  $D$   
11:    Add artificial patterns to  $P$   
12:    Normalize target values in  $P$   
13:    Scale pattern values in  $P$   
14:     $Q_{k+1} \leftarrow$  train neural network over  $P$   
15:     $k \leftarrow k + 1$   
16: return  $Q_N(s, a)$ 
```

Figure 3.2 showed the workflow of batch reinforcement learning. As it is visible, after a new value function has been approximated new sets of transition tuples are sampled. When gathering experience, it is often easier to sample entire episodes consisting of connected trajectories, because we only need to set up some initial conditions and then let the agent sample from the environment until a stopping condition is met.

The original version of Neural Fitted Q-iteration uses an adapted version of the Q-Learning update rule. While good results have been achieved [13, 14, 10], it does not take advantage of the episodic structure of the experience information, since the Q-value for a given state-action pair is calculated using only the immediate reward and the Q-value for the following state. A more informed approach could also use information from the rest of the trajectory.

Q-Batch uses the notion of n -step returns [8], wherein the Q-value of a state-action pair is determined looking further into the future.

$$\begin{aligned}
Q_{k+1}^1(s_t, a_t) &\leftarrow r_{t+1} + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_k(s_{t+1}, a) \\
Q_{k+1}^2(s_t, a_t) &\leftarrow r_{t+1} + \gamma r_{t+2} + \gamma^2 \min_{a_{t+2} \in A(s_{t+2})} Q_k(s_{t+2}, a) \\
Q_{k+1}^3(s_t, a_t) &\leftarrow r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 \min_{a_{t+3} \in A(s_{t+3})} Q_k(s_{t+3}, a) \\
&\dots \\
Q_{k+1}^n(s_t, a_t) &\leftarrow \sum_{i=0}^{n-1} \gamma^i r_{t+1+i} + \gamma^n \min_{a_{t+n} \in A(s_{t+n})} Q_k(s_{t+n}, a)
\end{aligned} \tag{3.16}$$

This scheme fits nicely in the Batch Reinforcement Learning setup, since data from whole trajectories is available during the training phase, and the Q-function approximator is updated synchronously.

The Q-value of each state transition is updated as the minimum n -step return, as shown in equation 3.17.

$$Q_{k+1}(s_t, a_t) = \min_n Q_{k+1}^n(s_t, a_t) \tag{3.17}$$

While at a first glance it may look like this update rule has a greater computational cost than the standard Q-Learning update rule presented in equation 3.13, it is possible to reach constant time performance by taking advantage of the fact that $Q_{k+1}(s_t, a_t)$ can be calculated

recursively as defined in equation 3.18, and that data for each episode can be processed in reverse order.

$$\begin{aligned}\min_n Q_{k+1}^n(s_t, a_t) &= \min(Q_{k+1}^1(s_t, a_t), r_{t+1} + \gamma \min_{a_{t+1} \in A(s_{t+1})} Q_{k+1}^{n'}(s_{t+1}, a_{t+1})) \\ &= r_{t+1} + \gamma \min(\min_b Q_k(s_{t+1}, b), Q_{k+1}(s_{t+1}, a_{t+1}))\end{aligned}\tag{3.18}$$

3.8 Applications in Robotic Soccer

Several successful applications of Reinforcement Learning algorithms to the context of robotic soccer have been reported, particularly within the Brainstormers team.

In [17], simulated agents learned how to kick the ball, and performance in tactical attack situations of 2-vs-1 and 2-vs-2 was also improved due to learning. In [18], a 7-vs-8 situation is explored, again with improved results.

In [19], both a grid based approximator and a neural network based approximator are used to learn to drive a Middle Size league robot to a target position and orientation. A second variant of this task is also presented, which includes avoiding collisions with the ball, placed at a fixed coordinate.

The problem of ball interception in the shortest amount of time within the 2D simulation league is discussed in [20].

Neural Fitted Q-Iteration is used to learn an effective aggressive defense behavior for 2D simulated agents, as described in [10, 21, 22]. An increase in success from 53% with an handcoded policy to 89% with the learned behavior is reported. The learned behavior was integrated into the Brainstormers 2D simulation team, and was used in competition matches, contributing to winning the 2007 and 2008 RoboCup 2D simulation championship.

Ball interception by a Middle Size League robot, both in a simulated environment as well as in real situations, is explored in [23]. The learned behavior was integrated in the competition code.

In [10, 21], NFQ is used to learn a motor speed controller and a ball dribbling behavior, both for a Middle Size League robot. The motor speed controller was learned after only 3 minutes of interaction and is reported to be very reliable. The ball dribbling behavior is both sharper and faster than the hand coded method. This behavior was integrated in the team

and was used in competition matches.

Table 3.1 shows a timeline of learned behaviors within the Brainstormers team, from the year 2000 until 2008.

	'00	'01	'02	'03	'04	'05	'06	'07	'08
Simulation League Brainstormers 2D									
Hard and precise kicking	•	•	•	•	•	•	•	•	•
Intercepting the ball	•	•	•	•		○	○		
Moving to position	•	•	•	•	•				
1-vs-1 aggressive defence								•	•
7-vs-8 attack	○	•	•	•	•		•	•	•
Penalty kick				•	•	•	•	•	•
Middle Size League Brainstormers Tribots									
Motor speed control								○	○
Moving to position							○	○	○
Intercepting the ball							•	•	•
Dribbling the ball								•	•

Table 3.1: Skills learned through the use of neural reinforcement learning methods in the Brainstormers 2D Simulation League team and the Brainstormers Tribots Middle Size League team, from 2000 to 2008 [21]. Filled dots represent skills that were used in competition, while empty circles are skills that, although successfully learned, were not used in competition.

Chapter 4

Learning Behaviors with the CAMBADA robots

4.1 Introduction

This chapter describes the behaviors targeted for learning within the scope of this thesis, as well as document the details related to their learning procedures, and present results and comparisons with previously existing explicitly-coded behaviors.

4.2 A Reinforcement Learning framework for CAMBADA

The CAMBADA behavior architecture is very flexible, and it is easy to integrate learning behaviors into it. Figure 4.1 shows how learning behaviors are defined as subclasses of a base learning behavior class, which itself inherits from the CAMBADA base behavior class. This allows one to add learning behaviors with different goals, while reusing the common functionality such as, for example, initializing the Q-function approximator or configuring some parameters like the maximum number of learning episodes to be executed.

The learning behavior classes are only concerned with the learning experiments, i.e. the experience gathering itself. Other relevant aspects of learning are extracted into separate classes: the task definition classes. These classes specify the action set, the reward function and stopping conditions. This separation of concerns is useful for when we try to learn the same task through different ways of gathering experience. For example, while learning on the

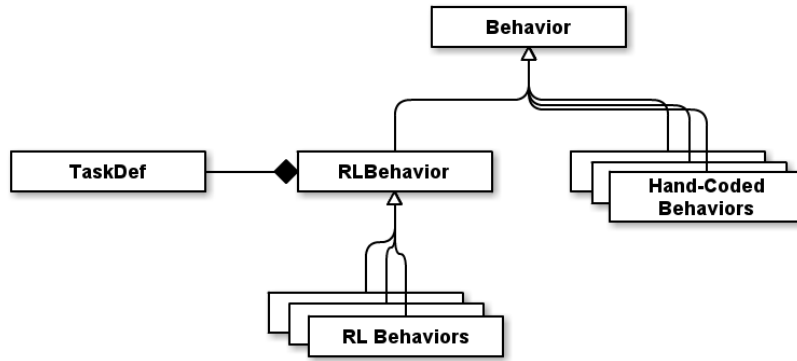


Figure 4.1: Class diagram of the behavior classes.

simulator, it is possible to take advantage of a large field, while such is not possible when learning with the real robots. On the other hand, this also allows us to implement the learning phase as a separate program without duplicating or coupling code. To simplify client code, an abstract factory was introduced. Figure 4.2 shows the class diagram for the task definition classes.

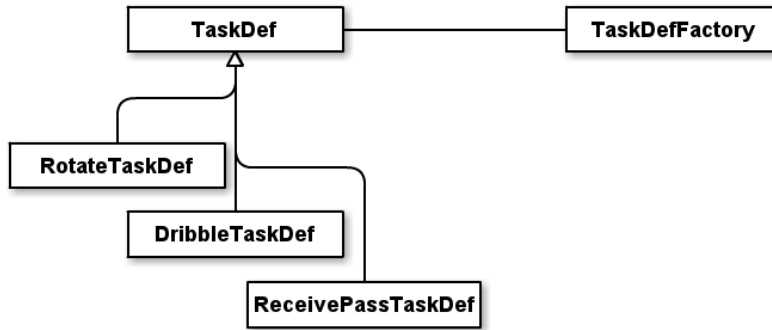


Figure 4.2: Class diagram of the task definition classes.

The learning phase is implemented as a separate program, rather than baked in the learning behavior’s code. This still allows the agent to trigger a learning phase by executing it as a separate process, but it is also possible to instead offload the learning step. Because the Q-function approximation phase can deal with large quantities of data and require high computational resources, it may be worthwhile to instead delegate this processing to faster, more powerful computers. After the learning phase is complete, a small file containing the neural network parameters can be sent back to the computer running the agent code, thus

repeating the cycle.

Since the experience gathering phase and the learning phase are logically separated, there are also two different configuration files for each behavior: the task and the trainer configuration files. The task configuration file defines parameters specific to the experience gathering phase, such as how many episodes should be executed per batch or the maximum number of cycles per episode. The trainer configuration is relevant for the Q-function approximation phase, and it details which update rule to use, the value of the discount factor parameter, how many times the inner loop of the Batch Reinforcement Learning framework should be run and how many neural network training epochs per each inner loop.

4.3 Learning in a simulated environment

Using the simulator developed for the CAMBADA platform [24] proved very useful, as it allowed to validate behavior specifications before proceeding to learn those behaviors with the actual robots.

The simulator was slightly modified to allow an agent to get noise-free data, which was useful for debugging and testing out experimental behavior specifications. Also, an additional field was added to the RTDB to implement a mechanism that allows the agent to signal the simulator to reset to some predefined settings, such as robot and ball positions and velocities, corresponding to the starting conditions of learning experiments. This allows one to execute several batches of experiments in a simulated environment without any human interaction.

A UNIX script was developed to automate the learning process. This script first backs up a copy of the current neural network, renamed to indicate in which learning iteration it was used. Then, it launches the CAMBADA agent, which, with the simulator and basestation already running and properly configured to execute the learning behavior, will execute for the specified amount of learning episodes and gather experience data. After it finishes execution, the NFQ algorithm is run as a separate program to find a new Q-function approximator. These three phases, backing up the most current neural network, gathering experience, and learning a new function approximator, are executed in a loop, either until a number of maximum learning iterations has been reached, or until execution is terminated by the user. By keeping a history of the Q-function approximators used in previous learning iterations, and with the help of a statistics file produced by the learning agent, we can roll back in time and analyze

the performance of the agent throughout the whole learning process.

The CAMBADA robots have a delay between the moment a given command is issued and the moment the actuators carry out that command. To minimize this problem, a prediction-based approach is implemented. However, this delay is not modeled in the simulator, and, as such, the prediction was turned off for experiments conducted within the simulated environment.

4.4 Learning in a real environment

All of the behaviors were learned in CAMBADA’s training field, also called Cambódromo. This field is rather small, and since the robots can move very fast, a precautionary measure was included in the agent code to limit the working area, as pictured in figure 4.3. This prevented collisions with walls, the goal and other obstacles, such as desks and chairs placed at one end of the field, but also limited the already scarce available space, making it harder to collect larger connected trajectories.

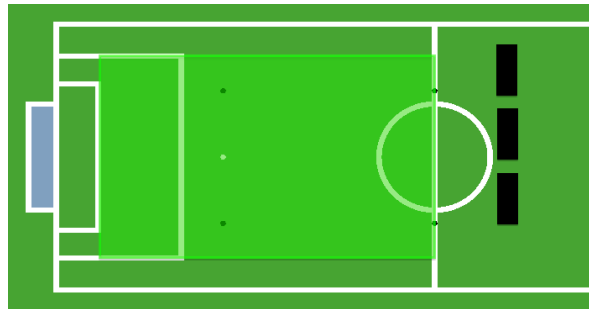


Figure 4.3: The CAMBADA training field. The highlighted region represents the working area. Black rectangles are the approximate location of desks and other obstacles.

4.5 Rotate to an absolute position in the field

4.5.1 Overview

The objective of this behavior is to have the robot rotate to face a given absolute coordinate on the soccer field as fast as possible and hold that orientation.

This behavior is an example of one of the simplest skills to be learned, and, although not

a great technical challenge, it was useful in the early stages of this thesis when starting to learn behaviors within the CAMBADA context, since there were a number of complexities to be dealt with not related to reinforcement learning itself.

4.5.2 Behavior specification

The goal of this behavior is for the robot to face an absolute target coordinate. That coordinate is internally converted to an orientation error relative to the agent.

The behavior is supposed to run until explicitly stopped. This means there are no terminal states.

In order for the agent to determine which action to apply, it needs to know both its angular velocity and its orientation error relative to the absolute target coordinate.

Reward functions can be defined in a very generic way, as proposed in [10]. We define a small subset of the state space as the goal region, denoted as S^+ , corresponding to a near optimal state. In the context of this behavior, S^+ could be the region where the orientation error is within, for example, ± 0.10 radians of the target orientation. Then, the following equation would be used as the reward function:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 0 & \text{if } s_{t+1} \in S^+ \\ 0.01 & \text{if } otherwise \end{cases} \quad (4.1)$$

This kind of reward function causes the agent to reach its target region in the shortest amount of time and remain within it; any other action that does not facilitate this outcome will accumulate a higher cost. There is, however, a problem with this approach. Because of the discrete separation of the state space into S^+ and S^{work} , the learned controller will only strive to reach S^+ in the shortest amount of time, but, once in it, it won't try to achieve a perfect orientation.

To counter this problem, we forego the discrete separation between S^+ and S^{work} , and instead of having a discontinuous reward function, we adopt a continuous one instead. An example of such a function [25] is presented in equation 4.2, and has the benefit of maintaining generality.

$$R(s) = C \times \tanh^2 \left(|s - s^{target}| \times \tanh^{-1} \left(\frac{\sqrt{0.95}}{\delta} \right) \right), \quad (4.2)$$

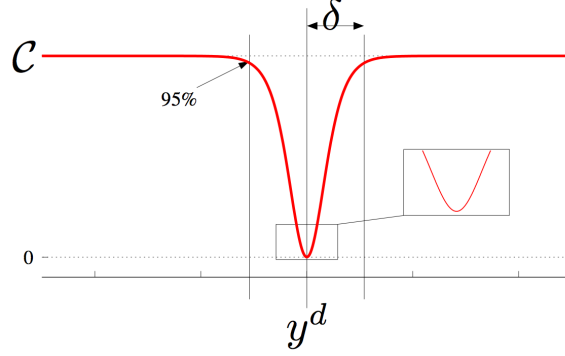


Figure 4.4: Annotated plot of equation 4.2, adapted from [25].

Intuitively, this function can be regarded the cost of a given state, where C is the maximum reward value, δ is the range in which the reward decreases from 95% to 0, and s^{target} is the goal state. For this behavior, the parameters were set as $s^{target} = 0$, $C = 0.01$ and $delta = 0.1$.

The action set is made up of values of target angular velocities to be sent to the robot's motors.

It is possible to reduce the size of the state space by taking advantage of the inherent symmetry of the situation [10]. The problem can be simplified if we only feed the agent the absolute value of its orientation error, and a modified angular velocity value, where whether it is positive or negative means that the velocity is directed to the target orientation or away from it. Additionally, we also restrict the actions to be values greater than or equal to zero, so that the actual command sent to the motors of the robot is always towards minimizing the orientation error. The command, which is a target angular velocity, is modified by the signal of the robot's relative orientation error, meaning that when the orientation error is greater than zero, the action that is returned by the agent is applied directly; otherwise, we apply a negative action with the same magnitude as the one returned. This way, we are able to reduce the orientation error range from $[-\pi, \pi]$ to $[0, \pi]$. We settled for an action set with three actions: $[0, \frac{\pi}{2}, \pi]$ rad/s.

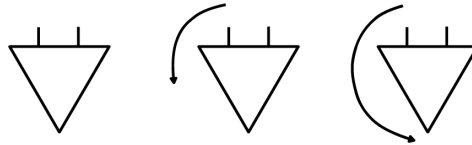


Figure 4.5: Visual representation of the action set used to learn the rotation behavior.

4.5.3 Learning procedure

For this behavior, we chose to use a neural network with 3 input nodes, 2 layers of 10 hidden neurons each, and an output layer of a single neuron. The 3 input nodes account for the 2 state dimensions and 1 action dimension. The pattern set was approximated with 300 epochs of the RPROP algorithm, using 10 inner loops of the NFQ algorithm. No discounting was used, that is $\gamma = 1.0$. The Q-Batch update rule was used, although this behavior served also to compare its performance to the Q-Learning update rule. The comparison of their results will take place in section 4.5.6.

The “hint to goal” heuristic was used, with an artificial experience set consisting the state $\langle \text{angular velocity} = 0, \text{orientation error} = 0 \rangle$, repeated 100 times for each action from the action set, and target output of 0.

Learning experiments were made up of 10 learning episodes, with each episode lasting for 70 control cycles. In the CAMBADA platform, a new control cycles is started every 0.033 seconds, so, since there are no terminal states, each learning experiment accounts for 33 seconds of interaction time.

The learning role created for this behavior executed the specified number of learning episodes, ending execution afterwards. At the beginning of each episode, a new target coordinate is generated. Targets are randomly chosen from a circumference around the robot’s position in order to ensure an uniform probability of amplitudes of starting orientation error. The behavior is then activated for the number of cycles specified for each episode.

4.5.4 Learning results in a simulated environment

A good controller was found after 38 learning iterations. Since every iteration consisted of 10 episodes of 70 cycles each, this means that 26600 experience tuples were gathered, which amounts for around 14,5 minutes of robot interaction time.

Figure 4.6 shows a comparison between the learned behavior and an explicitly coded one, averaged after 10 experiences each. The learned behavior is stiffer. This is because of the reduced number of actions and also because, for situations when the orientation error is small, these are too strong, i.e. the controller is over actuated. As such, the learned controller often overshoots its target and then compensates, and has trouble maintaining such a small orientation error as the explicitly coded behavior.

Nonetheless, an acceptable result is achieved. This goes to show how, even with a less-than-optimal action set, it is possible to learn a suitable controller. However, if we were to use this behavior in competition situations, or if our sole objective was to surpass the performance of the explicitly coded behavior, we would need to find a better action set. As this is only a proof-of-concept, we chose to focus our efforts in the rest of the behaviors.

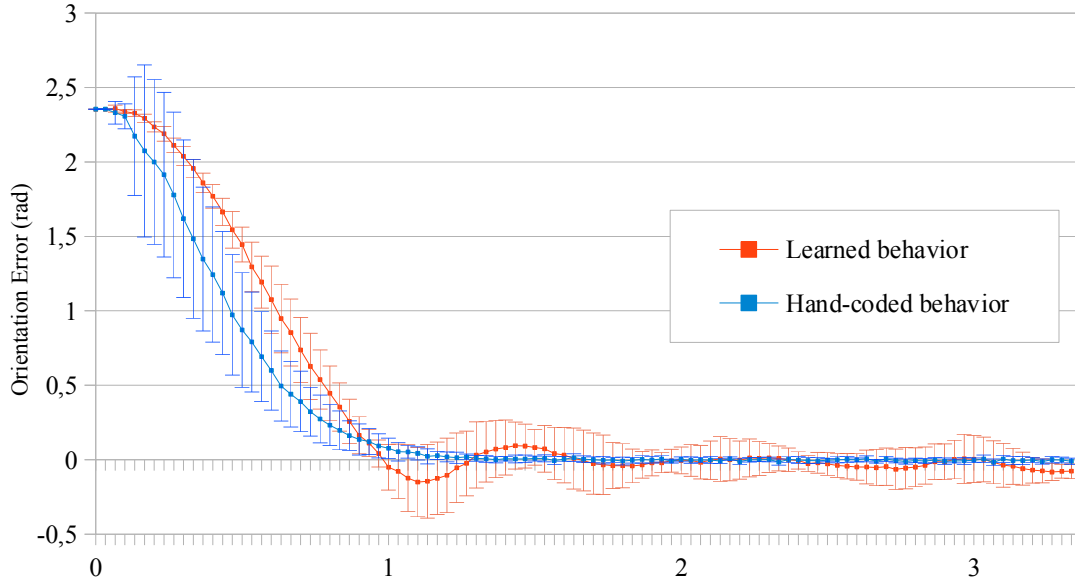


Figure 4.6: Comparison of the robot’s orientation error when using both the hand-coded behavior and the learned behavior.

4.5.5 Learning results in a real environment

The learning process lasted for 24 learning iterations, during which 16800 experience tuples were gathered. This accounts for around 9 minutes of interaction time.

In figure 4.7 we can see a comparison of the absolute orientation error between the explicitly coded behavior and the learned behavior, averaged after 10 experiments. Again, we see that the learned behavior is able to reach the target orientation very quickly. The robot’s omniwheels have a very tight grip, so the robot never overshoots as much as in the simulated environment.

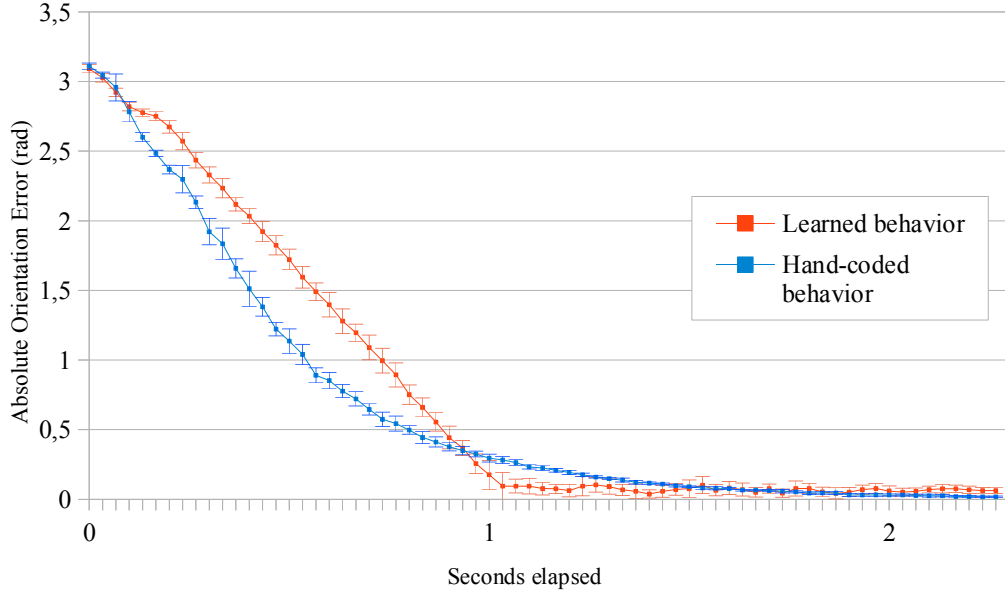


Figure 4.7: Comparison of the robot’s absolute orientation error when using both the hand-coded behavior and the learned behavior.

4.5.6 Comparison between Q-Learning and Q-Batch update rules

Both the Q-Learning and Q-Batch update rules were used in different experiments to learn this behavior. In order to draw conclusions over their performance, 10 learning experiments with a maximum of 50 learning iterations, with one NFQ inner loop per iteration, were carried out for each learning rule. The results were aggregated and are displayed in figure 4.8. As we can see, the Q-Batch update rule is faster to achieve a lower mean cost per cycle. Also, after the first 5 learning iterations, the average results for Q-Learning were never better than the average Q-Batch results. This is an interesting observation, because it shows that Q-Batch can reach better performances than Q-Learning in short learning times. However, it is important to remind ourselves that these experiments were bounded to 50 learning iterations, and so we cannot withdraw conclusions from how these update rules would perform past that limit. There is no guarantee that the higher performance observed would still be visible with, say, 100 learning iterations or more.

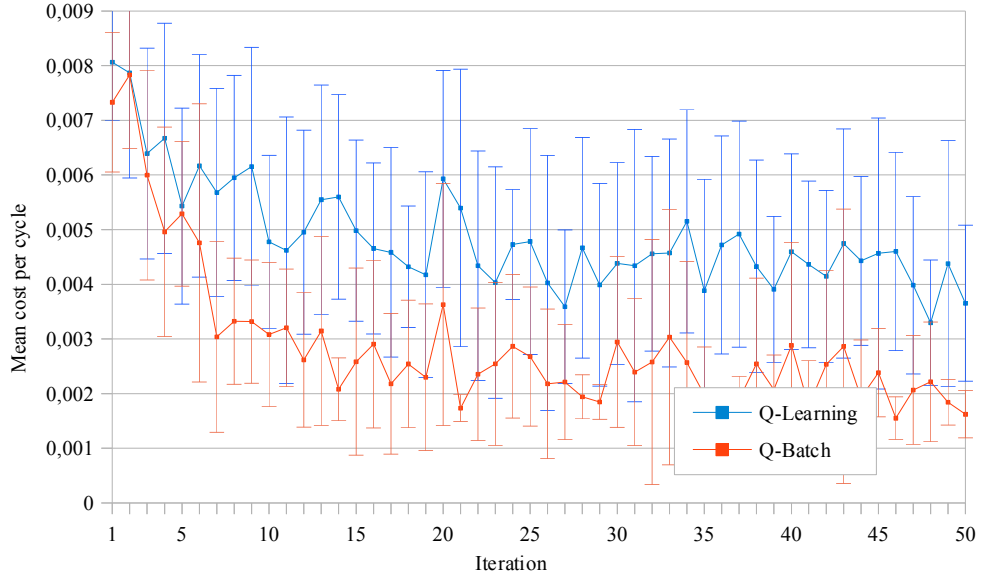


Figure 4.8: Comparison of the mean cost per cycle when using Q-Learning and Q-Batch update rules.

4.6 Dribble the ball in a given direction

4.6.1 Overview

Dribbling a soccer ball is an essential skill for effective soccer robots. The rules dictate that, at most, only a third of the ball may be covered by the robot, so it is a challenging feat.

4.6.2 Behavior specification

The objective of this behavior is to dribble the ball to a given point as fast as possible. That point is internally converted to an absolute direction of movement. The behavior seems to have two phases: first rotate to face the direction of the target point, and then move forward as fast as possible while keeping a low orientation error. The behavior is only activated when the ball is already under the robot's control, and if control of the ball is lost, the task is considered to have failed.

The state vector of this behavior includes the robot's relative orientation error to the target direction, its relative linear and angular velocities and a signal indicating whether the ball is engaged or not. This binary signal is further filtered, so that it is only negative after

five sequential control cycles in which the ball was perceived as not engaged by the robot. This is because the robot may lose control of the ball for only one or two control cycles, but regain control right after. In these situations, we choose not to punish the agent, and instead consider as if control of the ball was never lost.

As was described in section 4.5.2, we can take advantage of symmetry in order to reduce state dimensionality. In this behavior, we again only give the agent the absolute value of the orientation error, and use its sign to modify actions before they are applied.

The action set used to learn this behavior was adapted from a similar experience conducted by Riedmiller [10]. Actions are 3-tuples consisting of desired relative velocity values to be sent to the robot's motors, and follow the form $\langle v_x, v_y, v_\theta \rangle$, where the y axis points to the front of the robot. Six actions were defined: $\langle 0.0, 2.5, 0.0 \rangle$, $\langle 0.0, 2.0, 2.0 \rangle$, $\langle 0.0, 1.5, 2.5 \rangle$, $\langle 1.5, 1.5, 2.5 \rangle$, $\langle 1.0, 1.0, 3.0 \rangle$ and $\langle -1.0, 1.0, 3.0 \rangle$.

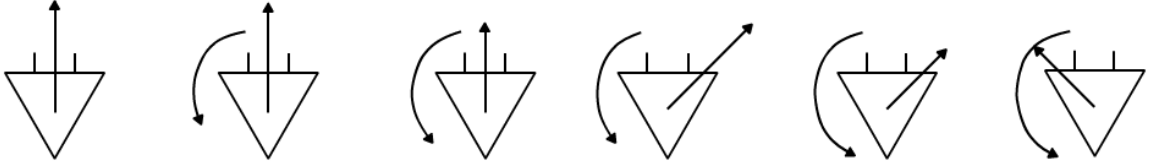


Figure 4.9: Visual representation of the action set used to learn the dribble behavior.

The reward function used to learn this behavior follows the same approach as described in section 4.5.2 for the rotation behavior, with $s^{target} = 0$, $C = 0.01$ and $delta = 0.1$.

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \text{if } s_{t+1} \in S^- \\ 0.01 \times \tanh^2 \left(\text{ORIErrror} \times \tanh^{-1} \left(\frac{\sqrt{0.95}}{0.1} \right) \right) & \text{if } otherwise \end{cases} \quad (4.3)$$

4.6.3 Learning procedure

The neural network topology used for this behavior consists of 8 input nodes, 2 hidden layers of 20 nodes each and one single output node. The pattern set was approximated over 300 epochs of the RPROP algorithm. A discounting factor of $\gamma = 0.99$ was used, since the immediate cost of a state is only zero when the robot's orientation error is exactly zero. If no discounting was used, the Q -values would drift to the maximum value, because, in practice,

it would be impossible to avoid collecting more costs. The Q-Batch update rule was used to generate the pattern set, and each learning iteration consisted of 10 inner loops of the NFQ algorithm.

Experience was gathered in sets of 15 episodes. Episodes lasted for a maximum of 100 cycles, but would terminate earlier if the robot lost the ball during the experiment. An artificial set of “hint” transition tuples was introduced, with state $\langle \text{ORIErrOR}, v_X, v_Y, v_A, \text{BALLENGAGED} \rangle = \langle 0, 0, 2.5, 0, 1 \rangle$, repeating 100 times for each action of the action set, and target output of zero.

For each experience-gathering experiment, the robot was positioned at the center of the field, and the ball was placed randomly in a circumference around the robot, to ensure a uniform distribution of total span of rotation to face the target direction. The robot moved to gain control of the ball, and a learning episode would start. If the robot lost the ball, the agent would activate the reset signal, thus repositioning itself and the ball. If, instead, the maximum number of control cycles was reached without losing control of the ball, the following episode would start without resetting the environment; the target direction of movement would be flipped 180 degrees, and the episode would start right away.

4.6.4 Learning results in a simulated environment

After 34 learning iterations, a good controller was learned. Around 17500 experience tuples were gathered during learning, which means the controller was learned under 10 minutes of interaction time.

Figure 4.10 presents a comparison of both the hand-coded behavior and the learned behavior for a specific test case. The initial starting conditions were the same for both behaviors, since we can customize the simulator to enforce certain specified conditions. In each test trial the robot started in the center of the field. It then grabbed hold of the ball and moved forward 2 meters, at which point the dribbling behavior was activated. The target point was specified as to require the robot to turn 180 degrees.

As observed, the learned behavior is much more sharper than the hand-coded one. In fact, using the learned behavior, the robot is able to rotate around the ball without losing it, even while carrying significant speed. This way, the robot could complete a full 180 degree turn in less space. When facing competitive teams, robots rarely find all the space they need to perform their maneuvers, as opposing playes are usually quick to close down on advancing

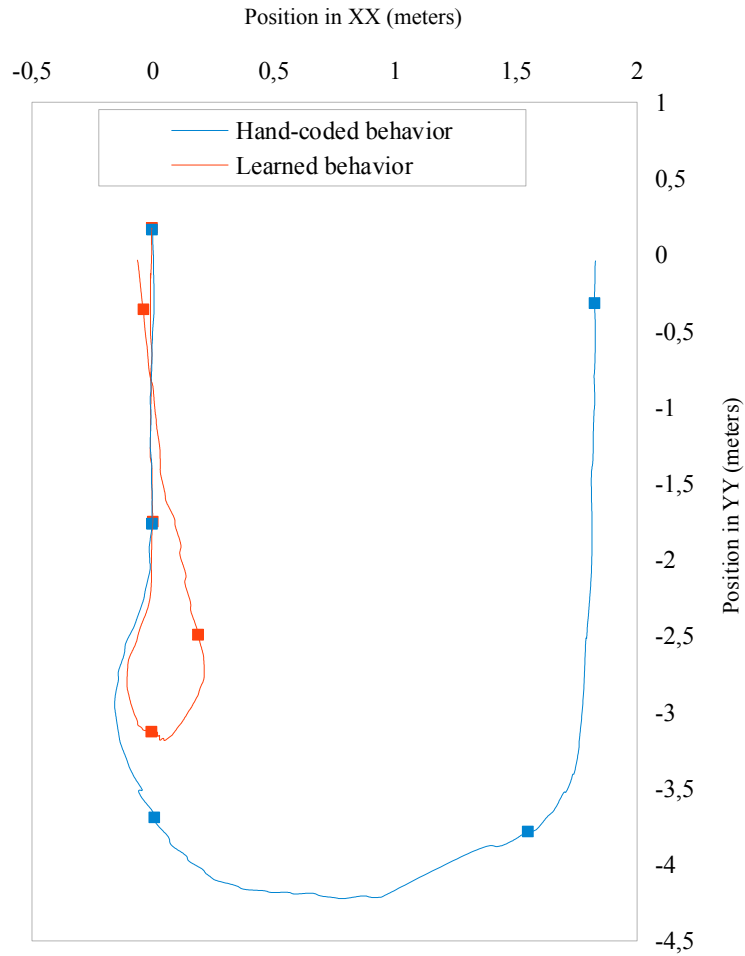


Figure 4.10: Comparison of the hand-coded and the learned behavior for a specific test case. Squares are placed on each trajectory every second, allowing for a temporal comparison. For each trial, the robot started in the $(0, 0)$ position. The dribbling behavior kicked in after the robot reached an YY position of less than -2 meters.

attackers.

Out of 10 test trials, the learned behavior lost control of the ball 5 times, while the hand-coded managed to complete the 10 test trials without ever losing control of the ball. This is understandable, since the hand-coded behavior is much softer. While this is less than ideal, we should consider just how important is the percentage of success for a dribbling behavior. Robotic soccer matches are very dynamic and can be very fast paced. During a game, many attempts to break into counter attack can happen. Robots may manage to never

lose control of the ball, but can fail to reach advantageous positions because they were too slow. Alternatively, if the robot loses the ball, then the chance for an attack was lost, and the opposing team may lose control of the ball, which is never good. However, it is possible to combine an aggressive dribbling behavior for when speed is more important than precision and a smooth dribbling behavior for when possession is of higher importance. In order to take advantage of counter attack situations, robots need to be fast, and since these plays are not usually well structured, precision has lesser importance. But if a team is leading the score and wishes to avoid defensive errors, then players should not take unnecessary risks and should avoid losing possession.

4.6.5 Learning results in a real environment

A suitable controller was found after 18 learning iterations. During learning, the robot collected 6535 experience tuples, which means only a little over 3.5 minutes of experimentation were needed. The whole learning process took around 1,5 hours, including batch training, preparation and execution of learning experiments.

To compare the performance of the learned behavior with the explicitly coded one, a test was carried out. For both tests, the robot was placed at approximately the same coordinate and orientation. The test consisted of the robot moving forward until its YY position surpassed -2.5, and then turning around 180 degrees. The test ended when the robot's YY position was less than -4. The robot carried a speed of around 1,2 m/s when the dribble behaviors took control.

Figure 4.11 shows the performance of both behaviors. A square is placed every second on each trajectory, in order to compare how long each behavior took to finish the test. As we can see, the learned behavior is sharper, and was able make a harder turn. Additionally, it achieved faster speeds after having reached the target orientation, and completed the test after 134 control cycles. The explicitly coded behavior, on the other hand, is softer and needs more space to achieve the same curve, and never moves as fast once it is facing the target direction. It finished the test after 164 cycles.

10 testing trials were carried out for both the hand-coded behavior and the learned behavior. While using the learned behavior, the robot lost control of the ball 3 times, while it only lost control once when using the hand-coded behavior. Considering the fact that the

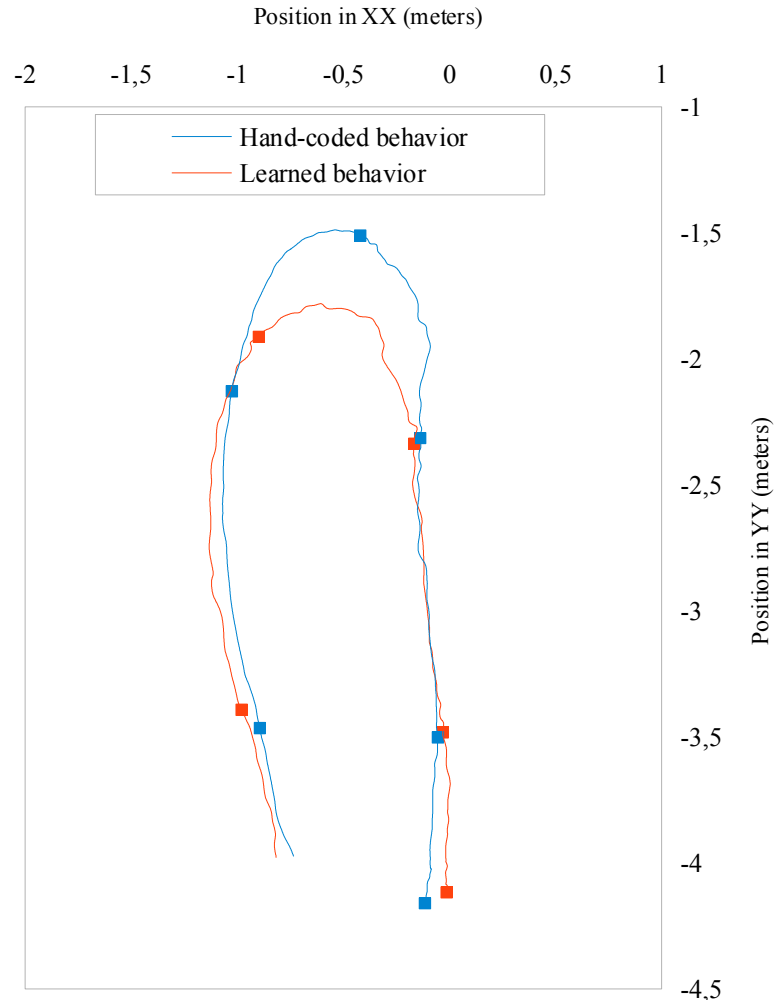


Figure 4.11: Comparison of the hand-coded and the learned behavior for a similar test case. Squares are placed on each trajectory every second, allowing for a temporal comparison. For each trial, the robot started close to the (0, -4) position. It then grabbed the ball and moved forward. The dribbling behavior kicked in after the robot reached an YY position of greater than -2.5 meters.

learned controller is faster, this should not be a disappointing result. However, the same considerations as were presented in the previous section apply, i.e. that it is possible and that there are benefits to using both of these controllers in game situations.

4.7 Receive a Pass

4.7.1 Overview

Many game situations require a player to pass the ball to a teammate. While the passer needs to kick the ball accurately to reach its teammate, the receiver needs to gain control of the ball without losing the advantage of its position, i.e. without moving too much around the field.

4.7.2 Behavior specification

This is the behavior that should be executed by the passee. In ideal conditions, the passer would kick the ball so accurately that the ball would move directly to the passee's position. On the other hand, the passee may need to compensate for the ball's velocity. If the ball approaches the passee too fast, it should move backwards as it is about to engage the ball, so the grabber can grasp the ball without any rebound. If instead the ball does not carry enough velocity to reach the passee, the robot should approach it. The objective of this behavior is for the passee to position itself in the ball's path and compensate for excess speed, with the minimum required movement, so as to not forfeit its current position. Since what we are trying to optimize is only the linear movement of the agent, we chose to delegate the task of facing the ball to a PID, thus simplifying the problem.

For such a behavior to be learned, the learning agent would need to know the position and velocity of the ball and the agent, as well as whether the ball is engaged or not, yielding a state vector with 9 variables: $\langle \text{BALLPOS_X}, \text{BALLPOS_Y}, \text{BALLVEL_X}, \text{BALLVEL_Y}, \text{ROBOTPOS_X}, \text{ROBOTPOS_Y}, \text{ROBOTVEL_X}, \text{ROBOTVEL_Y}, \text{CYCLESENGAGED} \rangle$. After careful analysis of the problem, it becomes obvious that we can reduce the state space if instead of considering coordinates as global, i.e. relative to a global axis, we consider them on the axis of movement of the ball. This way, we can represent the ball's velocity as a scalar value, instead of a vector. Additionally, we can define the origin of the axis of movement to be the ball, thus removing the need to include the position of the ball in the state vector. When the velocity of the ball becomes too small, the robot perceives the ball's movement too noisily. Therefore, we define a threshold, so that when the ball moves slowly we consider the axis to be pointing from the ball towards the robot, with its center in the ball. The threshold was defined as

$0.3m/s^2$. Following this strategy, we reduced the state space from 9 dimensions to 6. The final formulation of the state vector is $\langle \text{BALLVEL}, \text{ROBOTPOS}_X, \text{ROBOTPOS}_Y, \text{ROBOTVEL}_X, \text{ROBOTVEL}_Y, \text{BALLENGAGED} \rangle$.

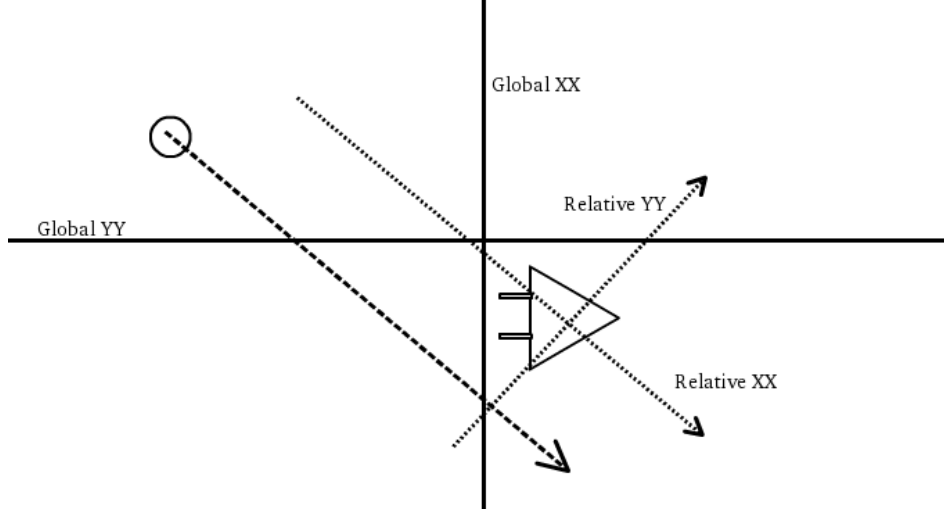


Figure 4.12: We can reduce the state space by projecting the robot’s coordinates onto relative XX and YY axes. The XX axis is parallel to the ball’s direction of movement, while the YY axis is perpendicular to it.

In a similar fashion to the previously discussed behaviors, we take advantage of symmetry to further reduce the state space. Given our formulation of the state vector, we can feed the learning agent the absolute value of the robot’s position in the YY component of the axis of the ball’s movement, and then modify the agent’s actions according to its sign. Also, the ROBOTVEL_Y state variable is altered so that its sign indicates whether it points to $YY = 0$ or not.

The fact that we model the problem by considering coordinates as being in the axis of movement of the ball allows us to simplify actions and reduce the size of the action set. Actions are defined to be 2-tuples of the form $\langle v_x^{rel}, v_y^{rel} \rangle$, which correspond to target velocity values in the axis of movement of the ball. This means that actions should have the same effect regardless of the orientation of the robot, thus taking advantage of the CAMBADA robots’ ability to move holonomically. The action set includes 6 actions: $\langle 0, 0 \rangle$, $\langle 0, -1 \rangle$, $\langle 1, 0 \rangle$, $\langle 1, -1 \rangle$, $\langle 0, -0.5 \rangle$ and $\langle 1, -0.5 \rangle$.

The reward function needs to express our wish of reducing the absolute value of the robot’s

relative YY position in the axis of the ball’s movement, as well as the intention of reducing movement in the x axis. On the other hand, we want to avoid situations where the robot is unable to intercept the ball. Such situations are characterized by a large negative relative x position. S^+ states correspond to when the robot has gained control of the ball for a certain number of consecutive control cycles, which we set as 5. S^- states include all states in which the robot’s relative x position is less than -0.5 meters. All other states belong to S^{work} . While S^+ states are terminating states, S^- states are not, since the robot may still act to engage the ball. Equation 4.4 represents the reward function used. The S^{work} condition expresses our desire for the robot to reduce its relative y position, while keeping movement in the x axis to a minimum. Also, an additional penalty is given to motivate the robot to gain control the ball.

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 0 & \text{if } s_{t+1} \in S^+ \\ 1 & \text{if } s_{t+1} \in S^- \\ 0.01 + r_{posY} + r_{velX} & \text{if } otherwise \end{cases} \quad (4.4)$$

$$r_{posY} = 0.01 \times \tanh^2 \left(\text{ROBOTPOSY} \times \tanh^{-1} \left(\frac{\sqrt{0.95}}{0.1} \right) \right)$$

$$r_{velX} = 0.01 \times \tanh^2 \left(\text{ROBOTVELX} \times \tanh^{-1} \left(\frac{\sqrt{0.95}}{0.1} \right) \right)$$

4.7.3 Learning procedure

The neural network used to approximate the Q -function has as its topology 8 input nodes, 2 hidden layers of 20 nodes each and an output layer with a single node. The Q-Batch update rule was used to generate pattern sets, with a discount factor of $\gamma = 1.0$. The pattern set was approximated over 300 RPROP epochs. 10 NFQ inner loops were executed in each learning iteration.

The “hint-to-goal” heuristic was used. The artificial pattern set included examples with input state as $\langle \text{BALLVEL}, \text{ROBOTPOS_X}, \text{ROBOTPOS_Y}, \text{ROBOTVEL_X}, \text{ROBOTVEL_Y}, \text{CYCLESENGAGED} \rangle = \langle 0, 0, 0, 0, 0, 5 \rangle$, repeated 100 times for each action of the action set, and with target output of zero.

A set of 10 learning episodes took place in each learning iteration. Each episode was limited to 100 control cycles, but would end earlier if a terminal state, i.e. a S^+ state, was reached.

Each episode started with the robot positioned in the center of the field and with the ball placed randomly, being the initial distance between them no less than 4 meters and no greater than 6 meters. The agent communicates it is ready to start the experiment, and so the ball starts moving to a target location, which is determined randomly around the robot, being at most 1 meter away from it. This way, we try to mimic situations where the passer produces a somewhat inaccurate kick. We allow for some control cycles to pass before starting the actual learning episode, to reduce noisy measurements of the ball velocity.

4.7.4 Learning results in a simulated environment

34 learning iterations were needed to for a good control policy to be derived. The learning agent needed less than 9 minutes of interaction time, and a total number of gathered experience tuples close to around 16000.

Taking advantage of the CAMBADA simulator, a repeatable test case was set up to compare the performance of both the learned and the hand-coded behaviors. This test case is similar to the learning experiments where experience data was gathered, except instead of randomizing the ball's target point and velocity, these parameters were fixed to allow for a fairer comparison. Figure 4.13 shows the gathered results of two test trials. For each plot, both the robot's and the ball's trajectories are represented. Contrary to what was presented in the sections 4.6.4 and 4.6.5, both squares and circles are placed on the trajectories every third of a second, so that squares represent the position of the robot and circles the position of the ball. Also, circles with a dark outline mean the ball is under the control of the robot.

As it is visible from figure 4.13, the learned behavior does not move backwards as much as the hand-coded one, which is in agreement with what we specified in the reward function presented in section 4.7.2.

The previously described test case was repeated ten times for each behavior in order to draw a comparison of the percentage of success of each behavior. The results show that the hand-coded behavior is a "safer" behavior than the learned one, as it was able to grab the ball eight times. On the other hand, the learned behavior only managed to successfully receive

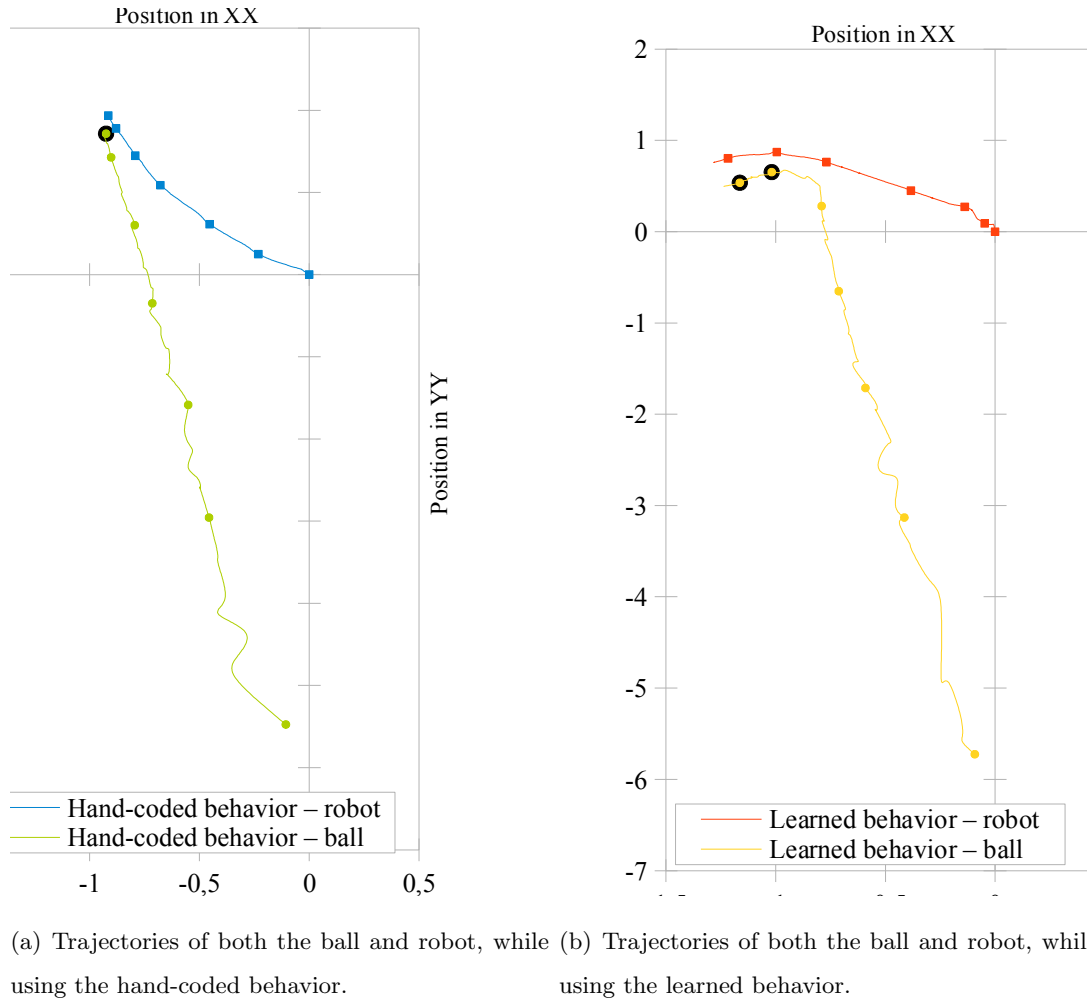


Figure 4.13: Comparison of the hand-coded and the learned behavior for a specific test case. Squares and circles, representing, respectively, the position of the robot and of the ball, are placed on each trajectory every third of a second, allowing for a temporal comparison. Circles with a strong outline signify the ball is grabbed by the robot. The receiving behavior is enabled shortly after the robot detects that the ball started moving.

the ball five times. This is far from perfect, as failing to receive a pass most often leads to unfavorable consequences. However, the case can be made that both behaviors can be used alongside one another, as, sometimes, the robot does not have all the space it needs to execute a more stable, but also wider, maneuver. For example, maybe the robot is close to a side line, and cannot back up as much as the hand-coded behavior would require it to. In these situations, using this learned behavior can be advantageous, as it requires less maneuvering

space, and the odds of successfully gaining control of the ball are already low.

4.7.5 Learning results in a real environment

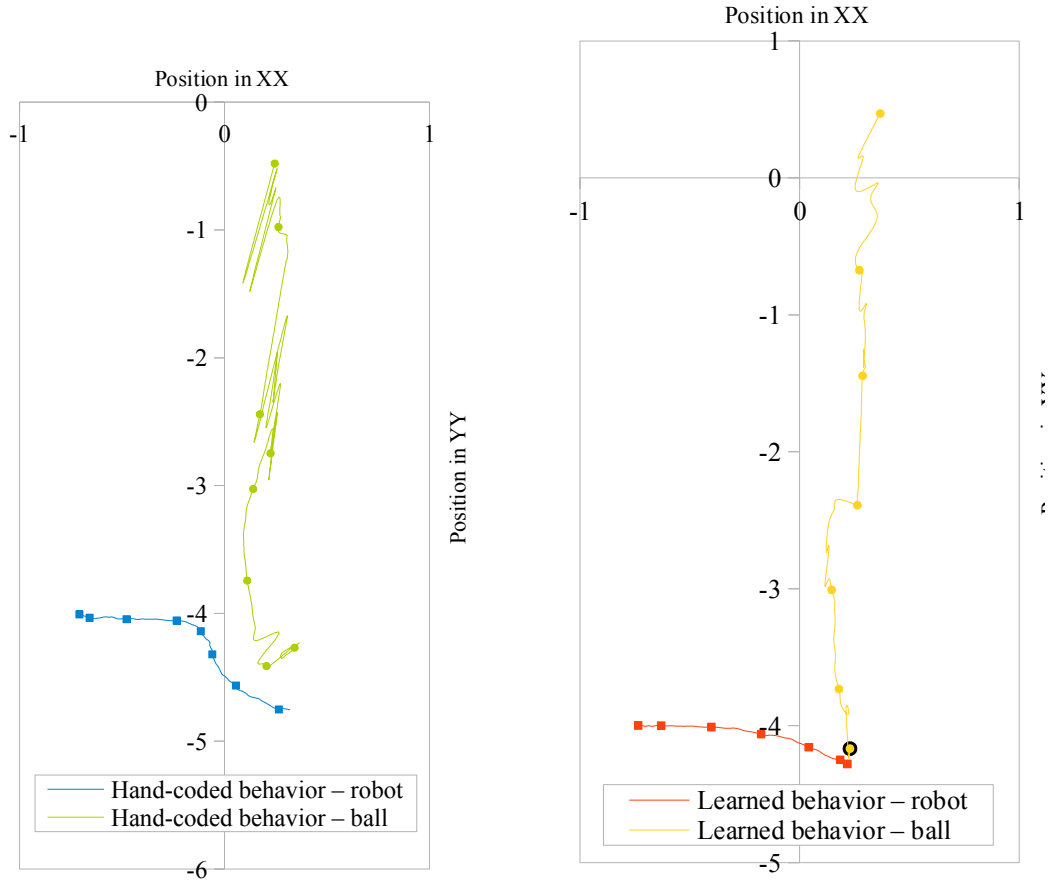
It took 28 learning iterations to find a suitable controlling policy. This amounted to close to 12 minutes of interaction time, during which 21303 experience tuples were collected. The actual time involved, including preparation and off-line Q-function approximation times was around 2 hours.

A test situation was devised to compare the performance of this learned behavior with its hand-coded counterpart. When experimenting with the simulator, it is easy to set up repeatable test situations. In the real world, however, this is impossible. The initial ball speed is never exactly the same, as well as its direction and starting point. Nonetheless, we believe some conclusions can still be withdrawn from a less than perfect testing situation.

The testing situation mimics the learning experiments carried out during the experimentation phase of the Batch Reinforcement Learning workflow. The robot was placed at approximately the same starting orientation and position on the field. The ball was released from a ramp, in such a direction as to require the robot to move sideways to catch it. The initial ball direction and speed were also attempted to be the same. Figure 4.14 shows the results gathered and allows a comparison of both behaviors. Just like the figure presented in the previous section, both squares and circles are placed on the trajectories every third of a second to enable a temporal comparison, and a dark outlined circle means the robot has the ball under control.

As it is visible, the learned behavior is faster to move its final position, where it grabs the ball safely. The hand-coded behavior, on the other hand, starts by moving sideways, but then changes the direction of its movement and starts backing up. In this trial, the hand-coded behavior failed to grab the ball, as it wasn't aligned with its trajectory at the moment of impact.

In order to draw a more educated comparison, this test case was repeated 10 times for each behavior. For this situation, the hand-coded behavior showed very poor performance, failing to grab the ball nine times. The learned behavior performed better, as the robot grabbed the ball five times. These results are somewhat surprising and contrast with the performance analysis presented in the previous section for a learned behavior in a simulated environment.



(a) Trajectories of both the ball and robot, while using the hand-coded behavior. (b) Trajectories of both the ball and robot, while using the learned behavior.

Figure 4.14: Comparison of the hand-coded and the learned behavior for a specific test case. Both the robot's and the ball's trajectories are marked with squares and circles every third of a second. A circle with a strong outline signifies the robot has grabbed the ball. The receiving behavior was enabled shortly after the robot detected that the ball started moving.

Two possible explanations arise: that the simulator has considerable error and is not well tuned, or that the test situation exposes a limitation of the hand-coded behavior. If the latter is true, then it shows the potential of automated learning to overcome limitations of hand-coded behaviors.

One might argue that the testing situation is not general enough. Due to the space constraints of the CAMBADA training field, it is hard to test a wide range of situations. Further

testing should be carried out when a larger field becomes available, such as during competitions like the Portugal Robotics Open (Robótica) and RoboCup.

Chapter 5

Conclusion

Three behaviors were successfully learned, all of which demonstrated how automated learning methods can exploit advantages that are too hard for a programmer to specify in code, or even to realize.

Through the use of Batch Reinforcement Learning we were able to derive control policies that rival, and even surpass, the performance of the hand-coded behaviors, with minimal input and without specifying how such behaviors were to be carried out. Not only is this important for the improvement of behaviors that are currently hand-coded, as it also enables exploration into potential behaviors that are too hard to be programmed explicitly.

Every currently existing hand-coded CAMBADA behavior is a candidate for improvement through Reinforcement Learning. On the other hand, learned behaviors can also complement the hand-coded ones, instead of replacing them altogether. This brings more options to the decision layer, enabling more complex strategies.

The use of the CAMBADA simulator helped testing out different formulations of the same task, such as experimenting with different action sets and reward functions. The fact that it is possible to run experiments in a loop allows one to test many variations and find which worked best. Sometimes, though, the conclusions didn't carry on to the real world. This is due to the degree of modeling error inherent to the simulator. One needs to be careful not to optimize the learning behavior for the simulator, but instead to be general enough that it will perform well on both the simulated and the real robots and environments.

The Q-Batch update rule was extensively used in the work presented in this thesis, since it is important to test these things in real applications. CAMBADA is not just a competitive

robotic soccer team, it is also a research project that fosters innovation. Nonetheless, this thesis is not about Q-Batch, so it does not aim to prove that it is superior to Q-Learning or other update rules.

This work laid the groundwork for more experimentation with Batch Reinforcement Learning in the CAMBADA team. The set of learning behaviors, task definition classes, configuration files and the trainer application will hopefully reduce the amount of work needed in the future to improve and learn behaviors.

The success of the application of Batch Reinforcement Learning to robotic soccer sets an optimistic outlook on what can be achieved in other areas. These algorithms can be applied to other problem domains, perhaps increasing the feasibility of applying robotic systems for every-day problems and activities.

5.1 Future work

Although the learned behaviors showed very good results, it is important to notice that more testing is needed to determine more accurately the performance boost brought by them. The test cases set up to sample the performance of the dribbling and the pass receiving behavior were limited by space constraints. Once the team moves to a more spacious training field, or when it participates in tournaments, it will be possible to test a wider range of situations. Ideally, these behaviors would also be used in game situations, thus collecting larger quantities of data with greater relevance.

The behaviors learned during this work may also be improved by experimenting with different action sets and reward functions. On the other hand, this would be time-consuming, and effort would be better spent on other matters.

Q-Batch showed promising results, but an in depth testing would be needed to be able to draw general conclusions about its performance.

Other Q-function approximators may be more suited for Batch Reinforcement Learning than neural networks. An example of such would be gaussian processes. The framework developed may need to be adapted to accommodate different approximators.

While three behaviors were learned, only two of them are relevant for in-game situations. More behaviors can be targeted for learning through Batch Reinforcement Learning.

The focus of this thesis was on individual agent behaviors. Future exploration could be

directed at using Reinforcement Learning in higher level contexts, such as the decision layer or the strategy layer. For example, an agent could learn which behavior would be the most suitable in a given game situation.

Experience gathered in the simulated environment was never used to learn the same behavior in a real setting. It would be interesting to explore ways of reusing the experience data, either fully or partially. A possible scheme could mix both the simulated and the real data before the Q-function approximation step. It would start with only simulated experience, and as more real data becomes available, less and less simulated experience tuples would be mixed in. Ideally, this would lead to less interaction time with the real environment, as, in most systems, it is expensive. Again, the simulation error, even if only minimal, could have a big impact.

Bibliography

- [1] Stuart Russel and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, USA, 3rd edition, 2010.
- [2] Cambada mechanical drawings, 2012.
- [3] António J. R. Neves, José Luís Azevedo, Bernardo Cunha, Nuno Lau, João Silva, Frederico Santos, Gustavo Corrente, Daniel A. Martins, Nuno Figueiredo, Artur Pereira, Luís Almeida, Luís Seabra Lopes, Armando J. Pinho, João Rodrigues, and Paulo Pedreiras. Cambada soccer team: from robot architecture to multiagent coordination. In Vladan Papić, editor, *Robot Soccer*, chapter CAMBADA soccer team: from robot architecture to multiagent coordination, pages 19–45. InTech, January 2010.
- [4] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L. Seabra Lopes. Coordinating distributed autonomous agents with a real-time database: the CAMBADA project. In *Proceedings of the ISCIS 2004, the 19 th Int. Symp. on Computer and Information Sciences*, 2004.
- [5] A. J. R. Neves, J. L. Azevedo, B. Cunha, J. Cunha, R. Dias, P. Fonseca, N. Lau, E. Pedrosa, A. Pereira, and J. Silva. Cambada’2012: Team description paper. 2012.
- [6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd, in progress edition, 2012.
- [7] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.
- [8] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, May 1989.

- [9] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.
- [10] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Auton. Robots*, 27(1):55–73, 2009.
- [11] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State-Of-The-Art*, chapter Batch Reinforcement Learning, pages 45–73. Springer Berlin Heidelberg, 2012.
- [12] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *NIPS*, pages 369–376. MIT Press, 1994.
- [13] Martin Riedmiller. Neural reinforcement learning to swing-up and balance a real pole. In *SMC*, pages 3191–3196. IEEE, 2005.
- [14] Martin Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005.
- [15] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, pages 586–591, 1993.
- [16] João Cunha, Nuno Lau, and António Neves. Q-batch: initial results with a novel update rule for batch reinforcement learning. *Advances in Artificial Intelligence - Local Proceedings, EPIA 2013 - XVI Portuguese Conference on Artificial Intelligence, Angra do Heroísmo, Azores, Portugal, 9-12 September*, (1):240–251, 2013.
- [17] Martin A. Riedmiller, Artur Merke, David Meier, Andreas Hoffmann, Alex Sinner, Ortwil Thate, and R. Ehrmann. Karlsruhe brainstormers - a reinforcement learning approach to robotic soccer. In Peter Stone, Tucker R. Balch, and Gerhard K. Kraetzschmar,

- editors, *RoboCup*, volume 2019 of *Lecture Notes in Computer Science*, pages 367–372. Springer, 2000.
- [18] Artur Merke and Martin A. Riedmiller. Karlsruhe brainstormers - a reinforcement learning approach to robotic soccer. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup*, volume 2377 of *Lecture Notes in Computer Science*, pages 435–440. Springer, 2001.
 - [19] Roland Hafner and Roland Riedmiller. Reinforcement learning on a omnidirectional mobile robot. In *In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), Las Vegas*, 2003.
 - [20] Thomas Gabel and Martin Riedmiller. Learning a partial behavior for a competitive robotic soccer agent. *KI Zeitschrift*, 20:18–23, 2006.
 - [21] Martin Lauer, Roland Hafner, Sascha Lange, and Martin Riedmiller. Cognitive concepts in autonomous soccer playing robots. *Cognitive Systems Research*, 11(3):287–309, 2010.
 - [22] Thomas Gabel, Martin A. Riedmiller, and Florian Trost. A case study on improving defense behavior in soccer simulation 2d: The neurohassle approach. In Luca Iocchi, Hitoshi Matsubara, Alfredo Weitzenfeld, and Changjiu Zhou, editors, *RoboCup*, volume 5399 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2008.
 - [23] Heiko Müller, Martin Lauer, Roland Hafner, Sascha Lange, Artur Merke, and Martin Riedmiller. Making a robot learn to play soccer using reward and punishment. In Joachim Hertzberg, Michael Beetz, and Roman Englert, editors, *KI*, volume 4667 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2007.
 - [24] Eurico Farinha Pedrosa. Simulated environment for robotic soccer agents. Msc. thesis, University of Aveiro, 2010.
 - [25] Martin Riedmiller. How to train neural controllers. *Robot Learning Summer School*, 2009.

