Generation of Motion Policies Applying Multiagent Reinforcement Learning in Simulated Robotic Soccer

David García and Leonardo Garrido
Department of Computer Science, Tecnológico de Monterrey, Campus Monterrey, México

1 Introduction

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

Machine learning is one of the most ambitious goals of AI. One of the main features of an intelligent agent is that it can adapt to the unknown environment. Learning from interaction with the environment is a foundational idea underlying nearly all theories of learning and intelligence. Reinforcement Learning (RL) is a learning type that allows that an agent acquires knowledge through the continuous interaction with its environment (Sutton & Barto, 1998).

RoboCup (RoboCup, 2011) is an international initiative that fosters research and education in robotics and AI, on multi-robot systems in particular, through competitions of RoboCup Soccer, RoboCup Rescue, RoboCup Home and RoboCup Junior. The ultimate goal of the RoboCup soccer project is by 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer. RoboCup chose to use soccer game as a central topic of research, aiming at innovations to be applied for socially significant problems and industries.

RoboCup currently includes a number of different robot soccer leagues (or categories) that focus on different research challenges. One of these leagues is the RoboCup Standard Platform League (SPL), which all teams compete with identical robots, the Nao robot. In addition to the SPL, RobotStadium is a parallel simulated league, where contestants can battle simulated Naos against each other. RobotStadium league is based on the RoboCup SPL and uses the Webots simulator by Cyberbotics (Cyberbotics, 2011).

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

This research work is part of the Intelligent Autonomous Agents research group from the Computer Science Department at Tecnológico de Monterrey, Campus Monterrey, which objective is to develop innovative
technology oriented towards distributed knowledge handling by researching about agent technologies and multi-agent systems. In the RoboCup domain, the research interest is coordination and cooperation among autonomous agents. Research team has an extensive experience in the simulation leagues of RoboCup, it has presented some research works related to the RoboCup 3D simulation league. A fuzzy bayesian approach for decision making in RoboCup Simulation 3D was presented in the RoboCup Symposium (Bustamante et al., 2006b). Later, in (Bustamante et al., 2006a) as published a comparison between fuzzy bayesian classifiers and gaussian bayes classifiers. Afterwards, a hybrid Monte-Carlo localization with Kalman filter sensor fusion approach was used for diminishing the effect of noise and uncertainty in the agent self localization process, and was published in (Bustamante & Garrido, 2007). Starting in 2008, with the launch of RobotStadium, our research team have been working primarily with the Webots simulator in the RobotStadium simulated robotic soccer league. Actually, in 2009 our Borregos-Nao team finished in fourth place in the RobotStadium competition and in 2010 we climbed to the second position (RobotStadium, 2011).

This research work implements RL in the RobotStadium simulated robot soccer competition in order to generate motion policies. Specifically, this work implements the Connectionist Q-Learning Framework (CQLF) (Kapusta, 2011) which uses a backpropagation neural network as a method of generalization. The framework is implemented to generate motion policies, where the goal is to learn how to move a group of servos in order to achieve a motion behavior. This problem is first addressed in a single agent approach where a single agent learns to move all the motors and then in a multiagent approach where every motor is a learning agent and the resulting individual policies conform the motion behavior.

This chapter is divided in seven sections. Current section presents an introduction. Section 2 states the problem definition and objectives. Section 3 describes the related work. Section 4 defines the testbed environment and section 5 the proposed solution method. Section 6 contains the implementation of the solution method to the learning motion policies problem. Finally, section 7 shows future work and section 8 obtained conclusions and contributions.

2 Problem definition and objectives

When designing intelligent autonomous agents, it is impossible to implement all the potential situations an agent may encounter and specify an optimal behavior in advance. An intelligent agent must be capable of using its past experience to learn how its actions affect the environment in which it is situated. The classical design of these agents is a laborious effort, requiring many hours of experimental work, designing, testing, and redesigning the control program until the desired behavior is achieved.

One of the main problems in the development of an intelligent robot (agent) that plays soccer is the design of motion behaviors. Motion behaviors allow the robot to perform basic movements to play soccer like walk, turn, shoot, stand up, etc. Implementing these behaviors is a hard task mainly due to the large and continuous state space and the parallelism in the movement of the motors.

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

This research work aims to implement RL in the international “RobotStadium” simulated robotic soccer competition in order to generate motion policies through the experience of the agent in its environment. With this approach the designer specifies the desired behavior of the agent through rewards, rather than the control program
that produces the desired behavior. However, automatic generation of motion policies is a difficult task mainly due to the large and continuous states spaces that the problem environment represents.

Specifically, this work implements the Connectionist Q-Learning Framework (CQLF) an open source Java library for developing learning systems. CQLF is based on a backpropagation neural network that is trained using the CQL algorithm by Sutton (Sutton, 1987). This framework uses a neural network instead of the Q-Table of classical RL techniques like Q-Learning. The neural network allows state generalization in order to avoid the continuous large state space problem. Generalization provides the agent the ability to learn without the necessity of exploring all the possible states of the environment (i.e. generalization = learn from limited experience by grouping together similar states).

This research work aims to implement and compare two empirical scenarios for learning motion policies: single agent approach and multiagent approach. By this is intended to prove that complex environments can be modeled as multiagent learning systems by a group of simple interacting agents with single state spaces (partial observations of states) as well as single action spaces (individual action selections). This work also aims to describe a framework for addressing problems of generating motion policies using single and multiagent approaches establishing the foundation for future work to address problems involving more complex policies as well as problems involving a larger number of interacting agents.

3 Related Work

One reason that RL is popular is that it serves as a theoretical tool for studying how agents learn to act. However, RL is very popular in practice, it has also been used by a number of researchers as a practical computational tool for constructing autonomous systems that improve themselves with experience. These applications have ranged from robotics, to industrial manufacturing, to combinatorial search problems.

Starting in 2008, with the launch of RobotStadium and the appointment of the robot Nao as the new official RoboCup SPL robot, many RoboCup teams have been working with Nao robots in either simulation (RobotStadium and RoboCup 3D simulation league) or real robots (RoboCup SPL). Currently there are few published results concerning reinforcement learning applied to robot control tasks in the robotic soccer field.

In (Saggar et al., 2007) is presented an implementation of the policy gradient machine learning algorithm that searches for a parameterized walk policy while optimizing for both speed and stability. Their main objective was to prevent unsteady camera motions which degrade the robot’s visual capabilities by achieving an stable and fast walking policy for the Sony Aibo (4-legged) robot, which was the official robot for the SPL until 2007. In their experimental results they achieved a walking policy reasonably fast and considerably more stable compared to their previous policies. They checked that the achieved stability significantly improved the robot’s visual object recognition. Unlike our research work, where the objective is to learn a motion policy (i.e. a mapping between states and actions), their objective is to learn the parameters of a “walking gait”. In this case, the policy is the set of parameters of the walking wait and the policy gradient algorithm looks for the parameters that show better speed and stability in the walking gait. The advantage of our solution method is that we don’t need any gait to produce motion policies.

In (Ogino et al., 2006) research work, authors show that a neural network can be used for learning a sensorimotor mapping between optic flow information and the parameter of motion primitives. This is done for the task of face-to-face pass between humanoid robots, decomposing it in several modules. Then, the optic flow information is correlated with the parameters of each module. Although we are also using neural networks in our research work, we use a more standard RL approach for learning motion policies through the experience of the robot in its environment. In order to achieve this, we propose the use the CQLF that uses a backpropagation
neural network as a method of generalization.

(Hester et al., 2010) presented a RL algorithm with Decision Trees called RL-TD that uses decision trees to learn the model by generalizing the relative effect of actions across states. The objective of their proposed algorithm is to be sample efficient (i.e., learn from very few real-world trials). Unlike our model-free approach, they used a model-based approach in order to learn from very few real-world trials. Model-based approach requires learning of an accurate explicit model and then use it to find the optimal actions via DP methods. In order to build and accurate model from limited experience, they used decision trees instead of neural networks as a generalization mechanism. Their experiments are based on a on physical Nao humanoid robot scoring goals in a penalty kick scenario. Their experimental work presents a comparison between classical model-free RL techniques against the RL-TD algorithm and demonstrate that RL-DT is a good choice for learning tasks on real robot. Their proposed approach allow them to learn a reasonable policy quickly without lot of exploration which can be very expensive on a real robot. In our approach we don’t need to worry about building an accurate model, but the agent learns by simple interaction with the environment.

In (Cherubini et al., 2009) is presented an extension of the Policy Gradient algorithm for learning humanoid walking gaits when there are few experiments available. Authors compare two of Policy Gradient algorithms focusing in curvilinear trajectories. Our research work is related to this one, but we do not focus in the “walking gaits” task. Instead, we focus in the learning of a motion for “kicking the ball” task comparing the single agent and the multiagent learning approaches.

The same authors presented another interesting research work in (Cherubini et al., 2010) where they present an application of another extension of the Policy Gradient algorithm for the “soccer attacking” task using quadruped robots now, instead of humanoid robots. They compare again their proposed algorithm against the classic Policy Gradient algorithm showing good results for this task. This is an interesting and related work to our own research work. However, as we have said before, we are proposing the use of the CQLF for the “kicking the ball” task and comparing the single agent and multiagent approaches.

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

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

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

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

(Martin & de Lope, 2007) presents a distributed approach to RL in multi-link robot control tasks. This approach avoids the combinatorial explosion when multiple states variables and multiple actuators are needed to optimally control a complex agent in a dynamical environment. The experimental results clearly show that it is not necessary that each individual agent perceives the complete state space in order to learn a good global policy but only a reduced state space directly related to its own environmental experience. This work uses another TD algorithm called SARSA (Sutton & Barto, 1998) as the basis for for the RL algorithm.

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

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

In the first phase, the robot is being controlled by a supplied control policy. This can either be actual control code, or a human directly controlling the robot. During this learning phase, the RL system is passively watching the states, actions and rewards that the supplied policy is generating. It uses these rewards to bootstrap information into its value-function approximation. The key element of this phase is that the supplied control policy exposes the RL system to the “interesting” parts of the state space (parts where the reward is non-zero).

In second phase, the learned policy is in control of the robot, as it would be in a standard RL implementation. By splitting the learning into 2 phases is ensured that, once the second learning phase starts, the robot will be capable of finding reward giving states. This sort of learning by demonstration has become quite popular recently.

4 Testbed: RobotStadium and Webots simulator

RobotStadium (Robotstadium, 2011) is a realistic robotic soccer simulation based on the Nao robot and the Webots simulator. RobotStadium simulation uses the rules of RoboCup Standard Platform League (SPL). Since 2008, RobotStadium has been a passionate on-line competition open to everyone and free of charge. RobotStadium warmly welcome RoboCup teams and robotics students from all around the world and is one of the main testbeds for the the RoboCup initiative in terms of AI techniques like reasoning, planning, learning, coordination, communication and opponent modeling. Actually RobotStadium is a very accurate simulation of the RoboCup SPL.
Participating teams just need to develop a controller and upload it to the competition server. Then a round will be executed and results can be viewed on the RobotStadium website, where also videos for all games are available (RobotStadium, 2011). Therefore developers can test their algorithms almost immediately and retrieve feedback to improve them and test again in the next round.

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

The Webots mobile robotics simulation software developed by (Cyberbotics, 2011), provides a rapid prototyping environment for modeling, programming and simulating mobile robots. With Webots it is possible to design complex robotic setups, with one or several robots, in a shared environment. There are many simulated sensors and actuators are available to equip each robot. Moreover, the robot behavior can be tested in physically realistic worlds.

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

There are two versions of the Webots simulator, the Webots Pro for researchers and the Webots Edu for educational proposes. However, the Demo version of Webots allow participating teams to work with the RobotStadium simulation. Figure 1 shows the Demo version of the Webots simulator running the RobotStadium contest environment.
Table 1: Example of a motion table. First column corresponds to simulation time in milliseconds and the remaining columns correspond to servo positions in radians.

<table>
<thead>
<tr>
<th>#Webots_Motion</th>
<th>V1.0</th>
<th>RShoulderPitch</th>
<th>LShoulderPitch</th>
<th>RShoulderRoll</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:200</td>
<td>Pose1</td>
<td>1.57</td>
<td>1.57</td>
<td>0</td>
</tr>
<tr>
<td>00:00:400</td>
<td>Pose2</td>
<td>1.37</td>
<td>1.37</td>
<td>0.1</td>
</tr>
<tr>
<td>00:00:600</td>
<td>Pose3</td>
<td>1.27</td>
<td>1.27</td>
<td>0.2</td>
</tr>
<tr>
<td>00:00:800</td>
<td>Pose4</td>
<td>1.07</td>
<td>1.07</td>
<td>0.3</td>
</tr>
</tbody>
</table>

In Webots distribution, the RobotStadium contest environment is provided in the file “robotstadium.wbt”. This is a text file which contains the description of every object within the environment (soccer field, walls, goals, robots, ball, lights and supervisor). The RobotStadium environment file also contains the controller information for robots and supervisor. The environment file can be edited with a text editor or with the “Scene tree” editor of Webots.

In RobotStadium environment, simulated robots correspond to the Nao robot, an autonomous medium-sized humanoid robot developed by the French company Aldebaran Robotics (Aldebaran-Robotics, 2011). Nao robot was designed for entertainment purposes, is able to interact with its owner with evolving behaviors and functionalities only limited by the imagination. RobotStadium uses a Webots model of the Aldebaran “NaoV3 RoboCup Edition” to simulate the Nao robot. This model is called “NaoV3R”, where the R means for RoboCup. In Webots the model is represented by a proto file called “NaoV3R.proto”. The robot model contains 22 servos (Head (2) + Shoulder (4) + Elbow (4) + Hip (6) + Knee (2) + Ankle (4)), a camera web, an accelerometer, a gyro, 4 ultrasounds and 4 Force Sensitive Resistors (FSRs) in each foot. It also contains radio emitter and receiver and many leds (Eyes, chest, ears, foot).

In the current Webots version, a robot controller can be developed with one of the following programming languages: C, Java, URBI, Python and Matlab. The controller can access the devices of the simulated Nao robot through a special library provided by the Webots simulator.

Every controller must be developed taking into account the basic “Time Step” of the simulation. The basic “Time Step” specified in the RobotStadium environment corresponds to the physics integration step. This is a virtual time duration that indicates how often the forces are recomputed and applied to the simulate rigid bodies. The computation of a time step is an atomic operation, it cannot be interrupted. Thus a sensor measurement and a motor force actuation can only happen between two basic time steps. By default, in RobotStadium, time step is defined as 40 ms, which means that controllers can read sensors and actuate motors at maximal virtual frequency of 25 Hz.

Design of motion behaviors is one of the most challenging problems in the development of a robot controller. A motion is a text file in comma separated value format which specify motion sequences that usually involve several servo motors playing simultaneously. Every participating team must design motion behaviors for basic abilities like walking, turning and standing up. These motion files are accessed by the robot controller and can be played, stopped, resumed and reversed. A motion file consists of a table in which there are one column for every motor and other for the simulation time. Every row specifies the servo positions (in radians) for the given time. The rows are sorted by the time column in ascending order. Time must be specified in multiples of basic “Time Step” (40 ms). Table 1 shows a simple motion example in which three servos are moved during 800 ms taking four different poses.
5 Solution method: Reinforcement learning plus generalization

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

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

5.1 Reinforcement learning

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

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

(Sutton & Barto, 1998) describe four main elements of a RL system:

**Policy** Defines the way in which a learning agent behaves at a given time. A policy is a mapping from perceived states of the environment to actions to be taken.

**Reward Function** Defines the goal in a RL problem. It maps each perceived state (or state-action pair) of the environment to a single number, a reward, indicating the desirability of that state.

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

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

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

5.2 Q-Learning

According to (Sutton & Barto, 1998), there are three fundamental classes of methods for solving the RL problem: Dynamic programming, Monte-Carlo methods, and Temporal-Difference learning.
Dynamic Programming (DP) methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo (MC) methods do not require a model, but are not suited for step-by-step incremental computation. Finally, Temporal Difference (TD) methods require no model and are fully incremental, but are more complex to analyze.

TD learning is a combination of MC and DP ideas. Like MC, TD methods can learn directly from raw experience without a model of the dynamics of the environment. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (bootstrapping).

Temporal Difference (TD) Learning methods are used to estimate value functions. TD methods learn their estimates in part on the basis of other estimates. The state values are updated every training step. Expressed formally:

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

(1)

Where \( s_t \) is the state in time \( t \), \( V(s_t) \) is the value of the state \( s \), \( s_{t+1} \) is the state in \( t+1 \) and \( r_{t+1} \) is the reward received in \( t+1 \). The parameters used in the update process are:

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

- \( \gamma \) the discount factor, also set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards.

There are 2 types of TD methods: On-Policy TD methods that learn the value of the policy that is used to make decisions and Off-Policy methods that learn different policies for behavior and estimation. In Off-Policy methods value functions are updated using results from executing actions determined by some policy. Two common policies are used for executing actions:

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

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

Q-Learning is an Off-Policy algorithm for TD learning. It can be proven that given sufficient training using any soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy (Sutton & Barto, 1998).

The procedural approach for Q-Learning is as follows:

**Algorithm 1: Q-Learning**

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

Formally the update process is described as follows:
\[ Q(s,a) = Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)] \]  

(2)

5.3 Generalization and function approximation

Estimates of value functions are represented as a table with one entry for each state \((s)\) or for each state-action pair \((s,a)\). This is a simple case, but it is limited to problems with small numbers of states and actions. For large tables, when the number of combinations between states and actions is huge, the main problem is the time needed to fill the table accurately, and probably the memory needed to store it won’t be enough (Sutton & Barto, 1998).

One of the simplest methods of dealing with large state spaces is discretization. This method splits a state space into small size areas, each being an input of the Q-table. The success directly depends on how well this splitting represents the Q-function. To achieve a greater accuracy, state space must be split into smaller areas resulting in a Q-table of a larger size. On the other hand, splitting into larger areas can result in the impossibility of reaching the optimal control policy (Kuzmin, 2002).

Several methods exist that help to speed up the process of learning when Q-tables of a large size are used. The objective is to produce a good approximation of a large state space based on a generalization over a limited subset of the state space. Generalization from examples has already been extensively studied in supervised learning field. The kind of generalization required is often called function approximation because it takes examples from a desired function (a value function) and attempts to generalize from them to construct an approximation of the function (Kuzmin, 2002).

There is a broad range of existing methods for supervised learning function approximation like gradient-descent, memory-based and decision-tree methods between many others. The range of possible methods is too large, and anyway too little is known. We can combine a wide range of RL methods with any function approximator by using the updates to an estimated value function of the RL method as training examples for the approximator (Sutton & Barto, 1998).

Function approximation methods based on gradient principles have been widely studied and used together with RL methods. These methods are promising and simple, actually they allow a natural function approximation extension to RL methods since there is an interesting interaction between function approximation, bootstrapping, the on-policy/off-policy distinction and even the eligibility traces (Sutton & Barto, 1998).

Linear gradient-descent methods like radial basis functions, tile coding, and Kanerva coding have showed that they work well in practice when provided with appropriate features. However, linear methods are limited to small state spaces since the number of base functions for qualitative approximation grows exponentially as the dimension of the input vector goes up (Sutton & Barto, 1998).

An ideal goal of function approximation is to find a global optimum. Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. These function approximators may seek to converge instead to a local optimum which typically is the best possible for them. For many cases of interest in RL, convergence to an optimum, or even true convergence, does not occur (Sutton & Barto, 1998).

It is well-known that multilayer perceptron is a good non-linear gradient-descent function approximator. The Kolmogorov neural network mapping theorem (Hubbard & Ilyashenko, 2003) states that feed-forward neural networks with three layers (input, layer, and output layers) can accurately represent any continuous function. The multilayer artificial neural networks using the error backpropagation algorithm is a function approximation that have been used widely together with RL.
5.4 Connectionist Q-Learning

When connectionist approach is employed in the Q-learning algorithm, the tabular representation of Q-function is replaced by a neural network. States are forwarded to the inputs of the network whereas estimates of Q-values serve as output data. The Connectionist Q-Learning (CQL) method consists in applying a separate neural network for each action as shown in figure 2.

During each iteration of the algorithm, the current state of the system is forwarded to the inputs of each neural network, but the weights are only updated for the network whose action was selected. Update process is performed by a standard back propagation algorithm calculating the output gradient with respect to the network weight correction error. Formally, the error calculation using CQL is:

$$w_t = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Where $w_t$ is the weight error and $r_t$ is the reward at time $t$.

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

**Algorithm 2: Connectionist Q-Learning**

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

For Connectionist Q-Learning algorithm it is necessary to store neural network weights, eligibility traces, the last value of Q-function and reward. Recall that output gradients are stored within the eligibility traces.

5.5 Connectionist Q-Learning Framework

The Free Connectionist Q-learning Framework (CQLF) (Kapusta, 2011) is an Open Source Java library for developing simple or complicated learning systems. It can be used anywhere, where an action can be chosen depending
on the environment state, and where executing the action can be rewarded or punished. The framework is small, easy-to-use and speeds up the development of learning agents.

The basic idea of the framework is based on a single neural network, which is trained using the CQL algorithm by Sutton (Sutton, 1987). Figure 3 shows the architecture of the framework. Sensor inputs are collected by the Perception module. One of the inputs is the reward given to the agent for actions performed in previous steps. This information is passed to the Brain module, which learns, by the reward value, whether what the agent did before was good or bad. The output of the network is the Q-function, so the number of output neurons equal to the number of actions.

6 Learning motion policies

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

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

6.1 The problem of generating motion policies

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

This research work implements a RL approach for automatic generation of motion policies. Through the experience of the agent in its environment, it will be able to generate the required motion policies. However the problem of enormous state and action spaces is still present for the RL approach.

The first approach trying to deal with big state spaces is discretization. For example, for a single servo
motor, its state space between $-\pi$ and $\pi$ can be split every 0.087 radians (approximately 5 degrees). In this case, the Q-table will have 72 rows, one for each state.

For the case of one servo motor, discretization can be applied successfully, but for motion policies involving several servo motors, this is not a good approach. Consider the task of learning to kick a ball. This scenario involves at least moving the servos from one leg. In the Nao robot each leg have 5 servo motors and supposing each one has a range between $-.7854$ and $.7854$ radians (approximately 130 degrees), splitting every 5 degrees result in 26 areas for each servo motor. Thus combining all servo motors in the Q-table result in $26 \times 26 \times 26 \times 26 = 11,881,376$ rows. This is an enormous quantity of possible state combinations. The state space grows exponentially increasing the number of servo motors. Therefore, discretization is not useful in learning of motion policies.

Generalization is the only way to infer from previously experienced states to ones that have never been seen, a very common situation on many interesting RL tasks.

### 6.2 Empirical scenarios

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

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

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

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

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

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

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

6.3 Learning to kick the ball

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

“Kicking the ball” experimental scenario consists of a RobotStadium environment in which there is only one robot on the soccer field. The right foot of the robot is aligned with the ball and separated 15 cm of this. The ball is at the center of the field. Also the environment contains a supervisor code encouraged to place the player and ball at the beginning of each training epoch and sends a message to the agent when the ball is touched. Figure 4 shows the RobotStadium environment for this experiment.

The learning task consists in kicking the ball with the right leg. For simplicity, and with the aim to reduce the state space, in this experiment robot do not use arms for kicking. Also the robot must kick the ball moving only three motors from right leg (RHipPitch, RKneePitch, RAnkleRoll) and two motors from left leg (LKneePitch, LAnkleRoll). The less servo motors are involved, the learning task will be less complex. However it must be considered the feasibility of doing the task properly by reducing the number of actuators.

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

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

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

Perception module use unipolar (output range from 0 to 1) or bipolar (output range from -1 to 1) sigmoid
transfer functions to pass input signals to the Brain. Thus, input signals must be scaled in order to accurately
represent the state for the neural network. For servo motors positions is used a scaling factor of 3 giving the
output range between −0.91 and 0.91 and for simulation time, is used the following formula:

\[ t/25 − 4 \] (4)

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

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

Also is important to mention that the CQLF initializes the network weights with small random values
at the beginning of each learning trial. This is a very common network weight initialization method due to its
simplicity and acceptable performance. There are many others network weights initialization methods. We left
the extension of the CQLF to support other initialization methods as a future work (section 3). We think that by
a good network weight initialization (i.e. setting the initial weights close to a good solution or at least not so far)
the training will be faster and the possibility of obtaining convergence will increase.

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

6.3.1 Single agent approach

For the single agent approach, the Perception module pass six input signals to the Brain: simulation time and
current position of the five servo motors. There are 11 Action modules, one for each possible move (backwards
and forwards) for the five servo motors and a simple “dummy” function that does nothing. Figure 5 shows the
overall framework architecture for the single agent scenario solving the learning motion problem experiments.

Table 2 shows the CQLF configuration parameters for the single agent approach for “kicking the ball”
experiment. Recall that in single agent approach, the policy is representing a single agent controlling all the
servomotors at the same time. However due to the high combinatorial number of possible actions, this approach
is adjusted to select only one movement of one servo motor at each time. By using this scenario, the objective
of the learning agent is to learn a sequence of single servo motor actions in order to produce a motion behavior. Single agent empirical scenario is a very simple approach as there is only a centralized controller taking care about all actuators. However it has a great disadvantage, it can only move one motor per simulation step (i.e. not allow action execution in parallel). In addition, given the centralized control, the policy can become very complex by increasing the number of servomotors needed for performing a task.

6.3.2 Multiagent approach

For the multiagent approach, the perception module for each learner is the same that for the single agent approach. Each Perception module receives the simulation time and position of all servo motors as inputs. Finally every learner has three possible actions: move forwards, move backwards and stay in current position. Figure 6 shows the overall framework architecture for the multiagent scenario solving the learning motion problem for the kicking the ball experiment.

When using multiagent approach, CQLF configuration for individual learning agents is the same as for the single agent approach (table 2). In multiagent approach, multiple agents learn to cooperate with each other by knowing actions generated by other agents. This is achieved by receiving the current state of all other agents in the own state representation. In this way, the agent (servo motor) learns how to act given the own state (servo position) together with the states of all other agents. Despite all the learning agents are fed by the same reward function, they manage to converge towards the common global goal. This verifies that the CQL algorithm is able to learn in noisy environments as the learning agent is able to converge towards the target even as their perceptions

**Table 2:** CQLF configuration for the single agent approach for “kicking the ball” experiment.

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

**Figure 5:** Single agent approach for learning motion problem.
may be distorted by other agents or by the nature of environment.

6.3.3 Results and discussion

For both empirical scenarios in the “kicking the ball” experiment there were run many trials. For single agent scenario, each trial consisted of 3000 training epochs unlike the multiagent agent scenario in which were 1500 epochs. This is because in multiagent approach convergence was achieved more quickly as as will be shown later. During trials, average reward is computed and saved in a text file. The average reward is the total amount of reward received by the agent during training epochs divided by the number of training epochs. This measure is very useful in determining how a learning agent is behaving in its environment. Average reward plots consist of a learning curve that clearly show if the learning process converges to a solution or if the agent has not learned the desired behavior.

Figure 7 shows the learning curve of the single agent approach for solving the “kicking the ball” experiment. The single agent approach had a regular performance achieving an average reward of 0.26. This approach converges to that value of average reward at which the agent learns to kick the ball, however, the kick is very weak so that the received reward is small. Although the single agent approach learns to kick the ball weakly, it can be concluded that this approach can simulate the behavior of the motors in parallel despite the serial execution is carried out.

Also, figure 7 shows the learning curve of the multiagent approach for solving the “kicking the ball” experiment. In the graph can be observed the good performance of multiagent approach achieving a very rapid convergence since only after 500 training epochs the average reward were close to 0.8 and reaching 0.86 after 1500 epochs. In this case the agent learned to kick the ball with considerable force as the final average reward is quite close to optimal. Thus it can be concluded that the multiagent approach had a very good performance in the experiment of “kicking the ball”, achieving a fast convergence and obtaining an average reward near to optimal.

Finally, to conclude with the “Kicking the Ball” experiments, there were compared both empirical scenarios: single agent and multiagent. When comparing both learning curves in figure 7, it can be noticed that multiagent approach clearly outperforms the single agent approach. Multiagent approach had a quicker convergence and also a higher final average reward. During this experimental scenario, multiagent approach showed a good performance in combination with the CQLF.

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

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

Framework used in this research work consists of a TD learning algorithm called Q-Learning plus a generalization mechanism (function approximator) with neural networks trained by backpropagation. TD and gradient-descent algorithms generally fails to learn the optimal policy. An optimal policy is better than or equal to all other policies. Almost all interesting RL problems which need generalization methods are very complex and convergence to an optimum policy, or even true convergence, does not occur. Optimal policies can be generated only with extreme computational cost. Cases in which optimal solutions cannot be found but must be approximated in some way. Moreover, some of these interesting problems are not so sensitive to non optimal solutions (i.e. local optimum solutions are acceptable).

For the problem of generating motion policies, given the reward function, we were looking for a motion for kicking hard and in straight line in a short period of time. We achieved a kicking policy near to optimal with a max average reward of 0.86 from an optimal of 1. In this case, the Connectionist Q-Learning method converged to a local optimum solution which actually is a really good solution for our proposes of generating a kicking motion from scratch.

Surely the most difficult activity in order to learn motion policies with CQLF is the configuration of the
parameters of both neural network and the CQL algorithm. There isn’t an established way to configure parameters for Connectionist Q-Learning algorithm. However, given the problem features some insights can be obtained. For this case, for the problem of generating motion policies, we knew that the environment is noisy, the reward is obtained until the absorbing states (delayed) and earlier states are important to achieve the goal and finally maintain a permanent exploration. After experimental phase, we found the following parameter configuration with good results and fastest convergence: \( \varepsilon = 0.1 \), \( \alpha = 0.2 \), \( \gamma = 0.99 \) and \( \lambda = 0.5 \).

Finally, another important configuration concerning the neural network is the amount of hidden layers and their number of neurons. Configure the amount of neurons in the hidden layer is really tricky and requires lot of testing (trial and error) and experience in neural networks. This parameter is crucial in determine the capability of the network to learn the policy. For our problem we found good results using only one hidden layer with twelve neurons.

### 7 Future work

Results of this work are part of an ongoing research in which the framework is also implemented to generate decision-making policies with the objective of generate single and multiagent strategies. The decision-making problem involves the selection of high-level actions in contrast with the motion problem where the selected actions are low level. Actually, in the decision-making problem the selected actions correspond to motion behaviors. For its part, the state consist of the perceptions of the environment that the agent gets through its sensors (vision, accelerometer, gyro, ultrasound, servo positions, and force sensitive resistors). In this way, the proposed framework can learn decision-making policies for strategies like defense, goal keeper, dribble and so on.

There are many possibilities for future work, principally addressing problems involving more complex policies and a greater number of interacting agents. There is a great variety of approaches to implement multiagent learning like cooperative, competitive, team learning and concurrent learning. However is important to assess to what extent the proposed solution method is able to tackle more complex problems without the need to use some other multiagent coordination techniques like negotiation, voting, contract net, etc.

For generating motion policies, future work can address more complex motions like walking and getting up policies which require more precise control and many servos playing at the same time. The framework can be extended to support a continuous action space instead a discrete number of actions. In this case action space could represent the positions of servos, by taking the outputs of the neural network as the position of servos. In this way the number of output neurons must be equal to the number of servos needed for playing the motion.

Also the CQLF can be extended to support other network weight initialization methods in addition to the currently supported random method. It is well known that the weight initialization influences the speed and probability of convergence. The weight initialization is a very important issue when dealing with neural networks and therefore many initialization methods have been proposed. The objective of adding a network weight initialization method to the CQLF is to set the initial weights close to a good solution in order to have a faster training and increase the possibility of obtaining convergence.

Moreover this framework can be implemented and tested on the real Nao. Thus it is possible to generate policies using physical robots or generate policies in simulation for further implementation in real robots. Finally the framework also can be implemented in different simulated robotic soccer leagues like RoboCup 2D and 3D simulated leagues and of course can be adapted to work in other simulated and real task environments.
8 Conclusions

This work proposed framework for addressing problems of generating motion and decision-making policies using single and multiagent approaches in the “RobotStadium” environment. This research work is the first attempt to solve the problem of generating motion and decision-making policies using a reinforcement learning technique in the “RobotStadium” environment.

The proposed framework uses the Connectionist Q-Learning algorithm which combines a reinforcement learning technique (Q-Learning) with a generalization method (backpropagation neural network). However, the key of the contribution are the proposed approaches for implementing the algorithm. This thesis proposed two different approaches: single agent and multiagent.

In this work it was showed that it is possible to automatically generate motion policies in continuous and dynamic environments through reinforcement learning as long as there is a method of generalization. It was concluded that multiagent approach together with the CQLF (RL + Generalization) is a good approach to tackle the problem of generating motion policies. The main contribution is the multiagent approach in which every servo motor learns its own motion policy. In this way, individual policies are simpler than the global policy and by learning them it is possible to achieve the global desired motion. Also the multiagent approach is very simple given that individual agents receive the same global rewards and state representation, focusing only on individual actions regarding its own servo motor.

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

References


