Cooperative Behaviors for RoboCup 3D through Deep Reinforcement Learning

Laureando
Jim Martin Catacora Ocana

Relatore
Daniele Nardi

SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA
UNIVERSITÀ DI ROMA

Cooperative Behaviors for RoboCup 3D through Deep Reinforcement Learning

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti"
Corso di laurea in Artificial Intelligence and Robotics

Jim Martin Catacora Ocana
Matricola 1754293

Relatore
Daniele Nardi

A.A. 2017-2018

Contents

Chapter 1: Introduction 5

1.1. Motivation 5

1.2. Overview 6

1.3. Outline 7

Chapter 2: Related Work 9

2.1. Review of Independent Learners 9

2.2. Review of Coordinated Learners 10

2.3. Review of Joint-Action Learners 12

2.4. Reinforcement Learning in the Soccer Domain 13

Chapter 3: (Deep) Reinforcement Learning 16

3.1. Markov Decision Processes (MDP) 16

3.2. Reinforcement Learning 17

3.2.1. Definitions 17

3.2.2. Temporal-Difference (TD) 18

3.2.3. Policy Gradient (PG) 20

3.2.4. Actor-Critic (AC) 21

3.2.5. Action Selection 22

3.3. Deep Reinforcement Learning 22

3.3.1. (Deep) Neural Networks 22

3.3.2. Stochastic Gradient Descent (SGD) Optimization 25

3.3.3. Deep Q-Network (DQN) 27

3.3.4. Deep Deterministic Policy Gradient (DDPG) 28

3.3.5. Further Improvements 29

3.4. Multi-agent Reinforcement Learning 30

3.4.1. Independent Learners (IL) 31

3.4.2. Joint-Action Learners (JAL) 32

Chapter 4: RoboCup Standard Platform League (SPL) Simulation 34

4.1. Overview 34

4.2. Agent's Inputs 37

4.2.1. Raw Inputs 37

4.2.2. High-level Inputs 38
4.3. Agent's Outputs 38
4.3.1. Raw Outputs 39
4.3.2. Macro Actions 39
4.4. Behavior Control 41
4.5. Neural Controller 42
4.6. Reinforcement Learning Modules 43
4.7. Agent's behavior 44
4.8. Learning with the Simulator 45
Chapter 5: Technical Approach 47
5.1. Task Specification: Learning Free Kicks 47
5.2. Experimental Setup 48
5.2.1. World Settings 48
5.2.2. MDP Formulation for IL 50
5.2.3. Markov Game Formulation for JAL 52
5.2.4. Deep RL Settings 53
Chapter 6: Results 58
6.1. Independent Learners 58
6.2. Joint-Action Learners 60
6.3. Discussion 61
Chapter 7: Conclusions and Future Work 63
7.1. Conclusions 63
7.2. Future Work 64
References 67

Chapter 1
Introduction

1.1. Motivation

The development of the first digital programmable computers in the 1940s marks the beginning of the modern era of Artificial Intelligence (AI). This was the first instant in the history of humankind in which the idea of machines surpassing the capabilities of the human brain, not only in terms of speed but also of sheer intelligence, became truly tangible. Fast-forward to the present, artificially created intelligent systems and robots are everywhere, nowadays. From virtual assistants such as Siri, Alexa or Cortana that interact effortlessly with humans, to CIMON, a balloon-like robot that helps astronauts inside the International Space Station. Intelligent software is being used with success in stock market trading, legal practices, interpretation of medical images, automatic image descriptions, and has also defeated human experts in challenging games, such as Jeopardy (IBM Watson) and Go (DeepMind AlphaZero). Moreover, intelligent robots such as Knightscope are patrolling parking lots and buildings, while Uber and Tesla are producing and developing semi-autonomous and self-driving cars and trucks.

Certainly, this selected handful of real-world examples reveals the extraordinary advancement of AI. However, we are still far away from the wildest dreams portrayed in science fiction, as for instance, conversational agents able to grasp the ambiguity of human language, machines capable of producing thought-provoking art without relying on imitation, machines with intrinsic goals, autonomous robots roaming the streets, and ultimately, general artificial intelligence. Much of the recent groundbreaking results in AI come from the field of Machine Learning, specifically from the area of Deep Learning, which thanks to large amounts of annotated data and computational power has been able to solve several difficult tasks. Yet, Deep Learning on its own will be unlikely to generate the highly complex and abstract behaviors we ambition, as they cannot be mastered merely through labeled examples. Thus, researchers are turning their attention towards non-supervised learning techniques that do not require experts, such as (Deep) Reinforcement Learning (RL), Evolutionary Algorithms (EA), Generative Adversarial Networks (GANs) and many others, which are expected to lead the next revolution in AI.

Following this shift in paradigms, this thesis is focused on Reinforcement Learning, and particularly, on Deep Reinforcement Learning. RL is likewise an area of machine learning, in which learning arises from the direct interaction of an agent with its environment and where the agent seeks to discover through trial and error a policy that maximizes some reward. In turn, the sub-area of Deep RL was born quite recently. It incorporates much of the same principles as classical Reinforcement Learning, but whereas the latter uses finite-size tables to encode the resulting policies, the former approximates such policies with deep neural networks in order to generalize over very large or continuous observation spaces.

Research in Deep RL has become imperative since the last three years, as a consequence of the seminal and highly successful works of: Mnih et. al, 'Human-level control through deep reinforcement learning' (2015), which showed that a deep Q-Network trained in model-free RL can achieve an above human-level performance in several Atari 2600 games; and Silver et. al, 'Mastering the game of Go without human knowledge' (2017), which developed an agent capable of defeating world-champion programs of Go solely by using Deep RL in combination with and Monte Carlo Tree Search, while allowing the agent to play against itself during the learning phase. These works decidedly demonstrate that Deep RL can surpass human mental proficiency with ease and without relying in brute force, making it one of the most promising current techniques for achieving a more general intelligence.

1.2. Overview

Despite outstanding previous results, Deep RL still faces several challenges that need to be resolved before it can be acknowledged as a truly practical and universally applicable method. Some known challenges are: poor sample efficiency, transfer learning, catastrophic forgetting in neural networks, domains with sparse and delayed rewards, multi-agent domains, to name a few. Specifically, this work deals with this last problem, namely that of learning in domains having multiple agents, where two or more of them are simultaneously and constantly adapting to its environment and to the other agents.

Three main different approaches have been proposed for handling multi-agent domains. In independent learners (IL), every agent performs RL independently, but in the presence of other agents. IL has the drawback that each individual sees the environment as non-stationary; and hence, convergence and optimality guarantees of single-agent RL do not longer hold. Meanwhile, in joint-action learners (JAL), the learning problem is reformulated such that instead of maximizing the independent rewards received by each agent, we maximize at once the group's reward given the joint action executed synchronously by all learning agents. JAL presents a scalability issue, as the complexity of learning grows with the number of agents. Lastly, coordinated learners (CL) lies between IL and JAL. It follows the premise that in many instances of a global task, individual agents can make optimal decisions on their own or by interacting with a small subset of other agents. Thus, it learns to identify which states require coordination and depending on this prediction it applies alternatively IL or JAL. However, because CL actively limits the interactions between agents, optimality might no longer be achievable as in IL.

In theory, domains such as simulated soccer, where the number of learning agents never becomes extremely large and that do not involve real-world communication, are expected to be more fitting for JAL rather than for IL or CL, since scalability issues should not be a major concern and since JAL can potentially discover a more optimal coordinated team policy. Despite this, of the very few known studies that have attempted to learn soccer team behaviors with Deep RL, most of them have implemented only IL [1, 2], and one study that experimented with both reported discouraging results for JAL [3].

It must also be mentioned that most previous studies dealing with cooperation in soccer domains had been carried out in RoboCup 2D Simulation or analogous frameworks, where the agents are moving circles on a planar field with no realistic physical attributes. However, nowadays, frameworks such as RoboCup SPL Simulation or 3D Simulation, in which the players are virtual NAO robots whose sensors and actuators have been simulated faithfully, present more demanding yet tractable environments. Thus, it is important to start facing these new challenging domains, as they could lead to brand new interesting solutions in the area of multi-agent Reinforcement Learning.

Based on the above, the specific goals of this thesis are the following:

1) Implement deep Reinforcement Learning solutions, specifically IL and JAL approaches, for solving a cooperative task in the simulated soccer multi-agent domain.
2) Verify under new settings if observed discrepancies between theory and practice regarding IL and JAL performances still hold; and if so, analyze which causes might lead to this outcome.
3) Validate Deep Reinforcement Learning approaches for multi-agent systems in the more demanding domain of RoboCup SPL Simulation.

1.3. Outline

The remainder of this thesis is organized as follows.

Chapter 2, provides a general background regarding classical and deep Reinforcement Learning methods, which are the focus of this work. It also procures the algorithmic formulation of their most widespread techniques. Furthermore, it describes current approaches for applying Reinforcement Learning in multi-agent domains.

Chapter 3, presents technical details of the virtual domain in which learning will take place, namely of the RoboCup SPL simulator. In addition, it exposes the modifications made by the author of this thesis to said domain in order to accommodate for the practical implementation of Deep Reinforcement Learning algorithms.

Chapter 4, summarizes similar studies found in the literature with respect to the application of Reinforcement Learning and Deep Reinforcement Learning in multi-agent domains. It covers results associated with general domains, as well as results specific to the soccer domain.

Chapter 5, initially presents the particular multi-agent cooperative task that will be employed for testing two Reinforcement Learning approaches, namely Independent Homogeneous Learners and Joint-Action Learners. Moreover, it reveals the formulation and implementation details leading to the final Deep Reinforcement Learning solutions.

Chapter 6, showcases and compares the performance results achieved by the tested approaches.

Chapter 7, discusses in depth the results obtained, while it provides conclusions and suggest possible directions for future work.

Chapter 2
Related Work

This Chapter, at first, provides a broad review of the several methods that has been proposed within the Reinforcement Learning literature for handling multi-agent domains, which have been organized into three approaches: Independent Learners (IL), Coordinated Learners (CL) and Joint-Action Learners (JAL). Subsequently, it delves into the many practical implementations of such techniques applied over the soccer domain.

2.1. Review of Independent Learners

Tan, 1993 [4], produced one of the earliest works that implemented IL. In this study, two virtual hunters, each interacting concurrently and independently with the environment, learned by means of standard Q-Learning to catch a virtual prey. The hunters shared the same policy and their experiences; hence, this was effectively a Homogeneous Learners implementation in which only one policy was maintained and improved at all times.

Lauer et. al, 2000 [5], proposed a new algorithm suited solely for cooperative tasks, called Distributed-Q, that keeps an individual Q-table for each agent, which is updated under the optimistic assumption that the other agents will act optimally. In this way, individuals are not penalized when they choose an optimal action, but the global joint action is suboptimal, due to poor decisions taken by their teammates. Distributed-Q is only valid for deterministic environments, where each state-action pair has a unique Q-value. Kapetanakis et. al, 2002 [6], employed single-agent Q-learning enhanced with FMQ-based (Frequency Maximum Q-value) action selection. FMQ is based on the Boltzmann strategy, such that the probability of choosing an action is associated with its average and maximum Q-values, the latter weighted by its frequency of occurrence. This strategy enables the agent to remember a high-reward action, which is likely an optimal joint action. FMQ shows good performance in some stochastic environments, but fails for highly noised scenarios. Matignon et. al, 2007 [7], developed Hysteretic-Q, which extends over Distributed-Q by considering two learning rates for Q-values of the same state-action pair. One activates after high rewards, and the other after low rewards. Hysteretic-Q is able to deal with some stochasticity in the environment.

Tampuu et. al, 2015 [8], provided a straightforward extension of IL to the deep learning framework. This work simultaneously and independently applies single-agent DQN for each of two learning agents which play either a cooperative or a competitive version of the game of Pong. Egorov, 2016 [9], also implemented DQN independently for every learning agent interacting with the world. Additionally, to combat non-stationarity, this work enforced an Alternating Learners procedure under which only one agent improves its policy at a time, while the remaining agents keep their policies fixed during that period of time. Omidshafiei et. al, 2017 [10], introduced Decentralized Hysteretic Deep Recurrent Q-Networks. Dec-HDRQNs integrate the robustness of Hysteretic-Q to non-stationarity and the suitability of DRQNs for handling to some extend partial observability. In addition, Concurrent Experience Replay Trajectories (CERTs) was proposed as a decentralized yet stable extension of experience replay that synchronizes sampling across independent and heterogeneous learning agents, such that all of them use examples from the exact same timesteps and episodes during each training step. This correlation merely requires that agents agree on the seed of their random number generators prior initiating learning. Palmer et. al, 2018 [11], extended lenient learning to deep RL as an alternative to hysteretic learning for stabilizing experience replay in Independent Learners. Lenient learners store temperature values that are associated with state-action pairs. Each time a state-action pair is visited, its associated temperature value is decayed, thereby decreasing the amount of leniency that the agent applies when performing a policy update for the state-action pair. The stored temperatures enable the agents to gradually transition from optimistic to average reward learners for frequently encountered pairs. Lowe et. al, 2017 [12], presented MADDPG, a deep learning actor-critic method that blurs the distinction between IL and JAL. Each agent has a joint critic and an independent actor. To learn a joint Q-value function specific for its agent, critics explicitly use during training observations and actions received from other agents as well as a global reward signal. On the other hand, actors have access only to local information, i.e. their own observations, as they learn local policies based on the joint Q-values provided by their critics. Foerster et. al, 2017 [13], proposed COMA, a new actor-critic method whose architecture is similar to MADDPG. Like MADDPG, each agent has a joint critic and an independent actor. Differently from MADDPG, COMA uses parameter sharing to speed learning, such that it trains only one critic and one actor. Moreover, in COMA, the joint critic computes an agent-specific advantage function that compares the estimated return for the current joint action to a counterfactual baseline that marginalizes out a single agent's action, while keeping the other agents' actions fixed.

2.2. Review of Coordinated Learners

Guestrin et. al, 2002 [14] and Guestrin et. al, 2002 [15], constitute the first works regarding coordinated learning. These studies proposed an agent-based factored approximation of a joint Q-value, where each agent keeps a partial Q-value. Each agent has full observability of the world state at all times. Furthermore, it receives information of the actions performed by other agents, only if they directly interact with it at any given state. Such interactions between agents are manually determined beforehand and expressed according to coordination graphs, with agents as nodes and beneficial interactions as edges. A Q-learning variant was formulated to discover coordinated policies, which updates the partial Q-value kept by each agent according to the observed global reward and by means of maximizing of the corresponding factored joint Q-value. This means that agents must communicate their current action-values at every update.

Kok et. al, 2004 [16], made two improvements over the previous method. First, based on predefined coordination graphs, each agent keeps two different types of Q-values, one for uncoordinated states and the other for coordinated states, and uses a modified Q-learning update rule for transitions between states of different types. Second, for coordinated states, a factored representation of the reward function is considered, such that the joint update rule of the system is fully decomposed at the level of each agent. Kok et. al, 2005 [17], developed an approach that automatically identifies which states need coordination between agents. Agents first learn optimal policies independently; then, as they execute their local policies they built statistics of the rewards received when interacting with other agents. If according to a t-test the mean reward gathered under interaction varies greatly with respect to the expected optimal uncoordinated reward of an agent, the corresponding state is marked as needing coordination. This approach shows good results at reducing detrimental interferences among agents, but not so much at exploiting synergies. Melo et. al, 2009 [18], also tackled the automatic identification of states that need coordination. Instead of relying on statistical tests, it considers a two-layered reinforcement learning structure that learns all by itself to differentiate between states. The action set of every agent is extended with an extra pseudo-action called "Coordinate". Thus, only when an agent selects this pseudo-action, it coordinates a joint-action with their neighbors, otherwise it selects an individual action independently. Both layers apply Q-learning keeping separate representations, the top layer operates at the local level of each agent while the bottom layer learns over the joint space. Lau et. al, 2012 [19], applied a two-level hierarchical reinforcement learning scheme, such that the top level automatically learns coordination rules by constraining the execution in a given state of certain joint actions for given pairs or groups of agents. The result is that the joint-action space where the optimal policy must be sought is effectively reduced. The bottom layer can then employ a joint-action learning approach over this reduced space to resolve the specific interaction. Yu et. al, 2014 [20], required that agents receive negative rewards when they experience conflicts or interferences, in order to learn "independent degrees" for each agent and state, which are indicators of how likely it is that coordination with other agents is needed. Independent degrees are then used in the same way as the epsilon parameter of a greedy exploration, in order to proactively drive the exploration of coordinated joint-actions. Ovalle Castañeda, 2016 [21], introduced Deep Loosely Coupled Q-Network (DLCQN), which is a deep Reinforcement Learning variant of previous work that relies on independent degrees for coordination.

2.3. Review of Joint-Action Learners

Joint-Action Learners can be further divided into Equilibrium Learners and Best-Response Learners. The former approach assumes that every agent in the environment eventually will act optimally, thus, it attempts to converge towards a game theoretic equilibrium within the multi-agent system, usually a Nash equilibrium. Whereas the latter aims at designing optimal agents given an environment inhabited by certain types of other agents, where optimality is defined simply in terms of maximum payoff and does not consider equilibrium as necessary or even relevant.

With respect to Equilibrium Learners, Littman, 1994 [22], proposed Minimax-Q algorithm for solving zero-sum two-player Markov Games. Minimax-Q is essentially identical to Q-learning, the only difference is that the max operator used by Q-learning in the update rule is replaced by a minimax operator that internalizes the actions of the opponent. Hu et. al, 1998 [23], developed Nash-Q as an extension of Minimax-Q algorithm to two-player general-sum Markov Games. Nash-Q requires that the learning agent maintains, besides its own Q-table, a separate joint-action Q-table for every opponent in the game; assuming also that actions and rewards of its opponents are directly observable. A quadratic programming problem is solved at each iteration in order to compute the Nash equilibrium for a given state, which replaces the minimax operator. Littman, 2001 [24], introduced Friend-or-Foe Q, which relies on knowing before-hand which agents in the environment are teammates and which are opponents with respect to the learner. This extra knowledge eliminates the need to keep multiple Q-tables, and additionally allows to replace the Nash operator with a max operator if another agent is a friend or with minimax operator if another agent is a foe. This algorithm reaches an optimal equilibrium only for two-player constant-sum Markov Games. Greenwald et. al, 2003 [25], conceived Correlated-Q learning that is similar to Nash-Q, but instead of seeking a Nash equilibrium, it settles for a correlated equilibrium, which is more general and easier to compute given some social convention. Particularly, the Nash operator is replaced by max, minimax or maximax operators, depending on an agreed equilibrium selection. To play according to a correlated policy, players must have access to a shared randomizer. Könönen, 2003 [26], proposed Asymmetric-Q learning. At each iteration, one agent (the leader) acts first, whereas to other agents (followers) act after observing the leader. The advantages of an asymmetric model are that the equilibrium point is unique for the leader in each state and a solution always exists in pure strategies.

On the other hand, the following studies aligned with the Best-Response Learners approach. Uther et. al, 1997 [27], builds a joint-action Q-table identical to Minimax-Q, while it also learns an explicit model of the opponent in the form of a stationary distribution over possible actions for every state. Learning is accomplished with the help of fictitious play. It was proven that this method outperforms Minimax-Q against some types of opponents. Suematsu, 2002 [28], implemented Extended Optimal Response (EXORL) for bimatrix games, which tries to learn an optimal response to an opponent with a fixed policy, but if the opponent adapts, it tries to reach a Nash equilibrium. EXORL requires that a learner can observe the other agent's actions and rewards, as it maintains joint-action Q-tables for all agents. Conitzer et. al, 2003 [29], introduced AWESOME, whose key idea is to adapt to other agents' strategies when they appear stationary, but otherwise retreat to a pre-computed Nash equilibrium strategy. It assumes that agents know in advance the payoff matrix of the environment and that they can directly observe the actions of other agents. Based on this, AWESOME determines the current distribution of actions of its opponents from a window of their last moves, and compares it against the equilibrium mixed strategy or a previously observed distribution. Weinberg et. al, 2004 [30], developed NSCP that finds a best-response strategy against non-stationary policies with a computational limit, after modeling said policies. This algorithm works within the framework of n-players general-sum stochastic games, and it requires direct knowledge of other agents' actions. Powers et. al, 2007 [31], provided two new algorithms, PCM(s) and PCM(A), that find best-response policies against stationary and memory-bounded adaptive opponents respectively, while guaranteeing a security value against any opponent. Chakraborty et. al, 2013 [32], formulated CMLeS which achieves three objectives: 1) it converges to a Nash equilibrium joint-policy in self-play; 2) it achieves close to the best response when interacting with a set of memory-bounded agents; and 3) it ensures an individual return very close to a security value when interacting with any other set of agents.

Gupta et. al, 2017 [33], extended three classes of single-agent deep Reinforcement Learning algorithms (based on policy gradient, temporal-difference, and actor-critic methods) to cooperative multi-agent domains while considering both, Independent Learners and Joint-Action Learners approaches. Several experiments were executed within three scenarios having tens of learning agents: 1) discrete pursuit-evasion, 2) continuous pursuit-evasion, 3) bipedal walkers moving a package. Results indicate that IL consistently outperformed JAL in all tested scenarios.

2.4. Reinforcement Learning in the Soccer Domain

The game of soccer and its variants, such as Keepaway, Half-Field Offense and Offensive Free Kicks, have been a widely used test-bed for multi-agent planning and learning algorithms. In soccer, there are two teams with the same number of agents at start, and the objective is to score more goals than the opponent during a match. In Keepaway, there are two teams, keepers and takers. The two teams do not need to have the same number of agents. The goal of the keepers is to maintain possession of the ball within a limited region for as long as possible, while the goal of the takers is to gain possession of the ball. In

Half-Field Offense, there are two teams, an offensive and a defending team. The game is played on one half of the regular soccer field. The objective of the offensive team is to score a goal, whereas, the objective of the defending team is to avoid goals. In Offensive Free Kicks, there is also an offensive and a defending team. The former is granted with a free-kick somewhere in the opponent's field and its objective is to score a goal before a time limit. The following studies have implemented learning solutions for the soccer domain or its variants.

Park et. al, 2001 [34], improved the performance of a team playing in NaroSot category, by training two physical robots that are close to the ball (coupled agents) to coordinate so that one of them kicks the ball towards the opponent's goal area. This work uses modular Q-learning. Each agent applies Q-learning independently. Additionally, there is a mediator module that is triggered only when two agents are identified to be coupled. Coupled agents communicate their Q-values to the mediator, and the later selects which agent will kick the ball based on the highest Q-value. Köse et. al, 2004 [35], uses Q(λ)-learning to select the roles of each agent in a soccer team. Teams are composed of 4 agents, and anyone of them can assume one of 4 roles: primary attacker, secondary attacker, primary defender, secondary defender. The global state vector internalizes environmental variables as well as the costs that would incur each agent if it takes a given role. Experiments were conducted within Teambots, a 2D soccer simulation. Stone et. al, 2005 [36], implemented a solution for Keepaway Soccer in RoboCup 2D simulation. It followed the Independent Learners approach within a SMDP options framework, i.e. keepers can choose only between predefined macro-actions, such as: pass ball towards keeper or go to ball. Each keeper learns independently through SARSA(λ) solely when it has possession of the ball. Keepers without possession of the ball follow a fixed policy; hence, non-stationarity is avoided. Celiberto et. al, 2008 [37], used HAQL (Heuristically Accelerated Q-Learning) for training a team of 4 agents in RoboCup 2D simulation to play against a hand-coded opponent team. Each agent learns independently based on Q-learning over macro-actions while also considering a heuristic term. Every state-action pair is manually assigned a different heuristic value, which is then combined with the corresponding Q-value only for the purpose of driving action selection.

Ma et. al, 2009 [38], trained a team of agents to play soccer against a hand-crafted opponent in RoboCup 2D simulation. This work proposed Policy Search Planning (PSP). In PSP, a plan is a team tactic. The designer can define plenty of plans by hand to establish a plan pool shared by all agents in advance. If the world state satisfies the precondition of a plan, the plan will be called an active plan and it could be executed. Policy search is used to find the optimal team policy in selecting these active plans. Kalyanakrishnan et. al, 2010 [39], extended previous work on Keepaway Soccer for RoboCup 2D simulation in order to learn, on top of a passing behavior, a get open behavior for the keepers without the ball. The passing behavior was learned exactly as in a previous work by using SARSA(λ) independently for each agent. The get open behavior was learned through policy gradient over a neural network. Parameter sharing was employed for the latter, so that only one network was trained based on the experiences of all keepers. Both behaviors were learned in an alternating fashion. Leonetti et. al, 2011 [40], combined Petri Net Plans (PNP) with Reinforcement Learning. This framework allows designing a partially specified plan for describing a complex task, using PNPs with non-deterministic choice points. Then, Reinforcement Learning is applied to learn which the optimal policy at these choice points is. This method was tested in the Keepaway Soccer domain, where keepers with the ball learned to choose between holding and passing the ball, while keepers without the ball learned to get open or the go to the closest corner in the field. Multiagent Reinforcement Learning with Regret Matching for Robot Soccer, Liu et. al, 2013 [41], enhanced Nash-Q with regret matching to speed up learning, where regret matching is expected to improve the relation between exploration/exploitation. This new algorithm was tested in a 2D simulation of 3 vs 3 soccer. Both teams learned concurrently over joint team policies, one using the new algorithm and the other using Nash-Q.

Hausknecht, 2016 [3], implemented a solution to a 2 vs. 1 Half-Field Offense domain in RoboCup 2D simulation with only a goalkeeper in the opponent team, which followed a fixed policy. Policies were learned at the level of primitive actions: dash(power, direction), turn(direction), tackle(direction), kick (power, direction). Learning was accomplished by using Deep Deterministic Policy Gradient (DDPG) over a parametrized action space. Experiments were carried out for IL and JAL approaches. Neither of them solved the task satisfactorily, little to none coordination was observed between agents; and particularly, JAL did not even discover a policy that would complete the task occasionally. Mendoza et. al, 2016 [42], trained a soccer team for resolving a free kick situation against opposing teams with static policies. Experiments were conducted using a physics-based SSL (Small-Size League) soccer simulation. The planning space of the problem was reduced by framing it as a multi-armed bandit problem, in which pre-computed plans are treated as actions, and the state is the position of the free kick on the field. Online learning was then performed by modeling an approximation of the reward function using Gaussian Processes-based regression, while employing the Upper Confidence Bound strategy for action selection.

Chapter 3
(Deep) Reinforcement Learning

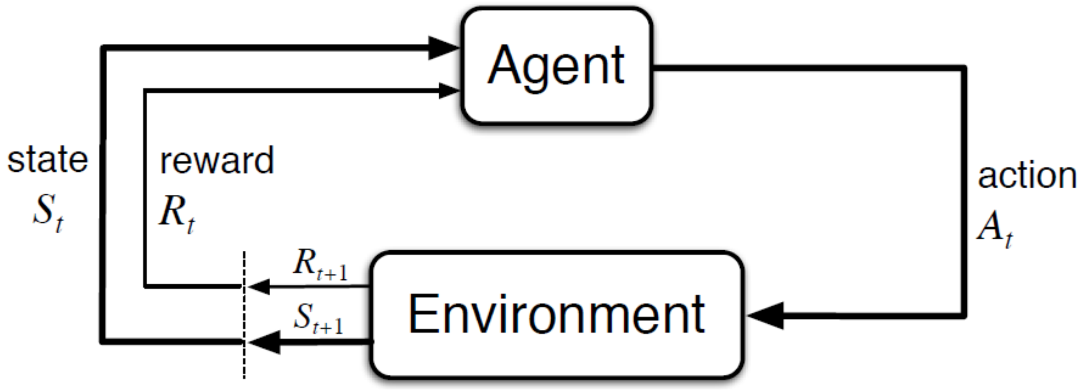
3.1. Markov Decision Processes (MDP)

MDPs [43] are a formalism for modeling sequential decision problems. An MDP is a tuple $\langle S, A, \delta, R \rangle$, comprising:

- The set S of all states in the world,
- The set A containing every action a learning agent can execute within its world,
- A probability distribution over transitions $\delta: S \times A \times S \rightarrow [0,1]$ from a current state to a next state given the action performed by the agent in the starting state,
- And a reward function $R: S \times A \rightarrow \mathbb{R}$ that quantifies the immediate goodness of the action selected in the current state.

According to this definition, MDPs encode concisely the interactions between agents and their environments (see figure below), which can be interpreted as follows. At timestep t the agent is allowed to observe fully or partially the current state S_t. After collecting this information, the agent selects and executes action A_t. As a result, it is granted with a reward $R(S_t, A_t)$, and the world state changes probabilistically to S_{t+1} abiding by $S_{t+1} \sim \delta(S_t, A_t)$. This cycle is repeated until the agent arrives at a final state.

Figure 1. The agent-environment interaction represented by an MDP.



Given an MDP, the goal is to find an optimal policy, i.e. a mapping from states to probabilities of selecting each action, that maximizes the expected sum of discounted rewards. Where a discounted reward is the result of applying a discount rate $\gamma \in [0,1]$ to the value of a future reward. In this way, γ denotes an additional parameter in the MDP formulation that controls how much effect future rewards have on the optimal decisions, with small values focusing on the short-term and larger values on the long-term.

It must be mentioned that MDPs are a proper representation only for problems that satisfy the Markov Assumption. Such assumption requires, in essence, that the next state and immediate reward depend solely on the current state known to the agent and on its current action; or in other words, they should not depend on the history of states and actions, or on hidden state information.

3.2. Reinforcement Learning

3.2.1. Definitions

Reinforcement Learning is concerned with learning in MDPs; usually without having a priori a model of the domain, i.e. without explicit knowledge of the next-state probability distribution and of the reward function. The main characteristics of RL is that learning is accomplished through raw experience, by actually trying out actions in the environment and then perceiving their positive or negative effects, represented as rewards; and that learning is synonymous with the maximization of rewards received during a complete episode, which means that not only the immediate but also the delayed consequences of actions must be taken into account [44].

As mentioned before, the core objective of RL is to find an optimal policy π^* for the underlying MDP. Where a policy π fully encodes the learning agent´s behavior and is defined as a generally stochastic mapping from perceived states of the environment to actions to be taken when in those states. For instance, at timestep t, policy $\pi_t(a|s)$ denotes the probability that action $A_t = a$ will be executed by the agent in state $S_t = s$. Policies could also be stationary, such as $\pi_t(s) \in A$, meaning that agents will always execute the same action in a particular state. Several structures have been used for representing policies; classical Reinforcement Learning commonly employs tables, with one entry per state; whereas, Deep Reinforcement Learning, for example, relies on deep neural networks.

On the other hand, the optimality criteria used in this work corresponds to the maximization of the cumulative reward collected by the agent during an entire episode. Formally, if an agent with policy π is at state S_t after t timesteps, and thereafter it receives a sequence of future rewards of $\{R_{t+1}, R_{t+2}, \dots\}$, then, for a discount rate γ, the cumulative reward G_t expected when starting at state S_t is expressed as:

$$\begin{aligned} E_\pi[G_t|S_t = s] &= E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s] \\ &= E_\pi\left[\sum_{k=0} \gamma^k R_{t+k+1} \,|S_t = s\right] \end{aligned}$$

Naturally, a stationary optimal policy can be determined as follows:

$$\pi^*(s) = \mathrm{argmax}_\pi E_\pi\left[\sum_{k=0} \gamma^k R_{t+k+1} \,|S_t = s\right], \forall s \in S$$

The term $E_\pi[G_t|S_t = s]$ is also called the value of state s under policy π, and can be thought as the expected cumulative reward the agent would receive if it follows π starting in s. A function v that encodes this information for every possible state in S is known as a value function over states, which is defined as:

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

The previous definition can be extended to every possible state-action pair in $S \times A$, i.e. defining the expected cumulative reward the agent would collect by following π after executing a in s, and in this case such function is usually known as action-value or q-function, which is formalized as:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a\,] = E_\pi\left[\sum_{k=0} \gamma^k R_{t+k+1}\,|S_t = s, A_t = a\right]$$

Several Reinforcement Learning algorithms, to be discussed next, involve estimating these value functions over states or state-action pairs, that indicate how beneficial in the long-run, i.e. in terms of cumulative future rewards, it is for the learning agent to be in a given state or to perform a given action in a specific state.

3.2.2. Temporal-Difference (TD)

TD methods follow a value-based approach, that is, the learning agent selects an action directly from the currently learned state-value or action-value function. They consist of two alternating steps: policy prediction and policy improvement.

During policy prediction, the state-value or action-value function associated with the current policy π is learned through iterative updates. Each update tries to move the current estimates $V(S_t = s) \approx v_\pi(s)$ or $Q(S_t = s, A_t = a) \approx q_\pi(s,a)$ for a given state s and action a closer to a target, which by definition should be the expected cumulative rewards $E_\pi[G_t|S_t = s]$ or $E_\pi[G_t|S_t = s, A_t = a]$. However, as these expectations are unknown in RL problems, TD methods approximate them with partial estimates of samples, as follows:

$$E_\pi[G_t|S_t = s] \approx G_t \approx R_{t+1} + f(\gamma, \lambda, V(S_{t+1}), V(S_{t+2}), \dots)$$

$$E_\pi[G_t|S_t = s, A_t = a] \approx G_t \approx R_{t+1} + f(\gamma, \lambda, Q(S_{t+1}, A_{t+1}), Q(S_{t+2}, A_{t+2}), \dots)$$

Where, R_{t+1} corresponds to the immediate reward collected by the agent, and $f(\cdot)$ is a function depending on the discount rate γ, on a decay rate λ, and the current state-value or action-value estimates for future states and actions (lookaheads). TD methods differ mainly on how this latter function is computed.

Updates are then implemented in one of the following ways, after introducing a learning rate α, in order to gradually reduce the error with respect to the latest target:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + f(\gamma, \lambda, V(S_{t+1}), V(S_{t+2}), \dots) - V(S_t)]$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + f(\gamma, \lambda, Q(S_{t+1}, A_{t+1}), Q(S_{t+2}, A_{t+2}), \dots) - Q(S_t, A_t)]$$

During policy improvement, policy π is updated towards greediness with respect to the current estimate for $v_\pi(s)$ or $q_\pi(s, a)$. In other words, the new policy is defined such that at every state the agent will execute the action reporting the highest value.

The following sections present two popular TD algorithms, namely: SARSA and Q-Learning. These will be described under the assumption that the next-state probability distribution and the reward function are unknown a priori; hence, only q-functions could be learned.

3.2.2.1. SARSA

SARSA [45] is an on-policy algorithm, which means that the target will be estimated based on policy π computed from the current estimate Q. It makes a lookahead of exactly one step into the future; thus, $\lambda = 1$ and this parameter now has no relevance in the estimation. The target is computed as $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$, where the next state S_{t+1} is observed by the agent and the future action A_{t+1} follows $\pi_t(a|s)$ or it is an exploratory action. The complete algorithm is presented below:

Box 1. SARSA algorithm.

```
1: Initialize $Q(s,a) \leftarrow 0, \forall s \in S, \forall a \in A$
2: Observe current state $S_0$
3: For each timestep $t = 1, \dots, T$ (until $S_t$ is a final state)
4:     Choose an action $A_t$ using policy derived from $Q$ given $S_t$ (e.g. $\epsilon$-greedy)
5:     Execute action $A_t$
6:     Observe next state $S_{t+1}$
7:     Collect immediate reward $R_{t+1}$
8:     Choose an action $A_{t+1}$ using policy derived from $Q$ given $S_{t+1}$ (e.g. $\epsilon$-greedy)
9:     Update Q-value as follows: $Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})]$
10:    Set $S_t \leftarrow S_{t+1}$
10: Optimal policy: $\pi^*(s) = \text{argmax}_{a \in A} Q(s,a)$
```

3.2.2.2. Q-Learning

Q-Learning [46] is an off-policy algorithm; therefore, the target estimation will not depend on the current policy. Instead, at every update Q-Learning takes the maximizing action. Like SARSA, it looks a single step ahead into the future. As a result, the target is estimated as $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$, $\forall a \in A$, where the next state S_{t+1} is observed by the agent and $\max_a Q(S_{t+1}, a)$ is a maximization over the currently learned action-value function given S_{t+1}. A pseudocode of Q-Learning is shown in the following box:

Box 2. Q-Learning algorithm.

```
1: Initialize $Q(s,a) \leftarrow 0, \forall s \in S, \forall a \in A$
2: Observe current state $S_0$
3: For each timestep $t = 1, \dots, T$ (until $S_t$ is a final state)
4:     Choose an action $A_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
5:     Execute action $A_t$
6:     Observe next state $S_{t+1}$
7:     Collect immediate reward $R_{t+1}$
8:     Update Q-value as follows: $Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a)]$
9:     Set $S_t \leftarrow S_{t+1}$
10: Optimal policy: $\pi^*(s) = \text{argmax}_{a \in A} Q(s,a)$
```

3.2.3. Policy Gradient (PG)

PG applies an opposite approach to TD methods. While the latter learns value functions that implicitly define policies, the former entirely disregards value functions and directly learns in the policy space. Hence, the goal of PG is to learn a parametrized policy π_θ that given a set of free parameters θ receives states as inputs and outputs a probability distribution over actions in the general case.

To this end, first, a score function $J(\theta)$ must be defined, that indicates how good policy π_θ really is. Abiding by the optimality criteria established earlier, this score function is set in terms of the cumulative reward the agent would gather on average per episode starting from the initial state S_0:

$$J(\theta) = E_{\pi_\theta}\left[\sum_{k=0} \gamma^k R_{k+1} \,|S_0\right]$$

Afterwards, PG alternates between two steps: policy interaction and policy improvement. In policy interaction, the learning agent enforces the current policy π_θ added with some random perturbation $\tilde{\pi}$ in its environment as it gathers examples of the form $[S_t, A_t, R_{t+1}, S_{t+1}]$, where $A_t = \pi_{\theta,t}(a|s) + \tilde{\pi}$. Then, in policy improvement, gradient ascent is employed to maximize the score function with respect to its parameters θ. This is done by using the collected examples of every complete episode to compute the direction of improvement $\nabla_\theta J(\theta)$, and lastly by applying the following update rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

The box below provides a rough outline of a PG implementation:

Box 3. Policy Gradient algorithm.

```
1: Initialize policy parameters $\theta$
2: Repeat:
3:     Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ using $\pi_\theta(A_t = a|S_t = s) + \tilde{\pi}$
4:     Compute gradient from collected data $\nabla_\theta J(\theta)$
5:     Update parameters $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
```

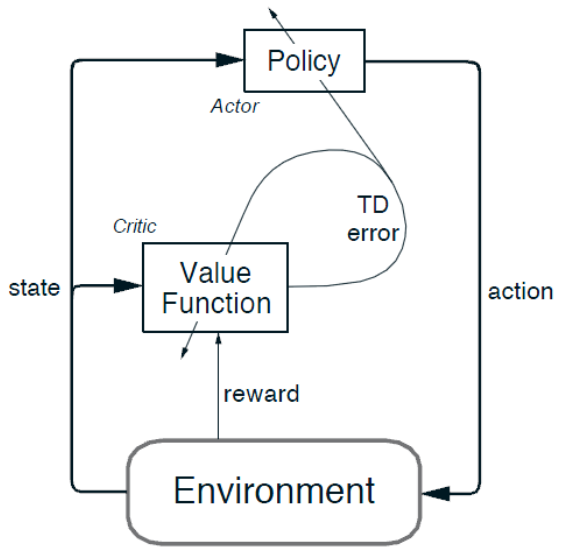
An important advantage of PG over TD is that it enables the application of Reinforcement Learning in very large or even continuous action spaces, since it learns the policy smoothly, instead of having the compute it at every step by maximizing over the entire action space associated with each state. A disadvantage of PG is that it is episodic in nature, i.e. it needs information of whole episodes in order to determine $J(\theta)$ and then $\nabla_\theta J(\theta)$; this can lead to slow convergence in some domains.

3.2.4. Actor-Critic (AC)

AC is a hybrid that combines temporal-difference and policy gradient approaches. It considers two separate learning structures (see figure below); on the one hand, there is a critic responsible for learning a parametrized action-value function; and on the other, there is an actor that directly learns a parametrized policy and that is in charge of selecting the agent´s actions. Improvement in both cases is driven by the TD error signal produced by the critic.

At the same time, actor-critic methods take the best of both worlds. They can handle very large or continuous action spaces just like PG, because they likewise learn an explicit policy; while they also take advantage of the fast convergence of TD, since they also apply bootstrapping to estimate the action-value target based on current action-value estimates, which allows them to potentially learn after every single step.

Figure 2. Actor-Critic architecture.



3.2.5. Action Selection

As it can be noticed in previous algorithms, some kind of exploratory behavior is always needed whenever the agent is required to select an action, as this is necessary in order to learn new and better policies; otherwise the same policy would remain unchanged forever. The simplest way to promote exploration is by adding some small randomness to the actions executed by the learning agent. Nonetheless, more complex solutions also exist which try to drive exploration towards uncertain or novel situations.

In this work, the simple ϵ-greedy selection rule was applied in all cases. This rule requires an $\epsilon \in [0,1]$ parameter to defined beforehand, which could also be allowed to decay with some rate as episodes go by. Then, every time the agent is requested to choose an action, a random number is generated between 0 and 1. If such number is higher than ϵ, the agent selects an action according to its current policy. If, otherwise, such number is lower than ϵ, the agent selects an action completely at random.

For discrete action spaces, ϵ-greedy is formalized as follows, where the exploratory selection simply implies to pick uniformly at random an element from a finite action set:

Box 4. ϵ-greedy strategy.

```
1: Given $\epsilon \in [0,1]$
2: Pick random action $a \in A$ with probability $\epsilon$
3: Otherwise, take the best action $a^* = \text{argmax}_{a \in A} Q(s,a)$ with probability $1 - \epsilon$
```

In continuous action spaces, an exploratory action can be specified by picking uniformly at random a point within the boundaries of the corresponding action space. Another alternative that was employed in this thesis is to add a small random perturbation to the action that would have been determined by the current policy.

3.3. Deep Reinforcement Learning

Deep RL combines classical reinforcement learning with a class of artificial neural network known as deep neural networks. The main benefits of this fusion are: 1) Deep RL better handles very large or continuous state and action spaces, 2) Deep RL is able to generalize across states, i.e. to provide optimal actions in states never encountered before, a feature that was lacking in classical tabular RL.

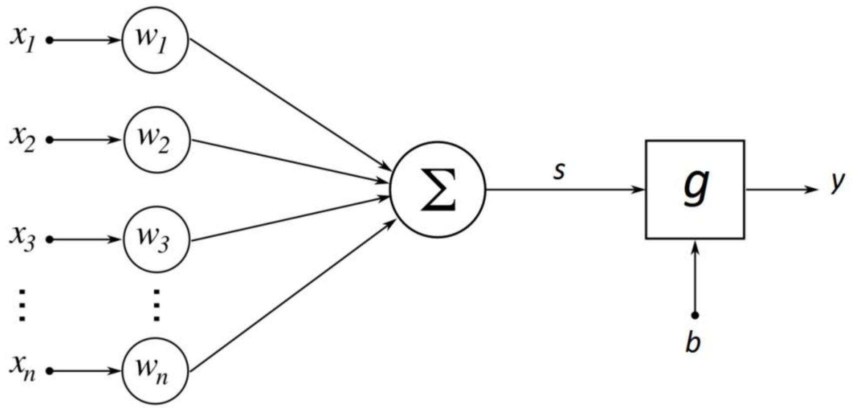
This section will start by giving a brief review of deep neural networks, and afterwards Deep RL hallmark algorithms and techniques will be presented.

3.3.1. (Deep) Neural Networks

Artificial neurons and networks have been around since the 1950s [47]. Artificial neurons are simplistic mathematical models that aim to simulate some information processing capabilities found in biological neurons. A basic artificial neuron is shown in the figure below, and can be formalized in the following equation:

$$y(\boldsymbol{x}) = g[s] = g\left[\sum_{i=1}^{n} w_i x_i + b\right]$$

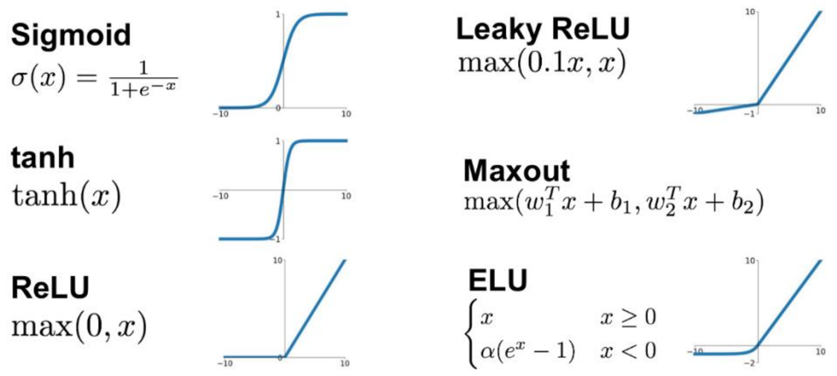
Figure 3. Basic model of an artificial neuron.



Where, $\boldsymbol{x}$ is an input vector and x_i are its components, w_i represent scalar synaptic weights that perform a linear combination s of the input, b is an additional weight connected to a bias input that is always $+1$, $g[\cdot]$ is an activation function $\mathbb{R}^n \to \mathbb{R}$, and y denotes the resulting scalar output. For completeness, it must be mentioned that there are other types of artificial neurons different from this basic unit, such as: Radial Basic Functions (RBFs), Continuous Time-Recurrent Networks (CTRNNs), Long Short-Term Memory Cells (LSTMs), Spiking Neural Networks (SNNs), and several others.

The choice of activation function is crucial for the behavior of an artificial neuron. Various alternatives have been proposed in the literature in order to tackle different kinds of problems, including: linear functions, sign functions (hard limiter), logistic functions, hyperbolic tangent functions, and nowadays the rectified linear unit function (ReLU) is the preferred choice in most deep learning solutions, as it usually enables a faster convergence and it is more resilient to the vanishing gradient problem.

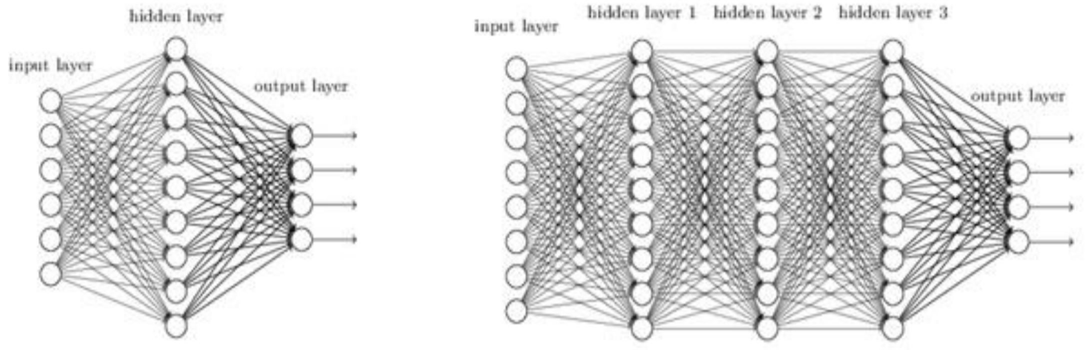
Figure 4. Various activation functions.



Artificial neural networks (ANNs) are architectures consisting of several artificial neurons, each connected to at least one other neuron, and all them working together both in parallel (within layers) and sequentially (across layers). Technically speaking, ANNs are non-linear adaptive systems that can be optimized to approximate any given function. In fact, the Universal Approximation Theorem for ANNs [48] states that a shallow network (having only one hidden layer of neurons and another output layer) can approximate to any degree of accuracy a continuous function provided enough neurons and a proper non-linear activation function.

Given this result, shallow networks played the lion role in ANNs for many years, as there were easier to train, and according to theory they could solve any problem. However, the previous theorem only shows potentiality, but it does not guarantee that there will always exists an algorithm capable of carrying out the training or that such approach will be the most efficient in all situations. In practice, shallow networks are not suitable for dealing with highly complex tasks, and this led to the development and current popularity of deep neural networks. These consist of two or more hidden layers connected sequentially, reaching even hundreds of layers. They are certainly harder to train, but they have proven to outperform shallow networks in a variety of difficult problems.

Figure 5. Shallow (left) and Deep (right) neural networks.



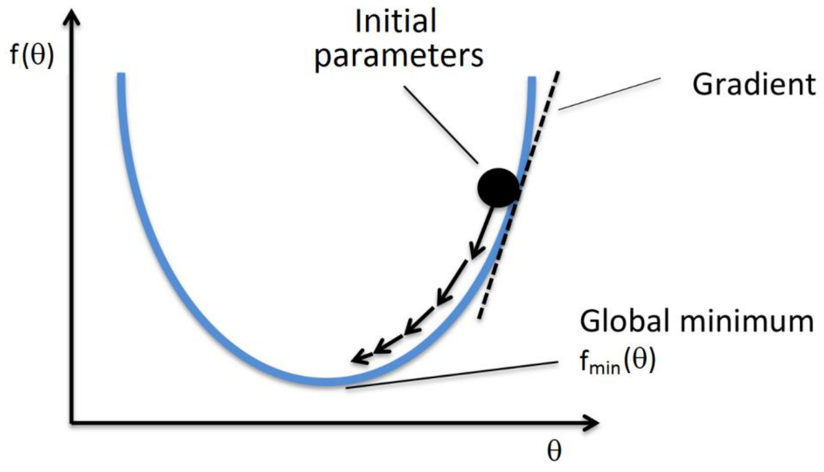
Large numbers of neurons can be arranged into a limitless number of network architectures. Three commonly used architectures are: sequential, recurrent and convolutional. In a sequential network, layers of neurons are stacked one after the other, all outputs of a given layer are connected only to its subsequent layer, and the outputs of the last layer do not go back to the network. This is the architecture employed in the present study. A recurrent architecture also stacks multiple layers, but their outputs, besides connecting to the subsequent layer, are allowed to loop back into network. Recurrent networks are useful for learning temporal relationships, when the final output depends on the history of individual inputs. Finally, convolutional networks aim at efficiently extracting spatial patterns from inputs by means of convolution operations. In this way, they can be used to encode the most relevant information of high-dimensional spaces (for example: images) into a much lower dimensional spaces (feature vectors).

3.3.2. Stochastic Gradient Descent (SGD) Optimization

Gradient descent is a centuries-old technique, whose core objective is to find the position of a minimum point in a continuous and differentiable function. Its key idea is to iteratively follow the anti-gradient direction computed at each current position, which always points towards a minimum. For instance, let f be a continuous and differentiable function with parameters θ, and α be a scalar learning rate, then at any given point θ_t the anti-gradient is defined as $-\nabla_\theta f(\theta)|\theta_t$ and the update rule of gradient descent takes the form:

$$\theta_{t+1} = \theta_t - \alpha\nabla_\theta f(\theta)|\theta_t$$

Figure 6. Gradient descent outline.



Gradient descent can be used for training ANNs under a supervised learning framework. To do so, the network θ must be provided with N matching pairs of example inputs $\boldsymbol{X} \in \mathbb{R}^{N\times d}$ and their associated desired targets $\boldsymbol{d} \in \mathbb{R}^N$. Afterwards, the inputs are fed into the network to generate predictions $\boldsymbol{y}(\theta) \in \mathbb{R}^N$, which are then compare against the targets with the help of a loss or error function $\mathcal{L}(\boldsymbol{d}, \boldsymbol{y}(\theta)) \in \mathbb{R}$ that outputs zero if the both vectors are identical, and a value greater than zero otherwise. Clearly, to get better predictions, θ should be set as the global minimum of the loss function. It is at this point that gradient descent can be applied to try to find such minimum.

The literature is saturated with numerous loss functions designed with diverse goals in mind. Some of these options are: mean square error (MSE), mean absolute error (MAE), cosine error, hinge loss, logistic loss, categorical cross-entropy loss, binary cross-entropy, likelihood loss, Kullback-Leibler loss, and several others. For example, MSE is widely-used in Deep RL and it is formulated as:

$$\mathcal{L}(\boldsymbol{d}, \boldsymbol{y}(\theta)) = (\boldsymbol{d} - \boldsymbol{y}(\theta))^T \cdot (\boldsymbol{d} - \boldsymbol{y}(\theta))$$

An important consideration to make for deep neural networks is that due to their intrinsic high non-linearity, any loss function defined for them will be riddle with local minima and in many cases this will correspond to bad local minima. Since gradient descent finds any minimum regardless of whether it is local or global and then it just stops, this vanilla formulation it is not suitable for dealing with deep networks. A solution is to implemented stochastic gradient descent, which is an on-line (as oppose to batch) variant of gradient descent in which a uniformly random perturbation e_t is added during each update. Adding such perturbation to the anti-gradient produces new directions of improvement that can actually move SGD away from local minima. SGD's learning rule is the following:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta)|\theta_t + e_t$$

At this point, it should be mentioned that evaluating the gradient of a loss function with respect to the weights of a neural network ($\nabla_\theta \mathcal{L}(\boldsymbol{d}, \boldsymbol{y}(\theta))$) is not a trivial task. It was only in the 80s that an efficient method for computing these gradients, called Backpropagation [49], was conceived for sequential ANNs. As its name suggests, this techniques propagates, thanks to an ingenious application of the chain rule of derivatives, the errors and gradients observed in the output layer backwards to the previous hidden layers. The following box contains a pseudocode implementation of Backpropagation for a sequential network and considering MSE as loss function:

Box 5. Backpropagation algorithm.

```
1:  Forward phase:
2:      For $j = 1,2,\dots,N_0$, where $N_0$ is to the number of inputs to the network:
3:          Set inputs: $x_j^0 = x_j$
4:      For $l = 1,2,\dots,L$, where $L$ is to the number of hidden layers:
5:          For $k = 1,2,\dots,N_l$, where $N_l$ is to the number of neurons in hidden layer $l$:
6:              Set bias: $x_0^{l-1} = 1$
7:              Compute linear combination of inputs: $s_k^l = \sum_{j=0}^{N_{l-1}} w_{kj}^l \, x_j^{l-1}$
8:              Compute outputs: $x_k^l = g(s_k^l)$
9:      For $j = 1,2,\dots,N_L$:
10:         Set outputs: $y_j = x_j^L$
11: Backward phase:
12:     For $l = L, L-1, \dots, 1$:
13:         For $k = 1,2,\dots,N_l$:
14:             Compute prediction and local error:
15:                 $e_k^l = \begin{cases} d_k - y_k & if\ l = L \\ \sum_{i=0}^{N_{l+1}} w_{ik}^{l+1} \, \delta_i^{l+1} & if\ l < L \end{cases}$
16:             Where $d_k$ are the targets
17:             Compute local gradients: $\delta_k^l = e_k^l \cdot g'(s_k^l)$
18:             For $j = 1,2,\dots,N_{l-1}$:
19:                 Update connection $w_{kj}^l$ from gradient $\delta_k^l \cdot x_k^{l-1}$
```

SGD together with Backpropagation could be used to train ANNs, but especially for deep networks they would likely report overfitting, slow convergence and possibly divergence, oscillations and instability. Deep networks still carry with them several inherent problems, such vanishing gradients, exploding gradients and model degeneracies. Nevertheless, at the present, there is a large battery of tricks that can alleviate this inconveniences, making training manageable in most cases; for instance:

- Momentum [50]: Adding a momentum term to SGD can speed up convergence, as updates tend to follow previous directions.
- Batch normalization [51]: Normalizing, usually to zero mean and unit variance, the distributions of every layer in a network reduces covariance shift and increases convergence speed.
- ReLU [52]: It is less affected by vanishing gradients than logistic or hyperbolic tangent functions, since its derivate is not constrained to values smaller than one as it is the case with the latter two.
- Regularization: Helps tackling overfitting by directly constraining the degrees of freedom possessed by a neural network.
- Dropout [53]: Also alleviates the problem of overfitting by randomly disconnecting at each learning step many weights of the network. Ideally, this promotes generalization among neurons and connections, thus, the overall network does not rely on just a few parameters.
- Faster hardware: In the end, the reborn interest in deep neural networks has been mostly driven by the significant increase in computational power in recent years. In particular, the widespread use of GPUs has been a game-changer for deep learning.

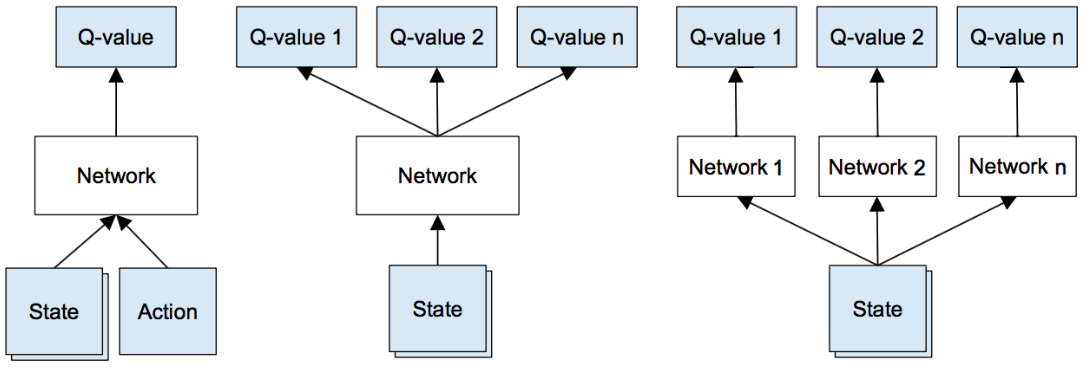
Nowadays, there are many ready to use optimizers based on SGD and Backpropagation that already incorporate the previous techniques along with additional improvements. A short list of these includes: RMSprop, Adagrad [54], Adadelta [55], Adam [56], Adamax [56], Nadam [57].

3.3.3. Deep Q-Network (DQN)

DQNs [58] are essentially the implementation of classical Q-Learning (although other TD methods can be implemented in the same way), while using deep neural networks to approximate the action-value function to be learned.

Different deep neural architectures have been proposed for representing q-functions (see figure below). For example, three common structures are: 1) using a single network fed with state $S_t = s$ and action $A_t = a$, which outputs the action value $Q(s, a)$, 2) using a single network fed only with state $S_t = s$ which outputs the action values for every action $a \in A$, 3) using multiple networks, as many as the cardinality of the action set $|A|$, each fed only with state $S_t = s$, so that each would output the action value corresponding to a different action.

Figure 7. Various DQN architectures.



When working with DQNs, the prediction step of standard Q-Learning is replaced with the training of the deep neural network with parameters θ by means of some variant of gradient descent. This training is performed at the level of each state-action pair (S_t, A_t). And the targets provided to the network are computed exactly as in standard Q-Learning. For instance, for the first architecture shown in the previous figure, and considering an interaction such that action A_t was executed in state S_t generating a reward R_{t+1} before transitioning to state S_{t+1}, then the examples (inputs) given to the network will be precisely the pair (S_t, A_t) and its associated target will be $R_{t+1} + \gamma \max_a Q(S_{t+1}, a|\theta), \forall a \in A$, where each $Q(S_{t+1}, a|\theta)$ is obtained by feeding the current network with the corresponding pair (S_{t+1}, a).

Moreover, the improvement step is no longer enforced as such; hence, the policy is never expressed in its explicit form. Instead, whenever the agent, as it interacts with the world, has to select an action according to its underlying policy, what is done is that the current state is fed to the DQN, which computes the Q-values associated with every possible action, and finally the action with the highest Q-value is greedily chosen as usual.

3.3.4. Deep Deterministic Policy Gradient (DDPG)

Just like Q-Learning, DQNs are not easily applicable to continuous action spaces. DDPG [59] was developed precisely to overcome this issue. It is a straightforward Deep RL extension of the actor-critic architecture, consisting of two separate deep neural networks. The critic learns a parametrized (θ^Q) q-function $Q(S_t = s, A_t = a|\theta^Q)$. Meanwhile, the actor learns and executes a parametrized (θ^π) policy $a = \pi(S_t = s|\theta^\pi)$.

The critic is trained in a similar fashion as in DQNs, by applying gradient descent and relying in bootstrapping. It is also given pairs (S_t, A_t) as examples. However, the targets are no longer estimated from a maximization procedure, since that becomes very expensive in continuous action spaces. Now, the best action in the next state S_{t+1} is retrieved directly from the current actor $A_{t+1} = \pi(S_{t+1} = s|\theta^\pi)$, which is expected that it will indeed converge towards an optimal policy. Thus, targets will be computed as follows:

$$R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1}|\theta^{\pi})|\theta^{Q})$$

On the other hand, the actor's goal is to minimize for each example the difference between its current output $a = \pi(S_t|\theta^{\pi})$ and the optimal action. In theory, the optimal action could be estimated from the critic, after realizing a maximization over the entire action space given a particular state. Yet, that is exactly what policy gradient methods are supposed to avoid.

Instead, the actor is trained in a smarter way by taking advantage of the gradients of the critic with respect to its action inputs $\nabla_a Q(S_t = s, A_t = a|\theta^{Q})$ [60], which are computed only after the critic has been updated with the latest targets. Since a is parametrized by θ^{π}, such gradient term provides a vector of improvement implicitly relative to θ^{π} that internalizes recent improvements in Q-values, which is then concatenated with the gradient of a with respect to θ^{π} to complete the chain rule and to determine an final direction of improvement, as seen below:

$$\nabla_{\theta^{\pi}} \pi(S_t) = \nabla_a Q(S_t, A_t|\theta^{Q}) \nabla_{\theta^{\pi}} \pi(S_t|\theta^{\pi})$$

3.3.5. Further Improvements

If DQN or DDPG were to be implemented as described above, learning would likely turn out to be unstable. The following two mechanisms have been commonly used to overcome this problem.

3.3.5.1. Experience Replay

One cause for this instability comes from the fact that when performing SGD optimization over deep networks it is assumed that samples are independent and identically distributed. However, that is not the case in Deep RL, examples collected from the same episodes are correlated to each other, and over larger scales, examples are biased by the preferences of the current policy.

Experience replay was first applied in [58]. The basic idea is to store examples $[S_t, A_t, R_{t+1}, S_{t+1}]$ gathered from a large number of episodes in a replay buffer. Afterwards, off-policy updates are enforced by drawing a batch of random experiences from this buffer. This selection process effectively removes correlations existing in the sequentially observed examples.

This has some intrinsic limitations as the replay buffer does not differentiate salient examples, gives equal importance to all examples and always overwrites with recent examples due to a finite memory. A more sophisticated strategy, called Prioritized Experience Replay, has been recently proposed in [61]. In this case, examples are stochastically drawn from the buffer with probability proportional to its importance, defined in terms of the current prediction uncertainty relative to each example, i.e. in terms of the TD error associated with each instance.

3.3.5.2. Target Networks

When using a neural network to approximate a q-function, each update aim at improving a single action-value $Q(S_t, A_t)$ will necessarily modify several weights of the network, and consequently, this will also modify in one way or another all current predictions of the network. Since both, DQN and DDPG, rely on bootstrapping, the fact that each update alters the estimates $Q(S_{t+1}, A_{t+1})$ used to compute that same update is a major source of instability during learning, usually causing oscillations or divergence of the policy.

Mnih et. al [58] initially introduced Target Networks to help with this problem in the context of DQN. In this case, a Target Network is a separate deep Q-Network whose goal is also to correctly predict action-values, which can then be employed to obtain the estimates needed to determine the training targets for the original DQN. However, whereas the latter is updated at every learning step, the former is updated less frequently; for example, once every C updates of the original DQN. As a result, this Target Network provides slow-varying estimates that avoid destabilizing the learning process.

A common way to smoothly update a Target Network θ' is to take a large moving average with respect to the original DQN θ given a small learning rate τ, resulting in:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

When adding a Target Network the DQN update rule is modified roughly as follows:

$$\mathcal{L}(\theta) = \left([R_{t+1} + \gamma \max_a Q(S_{t+1}, a|\theta')] - Q(S_t, A_t|\theta)\right)^2$$

$$\theta_{t+1} = \theta_t - \alpha\nabla_\theta \mathcal{L}(\theta)|\theta_t$$

In the case of DDPG, since the actor learns through the TD error as well, two Target Networks are required, one for the critic θ'^{Q} and one for the actor θ'^{π}, resulting in the following update rules:

$$\mathcal{L}(\theta^Q) = \left([R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1}|\theta'^{\pi})|\theta'^{Q})] - Q(S_t, A_t|\theta^Q)\right)^2$$

$$\theta^Q_{t+1} = \theta^Q_t - \alpha^Q \nabla_{\theta^Q} \mathcal{L}(\theta^Q)\big|\theta^Q_t$$

$$\theta^\pi_{t+1} = \theta^\pi_t - \alpha^\pi \nabla_a Q\left(S_t, A_t\big|\theta^Q_t\right)\nabla_{\theta^\pi}\pi(S_t|\theta^\pi_t)$$

3.4. Multi-agent Reinforcement Learning

Application of RL in multi-agent domains has been researched since the 90s. Since then, two main and opposing learning paradigms have been proposed and investigated, namely Independent Learners and Joint-Action Learners. These are described in the following sections. There is also a third approach, known as Coordinated Learners, that essentially combines ideas from the previous two.

3.4.1. Independent Learners (IL)

The first implementation of IL can be traced back to [4] in 1993. The idea behind Independent Learners is fairly simple; just place all learning agents in the same environment allowing them to interact with each other, and each applying its own RL algorithm; for example, Q-Learning or DQN. There is no direct communication among agents; thus, the state space of each agent will correspond only to the portion of the world said agent is able to perceive, furthermore, no agent is explicitly aware of the actions executed by other agents, although these could be inferred from the world state.

Additionally, each agent receives an independent reward, which can be problematic especially for cooperative tasks. For instance, in soccer, one would like to define per agent a global reward of +1 per scored goal, but this would incentivize every agent of the same team to attempt to score on its own, likely obstructing each other as a result. Therefore, almost irrevocably, a human expert must design a shaped reward that fosters cooperation between agents under IL; for example, going back to the previous case, a positive reward granted for passing the ball to a teammate could be added to each agent's overall reward.

This necessity for shaped rewards in cooperative tasks is ultimately a limiting factor for IL. As shaped rewards are usually riddle with bad local minima, as well as human bias. Moreover, shaped rewards are unable to handle tasks for which experts do not know a proper reward function, and would also hinder the emergence of systems with superhuman capabilities.

On top of this, in IL, from the point of view of each individual the world appears non-stationary as other learning agents are continuously adapting. Such non-stationarity leads to several serious problems. For starters, let's say that one time each of two agents select the right action for a particular cooperative situation, and as a result, both collect a high reward; then, on another time and for an identical situation, one agent performs the same correct action, while the other chooses an exploratory action that is far from optimal, resulting in a poor reward for both agents. This example reveals just how noisy the reward signal perceived by a single agent can become due to non-stationarity, and this problem is exacerbated by a larger number of agents, since then there will always be an agent performing an explorative behavior.

Hysteretic Q-Learning is a fairly satisfactory solution against the previous problem. It was originally proposed in [7] and extended to a deep RL context in [10]. Its key idea is to learn two different action-values for the same state-action pair. One activates if the reward received by the individual is high and the other if the reward is low. In this way individuals are not penalized when they choose an optimal action, but the global joint action is suboptimal. Nonetheless, Hysteretic Q-Learning performs poorly in highly stochastic domains.

Another issue arising from non-stationarity is that agents might get stuck in a never-converging spurious cycle of adaptations involving sub-optimal policies. This occurs when, for example, two agents go back and forth between two sub-optimal policies each, as they continuously try to adapt in response to the changes in the other agent's policy. Two workarounds proposed for overcoming this difficulty are: Alternating Learners and Homogenous Learners; which are presented next:

3.4.1.1. Alternating Learners

According to this approach, only one agent is allowed to learn at a time, while the remaining agents maintained their policies momentarily fixed. In the following learning steps, the identity of the sole learner is alternated cyclically. If the consecutive learning periods are long enough, then non-stationarity can be significantly reduces. However, a drawback of Alternating Learners is its inefficiency, since it requires all agents to interact together every time, but only one of them improves.

3.4.1.2. Homogeneous Learners

This approach is applicable only for cooperative tasks and when all agents have homogenous state and action spaces. In this scenario, it is possible to learn a unique policy shared by all teammates. Likewise, in the context of deep RL, only a single experience replay buffer needs to be maintained. Clearly, Homogeneous Learners alleviates the problem of non-stationarity since there effectively exists only one policy. It also efficiently makes use of the experiences collected by all teammates.

3.4.2. Joint-Action Learners (JAL)

This approach was first proposed by Littman in 1994 [22]. The key idea is to reformulate the learning problem such that instead of having as many independent MDPs as the number of agents, only a single overarching MDP, named Markov Game or Stochastic Game, is defined over the joint-action space corresponding to the group of agents. In a Markov Game, all agents can perceive the complete state of the world. In addition, in the general case, each agent has a distinct reward function, but for strictly cooperative tasks the entire team can be granted a single reward. The general formulation of a Markov Game is a tuple $\langle S, \mathcal{A}, \delta, R \rangle$, comprising:

- The set S of all states in the world,
- The collection $\mathcal{A}$ of action sets pertaining to each agent $A^1, \dots, A^k$,
- A probability distribution over transitions $\delta: S \times \mathcal{A} \times S \rightarrow [0,1]$ from a current state to a next state given the joint-action performed by the group of agents given the starting state,
- And a reward function per agent $R^i: S \times A^1, \dots, A^k \rightarrow \mathbb{R}$ that quantifies the immediate goodness of the joint-action selected in the current state.

Unlike IL, giving a single global reward to a team of agents is permitted by JAL for cooperative tasks; e.g. providing a soccer team a positive reward of +1 per scored goal. In fact, shaped rewards are no longer necessary for fostering coordination; hence, the limitations arising from them can now be dismissed. Instead, coordination emerges naturally as a result of learning optimal joint-actions.

Moreover, when a single reward is handed to a team in cooperative tasks, this effectively transforms a multi-agent problem into a single-agent problem; and by consequence, non-stationarity is completely avoided since now there only exists one agent interacting with the environment, perfectly self-aware of all actions being simultaneously executed that could alter the current state of the world.

On the downside, JAL presents a scalability issue, as the complexity of learning grows with the number of agents, i.e. the structure encoding the policy as well as the action space must increase. Notoriously, the joint-action space grows exponentially for action-value based methods such as DQN. Yet, it grows only linearly for parametrized policy methods such as DDPG.

On top of that, as the number of agents becomes very large, it is likely that individuals will not be able to perceive the same complete state of the world as required by JAL due to sensory, memory or communication difficulties. Thus, the problem would have to be reformulated as a more complex POMDP (Partially Observable Markov Decision Process).

Chapter 4
RoboCup Standard Platform League (SPL) Simulation

4.1. Overview

RoboCup [62] is an international scientific initiative with the goal to advance the state of the art of robotics and AI research by offering publicly appealing but motivating competitions, and by setting a challenging long term goal such as developing a team of robotic players able to defeat human soccer World Cup champions by 2050.

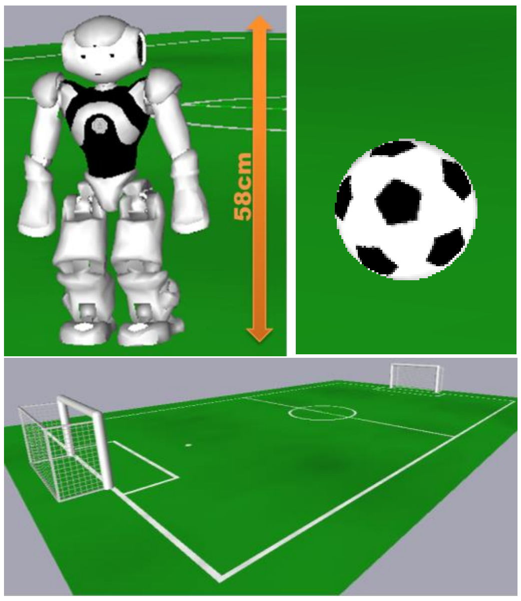
RoboCup officiates several leagues that go beyond soccer, and including leagues with real robots and leagues with simulated agents. Among these, the Standard Platform League (SPL) [63] allows for teams of NAO robots to compete against each other in 5 versus 5 soccer matches.

Figure 8. Standard Platform League.

The B-Human framework [64] is a physically grounded virtual environment developed in C++ that simulates complete soccer matches between teams of NAO robots, making it useful for testing solutions aim at the SPL. This framework recreates three main types of objects:

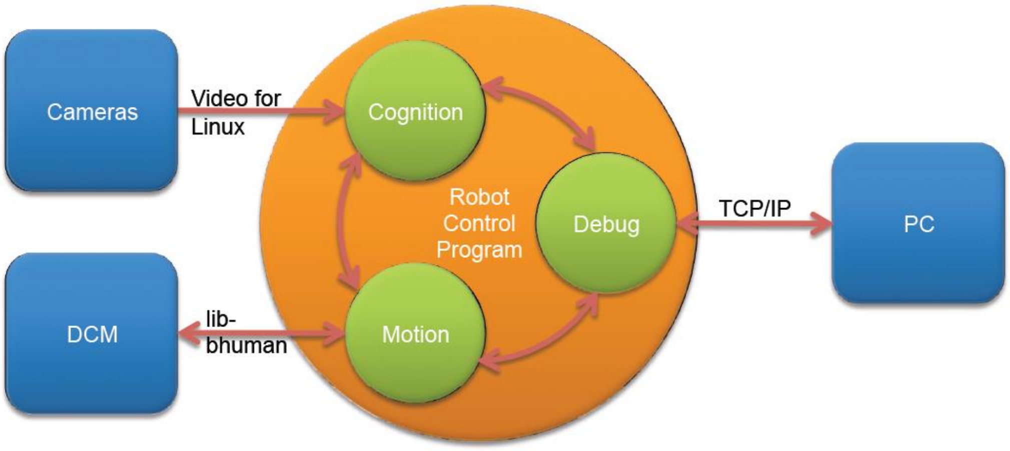
- Robots: NAO agents have been simulated with extreme detail in terms of dimensions, sensors and actuator. They are the only entities capable of actively influencing the environment.
- Ball: Diameter of about 25 cm. It is physically affected by friction and contact forces (collisions).
- Field: It is 9 meters long and 6 meters wide. The goal is around 2.2 meters wide.

Figure 9. Simulated NAO, ball and field.



Behind the scenes, the B-Human framework relies on the robotics simulator SimRobot [65] for handling the physics of the virtual world. Moreover, the controller of each virtual NAO agent is constructed as a complex hierarchy of processes and modules. At the top level there are three concurrent processes: Cognition, Motion and Debug. Cognition receives information from cameras and sensors, it processes this data, and generates high-level motion commands. Motion executes such commands by properly setting all joints of the robot at each timestep as it also receives sensor feedback. Whereas Debug gathers data from the other two processes and sends it to a host PC for supervision and analysis.

Figure 10. Processes of the B-Human framework.



Each process is composed of numerous modules, each handling the processing of data, control or management commands at a specific stage; for example, image acquisition, image transformation, feature extraction from images, object detection from features, and so on. Modules are defined by means of an interface, an implementation and an instantiating statement (see example in Box

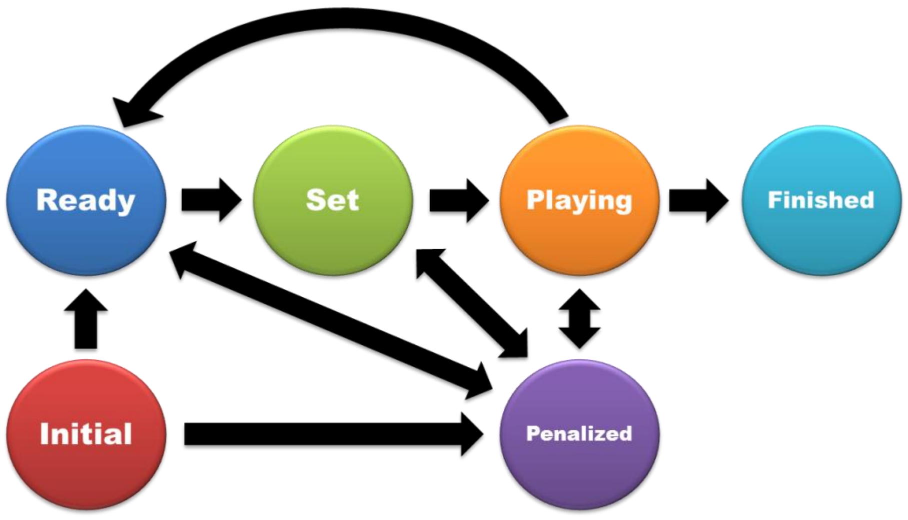
6). A module's interface must specify from which other modules it receives data and what output it produces. This definition will ultimately be used to manage the data flow of the controller during execution, establishing the order in which modules have to be computed. Each output of a module must correspond to a representation, which is a special structure in the framework that upon initialization allocates memory space in a globally shared blackboard. In the end, modules only directly interact with this blackboard retrieving data from it and inserting data to it.

Box 6. B-Human module template.

```
MODULE(Module1,
{,
        REQUIRES(Representation1),
        REQUIRES(Representation2),
        PROVIDES(Representation3),
});
class Module1 : public Module1Base
{
        void update(Representation3& Representation3)
        {
                //actual C++ implementation goes here
        }
}
MAKE_MODULE(Module1, modeling)
```

An important module, which is worth describing briefly, is the Game Controller module. It is in charge of managing the development of a soccer game, acting in the same way as a referee. It provides information regarding when a goal has been scored, when the ball has gone outside the lines, when the game has ended, and more. It also provides a flag that indicates the current state of the game, where possible game states are shown in the following figure:

Figure 11. Game states.



4.2. Agent’s Inputs

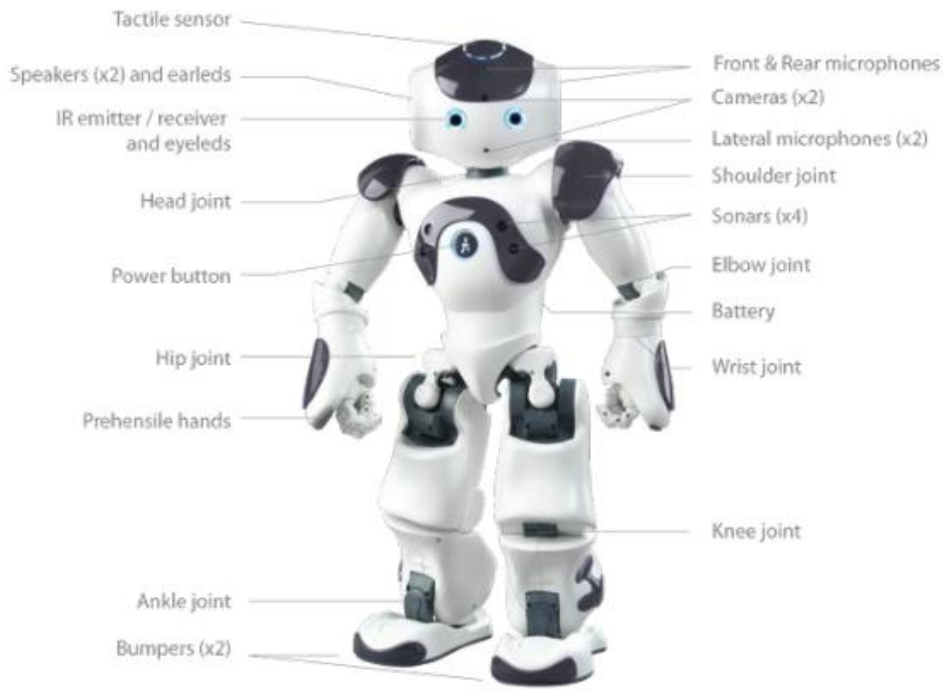
A simulated NAO robot receives raw camera and sensor data at an effective frequency of 60 Hz, and such information is processed by its Cognition modules ideally at the same frequency. The simulator also provides ground-truth information about the world at a similar rate. A RL agent could potentially observe any of these inputs, which are described next:

4.2.1. Raw Inputs

Most, if not all, sensors of a NAO robot have been simulated. The list below presents them:

- Upper camera: Located in the middle forehead of the robot. It provides a YUV422 image with resolution of 640x480 pixels.
- Lower camera: Located 4 cm. below the upper camera and tilted 39.7° with respect to it. It also provides a YUV422 image with resolution of 320x240 pixels.
- Inertial Measurement Unit: Consisting of three acceleration sensors in the range [-150, 150] m/s^2 and up to three gyroscopes in the range [-12, 12] rad/s.
- Eight (8) force sensors: Located in the feet of the NAO, 4 per foot. Within the framework, their output is between 0 and 5.
- Two (2) ultrasound sensors: Located in the chest.
- Joint angle sensors: One per joint, totaling 26. They have various joint limits.
- Load and temperature joint sensors: Likewise, there is one per joint of the robot.
- Several contact/touch sensors for manual control.

Figure 12. Arrangement of sensors in a NAO robot.



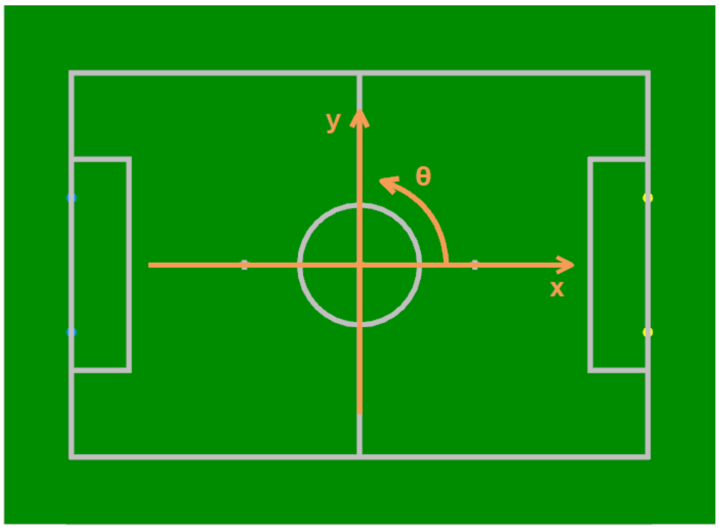
4.2.2. High-level Inputs

In the context of the Cognition process there are several modules that process the previous raw data into higher-level information such as: self-localization, localization uncertainty, detection of ball position/velocity, detection of teammates/opponents, detection of field lines/features, and more. However, for the most part, this information is also provided independently by the simulator as ground-truth without requiring any processing. Therefore, and since there is little difference between the two except for some extra noise, for the purposes of this thesis only the latter was considered in order to keep the simulation time low. In particular, the following high-level data was employed:

- Absolute pose of teammates: 3-dimensional vector per teammate, containing a robot's position on the 2D field and its orientation defined in reference to its torso.
- Absolute pose of opponents: Same as above, but for every opponent robot.
- Absolute ball position: 2-dimentional vector, containing the ball's position on the 2D field.
- Absolute ball velocity: 2-dimensional vector, containing the ball's velocity on the field. It is actually computed just as the difference between two consecutives positions.

These physical vectors are all measured with respect to the same reference frame shown below:

Figure 13. Arrangement of sensors in a NAO robot.



4.3. Agent's Outputs

At the most basic level, the framework can only send joint-angles to the NAO robots, aside from simulating the pressing of some touch buttons dedicated to manual control. Yet, modules within the Motion process, such as the walking engine and kick engine, can send a controlled sequence of joint-angles over several timesteps in order to effectively execute temporally extended actions or macro actions. Both these kind of outputs are presented next:

4.3.1. Raw Outputs

A NAO agent has 26 joints with various ranges of motion. These are synthesized in the following table:

Table 1. NAO joint ranges.

Joint	Min. angle	Max. angle
Head Yaw	-119.5°	119.5°
Head Pitch	-38.5°	29.5°
Left Shoulder Pitch	-119.5°	119.5°
Left Shoulder Roll	-18.0°	76.0°
Left Elbow Yaw	-119.5°	119.5°
Left Elbow Roll	-88.5°	-2.0°
Left Wrist Yaw	-104.5°	104.5°
Left Hand	0.0°	57.3°
Right Shoulder Pitch	-119.5°	119.5°
Right Shoulder Roll	-76.0°	18.0°
Right Elbow Yaw	-119.5°	119.5°
Right Elbow Roll	2.0°	88.5°
Right Wrist Yaw	-104.5°	104.5°
Right Hand	0.0°	57.3°
Left Hip Yaw Pitch	-65.6°	42.4°
Left Hip Roll	-21.7°	45.3°
Left Hip Pitch	-88.0°	27.7°
Left Knee Pitch	-5.3°	121.0°
Left Ankle Pitch	-68.2°	52.9°
Left Ankle Roll	-22.8°	44.1°
Right Hip Yaw Pitch	-65.6°	42.4°
Right Hip Roll	-45.93°	21.7°
Right Hip Pitch	-88.0°	27.7°
Right Knee Pitch	-5.9°	121.5°
Right Ankle Pitch	-68.0°	53.4°
Right Ankle Roll	-44.1°	22.8°

4.3.2. Macro Actions

4.3.2.1. Walking Engine

It allows omnidirectional walking. It relies on a feedback control that maintains the dynamical balance of the robot across its coronal and sagittal planes. This walking engine is based on three principles:

- The body swings left to right and back to shift away the weight from the swing foot, allowing it to be lifted above the ground and set back down again, as the torso simultaneously follows a pendulum-like motion.
- Feet sensors detect when the weight of the robot is shifted from the support foot to the swing foot. This event is used to set the transition between steps. Relying on a measurement makes the walk robust to disturbances.

- During each step, the readings of the pitch gyroscope are scaled and directly added to the pitch ankle joint of the support foot. As a result, the faster the torso turns forward, the more the support foot presses against this motion.

When executed, the walking engine takes full control of motion, i.e. no other set of joint-angles can be send to the NAO, for as long as the robot remains unbalanced. This prevents the robot from falling. On average, the walking engine locks motion for periods of time no longer than 0.5 seconds.

The framework already provides a couple of parametrized functions that internally makes use of the walking engine. Specifically, there is the command WalkAtRelativeSpeed(), the receives as input a 3-dimensional velocity vector, where the first component corresponds to a rotational velocity and the next two components to a Cartesian velocity, in both case relative to the current reference frame of the virtual NAO.

Given the above two considerations, reinforcement learning agents were allowed to call the command WalkAtRelativeSpeed() only once every 0.5 seconds as a permitted action, and as long as other macro actions are not currently in control of the agent.

4.3.2.2. Kick Engine

The kick engine receives, through a configuration file provided by the user, a list of dynamic points for various phases of motion, which are reference position points for each limb that outline a desired trajectory of motion for said limb. Internally, the engine employs a set of Bezier curves to generate smooth trajectories between dynamic points and between phases. Additionally, the overall motion of the robot is stabilized by the combination of center of mass balancing and a gyro feedback-based closed-loop PID controller.

Similarly to the walking engine, the kick engine takes full control of motion upon execution as well. However, its duration is longer as the robot goes through several phases of preparation, actual kick and repositioning to a balanced pose. Furthermore, in order to perform to consecutive kicks, an extra idle time after repositioning must be considered where the agent just stands in the same spot. The SPQR RoboCup team [66] has developed several configuration files for executing different kicks, some are fairly fast and others are slower. Fast kicks are usually unstable; the NAO falls down constantly especially when it does not touch the ball. Slow kicks have an overall duration of approximately 5 seconds, including at least 1 second of idle time.

The kick engine has the additional functionality that a symmetrical kick (using the other leg) can be executed simply by switching a single Boolean variable. For simplicity, this variable was fixed, such that the agent always kicks with the same leg. After taking all this into consideration, RL agents were permitted to call a kick command without parametrization only once every 5 seconds.

4.4. Behavior Control

The Behavior Control is a large module located inside the Cognition process. Essentially, it is the decision-making apparatus of the framework. It receives raw, processed or ground truth inputs coming from other modules; then, based on hand-coded rules and without realizing much computation on its own, it executes transitions between high-level motion commands. Clearly, the policy of a RL agent must eventually be encoded within this module.

The Behavior Control module is coded using the C-based Agent Behavior Specification Language (CABSL) [67]. This language enables modeling an agent's behavior as a hierarchy of state machines, renamed as options, where each encodes a single skill of the agent. An option graph synthesizes the underlying hierarchy among options. Options are composed of states, including the initial state that must always be present. Every state has its own associated transitions and actions. Ultimately, CABSL works by transitioning between options according to explicit calls and, inside options, by transitioning between states when their conditions are met, while executing the corresponding action as soon as a state is reached. The following is a template of an option:

Box 7. B-Human option template.

```
option(declare_name_of_option)
{
        initial_state(declare_name_of_state)
        {
                transition
                {
                        if(condition)
                                goto someState
                        else if
                                goto someOtherState
                }
                action
                {
                        //execute function or another option
                }
        }
        state(declare_some_state)
        {
                transition
                {
                        //define conditions and next states
                }
                action
                {
                        //execute function or another option
                }
        }
        //declare as many more states as needed
}
```

4.5. Neural Controller

The B-Human framework does not provide a neural controller; hence, in order to apply reinforcement learning it was imperative to integrate one within this C++ environment. This was accomplished by using Tensorflow C++, such that it is now possible to load and run a deep network directly within the framework, given a previously saved model of said network. The steps enabling this integration are commented next:

1) First of all, it is necessary to design the architecture of the network and to set its weights outside the framework. This can be done in Tensorflow C++ or in Tensorflow Python. Notice that this network will be used only for predictions; thus, it must only encode the current policy of the RL agent; or in other words, stand-alone critic or target networks do not need to be integrated.
2) Next, a frozen graph must be created that will contain our static network. This was done using Bazel's freeze_graph() tool, which requires that the network's architecture be saved as a graphdef object and the network's weights be saved in a checkpoint.
3) By this point, the frozen graph can already be used from within the framework. Therefore, at initialization of the simulation this graph is loaded only once (see implementation in Box 8) producing a Session object whose method Session.run() can be fed with desired inputs at any time to make a prediction.
4) Specifically, the graph was arbitrarily loaded in the Init method of the Cognition process; other Init methods could have been utilized, but it is important that the Session object is defined as global, so that it could be employed by other modules in the framework.

Box 8. Tensorflow C++ session initialization.

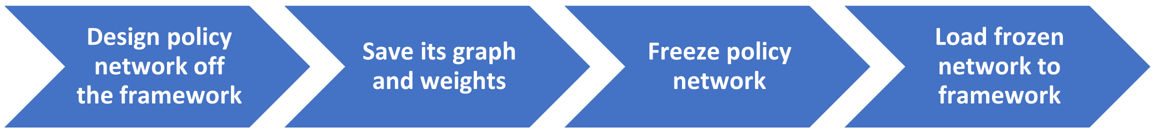
```
extern std::unique_ptr<tensorflow::Session> initSession;

Status LoadGraph(const string& graph_file_name,
                 std::unique_ptr<tensorflow::Session>* session)
{
       tensorflow::GraphDef graph_def;
       Status load_graph_status = ReadBinaryProto(
             tensorflow::Env::Default(), graph_file_name, &graph_def);
       session-
>reset(tensorflow::NewSession(tensorflow::SessionOptions()));
       Status session_create_status = (*session)->Create(graph_def);
       tensorflow::GraphDef graph_def;
}

void Cognition::init()
{
       string graph_path = "insert path to graph";
       Status load_graph_status = LoadGraph(graph_path, &initSession);
       ...
}
```

The previous integration process can be visualized in the following figure:

Figure 14. Integration process of the neural controller.


Design policy network off the framework
Save its graph and weights
Freeze policy network
Load frozen network to framework

As additional information, it is worth mentioning that the original Mare file from the B-Human framework was still use to build the executables that would run deep neural networks. Nevertheless, it had to be edited to include a handful of libraries required by Tensorflow C++, which were gathered from Bazel.

4.6. Reinforcement Learning Modules

From the previous section, there is now a Session object in the framework that encodes the current policies of one or more agents, and naturally, it can be used to interact with the environment. However, to implement RL techniques, the framework still lacks an action selection procedure that follows these policies but also enforces exploratory behavior, a reward function that informs agents about the goodness of their actions, and means to gather data from rewards and next states.

Because of this, two additional modules were added to the framework, consisting of a Selection module and a Reward module. Their role is to carry out the interaction agent/environment, the computation of rewards and the collection of data to be later used by RL algorithms, but they still do not perform any kind of learning.

The primary objective of the Selection module is decision-making; thus, it could potentially replace the Behavior Control module. However, such implementation would require major and not always wanted modifications of the framework. In view of this, this extra module was included in the Cognition process instead, but it is closely intertwined with the Behavior Control module. The complete set of duties of the Selection module is:

- First and foremost, when a game is in a "Playing" state, it applies an ϵ-greedy strategy as explained in section 3.2.5, such that either, it generates an action vector according to the current policy, i.e. by running the currently frozen deep network, or it generates a noisy action vector. This action vector is placed in the blackboard, from which it can be read by the Behavior Control module.
- It controls the duration of macro actions. It performs action selection only after the predefined time limit for the last executed macro action has expired. Remember that after each walking or kicking command agents must wait 0.5 and 5 seconds, respectively.

- Each time an action is selected, it sets to true a corresponding Boolean flag (firstWalk or firstKick), which informs to the entire framework when this new action starts.
- Each time an action vector is produced, it saves that vector along with the current state to corresponding globally defined memory buffers.

On the other hand, the Reward has the following responsibilities:

- It uses ground-truth information provided by the simulator to compute rewards.
- In case of shaped rewards that are assigned to the agents for every action, it interacts with the Selection module in order to keep track of when actions start and end.
- It saves all computed rewards to a globally defined memory buffer.

Additionally, the Game Controller was also modified, such that whenever it stipulates that a match has finished, it proceeds to write the information contained in the state, action and reward memory buffers into three separate text files.

4.7. Agent's behavior

The Behavior Control module was written in a manner that accommodates for the decisions taken by the Selection module. Particularly, it internalizes the flags handed by the latter within its transitioning conditions, and furthermore, it passes chunks of the current action vector as arguments to high-level motion commands that must be executed. For instance, the walking command is given a 3-vector representing a relative velocity direction. The final implementation tailored for a single agent is presented in the next box:

Box 9. Modified Behavior Control module.

```
//theNNPercept is a representation in the blackboard containing flags
//and neural outputs computed by the Selection module
option(Striker)
{
   initial_state(start)
   {
      transition
      {
         if(theNNPercept.firstWalk)
            goto walking;
         else if(theNNPercept.firstKick)
            goto kicking;
      }
   }
   state(walking)
   {
      transition
      {
         if(theNNPercept.firstWalk)
            goto walking;
         else if(theNNPercept.firstKick)
            goto kicking;
```

```
    }
    action
    {
      WalkAtRelativeSpeed(Pose2f(theNNPercept.neuralOutput[2]*2.f-
        1.f, theNNPercept.neuralOutput[3]*2.f-1.f,
        theNNPercept.neuralOutput[4]*2.f-1.f));
    }
  }
  state(kicking)
  {
    transition
    {
      if(state_time > 3800)
        goto stand_1s;
    }
    action
    {
      Kicks(true);
    }
  }
  state(stand_1s)
  {
    transition
    {
      if(theNNPercept.firstWalk)
        goto walking;
      else if(theNNPercept.firstKick)
        goto kicking;
    }
    action
    {
      Stand();
    }
  }
}
```

4.8. Learning with the Simulator

Lastly, the learning procedure was ultimately carried out outside the B-Human framework, in Python with Tensorflow. It is in this environment that all deep learning structures are kept and regularly updated, including critic and target networks, and obviously, the network encoding the current policy that must be transformed into a frozen graph to be used by the C++ simulator.

In fact, after each update of said network and before calling the simulator, a new frozen graph is programmatically created given the current weights of this model (overwriting the previous one), such that the simulator always operates with the latest policy available.

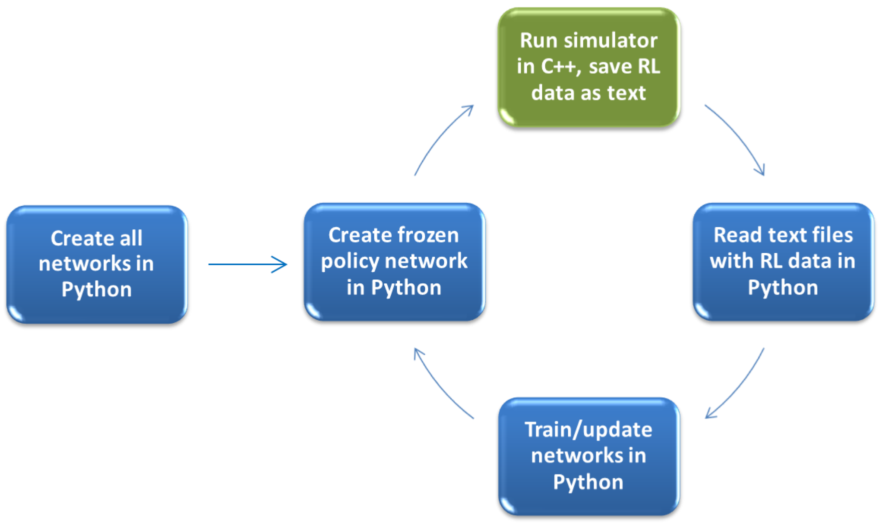
Since the simulator runs in a different environment, a couple of special commands (see box below) had to be applied in order to open and close the executable. It was also necessary to consider a sleeping time for Python, carefully synchronized to close the simulator only after a desired number of episodes has been completed. The final output of the simulator is three text files with collected RL data.

Box 10. Interface between Python and C++.

```
//execution in Python
p=subprocess.Popen([“path_to_C++_executable”, “path_to_ros2_file”])
time.sleep(//duration of episodes)
//execution in C++ while Python sleeps
p.terminate()
//resume execution in Python after sleep
```

At this point, these text files can be easily read by Python and used for updating the existing neural networks according the any given RL algorithm. The overall learning cycle, considering the communication between programming environments, is summarized in the next figure:

Figure 15. Learning cycle.


Run simulator in C++, save RL data as text
Create all networks in Python
Create frozen policy network in Python
Read text files with RL data in Python
Train/update networks in Python

Chapter 5
Technical Approach

5.1. Task Specification: Learning Free Kicks

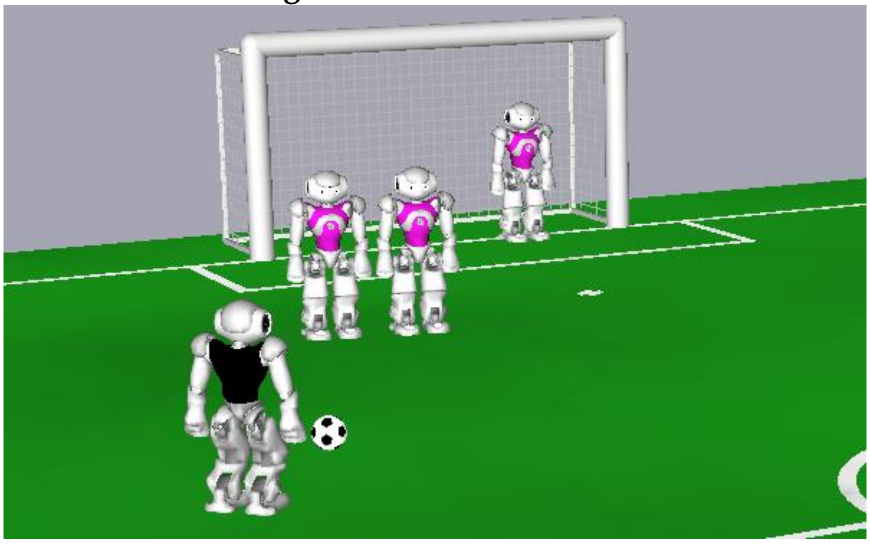
The goal of this thesis is to test current deep RL approaches for cooperative multi-agent domains. Hence, an intrinsically cooperative domain such as soccer was chosen for carrying out experiments. However, learning a good team policy valid at every situation in a complete soccer match is still out of reach for state-of-the-art RL techniques. For this reason, related works have evaluated their results over reduced tasks, such as: Keepaway Soccer [68], Half-Field Offense [69] or Free-Kicks [42].

Likewise, this study considers as test bed the specific task of learning team behaviors during an offensive free-kick situation. In RoboCup, as in real soccer, a free-kick is granted to a team due to a fault committed by the opposing team. The team awarded with the free-kick has sole passive possession of the ball, which ends when at least one player of the team makes contact with the ball or when a time limit expires (usually 30 seconds in RoboCup). During this idle period of time, the players of the opposing team are forbidden to remain less than 0.75 meters away from the ball (except for the goalkeeper as long as it is located inside the goal box).

While respecting the above rules, specific details regarding the offensive free-kick task implemented for this work are the following:

- The task involves two teams, each composed of two players.
- The team awarded with the free-kick is denoted as the offensive team, while the opposing team is denoted as the defending team.
- The offensive team has two attackers, while the defending team has a defender and a goalkeeper.
- At the beginning of every match, the ball and all four agents are positioned somewhere inside the half-field belonging to the defensive team. In particular, the defensive players maintain a proper distance with respect to the ball.
- During play, the defensive players always follow a fixed policy according to its role.
- Conversely, both attackers are allowed to learn by means of Deep RL techniques.
- The end goal of the task is for the attackers to score a goal against the opposing goalkeeper before the episode terminates (e.g. due to a time limit).

Figure 16. Free kick task.



5.2. Experimental Setup

Experiments were realized over the previously described offensive free-kick task. They involved testing the applicability and performance of deep-learning implementations of two Reinforcement Learning alternatives for multi-agent domains, namely the Independent Learners (IL) and Joint-Action Learners (JAL) approaches presented in section 3.4.

The full extent of the experimental setup is provided in the next sections:

5.2.1. World Settings

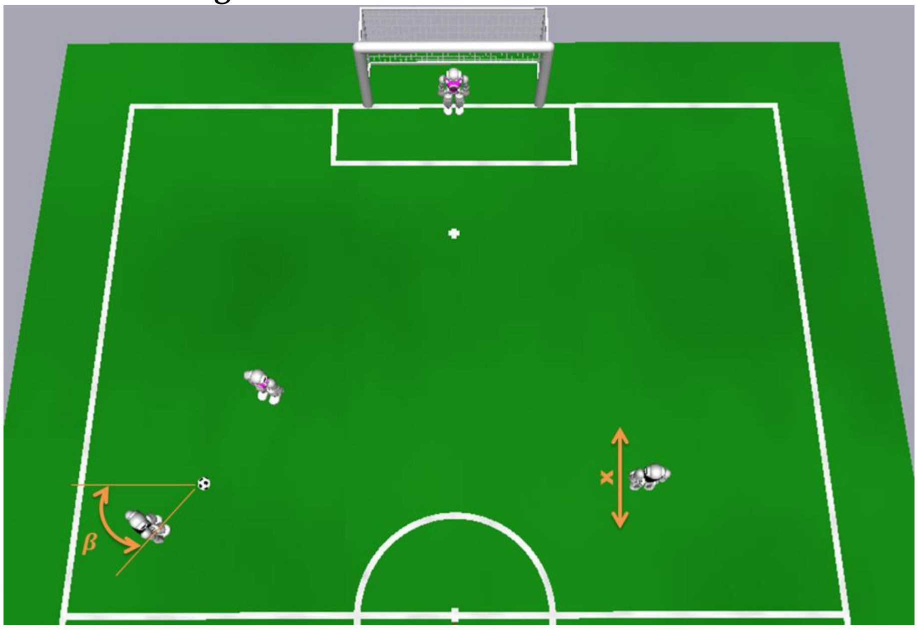
The offensive free-kick task is played within the B-Human framework described in Chapter 3, using players, ball and field with standard characteristics according to RoboCup SPL.

At the beginning of each episode, the ball is placed at position $[1.0, 2.0]$ meters. Furthermore, the defender is positioned exactly at a distance of 0.75 meters away from the ball and in between the ball and the center of the goal line; hence, it blocks a direct kick to goal. In addition, the goalkeeper is placed at the center of the goal line.

Moreover, a first attacker is initially positioned at a distance of 0.3 meters from the ball, along a direction that varies uniformly at random across episodes, and which depends on an angular variable β in the range $[0, \pi/4]$ rad, measured with respect to the world's y-axis. On the other hand, the second attacker is originally placed about the opposing sideline to its teammate; its initial position on the y-axis is fixed to -2.5 meters, whereas its position on the x-axis varies uniformly at random within $[1.0, 2.0]$ meters at each initialization.

The following figure shows a randomly generated initial state for the task:

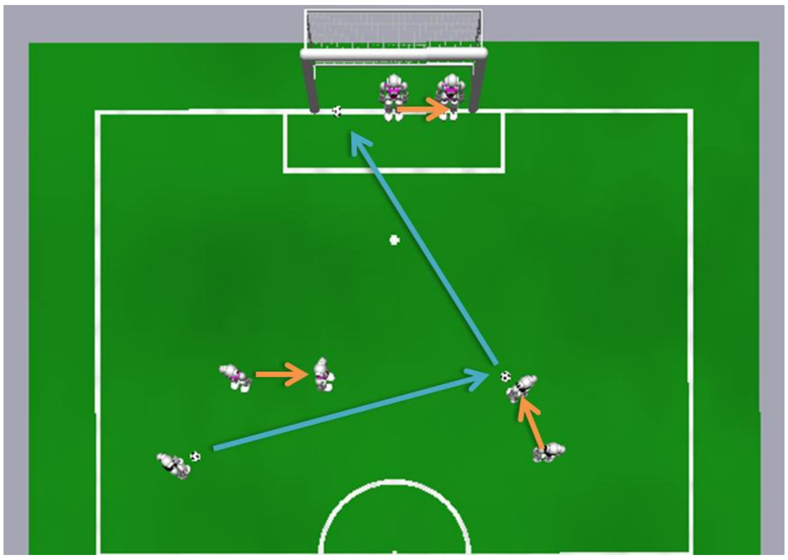
Figure 17. Random initial world state.



During game play, the attackers make decisions based on its action selection strategy and on its current parametrized policy. Meanwhile, the defensive players follow role-specific simple policies. For instance, the defender's policy considers two conditions: 1) before an agent of the offensive team touches the ball, it stands in-place blocking a possible direct kick; and 2) after an agent of the offensive team touches the ball, it approaches the ball as fast as it can by moving in the direction defined by the straight line connecting its current location and the ball's current location. In turn, the goalkeeper is only allowed to move from post to post along its goal line. It will always move towards the intersection of the goal line and the ball's velocity vector. However, if the ball's velocity vector is zero or points away from the goal line, the goalkeeper stands still; and if the intersection lies beyond a post, the goalkeeper moves in that direction but stops when it reaches said post.

Given the policies of the defending team, clearly a good scoring strategy is to perform a cross kick (see figure below). That is, the attacker initially closer to the ball should pass it across the width of the field to a position in front of the second attacker, while also taking care that the ball stops closer to its teammate than to the defender. At this point, the goalkeeper would have moved towards one goal post; hence, the second attacker should now just kick to goal, crossing the ball again towards the spot left open by the goalkeeper near the opposite goal post.

Figure 18. Cross-kick solution.



An episode is terminated when one of the following conditions is met:

- The offensive team has scored a goal. (successful episode)
- The time limit of 20 seconds has expired. This time was determined empirically to be sufficient for an offensive team to score. (failed episode)
- One player of the defending team has made contact with the ball. (failed episode)
- The ball has left the field. (failed episode)

5.2.2. MDP Formulation for IL

Specifically, in lieu of satisfactory results achieved by previous works [1, 39], a Homogeneous Learners strategy (see section 3.4.1.2) was enforced for IL; hence, effectively only one MDP was needed.

From the point of view of each agent, its associated MDP comprises an 18-dimensional state vector in continuous and discrete spaces, consisting of the following information:

- The own agent's absolute pose, $[x^A, y^A, \theta^A] \in \mathbb{R}^3$. This is a 3-dimensional real vector encoding the current Cartesian absolute position $\boldsymbol{p}^A = (x^A, y^A)$ and 2D orientation θ^A of the agent over the field with respect to the world's reference frame.
- The absolute pose of its teammate, $[x^T, y^T, \theta^T] \in \mathbb{R}^3$. Likewise, this is a 3-dimensional real vector encoding the same information as above, but relative to the teammate.
- The ball's absolute position, $\boldsymbol{p}^B = [x^B, y^B] \in \mathbb{R}^2$. This is 2-dimensional real vector denoting the current location of the ball over the field with respect to the world's reference frame.
- The ball's absolute velocity, $\left[v_x^B, v_y^B\right] \in \mathbb{R}^2$. This is 2-dimensional real vector denoting the current velocity of the ball over the field with respect to the world's reference frame.

- The defender's absolute position, $[x^D, y^D] \in \mathbb{R}^2$. This is 2-dimensional real vector indicating the current Cartesian position of the defender with respect to the world's reference frame.
- The goalkeeper's spot over the goal line, $y^G \in \mathbb{R}$. This is a real scalar indicating the current location of the goalkeeper on the y-axis of the world's reference frame.
- A 5-bit binary timestamp, $[t^4, t^3, t^2, t^1, t^0] \in \{0,1\}^5$, representing the current timestep of the episode. The episode starts with a timestep of zero, which is incremented by one unit of time after every 0.5 seconds of real-time play. Such timestamp is important because it provides a way for the agent to differentiate between normal states and final states, in case an episode ends due to the time limit. The other three types of final states (e.g. ball left field) are easily discriminated by the agent from the previous state components, without the need of additional information.

Each agent is permitted to execute only one of two macro actions every time it interacts with the environment, namely: walk or kick (refer to section 4.3.2). It must be noticed that the walk action takes exactly one step from the point of view of RL, i.e. it lasts 0.5 seconds; while the kick action takes ten steps, i.e its execution time is 5 seconds. These two actions are represented together in the MDP by means of a 5-dimensional action vector within a continuous space, $a \in \mathbb{R}^5$, whose components are:

- Two components correspond to walk and kick selectors, respectively. These lie in the range $[0.0, 1.0]$, and as their names suggest, they are used for selecting which action the agent will perform at the current decision point. If the walk selector has a higher value than the kick selector, then the agent will obviously walk; otherwise, the agent will kick.
- Three components represent the arguments taken by the function WalkAtRelativeSpeed(), which indicate the relative Cartesian and rotational velocities with which the agent will walk. They can take values between $[-1.0, 1.0]$, where opposite signs denote opposite directions. These components are only meaningful when the selected action is indeed walk.

Both agents are given rewards based on the same function, which is shown next:

$$R = D_{agent}^{ball} + \max\left(D_{ball}^{mate}, D_{ball}^{goal}\right) + G$$

Where:

$$D_{agent}^{ball} = d(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{old}^{A}) - d(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{new}^{A})$$

$$D_{ball}^{mate} = \begin{cases} if\ agent\ pushed\ the\ ball, & d(\boldsymbol{p}_{old}^{B}, \boldsymbol{p}_{new}^{T}) - d(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{new}^{T}) \\ else, & 0 \end{cases}$$

$$D_{ball}^{goal} = \begin{cases} if\ agent\ pushed\ the\ ball, & d_{goal}(\boldsymbol{p}_{old}^{B}) - d_{goal}(\boldsymbol{p}_{new}^{B}) \\ else, & 0 \end{cases}$$

$$G = \begin{cases} if\ goal, & +20 \\ else, & 0 \end{cases}$$

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\left(x_p - x_q\right)^2 + \left(y_p - y_q\right)^2}$$

$$d_{goal}(\boldsymbol{p}^{B}) = \begin{cases} if\ y^{B} \geq y_{left}^{post}, & d\left(\boldsymbol{p}^{B}, \boldsymbol{p}_{left}^{post}\right) \\ if\ y^{B} \leq y_{right}^{post}, & d\left(\boldsymbol{p}^{B}, \boldsymbol{p}_{right}^{post}\right) \\ else, & \left|x^{B} - x^{post}\right| \end{cases}$$

This shaped reward function integrates the following four different reward terms:

- D_{agent}^{ball}: This term computes the distances from the agent's previous and current positions to the current location of the ball. It rewards an agent for approaching the ball.
- D_{ball}^{mate}: This term computes the distances from the ball's previous and current positions to the current location of the agent's teammate. It is only granted to the agent if it directly pushes the ball. This term promotes cooperation, since it incentivizes an agent for passing the ball to a teammate.
- D_{ball}^{goal}: This term computes the shortest distances from the ball's previous and current positions to the fixed location of the goal line segment. Similarly, it is only granted to the agent if it directly pushes the ball. This term rewards an agent for hitting the ball to goal.
- G: This term assigns a large positive reward to all agents of the offensive team when a goal is scored, while it assigns them zero rewards when no goal is scored.

Shaped rewards were considered because, without extra mechanisms for learning from sparse payoffs it is highly unlikely that a random exploration leads the team of agents to score a single goal.

5.2.3. Markov Game Formulation for JAL

Since for the given task every agent is able to perceive the full world state, the state spaces for an MDP and for a Markov Game are identical to each other. Therefore, the world state of the Markov Game associated to the offensive free-kick task will be the exact same 18-dimensional vector existing in continuous and discrete spaces described in the precedent section.

Because a parametrized policy will be used to assess the actions of the agents, the joint-action vector corresponding to this Markov Game can be constructed simply by concatenating the individual action vectors associated to each member of the offensive team, which were presented in the previous section. As a result, such joint-action vector exists in a 10-dimensional continuous space,

$a_1 | a_2 \in \mathbb{R}^{10}$, which includes: 2 sets of walk/kick selectors and 2 sets of 3-vector relative walking velocities.

A shaped reward function was also considered for JAL, for the same reason as IL. Specifically, the offensive team as a whole is rewarded according to the following function:

$$R = \sum_i D_{agent_i}^{ball} + \max\left(\left\{D_{ball}^{agent_i}, \forall i\right\}, D_{ball}^{goal}\right) + G$$

Where:

$$D_{agent_i}^{ball} = d\left(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{old}^{A_i}\right) - d\left(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{new}^{A_i}\right)$$

$$D_{ball}^{agent_i} = d\left(\boldsymbol{p}_{old}^{B}, \boldsymbol{p}_{new}^{A_i}\right) - d\left(\boldsymbol{p}_{new}^{B}, \boldsymbol{p}_{new}^{A_i}\right)$$

$$D_{ball}^{goal} = d_{goal}\left(\boldsymbol{p}_{old}^{B}\right) - d_{goal}\left(\boldsymbol{p}_{new}^{B}\right)$$

$$G = \begin{cases} if\ goal, & +20 \\ else, & 0 \end{cases}$$

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\left(x_p - x_q\right)^2 + \left(y_p - y_q\right)^2}$$

$$d_{goal}\left(\boldsymbol{p}^{B}\right) = \begin{cases} if\ y^B \geq y_{left}^{post}, & d\left(\boldsymbol{p}^{B}, \boldsymbol{p}_{left}^{post}\right) \\ if\ y^B \leq y_{right}^{post}, & d\left(\boldsymbol{p}^{B}, \boldsymbol{p}_{right}^{post}\right) \\ else, & \left|x^B - x^{post}\right| \end{cases}$$

This shaped reward function is fairly similar to the one employed by IL, but it has a few slight but meaningful differences that are worth commenting:

- $\sum_i D_{agent_i}^{ball}$: This term rewards the offensive team for every attacker that approaches the ball.
- $\left\{D_{ball}^{agent_i}, \forall i\right\}$: This term rewards the team whenever the ball is passed to anyone of its members.
- D_{ball}^{goal}: This term rewards the offensive team if any agent hits the ball to goal.
- G: This term assigns a large positive reward to the team when a goal is scored and assigns zero when the episode terminates without a goal.

5.2.4. Deep RL Settings

The DDPG algorithm (see section 3.3.4) was used to enforce learning within the IL and JAL approaches. The implementation of this algorithm is detailed below:

5.2.4.1. Network Architecture

DDPG maintains four separate deep neural networks during learning, namely critic, actor, target critic and target actor networks. Since the offensive free-kick task does not require a memory of past states to be solved, and because the world state of its associated MDP or Markov Game is low dimensional, a sequential network architecture has been considered for all four networks. The architectures of critic and actor are similar to each other, whereas target networks are simply replicas of their references.

The critic network is built with three hidden layers of 64, 48 and 32 neurons, plus one output layer with a single neuron that predicts a single action-value. The input to this network is a vector, of size 23 when IL is considered (18D from the world state, plus 5D from the action vector of a single agent), and of size 28 in the case of JAL (18D from the world state, plus 10D from the joint-action vector).

In turn, the actor network is also composed of three hidden layers of 64, 48 and 32 neurons, but its output layer possess as many neurons as the cardinality of the action or joint-action vectors associated with IL or JAL approaches, respectively. In addition, in all cases, the actor receives the world state as input, i.e. an 18-dimensional vector.

Artificial neurons in all hidden layers have a ReLU activation function, which is known to be reliable in deep learning implementations. The output neuron of the critic is also endowed with a ReLu activation function. Meanwhile, every neuron in the actor's output layer has a logistic activation function that constrains the output values to the range [0.0, 1.0]. Afterwards, output values representing relative velocities associated with walking are normalized between [−1.0,1.0].

The figures below illustrate the specific architectures implemented within IL or JAL approaches:

Figure 19. Deep neural architecture for IL.

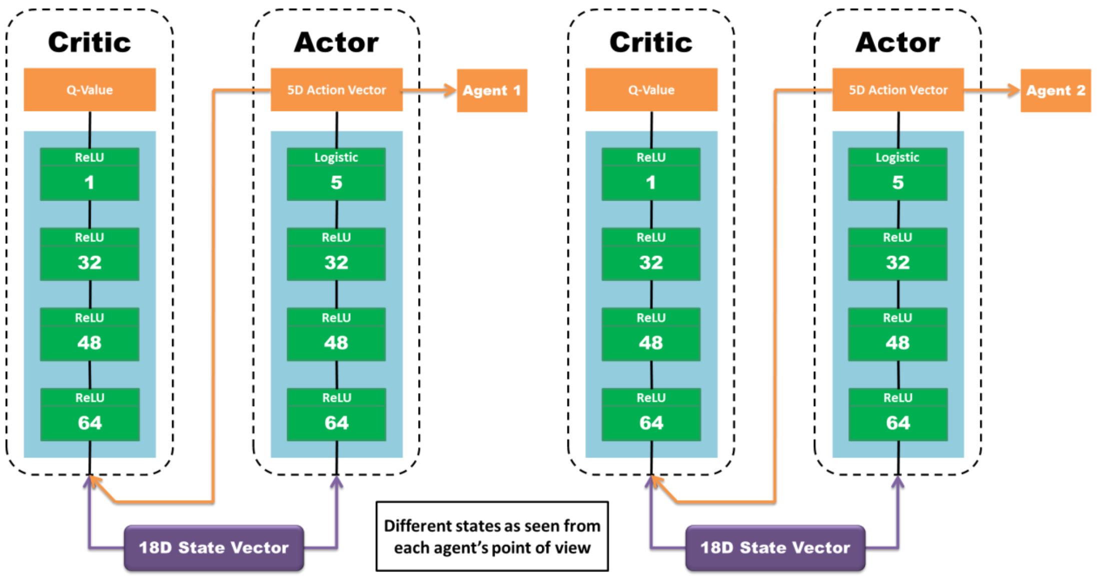
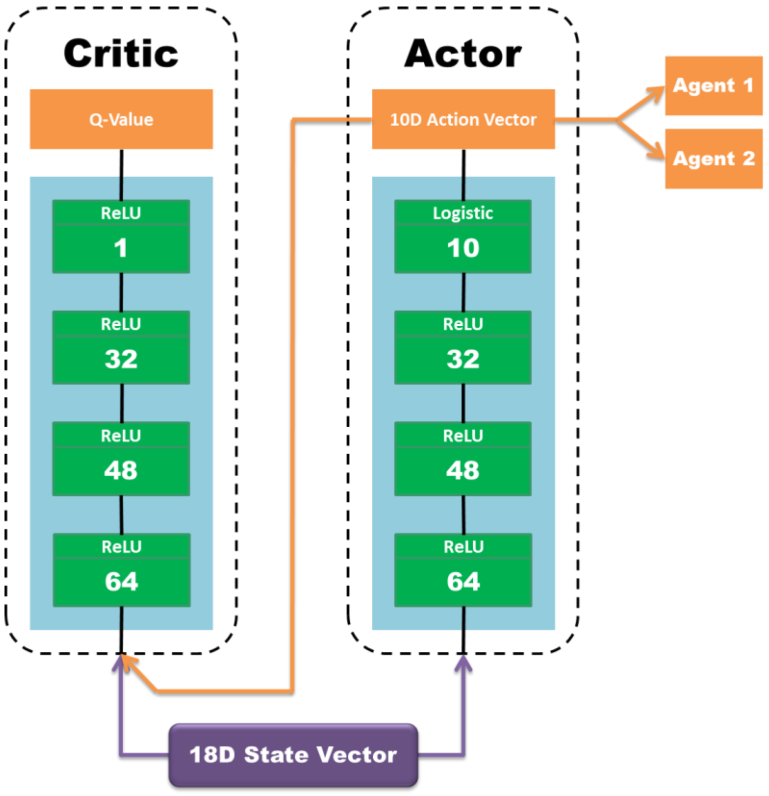


Figure 20. Deep neural architecture for JAL.



5.2.4.2. Network Training

The critic network θ^Q is trained through Q-value targets (Q_{target}) computed according to the DDPG algorithm. Hence, training requires a loss function that compares targets against predictions (Q_{pred}), which in this setup is defined to be the mean square error (MSE), evaluated as follows:

$$\mathcal{L}(\theta^Q) = \frac{1}{2}\left[Q_{target} - Q_{pred}(s,a|\theta^Q)\right]^2$$

Furthermore, the actor network θ^π is trained by means of gradients coming from the critic and passing through the actor. Thus, no loss function is needed in this case, as the actor will be updated according to the following improvement direction:

$$\nabla_a Q(s,a|\theta^Q)\nabla_{\theta^\pi}\pi(s|\theta^\pi)$$

In both cases, training is conducted with Adam optimizer and learning rate of $\alpha^Q = \alpha^\pi = 0.001$. Adam is commonly employed in deep learning, especially alongside sequential architectures and MSE as loss function.

5.2.4.3. Reinforcement Learning Specifications

Below is presented the pseudocode for the complete DDPG algorithm, including improvements (see section 3.3.5), as implemented in this work:

Box 11. DDPG algorithm.

```
1:  Set $L, I, m, n, E, \epsilon, \gamma, \alpha^Q, \alpha^\pi, \tau$
2:  Initialize critic parameters $\theta^Q$ and network $Q(s, a|\theta^Q)$
3:  Initialize actor parameters $\theta^\pi$ and network $\pi(s|\theta^\pi)$
4:  Set target networks' parameters equal to their references, $\theta'^Q = \theta^Q, \theta'^\pi = \theta^\pi$
5:  Initialize empty experience replay buffer $M = []$ with memory size $m$
6:  For $l = 1, \dots, L$ learning steps do:
7:          Interact with the environment for $i = 1, \dots, I$ episodes:
8:                  Start episode $i$ in state $S_0^i$
9:                  For each timestep $t$ until termination condition do:
10:                         Observe current state $S_t^i$
11:                         Select action $A_t^i$ according to $\epsilon$-greedy and current actor $\pi(S_t^i|\theta_l^\pi)$
12:                         Observe next state $S_{t+1}^i$ and immediate reward $R_{t+1}^i$
13:                         Collect information $\left[S_t^i, A_t^i, S_{t+1}^i, R_{t+1}^i\right]$
14:         If $M$ is full, remove elements at random
15:         Append collected data from last $I$ episodes to $M$
16:         For $e = 1, \dots, E$ training epochs do:
17:                 Draw $N$ elements (batch size) randomly from $M$
18:                 For $n = 1, \dots, N$ do:
19:                         Predict next action with target actor, $A_{t+1}^n = \pi(S_t^n|\theta_l'^\pi)$
20:                         Predict Q-value of next state with target critic, $Q(S_{t+1}^n, A_{t+1}^n|\theta_l'^Q)$
21:                         Compute training targets for the critic, $R_{t+1}^n + \gamma Q(S_{t+1}^n, A_{t+1}^n|\theta_l'^Q)$
22:                 Train critic through targets:
23:                 $\mathcal{L}(\theta_l^Q) = \frac{1}{N}\sum_{n=1}^{N}\left(\left[R_{t+1}^n + \gamma Q(S_{t+1}^n, A_{t+1}^n|\theta_l'^Q)\right] - Q(S_t^n, A_t^n|\theta_l^Q)\right)^2$
24:                         $\theta_{l+1}^Q = \theta_l^Q - \alpha^Q \nabla_{\theta^Q}\mathcal{L}(\theta_l^Q)$
25:                 Compute gradients from critic, $\nabla_a Q(S_t^n, A_t^n = a|\theta_{l+1}^Q), \forall n = 1, \dots, N$
26:                 Train actor through gradients:
27:                     $\theta_{l+1}^\pi = \theta_l^\pi - \alpha^\pi \frac{1}{N}\sum_{n=1}^{N}\nabla_a Q(S_t^n, A_t^n|\theta_{l+1}^Q)\nabla_{\theta^\pi}\pi(S_t^n|\theta_l^\pi)$
28:                 Update target critic, $\theta_{l+1}'^Q = \tau\theta_{l+1}^Q + (1-\tau)\theta_l'^Q$
29:                 Update target actor, $\theta_{l+1}'^\pi = \tau\theta_{l+1}^\pi + (1-\tau)\theta_l'^\pi$
```

Notice that for the Independent Learners approach, the interaction with the environment (lines 7-13) is carried out independently but simultaneously for every learning agent, and all data is gathered in the same experience replay buffer.

The next table presents the values set for the hyper-parameters of the DDPG algorithm:

Table 2. List of DDPG hyper-parameters.

Parameter	Symbol	Value
Number of learning steps	L	400000
Consecutive interaction episodes	I	1
Size of replay buffer	m	100000
Training batch size	n	4000
Training epochs	E	1
$\boldsymbol{\epsilon}$-greedy control parameter	ϵ	0.5
Discount rate	γ	0.9
Learning rate critic network	α^Q	0.001
Learning rate actor network	α^π	0.001
Update rate target networks	τ	0.01

Chapter 6
Results

IL and JAL were applied a total 10 different times over the offensive free-kick task. During each run, a validation step was realized after every 200 learning steps. Such validation consisted in setting the ϵ parameter to zero (dismissing any exploratory all together) and testing the performance of the current policy (actor network) by its own on 50 random episodes, while said policy remains fixed, i.e. training is paused during the entire validation period.

Several statistics were gathered throughout validation, such as: mean and standard deviation of the total reward per episode, maximum and minimum total reward amongst episodes and percentage of score goals within these 50 episodes. These results are presented next:

6.1. Independent Learners

Out of ten runs, only two times IL achieved successful team strategies able to always solve the task at hand. In the remaining 8 times, IL scored zero goals at every validation step; and in six out these eight times, IL also never scored a goal during training. In the other two, IL did score goals during learning steps, thanks to timely exploratory actions.

The figures below show the progress in terms of rewards and percentage of goals displayed by Independent Learners during one successful run (results for the other successful experiment were similar). It can be seen that IL converged to a highly satisfactory policy after about 200000 training episodes. Before that, learning remained stuck in a bad local minimum for approximately 150000 episodes, during which no goals were scored in validation. In this local minimum, the attacker further away from the ball, only approaches the ball but never kicks it; likely because it is seldom close enough to the ball to perform a kick with positive effects.

Figure 21. Average total reward per episode reported by IL.

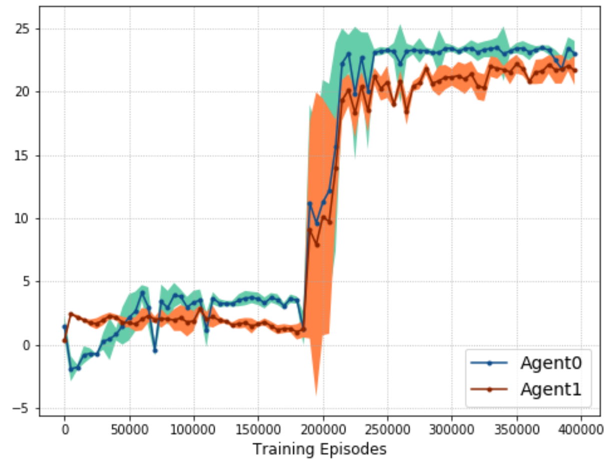
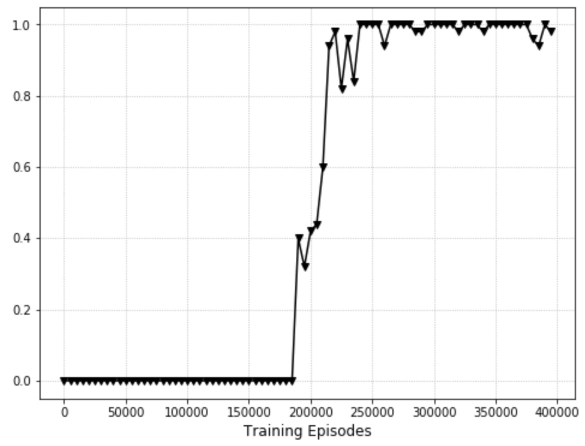
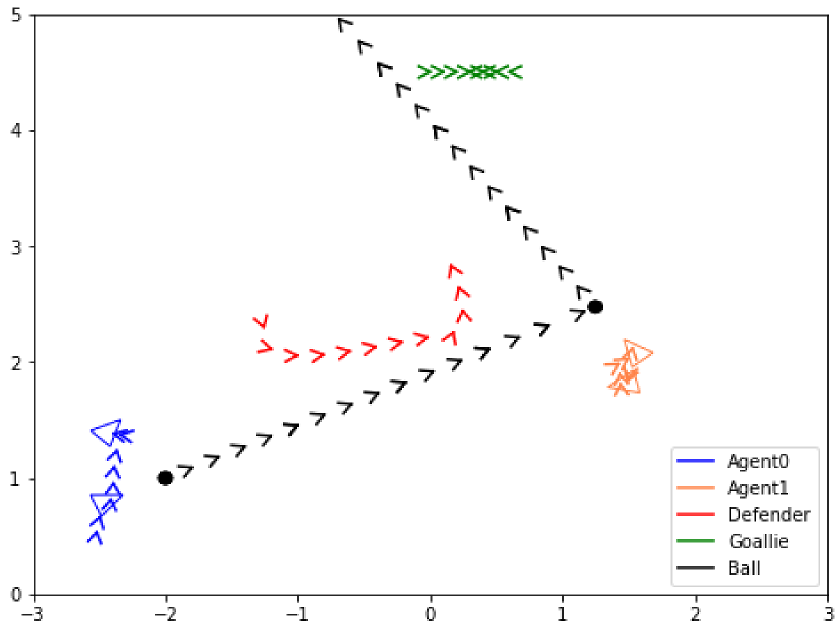


Figure 22. Goal percentage reported by IL.



The next figure reveals the behavior of the final policy obtained from a successful run on a random episode under validation conditions. In particular, it shows the trajectories followed by attackers, defenders and ball at different timesteps, where the chevrons' apices point to the direction of motion, triangles denote a kick action and circles indicate ball at rest.

Figure 23. Final team behavior reported by IL.



Two important observations must be made: 1) before the ball is passed, the attacker that will receive the ball displays an erratic motion, going forth and back senselessly, 2) the receiver waits until the ball slows down and finally stops, before it actually kicks it to goal. These facts demonstrate that IL does not produce an optimally coordinated behavior, as it would be the case if the receiver would move straight to the spot where it will intersect with the ball and kick it in mid motion.

6.2. Joint-Action Learners

Of the ten experiments in which JAL was applied, six generated successful policies with a perfect goal percentage in validation by the end of the learning procedure. Other two had final goal percentages of 0.52 and 0.78 respectively; thus, it could be expected that they will converge at some point not far away from the maximum number of learning steps of 400000. And the last two stayed at zero goal percentage the entire run.

The following figures present statistics of total rewards and goal percentages gathered from two successful JAL runs. They reveal that in some cases JAL suffers from the same prolonged local minima encountered by IL, where the second attacker never kicks the ball due to limited positive experiences. However, in other cases, JAL learns an effective yet inconsistent policy quickly, as soon as 80000 episodes, but it does not manage to truly converge to a proper policy until several learning steps later.

Figure 24. Average total reward per episode reported by JAL.

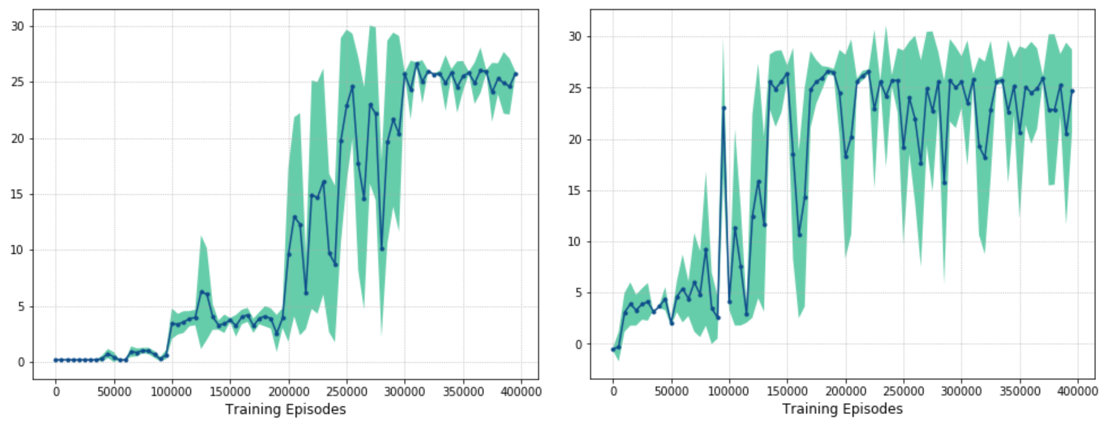
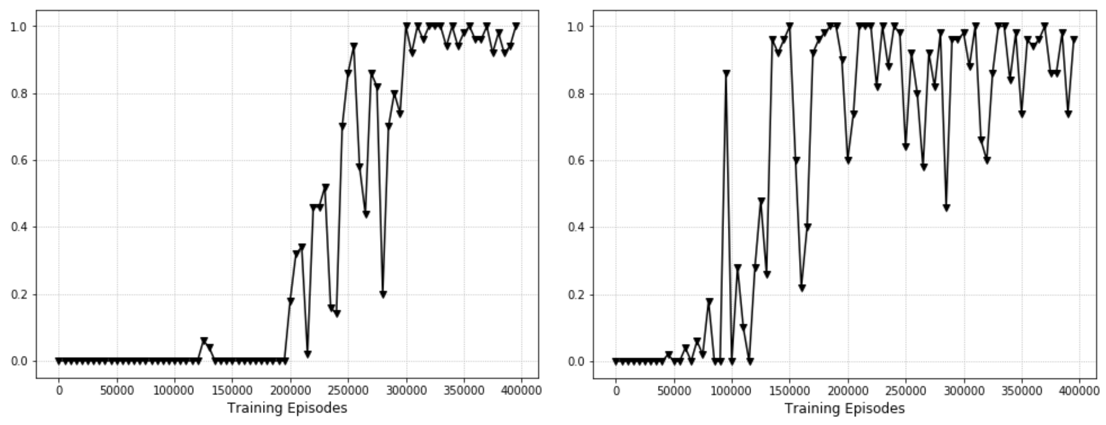
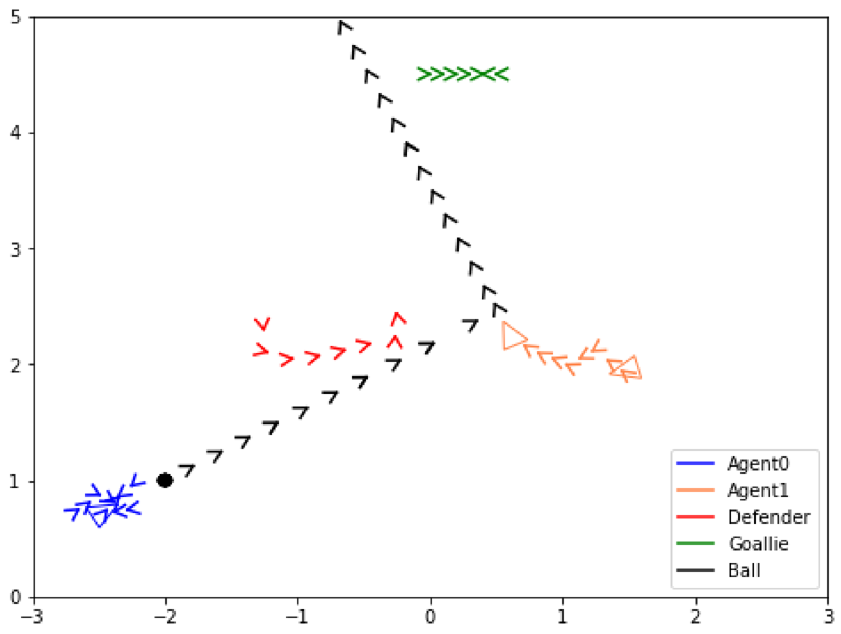


Figure 25. Goal percentage reported by JAL.



The figure below shows the response of the final policy resulting from a successful run when tested on a random episode. It should be noted that other favorable runs reported equivalent behaviors.

Figure 26. Final team behavior reported by JAL.



The previous result indicates that a JAL solution, similar to the case with IL, has the receiver acting erratically before the ball is passed to it, usually performing a couple of kicks when the ball is not around. In this case, the agent initially closer to the ball also moves fairly capriciously. Notwithstanding and unlike IL, there is a high degree of synchronization between the two attackers in JAL, because the receiver eventually gets a pass exactly to its position, such that sometimes it even kicks the ball to goal in one touch while it is still moving. Needless to say, if the receiver would hit the ball one timestep sooner or one timestep later it would likely not score a goal.

6.3. Discussion

By comparing the results obtained from Independent Learners and Joint-Action Learners implementations, it is evident that the latter outperforms the former in all aspects. JAL learned well-performing joint strategies much more consistently (60% of runs) than IL (20% of runs). In addition, final policies derived from JAL revealed a higher degree of coordination between agents, as one player learns to kick the ball in mid motion to score a goal. Not even in terms of converge rate wa IL able to surpass JAL, since they both converged after a comparable number of episodes.

Notwithstanding, the learning performance of both multi-agent RL approaches was far from ideal. First of all, both methods suffered from bad local minima; in fact, in at least a third of the learning steps (more than 150000 episodes) the agents were stuck in a local minimum. Clearly, this is an enormous waste of computational resources, especially since the replay buffer gets renewed after approximately 20000 episodes.

An important cause of these local minima was the use of shaped rewards to drive learning due to the lacking of additional mechanisms for dealing with sparse rewards. In particular, the reward functions considered in this implementation gave rise to unwanted behaviors such as: dribbling the ball to goal instead of kicking it, kicking the ball to goal instead of passing it to a teammate (in case of an agent blocked by defender) or passing the ball instead of kicking it to goal (in case of an agent in an open position).

This demonstrates once again that designing a shaped reward is not a trivial task even for fairly simple problems like the one tackled in this work. This issue would in general be aggravated in more complex problems that may require more convoluted reward functions, which in turn would originate many more local minima. Thus, there is a need for dismissing shaped rewards altogether, relying instead in new strategies for promoting exploration in sparse settings.

A second shortcoming that was observed is that standard DDPG is extremely sample-inefficient. Certainly, there are several sources of inefficiency; yet, the most fundamental and that affects this as well as most others Deep RL basic algorithms, is the lack generalization throughout the entire state-action space. Employing deep neural networks indeed provides some generalization, but this is limited to a small neighborhood around a specific point in the space.

Therefore, these basic algorithms still require examples gathered from all around the underlying space in order to learn an overarching policy. They do not exploit intrinsic symmetries that could exist within the state-action space, such that an optimal action learned for one small region would be instantly replicated to other analogous regions. Moreover, their learning is state-action specific instead of abstract, i.e. their goal is to discover an optimal action for every patch of state space, instead of learning universal rules such as “move in the direction of the ball until reaching it”; thus, they do not infer proper actions for unexplored states that are nonetheless contiguous to states for which an optimal action have already been defined.

Chapter 7
Conclusions and Future work

7.1. Conclusions

This work achieved successful implementations of both Independent Learners and Joint-Action Learners. Some key factors leading to this accomplishment were the following:

- A policy-based Reinforcement Learning method such as DDPG was crucial for handling the continuous state and action spaces of the RoboCup SPL simulator. As a bonus, in the case of JAL, DDPG avoids the exponential growth of the action space.
- The use of macro-actions, such as: walk and kick, which had to be considered as temporally-extended actions in order to avoid unbalanced poses of the virtual robots, simplified significantly the learning problem. As a matter of fact, decisions needed to be taken very sporadically; for instance, a solution to the offensive free-kick task requires less than 20 actions per agent.
- Given that shaped rewards were utilized, the addition of a cooperation term within the reward structure proved to be essential, especially for IL, since it fostered the emergence of passing behaviors.
- The insertion of time information within the state vector was yet another factor for success, as it provides the underlying feed-forward neural networks a means for recognizing which states are arbitrarily considered final due to the expiration of the episode’s time limit.

Despite this effectiveness in discovering a solution, the learning procedure itself was quite inefficient. It suffered from bad local minima, principally caused by the shaped rewards given to the learning agents. In addition, it did not exploit symmetries or abstractions found in the world in order to make use of fewer training examples.

With regard to the second objective of this thesis, Joint-Action Learners was clearly superior to Independent Learners in the offensive free-kick task, as it was suspected a priori. These results effectively contradict those obtained in [3] within a 2 vs. 1 Half-Field Offense domain, where it was reported that IL outperforms JAL simply because individual agents were able to solve the given task on their own occasionally. However, it must be highlighted that said work learned no meaningful cooperative strategies at all; hence, in practice, both approaches were equally and utterly unsuccessful in the multi-agent sense and the corresponding results should be taken with extreme care.

In turn, since in this thesis both approaches were truly successful, thanks to the addition of a defender that makes it impossible for a single agent to score a goal on its own, which eliminates a pernicious local minima, among other improvements presented as key factors previously; it is expected that the results obtained here will be more illuminating about the nature of the IL versus JAL dilemma, at least for domains equivalent to the offensive free-kick task.

Lastly, this work also proved the feasibility of learning team behaviors within the SPL simulation framework. Of course, several simplifications were made, such as: using ground truth information directly from the simulator to compose the world state, ignoring the cameras and learning over temporally-extended macro-actions instead of over the space of instantaneous joint angles.

Nevertheless, even if these simplifications were to be discarded, the learning problem will still be tractable. For instance, the B-Human framework includes numerous compute vision modules that can already process images provided by both cameras into estimates of the current world state. Furthermore, [58] showed that is possible to learn directly from raw pixel inputs, and [59] achieved successful results in high-dimensional continuous action spaces.

If anything, the biggest difficulty that the SPL simulation domain imposes is its large simulation time. Therefore, training for very complex behaviors might require several days of computation. However, this should be seen as an extra motivation to regard this domain as a benchmark to test new innovative and more sample-efficient solutions.

7.2. Future Work

Following studies should keep pushing the limits of Reinforcement Learning by removing some of the simplifications contemplated in this thesis. For instance, it would be more demanding and realistic to learn directly from raw images, instead of relying on ground truth information or on additional computer vision modules. On top of the added difficulty associated with the higher dimensional state space, another complication in this case will be that the learning problem cannot longer be formulated as an MDP or a Markov Game, since individuals and even a team of agents might not be able to observe the full world state every time. Therefore, the problem must then be treated as a decentralized or centralized Partially-Observable Markov Decision Process (POMDP). A possible resolution for this new scenario would be to employ deep recurrent neural networks that could maintain in memory estimates of state components that are no longer visible, as proposed in [70].

On top of that, the consideration of a bandwidth-limited communication between agents would make for an even more interesting problem under the JAL approach, which requires individuals to integrate their perceptions. Since images would be troublesome to transmit whole through the communication channel, some sort of encoding will be necessary. A fascinating alternative investigated in [71] would be to let agents develop an intrinsic emergent language, concurrently as they learn the task at hand.

As the task becomes more challenging, the problems with Deep Reinforcement Learning identified earlier will be exacerbated. Therefore, coming studies will have to incorporate new mechanisms into the learning method in order to be successful.

To start with, to reduce local minima issues, it would be preferable to cast aside shaped reward functions entirely. However, by doing so, the agent will almost certainly never receive a positive reward in complex tasks that cannot be resolved just by taking random actions; and hence, it will have no motivation to learn anything. Future work should look into intrinsic motivation methods for guiding exploration in the absence of extrinsic rewards; in particular, into novelty- or curiosity-driven exploration.

For instance, [72] first lets the agent interact randomly with the environment. Second, it identifies, through clustering techniques, 'just out of reach' purposes, i.e. states that have been reached infrequently. Third, Deep Reinforcement Learning algorithms are used to learn separate policies for achieving each purpose. Finally, these polices are embedded into macro-actions (also called options) and appended to the action set of the agent; thus, creating a self-reinforcing loop as the cycle repeats itself several times.

Moreover, [73] computes approximate density models across the state space as it is being explored and from these it derives a pseudo-count, i.e. a scalar score that estimates the novelty associated to each state (how many it has been visited). At last, such pseudo-counts are internalized within the action selection strategy to favor novel states.

In turn, to alleviate the generalization problem that makes learning very inefficient and time-consuming, especially for robotics or simulated robotics domains with long execution times, there are a couple of avenues following studies could take.

Particularly, to help with the lack of understanding of spatial symmetries a readily applicable technique is data augmentation. For instance, in the offensive free-kick domain reviewed in this work, a contactless walk or kick action could be translated and rotated in any way possible as long as it is checked that the augmented action does not cause extra collisions; similarly, a kick to a stationary ball could be rotated radially around the ball's center.

A more principled solution for this same problem, but for cases where the states are represented by images, would be to introduce Capsule Networks [74]. They have had an important success in the area of deep learning for vision applications, because they are able to acknowledge spatial relationships within the data, which leads to more robust predictions and fewer training examples needed. In short, Capsule Networks learn over vectors instead of over scalars as traditional networks do; thus, they can better preserve relative spatial information.

On the other hand, for dealing with the lack of abstraction, further studies should consider utilizing Hierarchical Reinforcement Learning (HRL). In HRL there is a structure composed of several separate learning modules, where only the top module perceives the world state and only the bottom module executes actions in the environment. Each module receives a more abstract action from the module above and tries to enforce it by setting a sequence of more specific actions sent to the module below for actual execution. HRL effectively breaks the state to action mapping at the level modules and allows higher-up modules to set abstract intrinsic goals, as for example, “move to ball”.

References

[1] Kurek, M.: Deep Reinforcement Learning in Keepaway Soccer. Master's thesis. Poznan University of Technology, 2015.

[2] Bai, A., Russell, S., Chen, X.: Concurrent Hierarchical Reinforcement Learning for RoboCup Keepaway. In: Proceedings of RoboCup 2017: Robot Soccer World Cup XXI, to appear, 2018.

[3] Hausknecht, M.J.: Cooperation and communication in multiagent deep reinforcement learning. PhD thesis, University of Texas at Austin, Austin, USA, 2016.

[4] Tan, M.: Multi-agent reinforcement learning: Independent vs. cooperative agents. In: Proceedings 10th International Conference on Machine Learning (ICML-93), pp. 330–337, 1993.

[5] Lauer, M., Riedmiller, M.: An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In: Proceedings 17th International Conference on Machine Learning (ICML-00), pp. 535–542, 2000.

[6] Kapetanakis, S., Kudenko, D.: Reinforcement learning of coordination in cooperative multiagent systems. In: Proceedings 18th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02), pp. 326–331, 2002.

[7] Matignon, L., Laurent, G.J., Le Fort-Piat, N.: Hysteretic Q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In: Intelligent Robots and Systems (IROS 2007), IEEE/RSJ International Conference on. IEEE, pp. 64–69, 2007.

[8] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, Ju., Aru, Ja., Vicente, R.: Multiagent cooperation and competition with deep reinforcement learning. In: PLoS One 12 (4), e0172395, 2015.

[9] Egorov, M.: Multi-agent deep reinforcement learning. University of Stanford, Department of Computer Science, Technical Report, 2016.

[10] Omidshafiei, S., Pazis, J., Amato, C., How, J.P., Vian, J.: Deep Decentralized Multi-task Multi-Agent Reinforcement Learning under Partial Observability. In: Proceedings 34th International Conference on Machine Learning (ICML-17), pp. 2681-2690, 2017.

[11] Palmer, G., Tuyls, K., Bloembergen, D., Savani, R.: Lenient Multi-Agent Deep Reinforcement Learning. In: Proceedings 17th International Conference

on Autonomous Agents and Multi-Agent Systems (AAMAS-18), to appear, 2018.

[12] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., Mordatch, I.: Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), 30, pp. 6379–6390, 2017.

[13] Foerster, J., Farquhar, G., Afouras, T., Nardelli, N., Whiteson S.: Counterfactual Multi-Agent Policy Gradients. arXiv preprint arXiv:1705.08926, 2017.

[14] Guestrin, C., Lagoudakis, M., Parr, R.: Coordinated reinforcement learning. In: Proceedings of the 19th International Conference on Machine Learning (ICML-02), pp. 227–234, 2002.

[15] Guestrin, C., Venkataraman, S., Koller, D.: Context specific multiagent coordination and planning with factored MDPs. In: Proceedings National Conference on Artificial Intelligence, pp. 253–259, 2002.

[16] Kok, J., Vlassis, N.: Sparse cooperative Q-learning. In: Proceedings 21st International Conference on Machine Learning (ICML-04), pp. 61–68, 2004.

[17] Kok, J., 't Hoen, P., Bakker, B., Vlassis, N.: Utile coordination: Learning interdependencies among cooperative agents. In: Proceedings IEEE Symposium on Computational Intelligence and Games (CIG05), pp. 29–36, 2005.

[18] Melo, F., Veloso, M.M.: Learning of coordination: Exploiting sparse interactions in multiagent systems. In: Proceedings 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-09), pp. 773-780, 2009.

[19] Lau, Q.P., Lee, M. L., Hsu, W.: Coordination guided reinforcement learning. In: Proceedings 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-12), vol. 1, pp. 215–222, 2012.

[20] Yu, C., Zhang, M., Ren, F., Tan, G.: Multiagent learning of coordination in loosely coupled multiagent systems. In: IEEE transactions on cybernetics 45 (12), pp. 2853-2867, 2014.

[21] Ovalle Castañeda, A..: Deep reinforcement learning variants of multi-agent learning algorithms. Master's thesis, School of Informatics, University of Edinburgh, 2016.

[22] Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: Proceedings 11th International Conference on Machine Learning (ICML-94), pp. 157–163, 1994.

[23] Hu, J., Wellman, M.P.: Multiagent reinforcement learning: Theoretical framework and an algorithm. In: Proceedings 15th International Conference on Machine Learning (ICML-98), pp. 242–250, 1998.

[24] Littman, M.L.: Friend-or-foe Q-learning in general-sum games. In: Proceedings 18th International Conference on Machine Learning (ICML-01), pp. 322–328, 2001.

[25] Greenwald, A., Hall, K.: Correlated-Q learning. In: Proceedings 20th International Conference on Machine Learning (ICML-03), pp. 242–249, 2003.

[26] Könönen, V.: Asymmetric multiagent reinforcement learning. In: Proceedings IEEE/WIC International Conference on Intelligent Agent Technology (IAT-03), pp. 336–342, 2003.

[27] Uther, W.T., Veloso, M.M.: Adversarial reinforcement learning. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, US, 1997.

[28] Suematsu, N., Hayashi, A.: A multiagent reinforcement learning algorithm using extended optimal response. In: Proceedings 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02), pp. 370–377, 2002.

[29] Conitzer, V., Sandholm, T.: AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents. In: Proceedings 20th International Conference on Machine Learning (ICML-03), pp. 83–90, 2003.

[30] Weinberg, M., Rosenschein, J.S.: Best-response multiagent learning in non-stationary environments. In: Proceedings 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-04), pp. 506–513, 2004.

[31] Powers, R., Shoham, Y., Vu, T.: A general criterion and an algorithmic framework for learning in multi-agent systems. In: Machine Learning 67(1-2), pp. 45–76, 2007.

[32] Chakraborty, D., Stone, P.: Multiagent Learning in the Presence of Memory-Bounded Agents. In: Journal of Autonomous Agents and Multiagent Systems (JAAMAS), 28(2), pp. 182–213, 2013.

[33] Gupta, J.K., Egorov, M., Kochenderfer, M.: Cooperative multi-agent control using deep reinforcement learning. In: International Conference on Autonomous Agents and Multiagent Systems, pp. 66-83, 2017.

[34] Park, K.H., Kim, Y.J., Kim, J.H.: Modular Q-learning based multi-agent cooperation for robot soccer. In: Robotics and Autonomous Systems, 5(2), pp. 109–122, 2001.

[35] Köse, H., Tatlidede, U., Meriçli, C., Kaplan, K., Akin, H.L.: Q-Learning based Market-Driven Multi-Agent Collaboration in Robot Soccer. In: Proceedings Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN 2004), pp. 219–228, 2004.

[36] Stone, P., Sutton, R.S., Kuhlmann, G.: Reinforcement Learning for RoboCup-Soccer Keepaway. In: Adaptive Behavior, 13(3), pp. 165–188, 2005.

[37] Celiberto, L.A., Ribeiro, C.H.C., Costa, A.H.R., Bianchi, R.A.C.: Heuristic Q-Learning soccer players: A new reinforcement learning approach to RoboCup simulation. In: RoboCup 2007: Robot Soccer World Cup XI, pp. 220–227, 2008.

[38] Ma, J., Cameron, S.: Combining policy search with planning in multi-agent cooperation. In: Proceedings of RoboCup 2008: Robot Soccer World Cup XII, pp. 532–543, 2009.

[39] Kalyanakrishnan, S., Stone, P.: Learning Complementary Multiagent Behaviors: A Case Study. In: Proceedings of RoboCup 2009: Robot Soccer World Cup XIII, pp. 153–165, 2010.

[40] Leonetti, M., Iocchi, L.: LearnPNP: A tool for learning agent behaviors. In: Proceedings of RoboCup 2010: Robot Soccer World Cup XIV, pp. 418-429, 2010.

[41] Liu, Q., Ma, J., Xie, W.: Multiagent Reinforcement Learning with Regret Matching for Robot Soccer. Research article, Harbin Institute of Technology, China, 2013.

[42] Mendoza, J.P., Simmons, R.G., Veloso, M.M.: Online Learning of Robot Soccer Free Kick Plans Using a Bandit Approach. In: Proceedings 26th International Conference on Automated Planning and Scheduling (ICAPS), pp. 504–508, 2016.

[43] Howard, R. A.: Dynamic Programming and Markov Processes. The MIT Press, Cambridge, Massachusetts, 1960.

[44] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Second edition, in progress. The MIT Press, Cambridge, Massachusetts, London, England. 2014-2015.

[45] Rummery, G.A, Niranjan, M.: On-line Q-Learning using Connectionist Systems. Cambridge University, Engineering Department, Technical Report, 1994.

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

[47] McCulloch , W.S., Pitts, W.: A Logical Calculus of Ideas Immanent in Nervous Activity. In: Bull. Mathematical Biophysics, Vol, 5, pp. 115-133, 1943.

[48] Cybenko, G.: Approximations by superpositions of sigmoidal functions. In: Mathematics of Control, Signals, and Systems, 2(4), pp. 303-314, 1989.

[49] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. In: Nature, 323 (6088), pp. 533–536, 1986.

[50] Qian, N.: On the momentum term in gradient descent learning algorithms. In: Neural networks: The Official Journal of the International Neural Network Society, 12(1): pp. 145–151, 1999.

[51] Ioffe, S., Szegedy, C.: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: Proceedings 32th International Conference on Machine Learning (ICML-15), pp. 448-456, 2015.

[52] Hahnloser, R., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., Seung, H. S.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. In: Nature, 405: pp. 947–951, 2000.

[53] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. In: Journal of Machine Learning Research, 15, pp. 1929–1958, 2014.

[54] Duchi, J., Hazan, E., Singer, Y.: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In: Journal of Machine Learning Research, 12, pp. 2121-2159, 2011.

[55] Zeiler, M.D.: ADADELTA: An adaptive learning rate method. arXiv preprint arXiv:1212.5701, 2012.

[56] Kingma, D.P., Ba, J.L.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[57] Dozat, T.: Incorporating Nesterov Momentum into Adam. In: International Conference on Learning Representations, Workshop, (1): pp. 2013–2016, 2016.

[58] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik,A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. In: Nature, 518, pp. 529-533, 2015.

[59] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[60] Hafner, R., Riedmiller, M.: Reinforcement learning in feedback control. In: Machine Learning, 84, (1-2): pp. 137–169, 2011.

[61] Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized Experience Replay. In: International Conference on Learning Representations, 2016.

[62] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: RoboCup: The Robot World Cup Initiative. In: IJCAI-1995 workshop on entertainment and AI-Alife, Montreal, Quebec, 1995.

[63] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) Rule Book. 2018.

[64] Röfer, T., Laue, T., Bülter, Y., Krause, D., Kuball, J., Mühlenbrock, A., Poppinga, B., Prinzler, M., Post, L., Roehrig, E., Schröder, R., Thielke, F.: B-Human: Team Report and Code Release 2017. Deutsches Forschungszentrum für Künstliche Intelligenz, Universität Bremen, 2017.

[65] Laue, T., Röfer, T._ SimRobot - Development and Applications. In: Heni Ben Amor, Joschka Boedecker, Oliver Obst (Eds.), The Universe of RoboCup Simulators - Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008), Lecture Notes in Artificial Intelligence, 2018.

[66] Cherubini, A., Leonetti, M., Marchetti, L., De Luca, A., Iocchi, L., Nardi, D., Oriolo, G., Vendittelli, M: SPQR Team Description Paper. RoboCup 2008: Robot Soccer World Cup XII, 2008.

[67] Röfer, T.: CABSL - C-based agent behavior specification language. In: RoboCup 2017: Robot World Cup XXI, Lecture Notes in Artificial Intelligence. Springer, 2018.

[68] Stone, P., Sutton, R.S.: Keepaway Soccer: A Machine Learning Testbed. In: A. Birk, S. Coradeschi, and S. Tadokoro (Eds.): RoboCup 2001, LNAI 2377, pp. 214–223, 2002.

[69] Kalyanakrishnan, S., Liu, Y., Stone, P.: Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In: Proceedings of RoboCup 2006: Robot Soccer World Cup X, pp. 72–85, 2007.

[70] Hausknecht, M., Stone, P.: Deep Recurrent Q-Learning for Partially Observable MDPs. arXiv preprint arXiv:1507.06527, 2015.

[71] Foerster, J., Assael, I.A., de Freitas, N., Whiteson., S.: Learning to communicate with deep multi-agent reinforcement learning. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), 29, pp. 2137-2145, 2016.

[72] Machado, M.C., Bowling, M.: Learning Purposeful Behaviour in the Absence of Rewards. arXiv preprint arXiv:1605.07700, 2016.

[73] Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., Munos, R.: Unifying count-based exploration and intrinsic motivation. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), 2016a.

[74] Hinton, G.E., Krizhevsky, A., Wang, S.D:. Transforming auto-encoders. In: International Conference on Artificial Neural Networks, pp. 44–51. Springer, 2011.