
# ACTOR-CRITIC ARCHITECTURES IN PARAMETERIZED ACTION SPACE FOR ROBOCUP SOCCER SIMULATOR 2D

**By**

*CRISTIANO SANTOS DE OLIVEIRA*

**B.Sc. Dissertation**


Federal University of Pernambuco
secgrad@cin.ufpe.br
www.cin.ufpe.br/~secgrad

Recife/2020

Cristiano Santos de Oliveira

# ACTOR-CRITIC ARCHITECTURES IN PARAMETERIZED ACTION SPACE FOR ROBOCUP SOCCER SIMULATOR 2D

*A B.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.*

*Advisor:* Tsang Ing Ren

Recife
2020

# Agradecimentos

Tem uma música que fala "o futuro não demora" e esse meu futuro, concluindo mais esse ciclo, não demorou de acontecer pelo imenso apoio que tive dos meus pais. Dona Marisa e Seu Ari, devo tudo que eu sou hoje a vocês que sempre colocaram minha educação em primeiro lugar e não mediram esforços para me proporcionar uma ótima formação, mesmo com todas as dificuldades. Não poderia deixar de agradecer às pessoas das quais vou levar pra vida toda e que fizeram dessa trajetória um processo maravilhoso, dentre elas estão: toda a família RobôCIn, onde me senti extremamente acolhido e que tenho muito carinho por todos que fazem parte desse grupo, além de tudo que pude aprender fazendo parte dele; meus irmãos de outras mães Mateus Gonçalves (Gonça) e Jailson Gomes (Panda), que me aguentaram e aguentam diariamente; e por último, e não menos importante, o amor da minha vida, Dandara Gomes que atura minhas ladainhas e me dá todo apoio e carinho. Amo todos vocês e como diria Emicida "quem tem um amigo tem tudo".

*"Life to me, like a box of chocolate"*

*—INSTITUTIONALIZED - KENDRICK LAMAR*

# Resumo

Deep Reinforcement Learning (DRL) é um tipo de algoritmo orientado a objetivo que combina Deep Learning e Reinforcement Learning, tendo como principal característica a capacidade de aprender a realizar atividades complexas partindo de uma mecânica baseada em ação e recompensa, aplicada a um determinado ambiente observável. DRL soluciona um vasto universo de problemas com essa configuração, tais como no domínio dos video-games, finanças, robótica entre outros. Como exemplo, no campo da robótica, existe o *RoboCup Soccer Simulator 2D*, um ambiente simulado de robôs autônomos que jogam futebol. Para esse tipo de ambiente com ampla complexidade, um tipo específico de arquitetura em DRL é frequentemente usado: os algoritmos Actor-Critic. O objetivo deste trabalho é avaliar arquiteturas Actor-Critic extendidas para espaços de ação parametrizáveis no ambiente simulado RoboCup Soccer Simulator 2D, além de propor funções de recompensa que proporcionem a realização de atividades de futebol por um agente ofensivo. A análise foi feita progressivamente realizando o treinamento das diferentes arquiteturas para os diferentes tipos de atividade, tais como ir para a bola e levar a bola ao gol, definidos durante a pesquisa com níveis de complexidade incrementais, indo do mais simples ao mais complexo. Esses níveis de complexidade foram definidos a partir da atividade que o agente deveria realizar e da quantidade de parâmetros na ação que o agente pode usar para aquela atividade. As funções de recompensa propostas foram avaliadas comparando a capacidade de convergência para cada modelo e análisando o comportamento dos modelos treinados, chegando em funções que apresentaram melhores resultados para as atividades de ir até a bola e levar a bola ao gol com e sem oponente.

# Abstract

Deep Reinforcement Learning (DRL) is a method that combines Reinforcement Learning and Deep Learning. This method is assigned as a goal-oriented algorithm which learn how to reach a complex objective performing actions and receiving evaluated feedbacks from the environment. DRL is applied to solve problems in a large range of applications in domains such video-games, finance, robotics and many more. These domains are regularly composed by complex environments that contains a large set of features and a continuous action space such the *RoboCup Soccer Simulator 2D* (RCSS2D), which is an autonomous agent based simulation to mimic soccer games. For this kind of environments with continuous action space is frequently used a sub-field of the DRL: the Actor-Critic algorithms. The objective of this work is to evaluate extended Actor-Critic architectures in a parameterized continuous action space using the RCSS2D environment and also propose reward functions to perform soccer tasks in this environment. The evaluation was made by training state of art Actor-Critic models incrementally for the set of tasks with low complexity to the high complexity, which the complexity was defined by the kind of task that the robot should learning and the configuration of the action that it should perform, taking parameterization configuration as the core of the actions changes for each experiment. The evaluation of reward functions proposed in this work was made by comparing with functions proposed in related works, taking into account the convergence factor and the behavior analysis.

# List of Figures

# List of Tables

# List of Algorithms

# Contents

# 1

# Introduction

Deep Reinforcement Learning (DRL) have being broadly used to the challenging problem of defining optimal behavior actions to an agent at a complex environment. Study fields as the game playing domain and robotics domain that face this action decision problem have already demonstrated relevant performances by applying DRL techniques. In game playing domain, Deep Q-Learning (DQN) approaches significant results the Atari environment (MNIH et al., 2013) and the table game Go at Alpha Go (SILVER et al., 2017). In robotics, Deep Deterministic Policy Gradient (DDPG) (LILLICRAP et al., 2015) and DQN MNIH et al. (2013) have being applied for navigation and control tasks (Chen et al., 2020; GU et al., 2017). Robotics tasks normally are characterized by having a continuous action space and for this set, there is a specific branch on DRL that can support more directly the problem.

Environments with continuous action space needs DRL methods that focus on the behavior of the agent and prioritizes the agent policy. There are DRL methods that directly optimize the behavior decision making, addressed as Policy Based or Policy Gradient (PG) methods. Extending the PG methods, there are the Actor Critic architectures which at the same time that perform the policy evaluation increasing the action probability also approximate the value function to determine the action effectiveness.

Some well known Actor Critic architectures are the Asynchronous Advantage Actor-Critic (A3C) (MNIH et al., 2016) where the core approach is the parallel training addressing multiple actors and synced globally from time to time; the Synchronous Advantage Actor-Critic (A2C), a synchronous version of the A3C; the Deep Deterministic Policy Gradient (LILLICRAP et al., 2015), a combination of the Deterministic Policy Gradient (DPG) (SILVER et al., 2014) and DQN applied to a model-free off-policy actor critic method; the Soft Actor-Critic (HAARNOJA et al., 2018) that in regard to expand the exploration introduce the entropy measure into the reward.

To apply Actor-Critic architectures in a parameterized action space we choose the RoboCup Soccer Simulation 2D (RCSS2D) (ROBOCUP, 2019) (ROBOCUP, 2018) Half-Field-Offensive environment (HFO) (HAUSKNECHT, 2015) that simulates a soccer game with eleven by eleven autonomous agents which contains well structured features and high complexity

agent dynamic. In the HFO framework, the hierarchies of action parameterization can be widely explored, since it provides a flexible configuration for each type of action that could be performed according to the complexity levels: low-level actions, mid-level actions and high-level actions.

Build a good player in the RCSS2D environment is a complex and challenging task since the agent receives a large amount of sensory information from the environment, with a continuous state space and having to perform parameterized continuous actions to achieve the goals. For this work, we focus on the individual agent learning centralized into creating an offensive player as a target of the DRL using extended Actor-Critic Architectures.

The objective of this work is to build and evaluate an Actor-Critic Architecture extended into a parameterized action space, applying to the RoboCup Soccer Simulation 2D (RSS2D) environment. The major focus is to implement three Actor-Critic Architectures: the Deep Deterministic Policy Gradient (LILLICRAP et al., 2015), Twin Delayed Deep Deterministic (TD3)(FUJIMOTO; HOOF; MEGER, 2018), and the Soft Actor-Critic (HAARNOJA et al., 2018). Also, we analyze different types of action space parameterization for the RCSS2D, considering the features available for the environment of training and the type of behavior that is going to be learned.

This work is separated into 5 chapters. In Chapter 2 we present the central concepts studied (e.g DRL methods), and related works. In Chapter 3 we describe our methodology to build the architectures that we evaluate. Chapter 4 presents the experiments and the results. In Chapter 5 we discuss the results and manifest some ideas to improve our results.

# 2

# Background

In this chapter, we describe the main concepts of Reinforcement Learning and the environment that is applied in this work. The chapter is structured to present the basis of Reinforcement Learning theory, following we describe the state-of-art Actor-Critic methods and the environment of RoboCup Soccer Simulator 2D. In addition we present MASSON; RANCHOD; KONIDARIS (2016), HAUSKNECHT; STONE (2016) and ZARE et al. (2019) related works.

## 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning directed to problems of self-learning decision making over time, based on trial and error in an environment. In this technique, an agent learns to act in an environment through interactions that returns reward values for each action.

The two main elements of RL are: agent and environment. The Figure 2.1 illustrates the interaction cycle of these elements, the agent obtains an observation state $S_t$ from the environment, and processes the policy $\pi$ that delegates a chosen action $A_t$ back to the environment. The action execution at the environment makes a transition step which provides a new observation state $S_{t+1}$ and a feedback value in form of reward $R_{t+1}$.

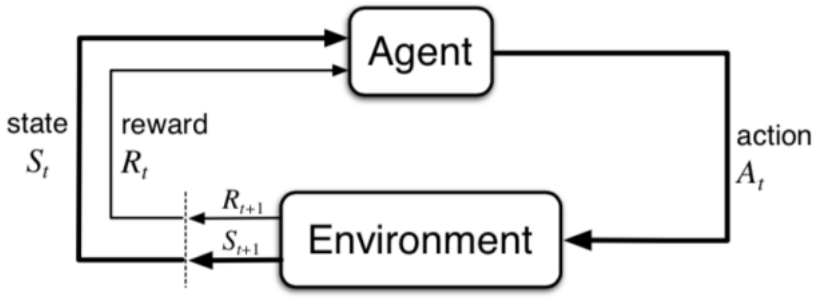



**Figure 2.1:** Reinforcement Learning interaction cycle. Image from SUTTON; BARTO (2011).

As a formal definition of this interaction structure, BELLMAN (1957) formulates the Markov Decision Processes (MDP), which determines the skeleton of most RL problems. The

key concept of the MDP frame refers to the fact that only the current state influences the future state. Markov decision process consists of a 5-tuple $M =< S, A, P, R, \gamma >$, assigned as:

- $S$ - a set of states $s$;
- $A$ - a set of actions $a$;
- $P$ - transition probability function;
- $R$ - reward function;
- $\gamma$ - future rewards discount factor.

The purpose of reinforcement learning is to achieve a behavior strategy that decides which action should be taken in a certain state and maximizes the reward accumulation, formally defined as optimal policy $\pi_*$:

$$\pi_* = arg\max_{\pi} V_\pi(s), \pi_* = arg\max_{\pi} Q_\pi(s,a) \tag{2.1}$$

As shown in Equation 2.1, the optimal policy calculation considers two main components: the state-value function or V-function, and the action-value function or Q-function. The V-function describes an evaluation value for a state s, and Q-function returns an evaluation value for an action a given state s. A further definition of the policy calculation is the Bellman Equations (BELLMAN, 1957), see Equations 2.2, 2.3, 2.4 and 2.5, which demonstrate a recursive update process built on both state-value and action-value functions.

$$V_*(s) = \max_{a \in A} Q_*(s,a) \tag{2.2}$$

$$Q_*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} V_*(s') \tag{2.3}$$

$$V_*(s) = \max_{a \in A} (R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} V_*(s')) \tag{2.4}$$

$$Q_*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} \max_{a \in A} Q_*(s',a') \tag{2.5}$$

Here we have some RL terms that will compose this work:

- *Model-based*: RL methods that depend on the model of the environment.
- *Model-free*: RL methods with no dependency on the environment model.
- *On-policy*: when the $Q(s,a)$ function is learned from actions perfomed in the current policy $\pi$.
- *Off-policy*: when the $Q(s,a)$ function is learned from taking different actions (e.g: random actions) without the need of a policy.

## 2.2 Actor-Critic Methods

In reinforcement learning, there is a branch of study called Policy Gradient algorithms. The essence of Policy Gradient algorithms is the Policy Gradient Theorem (SUTTON et al. (1999)), expressed in Equation 2.6, which implies a learning policy based on the gradient of a determined performance measure $J$ and embraces problems with continuous parameterization. Policy Gradient methods learn the policy directly by perturbing the parameter $\theta$ and the final purpose is to find the best $\theta$ that produces the highest returns using a gradient ascent. By definition: $s$ is a state $s \in S$; $a$ is an action $a \in A$; $d^{\pi}(s)$ is the on-policy state distribution under $\pi$.

$$\nabla_{\theta} J(\theta) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} Q^{\pi}(s,a) \nabla_{\theta} \pi_{\theta}(a \mid s) \tag{2.6}$$

This theorem introduces the problem of high variance due to the extensive environment dependency that the discounted reward has since it is proportional to the gradient. To address this problem, there are the Actor-Critic methods that use a model to learn the value function (Critic) in addition to the policy update that composes another model (Actor). See the vanilla Actor-Critic method in Algorithm 1.

**Algorithm 1:** Actor-Critic. Resource SUTTON; BARTO (2011).

Initialize parameters $s$, $\theta$, $w$ and learning rates $\alpha_{\theta}, \alpha_{w}$; sample $\alpha \sim \pi_{\theta}(a \mid s)$.

**for** $t \leftarrow 1$ ***to*** $T$ **do**

- Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s' \mid s,a)$
- Then sample the next action $a' \sim \pi_{\theta}(a' \mid s')$
- Update the policy parameters: $\theta \leftarrow \theta + \alpha_{\theta} Q_w(s,a) \nabla_{\theta} log \pi_{\theta}(a \mid s)$;
- Compute the correction (TD error) for action-value at time $t$:
  $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
  and use it to update the parameters of $Q$ function: $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
- Move to $a \leftarrow a'$ and $s \leftarrow s'$

The following subsections present three state-of-art Actor-Critic methods: Deep Deterministic Policy Gradient, Twin Dueling Deep Deterministic Gradient, and Soft Actor-Critic.

### 2.2.1 Deep Deterministic Policy Gradient

LILLICRAP et al. (2015) formulated the Deep Deterministic Policy Gradient algorithm, which is a model-free off-policy actor-critic that combines Deterministic Policy Gradient (DPG) (SILVER et al., 2014) with the Deep Q-Network (DQN) (MNIH et al., 2013). which SILVER et al. (2014) formulates a policy modelation as a deterministic decision in DPG, and MNIH et al. (2013) establishes the use of a frozen target network and an experience replay buffer to learn Q-function with the DQN method. Different from the original DQN method that operates in discrete space, the DDPG extends it to solve problems with continuous space using the actor-critic structure while learning a deterministic policy. See the DDPG pseudo-code in Algorithm 2.

**Algorithm 2:** DDPG. Resource LILLICRAP et al. (2015).

Randomly initialize critic network $Q(s,a \mid \theta^Q)$ and actor $\mu(s \mid \theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** $t \leftarrow$ *1 to* $M$ **do**
  Initialize a random process $N$ for action exploration
  Receive initial observation state $s_1$
  **for** $t \leftarrow$ *1 to* $T$ **do**
    Select action $a_t = \mu(s_t \mid \theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $S_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i \mid \theta^Q)^2)$
    Update the actor policy using the sampled gradient:
    $\nabla_{\theta^\mu \mu} \mid_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(s,a \mid \theta^Q) \mid_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu) \mid_{s_i}$
    Update the target networks: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

### 2.2.2 Twin Dueling Deep Deterministic Gradient

Twin Delayed Deep Deterministic (TD3) (FUJIMOTO; HOOF; MEGER (2018)) evolves from the DDPG method by incrementing some operations to avoid the overestimation of the Q-values. This kind of overestimation is a common issue for DDPG which the Q-function starts to maximize the Q-values leading to policy break. FUJIMOTO; HOOF; MEGER (2018) demonstrates by applying three critical methods, that compose the TD3 algorithm, it can prevent

the overestimation issue, the methods are:

- *Clipped Double Q-learning*: In Double Q-Learning WANG et al. (2016) method uses two independent networks to the action selection, the actors $(\mu_{\theta_1}, \mu_{\theta_2})$, and Q-value estimation, the critics $(Q_{w_1}, Q_{w_2})$. The Clipped Double Q-learning adopts the minimum estimation in the process, in favor of bias underestimation, the resulted Bellman targets are expressed in Equation 2.7 and 2.8.

$$y_1 = r + \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_1}(s')) \tag{2.7}$$

$$y_2 = r + \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_2}(s')) \tag{2.8}$$

- *Delayed Update of Target and Policy Networks*: this operation considers a lower frequency of policy updates in regards to Q-function which reduces the variance. The process is made by holding the policy network until the error reaches a low enough value after several updates.

- *Target Policy Smoothing*: introduces a smoothing regularization approach by increasing a random amount of noises to the target action.

See TD3 pseudo-code in Algorithm 3.

**Algorithm 3:** TD3. Resource FUJIMOTO; HOOF; MEGER (2018).

Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$ with random parameters $\theta_1$, $\theta_2$, $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $B$
**for** $t \leftarrow 1$ ***to*** $T$ **do**
  Select action with exploration noise $a \leftarrow \pi_\phi(s) + \varepsilon$, $\varepsilon \sim N(0, \sigma)$ and observer reward $r$ and new state $s'$
  Store transition tuple $(s, a, r, s')$ in $B$
  Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $B$ $\widetilde{a} \leftarrow \pi_{\phi'}(s') + \varepsilon$, $\varepsilon \sim clip(N(0, \widetilde{\sigma}))$ $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', a)$
  Update critics $\theta_i \leftarrow arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  **if** $t$ *mod* $d$ **then**
    Update $\phi$ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_i}(s, a) \mid_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks: $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$ , $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$

### 2.2.3 Soft Actor-Critic

Soft Actor-Critic (SAC) (HAARNOJA et al., 2018) is an algorithm that extends the maximum entropy reinforcement learning framework to boost exploration. HAARNOJA et al. (2018) show that this solution is capable of learning to achieve complex tasks by generating an extensive sampling of random actions with the inclusion of the entropy measure to the reward computation. We choose this method precisely because of its exploratory approach since the environment we apply is highly dynamic.

The main SAC components are the combination of actor-critic architecture, off-policy formulation, and entropy maximization. The policy training process intent is to maximize the expected return and the entropy in parallel, as expressed in Equation 2.9, which the **H** specifies the entropy measure and $\alpha$ defines the temperature parameter which controls the entropy influence. By definition: $s_t$ is a state in time $t$ and $a_t$ an action in time $t$.

$$\mathbf{J}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\rho_{\pi_\theta}} [r(s_t,a_t) + \alpha \mathbf{H}(\pi_\theta(. \mid s_t))] \tag{2.9}$$

The Algorithm 4 shows the SAC method.

**Algorithm 4:** Soft Actor-Critic. Resource HAARNOJA et al. (2018).

Initialize target network weights: $\widetilde{\theta_1} \leftarrow \theta_1$, $\widetilde{\theta_2} \leftarrow \theta_2$
Initialize an empty replay pool $D$
**for** *each iteration* **do**
  **for** *each environment step* **do**
    Sample action from the policy: $a_t \sim \pi_\phi(a_t \mid s_t)$
    Sample transition from the environment: $s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)$
    Store the transition in the replay pool: $D \leftarrow D \cup \{(s_t, a_t, r(s_t,a_t), s_{t+1})\}$
  **for** *each gradient step* **do**
    Update the Q-function parameters: $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1,2\}$
    Update policy weights: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    Adjust Temperature: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
    Update target network weights: $\widetilde{\theta_i} \leftarrow \tau\theta_i + (1-\tau)\widetilde{\theta_i}$ for $i \in \{1,2\}$

## 2.3 Environment

In this section, we describe the environment that we adopt to build our proposed agent and perform the experiments. It is divided into two subsections: RoboCup Soccer Simulator 2D and Extended Half-Field-Offensive Framework.

### 2.3.1 RoboCup Soccer Simulator 2D

The RoboCup Soccer Simulation 2D is an open-source environment maintained by the RoboCup Federation, which is used in the annual Robot World Cup competition and conference that has been held since 1997. It is composed of the main software RoboCup Soccer Simulation Server, rcssserver (ROBOCUP, 2018), and also by peripheral softwares that makes possible engines of game visualizations.

Rcssserver mimics a robot soccer game in a 2D dimensional environment based on an autonomous agent structure, where two teams with eleven players each play against each other. The rcssserver is implemented with a server-client architecture and adopts a UDP/IP communication protocol, that makes it possible the players management by each team through package exchanging, where the teams send the desirable commands to a particular player and the rcssserver executes this command. Therefore the rcssserver operates as a referee by executing and analyzing the game situations and also manages the players requests from each team. The Figure 2.2 illustrates the components of the RCSS2D complete system.

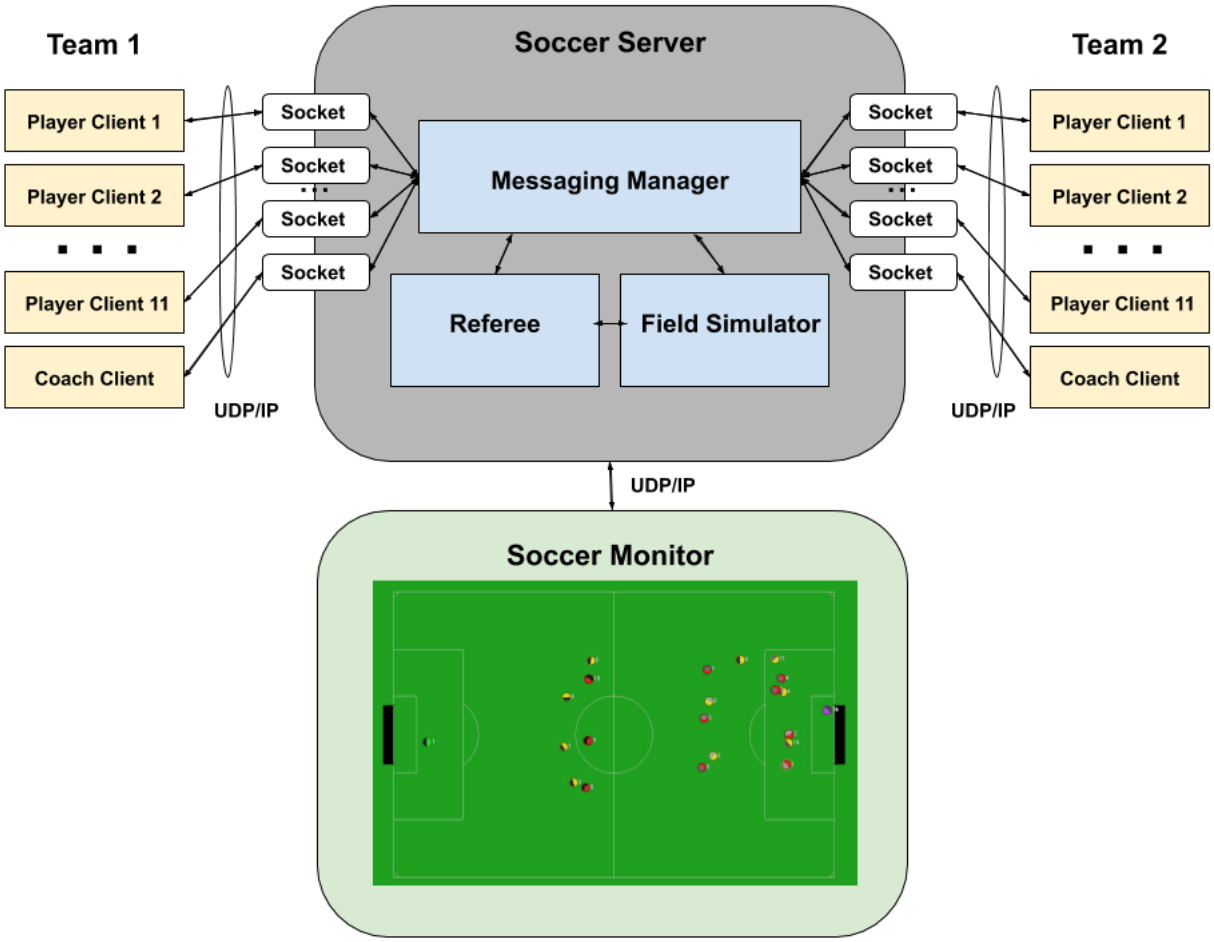



**Figure 2.2:** RoboCup Soccer Simulator 2D system diagram

Two types of client agents can interact with the rcssserver: coaches and players. The coaches are privileged clients that assist the players, they are capable of visualizing the whole

game state to make analyzes and inferences, each team has one coach-client available. The players are clients that represent the actual robot soccer agent in the RCSS2D field.

The RCSS2D player agent has a stable structure of sensor models to observe the game environment, which are: aural sensor model, vision sensor model, and the body sensor model. As described in the CHEN et al. (2003), the aural sensor detects messages sent by the referee, the coaches, and the other players; the visual sensor detects visual information about the field, like the distance and direction to objects in the player's current field of view, as shown in Figure 2.3; the body sensor detects the current "corporal" status of the player, like its vitality, speed, and neck angle.

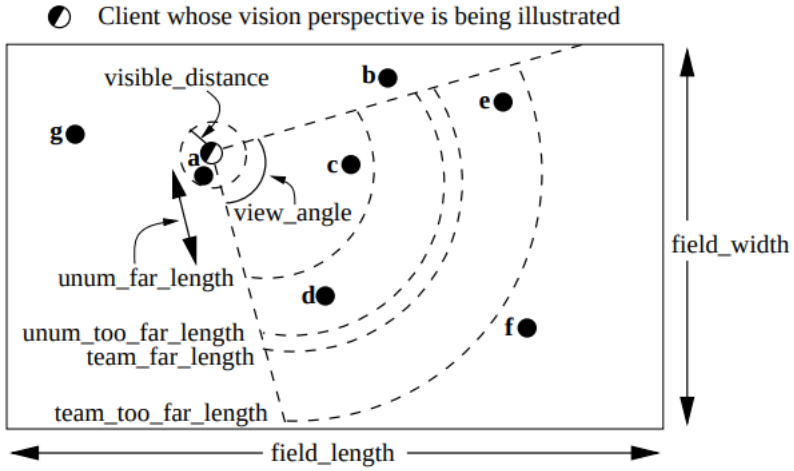



**Figure 2.3:** Individual player vision representation in the soccer server. The player is the one shown as two semi-circles and the light semi-circle is its front. The black circles represent objects in the world, which could be teammate players, opponent players or the ball.
Image from CHEN et al. (2003).

Besides the sensor models, the player agent handles a group of action models to interact with the context field, we explain these models below:

- ***Dash***: applied to accelerate the player in direction of its body, which takes the *acceleration power* $\beta \in [-100, 100]$ as a parameter.
- ***Kick***: used to move the ball to a given direction, which takes the *acceleration power* $\beta \in [-100, 100]$ and the *angle* $\alpha \in [-180, 180]$ the player wants to kick the ball to. Also, to be able to kick the ball the player has to be positioned in the kickable area as shown in the Figure 2.4.

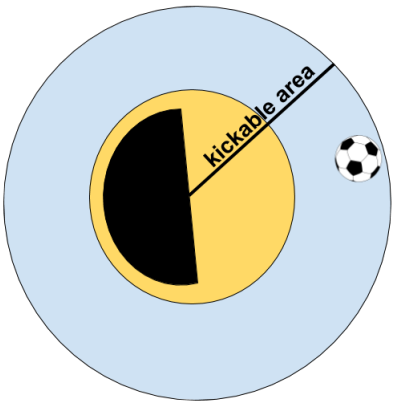



**Figure 2.4:** Player kickable area representation.

- ***Catch***: related to holding the ball when it is positioned in the catchable area, see Figure 2.5. The only player that can execute this action is the goalie.

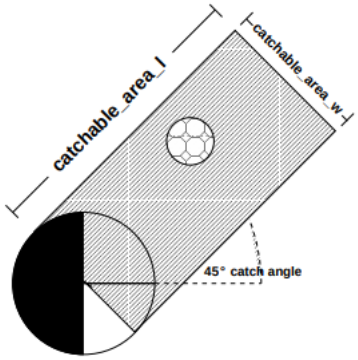



**Figure 2.5:** Representation of the goalkeeper catchable area using 45 degrees as parameter. Image from CHEN et al. (2003)

- ***Tackle***: this action is used to tackles the ball given a *acceleration power* $\beta \in [-100, 100]$ as parameter.
- ***Move***: this command is utilized to place a player directly onto a desired position on the field. This action can not be made through normal play, only when the game is paused.
- ***Say***: with this command the player is able to broadcast messages to other players.
- ***Turn***: is applied to change the player's body direction, which uses the wanted *angle* $\alpha \in [-180, 180]$ as parameter.
- ***TurnNeck***: is used to change the player's visual field direction, which takes the desired *angle* $\alpha \in [-180, 180]$ as parameter.

### 2.3.2 Half Field Offensive Framework

The Half Field Offense framework (HAUSKNECHT, 2015) is an extension of the RCSS2D which implements an interface with high-level abstraction and was built to give support to Reinforcement Learning researches. HFO is coupled upon the rcssserver software, as shown in Figure 2.6.

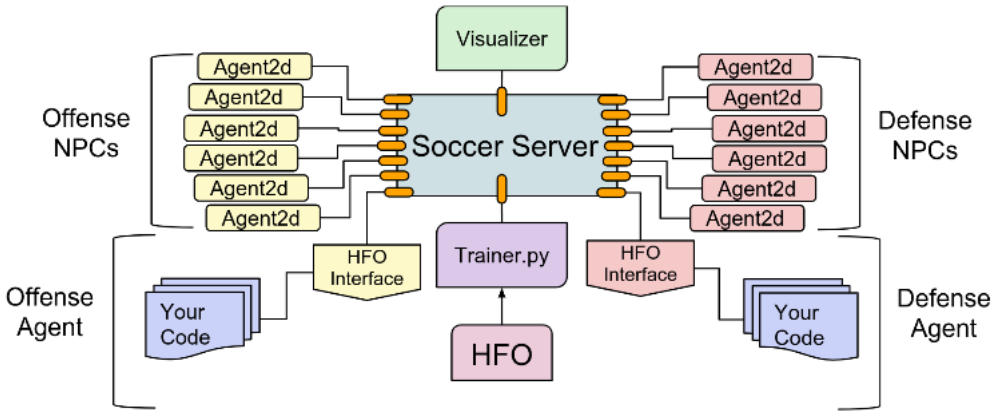



**Figure 2.6:** Half Field Offensive framework diagram. Image from HAUSKNECHT (2015).

The state representation in the HFO domain has two interfaces that can choose: low-level or high-level. These interface levels affect the complexity to achieve tasks in the environment and, for this work, we only use the high-level interface configuration. The high-level interface is a more compact representation of the macro state of RCSS2D, which includes the following features: agent's position and orientation; distance and angle to the ball: is kickable state; distance and angle to the goal; the nearest opponents and teammates.

For the action state interface, the HFO provides three levels of abstraction: low-level, mid-level, and high-level. We explore in this work the three action interface configurations and each one is defined as follows:

- Low-level interface: implements the four base action mentioned in Section 2.3.1 ($Dash(power, angle)$, $Kick(power, angle)$, $Turn(angle)$ and $Tackle(angle)$);
- Mid-level interface: implements four new types of action: $KickTo(tar_x, tar_y, speed)$, $MoveTo(tar_x, tar_y)$ and $DribbleTo(tar_x, tar_y)$;
- High-level interface: implements nine new types of action, however for this work we only use one of them, which is the *Shoot* action that executes the best available shot to the goal, applying implementations of the low-level actions.

## 2.4 Related Works

Complex continuous environment model as RoboCup Soccer Simulator 2D is widely used to consolidate researches in the Deep Reinforcement Learning field. In this section, we

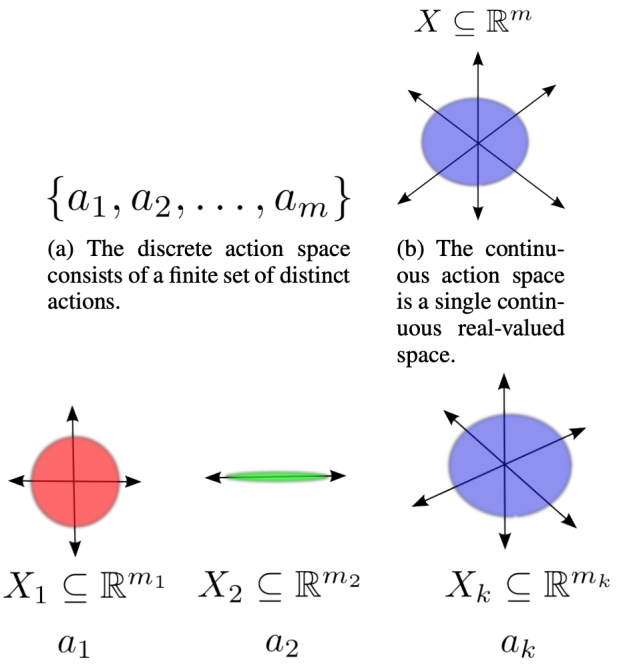
introduce some of these studies.

MASSON; RANCHOD; KONIDARIS (2016) introduced a model-free algorithm with parameterized actions, named as Q-PAMDP, that focus in achieving an action-selection policy given a pre-defined parameterization. The core goal of the Q-PAMDP is to alternate between learning action-selection (QLearn) and parameter-selection (P), as shown in the Q-PAMDP method in Algorithm 5.

**Algorithm 5:** Q-PAMDP. Resource MASSON; RANCHOD; KONIDARIS (2016).

Parameter update method $P$
Q-Learning algorithm $QLearn$
Initialize parameters $\theta_0$, $w_0$
Initialize weights $w \leftarrow QLearn^{(\infty)}(M_\theta, w_0)$
**for** *each iteration until* $\theta$ *converges* **do**
  $\theta \leftarrow P^{(k)}(J_w, \theta)$
  $w \leftarrow QLearn^{(\infty)}(M_\theta, w)$

To perform Q-PAMDP they consider canonical environments with continuous state space and a finite set of discrete actions that is modulated by a continuous set of parameters, see Figure 2.7, which fits with the RCSS2D environment structure that was used as a benchmark platform to their experiments.




(a) The discrete action space consists of a finite set of distinct actions.

(b) The continuous action space is a single continuous real-valued space.

(c) The parameterized action space has multiple discrete actions, each of which has a continuous parameter space.

**Figure 2.7:** Three action space representations: discrete, continuous, and parameterized. Image from MASSON; RANCHOD; KONIDARIS (2016).

Following a similar approach as MASSON; RANCHOD; KONIDARIS (2016), another research that demonstrates a more RCSS2D based structure was presented by HAUSKNECHT; STONE (2016), which introduces a study using an extended DDPG architecture to perform parameterized actions in RCSS2D environment, the Figure 2.8 illustrates the proposed architecture. However, they only explore one type of action parameterization using the HFO framework. This work was a great inspiration for our study.

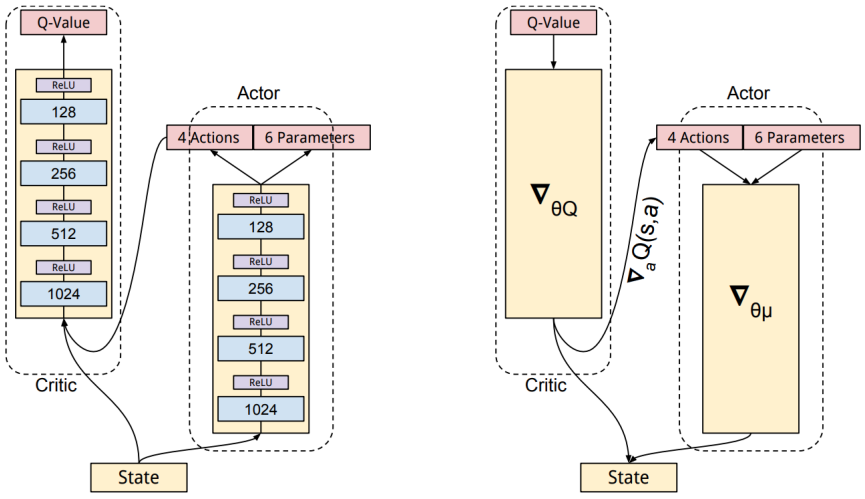



**Figure 2.8:** Representation of the parameterized agent architecture presented by HAUSKNECHT; STONE (2016). At left, the actor and critic networks topology. At right, the actor update flow, which backwards pass generates critic gradients $\nabla_a Q(s, a \mid \theta^Q)$ the action, resulting $\nabla_{\theta^\mu}$ through back-propagating the actor.

In the annual RoboCup competition of the Soccer Simulation 2D league, the participants have to submit a Team Description Paper showing relevant contributions in the RCSS2D field. The precursors using Reinforcement Learning approaches to develop RCSS2D agents came from the RoboCup conference, and we want to highlight the ZARE et al. (2019) work for the 2019 RoboCup edition. They implemented a defensive RCSS2D agent using Deep Deterministic Policy Gradient showing significant results against hand-coded agents, but they fixed a pre-defined discrete action space in the proposed model.

# 3

# Methodology

In this chapter, we present the extended Actor-Critic architecture with Parameterized Continuous Action space: AC-PA. We specify the building process of an RCSS2D robot agent that uses the proposed architecture to perform soccer tasks. For that we describe the soccer policy definitions setting up rewards functions associated with each soccer policy. We present how the state observation is mapped for each type of task and also defines the continuous action parameterization that composes the AC-PA architecture.

The soccer policies and the architecture settings take into consideration the complexity level of the type of exploration the model has to perform to achieve the optimal policy convergence. As the goal of the work is to evaluate how the AC-PA behaves to different kinds of continuous action parameterization, we encapsulate complexity levels from defining a simple base soccer policy with an atomic action parameterization to others more complete soccer policy with multi-parameter actions.

The more complete soccer policy of this work is an offensive robot player in the RCSS2D environment. We define and evaluate how the AC-PA is capable of reaching this policy by configuring the continuous action parameterization and setting one of the state-of-art Actor-Critic methods properly for this goal.

## 3.1 Soccer Behavior Policies

In a soccer game, the player has a diverse amount of behaviors that could proceed in a determined situation at the game. Each player role such as offensive, defensive, or even goalkeeper has different ways to interact during the match, and even going through these roles there are subgroups of roles which the positioning in the field impacts the individual type of player behavior. As the granularity of behaviors for a soccer robot agent are extensive and involves a multi-agent study, we decompose and choose the individual behaviors to define the soccer policies for this work.

The offensive soccer player individual behavior centers on some canonic characteristics: ball possession and scoring. With these attributes, we define three soccer policies as goals for our AC-PA architecture, which are: go-to-ball, ball-to-goal, and ball-to-goal-with-opponent.

- ***Go-To-Ball Policy***: represents the behavior which initially the robot is located in a distant position from the ball $p_0^{robot} = (x_0, y_0)$ in the instant $t_0$ and is capable of arriving at the position of the ball in the instant $t_{0+n}$ subsequently. As illustrates the Figure 3.1, initially the robot is placed distant from the ball and after $n$ iterations in the environment it reaches the ball.

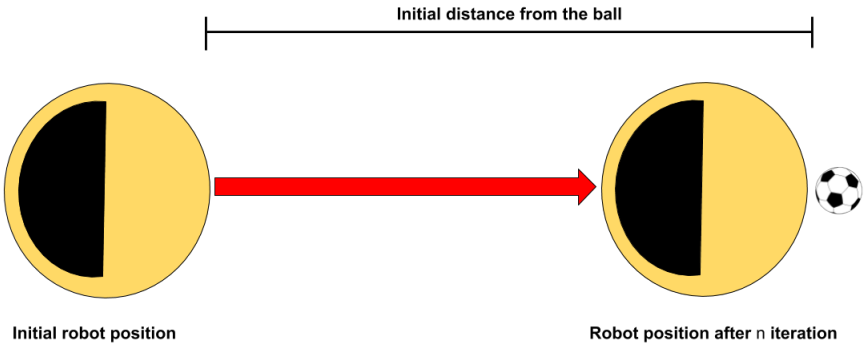



**Figure 3.1:** Go-to-ball behavior representation.

- ***Ball-To-Goal Policy***: represents the behavior which the robot is able to arrive at the ball located with a certain distance from it and following it takes the ball to score at the goal. As shown in Figure 3.2, the robot is located distant from the ball at initial instant $t_0$, the robot arrives at the ball at instant $t_{0+n}$ and scores the ball to the goal at instant $t_{0+n+m}$.

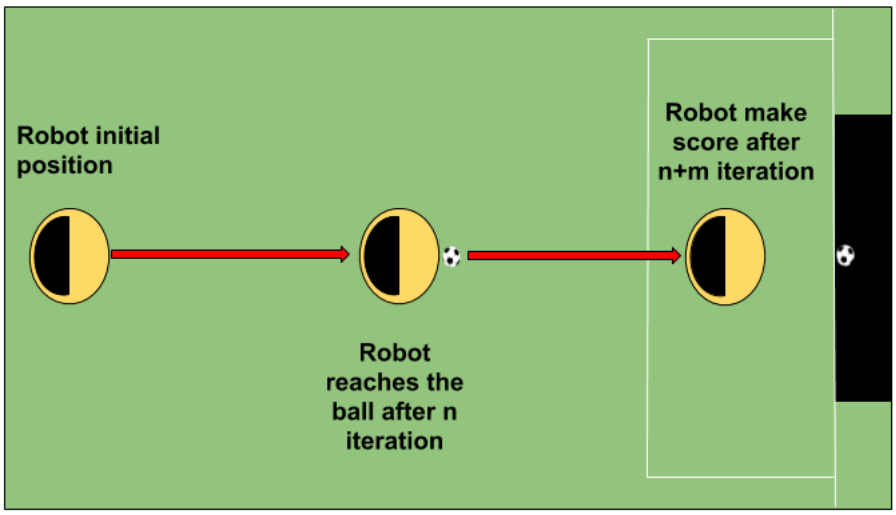



**Figure 3.2:** Ball-to-goal behavior representation.

- ***Ball-To-Goal-With-Opponent Policy***: represents a similar behavior as the Ball-To-Goal having the increment of an opponent in the field, which we define as a goalkeeper. So in this case the robot has to pass by a hand-coded goalkeeper to score, as shown in Figure 3.3.

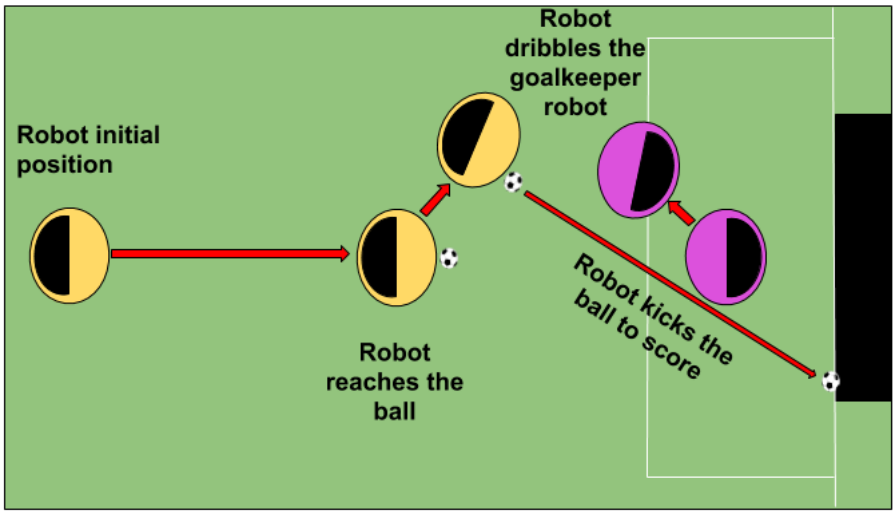



**Figure 3.3:** Ball-to-goal-with-opponent behavior representation.

## 3.2 Reward Functions

In this section we describe the reward functions settled in the environment for each soccer behavior defined. For the RCSS2D environment, the general reward function characterizes as winning a full game that takes an extensive sparse exploration performance to a DRL model reaches an optimal policy. To avoid this sparse exploration we focus on key features that directly affect each policy, detailed below.

The go-to-ball behavior's most important feature is the robot distance from the ball, this feature designates the core of our *go-to-ball reward function*. The inverse proporcional distance is considered to define the absolute reward value, which the lowest is the robot distance from the ball the highest is the reward value and the highest is the distance the lowest is the reward value. The robot moving direction also influences the final reward value, which we include a directional gradient of the robot to designate if the reward is going to be negative, when the robot moves away from the ball, and positive, when it moves towards the ball. The go-to-ball reward is represented on Equation 3.1 which $d_t^{rb}$ is the robot distance from the ball in time $t$ and $d_{t-1}^{rb}$ is the robot distance from the ball in time $t-1$, the constant $C^{ball} = 100$, and the kickable state represents when the robot is near to the ball, receiving the maximum reward value when the kickable state is equal to true.

$$r_t = \begin{cases} 1 & \text{if } isKickable = 1 \\ C^{ball}(d_{t-1}^{rb} - d_t^{rb}) & \text{otherwise} \end{cases} \tag{3.1}$$

In *ball-to-goal reward function* we designate three levels of reward values: the first level is to receive a reward for reach the ball, for that we use the robot potential gradient related to the ball based on distance, as in Equation 3.1, expressed as $(d_{t-1}^{rb} - d_t^{rb})$ which is the difference between the last robot distance from the ball represented as $d_{t-1}^{rb}$ and the current robot distance

from the ball represented as $d_t^{rb}$; the second level of reward value is to take the ball to the goal, which we consider the ball potential gradient base on the difference of the ball position to the center of the goal, expressed as $(d_{t-1}^{bg} - d_t^{bg})$ which is the difference between the last ball distance from the center of the goal $d_{t-1}^{bg}$ and the current ball distance from the center of the goal $d_t^{bg}$; the third level of reward value of the ball-to-goal reward function is to achieve the score, which receives the maximum reward value when scores. The ball-to-goal reward function is expressed in Equation 3.2, which each component has a constant factor to attribute the importance of the potential gradients, the constants are defined as $C^{ball} = 100$ and $C^{goal} = 1000$; the isGoal state defines a score state when is true or not a score state when false.

$$r_t = \begin{cases} 1000 & \text{if } isGoal = 1 \\ C^{ball}(d_{t-1}^{rb} - d_t^{rb}) + C^{goal}(d_{t-1}^{bg} - d_t^{bg}) & \text{otherwise} \end{cases} \tag{3.2}$$

HAUSKNECHT; STONE (2016) presented the Equation 3.3 to define a reward function for the same ball-to-goal behavior policy. Each component of this equation is expressed as: $d_t(x,y)$ is the distance between two elements $x$ and $y$ in instant $t$, a element represents the robot, b element represents the ball, g element represents the center of the goal, $I_t^{kick}$ represents the kickable constant that is equal to 1 when the robot is near to the ball in instant $t$, $I_t^{goal}$ is the score state constant that is equal to 1 when scores in instante $t$. We use this equation to establish a benchmark of comparison to our proposed Ball-To-Goal reward function.

$$r_t = d_{t-1}(a,b) - d_t(a,b) + I_t^{kick} + 3(d_{t-1}(b,g) - d_t(b,g)) + 5I_t^{goal} \tag{3.3}$$

For the *ball-to-goal-with-opponent reward function* we make a similar approach as described for the Ball-To-Goal reward, but for this case we increase the opponent factor as a component. As the Equation 3.4 describe, the ball-to-goal-with-opponent reward function returns an expressive negative value when the state ballWithOpponent is true which indicates the behavior policy to prevent the situation that the opponent takes the ball.

$$r_t = \begin{cases} 1000 & \text{if } isGoal = 1 \\ -1000 & \text{if } isBallWithOpponent = 1 \\ C^{ball}(d_{t-1}^{rb} - d_t^{rb}) + C^{goal}(d_{t-1}^{bg} - d_t^{bg}) & \text{otherwise} \end{cases} \tag{3.4}$$

## 3.3 Observation Space and Continuous Action Parameterization

For each soccer behavior, the agent needs a minimum observation scope to aggregate in the policy to define the type of action it should perform, and the HFO provides a flexible environment to be extended as a personalized implementation of the observation and action space.

This section describes the observation space configuration and the parameterized modulation of the continuous action space for the proposed architecture.

### 3.3.1 Observations Space

The observation space represents the input features in the AC-PA model, which for each soccer behavior we define an observation space domain. Robot players in RCSS2D matches have limitations in the observed features provided to it, which the environment declines the agent perception capacity depending on the game situation. We delineate the observation space domain for each soccer behavior defined in this work by considering canonical features of the robot agent as self position and ball position.

- ***Go-to-ball observation space***: this domain is defined as the relative position of the agent to the ball position $p_t^{rel} = (x_t^{rel}, y_t^{rel})$, which the ball is characterized by the origin. Having the robot agent position as $p_t^{robot} = (x_t^{robot}, y_t^{robot})$ and the ball position as $p_t^{ball} = (x_t^{ball}, y_t^{ball})$, the $p_t^{rel}$ components are defined as $x_t^{rel} = x_t^{robot} - x_t^{ball}$ and $y_t^{rel} = y_t^{robot} - y_t^{ball}$. the resulting observation space domain is $S^{gtb} = \{x_t^{rel}, y_t^{rel}\}$ which $x \in [-52, 52] \subset \mathbb{R}$ and $y \in [-34, 34] \subset \mathbb{R}$.

- ***Ball-to-goal observation space***: this domain is composed by the agent position $p_t^{robot} = (x_t^{robot}, y_t^{robot})$, the ball position $p_t^{ball} = (x_t^{ball}, y_t^{ball})$ and the kickable flag $k_t$, which the kickable flag represents a binary value that returns 1 when the agent is able to kick the ball otherwise returns 0, the resulting observation space is $S^{btg} = \{x_t^{robot}, y_t^{robot}, x_t^{ball}, y_t^{ball}, k_t\}$ which $x \in [-52, 52] \subset \mathbb{R}$ and $y \in [-34, 34] \subset \mathbb{R}$.

- ***Ball-to-goal-with-opponent observation space***: this domain is similar to the ball-to-goal domain by incrementing the observable opponent position to the state $p_t^{opp} = (x_t^{opp}, y_t^{opp})$, resulting in $S^{bgopp} = \{x_t^{robot}, y_t^{robot}, x_t^{ball}, y_t^{ball}, k_t, x_t^{opp}, y_t^{opp}\}$.

### 3.3.2 Continuous Action Space Parameterizations

The action space structure in the HFO framework, as mentioned in Section 2.3.2, is composed by low-level primitive actions, mid-level actions and high-level actions, that each action is defined as a function $f\colon X \to Y$ which $x$ is a n-tuple of the action parameters $x = (p_1, \ldots, p_n)$ , $p \in \mathbb{R}, x \in X$ and $y$ is an primitive value which represents the action on the RCSS2D environment, $y \in Y$. To support these groups of actions and its space domain, we extended the Actor-Critic architecture with continuous action parameterization (AC-PA).

The AC-PA contemplates the following parameterized action space: we have the infinite set of actions $A = \{a_1, \ldots, a_\infty\}$ that is the output of the AC-PA model, which $a$ is defined as n-tuple $a = (t, x_1, \ldots, x_n)$ with $t, x \in [-1, 1] \subset \mathbb{R}$, which $t$ maps the action type and $x$ maps the parameters; we consider the parameterization function $\psi\colon \mathbf{A} \longrightarrow \mathbf{P}$ to transform the AC-PA output in a valid action to be performed at the environment, which P is a infinite set of

actions $P = \{p_1, \ldots, p_\infty\}$, and $p$ is defined as a n-tuple $p = (d, x_1, \ldots, x_n)$ with $d$ being a discrete value which belong to a finite set of discretized action types $D = \{d_1, \ldots, d_m\}$ for $D \subset \mathbb{N}$, and $x \in [-1, 1] \subset \mathbb{R}$; the parameterization function $\psi$ is described in Equation 3.5, having $m()$ as the mapping function that transforms the continuous output $a$ of the AC-PA to a environment output. The parameterization function $\psi$ returns $m(t, x_1, ..., x_n)$ when the cardinality of the n-tuple $a$ is greater then 2, otherwise returns $(0, m(t))$ indicating the default action zero in the first component, defined as the Dash action, and with the atomic mapping function $m(t)$ that only maps the parameter of the action.

$$\psi(a) = \begin{cases} m(t, x_1, ..., x_n) & \text{if } |\mathrm{a}| \geq 2 \\ (0, m(t)) & \text{otherwise} \end{cases} \tag{3.5}$$

The Figure 3.4 illustrates the parameterization flow.

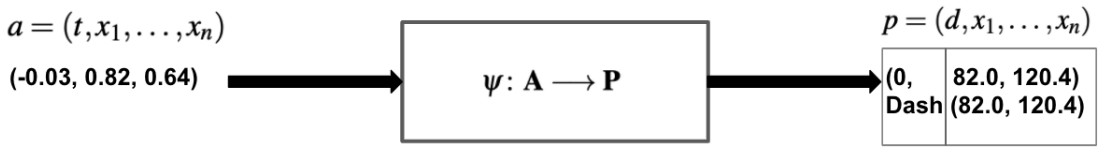



**Figure 3.4:** Represents the action parameterization flow, which the parameterization function $\psi$ receives a action $a = (-0.03, 0.82, 0.64)$ from the actor-critic model and returns a parameterized action $p = (0, 82.0, 120.4)$ to be performed in the environment as a $Dash(82.0, 120.4)$.

For each soccer behavior we consider different configurations of the action structure provided by the HFO framework, which contemplates distinct cardinalities on the n-tuple a in the AC-PA action domain A, distinct discrete action domains D and distinct mapping functions. We describe these configurations as follow:

1. Go-to-ball action parameterizations:

   - *Directional Dash*: this parameterization only uses the Dash action which originally takes two parameters: power and direction. For the directional dash we set a constant power $C^{power}$ and receive the direction as a parameter. Definition: discrete domain $D = \{d^{dash}\}$ which $d^{dash} = 0$; $P = \{p_1, \ldots, p_n\}$ for $p \in [-180, 180] \subset \mathbb{R}$ and the map function is $m\colon I \to O$ which $I \subseteq [-1, 1] \subset \mathbb{R}$ and $O \subseteq [-180, 180] \subset \mathbb{R}$.
   - *Complete Dash*: this parameterization considers the complete dash action Dash(power, direction). Definition: discrete action domain $D = \{d^{dash}\}$ which $d^{dash} = 0$; parameterized action domain $P = \{p_1, \ldots, p_n\}$, having $p = (x_1, x_2)$ which $x_1 \in [-100, 100] \subset \mathbb{R}$, $x_2 \in [-180, 180] \subset \mathbb{R}$ and the map function is $m\colon I \times I \to D \times O^d \times O^p$ which $I \subseteq [-1, 1] \subset \mathbb{R}$, $O^d \subseteq [-180, 180] \subset \mathbb{R}$ and $O^p \subseteq [-100, 100] \subset \mathbb{R}$.

2. Ball-to-goal parameterization:

   - *Low-level actions parameterization*: we select the $Dash(power, angle)$, $Kick(power, angle)$ and $Turn(angle)$ action from the low-level primitive actions to compose this parameterization. Definition: discrete action domain $D = \{d^{dash}, d^{kick}, d^{turn}\}$ which $d^{dash} = 0, d^{kick} = 1, d^{turn} = 2$; parameterized action domain $P = \{a_1, \ldots, a_n\}$, having $p = (t, x_1, x_2)$ which $t \in [-1,1] \subset \mathbb{R}$, $x_1 \in [-100,100] \subset \mathbb{R}$, $x_2 \in [-180,180] \subset \mathbb{R}$ and the map function is $m$: $I \times I \times I \rightarrow D \times O^d \times O^p$ which $I \subseteq [-1,1] \subset \mathbb{R}$, $O^d \subseteq [-180,180] \subset \mathbb{R}$ and $O^p \subseteq [-100,100] \subset \mathbb{R}$.

3. Ball-to-goal-with-opponent parameterizations:

   - *Low-level actions parameterization*: is the same of ball-to-goal parameterization.
   - *Mid-level actions parameterization*: we select the $KickTo(tar_x, tar_y, speed)$, $MoveTo(tar_x, tar_y)$, $DribbleTo(tar_x, tar_y)$ actions from the hfo mid-level actions to compose this parameterization. Definition: discrete action domain $D = \{d^{kick-to}, d^{dribble-to}, d^{move-to}\}$ $d^{kick-to} = 0$, $d^{dribble-to} = 1$, $d^{move-to} = 2$; parameterized action domain $P = \{p_1, \ldots, p_n\}$, $p = (t, x_1, x_2, x_3)$ which $t, x_1, x_2 \in [-1,1] \subset \mathbb{R}$, $x_3 \in [0,3] \subset \mathbb{R}$ and the map function is $m$: $I \times I \times I \times I \rightarrow D \times O^{tar} \times O^{tar} \times O^{speed}$ which $I \subseteq [-1,1] \subset \mathbb{R}$, $O^{tar} \subseteq [-1,1] \subset \mathbb{R}$ and $O^{speed} \subseteq [0,3] \subset \mathbb{R}$.
   - *Mid-level+High-level actions parameterization*: we select $MoveTo(tar_x, tar_y)$ and $DribbleTo(tar_x, tar_y, speed)$ from the hfo mid-level actions and the $Shoot()$ from hfo high-level actions to compose this parameterization. Definition:discrete action domain $D = \{d^{dribble-to}, d^{move-to}, d^{shoot}\}$ $d^{dribble-to} = 0$, $d^{move-to} = 1$, $d^{shoot} = 2$; parameterized action domain $P = \{p_1, \ldots, p_n\}$, $p = (t, x_1, x_2, x_3)$ $t, x_1, x_2 \in [-1,1] \subset \mathbb{R}$, $x_3 \in [0,3] \subset \mathbb{R}$ and the map function is $m$: $I \times I \times I \times I \rightarrow D \times O^{tar} \times O^{tar} \times O^{speed}$ which $I \subseteq [-1,1] \subset \mathbb{R}$, $O^{tar} \subseteq [-1,1] \subset \mathbb{R}$ and $O^{speed} \subseteq [0,3] \subset \mathbb{R}$.

## 3.4 Agent Architecture

In the previous sections of this chapter, we described the components of the AC-PA agent architecture. AC-PA architecture merges these components and includes the Actor-Critic model as the core of the structure, as shown in Figure 3.5 we consider the observation space S as the domain of the input component, passing through the Actor-Critic processing which outputs a raw continuous action $a \in A$, this action $a$ is parameterized by the parameterization component that results in the actual action $p \in P$ to be performed at the environment.

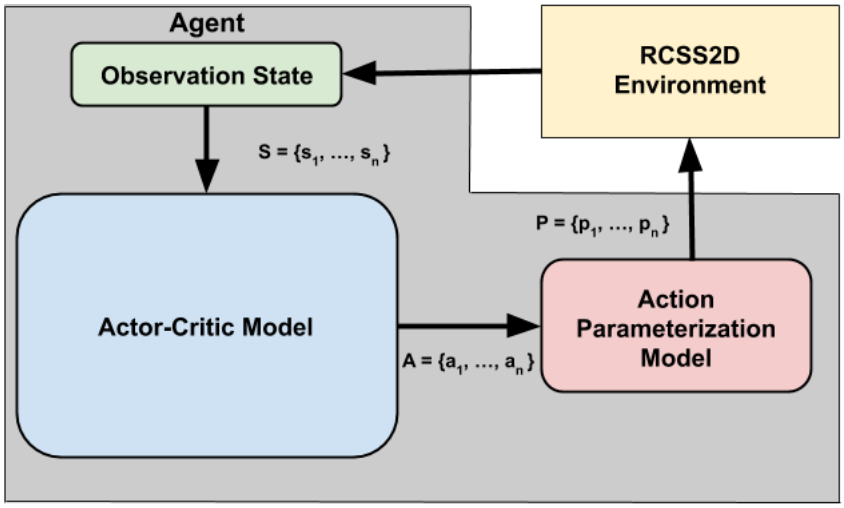



**Figure 3.5:** Diagram of the agent architecture.

We establish three state-of-art Actor-Critic algorithms to be evaluated in this architecture, which are: Deep Deterministic Policy Gradient, Twin Dueling Deep Deterministic Gradient and Soft Actor-Critic. The Table 3.1 details the models configuration for each algorithm.

**Table 3.1:** Actor-Critic algorithms configuration.

| **Actor-Critic algorithm** | **Neural Network Topology** | **Hyper-parameters** |
|---|---|---|
| Deep Deterministic Policy Gradient | Actor:<br>[State:256:256:#Actions]<br>Critic:<br>[(State+#Actions):256:256:1] | Value Learning Rate: 0.0001<br>Policy Learning Rate: 0.00001<br>$\gamma$: 0.99 |
| Twin Dueling Deep Deterministic Gradient | Actor:<br>[State:256:256:#Actions]<br>Critic:<br>[(State+#Actions):256:256:1] | Value Learning Rate: 0.00003<br>Policy Learning Rate: 0.00003<br>$\gamma$: 0.99 |
| Soft Actor-Critic | Actor:<br>[State:256:256:#Actions]<br>Critic:<br>[(State+#Actions):256:256:1]<br>SoftQNet:<br>[(State+#Actions):256:256:1] | Value Learning Rate: 0.00003<br>Policy Learning Rate: 0.00003<br>SoftQ Learning Rate: 0.00003<br>$\gamma$: 0.99 |

With the definition of the general AC-PA architecture presented in this section, we combine the different statements of the components to personalize the agent structure to each soccer behavior that will be evaluated in this work. We can see in Table 3.2 the agent architecture composition separately by soccer behavior.

**Table 3.2:** AC-PA agent architecture configurations.

| Architecture Configurations | | | |
|---|---|---|---|
| Configuration | Soccer Behavior | Observation Space | Action Parameterization |
| 1 | Go-to-ball | $S^{gtb}$: 2 inputs | Directional Dash AP: 1 output |
| 2 | Go-to-ball | $S^{gtb}$: 2 inputs | Complete Dash AP: 2 outputs |
| 3 | Ball-to-goal | $S^{btg}$: 5 inputs | Low-level AP: 3 outputs |
| 4 | Ball-to-goal-with-opponent | $S^{bgopp}$: 7 inputs | Low-level AP: 3 outputs |
| 5 | Ball-to-goal-with-opponent | $S^{bgopp}$: 7 inputs | Mid-level AP: 4 outputs |
| 6 | Ball-to-goal-with-opponent | $S^{bgopp}$: 7 inputs | Mid-level+High-level AP: 4 outputs |

# 4

# Experiments and Results

In this chapter we describe how each proposal architecture behaves for each soccer task and action parameterization associated, comparing the agent behavior and training characteristics for the least complex to the most complex soccer activity. For that, the models were trained until reach an optimal policy, determined by the reward accumulation per training episodes. Also, the trained models were evaluated with behavior metrics, which analyzes the way that the trained agent behaves for that soccer policy.

## 4.1 Go-To-Ball Experiment

For the training configuration of this activity, the robot player was positioned with a static distance from the ball to maintain consistency in the training evaluation for the three methods. We established a timeout of 200 cycles to reset the episode if it doesn't get to the kickable area, that is the position necessary to kick the ball. As shown in Table 3.2, this policy has two architecture configurations with the same observation space and two Action Parameterizations.

### 4.1.1 Go-To-Ball with Directional Dash Action

The three Actor-Critic models, DDPG, TD3, and SAC just needed 250 episodes of training to converge for the Go-To-Ball with Directional Dash AP architecture. As shown in Figure 4.1, the three Actor-Critic methods reached the optimal policy with some similarity, where the optimal policy is defined as the convergence point with the highest reward per episode. The SAC method arrived at the highest reward accumulation per episode more quickly than the others, demanding less than 50 episodes.

With the optimal policy of each algorithm, we have played the trained agent 1000 epochs for each method having the same environment configuration as the training, putting the agent at a static distance from the ball, and letting it perform the requested action chosen by the model. The Figure 4.2 shows the agent position distribution, representing the agent cartesian coordinate amount for each cycle played and the center of the plot is where the ball was located. Although the SAC method has reached the highest reward accumulation more quickly in the training part,

its trained agent behavior gets non-intuitively soccer performance for the go-to-ball policy, as shown on the figure the agent surrounds the ball before going to it, different from the TD3 and DDPG, which have a most direct path to the ball.

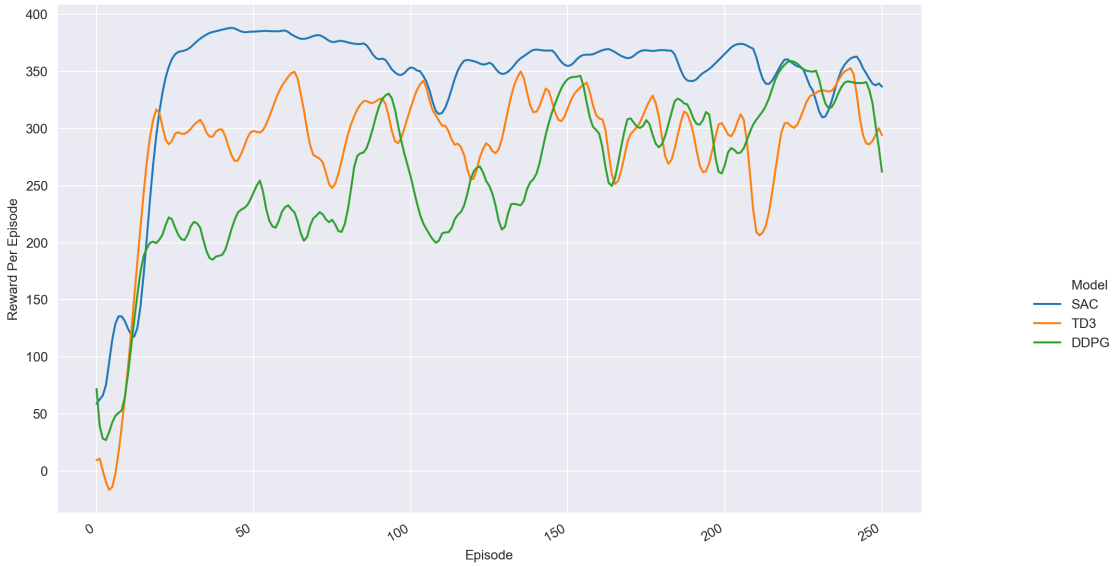
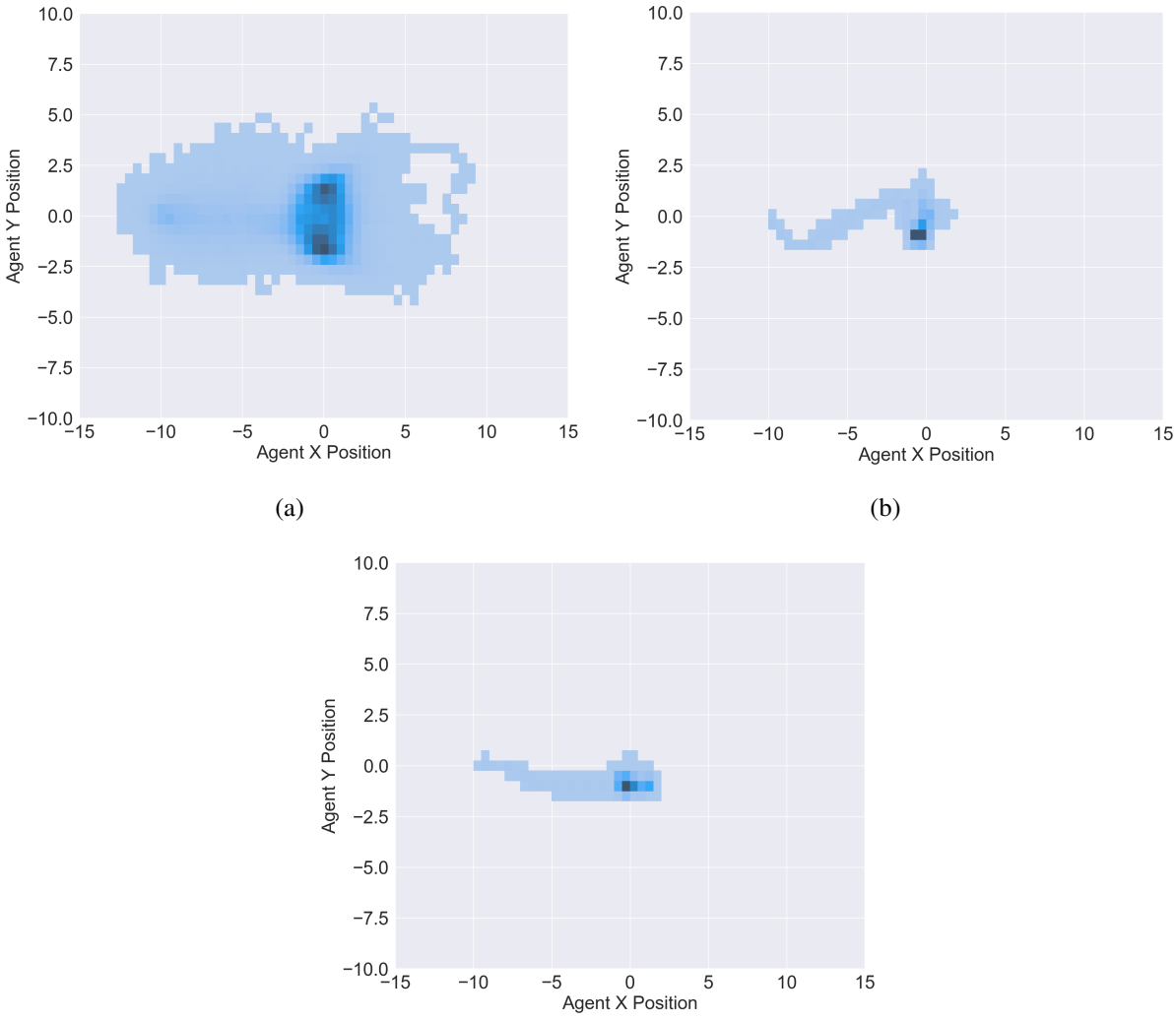



**Figure 4.1:** Go-To-Ball with Directional Dash Action, policy training reward per episodes




(a) (b)

(c)

**Figure 4.2:** Trained agent position distribution by performing Go-To-Ball with Direction Dash Action. The ball is fixed in the center of the plots. a) distribution for SAC method; b) TD3 method; c) DDPG method.

### 4.1.2 Go-To-Ball with Complete Dash Action

The complexity increasing of the action parameterization caused a different training dynamic, as shown in Figure 4.3, the SAC method approaches the optimal policy more slowly than the TD3 and DDPG. Also, it took more time of training: 5000 episodes for each algorithm.

We also played 1000 epochs for each trained model as in Directional Dash agent trained analyzes. The agent position distribution in Figure 4.4 shows a similar behavior as shown in the Directional Dash experiment with the difference that the SAC method this time concentrates the surroundings in the initial position of the agent and then goes to the ball.

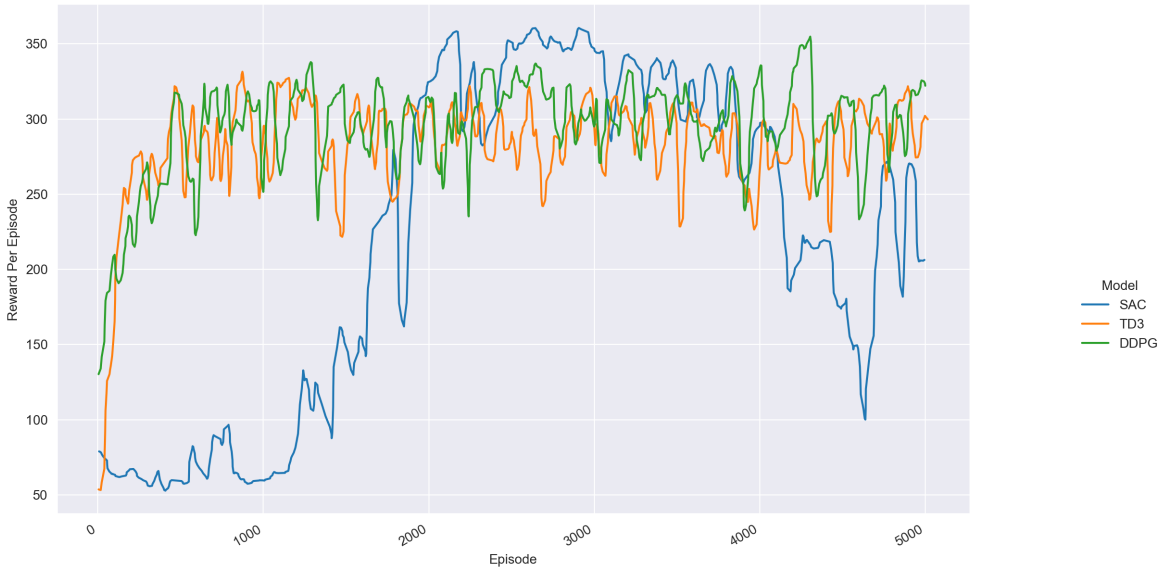



**Figure 4.3:** Go-To-Ball with Complete Dash Action, policy training reward per episodes

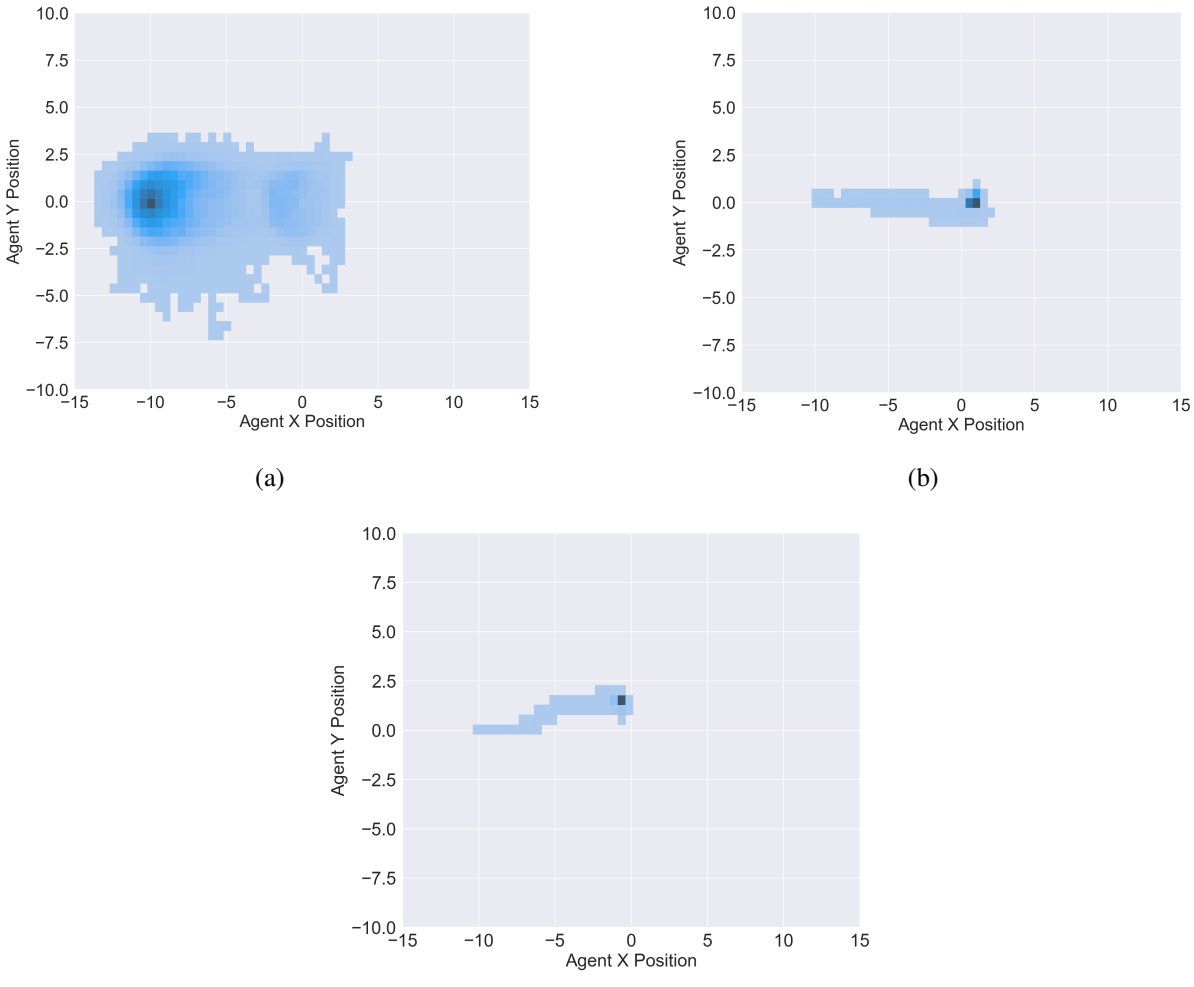



**Figure 4.4:** Trained agent position distribution by performing Go-To-Ball with Complete Dash Action Parameterization. The ball is fixed in the center of the plots. a) represents the SAC method; b) the TD3 method; c) the DDPG method.

## 4.2 Ball-To-Goal Experiment

The ball-to-goal experiment was made training the three Actor-Critic methods with the ball-to-goal action parameterization extended architecture. We trained with the HAUSKNECHT; STONE (2016) reward function and the proposed approximation ball-to-goal reward function. The environment was configured with the robot starting at a static distance from the ball and both at a static distance from the goal. At the end of the training the robot should be capable to approach the ball and take it to the goal.

Figure 4.5 shows the dynamics of the reward function during training session for HAUSKNECHT; STONE (2016) reward function. The three algorithms had positive and negative fluctuations in the reward accumulation per episode, without achieving a clear optimal policy.

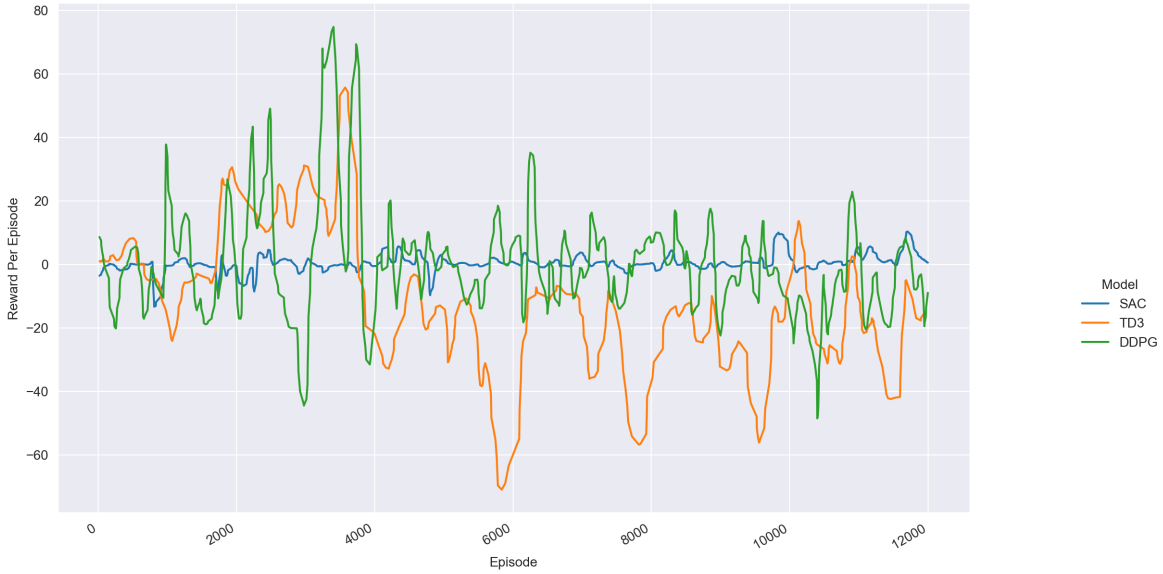



**Figure 4.5:** Ball-To-Goal with Low-level Actions, HAUSKNECHT; STONE (2016) policy training reward per episodes.

The training session of the proposed approximation go-to-ball reward function shown in Figure 4.6 demonstrates that it was possible to arrive at an optimal policy for one of the algorithms, which in this case was the SAC, the others TD3 and DDPG demonstrated fluctuations as the previous hfo reward function training.

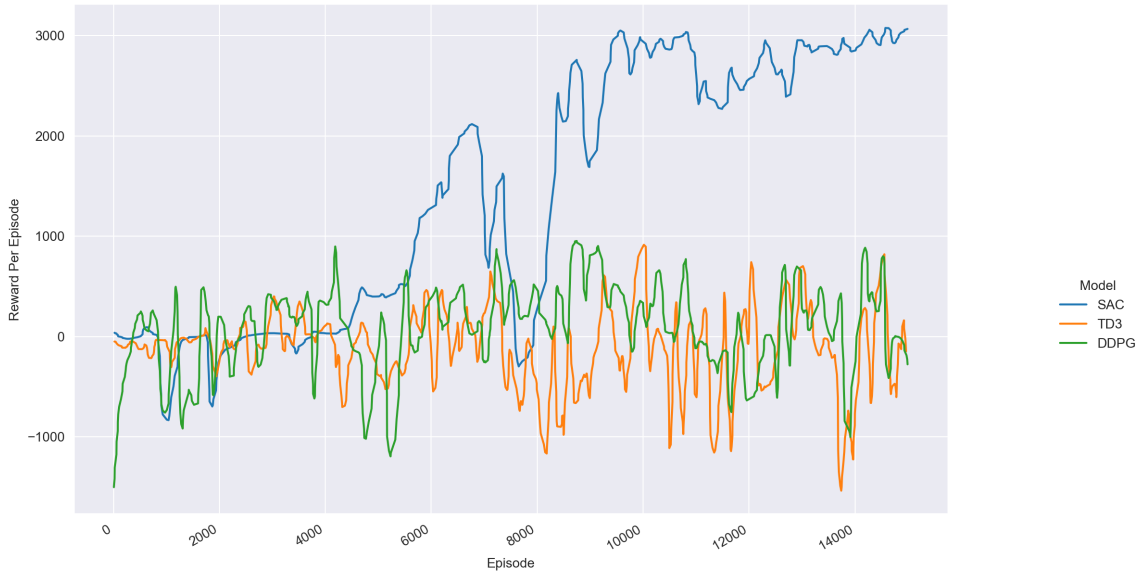



**Figure 4.6:** Ball-To-Goal with Low-level Actions, proposed policy training reward per episodes.

After training, the models were analyzed in game, with the environment configuration again having the robot in an initial position with static distance from the ball and both with static distance from the goal. We played 1000 sessions with a timeout of 200 cycles each or until scored.

The trained models resulted from the HAUSKNECHT; STONE (2016) reward function demonstrated a dispersed behaviour as shown in Figure 4.7. The plots at Figure 4.7 represents, at the left column, the agent position distribution for each cycle played and, at the right column, the ball position distribution for each cycle played. At the top (a) and (b), the trained SAC agent behavior showed a concentration in the initial spot moved to the ball and kicked to a random direction each time. In the middle (c) and (d), the trained TD3 agent moved more dispersed while carrying the ball and kicked the ball with some degree of dispersion but concentrating the ball position near to the goal. In the bottom (e) and (f), the trained DDPG agent learned to reach the ball and stay there.

Analyzing the trained model for the proposed approximation reward function, Figure 4.8 shows a more direct behavior in relation to ball-to-goal soccer policy. On the left column represent the agents positioning distribution and on the right side the ball positioning distribution for each test cycle played. In the first sample at the top (a) and (b), the SAC agent model demonstrated a tendency to follow a straight line carrying the ball to the goal. TD3 agent model in the middle (c) and (d), we had the player carrying the ball to a certain point and kicking towards, where in the distribution of the ball positioning it can be seen clearly that the ball follows a trajectory with a tendency to go to the goal. The model with DDPG at the bottom (e) and (f) showed a tendency of the player to go towards the goal, but it leaves the ball intact in the same position, which indicates that the model had difficulty choosing the kick action.

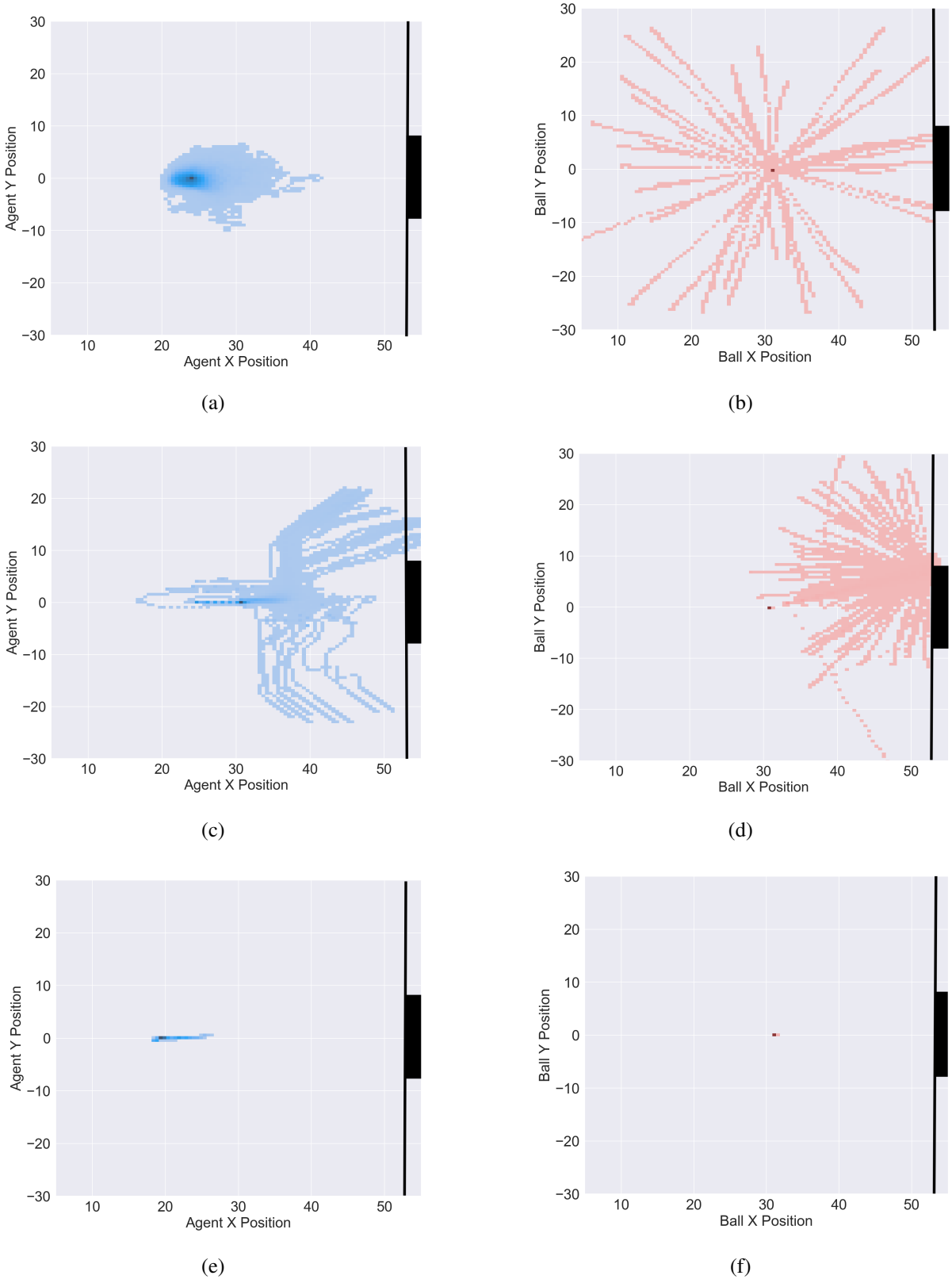



**Figure 4.7:** Trained agent position distribution and ball position distribution in the field by performing ball-to-goal task with Low-Level Action Parameterization and the HAUSKNECHT; STONE (2016) reward function.
a) and b) respectively represents the SAC agent position distribution and ball position distribution;
c) and d) respectively represents the TD3 agent position distribution and ball position distribution;
e) and f) respectively represents the DDPG agent position distribution and ball position distribution.
The goal is placed in the right position of each plot.

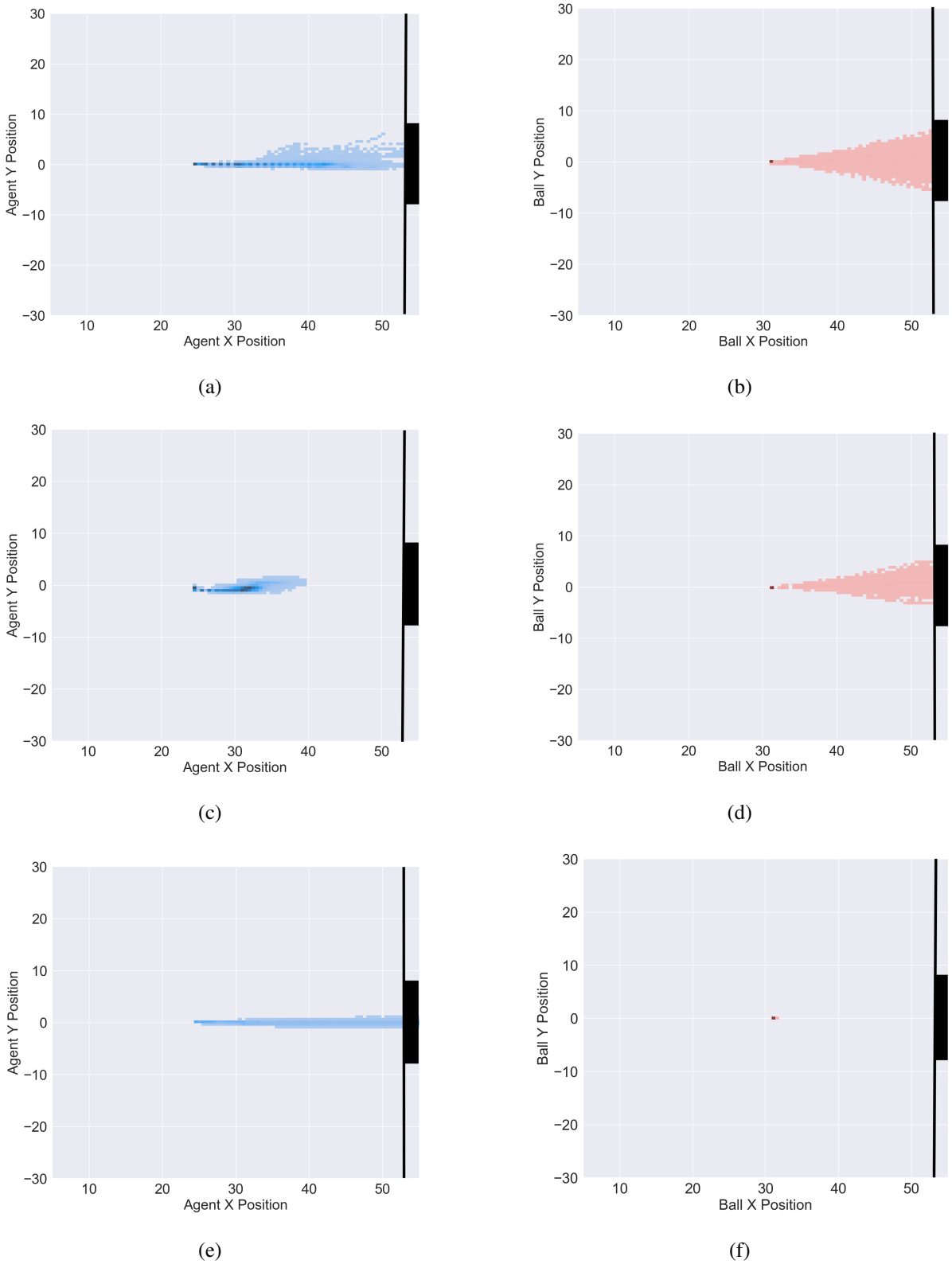



**Figure 4.8:** Trained agent position distribution and ball position distribution in the field by performing ball-to-goal task with Low-Level Action Parameterization and the proposed reward function.
a) and b) respectively represents the SAC agent position distribution and ball position distribution;
c) and d) respectively represents the TD3 agent position distribution and ball position distribution;
e) and f) respectively represents the DDPG agent position distribution and ball position distribution.
The goal is placed in the right position of each plot.

## 4.3 Ball-To-Goal-With-Opponent Experiment

The ball-to-goal-with-opponent adds a higher level of complexity compared to ball-to-goal soccer task, in this experiment it was included a goalkeeper to the environment, the goalkeeper has a standard hand-coded RCSS2D implementation. The settings of the training environment, as well as the ball-to-goal experiment, the player and the ball were placed in a certain position apart from each other and distant from the goal, also, the goalkeeper was placed in the middle of the goal, similar to a soccer penalty situation. The training session was done until the models converged, just like in previous experiments.

We performed three experiments with differents action parameterizations: the first was the low-level parameterization, the second was the mid-level parameterization and the third was the mixed parameterization with mid and high level. For the first two parameterizations, low-level PA and Mid-level PA), the model did not obtain good results, neither converging to a optimal policy nor behaving coherently with trained models. The last parameterization model that mixes the use of RCSS2D's mid-level and high-level actions managed to converge to an optimal policy, as shown in Figure 4.9. The architecture with the SAC method was the only model that converged to a optimal policy for this experiment with the Mid-level+High-level PA.

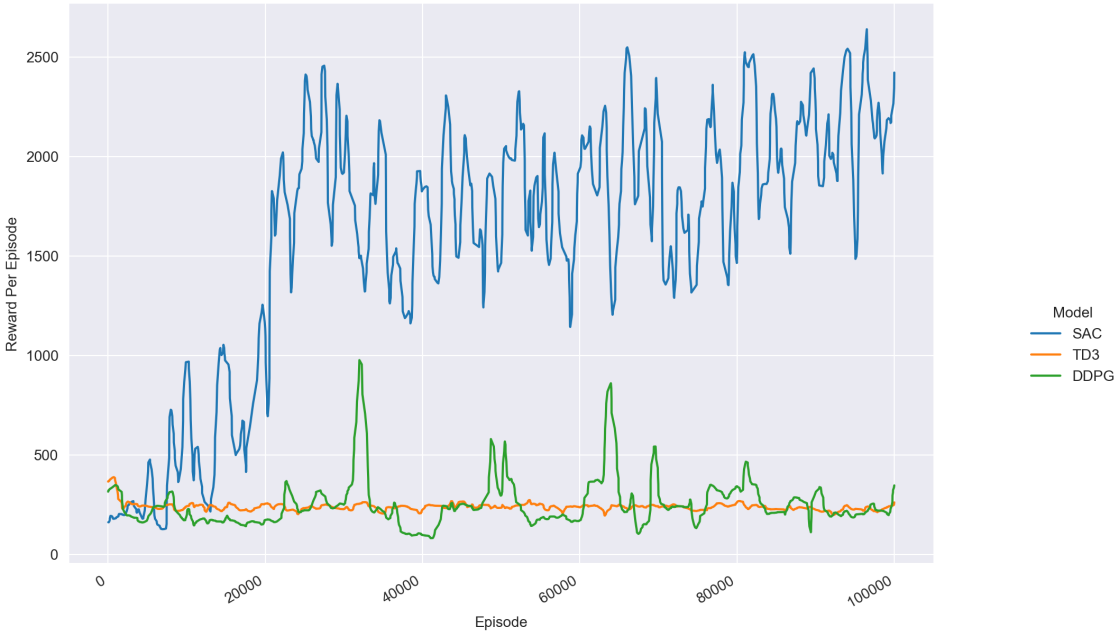



**Figure 4.9:** Ball-To-Goal-With-Opponent with Mid/High Level Actions, policy training reward per episodes

The tests of the trained models were performed by playing 1000 sessions with the same environment settings used at the training session. In the Figure 4.10 we have the graph of the player positioning distribution in the left column and the ball positioning distribution in the right column. As well as the results of the training, the SAC model demonstrated a significant

difference of behavior, consistent to the soccer policy adopted for this experiment, where the player carries the ball towards the goal, having an expansion in the curvature of the trajectory relative to the defense of the goalkeeper added to the experiment.

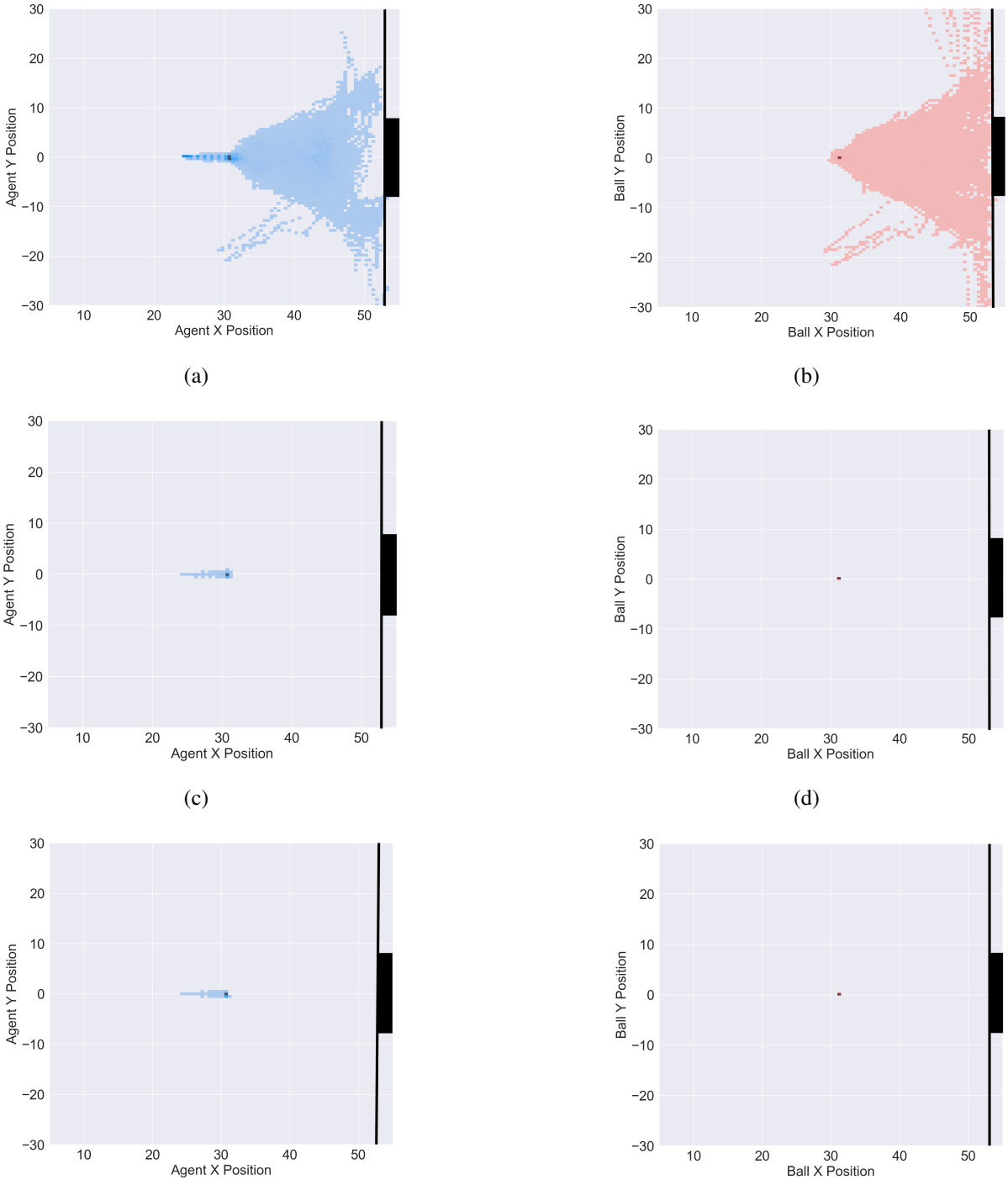



**Figure 4.10:** Trained agent position distribution and ball position distribution in the field by performing ball-to-goal-with-opponent task with Mid-level+High-level Action Parameterization. a)and b) respectively represents the SAC agent position distribution and ball position distribution; c)and d) respectively represents the TD3 agent position distribution and ball position distribution; e)and f) respectively represents the DDPG agent position distribution and ball position distribution. The goal is placed in the right position of each plot.

The last comparison was made getting the ball-to-goal-with-opponent SAC trained model with mid-level+high-level action parameterization, playing in 1000 penalty game sessions, and completing the same 1000 session for a RCSS2D offensive agent with hand-coded implementation. The results can be seen in the Table 4.1, the DRL model get a lower percentage of goal compared with the hand-coded implementation and it takes more cycles to achieve the goal, however, we demonstrated that it is possible to implement an offensive player only with a DRL approach.

**Table 4.1:** Goal percentage and average cycles to score resulted from the SAC Offensive Player and Hand-coded Offensive Player in test round of 1000 episodes.

| Offensive Player Algorithm | Goal Percentage | Average Cycles to Score |
|---|---|---|
| SAC Offensive Player | 68.9% | 67 |
| Hand-coded Offensive Player | 97.8% | 55 |

# 5

# Conclusion and Future Works

RoboCup Soccer Simulator 2D (RCSS2D) is a multifaceted environment which provides several set of complex soccer problems with numerous observation and action elements for each type of situation. This dynamism, the agent observation-action configuration, and the stochastic factor of the environment demands a Deep Reinforcement Learning (DRL) model with a higher power of exploration.

We evaluate three state-of-art Actor-Critic (AC) architectures extended through different sets of parameterized action space for RCSS2D which confirms empirically soft actor-critic algorithm (SAC) as an outperformed architecture besides other AC methods as TD3 and DDPG. The results show that the three methods converge to an optimal policy with some similarity for simple tasks and observation-action configuration like the go-to-ball problem with the unique action-parameter output. As the complexity of the task and action parameterization increases the SAC algorithm exceeds in reaches the optimal soccer objective, pointing to the great potential of this method exploration.

The higher the complexity of the soccer task gets the more exploration is needed for the model to performs actions with precision. The results demonstrate that the combination of SAC model with a solid parameterized mid/high-level action for the RCSS2D allows the algorithm to converge to the optimal policy more quickly for tasks with higher dynamism and complexity as taking the ball to the goal with the goalie interference (ball-to-goal-with-opponent).

For the soccer tasks defined in this work - go-to-ball, ball-to-goal, and ball-to-goal-with-opponent - is necessary an accurate reward function to be effective using DRL. We present three reward functions to performs the three tasks which manifest significant results comparing with other reward functions in related works. With the ball-to-goal-with-opponent model, we define an offensive robot soccer player algorithm using DRL that can be extended to other robot soccer environments besides RCSS2D - although, for the training set made in this work, this offensive model doesn't reach a better result in real RCSS2D game comparing with a base hand-coded offensive robot player.

We demonstrated the possibility to build an agent that performs complex tasks in a robot soccer environment using Actor-Critic architecture and solid continuous action space

parameterization. Which the SAC architecture reached the best policy system due to its capacity of exploration.

Since the SAC method had better results, which is a method that focuses strictly on exploration, for future works the focus is going to be on study the exploration factor of the architectures. This will be made by conducting experiments for different types of configuration in the exploration factors and modifying the granularity of the parameterization spaces of the actions types. Additionally, we will make a precise investigation on the proposed reward functions and include more features to the model, as game cycle count and agent energy. Also, the evaluation should be expanded to other tasks in the soccer environment, like the formulation of a defensive player, goalie, and expand the offensive capacity with the inclusion of pass task and multi-agent management.

References

BELLMAN, R. A Markovian decision process. **Journal of mathematics and mechanics**, p.679–684, 1957.

Chen, G. et al. Robot Navigation with Map-Based Deep Reinforcement Learning. In: IEEE INTERNATIONAL CONFERENCE ON NETWORKING, SENSING AND CONTROL (ICNSC), 2020. **Anais** 2020. p.1–6.

CHEN, M. et al. **RoboCup Soccer Simulator Documentation**. 2003. https://rcsoccersim.github.io/manual/soccerserver.html.

FUJIMOTO, S.; HOOF, H. van; MEGER, D. Addressing Function Approximation Error in Actor-Critic Methods. In: 2020 , Stockholmsmässan, Stockholm Sweden. **Anais** PMLR, 2018. p.1587–1596. (Proceedings of Machine Learning Research, v.80).

GU, S. et al. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: ICRA. **Anais** IEEE, 2017. p.3389–3396.

HAARNOJA, T. et al. Soft Actor-Critic Algorithms and Applications. **CoRR**, v.abs/1812.05905, 2018.

HAARNOJA, T. et al. **Soft Actor-Critic**: off-policy maximum entropy deep reinforcement learning with a stochastic actor. cite arxiv:1801.01290Comment: ICML 2018 Videos: sites.google.com/view/soft-actor-critic Code: github.com/haarnoja/sac. http://arxiv.org/abs/1801.01290.

HAUSKNECHT, M. **RoboCup 2D Half Field Offense**. 2015. https://github.com/LARG/HFO.

HAUSKNECHT, M. J.; STONE, P. Deep Reinforcement Learning in Parameterized Action Space. In: ICLR (POSTER). **Anais** 2016.

LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. **arXiv preprint arXiv:1509.02971**, 2015.

MASSON, W.; RANCHOD, P.; KONIDARIS, G. D. Reinforcement Learning with Parameterized Actions. In: AAAI. **Anais** AAAI Press, 2016. p.1934–1940.

MNIH, V. et al. Playing Atari with Deep Reinforcement Learning. **CoRR**, v.abs/1312.5602, 2013.

MNIH, V. et al. Asynchronous Methods for Deep Reinforcement Learning. In: AAAI PRESS, New York, New York, USA. **Anais** PMLR, 2016. p.1928–1937. (Proceedings of Machine Learning Research, v.48).

ROBOCUP. **RoboCup Soccer Simulation Server**. 2018. https://github.com/rcsoccersim/rcssserver.

ROBOCUP. **RoboCup Federation**. 2019. https://www.robocup.org/.

SILVER, D. et al. Deterministic Policy Gradient Algorithms. In: ICML. **Anais** JMLR.org, 2014. p.387–395. (JMLR Workshop and Conference Proceedings, v.32).

SILVER, D. et al. Mastering the game of Go without human knowledge. **Nature**, v.550, p.354–, Oct. 2017.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning**: an introduction. MIT Press, 2011.

SUTTON, R. S. et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: INTERNATIONAL CONFERENCE ON NEURAL INFORMATION PROCESSING SYSTEMS, 12., Cambridge, MA, USA. **Proceedings** MIT Press, 1999. p.1057–1063. (NIPS'99).

WANG, Z. et al. Dueling Network Architectures for Deep Reinforcement Learning. In: MIT PRESS, New York, New York, USA. **Anais** PMLR, 2016. p.1995–2003. (Proceedings of Machine Learning Research, v.48).

ZARE, N. et al. **Cyrus 2D Simulation 2019 Team Description Paper**. RoboCup 2019: Robot World Cup XXIII, 2019.