Playing Soccer with Deep Reinforcement Learning

Francisco Nuno Dias dos Santos Delgado

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha
Member of the Committee: Prof. Pedro Alexandre Simões dos Santos

November 2021

Acknowledgments

I would like to start by thanking my supervisor Alberto Sardinha, for his insight and help throughout this thesis. His guidance pushed and made it possible to overcome many of the obstacles faced.

I want to thank my friends and colleagues that helped me through the development of this thesis and over the course of my academic life, being there for me whenever I needed them, always keeping me motivated. Their help was invaluable.

Lastly, I want to thank my parents and my brother as they taught me so much, academically and personally throughout my life. The family support was crucial to complete my degree since they were encouraging and trusting my skills and pushing me to give my best.

This would not be possible without all of you, thank you.

Abstract

In this thesis, we use Deep Q-Networks to learn a policy in the Half Field Offense environment, where we have to work with teammates and score goals against a team of opponents. At the moment, there is no research performing an extensive analysis on changing various aspects like what features to use (type of state space), as an example, so we have proceeded with this research, ending with a discussion on what works best in this environment. Next, we noticed that most solutions use as a metric for their success the percentage of episodes ending in goal. We wanted to be sure why that happened. This thesis expands on this idea of examining performance to a deeper level, where we test why the team starts scoring more goals, which team member did more in that regard. To achieve this we ran our trained agents and gathered metrics regarding the number of goals each player scores, how many passes our agent does, and how many assists our agent has. Afterward, changing our teammates' strategy, seeing if the trends stay the same. Finally, switching our agent with an NPC, to compare what kind of results were obtained with and without our agent, this way assessing if our agent has a positive effect. Our results conclude that our agent at the worst performs the same as an NPC, but in most cases, he scores a large number of goals and dramatically improves the team's goals.

Keywords

Reinforcement Learning; Artificial Intelligence; Deep Reinforcement Learning; Half-Field Offense;

Resumo

Nesta dissertação, usamos *Deep Q-Networks* para aprender uma política no ambiente *Half Field Offense*, onde temos de trabalhar com colegas de equipa para marcar golos contra uma equipa de oponentes. De momento, não existe nenhum estudo que faça um análise extensiva sobre a mudança de vários elementos como mudar o estado (tipo de estado de espaço) por exemplo, assim sendo fazemos esta pesquisa, acabando como uma discussão sobre o que funciona melhor neste ambiente. Seguidamente, notámos que grande parte das soluções, usam como métrica para o sucesso a percentagem de golos. Embora tenham bons resultados, queríamos saber ao certo o motivo para tal. Esta dissertação expande esta ideia de examinar a performance a um nível mais profundo, em que testamos o porquê da equipa estar a marcar mais golos, verificando qual o jogador que marca mais. Para conseguir isto, corremos os nossos agentes treinados e acumulamos métricas sobre o número de golos que cada jogador marca, quantos passes e assistências o nosso agente faz. De seguida, trocámos o tipo de colega de equipa, analisando se a tendência notada se mantém. Finalmente, trocámos o nosso agente com um *NPC*, para comparar que resultados são obtidos com e sem o nosso agente, desta forma, concluindo se o nosso agente tem um efeito positivo. Com os resultados concluímos que o nosso agente no pior dos casos tem o mesmo efeito que um *NPC*, mas em grande parte dos casos ele marca uma grande parte dos golos e aumenta dramaticamente os golos da equipa.

Palavras Chave

Aprendizagem por reforço; Inteligência Artificial; Deep Reinforcement Learning; Half-Field Offense;

Contents

List of Figures

List of Algorithms

Acronyms

DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
FVI	Fitted Value Iteration
HFO	Half Field Offense
MC	Monte Carlo
MDP	Markov Decision Process
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

1

Introduction

Contents

1.1 Motivation

Over the last few years, the number of autonomous agents, either robotic or virtual, has been increasing exponentially, mainly due to the increase in processing power. As the need for these agents to execute more and more activities arises, the ability for these agents to perform well in whichever environment also increases. Given this, it is desirable for agents to be capable of learning any specific environment, be versatile, and capable of working with other agents to achieve a common goal and to have the best possible performance.

Robocup competition is one of these situations where a team of agents must work together and learn the best possible strategy to succeed in this competition. In this competition, multi-agent systems must work together to win a game of soccer [2]. This domain has been used extensively as a testbed for research regarding Reinforcement learning, multi-agent systems, and artificial intelligence development. The solutions to this domain have to deal with continuous state space, noisy actions, and multiple other agents in play, so a good and robust solution has to be created. This makes it a good domain to test new approaches since many of these challenges exist in more complex problems. So, continuing to improve on this aspect might make it viable the use of agents or robots in more delicate problems where they have to team up with humans or other agents [3] for example in helping during catastrophes.

In many of these problems, it is not only important the outcome of the problem, but also what was done to achieve it, analyzing what needs to be improved at a deeper level, to better fine-tune the agents' or robots' actions. With this in mind, our work introduces this new idea of scrutinizing the agent's actions in the Half Field Offense (HFO) environment, to examine the reasons behind the outcome.

1.2 Problem Description

This thesis focus on the problem of learning in the HFO environment, where we have our agent working with 1 or 2 teammates, and his team tries to score a goal, and a defending team consisting of 2 or 3 players tries to defend. The opposition consists of players using the *Helios* strategy, while our teammates use one of three possible strategies: *Helios, Autmasterminds, Agent2d*.

To the best of our knowledge, no extensive testing on the various elements of a Deep Q-Network in the HFO environment has been done, we start our thesis by doing this extensive analysis. Comparing the results when changing different aspects of learning, like adapting what features are given to the agent or testing different networks - this is explained in more detail in Chapter 4.

Most of the solutions created are tested based solely on the percentage of goals the team manages to score. We wanted to expand on this idea of testing the solutions, to analyze on a more granular level. We propose a new way of testing performance by seeing who scores more goals and plays better.

So, in this thesis, we expand on the research done on HFO when using a Deep Q-Network by testing

the impact of changing various elements, and develop the idea of a more deep analysis, and examine in detail what each teammate does regarding the team's performance as a whole.

1.3 Contributions

The main contributions of this thesis are:

1. Extend on previous work using discrete action spaces, which have no parameters to be selected, in the Half Field Offense;
2. Test the effect of changing the features used, changing reward function, using Fullstate flag, etc have on the learned policy of the Deep Q-Network. Since this is a complex and challenging domain we test in more detail these different elements;
3. Introduce a new way of in-depth analysis, to better demonstrate the impact that our agent has in comparison to his teammates. Examining who scores more goals, if our agent passes or shoots more and how many goals come from an assist from our agent;
4. Extend on this in-depth analysis to other types of teammates, to test if our agent always learns to do the same actions or if he adapts according to his teammates.

1.4 Document Outline

First, in Chapter 2 we provide an overview of the background of the main themes of this thesis. In Chapter 3 we talk about the existing work related to the problem in this thesis. The implementation done in terms of architecture, hyperparameters, and the performed tests are explained in Chapter 4. In Chapter 5 we show the results obtained and discuss its differences. Lastly, in Chapter 6 we conclude and provide a brief summary of the thesis and the future work plus its limitations

2

Background

Contents

In this chapter, we present some background knowledge related to the main topics of this thesis. We explain neural networks and basic reinforcement learning topics and algorithms. Followed by an explanation of a Deep Q-Network and how this is used to learn an environment. Finally, we explain the HFO environment used and where it originated.

2.1 Neural Networks

Neural networks are a subset of Machine Learning, that accomplishes a certain task by passing data through units. Each unit takes multiple inputs and outputs a single value, which is a weighted sum of its inputs. An activation function is then applied to the output, and this output is passed to all following units, in the case of fully connected layers. Each neural network is comprised of an input layer that takes the input, passing it to the next layer, and of an output layer that outputs the value of the network. Between these layers may be various hidden layers. Each of these layers is composed of multiple units. An example can be seen in Figure 2.1.

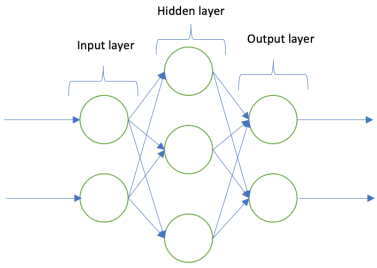


Figure 2.1: Neural network example, where the data flows from the left layer to the right layer

A neural network learns by optimizing a metric, that is normally given by a loss function like Mean Squared Error. The loss function informs the network of the error between the expected output and the real output. Then this value is backpropagated [4] through the network, starting from the output layer to the input layer, where every weight will be updated using gradient descent to try to minimize the loss function or, in other words, approximate the value that was output to the expected output.

Some important aspects of Neural Networks affect the learning and need to be considered. One aspect is the type of weight initialization, i.e. the initial value for the weights of the network, the most simple example is a random initialization where the weights are assigned random values. Another important aspect is the learning rate, which dictates how much the weights are adjusted between training steps, where a value too low will increase training time and too high might cause instability in the learning and harm performance. We use He initialization [5] for our network weights, this type of initialization is used mainly for Rectified Linear Unit (ReLU) activated layers.

2.2 Markov Decision Process

A Markov Decision Process (MDP) is defined by a tuple *(S, A, P, R,* γ*)* where:

- *S* is a set of possible states called state space;
- *A* is a set of actions that the agent can execute in any given state s called action space;
- *P* contains the probabilities of transition $P(s_{t+1}|s_t, a_t)$ from state s_t to state s_{t+1} given an action a_t;
- *R* contains the immediate rewards $R(s_t, a_t)$ from performing action a_t in state s_t.
- γ is the discount factor.

At each timestep t, the agent receives the current state s_t, selecting an action a_t to execute. The environment then returns a reward r_t and transitions to a next state s_{t+1}. This interaction is illustrated in Figure 2.2.

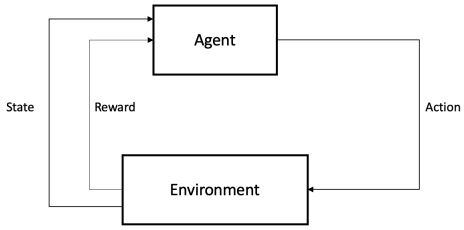


Figure 2.2: Interaction between agent and environment

So, the main objective of solving an MDP is to find a policy that maximizes the received cumulative rewards that are received during the course of interacting with the environment. So, by solving this MDP, we will obtain an optimal policy π^* that maps states to actions, such that this policy will maximize the expected discounted sum of rewards received. An MDP must also respect the Markov property, which expresses that each state only depends on the state and action selected at the previous timestep, being independent of all other previous states and actions.

2.3 Reinforcement Learning

In this thesis, we use Reinforcement Learning (RL) [6] where an agent tries to maximize the cumulative received reward by learning a policy, without being told what actions to execute. Considering a single agent within an environment, RL is an approach to find an optimal policy of an MDP where one does not know the transition probabilities of the states (i.e. the dynamics of the environment) and the reward function (i.e. the rewards received from the environment).

The main objective of the agents is to learn an optimal policy π^*, that dictates at each state s_t what action a_t will yield the highest reward. This agent will interact with the environment learning what actions give the best rewards, slowly learning the value of each state and which actions to execute. To calculate the value of each state, the agent also has to take into account the future rewards he will receive starting from that state. So, a value of a state s is the cumulative received reward from starting at state s and following policy π thereafter. The importance of the received rewards loses value the further into the future they are received, for example, a reward x is preferred in the next timestep than in 10 timesteps. So the agent's goal is to maximize the reward in the long run but favoring short-term actions. For this, we have discounted rewards using parameter γ, which is between [0, 1]. The lower the factor the more short-sighted the agent will be and as it increases the more important the future rewards are for the current decision.

The value function is defined as the expected discounted reward achieved by a policy π starting at state s_t and following π. To find this value we calculate the expected sum of received rewards, discounted by γ as seen in the following equation:

$$V^{\pi}(s_t) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{2.1}$$

The Q-value, which is the value of a pair state-action (s_t, a_t) that selects action a_t in state s_t and afterward follows policy π. We sum the reward r_t of performing action a_t in state s_t with the value function of state s_{t+1}, discounted by γ, as calculated in:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}[r_t + \gamma V^{\pi}(s_{t+1})] \tag{2.2}$$

There are two main algorithms to learn these policies, Q-Learning and SARSA.

2.3.1 Q-Learning

Q-learning [7] is an off-policy algorithm which means that the policy used to collect samples might be different from the optimal policy (π^*), in other words, it learns the optimal policy independently of the agent's actions.

$$Q^{\pi}(s_t, a_t) \rightarrow Q^{\pi}(s_t, a_t) + \alpha\Big(r_t + \gamma \max_{a \in A} Q^{\pi}(s_{t+1}, a) - Q^{\pi}(s_t, a_t)\Big) \tag{2.3}$$

It uses (2.3) to update its estimates of the Q-value for each pair state-action. At each time step *t*, the agent selects an action a_t, receiving a reward r_t and moving from state s_t to state s_{t+1}. The value α is the learning rate which is between [0,1] which dictates how important the new data is for the learning agent. While $\max_{a \in A} Q^{\pi}(s_{t+1}, a)$ is the maximum Q-value that the agent can obtain by executing an

action a in state s_{t+1}. So the objective of this algorithm is to find for each pair, state s and action a, an optimal Q-value ($Q^*(s,a)$), to find the optimal policy (π^*). We know that if every pair state-action is visited infinitely often the policy will converge to the optimal. So, to visit every pair state-action, we have to explore new pairs instead of always following the learned policy. So, we have to try to find a balance between exploring the environment and also exploiting the learned policy. In order to improve the exploration of the environment, we can use an ϵ-greedy strategy, where our agent explores by doing a random action with ϵ probability and exploits the policy (i.e. selects the action that has the highest Q-value) with 1-ϵ probability. An improvement to this is the ϵ-annealing strategy that starts with a high value for ϵ but slowly decreases it, so the agent starts by exploring more, but as he starts to learn an optimal policy he starts exploiting more.

2.3.2 SARSA

The SARSA algorithm [8] is another algorithm to learn an MDP policy, which is very similar to Q-learning, but instead of the off-policy approach of Q-learning, this algorithm is on-policy. This means SARSA, unlike Q-learning, learns by following its current policy instead of the greedy-policy. The update function reflects this difference:

$$Q(s_t, a_t) \rightarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big) \tag{2.4}$$

2.3.3 Deep Q-Network

When dealing with an MDP that has a state space with continuous features, the previous algorithms are not recommended. It is necessary to use a function approximation method that attempts to estimate an unknown function from available observations from the environment, while also being capable of learning with continuous state features. One example of such an approach is a Deep Q-Network (DQN) [1], where we combine the Q-learning aspect of reinforcement learning with a deep neural network. This means we have a deep network that receives the current state as input and outputs an estimated Q-value for each possible action that the agent can execute, an example can be seen in Figure 2.3. This approach is mainly used in challenging reinforcement learning domains where we have continuous state space but the action space is discrete, and the network learns to approximate Q-values.

The networks are parameterized by weights and bias denoted θ. The network learns to approximate the Q-value of a pair state-action *(s, a)*, based on the network parameters θ, this is denoted as *Q(s, a|θ)*. Additionally, the network can generalize similar states, so as to be prepared for new unobserved states if we have been in similar ones in the past, this is very useful if we have continuous state spaces.

Our Q-network then outputs a Q-value for each possible action and then chooses the action that

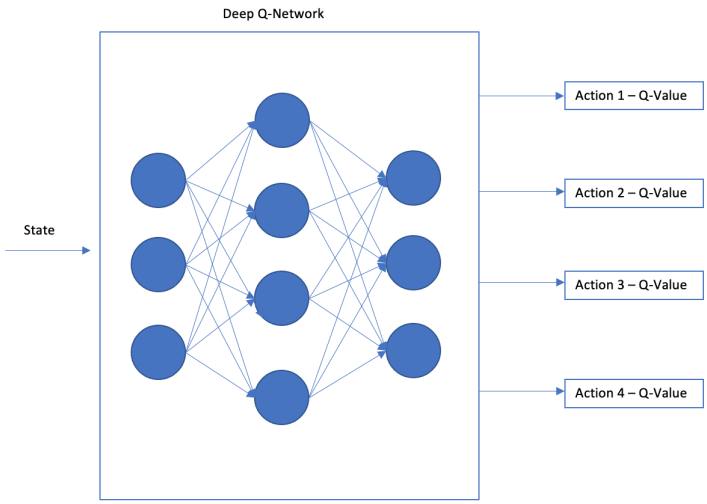


Figure 2.3: Deep Q-Network example

yields the highest Q-value, as we believe this will give us a better reward. There are some improvements to this architecture, where we use an additional network called target network that learns at a slower pace and is parameterized by weights and bias denoted as θ'. After each iteration, we only update one of them while keeping the other constant and after some time we synchronize both networks. The target Q-values used in the loss function are the ones calculated by the network that remains unchanged, the target network. When updating the Q-network, we want to minimize the loss function between the target Q-value and what we believe is the current Q-value. This is done by doing gradient descent on the following function:

$$L(s_t, a_t|\theta) = \mathbb{E}[(r_t + \gamma \max_{a \in A} Q'(s_{t+1}, a|\theta') - Q(s_t, a_t|\theta))^2] \tag{2.5}$$

The loss function is in (2.5), where the best Q-value in the next state ($\max_{a \in A} Q'(s_{t+1}, a|\theta')$) is calculated with the target network and the weights of that network (θ'). One of its advantages is that we have more stable updates based on the fact the network is not being constantly changed.

Another improvement is the replay buffer, where we store the received state and next state, the agent's action selection at each state, and their reward. Afterward, when training our agents we sample a batch of these experiences from the buffer, to mitigate the bias that comes from learning from sequential experiences. These improvements help in stabilizing the learned policy. The update target is not being constantly changed when we use a target network, and, in the case of a replay buffer, we keep learning from past experiences instead of just learning from new sequential actions.

So, with all of this, we can write the DQN algorithm as the following:

Algorithm 2.1: Deep Q-network Algorithm as presented by [1]

Data: γ

Initialize replay memory *D* to capacity *N*
Initialize Deep Q-Network with random weights θ
Initialize target network with weights $\theta' \Leftarrow \theta$
for episode = 1, *M* **do**
  Initialize sequence s_1 = x1
  for t=1, T **do**
    With probability ϵ select random action a_t otherwise use Deep Q-network and select action such that $a_t = max_a(Q(s_t, a|\theta))$
    Execute action a_t in simulator and observe reward r_t and next state s_{t+1}
    Store a new transition (s_t, a_t, s_{t+1}, r_t) in D
    Sample random minibatch of transition (s_j, a_j, s_{j+1}, r_j) from replay buffer D
    Set target $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma(max_a Q'(s_{j+1}, a|\theta')) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j|\theta))^2$ with respect to network parameters θ
    Every C steps reset $\theta' \Leftarrow \theta$

2.4 Half Field Offense

2.4.1 Robocup

Half Field Offense is a subtask of the Robocup 2D [2] competition. This competition is used as a platform for researching AI and machine learning, where multiple autonomous agents play a game of soccer between them. The teams must be able to cooperate with each other, while also avoiding losing the ball to the other team. There also exists the Robocup competition with robotic leagues, which uses physical robots to move.

2.4.2 Half Field Offense

In [9] they presented a novel subtask of Robocup, called Half Field Offense. This task is an extension of the previous simpler subtask of Robocup created in [10], called Keepaway, where a team must keep possession of the ball against a team that attempts to steal the ball. An open source[1] of HFO was released in [11], which is used for the implementation of the agents in this thesis. An example of this environment can be seen in Figure 2.4.

The HFO subtask emerged since it is an easier task to learn than the Robocup. In this task, an offense team must work together to score against a defending team that includes a goalie. We can define how many players will be in each team up to a maximum of 4vs5 players in the environment, where 4 players attack and 5 defend, in our work we use a 2vs2 and 3vs3 environment. This task is only

[1] https://github.com/LARG/HFO

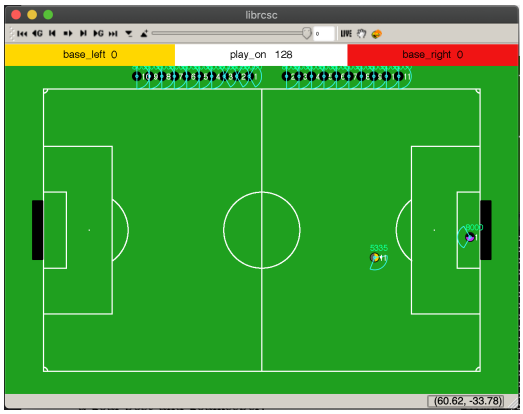


Figure 2.4: Half Field Offense Environment

played on half of a soccer field, from the half field line to the goal line, and each episode (each attacking play) ends when one of four events occur:

- **Goal** : A goal is scored;
- **Out Of Bounds** : The ball leaves the playing field;
- **Out of Time** : A certain number of timesteps have passed and the episode is ended;
- **Captured by defense** : The defense team catches the ball.

After one of these events occur, the episode is finished and a new attacking play is started, where the position of all players (attacking and defending) and the ball is randomized.

2.4.3 Action Space

There are three types of actions provided that each agent can use: the high-level, the mid-level, and the low-level. The high-level actions are discrete and not parameterized (*move, shoot, pass, dribble, catch, reduce angle to goal, defend goal, go to ball, mark player and reorient*) with the exception of the action ”pass” that needs the number of the teammate. In fact, behind each of these actions are parameters that must be chosen, but are automatically selected for the agent using a *Helios* Strategy [12]. The mid-level action space has discrete and parameterized actions (*kick to, move to, dribble to and intercept*). Finally, the low-level actions that are all parameterized, and the agents need to select the parameters to execute them (*dash, turn, tackle and kick*).

2.4.4 State Space

The state space used is one of the following: a high-level state space that is more compact, which uses fewer state features, but each one is more informative. In the case of a 2vs2 environment, we have

0 – X position
1 – Y position
2 – Orientation
3 – Ball X
4 – Ball Y
5 – Able to kick
6 – Goal Center proximity
7 – Goal Center angle
8 – Goal Opening angle
9 – Proximity to opponent
T – Teammate's goal opening Angle
T – Proximity of Teammate i to closest opponent
T – Pass Opening angle
3T – X, Y and Uniform Number of Teammates
3O – X, Y and Uniform Number of Opponents
+1 – Last action was successful
+1 – Stamina

(a) High-level features with T being number of teammates and O number of opponents

0 – Self Position Valid
1 – Self Velocity Valid
2-3 – Self Velocity Angle
4 – Self Velocity Magnitude
5-6 – Self angle
7 – Stamina
8 – Frozen
9 – Colliding with Ball
10 – Colliding with player
11 – Colliding with post
12 – Able to kick ball
13-15 – Goal center
16-18 – Goal Post Top
19-21 – Goal Post Bot
22-24 -- Penalty Box center
25-27 – Penalty Box Top
28-30 – Penalty Box Bot
31-33 – Center Field
34-36 – Corner Top left
37-39 – Corner Top right
40-42 – Corner Bot right
43-45 – Corner Bot left
46 – Out of bounds left line distance
47 – Out of bounds right line distance
48 – Out of bounds top line distance
49 – Out of bounds bottom line distance
50 – Ball Position Valid
51-52 – Ball angle
53 – Ball distance
54 – Ball Velocity Valid
55 – Ball Velocity Magnitude
56-57 – Ball Velocity Angle
8T – Various teammate features
8O – Various Opponents features
T – Teammate's uniform numbers
O – Opponents Uniform numbers
+1 – Last action was successful

(b) Low-level feature with T being number of teammates and O number of opponents

Figure 2.5: Both possible state spaces

24 features. A low-level state space, where we have much more features, where each one gives less important information. Features related to the agent's position and distance to various landmarks of the field, orientation, if his position is valid, etc, but also many features regarding all other players in-game. Making this state space much more informative. In the case of a 2vs2 environment, we have 86 features. Both these spaces are provided by the simulator. The complete state features are in Figure 2.5.

2.4.5 Teammate Strategies

We used three different teammate strategies: *Autmasterminds*, *Helios* and *Agent2d*. The *Agent2d* [13] strategy decides which actions to execute based on three classes. The agent class decides what strategy to run based on what role the agent thinks he has on the team. This decision is passed to the role class, that executes tactical behavior based on the selected strategy. Finally, the behavior class

executes the actual action based on the selected behaviors. The *Helios* [12] strategy is an extension of the agent2d agent, that uses a tree search to try to improve cooperation between teammates. It takes into account teammates' actions and positions, evaluates possible future states and actions and selects the actions that will achieve better results. The *Autmasterminds* agent [14], when it does not have possession of the ball, tries to position himself to receive a pass using Voronoi Diagrams, that selects the best and safest positions to receive a pass without losing possession of the ball. When in possession of the ball the agent decides between shooting, passing, and dribbling based on a task evaluator class, that based on the environment around the agent grades each action. For the task of dribbling, the agent treats the state space as grids on the field and gives a higher reward for the target grids. They then use RL to solve an MDP in the process the agent learns what actions to execute in order to dribble into higher value states.

3

Related Work

Contents

Over the years many approaches have been used to resolve this challenge of playing in Half Field Offense. In this HFO environment, authors have tried to create agents capable of scoring goals reliably either on an empty goal or against a goalkeeper. Agents that are capable of scoring goals when having opponents try to steal the ball and, lastly, agents that in these same circumstances have to work together to achieve some objective. Our work focuses on the problem of learning a good policy, where our agent must work with one or more teammates against a team of opponents without any communication and no pre-coordination. We test many different aspects of RL to try to improve the performance and discuss which aspects provide better results.

In this section, we discuss some approaches in HFO using both the high and low-level action space. Starting with the solutions found by authors when having an action space that has continuous parameters to be learned, seen in Section 3.1. Since this environment is very complex most of the solutions do not have more than the agent in the environment, testing with no other players in the game. Ending with a discussion on solutions that try to learn good policies using the high-level action space that has actions with no parameters, seen in Section 3.2.

3.1 Learning policies using a parameterized action space

Recently, some research has been used in the environment HFO using the low-level features. Such work must learn a policy for the agent, using the low-level action space, which contains discrete parameterized actions spaces with continuous parameters. We can define the action space like the work in [15] as the following:

$$A = \bigcup_{a \in A} \{(a, x) | x \in X_a\} \tag{3.1}$$

Where A is the available action space and X_a is all the possible values for the parameters.

3.1.1 Actor-critic architecture

In [16] the authors used an actor-critic architecture to learn in these conditions. They used an extended version of a Deep Deterministic Policy Gradient (DDPG), that allows the agent to select discrete actions and the respective parameters for these actions. The architecture uses four neural networks, an actor, a critic, a target actor, and a target critic. The actor-network takes the current state as input and outputs a Q-value for each action and all the parameters for each possible action. The agent selects the action that has the highest Q-value and pairs it with the correct parameters and executes that action. Afterward, the actor's selection and the current state are given to the critic that will output a Q-value for these selections. The actor-network is parameterized by weights and bias denoted as θ^{μ} and the critic-network by weights

and bias, denoted as θ^Q.

The critic learns by doing gradient descent on the following loss function:

$$L = \frac{1}{N}\sum_{t=0}^{N}(r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}) - Q(s_t, a_t|\theta^Q)))^2 \tag{3.2}$$

Where this loss function is a mean value of the difference between the Q-value of the current state calculated by the critic-network $Q(s_t, a_t|\theta^Q)$ and the calculated Q-value of the current state which is the reward gained from the current action plus the discounted Q-value from the next state, calculated by the target critic-network $(r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}))$.

The target is calculated using the target critic-network and the target actor-network while the current Q-value is the output from the critic, modifying the network weights and bias to minimize the difference between target Q-value and current Q-value. While the actor learns by adjusting his parameters to maximize the Q-value calculated by the critic. In other words, the Q-value is backpropagated through the critic to create a gradient that indicates how the action should change to increase the Q-value. A factor that improves the stabilization of the updates is the use of a replay memory like in DQN. The authors use epsilon annealing for their exploration, where they anneal the value of ϵ from 1 to 0.1 over the course of the first 10,000 updates.

The authors argue that the use of reward shaping on the reward function is needed when dealing with these actions, the more simple reward functions based on the outcome of the episodes is far too sparse for any meaningful learning. As such, they use the following reward function:

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

Where after each action the agent is rewarded if he approaches the ball between timesteps $d_{t-1}(a,b) - d_t(a,b)$, he is rewarded if his action led to the ball approaching the goal $3(d_{t-1}(b,g) - d_t(b,g))$, multiplied by 3 to counter the natural effect of moving away from the ball when kicking it. He is rewarded the first time per episode when he approaches the ball enough to kick it $\mathbb{I}_t^{kick}$ and finally, he is rewarded if he scores a goal $5\mathbb{I}_t^{goal}$.

In [17] the same authors expand this architecture by using an additional method for the action selection. They use an on-policy Monte Carlo (MC) approach in addition to the same DDPG architecture used in [16]. The reward function used is shown in (3.3).

This MC approach applies updates to the policy without any bootstrapping. This means that the target values used in the update are estimated directly from the rewards given to the agent during the course of the episode. These target values are calculated after each episode ends, where we get from the replay memory all the transitions regarding the current episode, and based on the outcome of the episode, the target from each tuple of experience is changed to the desired value. Afterward, this target

value is added to the already calculated target value from the DDPG, where we have a weighted average between both values, where the importance of each target is determined by a value β that is between [0, 1]. The new target value is then the following:

$$Q(s_t, a_t) = \beta \cdot Q(s_t, a_t)_{MC} + (1 - \beta) \cdot Q(s_t, \mu'(s_t|\theta^{\mu'})|\theta^{Q'})) \tag{3.4}$$

Where $Q(s_t, a_t)_{MC}$ is the target calculated by the MC approach and the $Q'(s_t, \mu'(s_t|\theta^{\mu'})|\theta^{Q'}))$ is the value calculated by the DDPG that uses the target critic and target actor-networks. With this approach, the authors managed to score consistently in a 1vs1 environment against a goalkeeper.

More solutions using an actor-critic architecture have been created like [18, 19]. These studies test new modifications to the architecture, either by changing the number of actor or critic networks or testing new optimization algorithms, seeing if the learned policy shows good results.

There are more solutions to this problem using an actor-critic architecture, like [18, 19], where the authors assess different changes to the DDPG algorithm, adding more actor or critic networks or changing the optimization algorithm to try to improve the results.

3.1.2 Summary

The main limitations of these works are that they only work with a single agent in the attacking team, when playing a more complex game these solutions do not scale well. We intended to use one of these approaches to learning a policy in the low-level action space and to analyze the agent's performance as we do in the high-level action space. Unfortunately, we noticed that these solutions in this action-space only work with a minimal number of players, most solutions only test in a 1vs0 environment, and one of the solutions tests against a goalkeeper (1vs1 environment). Since our work is focused on learning policies and examining in depth the performance in environments with teammates and opponents, that is, games more complex than 1vs0, a solution using this action-space can not be used.

3.2 Learning policies using discrete action space

In this section, we discuss some approaches to learning in the high-level action space, where there exist no parameters to be learned. There are 10 actions in this action space, only one of them being goalkeeper-specific and three can only be used by defending players, and as such we ignore them, as the solutions discussed are only for the team which is attacking.

3.2.1 Sarsa and Q-learning agents

There have been several approaches over time when it comes to learning in these conditions, the first approach used was in the paper that first introduced HFO [9], where the authors learn a policy by using Sarsa, explained in Section 2.3.2. Here, each agent uses a function approximation to calculate the Q-value for each pair state action (s, a). They use a different set of features containing a fewer number of features, created by them for this problem, having features more focused on the distance between players, the angles between them and the angle between the ball and goal. In the case of a 4vs5 environment, they only have 17 state features. For the action space, they use the actions: dribble, shoot, and pass. For the latter they discretize the action, having a single action pass for each possible teammate, in the case of 4vs5, having three different pass actions. When in possession of the ball, they use their current learned policy and when not in possession of the ball they use a static policy. The defending team also plays using a static policy that does not change. Regarding the reward function, the reward given depends on the outcome of the action.

$$r(s,a) = \begin{cases} 1 & \text{if action led to a goal} \\ -0.1 & \text{if action ended in ball out of bounds or captured by goalkeeper} \\ -0.2 & \text{if action led to defenders capturing the ball} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

The authors tested this same reward function but instead of rewarding after each action, waited until the end of the episode and retroactively rewarded the actions. They argued that this harmed learning, since an agent, for example, should not receive a negative reward for passing to his teammate and him missing the shot. Given this, they reward the action right after executing it so the action does not depend on the outcome of the episode. In their first test, that pass would receive a negative reward because the episode ended out of bounds. In this new strategy, a neutral reward would be given. They test these agents in multi-agent systems, in a 4vs5 environment, where the agents communicate between them the rewards received to improve the overall learning of the team. They also noted that communicating the received rewards between the team elevated the performance of each players' learned policy.

Other approaches like the one in [11] still use an agent that updates his policy using Sarsa, but instead of using the continuous state space and using a function approximation to calculate the Q-value for each pair action state. The authors discretize the state space using tile coding to have a more traditional tabular Sarsa approach, where we have a finite number of states. They also use only four features instead of the full state space: distance to the goal, angle to goal, open goal angle, and distance to nearest opponent. The reward function is similar to (3.5), changing the reward for out of bounds or captured by defense or goalkeeper to -1. The updates to the policy are only done when the agent is in possession of the ball or when the episode ends. They also tested a hand-coded agent that does actions guided by rules, that depending on the situation tell the agent what to do.

The work in [20] which is an extension of [21], use a similar approach as the last works, but instead use Q-learning to update the agent policy, while still using tile coding to discretize the state space. The tile coding used depends on the type of player, for example, a goalkeeper has only grids composing the state space in the goal area, while a striker has a higher density of grids closer to the goal so that he can better fine-tune his actions. This helps in learning since each position has a state space more appropriate for their tasks. Their work focuses on accelerating the Q-learning algorithm using heuristics. These heuristics come to play when the agent is doing exploitation. When the Q-values are calculated for each action in the current state, these heuristics are added to the estimations of the Q-value. This way improving the estimates without the need of doing many actions to discover these insights, accelerating the discovery of which actions are better or not. The heuristics are used to accelerate the learning process, to influence the agent to select actions that the algorithm believes are better, this knowledge comes directly from the domain (i.e. prior knowledge) or from playing in the environment and learning what heuristics to use to accelerate the learning. The Q-learning updates are not altered, since these heuristics only help in choosing the better actions, the rest of the Q-learning algorithm stays the same.

3.2.2 Neural Network

In [22] the authors propose a new approach to learning in the discrete action space of HFO. They use a subset of the full state-space, keeping 8 features, containing information about the position of the players and goal opening angle. They use several high-level actions created in the code released by Helios. This new high-level action space includes the following actions:

- **Shoot** - shoot the best available shot;
- **Short dribble** - dribble the ball a short distance;
- **Long dribble** - dribble the ball a long distance;
- **Cross** - cross the ball into the box in front of goal;
- **Pass** - performs a pass to a teammate.

They learn a policy by using the Fitted Value Iteration (FVI) algorithm [23], that iteratively improves its estimates of the value function of each state. When calculating the value of each pair state-action, instead of looking at every possible outcome of each state, this algorithm utilizes a sample of the next states to approximate these values. It then requires a neural network to estimate the Q-values $(Q(s_t, a_t))$ of each pair.

$$r(s,a) = \begin{cases} 1000 & \text{if action led to a goal} \\ -1000 & \text{if action led to terminal state other than goal} \\ -1 & \text{otherwise} \end{cases} \tag{3.6}$$

The reward function used is shown in (3.6). When the teammate has the ball, the agent follows a fixed policy. The authors also tested their agent with other teammate types, also concluding that this approach could be used and showed good results with other teams.

3.2.3 Deep Q-Network

Other solutions have been explored like the solution using a DQN in [24, 25]. They create agents that are capable of performing well when using the high-level action space. Although their main focus is the ad-hoc teamwork problems, where they test new ways to deal with an unknown team, they had to create an architecture that was able to make an agent learn in this environment. The DQN receives the current state and outputs a Q-value for each possible action. This architecture is a good fit for high-level actions since the actions do not possess parameters to be learned. They tested their agents when learning with teammates and against opponents and managed to achieve good results even when dealing with multiple players in the environment.

In [24] the author uses the following high-level actions: Shoot, dribble, pass, move, reorient and go to ball. He makes some actions legal and illegal, depending on the state space to improve and accelerate the agent's learning. The author managed to achieve around 45% of episodes ending in goal. For the reward function he uses the following:

$$r(s,a) = \begin{cases} 1 & \text{if action led to a goal} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

In [25] the author uses a subset of the high-level features to input to the DQN. For the action space, the author extended to have 13 discrete actions. When he has the ball the agent can execute the action Shoot, Short dribble where he dribbles for 4 steps, Long dribble where he dribbles for 10 steps, and a pass action for each teammate. When he does not have the ball he can perform the action do nothing during 4 steps, move towards the ball for 4 steps, move towards the nearest opponent for 4 steps, move towards the nearest teammate for 4 steps, move away from the nearest teammate for 4 steps, move towards the goal for four steps, move away from the nearest opponent for 4 steps. While using the reward function in (3.6).

3.2.4 Discussion

Most of these solutions show good results and are a viable way to learn a good policy in the Half Field Offense environment when using the high-level action space. None of the discussed research attempts to test extensively the various elements of learning a policy when using a DQN. In [24] they tested a few changes like different state spaces or extensions to the DQN like Double DQN [26] or Dueling DQN [27]. However, no study attempts to test various elements like reward function or impact of feature selection.

Another observation is, most if not all these solutions use only as metrics for the agent's performance the percentage of goals scored. However, we feel this does not give a full picture of the learned policy and why our team manages to score more goals since many factors may be affecting the performance and our agent might be reacting differently than we expect. For example, our agent might be scoring more goals but is he playing with the team, or he does everything alone, we probably do not want the latter behavior, so this kind of analysis is needed to improve our policies.

4

Implementation

Contents

In this section, we describe our solution to the problem of learning a policy in the Half Field Offense environment, when working with and against different agents, while using different state features. More specifically, first, we explain the different approaches and architectures used and the different tests performed to try to improve the results in the high-level action space. Afterward, we explain the more in-depth analysis done to better grasp the impact our agent had on the team as a whole. Finally, we did some more tests in a 3vs3 environment. All the implementation was done using Python and the models constructed using the machine learning framework Pytorch.

The architecture used is in 4.1, where our agent receives the current state and returns the selected action after passing it through the DQN. While this decision is happening, our teammates also choose their actions.

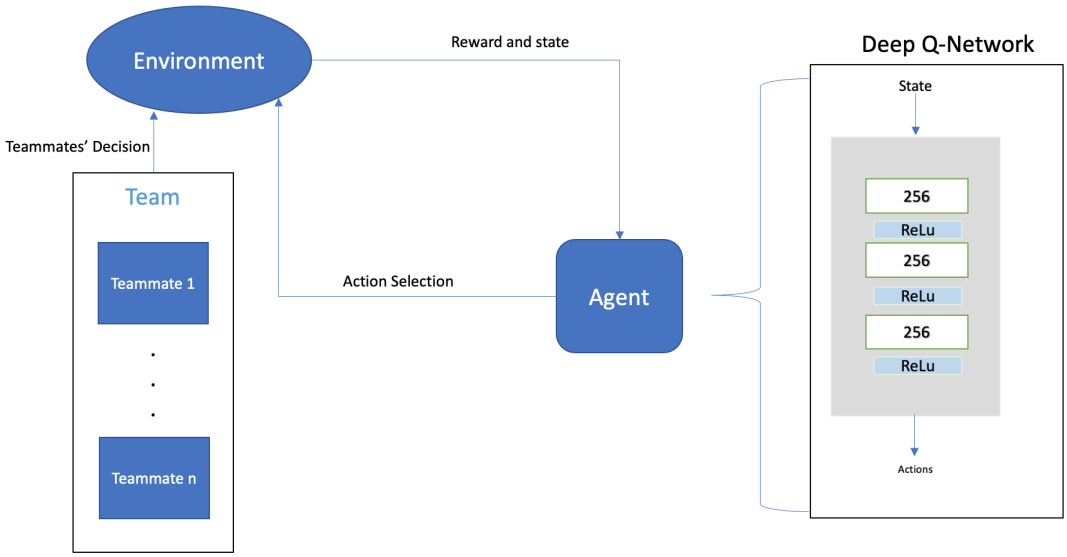


Figure 4.1: Overview of the architecture used

4.1 Learning a Policy in high-level action space

Our agents, both using low-level or high-level state space, use a DQN to train their policies. We use a DQN since it is appropriate for the problem, given that we have discrete actions and continuous state features and a DQN approximates Q-Values for each discrete action. Our tests are all done in the Half Field Offense environment, in a 2vs2 scenario, where we have one Helios Teammate against two Helios Opponents, learning a policy for the actions of a single agent. We also did not initially use the flag Fullstate that the simulator provides, which makes the learning easier for our agent, since without this flag the features given to our agent contain noise and make the optimal policy harder to learn. We did, in the end, test our best solution using the Fullstate flag to see the impact that this flag has.

4.1.1 Action Space

The full action space regarding all actions that can be executed is: Move; Shoot; Dribble; Pass; Go to Ball; Reorient. With exception of the pass action that needs the teammate number, these actions do not need the agent to select the parameters to be executed, these parameters are automatically selected. Since in the 2vs2 environment, we only have one teammate we can discretize the pass action because if he chooses this action there is only one possible choice for the parameter.

We first tested many approaches, changing the network structure, testing several reward functions, and changing many hyperparameters before being able to make our agent learn. Our agent had trouble learning in our first attempts because we allowed all actions to be performed at all times. This means, for example, the agent could try to kick the ball even when not in possession of the ball and this made the learning extremely difficult and time-consuming. With this in mind, we had to force some restrictions on the agent, making some actions illegal and legal, to help him learn the environment. Additionally, this adjustment prohibits, when calculating the target value for the update, the selection of the best Q-Value for the next state to be from an action that can not be performed. This means when selecting the best Q-value for the next state, we do not take into account illegal actions, removing them from the possible choices. When exploring, we had to redefine the probability of executing the actions and this is modeled according to [28]:

$$\pi(s,a) = \begin{cases} 0 & \textit{if a} \notin L(s) \\ \frac{1}{|L(s)|} & \textit{if a} \in L(s) \end{cases} \tag{4.1}$$

Where *L(s)* is the list of legal actions in any given state *s*. This list depends on if the agent can kick the ball or not. If he can kick, the list contains the action shoot, pass, and dribble and if he can not kick, the list contains the action move, go to ball, and reorient.

4.1.2 State Space

We used both the high and low-level state-space provided by the simulator. In low-level state space, we tested using feature selection, while for the high level we tested using a subset of the full space, seeing the impact it has on the agent. This subset of features was based on the ones kept in [25], having 14 features instead of the normal 24, where we kept:

1. Agent's X position;
2. Agent's Y position;
3. Orientation;
4. Ball's X Position;

5. Ball's Y Position;
6. Agent being able to kick the ball;
7. Agent's Proximity to closest opponent;
8. Teammate's opening goal angle;
9. Proximity that the teammate has to closest opponent;
10. The pass opening angle to teammate;
11. Teammate's X position;
12. Teammate's Y position;
13. Tells if the agent's last action was successful;
14. Agent's Stamina;

We also tested whether adding a new feature to the base high-level state-space had any impact on the performance of our agent. We wanted to analyze what impact adding and removing features had on this state space. This feature was added in [25] since this was an extremely important feature for the authors' agent and indicates if our teammate has possession of the ball or not. This feature is calculated the same way as the author, where we calculate the euclidean distance between the ball and the teammate, if it is smaller than 0.15 we assume he has possession of the ball.

For the low-level state space, we did feature selection based on a Correlation-Based Filter Solution done in [29]. In this solution, the authors remove features that are strongly correlated to each other, to try to reduce redundancy. So, if we have two features that have a correlation larger than 0.95, we remove one of them. We gathered the needed observations by saving all the received states that our agent had during the course of 1000 episodes, gathering around 200 thousand observations. We then compared the correlation of each feature to each other, removing those that had a high correlation, and also removed 12 features that never changed values on the observations we gathered since these give no useful information. These are:

1. Indicates if the agent's position is valid - removed due to never changing;
2. Indicates if the agent's velocity is valid - removed due to never changing;
3. Indicates if the agent is colliding with the post - removed due to never changing;
4. The Cosine value the agent has to the upper goal post - removed due to having a high correlation to the goal center cosine value;

5. The Sine value the agent has to the upper goal post - removed due to having a high correlation to the goal center sine value;
6. The proximity of the agent to the upper goal post - removed due to having a high correlation to the proximity to the goal center;
7. The Cosine value the agent has to the bottom goal post - removed due to having a high correlation to the goal center cosine value;
8. The Sine value the agent has to the bottom goal post - removed due to having a high correlation to the goal center sine value;
9. The proximity of the agent to the bottom goal post - removed due to having a high correlation to the proximity to the goal center;
10. The agent's proximity to the top right corner - removed due to having a high correlation to the proximity to the top right corner of the penalty box;
11. Distance between the agent and the left goal line - removed due to having a high correlation to the agent's distance to the right goal line;
12. Distance between the agent and the bottom field line - removed due to having a high correlation to the agent's distance to the upper field line;

4.1.3 Structure of the Deep Q-Network

For the architecture of our Deep Q-Network, we tested three different networks. We tested one which had two hidden layers and two networks that had three hidden layers, changing only on how many hidden units each one had. All networks use mean squared error as their loss function and use Adam optimizer [30] with a learning rate of 0.00025. Regarding the architecture of each network we have:

	Input Layer	**Hidden Layers**	**Output Layers**
First Network	Number of units depending on features.	Hidden layers with 256, 256, 256 units.	6 units for each action.
Second Network	Number of units depending on features.	Hidden layers with 512, 512, 512 units.	6 units for each action.
Third Network	Number of units depending on features.	Hidden layers with 256, 64 units respectively.	6 units for each action.

All layers used He initialization, input and hidden layers use ReLU activation and the output layer used linear activation.

In summary, we tested the impact of having different DQN architectures, having fewer hidden layers with fewer hidden units, and assess the impact that has on the agent's learning. The networks used are based on the networks in [24, 25]. We used these networks as it was already proven in the work of our colleagues that they could learn in this environment. The tested configurations of the DQN are the ones shown in Figure 4.2.

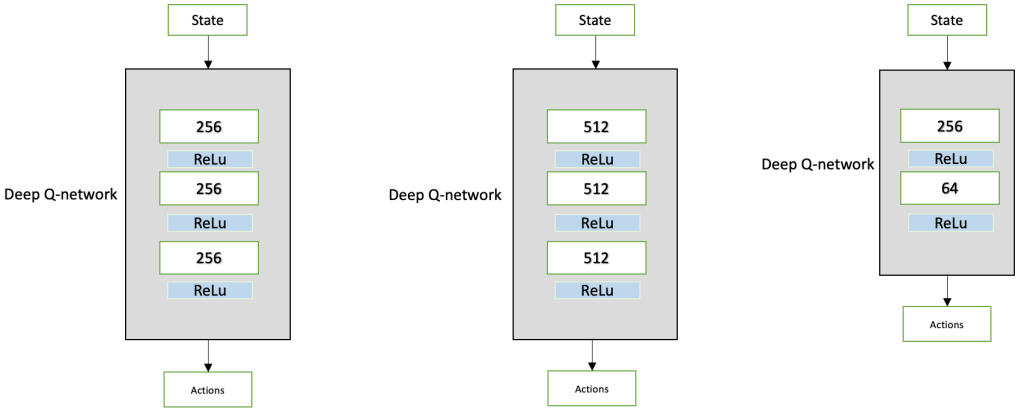


Figure 4.2: Overview of the DQN structures tested for learning in high-level action space

4.1.4 Hyperparameters

The hyperparameters used are the following:

- **Learning Rate** : 0.00025
- **Epsilon** : [1.0 - 0.01]
- **Gamma**: 0.99
- **Batch-Size** : 32
- **Target Network Update Frequency** : After every 75 episodes
- **Network Update Frequency** : Every timestep
- **Final Exploration Time Step** : 5 million timesteps

The hyperparameters are based on the ones in [1], changing some hyperparameters based on the work done by our colleagues to better fit the HFO environment. The hyperparameters were not extensively tested, keeping these constant for all of our tests, with the exception of the exploration vs exploitation strategy where we tested two different approaches.

4.1.5 Reward Function

Several reward functions were tested to analyze the impact these had on learning the policy and the needed reward function for this environment. We first use the following reward function:

$$r(s,a) = \begin{cases} 1000 & \text{if action led to a goal} \\ -1 & \text{if action led to non terminal state} \\ -1000 & \text{if action led to terminal state other than goal} \end{cases} \tag{4.2}$$

We tested the impact of the reward function, which rewards 1 if the agent scored and 0 otherwise. A reward function that rewards 1000 for scoring a goal and 0 otherwise, and finally the reward function similar to (4.2), only receiving 0 if the action led to a non-terminal state. The advantage of these more simple reward functions is that there is no reward shaping that might make the agent do unexpected actions, having a wild behavior by exploiting the reward function in non-intended ways. However, we wanted to test the impact of using reward shaping, as seen in (3.3), where the agent receives rewards based on the outcome of their actions, even when leading to non-terminal states. This might make our agent have unintended behaviors but, as it is a more complex function, it might improve the learned policy.

4.1.6 Exploration vs exploitation

Here we explain the two different strategies for the ϵ-annealing problem that were tested.

We started our analysis by using a fixed ϵ=1 and after 1 million actions this value was fixed at ϵ=0.01, to try to explore the environment as much as possible before exploiting more. We noticed some undesirable results and, as such, we changed to a more traditional ϵ-annealing strategy, where we start at ϵ=1 but after each action decrease the value slightly over the course of 5 million actions being fixed at ϵ=0.01.

4.1.7 Summary

So in conclusion, regarding the learning in high-level action space in a 2vs2 environment, where we have in our team a Helios teammate and where we are against 2 Helios agents, we tested:

1. Two different exploration vs exploitation strategies, lowering the value of epsilon from 1 to 0.01, either after every action or changing abruptly after a fixed number of actions;

2. The impact of changes in the state space. In the high-level, we compared the performance of using the base set with an added feature to the base set and finally using a subset. Examining the performance of the best solution in high-level state space with the base low-level state space. Afterward, for the low-level state space analyzing the impact of doing feature selection;

3. The impact of using three different DQN structures in the low-level state space. A DQN that uses 3 hidden layers with 256 nodes each. A DQN that uses 3 hidden layers with 512 nodes each. Finally, a DQN that uses 2 layers with 256 and 64 nodes respectively.

4. The impact of the reward function, testing five possibilities. Four of which are a simpler function, that rewards the agent on the outcome of the episode and penalizes him for each action, and a

more complex function used mainly in low-level action space solutions but we analyzed the impact it had on our agent.

5. Finally, the impact on the percentage of goals scored if we used the Fullstate flag, that removes noise from the features given, on our best solution.

The best solution for each test is used in the following tests.

4.2 In-depth agent analysis

After these experiments, we concluded what was our best solution for the environment of 2vs2. We then analyzed in-depth the performance of our agent when using our best approach. This analysis came from the fact that we were unsure of the impact of our agent. Although the performance increased as the training progressed, we wanted to analyze if that increase was due to our agent scoring more goals or simply because he learned to pass the ball and wait for his teammate to score goals. We analyzed how many of the goals came from our agent scoring and compared it with our teammate and then we analyzed how many goals came from an assist of our agent and saw how many passes on average our agent performs per episode, to see if our agent learned to just play by himself. We also assessed if over the course of training our agent scored more goals or if the number of goals remains constant.

After this in-depth analysis, to gauge the impact our agent had on the team's performance, we calculated the performance of a team consisting of our agent and a Helios teammate and compared it with a team of two Helios teammates, in both, they are against 2 Helios opponents. Additionally, we added a new NPC strategy, the *Autmasterminds*, gathered from the binaries of the 2013 Robocup 2D competition and trained our agent in the environment where we have either an *agent2d* or an *Autmasterminds* teammate against two Helios opponents and compared with a team of two *agent2d* or two *Autmasterminds* versus two Helios.

4.3 Other Environments

Finally, we tested our agent in environments other than a 2vs2. We analyzed the performance of our agent in a 3vs3 environment, where we have to work with 2 Helios teammates versus 3 Helios opponents. Since we have more teammates, we had to add a new action to the possible pool of actions. The pass action in the high-level action space needs a parameter that tells which teammate to pass, so we added a new action Pass to allow us to pass to our second teammate. We then did an in-depth analysis, checking how many goals he scored when compared to his team and the improvement of the team's goals over time.

5

Results

Contents

Here we describe and discuss the results obtained in our work. These results are used to validate the solution that we have described in Chapter 4. Firstly, we explain how we gathered these results and the metrics used to compare approaches presented in Section 5.1. Afterward, we discuss the results obtained in learning a policy in both the action spaces.

5.1 Evaluation Procedure

Firstly, we explain our evaluation procedure for the task in Section 5.2, where we learn a policy for the high-level state space. Our first task consisted of testing and seeing the impact of changing various hyperparameters, state feature modifications, and the type of information that we get from the environment and comparing the various learned policies that our agents obtained. We train the agent's policy by running our agent with a Helios teammate versus two Helios opponents during 200 thousand episodes. After each 5000 episodes of training, we save a snapshot of both the network weights and target network weights, so later we can gauge the evolution of our agent over the course of their learning to assess our current solution (i.e. current network topology, current state features, etc). We run 10 different agents (i.e. repeating the steps explained above saving snapshots for each) in the same circumstances, so we can later do an average of the performance of each agent and try to minimize the possible discrepancies that an agent might have, for example in the weights initialization.

We did the same procedure for training an agent with other teammate types as can be seen in Section 5.3, where we load the network weights after being fully trained (i.e. after 200 thousand episodes of training). Without exploring and without modifying the networks' weights, we saw how many goals our agent scored when compared with his team. For each teammate type, we ran for 1000 episodes a game of 2 of these teammate types versus 2 Helios, for example, 2 agent2d vs 2 Helios. This is done to compare the number of goals scored when we change a member of the attacking team to our agent.

Finally, for the task in Section 5.4, where we run our agent in a 3vs3 environment, we also trained our agents for 200 thousand episodes to analyze if they can learn using the same hyperparameters used in a 2vs2 environment, concluding with an analysis of our agent's performance.

5.1.1 Metrics

Our performance measure for the initial tests is the same used in [11], where we see the percentage of episodes that ended in a goal. We assume this is a good measure for the team's performance because even though it does not give a full picture of the agent's performance our main objective is to score the most goals possible, so a policy is better if it scores more goals. In the following graphs, we have the average of 10 runs, at different levels of training and we start with the rollout (i.e. the fixed DQN weights and no exploration at that time) of each agent after 5000 episodes of training and we see the evolution

of the performance after each 5000 episodes of training until the end of the 200 thousand episodes of training. For each rollout, we run the networks without changing their weights and with no exploration and run them for 500 episodes to see the percentage of episodes that end in a goal. In the following plots, the grey area beside the mainline is the confidence interval, where we assume the performance has a normal distribution and use a 95% confidence value.

For the in-depth analysis, we load the network weights after the training (i.e. after 200 thousand episodes) and run each trained agent for 1000 episodes and do an average of how many times they scored, the goals that came from an assist, the number of passes and how many goals scored by our teammates. For the tests involving the evolution of goals and passes over the course of training, we run each trained network rollout for 500 episodes. Each rollout is the state of the network weights after every 5000 episodes of training.

For the tests regarding the 3vs3 environment, we use a combination of these last two metrics to assess the agent's capability.

5.2 Learning a policy using the high-level action space

Here we present the tests that were done regarding the high-level action space and both state spaces. The action space is explained in Section 4.1.1.

5.2.1 Learning a policy using high-level state features

In this section, we discuss the results gathered, regarding the learning of an optimal policy using the high-level state features. Concluding with the approach that gathered the best performance.

To learn a good policy in this environment, we employ a model similar to the one used in [25], where we utilize a DQN, with a target network, that receives the current state features and outputs the high-level action to execute. The network has 3 hidden layers, where each layer has 256 nodes.

Our first test was using the base high-level features, with no changes. In this first test in Figure 5.1 we used ϵ=1.0 for 1 million actions and after that, we set a fixed ϵ=0.01 until the end of training. We were able to achieve around a 22% chance of scoring but we noticed that the learned policy was unstable.

In our second test in Figure 5.2 we added to the base features, a new one that tells our agent if a teammate has possession of the ball or not, this value is a boolean that is true if the teammate has the ball and false otherwise. We added this feature since it had improved the results and was extremely important in the work in [25]. So we tested how our agent learns when using this addition, making the total number of features 25, while also using the same ϵ strategy. We were able to achieve around an 18% chance of scoring but we started to see a trend where the agent starts to decrease his performance by the end of training and again the networks become very unstable in terms of performance. Noticing

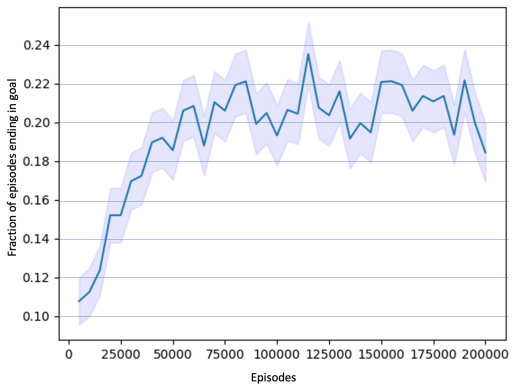


Figure 5.1: High-level base set without epsilon annealing

that adding the new feature hurts the performance, one hypothesis to this is the fact that we have noise in our state space this new feature might sometimes be wrong.

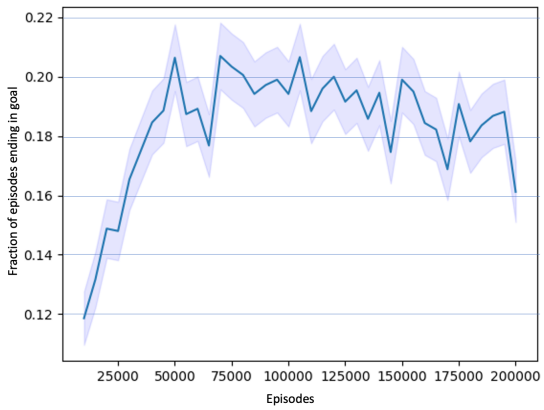


Figure 5.2: High-level set with added feature

We then tested an agent in the same circumstances as in Figure 5.1 but instead of being completely random during 1 million actions and then having only 1% chance of exploring, we used an epsilon annealing where an agent starts with ϵ=1 and after each action, the epsilon decreases slightly during 5 million actions until being fixed at ϵ=0.01, having a gradual decrease in epsilon.

We noted in this new test that the networks did not seem unstable and did not suffer from the problem of losing performance during the course of training. The performance of the agents using the epsilon annealing was not impacted when compared to the first test, still having around a 22% chance of scoring. This comparison can be noted in Figure 5.3.

This instability, when learning a policy with a full exploration mindset before switching to 1% exploration, might be caused by the fact that since we are only exploring in the beginning and having a minimal exploration probability, this might harm our agents' ability to learn the environment. Additionally,

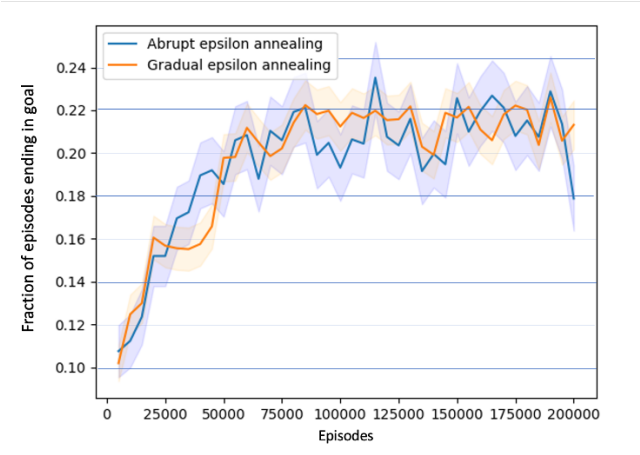


Figure 5.3: Comparison between epsilon annealing strategies

our agent does not move from exploring to exploiting slowly, this harms our agent since he does not slowly learn the policy while also exploring new actions and instead does completely random actions without following the policy.

Given this and the fact that epsilon annealing had better results and is widely used in literature, we decided for our future tests to continue using epsilon annealing. We then tested using a subset of features, keeping the same ones mentioned in 4.1.2. Using this subset we were able to achieve slightly better results than the base high-level features, scoring around 24% to the previous 22%. This increase can be explained by the fact that the features removed can cause unnecessary noise to our agents' selection.

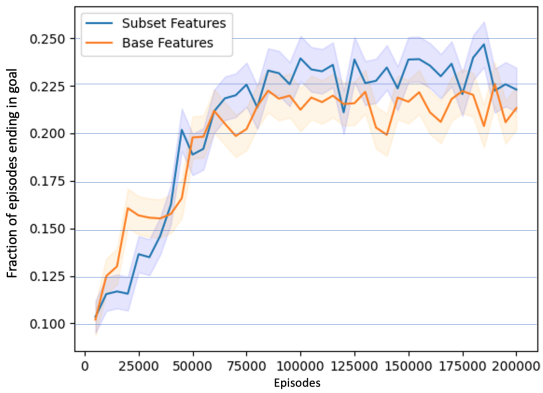


Figure 5.4: Comparison between subset and base features

5.2.2 Learning a policy in Low-level state features

In this section, we discuss the different tests made to the learning of our agents, when learning with low-level features. This type of state contains a lot more features to deal with. When playing with one teammate and two opponents we have to deal with 86 features instead of 24 like in the high-level features.

Changing DQN Structure

Our first test was to train our agent using the same network and hyperparameters as our high-level state space approach and see what results we were able to achieve. We only changed the input layer to be able to take more features as input. As such we used a DQN containing 3 hidden layers, with 256 units each. This resulted in a performance seen in Figure 5.5, we were able to achieve around 25% of episodes ending in goal.

We believed this was a low percentage since our colleagues using different networks had better results in the same environment, and, therefore, we tested a few more network structures to try to increase the performance. As we had more features for our agent to take into account, we thought that increasing the number of units in each hidden layer might improve our agents' ability to generalize better and boost the performance. We then raised the number of units in each layer to 512, which means we had a network with 3 hidden layers and 512 units in each.

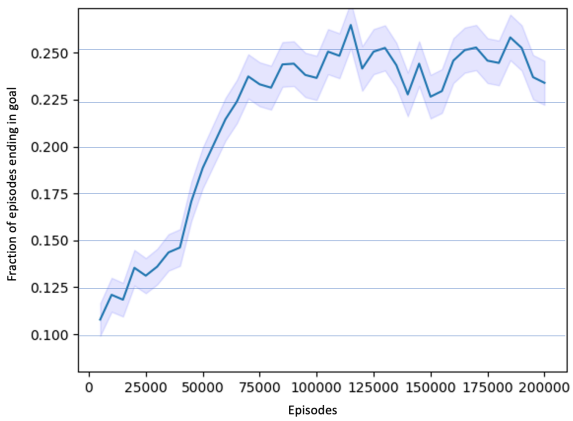


Figure 5.5: Agents performance using a DQN with 3 hidden layers with 256 units each

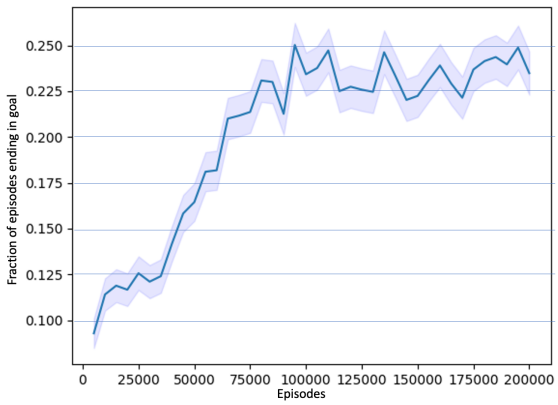


Figure 5.6: Agents performance using a DQN with 3 hidden layers with 512 units each

The results from this network structure are presented in Figure 5.6, but we noted that this did not make a difference, the results still being around 25%, slightly worse than our previous network structure. Since increasing the number of units, decreased the performance of our agent, we tested reducing the complexity of our network to see the impact this had on the agent.

Given this, we changed the network structure to the one used by [24], where we changed the number of hidden layers to only two, and the layers contained 256 and 64 units, respectively. Our reasoning for this change is, as explained before, to see the impact of having a less complex network. So, we ran 10 agents using the same hyperparameters only changing the structure of the network and seeing how our agent performed in these circumstances. This resulted in an agent's performance seen in Figure 5.7. We managed to improve the results to around 28%. As this approach was the one that got better results and the time to train this shallower network was smaller, we decided to use this network structure for our future tests with low-level features.

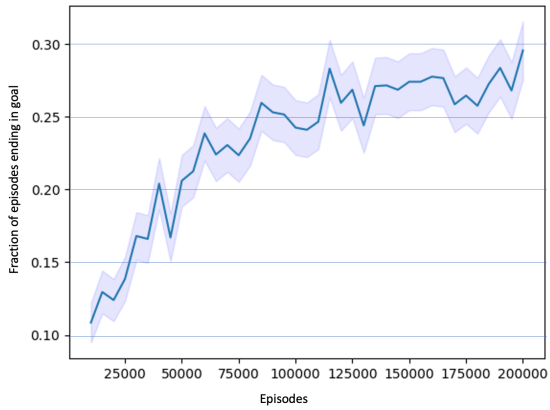


Figure 5.7: Low-level state performance using a DQN with 2 hidden layers with 256 and 64 units respectively

We did a quick test on the impact of doing feature selection on the low-level feature set. The features removed are explained in 4.1.2. We obtained the results in Figure 5.8, where we did not have a significant difference from our previous solution and this can be explained by the fact that the removed features either do not change or are strongly correlated to ones we kept, so we do not gain or lose any information. Thus, we obtained around 28% of results as the previous solution.

Changing Reward Functions

In this section we discuss the various tests done to the reward function, to assess the impact that this element has on the agent's learning. Due to time restrictions, all the tests in this subsection were only done using 5 runs instead of the normal 10 runs.

Our first test was changing the rewards to smaller values, changing the reward to 1 for an action ending in goal and 0 to the others. The results are what was expected, where the reward was too small and scarce for the agent to learn anything significant. The agent showed no improvement in training as can be seen in Figure 5.9.

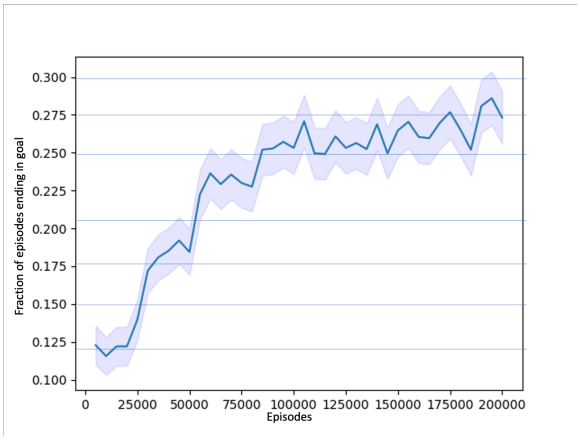


Figure 5.8: Agents performance after using feature selection on the low-level feature state

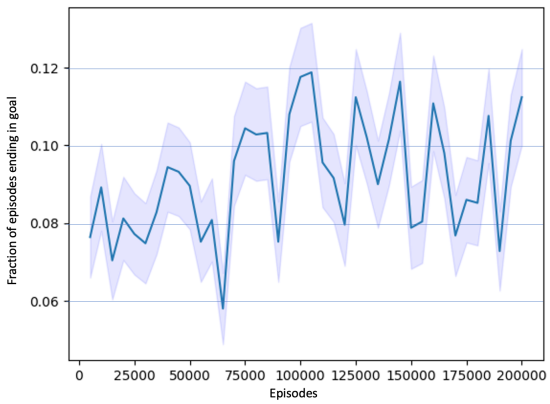
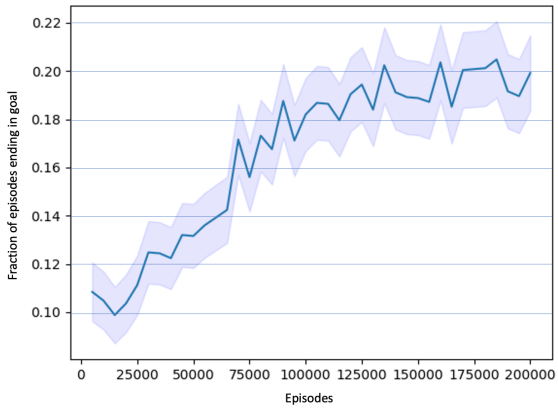


Figure 5.9: Agents performance when given smaller rewards

$$r(s,a) = \begin{cases} 1 & \text{if action led to a goal} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

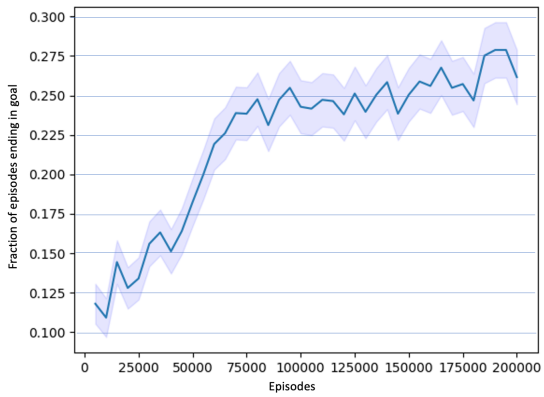
Having concluded that an agent with a smaller reward has trouble learning in this environment, we increased it to 1000 when scoring and 0 the rest. We noticed in this experiment, that by giving bigger rewards the agent could indeed learn and improve its performance. With these parameters, we noticed in Figure 5.10 that the agent managed to have around 20% of episodes ending in goal, which means, although he improved over time, his performance still did not match the performance seen in Figure 5.7.



$$r(s,a) = \begin{cases} 1000 & \text{if action led to a goal} \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Figure 5.10: Agents performance when given bigger rewards

The next setting for our tests was to give a penalty of -1000 for actions that ended in terminal states other than goal (i.e. out of bounds, out of time, captured by defense). The results with this setting were, as expected, better than the last test, as shown in Figure 5.11, having around 27% of episodes ending in goal. This means that it reached around the same results of Figure 5.7 of around 28%, which tells us that the best reward function for this particular environment with low-level features is having a bigger reward, giving penalties to our agent if reaching non-desirable terminal states.



$$r(s,a) = \begin{cases} 1000 & \text{if action led to a goal} \\ -1000 & \text{if action led to terminal state other than goal} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Figure 5.11: Agents performance when given bigger rewards and penalties

Finally, we tested a more complex reward function, the one used by in [16], where the reward is based on the position of the agent and the position of the ball regarding the last step. This more complex function is needed when learning with low-level action space but this function performed worse than expected, see the results in Figure 5.12. The results are around 12% doing much worse than the simpler reward. Since high-level actions do not require parameters, this reward shaping might give too much information to our agent, making him do unexpected actions and resulting in poor results. Another theory, is that this reward function has trouble learning without the Fullstate flag since without this flag

the reward calculations might be wrong and need more time to learn. Nevertheless, in the same training span of 200 thousand episodes this approach performs worse than the simpler reward function.

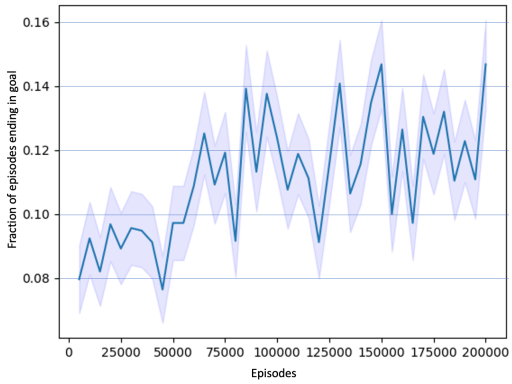


Figure 5.12: Agents performance when using reward shaping

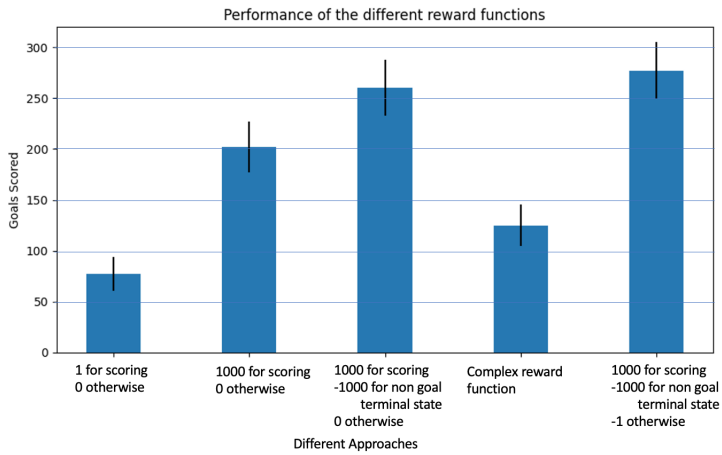


Figure 5.13: Comparison of each reward function

We have in Figure 5.13, a side-by-side comparison of the different reward functions, where we see the number of goals each reward function managed to score. These results are gathered using an average of the agents fully trained (i.e. after 200 thousand episodes), running each agent for 1000 episodes and analyzing how many goals they scored. These results are averaged through the 5 trained agents of each reward function. We reached the same conclusions as before, where the reward functions that reward 1000 for scoring and -1000 for not scoring, are the best. They achieve similar results and as such, either one can be selected. So, we choose the one that also gives a -1 reward for non-terminal actions.

Fullstate Flag

We wanted to test the impact of the flag Fullstate has on the results obtained. As such we ran an experience using our best solution and using the flag Fullstate to remove the noise from the features given to our agent. The reward function and the network architecture used was the one that gave the agent their best performance, the one used in Figure 5.7. The obtained results are what we expected, as the performance of our agent improved dramatically, as the noise of the features given to our agent no longer mislead him to do worse actions at each timestep. As it can be observed in Figure 5.14, we got around 45% episodes ending in goal, using the exact same hyperparameters and network as Figure 5.7.

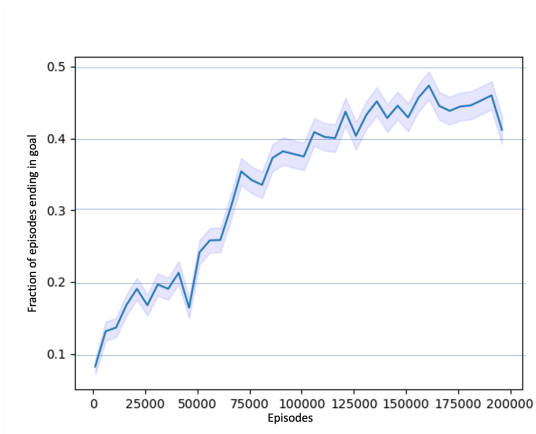


Figure 5.14: DQN with 2-hidden layers 256 and 64 units respectively using the fullstate flag to remove noise from features

So, we can conclude that our best approach achieved around the same results as in [24], even when having some different elements like the reward function.

5.2.3 Discussion

So we can conclude that, when learning a policy using the discrete action space, the low-level feature space has a better performance than the high-level. Even though we have more features to take into account, the agent can better fine-tune their behavior as he receives more information.

In the case of the high-level state space, using a subset of features improves the percentage of goals scored and this might happen because some of the features removed harmed the learned policy. Using the base set with the added feature and noise, it affects the performance negatively since it showed the worst results, while using epsilon annealing gave stability to the learned policy.

When dealing with the low-level state space, the less complex network showed the best results, this might happen because a more complex network might try to find too complex patterns and end

up harming the policy learned. While feature selection using a correlation-based criteria, did not affect the learning of the agent, one hypothesis to this is the features removed are correlated to ones kept, so removing them does not affect the agent's learning. When we discuss what reward function to use, we have to use a reward that does not give small rewards, because it is too scarce to learn anything, which makes the agent's learning impossible. If we penalize our agent for not scoring a goal he performs better and penalizing him for each action, does not affect drastically the learned policy. In the case of an environment not using the Fullstate flag, the complex reward function is not advised, since the noise added disturbs our agent too much for him to learn a good policy.

Finally, the flag Fullstate boosts our agent's performance dramatically since he has more accurate information about the environment.

5.3 In-depth Analysis

We wanted to see the real impact the learned policy had on the team's performance as a whole. We wanted to know for certain if our agent was the one scoring more goals and being the best player on the team, or if he just assisted his teammates and just took a passive attitude in the game. So we concluded that one way of analyzing this is to see the number of goals our agent scored when compared to his teammate and to see if this has an increase over the course of training. All the tests in this section are done using a DQN with 2 hidden layers with 256 and 64 nodes respectively, using the flag Fullstate and using the low-level state space. In a 2vs2 environment where we have 1 Helios teammate and 2 Helios opponents.

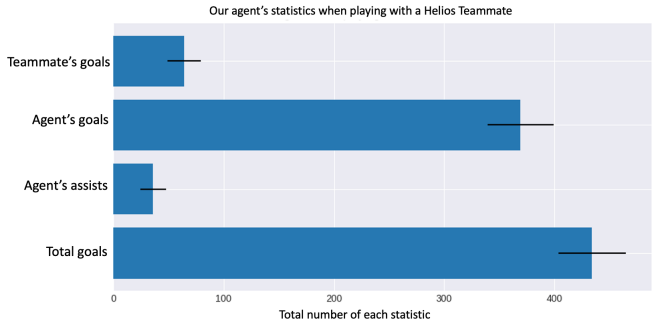


Figure 5.15: Agent's statistics when running in a 2vs2 environment

In Figure 5.15 we ran each of our trained agents, in the configuration explained in Section 5.3, loading the network weights after 200 thousand episodes of training. For each agent, we played 1000 episodes and saw how many goals he scored, how many goals his teammate scored, and how many goals came from a pass from our agent (i.e. an assist from our agent), and we averaged the results over the agents. We noticed from the around 430 goals scored, around 370 came from our agent. Only around 60 goals

came from our teammate and of those around 40 were assisted by our agent. The number of assists might be slightly wrong since we can only check our last action, if our last action when we had the ball before a goal was a pass we count that as an assist. So if, for example, we pass and an opponent scores an own goal we count that as an assist. Furthermore, the game has no game logs that we can access that tell us what actions were executed by each player at each timestep, which would make this analysis easier and we would not have this problem of having slightly wrong assist numbers. We can conclude by observing this image that our agent scores more goals than his teammate and that the policy learned makes him be more aggressive and be the main goalscorer instead of being passive and playing for the team.

In Figure 5.16, we can see the evolution of both the contributions of our agent and his teammate, where our agent starts having similar scoring numbers as our teammate but after some training, his amount increases dramatically and by the end, he scores the majority of goals. So we can still conclude that our agent is the most important player on his team, in terms of scoring.

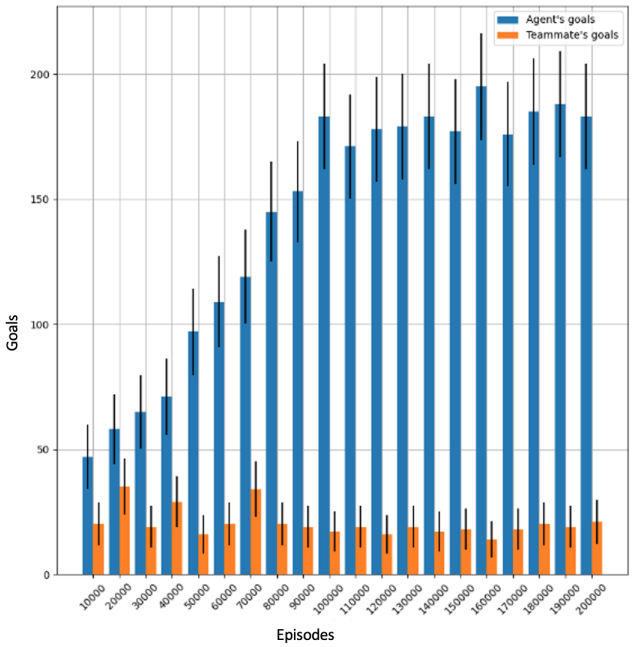


Figure 5.16: Agent vs Teammate goals

Our next analysis was to see if our agent learned to play alone and just waits for a pass from our teammate and tries to do everything alone, or if he continues to play as a team but just shoots more. To analyze this we calculated how many passes per episode on average our agent does. So to calculate this we ran each trained agent over the course of training and analyzed how many passes on average he does after each 5000 episodes of training.

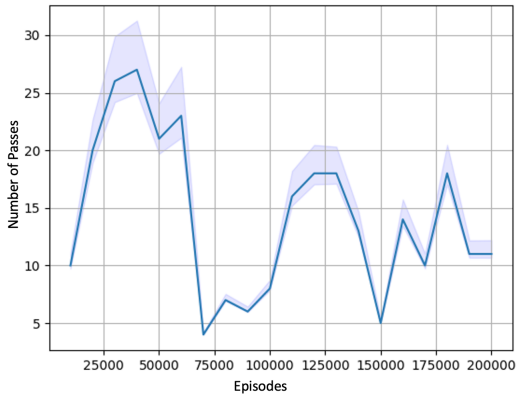


Figure 5.17: Number of passes over the course of training

We noticed by analyzing Figure 5.17 that our agent learns to not pass as much. He starts by passing more because he is exploring and doing random actions, but after some training, he starts to pass less at around 10 passes per episode. Also since we are in a 2vs2 environment with a single teammate, the number of times he is in a favorable position might be few. So we can conclude that our agent learns to shoot more while passing less, which means he acts as the team's main scorer. Since every episode has around 165 actions, around every 16 actions our agent does one pass.

5.3.1 Other teammate types

In this section, we use other teammates' strategies and see the performance against 2 Helios opponents. The teams that will be compared are our agent with an NPC type T and a team with 2 NPC type T, where T will be one of three strategies: Helios; Autmasterminds; Agent2D.

Our first test was working with an agent2d teammate type. We did the same in-depth analysis as before. This can be seen in Figure 5.18

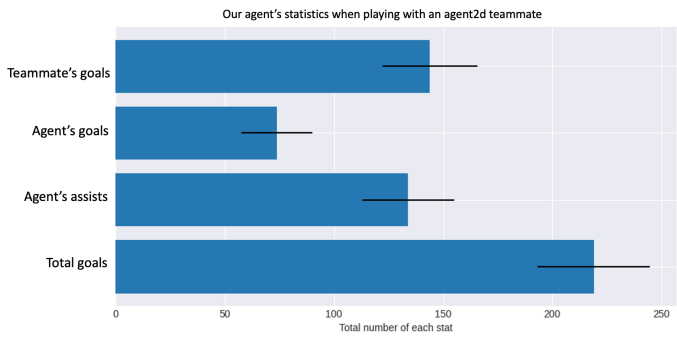


Figure 5.18: Agent's goals and assists in a team with an agent2d teammate

When being on a team with an agent2d, we see that unlike the previous test with a Helios teammate

here our agent scores less, passing more to his teammate and waiting for his teammate to score. From the around 230 goals scored only around 60 came from our agent. This might happen due to our teammate scoring with a higher percentage than us so we have a bigger chance of getting the goal reward by passing to our teammate.

Next, we did the same tests with an Autmasterminds teammate and analyzed what the performance was. This can be seen in 5.19.

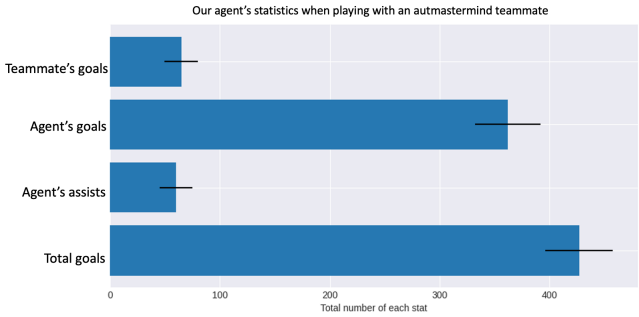


Figure 5.19: Agent's goals and assists in a team with an Autmastermind teammate

We noticed the same trend as with a Helios teammate, where our agent scored a high percentage of the team's goals. Most of our teammate's goals came from an assist from our agent.

Finally, we compared the performance of a team of 2 NPC types and a team of 1 NPC type and our agent, to see if our agent has a positive impact on the team, or if he just obstructs the team. So we ran our trained agents for 1000 episodes and do an average of the goals scored. Afterward, we ran each team of two NPC types 10 times against 2 Helios and do an average of the goals scored. The results are in Figure 5.20.

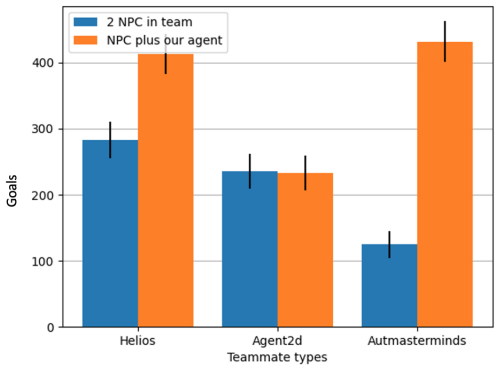


Figure 5.20: Comparison between teammate types and our agent

Regarding the Helios and Autmasterminds NPC type, we can see a great improvement when we switch one of the players on the attacking team to our agent. Regarding agent2d, the amount of goals

is similar, so we can conclude that at the worst our agent performs the same as the NPC, but with the Helios and Autmasterminds, we see an exponential increase in goals scored.

5.4 3vs3 Environment

Here we discuss the performance of our agent in a 3vs3 environment, where he has 2 Helios teammates and is against 3 Helios opponents. Our agent used a DQN using two hidden layers with 256 and 64 units respectively, we also used the flag Fullstate, since this flag improves the learning while using the low-level state space.

We analyzed the performance of our agent, seeing the percentage of episodes that ended in goal and seeing if this improves throughout learning. This is done so we can guarantee that he is learning a better policy over time. This can be seen in Figure 5.21, where we can see we were able to achieve around 27% episodes ending in goal and we can conclude that the agent was able to learn in this environment as he improves his performance over the course of training. As expected he loses some performance when compared to the 2vs2 environment, as in this environment using the flag Fullstate we get around 45% of episodes ending in goal. This happens because in a 2vs2 environment we have fewer players in-game being a less challenging problem making the policy learning easier.

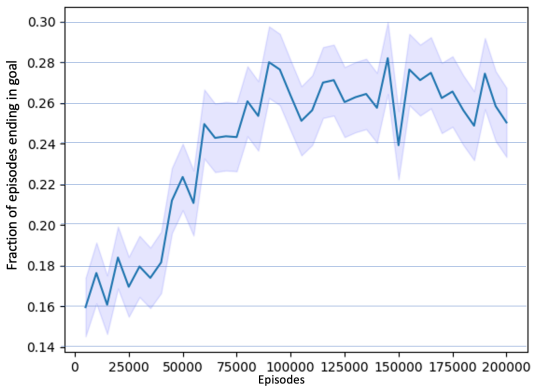


Figure 5.21: Percentage of goals in a 3vs3 environment

Following that, we do an in-depth analysis of the performance of our agent, comparing the number of goals he scores when compared to his team and the number of assists our agent had. This can be seen in Figure 5.22, from the around 310 goals scored, our agent scored 250. This means we still see the trend observed in Figure 5.15 in a 2vs2 environment using the same type of Helios teammate, where our agent is the player that scores more goals. We also analyze that the number of assists doubles from the 2vs2 tests, which is to be expected as there are more teammates capable of scoring goals, so there exist more possibilities of teammates in favorable scoring positions.

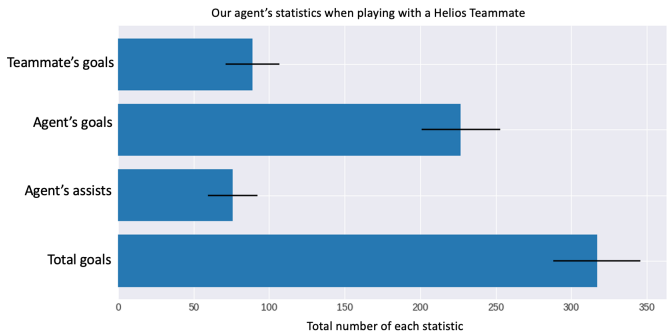


Figure 5.22: Agent's statistics in a 3vs3 environment

Once again we tested the average number of passes our agent does, to gauge if our agent learns to do everything alone or works as a team to improve results. The results can be seen in Figure 5.23.

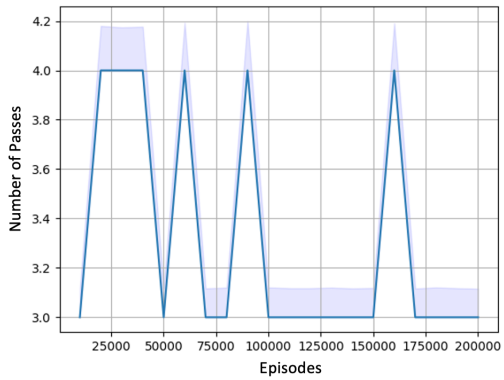


Figure 5.23: Number of passes our agent does in a 3vs3 environment

Unfortunately, we see the same trend as the 2vs2 environment, where our agent, quickly learns to do fewer passes and he ends training doing around 3 passes per episode. Again one reason for this might be the fact our agent scores with a higher percentage than his teammates and as such, it yields better results to score himself than relying on his teammates.

5.4.1 Discussion

We can conclude that our agent, using the best approach for the 2vs2 environment, can still learn to score in a 3vs3 environment. More tests are needed to assess if changing hyperparameters or the number of layers affects positively the performance of the agent. Nevertheless, he still scores more than his teammates, not relying on his team to score goals. So, this approach can still be used to learn a good policy for scoring goals in this particular environment.

6

Conclusion

Contents

In this chapter we conclude on our work, stating the work done and conclusions that were achieved, ending with the system limitations and future work.

6.1 Conclusions

Many solutions and approaches have been done to solve the problem of learning a policy in HFO, either using the low-level or high-level action space. Research and solutions that show good results and expand the state-of-the-art.

One problem that we found on the already done research, was that none of them test to see what the agent is learning. So our agent could learn to just pass the ball and wait or he could actively try to score goals and create better chances.

So this thesis tries to address this problem, firstly by doing extensive testing on the elements of reinforcement learning when learning a policy in high-level action space, seeing what works best and what hinders the performance. Testing the impact of changing what kind of network we use, what kind of reward function to use, if the flag Fullstate has a big impact on the performance, and seeing what features to use. After this concluding what the best solution is.

Afterward, we did an in-depth analysis of the performance of all the team members, assessing the impact our agent had on the team's performance. We do this by comparing the goals scored by our teammate and our agent and seeing how many of our teammate's goals came from a pass from our agent. Seeing if this performance is the same when using other types of teammates. Finally, we compared what the team's performance is when we use our agent and an NPC type or a team of two of the same NPC type seeing if our agent has a positive effect on the team.

We could conclude that using low-level state-space improves the learned policy, we concluded that of the tested reward functions the one that gives a larger reward for scoring and a larger penalty for not scoring also improves the learned policy. The flag Fullstate improves dramatically the learned policy, even without changing other aspects of the learning process the flag enhances the policy. Our agent at the worst performs the same as an NPC, but in most other cases, like working with a *Autmasterminds* or *Helios*, he improves the scoring of the team scoring more than his teammates. On the flip side of this, the agent, in these situations where he scores more, also passes less, doing most of the shooting, while in the case of having an *agent2d* as a teammate he shoots less and assists more. Nevertheless, we can conclude that our agent has a positive effect on the team as a whole since the team performs better with him. We still saw this trend in an environment with more players, testing in a 3vs3 environment having 2 *Helios* teammates. Concluding that our agent still scored most of the goals and that this approach learned a good policy in this environment.

6.2 System Limitations and Future Work

As we alluded to in the related work, there have been some approaches using the low-level action space, but unfortunately, this environment is far too complicated and as such can only be used to learn a policy with very few players in-game. So it would be interesting to test some solutions to learning with more players in-game and do an in-depth analysis to see what kind of actions our agent takes.

Another area we could do this in-depth analysis is when dealing with the ad-hoc teamwork problem, using a state-of-the-art approach like PLASTIC-policy [31], where they need to create a strategy to learn a policy when working with an unknown team. We could test the actions our agent is taking and what attitude towards scoring goals when working with an unknown team he is having.

Another improvement that could be made, is to analyze the environment where we use different NPC types that can be gathered by the binaries of the 2013 Robocup 2D competition. Seeing how the agent changes his behavior and analyzing more in-depth each NPC type to see why his behavior changes. Additionally, we could analyze changing the defending team to see how our agent molds his actions to score more goals, studying again the different NPC types to discover why the agent changes his behavior.

One limitation of our work is when learning a policy in a 3vs3 environment when do not do extensive research in this environment so the policy learned and the performance observed could be improved.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara, "Robocup: A challenge problem for ai," *AI magazine*, vol. 18, no. 1, pp. 73–73, 1997.

[3] P. Stone, G. A. Kaminka, S. Kraus, and J. S. Rosenschein, "Ad hoc autonomous agent teams: Collaboration without pre-coordination," in *Proceedings of the Twenty-Fourth Association for the Advancement of Artificial Intelligence AAAI Conference on Artificial Intelligence*, 2010.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[7] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[8] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[9] S. Kalyanakrishnan, Y. Liu, and P. Stone, "Half field offense in RoboCup soccer: A multiagent reinforcement learning case study," in *RoboCup-2006: Robot Soccer World Cup X*, ser. Lecture Notes in Artificial Intelligence, G. Lakemeyer, E. Sklar, D. Sorenti, and T. Takahashi, Eds. Berlin: Springer Verlag, 2007, vol. 4434, pp. 72–85.

[10] P. Stone, R. S. Sutton, and G. Kuhlmann, "Reinforcement learning for robocup soccer keepaway," *Adaptive Behavior*, vol. 13, no. 3, pp. 165–188, 2005.

[11] M. Hausknecht, P. Mupparaju, S. Subramanian, S. Kalyanakrishnan, and P. Stone, "Half field offense: An environment for multiagent learning and ad hoc teamwork," in *AAMAS Adaptive Learning Agents (ALA) Workshop*, May 2016.

[12] H. Akiyama, T. Nakashima, K. Yamashita, and S. Mifune, "Helios2013 team description paper," *RoboCup*, 2013.

[13] H. Akiyama and T. Nakashima, "Helios base: An open source package for the robocup soccer 2d simulation," in *Robot Soccer World Cup*. Springer, 2013, pp. 528–535.

[14] M. Malmir, S. Boluki, and M. Simchi, "Aut-masterminds team description paper 2013," 2013.

[15] W. Masson, P. Ranchod, and G. Konidaris, "Reinforcement learning with parameterized actions," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[16] M. Hausknecht and P. Stone, "Deep reinforcement learning in parameterized action space," in *Proceedings of the International Conference on Learning Representations (ICLR)*, May 2016.

[17] ——, "On-policy vs. off-policy updates for deep reinforcement learning," in *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI Workshop*, July 2016.

[18] Z. Fan, R. Su, W. Zhang, and Y. Yu, "Hybrid actor-critic reinforcement learning in parameterized action space," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 2279–2285. [Online]. Available: https://doi.org/10.24963/ijcai.2019/316

[19] E. Wei, D. Wicke, and S. Luke, "Hierarchical approaches for reinforcement learning in parameterized action space," in *2018 Association for the Advancement of Artificial Intelligence AAAI Spring Symposium Series*, 2018.

[20] L. A. Celiberto, C. H. Ribeiro, A. H. Costa, and R. A. Bianchi, "Heuristic reinforcement learning applied to robocup simulation agents," in *Robot Soccer World Cup*. Springer, 2007, pp. 220–227.

[21] L. A. Celiberto, J. Matsuura, and R. A. Bianchi, "Heuristic q-learning soccer players: a new reinforcement learning approach to robocup simulation," in *Proceedings of the Portuguese Conference on Artificial Intelligence*. Springer, 2007, pp. 520–529.

[22] S. Barrett and P. Stone, "Cooperating with unknown teammates in robot soccer," in *Workshops at the Twenty-Eighth Association for the Advancement of Artificial Intelligence AAAI Conference on Artificial Intelligence*, 2014.

[23] R. Munos and C. Szepesvári, "Finite-time bounds for fitted value iteration." *Journal of Machine Learning Research*, vol. 9, no. 5, 2008.

[24] J. P. Santos, “Playing soccer with unknown teammates,” Master’s thesis, Instituto Superior Técnico, 2021.

[25] P. M. Santos, J. G. Ribeiro, A. Sardinha, and F. S. Melo, “Ad hoc teamwork in the presence of non-stationary teammates,” in *EPIA Conference on Artificial Intelligence*. Springer, 2021, pp. 648–660.

[26] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[27] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, “Dueling network architectures for deep reinforcement learning,” in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[28] M. Lanctot, E. Lockhart, J. Lespiau, V. F. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. D. Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. W. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis, “Openspiel: A framework for reinforcement learning in games,” *CoRR*, vol. abs/1908.09453, 2019.

[29] L. Yu and H. Liu, “Feature selection for high-dimensional data: A fast correlation-based filter solution,” in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 856–863.

[30] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, 12 2014.

[31] S. Barrett and P. Stone, “Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork.” in *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*, vol. 15, 2015, pp. 2010–2016.