Playing Soccer with Deep Reinforcement Learning

Francisco Delgado
franciscodelgado9187@gmail.com

Instituto Superior Técnico, Lisboa, Portugal

November 2021

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. No current solution does an extensive analysis on changing various aspects of the learning process, like changing what features to use (type of state space and feature selection) and other changes. We do this analysis, 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 performance 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 and how many passes and assists our agent has. Afterward, testing with other types of teammates, 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 improves the team.

Keywords: Reinforcement Learning, Artificial Intelligence, Deep Reinforcement Learning, Half-Field Offense

1. Introduction

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, agents should be capable of learning any specific environment 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 [6]. This domain has been used extensively as a testbed for research. 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 in the future the viability of using agents or robots in more delicate problems involving teaming up with humans or other agents [12] in helping for example during catastrophes.

In many of these problems, it is not only important the outcome of the problem, but also what he did to achieve them. To analyze what needs to be improved to better fine-tune the agents’ actions. With this in mind, our work introduces this new idea of scrutinizing the agent’s actions, to examine why the outcome happened.

1.1. Problem Description

This thesis focus on the problem of learning in the Half Field Offense (HFO) environment. Where we have our agent working with 1 teammate, trying to score a goal, and a defending team consisting of 2 players trying 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 Reinforcement Learning (RL) and 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, either changing what features to give to the agent, test different networks and others, explained in more detail in chapter 3.

Most of the solutions created for this problem like
learn in this domain either using function approximation or discretizing the state space and doing a more traditional solution, like Q-learning. Additionally, they test their solution 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 seeing who scores more goals and plays better, examining in detail what each teammate does regarding the team’s performance.

1.2. Contributions
The main contributions of this thesis are the following: Extend on previous work using discrete action spaces with no parameters in Half Field Offense; Test the effect of changing different elements of the learning process 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; 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; Extend on this in-depth analysis to other types of teammates, to test if he always learns to do the same actions or if he adapts according to his teammates.

2. Background
2.1. Neural Networks
Neural networks accomplish a certain task by passing data through units. Each unit takes multiple inputs and outputs a weighted sum of its inputs. An activation function is then applied to the output and 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 and passes 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 comprised of multiple units.

A neural network learns by optimizing a metric, given by a loss function. The loss function informs the network of the error between the expected output and the real output. Then this value is backpropagated through the network, starting from the output layer to the input layer, where every weight will be updated using gradient descent to minimize the value of the loss function.

2.2. Markov Decision Process
A Markov Decision Process (MDP) is defined by a tuple \((S, A, P, R, \gamma)\) 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 \(s_t\); \(\gamma\) 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 \(s_{t+1}\).

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 \(\pi^*\) 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
We use RL [13] 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 \(\pi^*\), that dictates at each state \(s_t\) what action \(a_t\) will yield the highest reward. This agent will interact with the environment 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 \(\pi\) thereafter. The importance of the received rewards loses value the further into the future they are received, for example, a reward \(r\) 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 \(\gamma\), which is between \([0, 1]\). We can define the value function as the expected discounted reward achieved by a policy \(\pi\) starting at state \(s_t\) and following \(\pi\). To calculate this we use a function seen in Equation (1) that is the expected sum of received rewards, discounted by \(\gamma\).

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

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 \(\pi\), can be calculated using...
Equation (2). Where 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 \( \gamma \).

\[
Q^\pi(s_t, a_t) = E_\pi[r_t + \gamma V^\pi(s_{t+1})]
\]  

(2)

2.3.1 Q-Learning

Q-learning [14] is an off-policy algorithm which means that the policy used to collect samples might be different from the optimal policy (\( \pi^* \)), in other words, it learns the optimal policy independently of the agent’s actions. It uses the Equation (3) to update his 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 \( s_{t+1} \).

\[
Q^\pi(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a \in A} Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)\right)
\]  

(3)

The value \( \alpha \) 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 \( Q^\pi(s, a) \), to find the optimal policy \( \pi^* \). We know that if every pair state-action is visited infinitely often the policy will converge to the optimal. 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 \( \epsilon \)-greedy strategy, where our agent explores by doing a random action with \( \epsilon \) probability and exploits the policy (i.e. selects the action that has the highest Q-value) with \( 1-\epsilon \) probability. An improvement to this is the \( \epsilon \)-annealing strategy that starts with a high value for \( \epsilon \) but slowly decreases it, so the agent starts by exploring more, but as he starts to learn a policy he starts exploiting more.

2.3.2 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) [8]. 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 a Q-value for each possible action that the agent can execute, an example can be seen in Figure 1. 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 \( \theta \). The network learns to approximate the Q-value of a pair state-action \( (s, a) \), based on the network parameters \( \theta \), this is denoted as \( Q(s, a|\theta) \). 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 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 \( \theta' \). After each iteration, we only update one of them while keeping the other constant. 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:

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

(4)
The loss function is in Equation (4), 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 \((\theta')\). One advantage of this 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.

2.4. Half Field Offense
Half Field Offense is a subtask of the Robocup 2D [6] 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.

In [5] they presented this novel subtask of Robocup, called Half Field Offense. This subtask emerged since it is an easier task to learn than Robocup. In this task, an offense team must work together to score against a defending team that includes a goalkeeper. We can define how many players will be in each team up to a maximum of 4 vs 5 players in the environment, where 4 players attack and 5 defend, in our work we use a 2 vs 2 environment. This task is only played from the half field line to the goal line, and each episode (each attack) 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 episode is ended; Captured by defense: The defense team catches the ball. After one of these events occurs, a new attacking play is started, where the position of all players (attacking and defending) and the ball are randomized.

2.4.1 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 action pass that needs the number of the teammate to pass. The mid-level actions have discrete and parameterized actions. Finally, the low-level actions that are all parameterized, and the agents need to select the parameters to execute them.

2.4.2 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. A low-level state space, where we have much more features, where each one gives less information. Both these spaces are provided by the simulator. In the case of a 2 vs 2 environment, we have 24 features in the high-level state space while we have 86 for the low-level.

3. Implementation
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 firstly 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 we did to better grasp the impact our agent had on the team as a whole.

3.1. 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 2 vs 2 scenario, where we have one Helios Teammate against two Helios Opponents. Where we learn 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 as such make the optimal policy harder to learn.

3.1.1 Action Space
The full action space regarding all actions that can be executed is: Move; Shoot; Dribble; Pass; Go to Ball; Reorient. Only the action pass needs a parameter, that needs the teammate’s number to pass. Since 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.

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. This made the learning extremely difficult and time-consuming. So, we forced some restrictions on the agent, making some actions illegal and legal, to help him learn the environment. Additionally, when selecting the best Q-value for
the next state do we not take into account illegal actions, removing them from the possible choices. When exploring we had to redefine the probability of executing the actions, this is modeled according to [7]:

\[
\pi(s, a) = \begin{cases} 
0 & \text{if } a \notin L(s) \\
\frac{1}{|L(s)|} & \text{if } a \in L(s) 
\end{cases} \quad (5)
\]

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

3.1.2 State Space

We used both the high and low-level state-space provided by the simulator. In 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 features kept by [11], having 15 features instead of the normal 24.

3.1.3 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 had. All networks use mean squared error as their loss function and use Adam optimizer with a learning rate of 0.00025. The network with two layers had 256 and 64 nodes respectively and regarding the networks, with three layers we had one network with 256 in each layer and another with 512 in each layer. The networks used are based on the networks done by our colleagues [10, 11]. We used these networks as it was already proved in the work of our colleagues that they could learn in this environment.

3.1.4 Hyperparameters

The hyperparameters used are the following: Learning Rate set to 0.00025; \(\epsilon\) starts at 1 decreases to 0.01; \(\gamma\) set to 0.99; Batch-size set to 32; Target network update frequency updates after every 75 episodes; DQN updates at every timestep; Final exploration timestep (i.e timestep \(\epsilon\) becomes 0.01) after 5 million timesteps. The hyperparameters are based on the ones by [8], 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 same hyperparameters for all of our tests.

3.1.5 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 reward function that rewards 1000 for scoring, -1000 for a terminal state other than goal, and -1 otherwise. 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 a reward function that rewards 1000 for scoring and -1000 for terminal states other than goal and 0 otherwise. The advantage of these more simple reward functions is there is no reward shaping that might make the agent do unexpected actions, having a wild behavior that is not intended. Additionally, we wanted to test the impact of using reward shaping, as seen in [4], where the agent receives rewards based on the outcome of their actions. This reward function rewards our agent if the ball approaches goal, if he approaches the ball, rewards the first time per episode he is close to the ball, and finally rewards him for scoring.

3.1.6 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 we are against 2 Helios agents, we tested these modifications: Test the impact of changing what features are used; Test the impact of using the high and low-level state space; Test the impact of changing the DQN network, testing three different architectures; Testing the impact of the reward function, using a simpler reward function, changing only how much is given to the agent and a complex reward function; Test the impact that using the Flag Fullstate, which removes noise from features, has on the learned policy of our agent.

3.2 In-depth agent analysis

After concluding what the best solution is for the 2vs2 environment, we analyzed in-depth the performance of our agent in this solution. This analysis came because we wanted to know what role our agent had in the team’s performance. So we analyzed how many of the goals came from our agent scoring and compared it with our teammate. Then we analyzed how many goals came from an assist of our agent and how many passes on average our agent performs per episode, to see if our agent learned to just play by himself.

Afterward, we calculated the performance of a team consisting of our agent and a Helios teammate and compared it with a team of 2 Helios teammates, in both, they are against 2 Helios teammates. Ad-
ditionally we added a new NPC strategy, the *Autmasterminds* and trained our agent in the environment of having either a *agent2d* or a *Autmasterminds* teammate against 2 *Helios* opponents and compared with a team of 2 *agent2d* or 2 *Autmasterminds* versus 2 *Helios*.

### 4. Results

#### 4.1. Evaluation Procedure

First, for the task of learning a policy for the high-level action space in section 4.2 and 4.3, we train the current solution for 200 thousand episodes. Our solution tests a wide variety of elements of the learning process seeing what works best. We train our solution in the 2vs2 environment where we have 1 *Helios* teammate and 2 *Helios* opponents, for 200 thousand episodes saving a snapshot of the networks’ weights after every 5000 episodes of training, so later we can gauge the evolution of the policy to evaluate the solution. We run 10 different agents for each solution, for each saving the snapshots of the networks’ weights, so later when examining the performance we average the performance of the 10 agents to minimize discrepancies that might occur like weight’s initialization.

We did the same procedure for training an agent with other teammate types as can be seen in section 4.4. Where we load the network weights after being fully trained (i.e. after 200 thousand episodes of training), where we did no exploration and we did not modify the networks’ weights and 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.

#### 4.1.1 Metrics

Our performance measure for the initial tests is the one used by [3], where we examine the percentage of goals. We assume that is a good measure since the agent learns the environment the more goals the team scores. In the following graphs, we average the 10 runs, where we analyze the rollout (i.e. the fixed DQN weights) of each agent after 5000 episodes until the end of training of the 200 thousand episodes of training. For each rollout we run the networks without exploration and no change to the weights, running the agents for 500 episodes. In the following plots, the grey area beside the main-line is the confidence interval, where we assume the performance has a normal distribution and use a 95% confidence value. For the tests in section 4.4, for the tests regarding the evolution of the policy, we run the same tests as explained before. Saving the snapshots and then loading the networks after every 5000 episodes of training and taking the needed metrics. For the rest of the tests, we load the networks after being fully trained (i.e. after 200 thousand episodes) and averaging how many goals, assists, and passes our agent does and how many goals our teammate scores.

#### 4.2. Learning a policy using high-level state features

To deal with this situation of learning in these circumstances, we employ a model similar to the one used by [11], where we use 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, each having 256 nodes.

Our first test was running an agent using the explained DQN using the base features, using the hyperparameters explained in section 3.1.4. The results are seen in orange in Figure 2. We noted that the performance of the agents was around a 22% chance of scoring.

We then tested using a subset of features, using the features used by [11]. 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. The comparison between base features and subset can be seen in Figure 2.

![Figure 2: Comparison between subset and base features](image)

#### 4.3. 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. Where we have 86 features instead of the 24 of high-level state space.

#### 4.3.1 Changing DQN Structure

Firstly we used the same network and hyperparameters as our high-level state space approach examining what results are achieved. Changing only the
input layer to take more features as input. So, we ran a DQN with 3 hidden layers and 256 units each. The results are in Figure 3, having around 25% percentage of goal.

Since our colleagues using different networks had better results in the same environment, we wanted to test more solutions to improve performance. Seeing as we had more features to process, we thought that increasing the number of units of each hidden layer might improve results, being better at generalizing the domain. So, we increased the units in each hidden layer to 512. The results from this network are presented in Figure 4, noting no significant difference, still being around 25%, being slightly worse. Since increasing the number of units, decreased the performance of our agent, we tested a simpler network to see the impact this had.

4.3.2 Changing Reward Functions

Our first test was changing the reward for an action that ends in goal to 1 and the rest to 0. The results show that the reward was too small and scarce for the agent to learn anything significant. The agent showed no improvement in training.

Then we tried to increase it to 1000 when scoring and 0 the rest. We noticed that by giving bigger rewards the agent could indeed learn and improve its performance. His performance still did not match the performance seen in Figure 5.

Next, we extended the reward function to give a penalty of -1000 for actions that ended in terminal states other than goal. The results with this setting were, as expected, better than the last test. It reached around the same results of Figure 5 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.

Finally, we tested a more complex reward function, the one used in [4], 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 performed worse than expected, performing 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 worse results. Another theory is that this reward function has trouble learning without the Fullstate flag.

We have in Figure 6, 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. We reached the same conclusions as before, where the reward functions that reward 1000 for scoring and -1000 for not scoring are the best. There seems to be no statistical significance between them, so any one of them can be chosen as the best reward function. So, we choose the one that also gives a -1 reward for non-terminal actions. Where the results are shown from left to right are Reward 1 for scoring 0 otherwise; Reward 1000 for scoring 0 otherwise; Reward 1000 for scoring, Reward -1000 for a terminal state other than goal and 0 otherwise; Complex Reward shaping; Reward 1000 for scoring, Reward -1000 for a terminal state other than goal and -1 otherwise.

Figure 6: Comparison of each reward function.

4.3.3 Fullstate Flag

We wanted to test what impact the flag Fullstate had, so we ran our best solution changing only the value of the flag. Keeping the same reward function and network architecture. The obtained results are what was expected, as the performance improved dramatically, as no noise is given to the features received it no longer misled our agent’s actions. As it can be observed in Figure 7, achieving around 45% of episodes ending in goal.

4.3.4 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, the use of a subset of features improves his performance, this might happen because some of these features with noise might harm the ability of our agent to learn. 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 patterns too complex and end up harming the policy learned. 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 in turn makes the agent’s learning impossible. If we penalize our agent for not scoring a goal he performs better while penalizing him for each action does not change significantly the performance. In the case of an environment not using the Fullstate flag, the complex reward functions are not recommended since the noise disturbs the policy. Finally, the flag Fullstate increases our agent’s performance dramatically since he does not have slightly wrong information about the environment.

4.4. In-depth Analysis

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. To analyze this we see the number of goals our agent scored when compared to his teammate. 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 7: DQN with 2-hidden layers 256 and 64 units respectively using the fullstate flag to remove noise from features

Figure 8: Agent’s statistics when running in a 2vs2 environment
In Figure 8 we ran each of our trained agents, in the configuration explained in section 4.4, loading the network weights after 200 thousand episodes of training. Using the procedure explained in section 4.1. We noticed of 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.

Next, we analyzed if our agent played alone or if he tries to play with his team, but just shoots more than the teammate. So, we calculated the number of passes on average that our agent does. Analyzing the average over the course of training after each 5000 episodes of training.

![Figure 9: Number of passes over the course of training](image)

We notice by analyzing Figure 9, that our agent starts passing more due to still exploring more, but as learning goes on he learns to pass less. Ending with around 10 passes per episode. We can conclude that he shoots more but passes less. 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. Since every episode has around 165 actions, around every 16 actions our agent does one pass.

### 4.4.1 Other teammate types

In this section, we compare other teams' 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 a teammate type agent2d. We did the same in-depth analysis as before. This can be seen in Figure 10.

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. Scoring only around 60 of the 230 total goals scored. 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. 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. 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 11.

![Figure 10: Agent’s goals and assists in a team with an agent2d teammate](image)

![Figure 11: Comparison between teammate types and our agent](image)

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.

### 5. Conclusions and Future Work

This work addresses two main problems, the first trying to do an extensive analysis on changing various aspects of the policy learning in the HFO envi-
environment, examining why and what is best for this domain. The second is introducing a novel way of looking at the agent’s performance, by looking directly at the agent’s actions and comparing them with his teammate. We used this approach to look at performance to evaluate our best solution regarding the first extensive analysis. While also examining when changing to other teammate types to gauge how important our agent is to the team when we change the team he is playing with. Concluding with an examination on the results and why it might happen.

There are interesting ways to expand on this work, such as using the in-depth analysis for the problem of ad-hoc teamwork problems, and when testing the policy with an unknown team, analyzing more concretely the agent’s actions. Another way to improve on this work is to test against other types of NPCs either as teammates or opponents and see how our agent molds his actions to better suit those NPCs.

Acknowledgements
I want to thank my supervisor Alberto Sardinha that helped me through the course of this thesis, his knowledge and guidance made it possible the completion of this thesis. Thank my friends for supporting me through the years. Finally thank my parents and brother for being there for me, making possible the conclusion of this work and my studies.

References


