Cooperative Agents for Rocket League using RLBot

Project Report

Diogo Silva[*]
diogo.goncalves.silva@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisbon, PT

Jorge Brown[†]
jorge.brown@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisbon, PT

Tiago Melo[‡]
tiago.melo@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisbon, PT

ABSTRACT

Team-Based Multiplayer Video Games are games which force the cooperation of players in order to achieve a common goal, commonly by defeating an opposing team of players. These types of games represent a big chunk of the current industry market and as such a lot of effort has been dedicated into their development. A problem they face, however, is the fact that a lot of times players may not be able to fill an entire team, and as such, most games employ the usage of bots to sub in for the missing human players. Unfortunately, many of these agents are very limited and are often unable to coordinate and accomplish impromptu team tactics. One such game where this happens is Rocket League - a vehicular soccer game. By developing a team of agents capable of seamlessly coordinating with each other and doing tactical squad plays we aim to fix this issue and give human players more capable artificially intelligent teammates capable of fulfilling the void left by the lack of real human colleagues.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; **Multi-agent planning**; **Multi-agent systems**.

KEYWORDS

Multi-Agent System, Artificial Intelligence, Video Games, Team Coordination

ACM Reference Format:

Diogo Silva, Jorge Brown, and Tiago Melo. 2018. Cooperative Agents for Rocket League using RLBot: Project Report. In *Proceedings of* . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The evolution of the video games industry has fomented the creation of a diverse array of real-time cooperative challenges which force players to coordinate towards a common goal. Although these multiplayer games are designed for Human-Human cooperation, a lot of the times players may require the assistance of Artificial Intelligent teammates in order to have a "full team" and be able to play. This may occur due to the lack of ability for the players to find enough colleagues to fill a team, or, more notably, due to players leaving mid-matches, for example, due to connectivity issues.

A good representative of such games, where team play and coordination is paramount to success is **Rocket League** - a 3D, third-person perspective, soccer video game in which each player controls a car in a team environment with the goal of scoring goals by bumping the ball into the opposing team's net, while protecting their own. Although the game offers several game modes, including a *1v1* mode, our focus will be directed towards matches of *3 versus 3* players, since these are the ones which require the most coordination and team tactics. Rocket League quickly soared to one of the most popular competitive online sports games and has sparked the interest of many intelligent agents developers, fomenting a healthy community of not only players but also avid programmers who pit their own bots against each other in organized tournaments, not unlike those seen in *Robot Soccer*.

Naturally, *Rocket League* is a game that lends itself to diverse team strategies and technical prowess. By developing a team of intelligent agents in this multi-agent system, capable of coordinating and fulfilling team roles, we can then use our agents to play with human players, seamlessly filling the void that the lack of a human teammate may create. Ultimately, our goal is to develop a set of bots capable of implementing effective team strategies and working together to achieve victory.

In terms of implementation, the team looked into a framework called RLBot. This tool allows, not only the development of bots using an array of languages (such as Java, C++ and Python), but also provides a graphical interface that allows us to load in our bot scripts directly into the game, enabling us to play matches utilizing our bots. Using it the team built three different bots - **Primus**, **Capitão** and **Neurocket** - using different architectures and decision making techniques in order to compare results and touch upon few of several possible paradigms of A.I. development for team-based games.

In terms of report structure, during section 2, we will talk more in depth about *Rocket League* as a game, how it works, what the structure of each match is and so on. For section 3 we delve deeper into the *RLBot* framework, what it is capable of, what its architecture is and what some of the most famous bots are (and which we chose to pit our own bots against). In section 4, 5 and 6 we thoroughly describe and present statistics and results about our first bot built, **Primus**, our second bots, **Capitão** - which employs direct

[*]IST-ID: 98776
[†]IST-ID: 82416
[‡]IST-ID: 98773

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
, ,
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

communication and delegation between the team - and our third bot **Neurocket**, which was created using *Reinforcement Learning*. Finally, during section 7 we provide some conclusions and final thoughts about our bots and their relevance,

2 ROCKET LEAGUE

2.1 Game Description

"Rocket League is a high-powered hybrid of arcade-style soccer and vehicular mayhem with easy-to-understand controls and fluid, physics-driven competition" [1]. Rocket League is a team-based robot-soccer-like game in which teams of players controlling cars have to compete against each other in order to score goals against the enemy's net, whilst avoiding being scored on. **The game can be downloaded for free on the game's Epic Games Store page** (https://www.epicgames.com/store/en-US/p/rocket-league). The game can be played in matches of 1v1, 2v2, 3v3 and 4v4 (also called "Chaos Mode"), but by far the most popular and most played mode, both casually and in competitive E-Sports tournaments, is 3v3 [2] (and as such, our bots were built to work best in this environment).

It should be noted that the game does offer bots built by its developers ranging in level of difficulty - Rookie, Professional and All-Stars - and these were used as a baseline of performance for our own bots. However they are very lacking, especially when compared to bots built by *RLBot's* Community [3], both in terms of aptitude (lacking the ability to perform advanced moves like Speedy Kickoffs and Aerials) and coordination.

2.2 Player Movement and Controls

Rocket League is a game that employs pseudo-realistic physics. Players **drive** around in cars (of which there is a variety of, each with slightly different steering) the field. In terms of baseline inputs, players can **accelerate**, **decelerate** (and drive backwards), **jump**, **double jump**, **tilt their car** while in the air (which, when combined with the double jump, allows for things like **front flips** which increase the car's velocity) and **boost**.

The car's velocity has a capped limit which can only be overwritten temporarily when boosting. Additionally, to use the boost, players need to have enough **fuel**. Fuel goes from 0 to 100 and is only used when boosting. Players can get more fuel form certain "*boost pads*" which are scattered across the map and that either fully or partially refuel the player's car.

The player can perform a jump and then, whilst in the air, perform a second jump which further increases the car's altitude. The double jump is recharged each time all four wheels of the car touch a surface (be it the floor, wall or ceiling).

In terms of driving, the games' matches take place in a closed off field (as seen in the following section), meaning that there are walls and a ceiling surrounding the field on which the player can drive (granted they have enough speed. The cars have a certain amount of grip which allows players to drive on walls with some ease, but this does not apply to the ceiling since the player can only drive on it very momentarily before starting to fall. If two cars collide they either bump off of each other, or if one of them is moving at a very high speed, the one with lower speed is demolished, being removed from the field for a few seconds - we call this **demolitions**.

All of these baseline inputs can be combined to perform advanced movements such as *WaveDashes* and *SpeedFlips*, giving the game a very high skill ceiling and allowing players to exploit the physics engine to benefit themselves and their team. Of special importance are **Aerials**. By combining the usage of boost, car tilting, jumps and mechanical skill the player is able to "fly" around, giving the game a whole new dimension of vertically. This technique is not easy to perform initially, with many new players struggling to control their cars midair, but it is second nature to more professional and advanced players as seen in figure 1.

Figure 1: Example of a car performing an aerial to hit the ball before it hits the ground.

2.3 Environment

Rocket League takes place in 5 minute matches. Each player in the team is placed in one of several preset starting positions which vary in distance to the ball (with both teams mirroring each other). The ball is placed statically on the center of the field. After a team scores the world is reset back to this initial state of "kickoff".

In terms of the field, the game takes place in a closed off arena with dimensions (measured in in-game units) of 4096x5120x2044 and with walls and a ceiling on which the player can drive. There are several boost pads in predetermined locations which are re-instanced after being picked up after a given time. The overall layout of the map can be seen in figure 2.

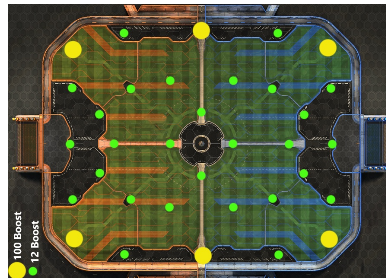

Figure 2: Overview of the field where Rocket League matches take place.

For a full categorization of the environment, we can say that it is **Multi-Agent**, **Stochastic**, **Sequential**, **Dynamic**, **Episodic** and **Partially Observable**.

3 RLBOT

3.1 Framework Description

RLBot is a framework which allows for the development and direct injection of custom made bots into Rocket League in a variety of programming languages, of which we chose **Python**. This framework is fully endorsed by **Psyonix** (the developers of the game) and only permits bots to be used in offline or LAN (local) matches. Additionally, RLBot also provides debugging tools, allowing bots to display messages and figures (lines and other polygons) for developers to better understand why each agent is choosing to do it's play. The framework even permits their visual configuration (using the wide range of cosmetics available in Rocket League) and can be installed following the tutorial in this https://rlbot.org/ (https://rlbot.org/).

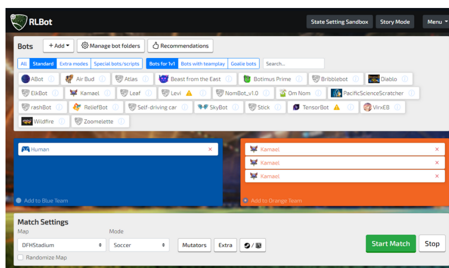

Figure 3: The GUI of the RLBot framework which allows for matches between bots to be created.

3.2 Extra Packages

Besides providing some out-of-the box utilities, there are also several packages that can be added to further increment RLBot's capabilities. Of note we have **RLUtilities** which provides extra information about the game's current state and a lot of *linear algebra* and *vector calculus* methods. Additionally, **RLBotTraining** allows for the creation of training scenarios which can be used to test out the bot's reactions and their plays in a controlled setting (rather than during the chaos of a real match). Finally we have the extra software used for reinforcement learning: **RLGym**[7] which is an OpenAI Gym-style environment for Rocket League and **TensorFlow** which is a machine learning library.

3.3 Well-known Bots

RLBot has a booming and very much alive community of members and developers which organize tournaments, discuss strategies and share a passion for creating AI for this game. Included in the RLBot framework are some of the bots the community has already created and that can be used to pit our own bots against. Figure 4 showcases some of these bots and their overall ranking of ability (S being the best).

Out of these, we handpicked a few to test our bots against:

- **Diablo** - An A-Tier bot built in Python capable of a lot of agressive plays.
- **Botimus Prime** - An S-Tier bot built in Python with both 1v1 and 3v3 strategies and a lot of handmade maneuvers achieving a well-balanced mix of defense and offense.

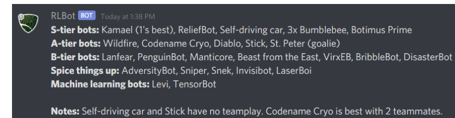

Figure 4: Ranking of the included bots made using RLBot.

- **ReliefBot** - An S-Tier bot built in Java which actually employs intra-bot communication and coordination.
- **Psyonix AllStars** - Whilst included in the RLBot install, these are actually the game's default bots (built by the game's developers) at their highest difficulty. They were **not** made using RLBot and hence have no ranking and were used as a baseline of performance.

4 PRIMUS

4.1 Description

The first bot we built was dubbed **Primus** and it was built using Python with *RLUtilities* for linear algebra computations and some ball prediction/simulation features. It's a very standard bot in terms of implementation compared to other RLBot bots such as *Botimus Prime* and *Kamael* (which were the primary inspirations for Primus' behavior). Figure 5 showcases the physical appearance of **Primus** in-game. A full-match between **Primus** and **ReliefBot** can be found at **https://www.youtube.com/watch?v=Y0trXzKRuYI**.

Figure 5: Primus' appearance when on the blue team.

4.2 Architecture

At its core **Primus** is a fairly complex **Reactive Agent** with **predicate logic** inserts. The way it works is the agent has, at all times, a **Play** that it is trying to perform. When a *play* is finished or something calls for it to be interrupted (for example an emergency situation like avoiding being demolished might call for it to stop what it's doing in order to prevent death), the agent analyzes the current environment (stored within the **GameInfo**) to choose what to do.

Primus does not employ explicit communication with its peers. Instead, it **assumes that all of its teammates are rational**, able to contribute to the team without communication and smart to the point of achieving implicit team play (e.g it assumes that its teammates know when they should stay back on defense, or be more aggressive). An advantage to this is that it allows Primus to play with other bots or even humans without any issues. It should be stated that even without explicit communication, emergent team

plays do arise, like one of the bots setting the ball up, whilst the other waits and switches out at the right time.

4.3 GameInfo

We dubbed the class that stores the current environment's *state* as **GameInfo** and is responsible for reading the packets received from the game and updating it's own information. Each agent has its own GameInfo storing information such as what the agent's team index is, what are its teammates and opponents, what's the position of it's net and the opposing team's, whether the agent's team has scored or got scored on (in order to reset the episode), what the position of each of the boost pads is and whether or not their active (stored in a Pad object) and possibly most important, an array containing several predictions of where the ball will be within a threshold of time and with a step delay between predictions. Additionally it also possesses support functions to detect collisions between cars and predict car positions similarly to the ball predictions.

4.4 Plays

The agent's *actions* were dubbed **Plays** and they employ rather complex behaviors that allows **Primus** to interact with the environment. All plays derive from a base **Play** class and they can even be composed of several other plays or inherit from other plays. Each play has a sequential set of actions to perform and contains a **step** function which sets what the agent's controls should be (i.e what input to perform such as accelerating, steering, and so on) at the current step of the play's execution. They also all have a *finished* flag which is set when the play has finished executing and a method to check whether or not they can be interrupted. Plays can be subdivided into **Strikes**, **Defensive**, **Kickoffs**, **Dribbles**, **Utility** and **Actions**. It should be noted that the specific controls that each play outputs at each given time had to be set manually and as such the team allocated a lot of time into creating these plays. The full list of available actions can be found in **Appendix A**.

4.5 Decision Making

Primus' decision making is three-fold and was very much inspired by professional player's decision making and conditional reasoning. Firstly, during the *getOutput* main loop, Primus checks whether it has a play selected. If not, then it calls the *choosePlay* method which is responsible for analyzing the current state and pick the most appropriate play. If not, then Primus starts by checking whether it's in a collision course with another car. If so, and if it's current action can be interrupted, then it'll drop the current play and do a double jump to avoid the collision. If no collision is abound, then Primus performs the next step in it's current play, assigns the controls to the play's output, and checks if the play has finished. If so, then it calls *choosePlay* again.

Within the *choosePlay* method is where most decision making takes place. This method works in a "decision flowchart" fashion, choosing what to do based on the state. First if the agent is not on the ground, it'll pick the **Recovery** play in order to land safely. Then, it analyzes the ball's and team's positions. If the ball is perfectly on the center, this means we're in a *Kickoff*. In this case, Primus checks all its teammates. If it's the closest to the ball, it'll pick which kickoff to perform (based on its distance and angle to the ball). If it's the second closest, it'll go **Refuel** and in case it's the farthest away, it'll **GoToNet** (getting small boosts along the way), in order to prepare for a defense.

After kickoffs, Primus' second priority is to get boost if it's low on it. Otherwise, it'll create some ball predictions and compute all likely interceptions both it's team and it's enemies'. It then checks if the best intercept is his and if so, it'll pick which type of strike to make (if in position to strike, otherwise it'll pick a clear instead), based on both on it's and the enemies' distance to the net and ball. The third fold of decision comes from deciding which of the possible interceptions is best. This is done using predicate logic (each possible strike has a predicate that states whether or not the agent should go for it), and using *lookup tables* containing combos of boost and throttles to compute the goodness of the interception. This is why Primus' behaviour can be classified as *pseudo-hybrid*. It's reactive at its core, but it does employ a small mix of lookups and predicate logic into it's decision making.

Continuing the decision flow, Primus checks if it's the closest to it's own net. If so, and if there's a lot of enemies on it's side of the field, it'll go to net. If the ball is dangerously close to the net, it'll clear it, otherwise, it'll prepare for a setup at a safe distance. If it's not the closest to net, it'll just go setup to wherever it thinks there will be an interception opportunity (based on the ball predictions). Full decision trees to better visualize Primus' decision making can be found in **Appendix B**.

4.6 Results

For benchmarking, two types of tests were performed. Initially, when debugging each play as it was being implemented, the team created some **RLBotTraining** scenarios, specifically for defense, offense and kickoffs. This was mainly done for debugging and at this point Primus still did not have any decision making. For actual tests and performance, we pitted Primus against other bots in 3v3 matches. In particular, we had Primus face off versus, in this order, **Psyonix All-Stars**, made by Rocket League's developers, **Botimus Prime**, which has a similar structure to Primus, **Diablo** a very offensive-based bot and **ReliefBot** which employs explicit intra-agent communication. The first set of bots was chosen since they're the best the game has to offer by default. The remainder were chosen due to their ranking within the RLBot community and number of tournaments they won in the 3v3 category.

Overall, **Primus** obtained an **84.3%** win rate (20 out of 24 matches). It won all matches against both *Psyonix All-Stars* and *Diablos* with quite a margin of goals (around 8 goals of difference per win). This indicates that Primus can consistently perform at minimum "*A-Tier*" within the RLBot community. Looking at the graphs produced (available in **Appendix C**), we can tell that both these bots struggled to keep up with Primus' agression, noted by the fact that most score attempts ended in a goal. When pitted against *Botimus* and *ReliefBot*, both S-Tier bots known for their astounding performance in 3v3 matches, Primus put up a good fight, and achieved a **66.6%** win rate against them, managing to beat them in most matches, but still suffering a few losses. Still, with this we can confidently say that **Primus can play and win against "S-Tier" bots, the highest tier in the RLBot community**. These bots are much more

well-rounded and can actually both save and prevent Primus from making shot on goal attempts better than the latter two. This resulted in much more even matches. More in depth statistics such as number of shots and saves can be found in **Appendix C**.

During plays we noted that where Primus mainly faltered were defensive maneuvers. This, however, seems to be an overall issue with most bots. There is one exception in the RLBot community - the bot **St. Peter** - but this is a specialized goalie bot who has no attacking plays and was designed to sit on net all match just blocking shots. Other than that, Primus excelled at attacks and was able to pull of a decent level of team play and apparent coordination, but there were moments where due to syncing issues two bots would go for the ball at the same time even when they shouldn't. In an attempt to avoid this, we moved on to our next bot - **Capitão**.

5 CAPITÃO

5.1 Description

Once we had a well-performing bot with implicit deliberation with its peers, we set out to design and develop a bot capable of assigning tasks explicitly, leveraging off of RLBots's communication modules and using message protocols. As might already be obvious to the reader, Capitão conceptually maps to a pirate crew. There is an indisputable leader responsible for sending "stances" to its Marujos which, programmatically, are also instances of Capitão, but for ease of reference will henceforth be declared as such. Traditional team-based decision making like voting schemes would be prone to a lot of overhead in message passing and would most likely result in delayed decisions. We have come to realize that given the extremely dynamic nature of our environment, such strategies could very easily hinder the performance of an explicit communication team model: by the time the team had come to a consensus, it would already be too late. By agreeing on who the leader is beforehand, this negotiation is shifted to the beginning of the game, which is usually just the countdown, and so has a lot less impact on each agent's performance. On the other hand, having a clear leader centers deliberation on a single agent, which can lead to performance issues as the number of teammates increases. We find that, for the tested examples, a team composed of 3 Capitão Bots is still able to perform reasonably well, as discussed ahead. A sample match between Capitão and Diablo can be found here.

Figure 6: Capitão's appearance when on the orange team.

5.2 Architecture

The emphasis of this bot lies on the communication between different instances, as well as testing delegation and assignment strategies by a single entity. As such, this bot does not offer new mechanics or actions, in the sense that it employs the same actions as Primus. We can decompose the core architecture of Capitão into 3 major blocks: policy, strategy and actions, as suggested by Figure 16. Much like Primus, Capitão is a **hybrid, vertically layered and stateful** agent. It selects an action that it is trying to perform at every game tick and, when it finishes or the action is interrupted, the agent re-evaluates the assumed state (henceforth referred to as "stance"). Just like Primus, it collects and updates information about its environment using GameInfo packets, which can hence be considered its sensors. In a broad sense, since every agent has access to the same information about the game, the leader Capitão is able to perform as much of an informed decision as any other. That being said, the leader uses a policy to decide upon its own and its Marujos' stances. Using this stance and the current action, each agent is then able to update its own action if possible or, otherwise, store the stance and, once available, start performing the action it was tasked with.

It is also worth highlighting the structure of Capitão's `get_output` function which is called by the framework many times per second to fetch an agent's controls. It starts by updating Capitão's inner state of the game, which is recorded using an object of type `GameInfo`: teammates and enemies' positions, ball position, direction and predictions, and so on. This ensures the agent is up to date with the fast-changing environment and is not using stale information to deliberate. Afterwards, the agent decides whether or not it should change it intention. Being in a fast-paced, dynamic environment Capitão adopts an open-minded commitment, in the sense that it only pursues actions while they are possible and make sense. Thus, we consider two scenarios in which it is worth to change intentions: whenever a car touches the ball and if the ball is entering a "danger zone". Car touches on the ball change its trajectory, demanding a re-evaluation of the plan. As for the latter scenario, should an agent notice the ball is too close to the goal, it breaks free from the General Defense and re-evaluates. Once it has re-adjusted its own strategy, Capitão prepares assignments. Afterwards, if there is no action selected, Capitão will assign a stance to himself and his fellow Marujos using the policy using TMCP. According to the selected stance, each agent will then select the corresponding action using his strategy. Lastly, the action is executed, returning a set of controls that are sent to RLBot, which promptly sends them to Rocket League.

5.3 Communication

Communication between agents is achieved using Matchcomms [4], a low level system that allows broadcasting messages between participants in a match supported by RLBot. Put simply, Matchcomms is a server that is also ran whenever the RLBot framework is running, exposing asynchronous websockets which allow for the exchange of messages between all the participants in a game. Whenever a message is received, the server ensures it is broadcast to the remaining players, as represented in Figure 7.

A layer above Matchcomms lies TMCP - Team Match Communication Protocol. TMCP is an attempt at standardizing inter-bot communication [5], supported only by Python bots. Relying on a JSON server, TMCP supports sending a pre-defined set of JSON messages, with only a few alterable parameters. The envisioned

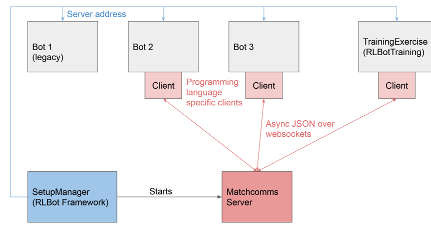

Figure 7: Matchcomms architecture diagram.

use-cases when Matchcomms was designed were primarily notifying other bots of one's own intentions, which resulted in only a few very specific types of messages to be developed like manifesting interest in going for the ball, a boost, a demolition and so on. This is the antithesis of what we are trying to achieve: a bot that lets other bots know what they should be doing. Furthermore, we do not wish to send the same message to everyone, which also conflicts with the broadcast implementation offered by RLBot. The first priority was to allow targeted messages between the Capitão and a given Marujo: the leader should be able to tell the right bot what it should be doing. This required overriding the meaning of `index` already supported by TMCP. As such, whenever a bot receives a message it checks if it is meant for him: if the message's index is his own and the team from the sender is his own. We ignore the fact that it is possible to spoof Capitão messages for our current version (*id est*, an enemy player impersonates the enemy Capitão and purposely sends bad stances to the enemy Marujos). Afterwards, it was time to delegate specific stances. Distinctions between defenses, assigning kick-offs and strikes demanded us to create a new set of macro variables. Every TMCP message sent is of the type `BOOST`, and the payload contains the macro for the stance to be assumed. This is one of the main reasons why we chose stances: macros (which boil down to just integers) are far easier to send than Action classes, which would require serialization and would mean larger messages. Currently, Capitão is able to assign `ATTACK, DEFENSE, BOOST, CLEAR, PREEMPTIVE_DEF` stances to its Marujos. Moreover, TMCP advises to only send a packet when something has changed., and this is enforced by not allowing messages to be sent unless the time difference is superior to 0.1 seconds. So, Capitão keeps a tab of last assigned stances, and only sends messages should they be updated. However, this cannot hold for a stance such as the Kickoff given each bot has a varying initialization time. This poses a major problem at the beginning of each match: since some bots are instanced before others, some queues take longer to be ready and messages that are sent in this time span are not received. So, at the beginning of each match, when Capitão may delegate Kickoffs to Marujos that were not ready yet. To circumvent this, every Marujo assumes an `UNDEFINED` stance when instanced, and will be stuck consuming messages until this state is changed. On the other hand, Capitão will frequently send Kickoff messages to the right Marujo during the Kickoff pause, ensuring someone is tasked to kickoff the ball. Nonetheless, it is worth highlighting that in some situations (although rarely, as seen in the clip) Marujos do not Kickoff. They still read stale messages or begin the round by continuing the last assigned action. This occurs when the message queue still has messages respective to the last round or the last action was not interruptible and they are "forced" to finish it before taking on a new one. We consider this to not be much of a problem given its low frequency.

Lastly, for the topology of the "network" we decided to adopt a *publisher subscriber* strategy. As a general rule and in order to avoid a single agent having to send and receive messages on every game tick, the Capitão is **only** responsible for sending actions and each Marujo is **only** responsible for receiving and parsing them. This way we can easily include a section in the main loop in which all communication is done and depending on the role each agent has act accordingly. As hinted in the previous section, using its policy the Capitão will assign a high-level stance to each agent in its team. Then, iterating over every car (identified by index) in his team, it can send the corresponding stance if needed or, if the index matches his, just update his own stance and prepare to act accordingly.

5.4 Roles

As mentioned, every instance is running Capitão code but depending on whether they are the leader or not, each agent will display different behavior. Nevertheless, we believe a Capitão should also be a (albeit privileged) Marujo, in the sense that the way they select an action based on a given stance is the same. To sum up, every agent is both a Capitão by name and a Marujo by structure/code. The core difference is that only of the agents is actually the leader. As for nominating the leader, the agent with the smallest index in the team is considered the Capitão. It is worth noting that this information is not publicly known the moment an agent is instanced: there is still a need for some information exchange. As such, each agent assumes he is the captain until proven otherwise. Only when everyone receives their first GameInfo packet they have access to the indices of their teammates, allowing them to adjust. Once they realize there is someone above them in the hierarchy, they send him an acknowledgement and update their status. This is also particularly tricky when the game begins: the agent prepares to assign Kickoffs and (since there needs to be a 0.1 second delay between messages) only a few ticks later it realizes that it is not his job to assign but rather to be assigned. However, we believe our current implementation deals with this just fine, given the details discussed in the previous section.

5.5 Decision Making

When discussing Capitão's decision making, it is worth covering both the policy of the leader and the policy of each Marujo. Assignments are done by associating an agent's index with a stance, and iteratively instancing and forwarding the appropriate messages to the teammates. The leader policy delegates the closest element to perform the kickoff. Should two agents have the same distance between themselves and the ball, both go for it. Marujos have the responsibility of picking the right Kickoff action (between SpeedFlipDodgeKickoff and SimpleKickoff). Every agent that is not responsible for kicking the ball off is assigned to General Defense. To delegate strikes or clears, the captain starts by calculating the possible intercepts of the team and selects the best. If the Capitão figures the enemy can intercept the ball sooner than the teammate

with the best interception, that teammate is assigned to assume a preemptive defensive stance. Otherwise it assigns a clear or strike stance depending on how well aligned the interception is with the enemy goal. It is worth highlighting that the Capitão never assigns two agents to perform a strike or a clear. This is extremely useful to prevent bumps and clunky, undesirable hits on the ball. If the leader fails to find an appropriate interception for a given Marujo, it tasks it with replenishing its boost or assuming a general defensive stance, depending on its available boost level.

As for Marujo's, as it has been hinted before, in order to save precious computation time in the Capitão each Marujo has been granted some level of independence. In other words, whenever they are assigned to do a clear of the ball or strike, they are free to choose the best fit. Moreover, they do not need to wait for the Capitão's orders to break out of a General Defense stance in case the ball gets dangerously close to their own goal, as previously mentioned. More importantly, Marujos are able to keep themselves on their feet, in the sense that whenever picking an action, regardless of the adopted stance (except for Kickoff, since they were often recovering halfway through) they assume a Recovery action if the car is not on the ground. As for strikes and clears, in broad terms each Marujo considers his orientation, height, direction and speed of the ball as well as how close the ball is to the enemy goal to select between a Close Shot, a Ground Shot, an Aerial Strike or a Mirror Strike. As for clears, it simply chooses between a Dodge Clear or an Aerial Clear, depending on how quickly the intercept can occur. When snapping out of a defensive stance to try and clear the ball, since agents do this without any orders from the Capitão it is very possible that bumps and clunky hits occur. We decide that in such dangerous situations it is better to adopt a "better safe than sorry" mentality and opted to keep this policy. Additionally, it has been empirically noted that these scenarios occasionally lend themselves to interesting Marujo interactions, such as combined clears.

5.6 Results

To benchmark Capitão, we are interested not only on evaluating how well it performs as a general Rocket League bot, but also in witnessing team behavior and measuring coordination. As such, it has been tested against Pyonix Allstar, Diablo and Botimus Prime. The game results have been recorded, as well as some metrics that should give us a better understanding on how well it is performing.

Looking at the game results and charts in 7, there are a couple of key observations that ought to be highlighted. As it was to be expected, as the skill of the opposing bots increases, the percentage of time the ball spent in Capitão's field increases as well, which generally translates into poorer results for Capitão. Moreover, we also notice some similarities between the shapes of the statistics generated by All-Stars and Diablo, meaning that Capitão is able to perform reasonably well against opponents from Psyonix and A-tier bots. Perhaps more importantly, we can clearly see Capitão struggles against Botimus Prime, even having an average of 1 own goal per game. We thus infer that against more sophisticated bots, clunky behavior can start to become a problem, and sharp coordination becomes more of a need. Although the percentage of time mirrors the match against Diablo, the statistics tell a different story. Despite only winning 50% of the games, Capitão is still able to perform a decent amount of strikes, of which most were well directed to the goal. Capitão also displays a level of assists equal to Botimus', which possibly means it relies more on its teammates to score goals against more complex enemies, as expected. We would like to finally point out that we consider these results acceptable, given the large overhead that explicit communication is when compared to its implicit counterpart. It is easy to expect that a "decentralized" bot with no messages and negotiation with its peers to be simpler and, as such, generally better.

6 NEUROCKET

The final agent we developed is called **Neurocket** and it was implemented via Reinforcement Learning. As programmers, as soon as we started this project, we knew that the only way for our agents to reach a performance level similar to the highest levels of play would be via reinforcement learning. With that in mind the first approach was to try to use the algorithm we are most familiar with, Q-learning[12].

6.1 Q-Learning

The Q-Learning, in essence, consists in the mapping of a State-Action pair to an expected value. Traditionally these mappings are stored in a States x Actions matrix. However, Rocket League's environment is continuous, which means that we would need to have a matrix with an infinite number of rows, which is impossible.

The first solution to this problem that came to mind was to discretize the state space so that the number of states would be finite. However, given the high amount of variables associated with the environment, especially in 2v2 or 3v3 modes, it would require an extremely high amount of memory to hold a matrix so big. This can be mitigated by doing a coarser discretization, however the information loss was too high to get a small enough state space.

The second solution was to turn to Deep Q-Networks.

6.2 Deep Q-Networks

Deep Q-Networks(DQN), originally developed by DeepMind to play Atari games[8], are an adaptation of regular Q-Learning where the matrix is replaced with a neural network which makes training in continuous environments much more straightforward.

The DQNs developed by DeepMind were done with a goal in mind, to have the same input(pixels) in all of them in order to preserve the network's architecture in between games. In our case, since the RLBot framework allows us to have direct access to the game state, our inputs are the actual state of the game and not what it has rendered.

Luckily the RLBot community has developed an OpenAI Gym-style[6] environment for reinforcement learning in Rocket League[7], called RLGym, however, unlike the OpenAI gym environments we need to build what they call an ObservationBuilder, which is an object that maps the game state to our network's input, and a reward function.

6.2.1 ObservationBuilder. The first instinct when implementing the ObservationBuilder is to simply concatenate the multiple vectors that describe the ball and the players. However, one needs to take into the account the fact that different players can be on

different teams. For example, if a player in front of the blue team's goal scores a goal it will get a very high reward and so now the model will try to score if in that position again, but, if a player is in that same position but is from the blue team we don't want it to try and score because that would result in an own goal. What this ends up meaning, is that for one of the teams the input values must be mirrored in order to avoid this issue. After mirroring, we can concatenate all the vectors like we intended to originally.

6.2.2 Reward Function. The reward function is one of the most important components of reinforcement learning since it is what will guide the agent's behavior in the future.

Typically, reward functions with frequent and not very significant rewards will lead to sub-optimal agents, even if slightly faster, than sparser and more significant rewards. That being the case, we decided to not reward the agent for minor things like touching the ball or driving towards it, but to reward more meaningful events like goals, saves and shots and to punish getting scored on.

6.2.3 Network Architecture. The network architecture consists in 2 hidden layers and one output layer with the hidden layers being fully-connected and consisting of 256 units using a Rectified Linear Unit(ReLU) activation function and the output layer being a fully-connected linear layer with an output to match each possible action, similar to DeepMind's architecture for beating Atari games[8].

Experimentation with other architecture and parameter variations were not possible due to some issues encountered. This will be discussed in section 6.4.

6.3 Using replays

Rocket League has a built-in replay system that allows games to be saved and replayed at a later date. The format of the replay files consists in a log file of the server that hosted the game, which allows us to obtain the full state of the game and all the input given by the players in a given timestamp if we parse it correctly.

The website ballchasing.com is an online replay archive where 25,882,778 replays are stored at the time of writing, which equates to, considering the game time of 5 minutes, 129,413,890 minutes of Rocket League gameplay to learn from. Even if we limit the replays to the only ones of games of the highest rank in the competitive ladder, we still obtain over 10,000 results.

To utilize all this information, it is necessary to parse the replay files and then to use an imitation learning[9] algorithm. The parsing of the files can be done using a library like oxrock's TrainingDataExtractor that allows us to turn the low-level .replay file into a higher level representation that we could use to feed an imitation learning algorithm.

6.4 Time constraints and hardware issues

Since the Rocket League environment is a complex environment the training time for a competent agent is massive. By lack of expertise in deep reinforcement learning field and lack of proper hardware it was impossible to arrive at an agent of meaningful skill.

Although, we installed all the required drivers and libraries for TensorFlow to utilize the GPU of the computer, its utilization did not go above 5% which seems too low for proper functioning, but it did fill up the 8 gigabytes of dedicated memory. Looking up the problem online didn't yield any meaningful results so we are not sure if the processing power of the GPU is supposed to be left unused. In the end, this meant that each game tick took our application approximately 5 seconds to process which meant that we were unable to reach a single satisfactory model, much less try out different parameters and network architectures. We could have taken a different approach and worked with stacks of game ticks instead of processing them as they come, but the initial goal was to obtain a baseline model to then experiment upon and optimize.

The lack of a model to compare to and the required research on a completely new topic, meant that there wasn't much incentive to dive deeper into imitation learning techniques even though we think there could be a big potential gain by following that route, especially if a hybrid approach was taken where the agent would first be trained via imitation and then via reinforcement learning.

We believe that if we had familiarity with the TensorFlow framework and some real experience with training reinforcement learning agents in a high complexity environment, some of our issues would probably have been avoided and it could have been possible to reach at least one functioning model.

7 FINAL THOUGHTS

In the end, **Primus** ended up being the best bot in terms of raw performance. Whilst both **Capitão** and **Neurorocket**'s implementations have their advantages, the rule of "keeping it simple" proved true. In fact this holds true for most video games. One of the reasons machine learning, for example, isn't too common place in the development of commercial video games is due to it's complexity and unpredictability especially when faced with complex and dynamic environments. Moreover, situations like kickoffs lend themselves to "hard coded" rules. Perhaps a hybrid of reinforcement learning with some pre-coded situations could've worked well given enough time to train. Looking at Capitão's architecture, the idea of having a "team captain coordinating all others" does intuitively seem effective, but having an agent having to keep track of all other's and effectively give out orders to others does overload it with a lot of strain due to all predictions that must be made, which may lead to some less than optimal behaviors. The biggest struggle for both Primus and Capitão, was the creation of **Plays**, due highly in part to the "realistic" physics engine of Rocket League. Without RLBot's community and documentation, this would have been nigh impossible. Still we ended up with a good fleet of bots, some S-Tier, others a bit worse, but all of them ready and excited to compete.

REFERENCES

[1] Psyonix, Rocket League store page, Sep 23, 2020
[2] Liquidpedia, Main page listing an assortment of Rocket League professional tournaments, May 19, 2021
[3] Max Thielmeyer, 'Rocket League' Players Are Competing To Create The Most Skilled Bot, Oct 18, 2018
[4] Eric Veilleux, Matchcomms, Mar 29, 2021
[5] L0laapk3, Team Match Communication Protocol, Mar 29, 2021
[6] OpenAI, Environments for reinforcement learning
[7] RLGym, OpenAI Gym style environment for reinforcement learning in Rocket League
[8] DeepMind, Playing Atari with Deep Reinforcement Learning, Dec 19, 2013
[9] Hussei, Gaber, Elyan, Jayne, Imitation Learning: A Survey of Learning Methods
[10] Dominik Schmid, RLBotTraining's code repository
[11] RLBot Community, RLBot's Wiki
[12] Watkins and Dayan, Q-Learning

Appendix A - Primus & Capitão Actions Table

Jump	Makes car output a jump
AirDodge	Double Jumps in a direction
SpeedFlip	Advanced corkscrew flip used to gain speed and hit the ball
HalfFlip	Quickly reverses direction by double jumping and tilting the car so it lands facing the opposite direction
AimDodge	Quickly turn/flip towards target while not touching the ground
Drive	Drive in a direction at given max speed. Can also drive backwards
AdvancedDrive	Travel to target and use halfflips and wavedashes to gain speed if necessary
Arrive	Arrive at target in given time
Stop	Stop and stay still
SimpleKickoff	Drive to ball and front flip.
SpeedFlipDodgeKickoff	Fast kickoff with speed flips.
Strike	Base strike class. Hit ball towards position
BumpStrike	Bump ball towards enemy net
DodgeStrike	Strike ball by flipping into it
CloseStrike	Strike ball by lightly flipping it
SetupStrike	Strike ball against wall so it bounces to a given position
DribbleStrike	Dribble ball and flip it when appropriate
Dribble	Dribble ball on roof of car
AerialStrike	Strike ball high in the air by flying
DoubleAerialStrike	Attempt to AerialStrike twice in a row
Setup	Go to a position and wait for an opportunity
DodgeClear	Clear ball using a DodgeStrike
BumpClear	Clear ball using a BumpStrike
AerialClear	Clear ball using an AerialStrike
Setup	Go to a position and wait for an opportunity
DodgeClear	Clear ball by using a DodgeStrike
BumpClear	Clear ball by using a BumpStrike
AerialClear	Clear ball by using an AerialStrike
Refuel	Drive towards the nearest available full boost pad
Recovery	Land smoothly and gracefully

Table 1: All Plays Primus and Capitão can perform

Appendix B - Primus Decision Flow Charts

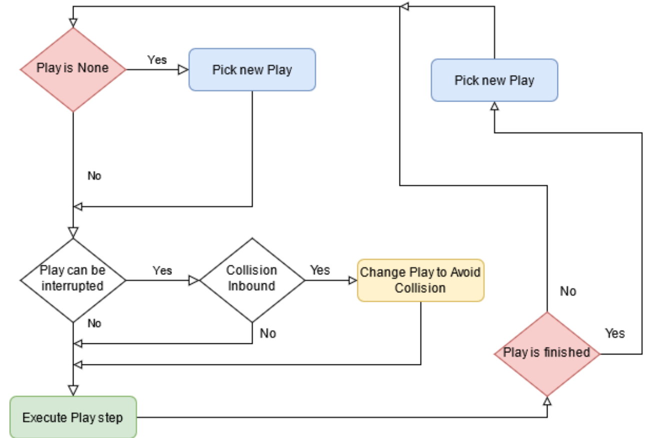

Figure 8: Primus' GetOutput decision flow.

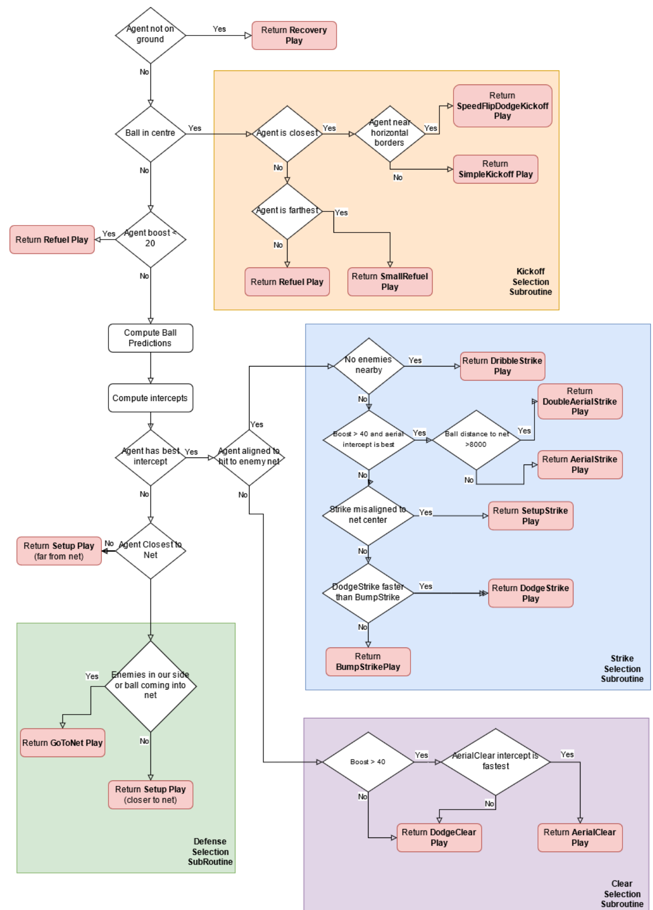

Figure 9: Primus' Choose Play decision flow.

Appendix C - Primus' Benchmarks

All-Stars	7-1	11-2	12-2	10-5	13-1	10-2
Botimus	5-2	4-1	5-3	1-3	3-4	5-6
Diablo	4-1	13-2	8-0	7-0	6-2	10-1
ReliefBot	3-5	4-2	5-4	5-1	3-1	4-6

Table 2: Primus' match scores (where the first number is Primus' team's number of goals and the second is the opposing bot's)

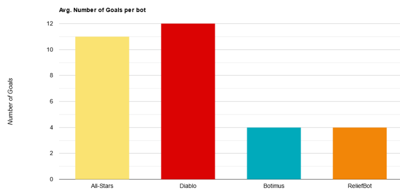

Figure 10: Average number of goals Primus did versus each bot

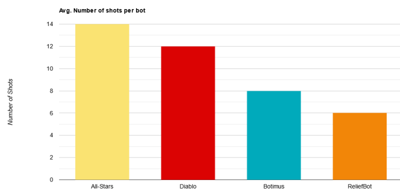

Figure 11: Average number of shots Primus did versus each bot

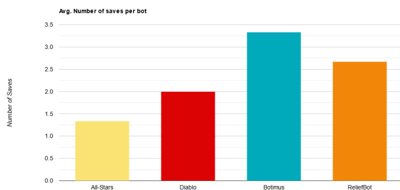

Figure 12: Average number of saves Primus did versus each bot

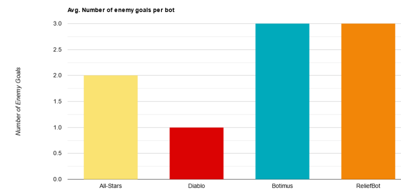

Figure 13: Average number of enemy goals Primus suffered versus each bot

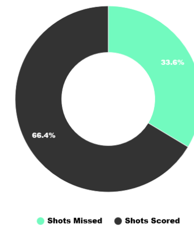

Figure 14: Primus' Percentage of shots scored versus shots missed

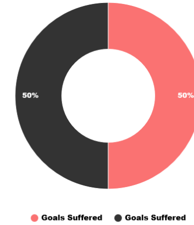

Figure 15: Percentage of shots saved versus enemy goals

Appendix D - Capitão's Benchmarks

All-Stars	7-1	9-0	6-1	8-2	7-1	7-0
Botimus	3-1	0-1	3-0	1-2	3-4	3-2
Diablo	8-0	9-0	6-1	5-1	11-0	7-1

Table 3: Capitão's match scores (where the first number is Capitão's team's number of goals and the second is the opposing bot's)

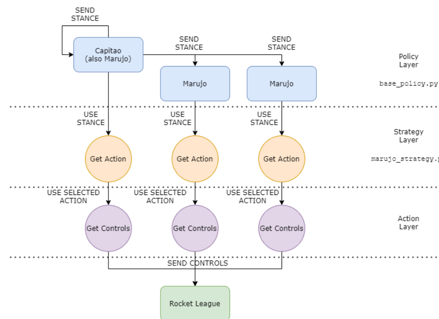

Figure 16: High-level overview of Capitão's architecture.

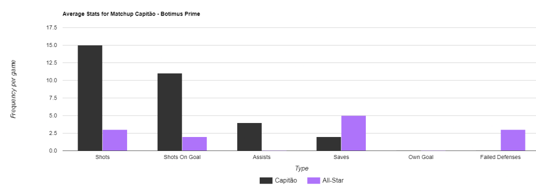

Figure 17: Average stats over the course of three matches between Capitão and All-Star (values have been rounded because of chart generating software)

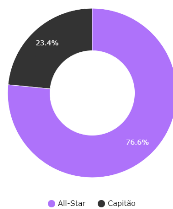

Figure 18: Average time spent on each half of the field over the course of three matches between Capitão and All-Star

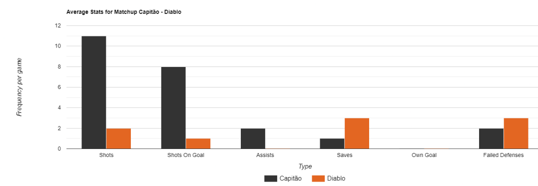

Figure 19: Average stats over the course of three matches between Capitão and Diablo (values have been rounded because of chart generating software)

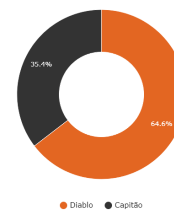

Figure 20: Average time spent on each half of the field over the course of three matches between Capitão and Diablo

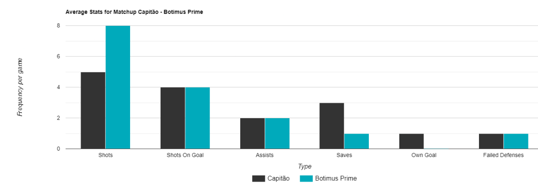

Figure 21: Average stats over the course of three matches between Capitão and Botimus Prime (values have been rounded because of chart generating software)

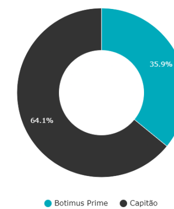

Figure 22: Average time spent on each half of the field over the course of three matches between Capitão and Botimus