Comparison of Reinforcement Learning Hyperparameters in the 3D Game Rocket League

Jacob Critch

Department of Computer Science

Memorial University

Supervised by Dave Churchill

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science

April 2023

Abstract

This dissertation explores the use of reinforcement learning to train a machine learning agent to play Rocket League, a 3D video game.

The study evaluates the effectiveness of various hyperparameters in training agents to control Rocket League at a rudimentary level. The study's methods include a Windows 10 PC, Rocket League, RLGym, Stable-Baselines 3, Bakkesmod, and Python.

In conclusion, a learning environment was successfully established to train reinforcement learning agents to control Rocket League at a rudimentary level, and several agents were trained before being compared and tested. The agents showed promising results, and given sufficient training time and tuning, these agents could become competent players of the game.

Acknowledgements

Thank you to my supervisor, Dr. Churchill, for his advice and guidance through this research and helping to make this dissertation possible. Thank you to the Memorial University faculty for the excellent Computer Science education I received these past years. Thank you to my friends, my family, and my partner Emily for enduring many late nights and my absentmindedness as I attempted to balance schooling, work, and other pursuits.

Table of Contents

List of Tables

List of Figures

Chapter 1:

Introduction

Reinforcement learning is a subfield of machine learning that deals with the development of intelligent agents capable of making optimal decisions in complex environments. It involves training an agent through trial and error, where the agent receives feedback in the form of rewards or penalties for its actions. Reinforcement learning has found applications in several fields, including robotics, gaming, and finance.

One of the most significant challenges in reinforcement learning is the tuning of hyperparameters such as reward functions, learning rates and more. Notably, the design of reward functions that provide appropriate feedback to the agent is especially important in 3D gaming contexts.

This dissertation explores the application of reinforcement learning in a 3D game environment using the popular video game Rocket League. Rocket League is a soccer game played with cars, making it an ideal platform to test the effectiveness of reinforcement learning in a 3D game environment.

The study evaluates the impact of different reward functions and learning rates in training an ML agent to control Rocket League at a rudimentary level. It employs the Rocket League Gym (RLGym), a Python API that treats Rocket League as an OpenAI Gym-style environment for Reinforcement Learning projects, in combination with Stable-Baselines 3 and BakkesMod. RLGym provides a standardized framework for testing the performance of agents in Rocket League, enabling comparisons of different agents' performance.

This dissertation aims to contribute to the growing body of research on reinforcement learning, especially in the context of Rocket League, and its application in gaming and other fields that require strategic decision-making.

Chapter 2:

Background

2.1 Reinforcement Learning

Reinforcement learning is a type of machine learning that involves an agent learning to make decisions by interacting with an environment. In this approach, the agent is not given labeled data to learn from, but instead, it learns by receiving feedback in the form of rewards or punishments based on its actions. The goal of the agent is to learn a policy that maximizes its long-term expected reward.

In reinforcement learning, the agent observes the current state of the environment and selects an action based on its current policy. The environment then transitions to a new state, and the agent receives a reward based on the quality of its action. The agent uses this feedback to update its policy, so that it can make better decisions in the future.

Reinforcement learning has been successfully applied to a wide range of tasks, such as game playing, robotics, and autonomous driving. One of the key strengths of reinforcement learning is its ability to learn from experience and adapt to new situations, making it a promising approach for solving complex and dynamic problems.

2.2 Rocket League

Rocket League is a popular video game that combines soccer and racing. The game is played with cars instead of human players, with the aim of hitting a ball into the opponent's goal while preventing the opponent from doing the same. The game is fast-paced and highly dynamic, with players using boosts, jumps, and flips to control their cars and hit the ball. The game's complexity and dynamic nature make it an ideal candidate for testing the effectiveness of reinforcement learning in gaming.

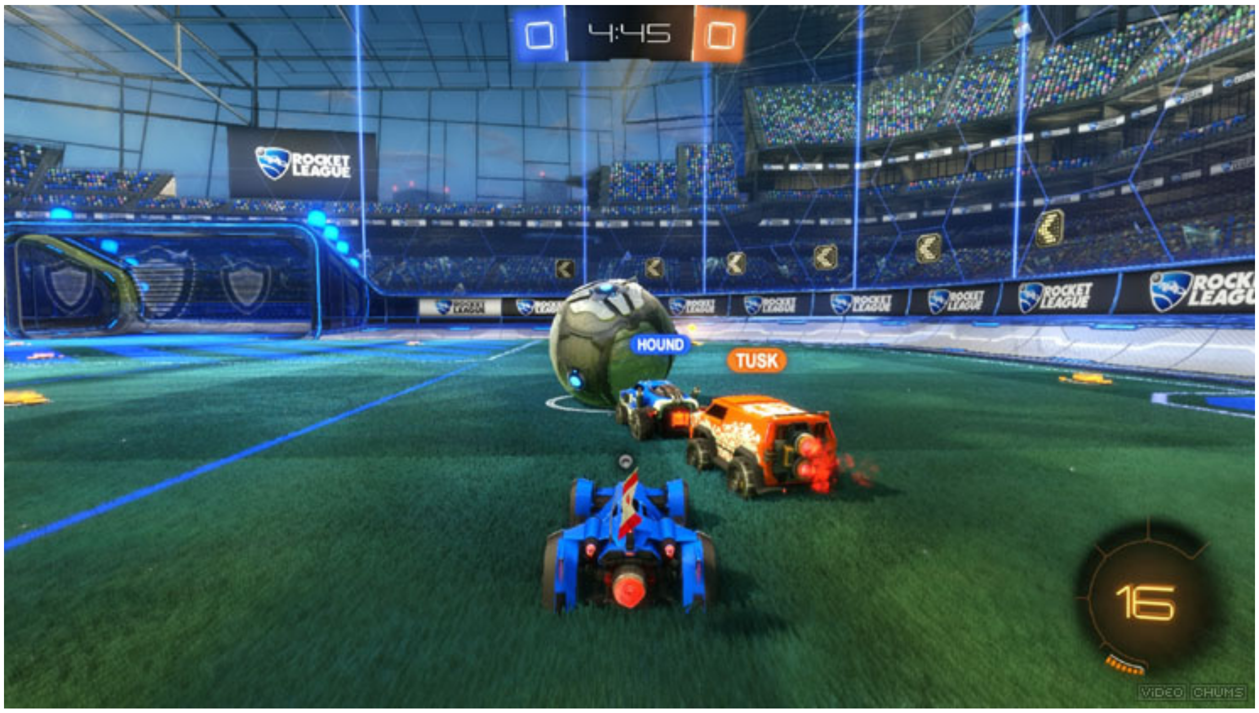


Figure 2-1: Rocket League Gameplay

2.2.1 Description of Play

Rocket League is played in a 3D environment, with players able to move their cars in any direction and perform complex maneuvers. The game involves physics simulations, with the ball and cars reacting realistically to the environment and each other. The objective of Rocket League is to use your car to hit a ball into the opposing team's goal while defending your own goal from the other team's attempts to score. Players can perform a “demolition” by impacting another player at high speeds, causing an explosion and death of said player. The game is typically played in matches of five minutes, with the team that scores the most goals at the end of the match declared the winner. Each team consists of up to four players, each controlling their own car.

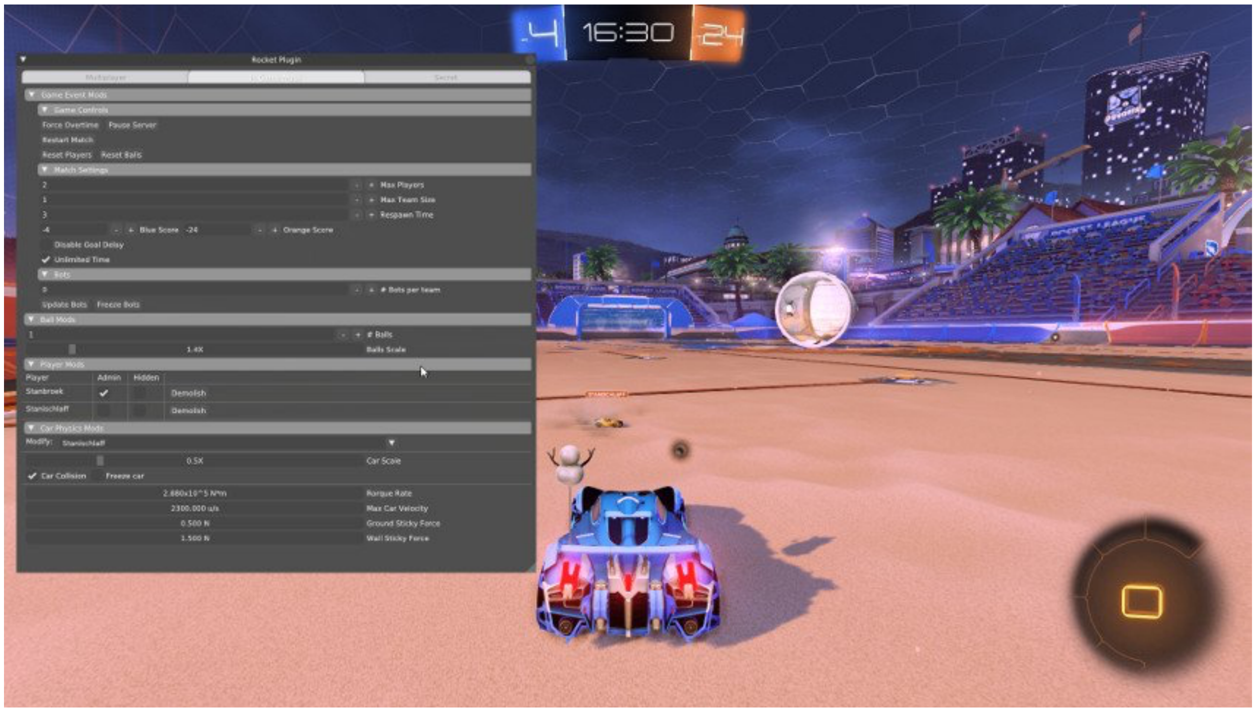
The controls in Rocket League are simple and intuitive, with players using the analog sticks to move and steer their car, and the trigger buttons to accelerate and brake. The game also includes a boost feature, which allows players to gain speed quickly or make quick jumps by using a limited amount of boost meter.

In Rocket League, there are different game modes to choose from, including 1v1, 2v2, 3v3, and 4v4. There are also special modes such as “Hoops" which is played on a basketball court, and "Dropshot" which involves breaking the floor tiles of the opposing team's side to score. Rocket League also includes a ranking system, which allows players to compete against others of similar skill levels. As players win matches and improve their skills, they can climb the ranks and compete at higher levels. One of the most important aspects of Rocket League is teamwork. Players must communicate and work together to score goals and defend their own goal. Successful teams often have players with different roles, such as a striker who focuses on scoring goals or a defender who stays back to prevent the other team from scoring.

This paper will focus mostly on the 1v1 “Training” mode of Rocket League which has a highly customizable environment, and no time limit.

2.3 Bakkesmod

Bakkesmod is a popular third-party mod for Rocket League that adds a range of features and tools to enhance the gameplay experience. Developed by a community of Rocket League enthusiasts, Bakkesmod is free to download and use, and is compatible with both Windows and Mac operating systems.



2 Max Players
1 Max Team Size
3 Respawn Time
-4 Blue Score -24 Orange Score
Disable Goal Delay
✓ Unlimited Time
0 # Bots per team
1 # Balls
1.4X Balls Scale

Player	Admin	Hidden	
Stanbroek	✓		Demolish
Stanlschlaff			Demolish

Modify: Stanlschlaff
0.5X Car Scale
✓ Car Collision
Freeze car
2.880x10^5 N*m Rorque Rate
2300.000 u/s Max Car Velocity
0.500 N Ground Sticky Force
1.500 N Wall Sticky Force

Figure 2-2: Bakkesmod Window in Rocket League

One of the key features of Bakkesmod is the ability to customize training scenarios. Players can create their own training packs, or download and share packs created by others. This allows players to practice specific skills and techniques, such as aerial shots, wall hits, or dribbling. The mod also includes a range of training tools, such as ball physics settings, shot randomization, and replay analysis.

In addition to training tools, Bakkesmod offers several gameplay enhancements. For example, the mod allows players to create and join private matches with custom settings, such as unlimited boost, infinite time, or low gravity. It also includes a variety of game modes, such as "Speed Demon” which increases the speed and boost power of all players, and "Boomer Ball" which replaces the regular ball with a larger, faster ball.

Another notable feature of Bakkesmod is its replay system. The mod allows players to save and analyze replays from their matches, and includes a range of tools for editing and viewing replays. This can be useful for identifying mistakes or weaknesses in gameplay, as well as for creating highlights or montages to share with others. This is particularly useful for researchers and developers looking to create algorithms that can analyze and predict gameplay strategies, or to create intelligent agents that can play Rocket League at a high level.

2.4 OpenAI Gym

OpenAI Gym is an open-source platform that provides a standardized environment for developing and testing reinforcement learning algorithms. The platform offers a wide range of simulated environments, or "environments," that developers can use to train and evaluate their algorithms.

Each environment in OpenAI Gym consists of a set of observations and actions that the algorithm can take. For example, the "CartPole" environment simulates a cart with a pole balanced on top, and the algorithm must learn to move the cart left or right to keep the pole from falling over. Other environments include classic video games like Atari, as well as more complex scenarios like robotics simulations.

OpenAI Gym has the ability to benchmark algorithms against each other. The platform provides a standardized set of performance metrics, or "benchmarks," that developers can use to compare the effectiveness of their algorithms. This makes it easier to evaluate the strengths and weaknesses of different approaches to reinforcement learning, and to identify areas for improvement.

2.5 Stable Baselines 3

Stable Baselines 3 is a popular open-source library for developing and training reinforcement learning algorithms. It is built on top of the TensorFlow library and provides a wide range of pre-implemented reinforcement learning algorithms, making it easier for developers to get started with training their own agents.

The library is structured into three main components: the environment, the policy, and the algorithm. This modular design allows developers to mix and match different components to create custom reinforcement learning models tailored to their specific needs.

Stable Baselines 3 also includes several advanced features for training reinforcement learning agents. For example, it supports multi-environment training, allowing developers to train agents on multiple environments simultaneously. It also includes support for distributed training, allowing developers to train agents across multiple machines or processors.

In addition to its advanced training features, Stable Baselines 3 also provides several tools for evaluating and visualizing the performance of trained agents. These include built-in metrics for tracking training progress, as well as tools for visualizing the behaviour of trained agents in different environments.

The library can be used with OpenAI Gym environments, making it easy to test and compare agents across different environments.

2.6 RLGym

The Rocket League Gym (RLGym) is a Python API that can be used to treat the game Rocket League as though it were an OpenAI Gym-style environment for Reinforcement Learning projects.

2.6.1 Technical Requirements

- A Windows 10 PC
- Rocket League (Both Steam and Epic are suppported)
- Bakkesmod
- The RLGym plugin for Bakkesmod
- Python between versions 3.7 and 3.9.

2.6.2 Game Control

RLGym communicates with a Bakkesmod plugin to control the game while it is running. This enables the Python API to manipulate the game just like a standard Gym environment, with familiar functions like make(), reset() and step(). Bakkesmod also enables RLGym to control the rate at which the physics engine updates while the game is running, so matches inside the game can be run much faster than real-time.

2.6.3 Environment

The default configuration of RLGym will not produce a competent game-playing agent. This configuration is meant as a testing ground for users to quickly verify that they have installed RLGym successfully, and their learning algorithm is working. When the default reward is maximized, the agent should have zero angular velocity at all times. To train a game-playing agent, users will need to configure an RLGym environment with an appropriate Reward Function, Observation Builder, and a set of Terminal Conditions.

At their core, RLGym environments are configured with 3 basic objects: A RewardFunction, an ObsBuilder, and a list of TerminalCondition objects. RLGym uses these objects at every step to determine what reward should be assigned to each action, what observation should be returned to the agent, and when an episode should be terminated. The flowchart in Figure 2-3 depicts how each of these objects is used by RLGym.

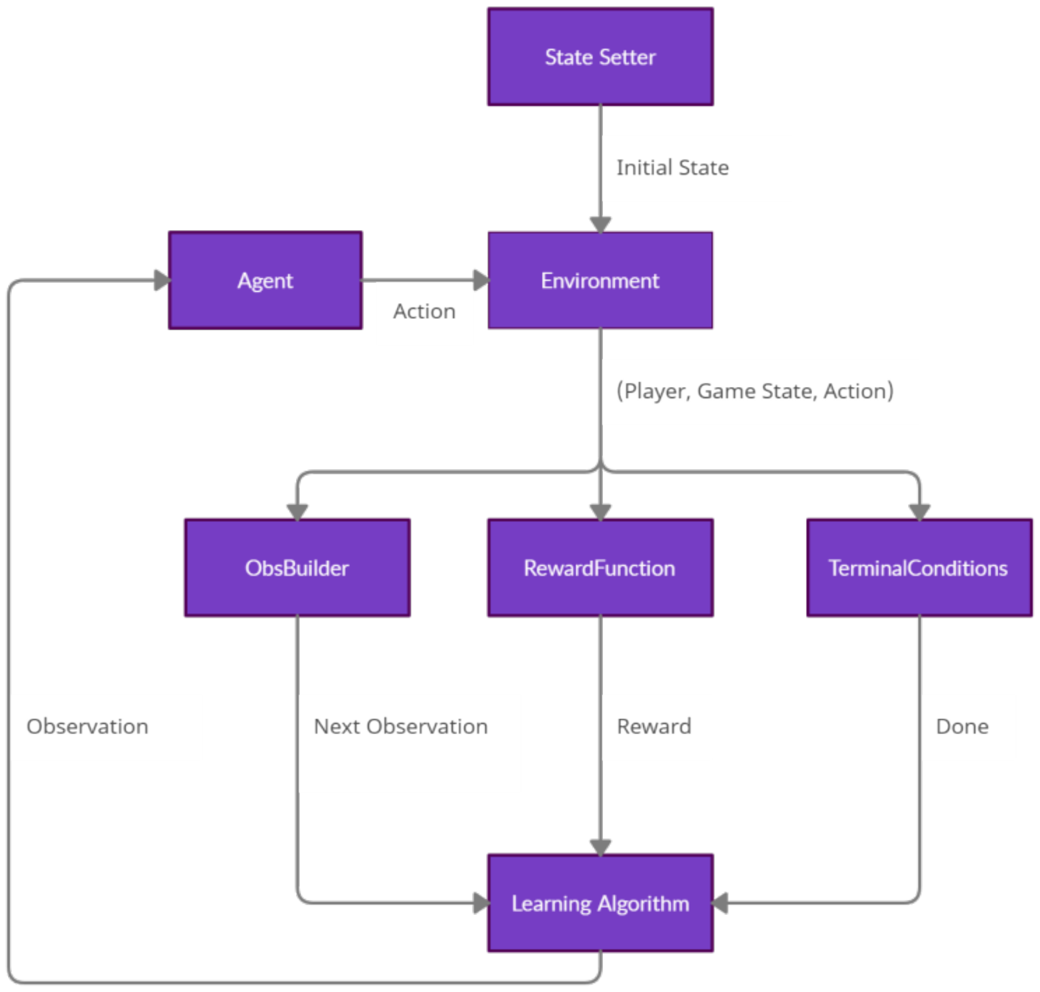


Figure 2-3: Flow of RLGym environment

2.6.4 RLGym Reward Functions

A RewardFunction is an object used by RLGym to compute the reward for each action every step. These methods are called by an RLGym environment during an episode. A *CombinedReward* is an object that allows several reward functions to be used in conjunction, for example, a CombinedReward object might consist of VelocityPlayerToBall,VelocityBallToGoal and EventReward. An *EventReward* object is a reward that is awarded on some specific events (team goals, conceding, shooting, saving, demolition). The following are the common reward functions that are included with the RLGym library:

2.6.4.1 Ball To Goal

Ball to goal rewards are functions that measure some relationship between the ball and the opponent's goal:

Liu Distance Ball To Goal Reward

1. Determine which team the agent is on, and set the opponent's goal as the objective.

2. Compute the normalized distance between the position of the ball, and the center of the opponent's goal. Note that the point returned is in the center of the net, shifted to the back wall inside the net, such that the distance between the ball and the objective can never be zero.

Velocity Ball To Goal Reward

1. Determine which team the agent is on, and set the opponent's goal as the objective.

2. Get the linear velocity of the ball.

3. Determine the difference between the objective (goal from step 1) and the current ball position.

4. Return the scalar projection of the ball's velocity vector on to the objective vector.

Ball Y Coordinate Reward

Incentivize higher ball heights

2.5.6.2 Conditional Rewards

Conditional rewards are rewards issued when a condition is met:

Reward If Closest To Ball

Return True if the current player is the closest player to the ball

Reward If Behind Ball

Return True if the current player is behind the ball

2.5.6.3 Misc Rewards

Velocity Reward

Velocity Reward is a simple function to make sure models can be trained. The velocity reward function returns either the positive or negative magnitude of the agent's velocity, determined by the negative flag in the constructor.

Save Boost Reward

Each step the agent is rewarded with sqrt(player.boost_amount). We take the square root here because, intuitively, the difference between 0 and 20 boost is more impactful on the game than the difference between 80 and 100 boost.

Constant Reward

Provides a constant reward of 1 to agent every step.

Align Ball To Goal

1. Determine which team the agent is on (and by extension which net we should be attacking / defending)

2. Compute defensive reward for when the agent aligns the ball away from their goal

3. Compute offensive reward for when the agent aligns the ball towards the opponents goal

4. Sum defensive and offensive rewards and return the total

2.5.6.4 Player Ball Rewards

Reward functions that measure relationships between the agent and the ball:

Liu Distance Player To Ball Reward

Provides a constant reward of 1 to agent every step.

Velocity Player To Ball Reward

Returns the scalar projection of the agent's velocity vector on to the ball's position vector.

Face Ball Reward

Returns positive reward scaled by the angle between the nose of the agent's car and the ball.

Touch Ball Reward

Returns positive reward every time the agent touches the ball with an optional scaling factor for how high the ball was in the air when touched.

2.6.5 Observation Builders

An ObsBuilder is an object used by RLGym to transform the game state into an input for the agent at every step. Observation builders are used similarly to Reward Functions by the environment. At each step, the observation builder will be called once for every player in the current game state.

2.6.6 Terminal Conditions

A TerminalCondition is a simple object that examines the current game state and returns True if that state should be the final state in an episode, and False otherwise. Terminal conditions can be paired together in a list, in which case an episode will be terminated if any of the provided terminal conditions are evaluated to True.

2.6.7 Action Parser

RLGym expects an array of 8 actions per agent on the pitch. Each action corresponds to one control input: throttle, steer, yaw, pitch, roll, jump, boost, handbrake. The first five values are expected to be in the range [-1, 1], while the last three values should be either 0 or 1. To allow a variety of action inputs while still adhering to requirements of RLGym, we use an ActionParser.

2.7 RLBot

RLBot is a framework that enables custom bots to be developed and used in Rocket League. The framework handles retrieval of game data (such as positions and rotations for the players and the ball), so developers only need to develop the bot's logic and give RLBot the controller inputs we want to use. Most RLBots are hardcoded and do not use machine learning, however since the release of RLGym, there are several RLBots which are ML-based (Nexto, Levi, TensorBot, Element, Eagle). RLBot supports Python (most common), Java, Scratch, C#, Rust, C++, JavaScript, and Go. There is also a graphical view for RLBot: RLBotGUI, a streamlined user interface for RLBot.

2.7.1 Technical Details

RLBot uses an API in Rocket League specifically made for RLBot. This API is activated when the game is launched with the '-rlbot' flag, which simultaneously disables all online play. The RLBotCore dll communicates with the game through this API. Communication with bots are done through sockets (shared memory in the past) which allows support for a wide range of programming languages.

The GameTickPacket is a packet sent to your bot each tick. It contains information about everything that tends to update during the game. For instance, the location and velocity of all cars and the ball, the state of boost pads, the time elapsed.

The FieldInfo contains information that is static throughout the game. For instance, the location and size of boost pads and the location of goals.

Chapter 3:

Methodology

3.1 RLGym Training Environment

3.1.1 Configuration

For the purposes of this project, a virtual machine running Windows 10 was configured with the Epic Games Launcher version of Rocket League, Python 3.7, Bakkesmod, and the RLGym plugin for Bakkesmod.

3.1.2 RLGym Python Script

A Python project was created using a starter template for a Rocket League reinforcement learning bot [1]. This script sets up and trains a reinforcement learning model using the Stable Baselines library and the Rocket League Gym environment. The script imports several packages and libraries including NumPy, RLGym, Stable Baselines 3, and various modules from the RLGym Utils and RLGym Tools packages.

The script sets up a Match object from the Rocket League Gym environment and defines several variables such as the frame_skip and half_life_seconds. It then creates a SB3MultipleInstanceEnv object from the Match object to train the reinforcement learning model on. The environment is wrapped with a VecMonitor object to log mean reward and episode length to Tensorboard, and a VecNormalize object to normalize rewards. The script then checks whether a saved model exists and loads it if it does or creates a new model if it doesn't. The new model is set up using the Proximal Policy Optimization (PPO) algorithm [2]. The PPO object is instantiated with the MlpPolicy [3], the SB3MultipleInstanceEnv object, and various hyperparameters such as the number of epochs, learning rate, and batch size. Finally, the script sets up a callback to save the model every so often.

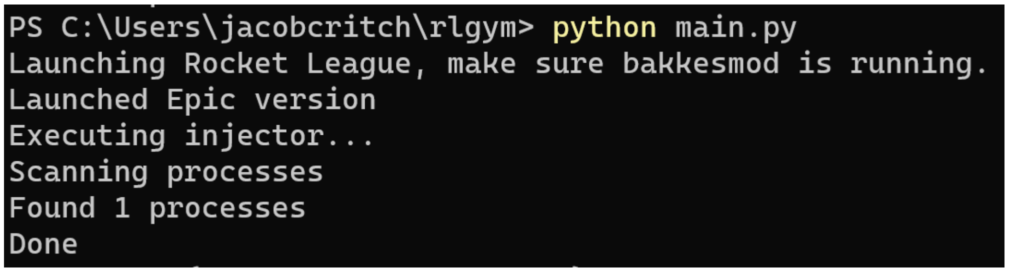
3.1.3 Training Agents

Several models were defined, each using a CombinedReward consisting of different reward functions. The first agent (which will be referred to as 'Velocity Agent') was defined using a CombinedReward consisting of VelocityPlayerToBallReward, VelocityBallToGoalReward, and EventReward. The second agent (which will be referred to as 'Liu Agent') was defined using a CombinedReward consisting of LiuPlayerToBallReward, LiuBallToGoalReward, and EventReward.

To initiate the training of a model, it must be ensured that Bakkesmod is running. Then, the Python script is executed (as seen in Figure 3-1). The Python script launches Rocket League, and uses Bakkesmod to inject into Rocket League using the Bakkesmod RLGym plugin. The game is typically run at a much faster rate than during normal play, so as to speed up the training process (by default, game speed is set to 100). In this case, a frame skip of 8 was initialized to improve performance. The environment was configured for two agents to compete against each other (one on each team) such that both agents are learning concurrently. The model uses the Stable Baselines 3 implementation of the OpenAI Proximal Policy Optimization algorithm

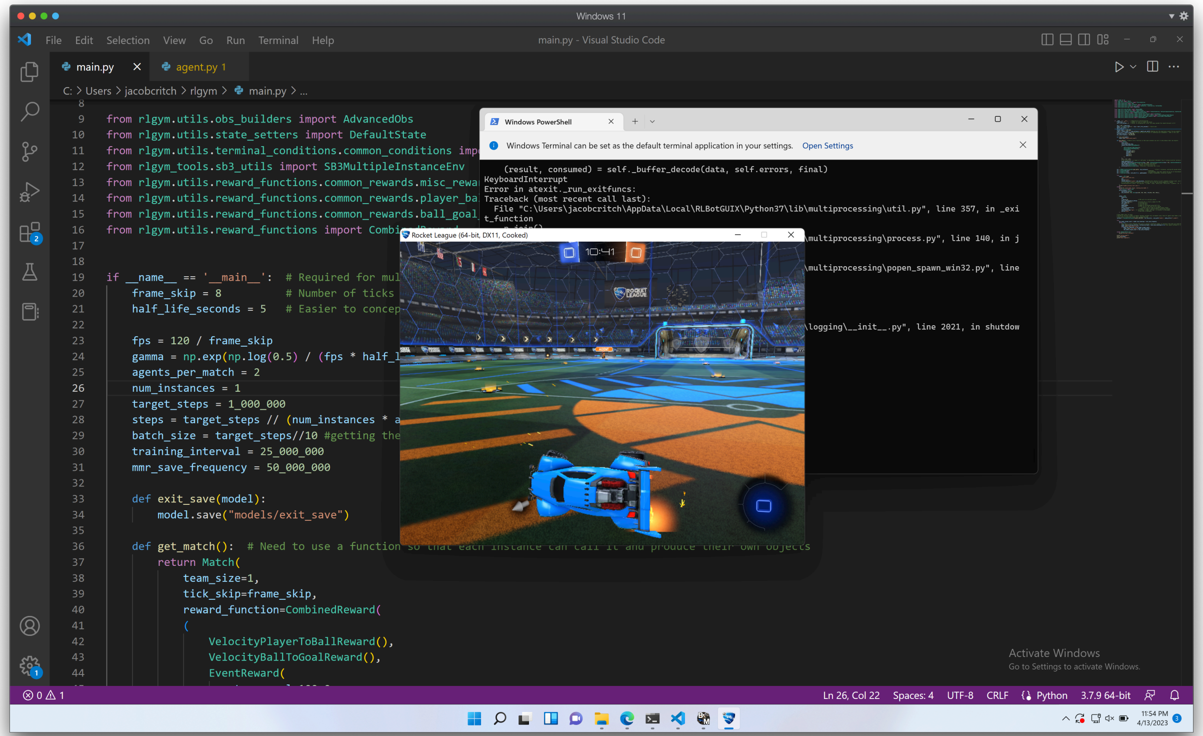
(PPO) [2], and a multilayer perceptron (MLP) policy [3]. The target number of steps to perform before optimizing the network was set to 1,000,000. The training interval was set to 100,000,000. The number of epochs was set to 10.

For the first round of training, the learning rate was set to 5e-5. This Velocity Agent was trained for approximately 10 hours. A Liu Agent was also trained at a 5e-5 rate for approximately 3 hours (training ended early due to a technical error, so data was no collected for this round). After this, two more agents were trained; the first was a Velocity Agent with a learning rate of 0.001, and the second a Liu Agent also with a learning rate of 0.001. These agents were trained for approximately 30 hours each.



```
PS C:\Users\jacobcritch\rlgym> python main.py
Launching Rocket League, make sure bakkesmod is running.
Launched Epic version
Executing injector...
Scanning processes
Found 1 processes
Done
```

Figure 3-1: Executing RLGym Python Script


Windows 11
File
Edit
Selection
View
Go
Run
Terminal
Help
main.py - Visual Studio Code
main.py
agent.py 1
C: > Users > jacobcritch > rlgym > main.py > ...
Windows PowerShell
Windows Terminal can be set as the default terminal application in your settings.
Open Settings
Rocket League (64-bit, DX11, Cooked)
10:41
Activate Windows
Go to Settings to activate Windows.
Ln 26, Col 22
Spaces: 4
UTF-8
CRLF
Python
3.7.9 64-bit
11:54 PM
4/13/2023

```python
from rlgym.utils.obs_builders import AdvancedObs
from rlgym.utils.state_setters import DefaultState
from rlgym.utils.terminal_conditions.common_conditions imp
from rlgym_tools.sb3_utils import SB3MultipleInstanceEnv
from rlgym.utils.reward_functions.common_rewards.misc_rewa
from rlgym.utils.reward_functions.common_rewards.player_ba
from rlgym.utils.reward_functions.common_rewards.ball_goal
from rlgym.utils.reward_functions import Combi


if __name__ == '__main__':  # Required for mul
    frame_skip = 8          # Number of ticks
    half_life_seconds = 5   # Easier to concep

    fps = 120 / frame_skip
    gamma = np.exp(np.log(0.5) / (fps * half_l
    agents_per_match = 2
    num_instances = 1
    target_steps = 1_000_000
    steps = target_steps // (num_instances * a
    batch_size = target_steps//10 #getting the
    training_interval = 25_000_000
    mmr_save_frequency = 50_000_000

    def exit_save(model):
        model.save("models/exit_save")

    def get_match():  # Need to use a function so that each instance can call it and produce their own objects
        return Match(
            team_size=1,
            tick_skip=frame_skip,
            reward_function=CombinedReward(
            (
                VelocityPlayerToBallReward(),
                VelocityBallToGoalReward(),
                EventReward(
```

```
    (result, consumed) = self._buffer_decode(data, self.errors, final)
KeyboardInterrupt
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "C:\Users\jacobcritch\AppData\Local\RLBotGUIX\Python37\lib\multiprocessing\util.py", line 357, in _exi
t_function
\multiprocessing\process.py", line 140, in j
\multiprocessing\popen_spawn_win32.py", line
\logging\__init__.py", line 2021, in shutdow
```

Figure 3-2: Agent Training in Rocket League

3.1.4 Evaluating Agent Performance

Agent performance is recorded by Stable Baselines 3's VecMonitor. By default, mean reward and episode length are logged. These logged metrics are viewable via Tensorboard (Figure 3-3). Additional logs and metrics are also logged and printed to the console running the Python script at each iteration, such as loss, entropy loss, and policy gradient loss.

Agent performance was also be evaluated by wrapping the models as RLBot agents. In this way, the agents can be loaded into the Rocket League using the RLBot GUI, and the agents can play against real players or compete against other ML or non-ML bots.

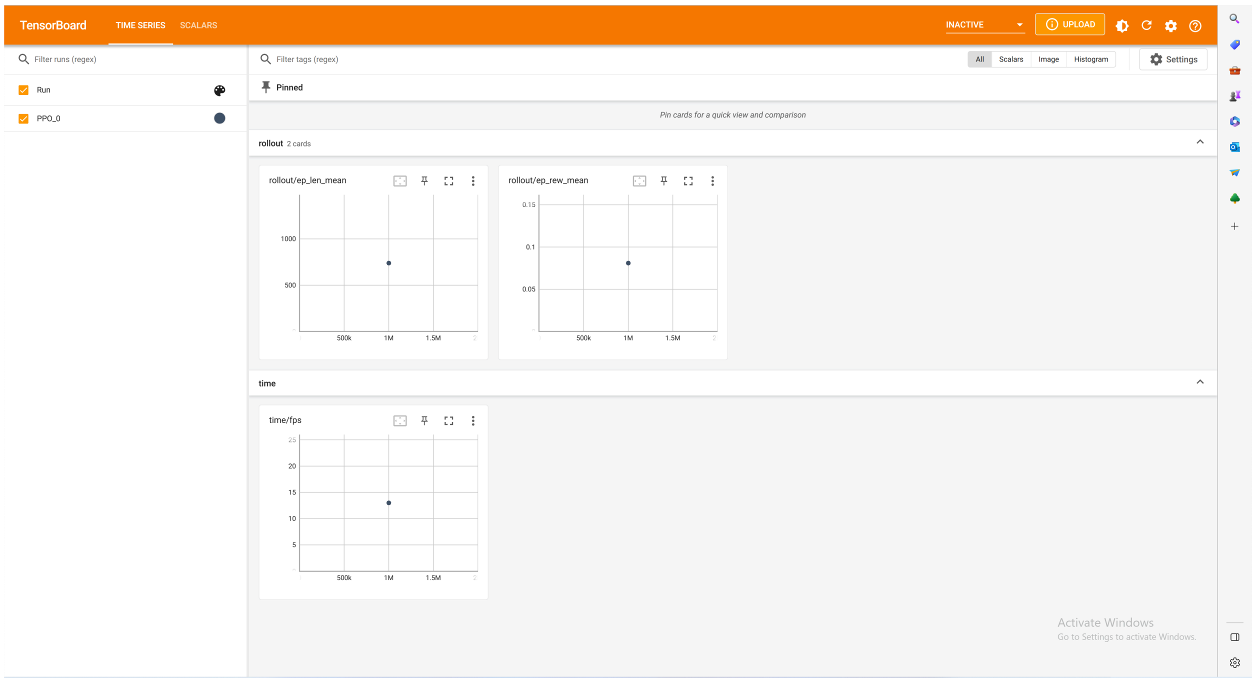


Figure 3-3: Tensorboard View of Agent Performance Metrics

Chapter 4:

Results and Discussion

Since the three agents were trained for a relatively short time (<=30 hours), they were not expected to be able to play Rocket League at a competent level. However, we can observe that they were able to take actions, control their movement, and make progress towards learning. Several learning metrics were logged and graphed using Tensorboard.

4.1 Observations While Training

The Velocity Agents - during the early phases of training - were spinning around, not appearing to take intelligent actions. However, when observed near the end of the training sessions, the 5e-5 agent was inclined to go for a tap on the ball if it was already near it, and the 0.001 agent was able to line up with the ball and strike it towards the net. Interestingly, this agent would commonly do one pass towards the ball, missing it, and then doing a "u-turn" before coming back and impacting the ball in a pendulum-like fashion. In the case of the 5e-5 Velocity Agent, a goal was scored approximately every 15 minutes of observed time (at around 10 hours of training time). When observing the 0.001 agent, a goal was scored approximately every 30 seconds (at 100x normal speed), leading to a score of approximately 1000-1000 at the end of training.

The Liu Agent, when observed, appeared to control the car more sanely (less wild spinning, driving in a specific direction more often, more ball taps early on, more goals faster), with a slightly higher number of goals scored per hour than the Velocity Agent.

4.2 Training Metrics

4.2.1 Velocity Agent (5e-5 learning rate, ~10 hours)

On average, training took approximately 0.8 hours per iteration (1M steps). Since the number of iterations was so small, it is difficult to draw meaningful observations from this model alone. However, the data signifies that the methods for training were operating correctly. It can be noted that as iterations continue, entropy loss moves closer to 0 (initially starting from negative values). Higher entropy loss sometimes correlates with an agent taking more exploratory actions, exploring the state space more thoroughly, and being less deterministic.

Iterations (1M steps)	Mean Reward	Entropy Loss	Loss	Mean Episode Length
1	-0.11	-7.57	0.00285	705
2	-0.14	-7.56	0.00285	670
3	0.10	-7.56	-0.00334	710
4	-0.16	-7.56	0.0151	699
5	-0.27	-7.56	0.0251	743
6	-0.17	-7.55	0.0365	755
7	0.32	-7.55	0.029	691
8	0.23	-7.55	0.029	717
9	-0.10	-7.55	0.0612	769

Iterations (1M steps)	Mean Reward	Entropy Loss	Loss	Mean Episode Length
10	-0.15	-7.55	0.0376	685
11	-0.07	-7.54	0.0421	726
12	-0.17	-7.54	0.0371	717

Table 4-1: Sample of Training Metrics for 5e-5 Velocity Agent

4.2.2 Velocity Agent (0.001 learning rate, 30 hours)

After training the Velocity Agent at a 5e-5 learning rate for 10 hours and analyzing the results, it was decided to experiment by training a new Velocity Agent at a higher learning rate. A new Velocity Agent was defined with a new learning rate of 0.001, and trained for approximately 30 hours. Additionally, more metrics were enabled to be logged for Tensorboard.

4.2.3 Liu Agent (0.001 learning rate, 30 hours)

A final agent was defined using the Liu reward functions (as mentioned in the Background section of this paper) with a learning rate of 0.001, and trained for approximately 30 hours.

4.2.4 Comparison Graphs

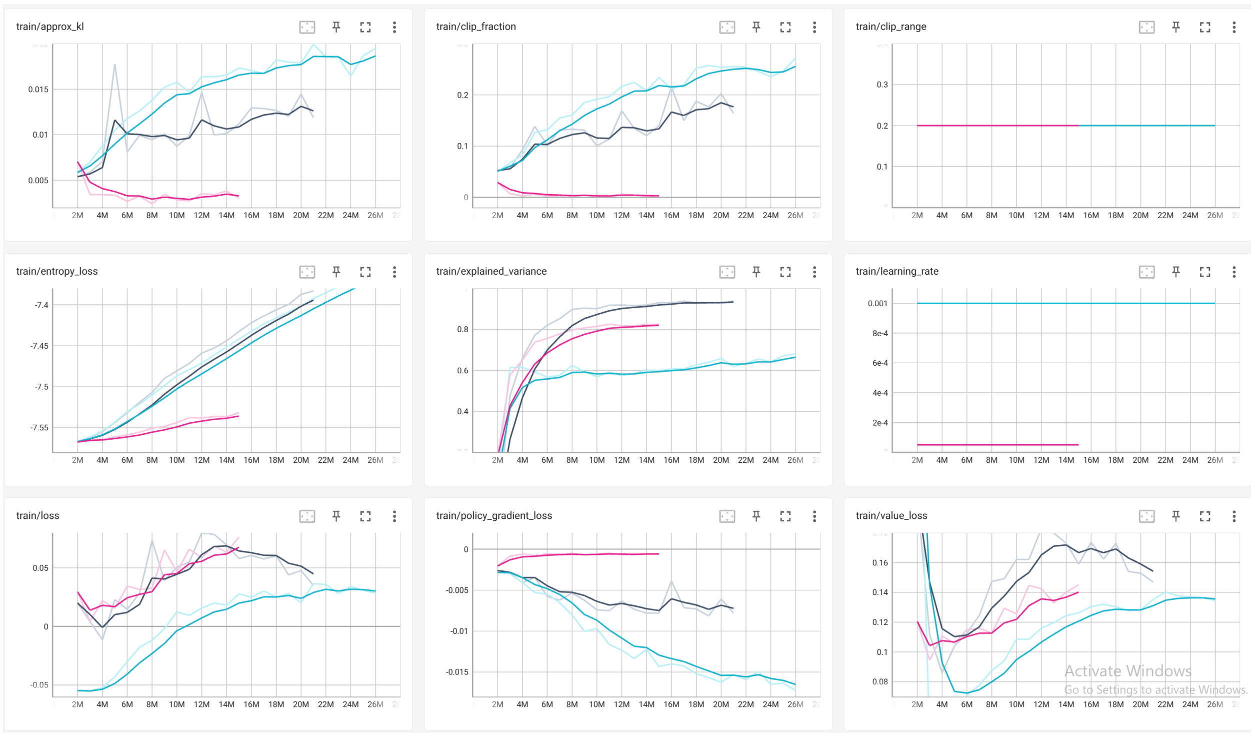


Figure 4-1: Training Metrics Overview (all agents)

[Pink = '5e-5 Velocity Agent', Black = '0.001 Velocity Agent', Blue = '0.001 Liu Agent']

Figure 4-1 charts the training results for all three agents. From top left to bottom right, the metrics graphed are: Approximate Kullback-Leibler Divergence [4], Clip Fraction, Clip Range, Entropy Loss, Explained Variance, Learning Rate, Loss, Policy Gradient Loss, and Value Loss.

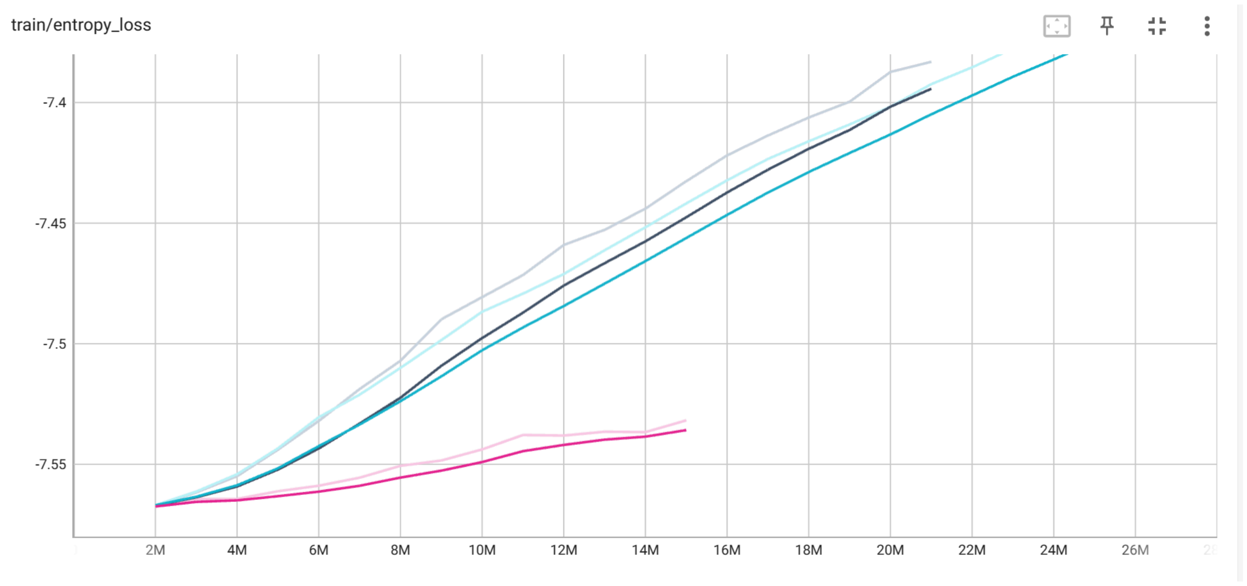


Figure 4-2: Entropy Loss (all agents)

[Pink = '5e-5 Velocity Agent', Black = '0.001 Velocity Agent', Blue = '0.001 Liu Agent']

Figure 4-2 charts the entropy loss for all three agents. The 5e-5 Velocity Agent was shown to have a markedly lower entropy loss than the other two 0.001 learning rate agents, suggesting that the 5e-5 agent was being less exploratory in its actions.

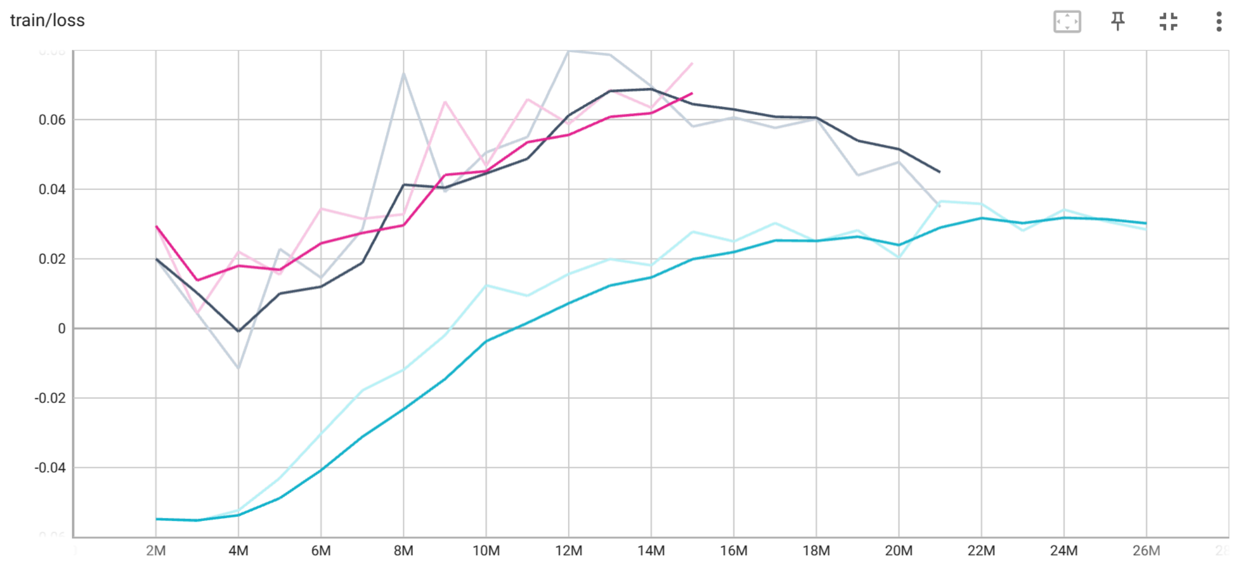


Figure 4-3: Loss (all agents)

[Pink = '5e-5 Velocity Agent', Black = '0.001 Velocity Agent', Blue = '0.001 Liu Agent']

Figure 4-3 charts the loss for all three agents. The 0.001 Liu Agent had the lowest loss, while the two Velocity Agents had similar loss.

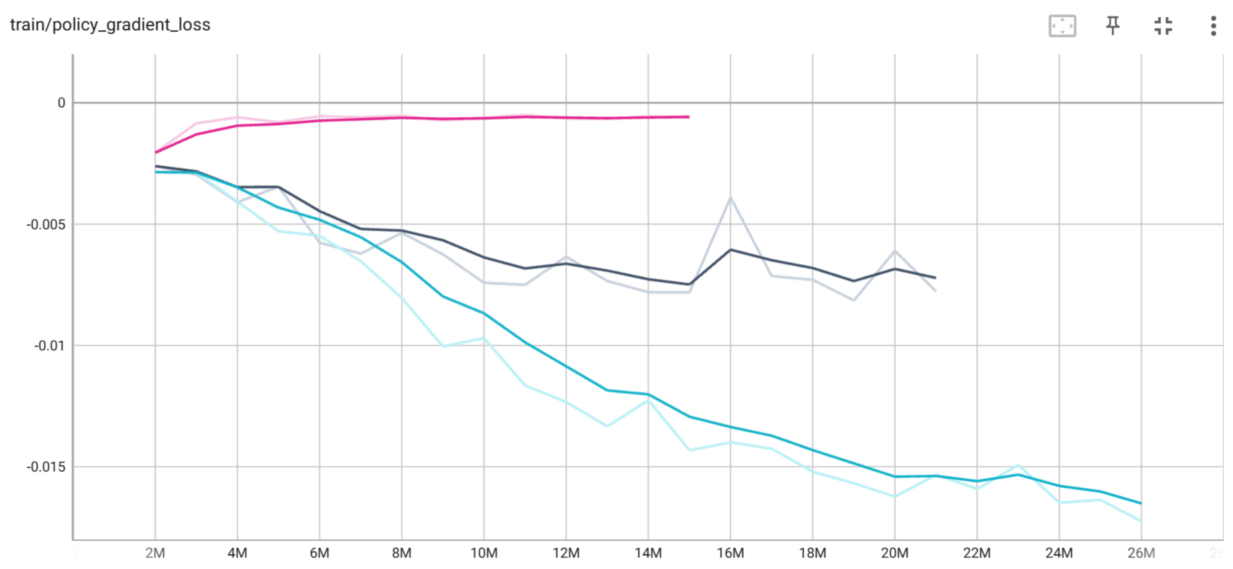


Figure 4-4: Policy Gradient Loss (all agents)

[Pink = ‘5e-5 Velocity Agent’, Black = ‘0.001 Velocity Agent’, Blue = ‘0.001 Liu Agent’]

Figure 4-4 charts the policy gradient loss for all three agents. The 0.001 Liu Agent had the lowest loss, while the 5e-5 Velocity Agent had the highest.

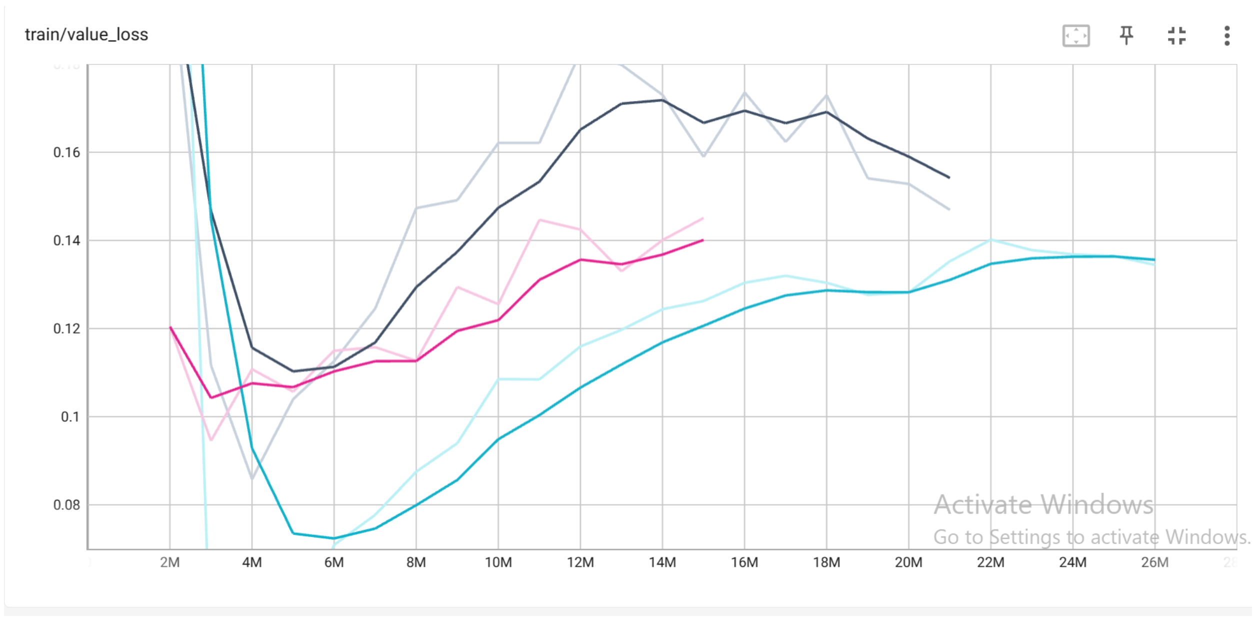


Figure 4-5: Value Loss (all agents)

[Pink = ‘5e-5 Velocity Agent’, Black = ‘0.001 Velocity Agent’, Blue = ‘0.001 Liu Agent’]

Figure 4-5 charts the value loss for all three agents. The 0.001 Liu Agent had the lowest loss, while the 0.001 Velocity Agent had the highest.

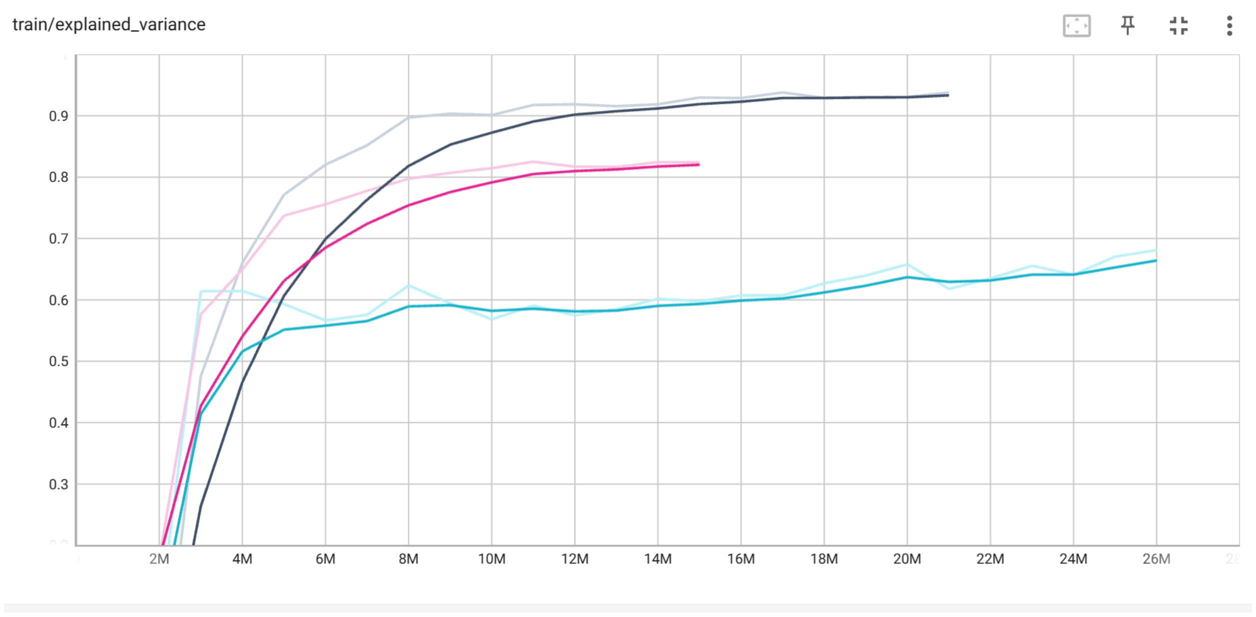


Figure 4-6: Explained Variance (all agents)

[Pink = ‘5e-5 Velocity Agent’, Black = ‘0.001 Velocity Agent’, Blue = ‘0.001 Liu Agent’]

Figure 4-6 charts the explained variance for all three agents. The 0.001 Liu Agent had the lowest explained variance, while the 0.001 Velocity Agent had the highest variance.

4.2.5 Analysis

In summary, the 0.001 Liu Agent showed the most promising results as a candidate for future training and tuning. This agent demonstrated the lowest loss, and a fast rate of entropy loss growth. These are important factors to keep in mind when training an agent on such a complex space.

4.3 Play Test (RLBot)

The 0.001 Velocity Agent and 0.001 Liu Agent were wrapped as RLBots and imported into RLBotGUI. The game was set to 5x speed, and then 10x speed using Bakkesmod. We intended to set the game speed to 100x similar to the training script, but the Bakkesmod console did not allow this (max of 10x). Several rounds were started and observed for some time (~30 in-game minutes each). One of these runs was recorded (screen captured) for approximately 6 minutes, and uploaded to YouTube (linked here: https://youtu.be/A70NOuNnFzw). On the first run, 2 goals were quickly scored, but for the rest of this run and for the subsequent runs, no goals were scored, although the agents did hit the ball several times. The agents frequently performed flips, side rolls, boosts, and other movements. Interestingly, both agents would frequently get 'stuck' in a spot, moving only small amounts for extended periods of time (even up to an hour of in-game time). I theorize that this could be a case of the model reaching a local minima/maxima.

Chapter 5:

Conclusion

In conclusion, we successfully set up an environment for training reinforcement learning models in Rocket League, visualizing and comparing their training metrics, and testing them in actual game scenarios. The resulting training metrics showed that these agents were indeed learning, and when comparing the Velocity Agent(s) and Liu Agent, the data showed marked differences in their approaches. The play test was revealing in that it demonstrated the skill level of the agents tested in an actual match setting. Even after 30 hours of training, the bots still have a long way to go before they could be considered “competent” in Rocket League. Still, the agents were able to hit the ball, and sometimes score, which is impressive given that Rocket League is such a complex space. The study’s findings show that the 0.001 Liu Agent would be a promising candidate for future training and tuning due to its low loss and exploratory nature. Additionally, a Liu-like reward function has already been shown to be successful when used in a study on reinforcement learned agents in simulated humanoid soccer [5], in an environment not so different than Rocket League.

5.1 Future Work

These agents will continue to be trained for a much longer time - on the order of months rather than days. In future training, a more powerful computer system will be used as a host (potentially cloud based), in which many instances of training could be run concurrently (for example, 50 clients all training at 100x speed). This would greatly improve results, and allow for many models with differing hyperparameters to be trained simultaneously. Moreover, there are new open-source ‘Rocket League environment simulators’ being developed that could be used to drastically improve compute performance.

Bibliography

[1] Impossibum, *rlgym_quickstart_tutorial_bot,* (2022), GitHub repository, https://github.com/Impossibum/rlgym_quickstart_tutorial_bot

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, August 28). Proximal policy optimization algorithms. arXiv.org. Retrieved April 14, 2023, from https://arxiv.org/abs/1707.06347

[3] Popescu, Marius-Constantin & Balas, Valentina & Perescu-Popescu, Liliana & Mastorakis, Nikos. (2009). Multilayer perceptron and neural networks. WSEAS Transactions on Circuits and Systems. 8.

[4] S. Kullback, R. A. Leibler. "On Information and Sufficiency." The Annals of Mathematical Statistics, 22(1) 79-86 March, 1951.

https://doi.org/10.1214/aoms/1177729694

[5] Liu, S., Lever, G., Wang, Z., Merel, J., Eslami, S. M. A., Hennes, D., Czarnecki, W. M., Tassa, Y., Omidshafiei, S., Abdolmaleki, A., Siegel, N. Y., Hasenclever, L., Marris, L., Tunyasuvunakool, S., Song, H. F., Wulfmeier, M., Muller, P., Haarnoja, T., Tracey, B. D., … Heess, N. (2021, May 25). From motor control to team play in simulated humanoid football. arXiv.org. Retrieved April 21, 2023, from https://arxiv.org/abs/2105.12196