Comp767-Reinforcement Learning: Final Project

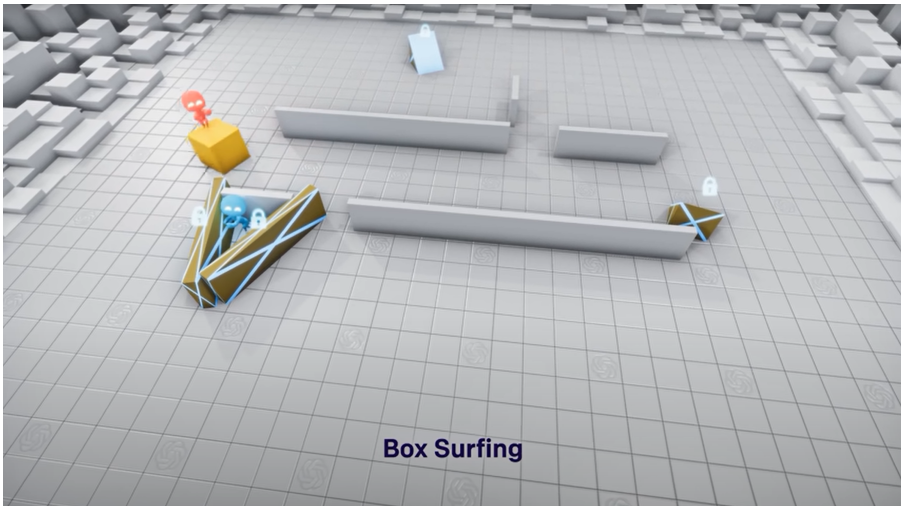
RL-RL: Reinforcement Learning Rocket League

Daniel Bairamian - McGillID : 260669560
daniel.bairamian@mail.mcgill.ca

1 Introduction

Video games are now the most popular entertainment available, and developers are always pushing to create new experiences. With this, the environments created are getting more complex and innovative, to the benefit of consumers. However, with increasingly intricate environments, it has become quite common to see bugs appear in even triple A game releases. My project idea is to use reinforcement learning to discover game mechanics that were not initially intended by the developer.

I initially got the idea from an OpenAI video about multi-agent RL [1], where two RL agents were playing hide and seek and constantly changing their strategies. At some point, one of the agents started to use a mechanic described as "Box Surfing", where he would get on top of a box and as he would start walking, the box would move with him.



This mechanic was obviously not intended by whoever made this test environment.

The seeker, here the red agent, was able to defeat the hider's strategy, the blue agent, of barricading himself, by box surfing then jumping inside his shelter thanks to the ünfair"gain of height. Since this is a custom made RL test environment, the consequences of this bug abuse isn't that important. But what if this hide and seek game was a competitive game? Such unintended game mechanics would either be abused by players, or would need to be patched by the developers immediately.

The question of this project is then, can we take a popular competitive game, and using RL, discover unintended game mechanics?

2 Work done & Results

For this project, I decided to use the game Rocket League [2]. Rocket League is an incredibly popular competitive online consisting of cars playing soccer game, with a huge esport scene. Using a framework called RLBot [3], we can read game packets and set commands to the game via external code. This may seem against the terms of service of the game, but Psyonix, the developers of Rocket League, support this framework so long as it is not used in competitive mode to gain an unfair advantage. The goal of this project isn't to train a bot to play the game conventionally, ideas like this have been done before from the likes of the OpenAI Five project [4]. The goal here is to pick a specific task in the game, which presumably has an optimal strategy, and see if our agent can come up with a better one, including potentially finding unintended mechanics.

Defining the task

The task I have chosen to solve is referred to as "Kick Off". At the beginning of every game, or after a goal is scored, both teams are equally distanced from the ball, which is placed at the center. When the timer starts, the two closest players on each team rush to get to the ball first.

State space

A total of 16 values are needed to represent this state space. Location, Velocity, Angular Velocity, and Rotation are 3d vectors (12), has jumped, has double jumped are booleans (14), boost amount and distance to ball are floats (16).

Action space

Steer	[-1.0, 1.0]
Throttle	[-1.0, 1.0]
Roll	[-1.0, 1.0]
Pitch	[-1.0, 1.0]
Yaw	[-1.0, 1.0]
Jump	True/False
Boost	True/False
Handbrake	True/False
Use Item	False (Never used)

The RLBot framework contains a ControllerState class, which emulates a game controller. There are 8 different actions to give at a given time to represent a controller for our task, 5 of them are continuous floats between -1 and 1, while 3 of them are booleans.

Learning Algorithm

Both states and actions have a mixed of continuous and discrete values. In order to make the learning possible, I transformed every value into a continuous value. For the discrete ones, I simply clamp to the nearest value.

The learning algorithm used for this project was Soft Actor Critic. I used the OpenAI SpinningUp library for their implementation of soft actor critic [5].

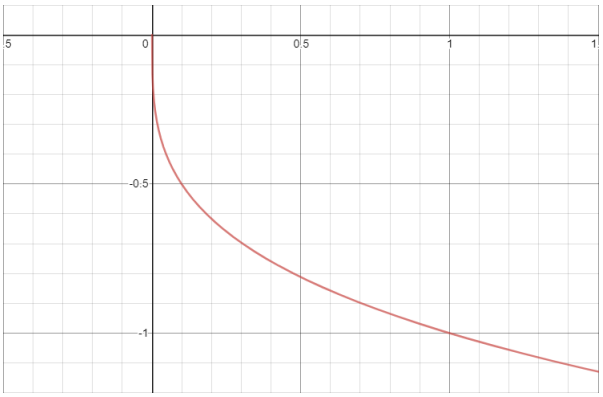
Since the library was made for gym environments specifically, I had to re-implement it to make it work with Rocket League.

Reward Shaping

In order to incenvitise the bot to go towards the ball as fast as possible, my reward function is defined as the following:

$$reward = (-\frac{d}{d_0})^{\alpha}$$

where d is the current distance to the ball, d_0 is the original distance to the ball when the episode started, and α is a hyper-parameter to weigh the importance of proximity to the ball



The following graph is the reward function I used, with $\alpha = 0.3$. The idea is the to give more importance to closer distances. The agent always gets negative reward at every step, scaled by the normalized distance to the ball. If the agent hits the ball, he gets a flat positive reward. If the agent does not hit the ball for some amount of time, the episode is over.

Action Masking

This section is arguably the most important one. In Rocket League, there are many player inputs that would make no sense, without environment feedback. In the game, the car can be in two main states, either on the floor, or in the air. In order to go from the floor to the air, the player needs to perform a jump. However, depending on which state, some actions can be inputted with absolutely no impact, and no environment feedback.

When the agent is in the air, **steering** and **throttling** have no effect. The commands tell the wheel to spin and turn in a certain way, however since there is no contact to the ground, nothing would happen.

When the agent is on the ground, **roll**, **pitch**, or **yaw** commands have no effects. The agent is only allow to spin his car when he is in the air.

Finally, jumping is also special. When the agent jumps, the agent is allowed to jump again for a small window, (referred to as double jumping). After that, until the agent reaches the floor again, the agent is not allowed to jump again.

Why is this important? Even though these actions are useless, a player can still input them. I noticed that after exploring, the agent would wrongfully learn that some actions are good (when in fact they are terrible). For example, when the agent is in the air with a forward velocity towards the ball, he would be steering in some direction. The $(S, A, R, S', Done)$ tuple would then contain a relatively good reward, with S having the steer value active. The agent would learn that steering in that direction must be good then, when it's not since it

has no effect.

To deal with this issue, after a controller state is predicted, I mask the illegal actions before adding it in the experience replay buffer (Zero the values). In the long run, with enough data, this mistake would erase itself eventually, however I observed that this work significantly improved the training.

Environment Work

The RLBot framework was not designed easily do reinforcement learning. The main access point where you can write your code is a function where you can read the game state and output a controller state. This means if you want to read the state after you submit an input, it would not be possible without reworking the library. For this, I had to expose the framework and change the main logic in order to conform to RL environments. Which means, being able to distinctively separate the $(S, A, R, S', Done)$ tuples. The modified library folder is included in the github repository.

3 Discussion

Prior to implementing action masking, the bot seemed to never learn anything. However, after implementing the latter, the bot surprisingly started to complete the task at hand. After training for about a day, the bot started to learn, however with a lot of instability.

Another interesting observation that when reset episodes, sometimes game packets and game states lag behind for a couple frames. Meaning when the episode is reset, even though the car is in the starting position, some information in the game packets are not correctly reset. The fix for this was to skip the first few frames of each episodes. This made the data really stable and logical.

The bot is given 10 000 steps of random action sampling, which is about 30 minutes of in-game time. After that, the action selection is switched to be from the actor network.

Results

Despite the really high instability, the bot seems to converge towards the correct solution, which is rushing towards the ball. Most runs after a couple hours of training are still almost random, however some episodes show really impressive results. The frequency of these successful scenarios increase with time.The amount of training needed to achieve a kickoff strategy which is on par with current professional players seem to be too high for this current project.The improvement shows promising signs that this idea can be used to eventually

discover optimal strategies.

Videos of the bot are included in the github repository.

Improvements

Due to the time constraint and nature of this project, since most time has been invested into creating the environment, I didn't have time to tune hyperparameters, which include network size, learning rate, buffer size, update frequency, and reward function exponent.

Another improvement would be the experience replay buffer sampling. The OpenAI implementation of the replay buffer's sampling function only include random sampling of batches. I was originally planning to implement prioritized experience replay sampling [6], but due to time constraints this was not possible.

I also didn't have time to implement a functioning save feature to save training of the agent, which should be the top priority of features to add.

4 Links

4.1 Code

`https://github.com/danielbairamian/RL-RL`

The repository contains more information about how to make consistent episodes with the given environment, installation guide, and result videos as well

4.2 Youtube

`https://www.youtube.com/watch?v=Qd_ZJu7q7vU`

5 References Links

[1] Multi-Agent RL, OpenAI: `https://www.youtube.com/watch?v=kopoLzvh5jY`
[2] Rocket League: `https://www.rocketleague.com/`
[3] RLBot: `https://www.rlbot.org/`
[4] OpenAI Five: `https://openai.com/blog/openai-five/`
[5] OpenAI SpinningUp: `https://spinningup.openai.com/en/latest/`
[6] Schaul, Tom et al. (2015). "Prioritized experience replay". In: arXiv preprint arXiv:1511.05952