RL^2 Bot

Bryant McArthur: 234009361
Anthony Pasala: 132004067

Github: https://github.com/wovw/rlbot
YouTube: https://www.youtube.com/watch?v=3LscrUrMYE0

December 2023

1 Introduction

In this paper we explore deep reinforcement learning methods for training a bot to play the game Rocket League. We used RLGym [luc21] to build an autonomous bot for Rocket League to perform at a beginning level. The bot will be a player/car on an enclosed soccer field, with the ultimate objective to win the game by scoring the most goals. RLGym API serves as an OpenAI Gym [Bro+16] wrapper to define a game environment. In Python, we imported the RLGym package and used the tutorials and documentation to help us get started. We trained the bot through on-policy learning using a Proximal Policy Optimization Algorithm (PPO) [Sch+17], having it play against itself in numerous sped-up simulated environments.

This problem is sequential because the bot must make a series of decisions while playing the game, affecting the state and possible future actions with each decision. For example, if a bot has possession of the ball, then it can choose to attempt scoring, pass it to a teammate, or dribble past the opponent. Based on the chosen reward and transition functions, the policy will need to adapt to exploit actions with high rewards while also having the ability to explore different actions, all of which is possible because of the sequential nature of the problem.

2 Motivation

The motivation for this project was of course our love for the game Rocket League. But more than that, Rocket League provides us with a complex environment in terms of the observation and action spaces that make training a bot to play the game well a very difficult task. The game is played in a 3-dimensional stadium with a bouncing ball and the ability of the player to jump and soar through the air. It is a game that some individuals have nearly mastered with incredible passing strategies and trick shots. Although the single objective is to score more goals than your opponent, the observation space provides opportunity for much more creative reward functions.

Furthermore, there is a nice plugin to the game that simplifies reading the environment, taking actions and training a pytorch model. With the Bakkesmod plugin installed and the rlgym package downloaded we would have a great starting point to begin exploring different algorithms, methods, and reward functions to build our own bot.

Because of the complexity of the game, the current State of the Art bots still have their flaws and there is room for improvement. With all this in consideration we thought it would be a great project to put our new reinforcement learning knowledge and skills to the test.

3 Related Works

Several papers have already been published exploring the Rocket League problem.

In 2022 Daniel Pascual shared a similar goal to us to “create a bot capable of competing and winning against the current [hard-coded] Rocket League bots”, but quickly had to readjust and limit their objective to “being able to lay the foundation for a Rocket League reinforcement learning bot [Sar22].

In order to do this, they limited their action space to throttle steer and boost. By excluding jump they removed aerial play and stuck to a more two-dimensional game. They also limited their reward function to be simply ”angle_to_ball” and “distance_to_ball” and later added relative velocity.

Finally Pascual was able to experiment with reinforcement learning algorithms. They were able to conclude that the ‘actor-critic’ method that combines the function-approximation and policy-gradient methods known as Proximal Policy Optimization (PPO) outperformed all the others [Sar22]. Thanks to this foundation we were able to start our training.

Seer was another bot built by Neville Walo in 2022 that performed better than Pascual’s. Seer also used PPO with an Adam optimizer. Seer uses a much more complex reward function than the previous paper, with 16 different variables as weights to the reward with the highest being of course “Goal_Scored”. Some of the rewards are conditional like if a goal is scored, while others are continuous and received after every action like “Distance_to_Ball”. In this paper they introduce “Self Play” where the bot they are training plays against itself in order to optimize training [Wal22]. From Seer we were able to see how a more complex reward function would look with several variable weights, and we used the idea of ”Self Play” to train the bot against itself.

Necto and Nexto were until recently the State-of-the-Art Rocket League bots. They were able to achieve Diamond and Grand Champion status that is the top 5% and 1% of all players respectively. These bots also used a custom PPO algorithm and complex reward function found in a pip installable python package they developed called rocket-learn [Rol21b]. The developers used replay files to expose the bots to years of gameplay before they even entered the field. This experience replay ”jumpstarted” training and was followed up with live reinforcement learning on large distributed learning systems [Rol21a]. This paper has helped us realize we don’t have the means or resources to train a top performing bot. We don’t have access to years of replay files nor the time or money for large distributed GPU system training that Necto required.

Lucy-SKG is the current State-of-the-Art model that outperformed Necto and Nexto this year. They propose and use Kinesthetic Reward Combination (KRC), which is an alternative to linear reward combinations and useful in measuring utility of complex phenomena [Mos+23].

4 Methods and Procedure

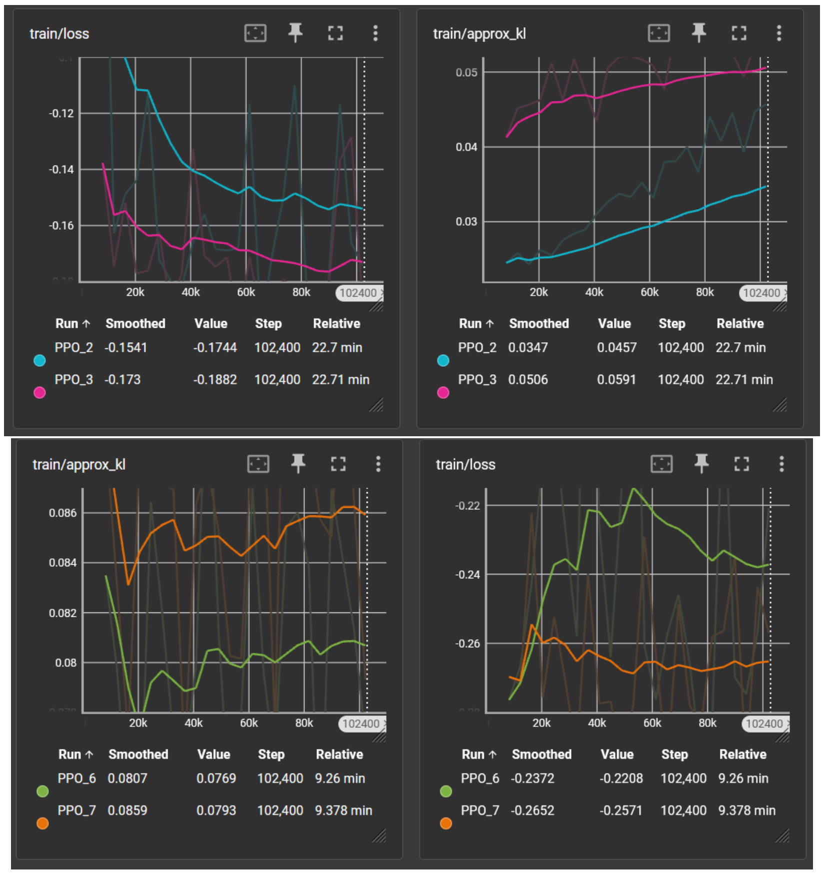
To get the simplest possible first result, we first defined an environment with two opposing independent agents. The reward function is based on the agent hitting the ball towards its desired goal and decreasing the distance between the ball and the goal. We used the provided PPO algorithm from SB3 since it requires little hyperparameter tuning and performs well in a wide range of tasks. We defined the baseline as the agent winning against a player at the lowest competitive rank in the game, which is Bronze 1. After we achieved a working prototype, we iterated on existing research and implementations for the final version of our agent.

4.1 State and Action Space

In RLGym, the ‘StateWrapper’ object contains definitions representing every physics object in a game environment, including attributes for information and acting forces for each object. This `link` has all the possible game values available to access and manipulate in an environment through that object. Some state values that will be commonly used: [car position, car velocity, ball position, ball velocity, other cars positions, other cars velocities, boost]

The following actions are available at every step for an agent: [throttle, steer, yaw, pitch, roll, jump, boost, handbrake]. Each action corresponds to one control input.

While there is a discussion to use all discrete actions or continuous actions when possible, we opted to use continuous actions from the results shown in Figure 1. We trained two agents using discrete vs. continuous action spaces for about 400,000 time steps each, and the results show that the agent using the the continuous space performed better in terms of loss and KL-divergence.



train/loss

Run ↑	Smoothed	Value	Step	Relative
● PPO_2	-0.1541	-0.1744	102,400	22.7 min
● PPO_3	-0.173	-0.1882	102,400	22.71 min

train/approx_kl

Run ↑	Smoothed	Value	Step	Relative
● PPO_2	0.0347	0.0457	102,400	22.7 min
● PPO_3	0.0506	0.0591	102,400	22.71 min

train/approx_kl

Run ↑	Smoothed	Value	Step	Relative
● PPO_6	0.0807	0.0769	102,400	9.26 min
● PPO_7	0.0859	0.0793	102,400	9.378 min

train/loss

Run ↑	Smoothed	Value	Step	Relative
● PPO_6	-0.2372	-0.2208	102,400	9.26 min
● PPO_7	-0.2652	-0.2571	102,400	9.378 min

Figure 1: Loss and KL-divergence results of agents trained with a discrete action space (top) and a continuous action space (bottom) for about 400,000 time steps. Each graph plots the metrics of the last 200,000 time steps for the agents.

Thus in our resulting action space, the first five values are expected to be in the range [-1, 1], while the last three values should be either 0 or 1.

4.2 Transition Function

Because five of the eight values in the action space are continuous we must use a transition function for continuous values. The current State of the Art functions for continuous distributions are SAC, TD3 and TQC [Ibr+21]. We planned on first experimenting with these 3 algorithms and using RLzoo to automatically tune our initial hyperparameters, however, our research has shown that the optimal learning algorithm to use for the Rocket League problem is PPO [Sar22; Wal22; Mos+23; Sch+17; Rol21a; Ibr+21]. We chose to implement a parallelizable form of PPO from the Stable Baselines3 (SB3) library [Raf+21].

PPO offers a balance between performance and simplicity. The authors of PPO wanted a model that had the same data efficiency and reliable performance as Trust Region Policy Optimization (TRPO), but that was a simpler, first order system that is more robust to noise [Sch+17].

They achieve this by modifying the surrogate objective function of TRPO by clipping the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta\text{old}}(A_t|S_t)}$$

This removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Next, they take the minimum of the clipped and unclipped objective, so the final objective is a lower bound on the unclipped objective. With this scheme, they only ignore the change in probability ratio when it would make the objective improve, and include it when it makes the objective worse. For their experiments the optimal value of epsilon was 0.2. The clipped surrogate objective function limits the size of policy updates, preventing dramatic changes in the policy that might harm performance. This can be particularly useful in continuous action spaces, where large changes in policy could result in a wide range of different actions [Sch+17].

PPO trains with on-policy learning, which allows the agent to explore the action space by sampling actions according to the latest version of its policy. The randomness in action selection encourages exploration, which is especially beneficial for continuous action spaces. The complexity of continuous action space does not add to the complexity of the algorithm. The algorithm is less sensitive to hyperparameters than other algorithms, making it a reliable choice for many tasks [Sch+17].

4.3 Reward Function

RLGym provides many reward functions we can use to start out with and eventually develop into a more complex reward function. We started out with a straightforward goal-based reward with a large positive or negative value for scoring or conceding a goal, and a position-based reward like the distance between the ball and the goal. RLGym provides various goal-based rewards and initially we used LiuDistanceBallToGoalReward(), VelocityBallToGoalReward(), and TouchBallReward(). RLGym provides an ‘AnnealRewards’ object to define a combination of multiple rewards and transitions between when to use them or count them. We experimented with the given reward functions as well as the reward functions used by Necto.

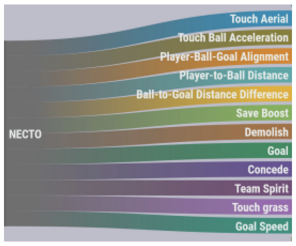


Figure 2: Variables in Necto's Reward function

Our eventual reward function was heavily inspired by Necto's above since it performed the best with some slight modifications to the weights.

We adjusted the goal weight from 10 to 12 to greater incentivize scoring goals. We lowered the weight for spending boost from .8 to .6. We don't want to spend boost as soon as we get it so there is a negative weight associated to it so that boost is only consumed when it will lead to a greater reward than the negative reward from spending it. We just lowered that threshold. With that Necto only punishes consuming weight when the car is in the air, and the higher the car the more costly it is to boost. We slightly adjusted it so that it is still more costly to boost the higher you are but there is still a punishment when the car is on the ground. We changed the function:

$$\text{player\_rewards[i] += self.boost\_lose\_w * boost\_diff * }\left(1 - \frac{\text{car\_height}}{\text{Goal\_Height}}\right)$$

to be instead:

$$\text{player\_rewards[i] += self.boost\_lose\_w * boost\_diff * }\left(1 - \frac{\text{car\_height} + 1}{\text{Goal\_Height} + 1}\right)$$

Note that boost_diff here is negative and the reward is only applied when car_height < Goal_Height.

This way when car_height is 0 then it is still punished for using boost. Again, we don't want to spend boost as soon as we get it even if we're on the ground.

4.4 Terminal Conditions

The terminal condition will check if the current state should be the final state of the episode in order to move on to the next episode. We use three simple terminal conditions to terminate the episode if any one is met:

1. GoalScored – If a goal is scored then we end the episode.
2. NoTouchTimeout – If the agent doesn't touch the ball for 30 seconds we end the episode.
3. Timeout – After 5 minutes we end the episode.

4.5 Observation Builder

The default observation space provided by RLGym has information for the ball and each player as well as the previous action and the state of the boost pads (whether boost is available or not).

For the ball we can observe the position, linear velocity and angular velocity. For each player we can observe the position, linear and angular velocity, boost amount, 'on_ground', 'has_flip', and 'is_demoed'.

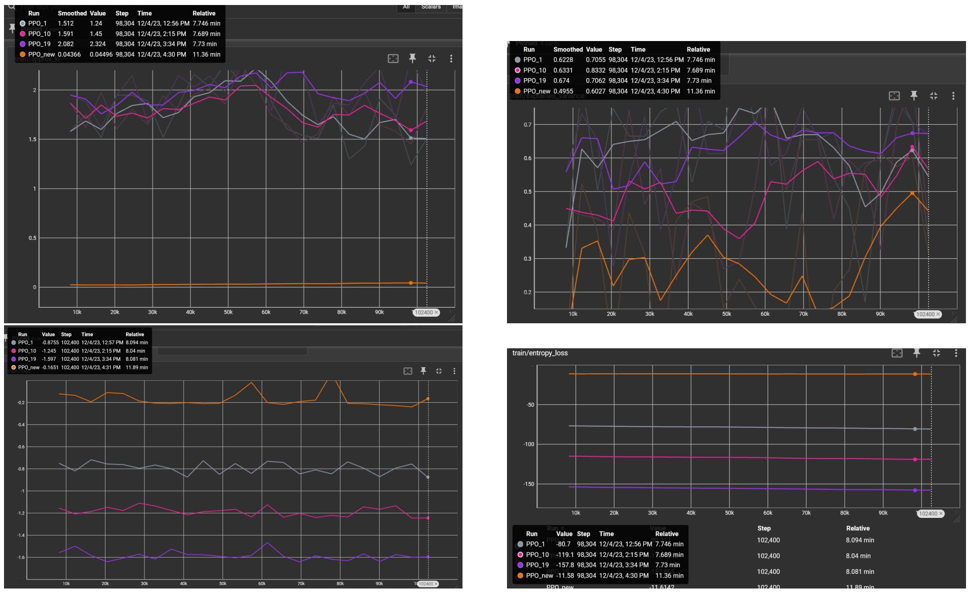
We added as well to the default the relative position and velocity between the player and every other player (which in our case is just one other player since we trained it on 1v1 games.

5 Evaluation and Results

Below are shown the tensorboard plots of our training of our best model trained with a continuous action space on 10 million time steps with each line being a different subportion of training where the orange line was the first 100,000 time steps and the purple line was the last 100,000 time steps.

In the top left plot we can see the large difference between the KL divergence at the start of training and at the end. There is a large jump from .02-.04 in the first 100k steps to converging around 2 in the last 2 million steps. This helps indicate we made significant progress near the beginning of training while progress slowed later on.

In the top right plot we can see the differences of the explained variance at various timesteps. Although, there is not as much of a noticeable margin it is clear that more of the variance is explainable by our trained model at 70%, where it was around 20% at the start of training.



	Run	Smoothed	Value	Step	Time	Relative
●	PPO_1	1.512	1.24	98,304	12/4/23, 12:56 PM	7.746 min
●	PPO_10	1.591	1.45	98,304	12/4/23, 2:15 PM	7.689 min
●	PPO_19	2.082	2.324	98,304	12/4/23, 3:34 PM	7.73 min
●	PPO_new	0.04366	0.04496	98,304	12/4/23, 4:30 PM	11.36 min

	Run	Smoothed	Value	Step	Time	Relative
●	PPO_1	0.6228	0.7055	98,304	12/4/23, 12:56 PM	7.746 min
●	PPO_10	0.6331	0.8332	98,304	12/4/23, 2:15 PM	7.689 min
●	PPO_19	0.674	0.7062	98,304	12/4/23, 3:34 PM	7.73 min
●	PPO_new	0.4955	0.6027	98,304	12/4/23, 4:30 PM	11.36 min

	Run	Value	Step	Time	Relative
●	PPO_1	-0.8755	102,400	12/4/23, 12:57 PM	8.094 min
●	PPO_10	-1.245	102,400	12/4/23, 2:15 PM	8.04 min
●	PPO_19	-1.597	102,400	12/4/23, 3:34 PM	8.081 min
●	PPO_new	-0.1651	102,400	12/4/23, 4:31 PM	11.89 min

	Run	Value	Step	Time	Relative
●	PPO_1	-80.7	98,304	12/4/23, 12:56 PM	7.746 min
●	PPO_10	-119.1	98,304	12/4/23, 2:15 PM	7.689 min
●	PPO_19	-157.8	98,304	12/4/23, 3:34 PM	7.73 min
●	PPO_new	-11.58	98,304	12/4/23, 4:30 PM	11.36 min

Run	Value	Step	Relative
		102,400	8.094 min
		102,400	8.04 min
		102,400	8.081 min
PPO_new	-11.6142	102,400	11.89 min

Figure 3: Results of our bot trained with a continuous action space on 10 million time steps. In all the plots the orange line refers to the first 100k time steps, the light blue line refers to time steps 8 - 8.1 million, the pink line is from 9 - 9.1 million and the purple line is from 9.9-10 million. The top Left graph is the approximate KL divergence, the top right is the explained variance, the bottom left is the loss, and the bottom right is the entropy loss.

In the bottom two plots we see the value loss on the left and the entropy loss on the right. In both plots we can see a steady decrease with each successive portion of training. For the loss we go from -.2 to -.8, to -1.2 and finally to -1.6. For the entropy loss we go from -10 to -80 to -120 to -155. It is clear that the model is still improving up until the end of training and if we had the resources we likely could have continued training for further improvement.

Unfortunately, we had to prematurely stop the training of our model due to GPU and time constraints. At the point of termination we were not yet able to achieve our proposed baseline performance level of beating a Bronze 1 player. Although it was clearly better than random and able to approach and hit the ball in the direction of the goal, our agent still looked like it was taking sporadic actions that a beginning level human player should be able to outperform.

6 Future Work

Our original stretch goals included training the bot for 2v2 and 3v3 gameplay, and implementing an off-policy approach where the agent trains on a dataset of completed human games to optimize a target policy. However, since we didn't make too much progress in achieving our baseline, we planned several improvements and techniques we can use to improve the training performance of the 1v1 model we've been working on.

To expand the knowledge-base of the agent in its interactions with the environment, we could've included other information from the ‘PlayerData‘ object of each agent from the RLGym library in the observation space to let the agent know the exact outcomes of its actions other than just getting a reward. The ‘PlayerData‘ object includes the number of goals, shots, and saves achieved by the agent as a result of an action, or whether it touched the ball or can jump at any given moment.

With more time and computing resources, the agent can be trained longer and faster since the evaluation results show overall performance gain in the training process. One major benefit to the PPO algorithm is the fact that it is parallelizable and the Stable-Baselines3 library also supports multi-instance environments to train the agent in parallel environments, which can significantly speed up training with more computing resources.

We can also implement sim-to-sim transfer. In the context of Rocket League, the individual behaviors of goalies and strikers can be learned by the agent using a simulated environment using Unity, attempting to mimic the mechanics of Rocket League, and then transferred the agent the original game. Despite the inaccuracies of the implemented training simulation, the resulting agents show a high level of performance. “The goalkeeping agent, for instance, saves nearly 100% of its faced shots once transferred to the actual game, while the striking agent scores in approximately 75% of cases” [Ple+22].

Lastly, we can test the Kinesthetic Reward Combination used by Lucy-SKG that proved to outperform Necto [Mos+23]. A KRC would help in measuring the complex phenomena introduced by the Rocket League environment.

7 Conclusion

In conclusion, we were able to build the framework for a deep reinforcement model to train a bot to play Rocket League. We were not able to reach adequate performance, but we were able to make significant improvements from a completely random policy that had potential to continue to progress if we had more resources and training time. We are pleased with the foundation we laid in setting up a transition function, reward function and observation builder in order to train a bot against itself and we recognize that there is significant work to be done to develop a bot that can compete at a high level.

References

[Bro+16] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[Ibr+21] Daanesh Ibrahim et al. "Rocket Learn". In: *SMU Data Science Review* 5.2 (2021), p. 12.

[luc21] lucas-emery. "RLGym". [Computer software]. https://github.com/lucas-emery/rocket-league-gym. 2021.

[Mos+23] Vasileios Moschopoulos et al. "Lucy-SKG: Learning to Play Rocket League Efficiently Using Deep Reinforcement Learning". In: *arXiv preprint arXiv:2305.15801* (2023).

[Ple+22] Marco Pleines et al. "On the Verge of Solving Rocket League using Deep Reinforcement Learning and Sim-to-sim Transfer". In: *2022 IEEE Conference on Games (CoG)*. IEEE. 2022, pp. 253–260.

[Raf+21] Antonin Raffin et al. "Stable-baselines3: Reliable reinforcement learning implementations". In: *The Journal of Machine Learning Research* 22.1 (2021), pp. 12348–12355.

[Rol21a] Rolv-Arild. *Necto.* [Computer software]. https://github.com/Rolv-Arild/Necto. 2021.

[Rol21b] Rolv-Arild. "rocket-learn". [Computer software]. https://github.com/Rolv-Arild/rocket-learn. 2021.

[Sar22] Daniel Sarrià Pascual. "Development of a competitive Rocket League bot using reinforcement learning". B.S. thesis. Universitat Politècnica de Catalunya, 2022.

[Sch+17] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[Wal22] Neville Walo. "Seer: Reinforcement Learning in Rocket League". In: (2022).