Reinforcement Learning for Rocket League $(RL)^2$

Anshuman Dash, Phillip Peng, and Frank Yao

Department of Electrical and Computer Engineering, University of California Santa Barbara

1 Abstract

Reinforcement Learning has a variety of algorithms to promote actions towards an optimal behavior. Through this project we will be analyzing and comparing the results of three popular RL algorithms — Advantage Actor Critic (A2C), Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO) — through the environment of the game Rocket League. We will mainly be exploring how PPO — the current state of the art algorithm compares to TRPO and A2C, the algorithms that PPO has built upon.

2 Introduction

Rocket League is online multiplayer game where teams of one to three players compete against each other in a game of soccer with cars. One of the key skills needed for the game is being able to "dribble" the ball. This skill is quite difficult as most of the playerbase is unable to dribble consistently, so we believe that it is a good benchmark on how well the algorithms learn. Additionally, we chose the Rocket League as our environment to test this as it's state space is quite large which presents additional challenges.

Through this project we will explore the rates that A2C, TRPO, and PPO are able to learn this skill. Through this paper we will go over the setup of the environment, short summaries of the algorithms used, and then showing and comparing the results of said algorithms.

3 Environment

In order to model the Rocket League environment in a way that we can interface with it, we leveraged the RLGym library which provides us with an API interact with. RLGym allows us to create custom environments through the use of various configuration objects.

The core functionality for RLGym environments are based in 3 objects: the Obsbuilder, a list of TerminalCondition objects, and a RewardFunction. RLGym uses these objects at every step to determine what (state) observation should be returned to the agent, when an episode should be terminated, and what reward should be assigned to each action. Figure 1 depicts how each of these objects are used by RLGym.

3.1 State and Action Representation

The state in our model is represented by a 70-dimensional tensor which includes all the information the agent may need to learn its objective of dribbling the ball and more. This includes the position, rotation, velocity, and angular velocity of both the agent's car and the ball, if the car can flip, as well as the locations and states of the boost pads (available or empty).

This means that the model has perfect information unlike a human player, who would only have access to what the game's graphics display to them. However, since all of the important properties are in the player's field of view while in game, we felt that this was fair.

Our actions are represents by a 8-dimensional continuous tensor that represents all actions that the agent's car could take. This tensor corresponds to the throttle, steer, yaw, pitch, roll, jump, boost, and handbrake values for the car.

Although the actual control scheme for players is different from this, all of these values can be changed through careful use of the game's controls.

3.2 Episode Specification

To begin each episode, we initialize the starting state with the agent's car at the end of the field (in front of their goal) and the ball at a slightly randomized location above the hood of the agent's car. The ball's location is randomized to ensure that the models does not become overfit to dribble the ball when only starting at a specific location.

We then set the terminal state to be when the ball's location reaches a certain y-value, touching the ground.

3.3 Reward Structure

At first, we initialized the reward function to give a constant reward to the agent at each timestep. This essentially rewarded the agent for keeping the ball above the ground for as long as possible. However, this immediately led the model to a local optimum where it would simply launch the ball as high as it could to avoid ending the episode as long as possible.

In order to avoid this, we changed the final reward function to reward the agent for keeping the ball below a specific y-value. This value was decided by estimating how high the ball usually goes when dribbled by top players. Although this helps the model learn how to dribble faster, it may have prevented it from learning a more unconventional policy than traditional dribbling. However, we felt that this was necessary in order to promote dribbling more effectively and allowing for faster training.

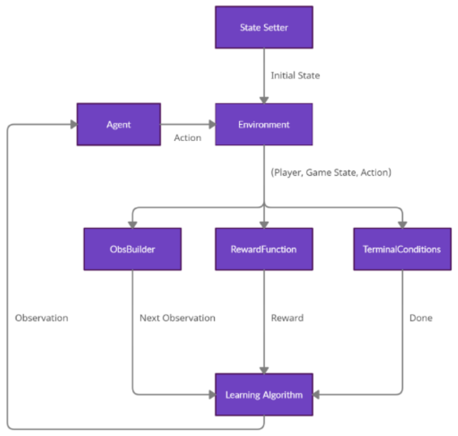


Figure 1: RLGym Configuration Objects

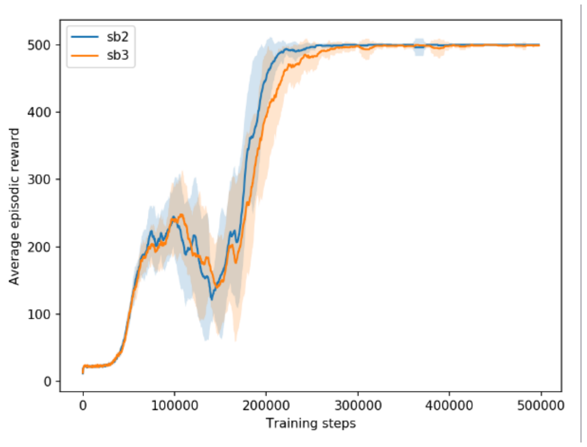


Figure 2: A2C Results for CartPole

4 Algorithms

This section will give a brief overview of the different reinforcement learning algorithms used in this project.

4.1 Advantage Actor Critic (A2C)

Advantage Actor Critic (A2C) is a version of traditional Actor critic Methods with the valuation function being some "Advantage" term. Actor critic methods in general have 2 parts. A Critic who estimates a value function and an Actor who updates a policy distribution in a direction suggested by the Critic.

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))] \quad (1)$$

They will update a policy in the direction of the gradient, in a fashion similar to Equation 1. $\nabla_\theta \log \pi_\theta(a_t|s_t)$ just represents the direction of the steepest increase of log probability of selecting an action at state s_t. The $G_t - b(s_t)$ term is just a valuation of the current policy. Thus, Equation 1 is essentially just saying that we are pushing a policy in the direction of more or less optimal actions. The difference A2C has from different Actor critic methods is that the $G_t - b(s_t)$ term is replaced with an "Advantage" function of $A(s_t, a_t)$ which can be seen in Equation 2.

$$A(s_t, a_t) = Q_w(s_t, a_t) - V(s_t) \quad (2)$$

This advantage function is essentially made of the Q-valuation of the state and action from the valuation of the current state. This essentially states how much better some action is compared to an average action at state s_t. In general this would require two neural networks to value, one for the Q-function and one for the valuation function. However with Bellman's Optimality Equation seen in Equation 3 we can express Q in terms of V. Substituting this Equation 3 into Equation 2 results in Equation 4 which allows us to solve for Advantage with one neural network. We then follow Equation 1 with $G_t - b(s_t)$ replaced with the Advantage function to update our policy.

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})] \quad (3)$$

$$A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \quad (4)$$

A feature we should notice with A2C is that experimentally, there is a period of exploration where it seems that the algorithm is reverse learning. This can be seen in Figure 2 which is the results of A2C on the RL benchmark CartPole. We will later see that our implementation demonstrates similar behavior.

4.2 Trust Region Policy Optimization (TRPO)

Policy Gradient methods are popular in reinforcement learning due to their better convergence properties and effectiveness in continuous action spaces. However, they have their own set of issues. Mainly that determining an effective learning rate is a challenging task. Trust Region Policy Optimization (TRPO) aims to solve this issue through use of trust regions to ensure that policy does not change too much each time it updates.

In order to measure how much the policy changes each time it updates, we compare the new policy θ to the old policy θ_k by calculating the KL-divergence D_{KL} across states visited by the old policy as shown in Equation 5. This is achieved by taking the negative sum of probability of each event in P multiplied by the log of the probability of the event in Q over the probability of the event in P.

The sum is the divergence of policies for a given event. At a high level, the formula for KL divergence calculates much more often the new policy takes a different action compared to the old policy when at the same state. This is allows us to calculate how much our policy has changed when compared to the old policy and enables us to use trust regions.

$$D_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{E}[D_{KL}(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s))] \quad (5)$$

Unlike normal policy gradient methods, TRPO calculates a surrogate advantage between the new policy θ to the old policy θ_k to ensure that updating to the new policy actually improves its performance. This is found through Equation 6. In order to achieve this, TRPO divides the probability of the new policy π_θ taking action a at state s by the probability of old policy π_{θ_k} doing the same action at the same state and multiplying it by the advantage function $A^{\pi_{\theta_k}}(s,a)$ used by other Actor Critic methods. Thus, the surrogate advantage function will return positive values proportional to how much more often the new policy does a more advantageous action when placed at the same state.

$$\mathcal{L}(\theta_k,\theta) = \underset{s\sim\pi_{\theta_k}}{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right] \quad (6)$$

The policy update rule is represented by Equation 7 where the next policy is chosen to be the one with the highest surrogate advantage within the trust region bounded by the divergence constraint δ.

$$\theta_{k+1} = \underset{\theta}{\text{argmax}}\mathcal{L}(\theta_k,\theta),\ \text{s.t.}\bar{D}_{KL}(\theta||\theta_k) \leq \delta \quad (7)$$

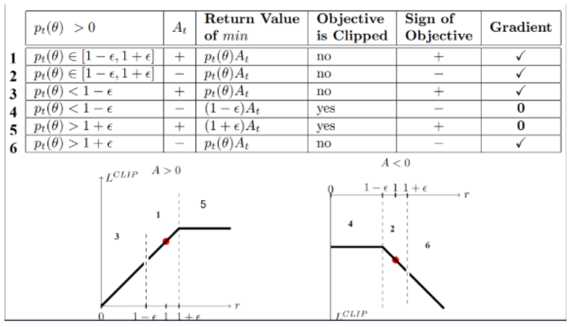
4.3 Proximal Policy Optimization (PPO)

PPO builds upon TRPO and A2C. It computes the policy update through Equation 8 where the expression for L^{CLIP} can be seen in Equation 9. This L^{CLIP} function is much simpler than the KL divergence that is used in TRPO. Thus PPO is computationally faster as it doesn't need to calculate the 2nd order function that's described by the KL divergence. Additionally as we'll soon find that it is simpler to understand conceptually.

$$\theta_{k+1} = \text{arg}\ \underset{\theta}{\max}\mathcal{L}^{CLIP}_{\theta_k}(\theta) \quad (8)$$

$$\begin{aligned} L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[&\min(r_t(\theta)\hat{A}_t, \\ &\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \end{aligned} \quad (9)$$

Essentially all L^{CLIP} is doing is preventing the current policy from straying from its "safe" policy. If the policy approaches an area that is too far from it's current region, expressed by parameter ϵ then the function gets rid of all optimization incentive by clipping it, so that it doesn't stray too far. However, if the policy goes in the worse direction it doesn't clip and it lets it avoid that negative areas. Thus it is fine to explore this area as the policy will update to avoid that area. Thus PPO implements a structure similar to the trust region in TRPO without the need of KL divergence. We can more easily see the values that L^{CLIP} will return through Figure 3. Lastly the $r_t(\theta)$ term is the proportionality term from Equation 6 and $\hat{A}_t$ is the Advantage function from A2C.



	$p_t(\theta)\ > 0$	A_t	**Return Value of** *min*	**Objective is Clipped**	**Sign of Objective**	**Gradient**
1	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	+	$p_t(\theta)A_t$	no	+	✓
2	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	−	$p_t(\theta)A_t$	no	−	✓
3	$p_t(\theta) < 1-\epsilon$	+	$p_t(\theta)A_t$	no	+	✓
4	$p_t(\theta) < 1-\epsilon$	−	$(1-\epsilon)A_t$	yes	−	**0**
5	$p_t(\theta) > 1+\epsilon$	+	$(1+\epsilon)A_t$	yes	+	**0**
6	$p_t(\theta) > 1+\epsilon$	−	$p_t(\theta)A_t$	no	−	✓

Figure 3: PPO Clipping Regions

5 Results

Figure 4 shows the results for the aforementioned algorithms with the reward structure that was highlighted above. The algorithms A2C, TRPO, and PPO are the colors Red, Black and Blue respectively. The thicker lines are the functions when smoothed out while the faded lines are the actual rewards at each evaluation step.

From this we can clearly see PPO outperforms TRPO and A2C as it's able to achieve better results than both functions. It's able to reach a similar if not better outcome than TRPO in less steps while A2C appears to still be in exploratory phase that we highlighted in the Algorithm overview.

We should note that while we see A2C decreasing in average reward there are episodes that we do see the performance of the bot increase at specific timesteps while the average reward is decreasing. We expect the average reward to be decreasing for such a long period of time since the state space is so large that the algorithm takes a much longer time to actually explore.

Additionally, while the bot is not able to perfectly dribble the ball, it is able to outperform the average player, as dribbling itself is not a skill picked up in basic levels of play. We expect with more training time that the bot would learn to perfectly dribble and control the ball

6 Conclusions

We can easily see that the state of the art algorithm PPO does vastly outperform the results of the algorithms that it builds upon, TRPO and A2C. Additionally we are able to find that through relatively small training times that the bot is able to outperform the average human and with more training time we expect it to be able to completely control the ball, essentially perfecting the task.

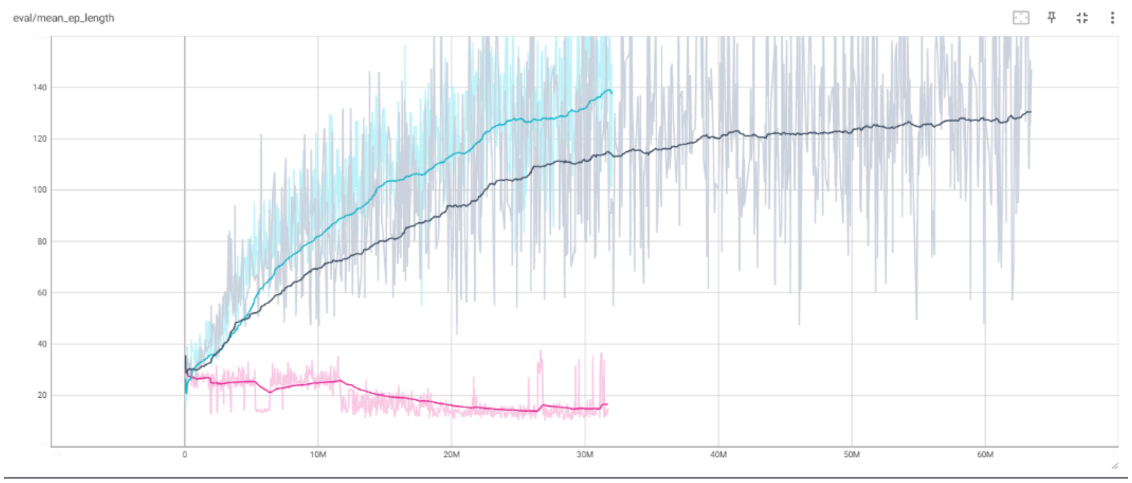


Figure 4: Algorithm Results based on Timesteps

References

[1] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel. *Trust Region Policy Optimization*, `https://arxiv.org/abs/1502.05477`

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*, `https://arxiv.org/abs/1602.01783`

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. *Proximal Policy Optimization Algorithms*, `https://arxiv.org/abs/1707.06347`