USING REINFORCEMENT LEARNING WITH CURRICULUM LEARNING AND REWARD SHAPING TO TRAIN A ROCKET LEAGUE KICKOFF BOT

CS 238 FINAL PROJECT

Arjun Karanam
Department of Computer Science
Stanford University
Stanford, CA
akaranam@stanford.edu

Ronak Malde
Department of Computer Science
Stanford University
Stanford, CA
rmalde@stanford.edu

December 8, 2021

ABSTRACT

This paper seeks to use Reinforcement Learning in order to play a game known as Rocket League, where players play soccer in a virtual environment using cars. Rocket League's large action spaces, large state spaces, and delayed reward (scoring a goal) make it a challenging yet fruitful endeavor. By using the Rocket League Gym environment (similar to OpenAI's Gym Environment), we train a bot to reliably score in the kickoff phase of the game (the beginning phase where the player is randomly placed on the field and has the ability to make a goal). In addition to current approaches in Rocket League literature, we tried negative reward functions as well as curriculum learning. While curriculum learning failed to show improved results, negative rewards allowed us to reach kickoff metrics that are comparable to a human player and current leading hard-coded bots. In the future, we would like to take other phases of the game (such as defending a goal, attacking a goal, etc.) and attempt RL methods with negative rewards to one day train a bot to play the game in its entirety.

1 Introduction

Rocket League is a popular competitive video game where players control cars with jetpacks and try to score a ball into a net on a soccer pitch. The game is driven by a deterministic, 3-dimensional physics engine that emulates real-world physics. The player interfaces with Rocket League using a game controller that has several inputs that alter the car's movement: throttle, steer, yaw, pitch, roll, jump, boost, handbrake. Given the physics-based nature of the game, most game state values are continuous and 3-dimensional, like the positions and velocities of the players and the ball. These factors make the game difficult for human players, and much more so for computer players. Various hard-coded, rules-based bots and AI-driven bots have been created to play the game, all to minimal levels of success.

1.1 Associated Challenges

The challenges that present themselves are mirrored in the domain of Autonomous Vehicles, including having a large action space, and state space. In terms of the action space, you have 8 different options, as stated above: throttle, steer, yaw, pitch, roll, jump, boost, handbrake. Additionally, these actions are not mutually exclusive, meaning that multiple can be taken at the same time. For example, the care can throttle, and jump at the same time. If we assumed that each action was binary, this would give us 2^8, or 256 distinct actions. However, while some of these actions are binary and discrete (such as jumping and triggering the handbrake), others, such as throttle and steer, are not. This explodes the action space even more.

Due to the size and 3D nature of the field, we also have a large state space, with states that could be reached only once across several games. Assuming the car is comparable to a standard car that is 5 meters in length, the field would be a

Figure 1: An image of a car and a ball in Rocket League

whopping 375 meters long and 300 meters wide. Additionally, the car can jump and double jump at every time-step, creating a large 3D arena for the car to navigate within. Given these large state spaces, it is nearly impossible for an agent to gather adequate information on all possible scenarios and optimal actions in the game, and must generalize to develop a working strategy.

2 Related Work

Alphabet's research lab DeepMind has been focusing on training agents to perfect video games using reinforcement learning for the past decade. They were easily able to beat professional players of Go, a board game with close to 3361 possible discrete game spaces. In 2019, DeepMind created AlphaStar [4], a bot that achieved the highest possible rank in the competitive video game StarCraft. Starcraft has a continuous state space in a 2-dimensional world and has an immense action space. DeepMind states that their training lasted 14 days and used many thousands of parallel instances trained on Google Cloud TPU's, amounting to the agent experiencing nearly 200 years of real-time StarCraft play. They trained their agent using an LSTM fed into a reinforcement learning algorithm, which they called "relational reinforcement learning".

These previous successes in video game agents were all navigating a 2-dimensional world, however. A game like Rocket League adds a further complexity of one more dimension. Only in the past few years, human-level performance has been achieved in a few 3-dimensional, continuous-state video games. In 2019, Jaderberg et al. trained an agent to play the first-person shooter game Quake 3 and competed successfully against humans [1]. Relevant to our project, it divided the training of the agent into 3 phases: a) Learning the basics and rules of the game, b) Increasing navigation and coordination skills, and c) perfecting higher-level strategy and memory.

Within the problem of Rocket League, there is a large community that builds hard-coded decision-making bots that can play the game at the level of a Gold or Platinum player (someone who has played the game for roughly 500 hours). But they are easily beaten by professional players, many of whom have upwards of 10,000 hours played in the game. There has been only one large success in the application of Reinforcement Learning to Rocket League [2]. It can beat the bots that come with the game but only plays at a Gold-level player and still under performs compared to the hard-coded bots made by the community. The bot was trained on human replays of the game and then used reinforcement learning to play against itself and improve.

3 Problem Scope

Because the state space of a Rocket League game is so vast and there has been little success in creating a general game-playing agent in Rocket League using reinforcement learning, we decided to narrow our problem scope to perfecting an agent's kickoffs. Similar to real soccer, a kickoff in Rocket League occurs at the start of each game and after a goal is scored in the game. There are five possible locations that the car can randomly start in for a kickoff, each a different distance away from the ball, and a different orientation relative to the ball. The ball, however, always starts at the center of the field [See figure 2]. It is important to note that only one kickoff position, the position where the car is furthest from the ball, has the car directly facing the ball. In all of the other positions, the car is not directly facing the

ball and would miss the ball if it just drove straight forward. For this paper, we wished to create an agent that could consistently score the ball into the opponent's net from any of the kickoff positions.

Additionally, we constrained the problem further to just the agent on the field. In a normal game of Rocket League, it is either a 1v1, a 2v2, or a 3v3. But in order to not add noise to the training with an opponent, we decided to train as a 1v0 game, with just our agent on the field.

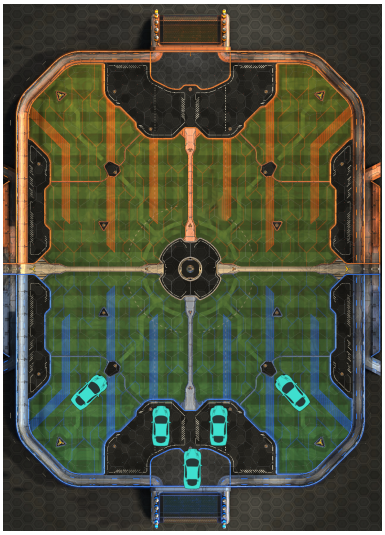

Figure 2: Possible Kickoff Positions for cars on the Blue Team

4 Methods

The Rocket League community developed an API that interfaces with the game to read game data and converts the data into an environment similar to OpenAI's Gym, which supports many common reinforcement learning algorithms. At each game "tick", an in-game timestep, the API allows the programmer to send an action to the game, which consists of values for throttle, steer, yaw, pitch, roll, jump, boost, and handbrake, to step through the environment. The API also allows the game to be sped up by a factor of 100, allowing for faster training. We used this API during our project for training our agent.

4.1 Evaluation Methods

We considered three factors for determining the success of our agent in performing kickoffs. To be clear, these are metrics we used after the fact to evaluate our agent. We used separate values (as we will discuss later) for the agent's own reward function. First, we measured the percentage that the agent scored the ball after kickoff. Second, as a similar metric, we measured the percentage that the agent hit the ball after kickoff. Finally, we also evaluated the average time it took for the agent to hit the ball after kickoff. This is important because to have a better kickoff in Rocket League, a player needs to hit the ball before the opposing team does.

4.2 Baseline

For a baseline agent, we created a policy that drives forward and boosts towards the ball. As noted at the beginning of Section 3, most of the kickoff positions are not correctly lined up with the ball. Thus, the policy also adjusts the steering of the car based on the angle the car is facing and the angle of the car to the ball. Algorithm 1 provides the pseudocode for this approach. This entire policy is similar to a simple kickoff that most beginner Rocket League players use. The code for our baseline bot is given in Algorithm 1.

4.3 Proximal Policy Optimization

We chose to use reinforcement learning to train our agents to score kickoffs. We used a reinforcement learning algorithm called Proximal Policy Optimization (PPO) [3], which was made by OpenAI for Atari game-playing, and is currently a

Algorithm 1 Baseline Agent Policy

$Build\ blank\ action$
$action.throttle$ = 1
$action.boost$ = 1
$agent\_ball\_angle$ = arctan($gamestate.bot\_position - gamestate.ball\_position$)
$angle\_front\_to\_target$ = $agent\_ball\_angle - gamestate.bot\_yaw$
if $angle\_front\_to\_target$ > 10 degrees to the left **then**
    $action.steer$ = right
else
    if $angle\_front\_to\_target$ > 10 degrees to the right **then**
        $action.steer$ = left
    else
        $action.steer$ = straight
    end if
end if
Return $action$

leading algorithm that balances performance and dramatically speeds up training speed. PPO is an on-policy method, which means it constantly updates the policy and explores the environment based on the updated policy.

PPO updates the policy using an objective function. In a normal Policy Gradient Method, this objective function is the loss of the policy gradient, given by

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t\right],$$

where $\hat{A}_t$ is the advantage function, which compares how a certain action compares to the average action from that state. The formula for the advantage function is given by

$$A(s,a) = Q(s,a) - V(s).$$

$V(s)$ is expressed as a neural network for large state spaces.

Vanilla Policy Gradients using the advantage function can have high variance and improvement is often noisy. Another algorithm proposed in 2015 by OpenAI, Trust Policy Region Optimization, creates a trust region for the policies using a Kullback-Leibler divergence (KL Divergence) statistic. The problem is, TRPO is very computationally expensive because it has to optimize with the additional KL constraint.

In 2017, OpenAI released an improved set of algorithms called Proximal Policy Optimization that achieves the same goals as TRPO but is far more efficient. It defines the probability ratio between the old and new policy,

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}.$$

Then the objective function becomes

$$J(\theta) = \mathbb{E}\left[r(\theta)\hat{A}_{\pi_{old}}(s,a)\right].$$

Finally, instead of adding a KL constraint like TRPO did, PPO adds a clipping function, that serves the same purpose but it much faster to optimize. The final objective function for PPO is given with a clipping function that truncates the probability ratio to be within ϵ of 1. ϵ is a hyperparameter, in the original paper and in our implementation it was 0.2.

$$J^{PPO}(\theta) = \mathbb{E}\left[min\big(r(\theta)\hat{A}_{\pi_{old}}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\pi_{old}}(s,a)\big)\right]$$

We used the PPO algorithm for all of the reinforcement learning training used in this paper.

4.3.1 Reward Shaping

To train our agent, we had to determine the best reward function to use in our reinforcement learning algorithm. Because the state space and action space for scoring from a kickoff is still so large, it would take much too long to train an agent

Strategy	Reward	Reward function
Maximize positive	Distance of player to ball	.001 * exp(-distPlayerToBall) + 1000*goalScored
Maximize positive	Distance of player to ball + Distance of ball to goal	.00001 * exp(-1 * distPlayerToBall) + .0001 * exp(-1 * distBallToGoal) + 1000 * goalScored
Minimize negative	Distance of player to ball	-.0001 * distPlayerToBall + 1000 * goalScored
Minimize negative	Distance of player to ball + Distance of ball to goal	-.00001 * distPlayerToBall + -.0001 * distBallToGoal + 1000 * goalScored

Table 1: Reward Functions

with only the final reward of scoring a goal. Therefore, we had to use reward shaping in order to speed up training for the agent to reach the eventual objective of scoring.

We proposed two general metrics to use for reward shaping. The first was to minimize the distance from the player to the ball, to encourage the car to reach the ball earlier in training. The second was to minimize the distance from the ball to the goal, to encourage the bot to hit the ball in the general direction of the goal. We used the first reward in isolation, and also tested out a combination of the two rewards added together. Since these rewards are given at every time step, they are much more immediate than the delayed reward of a scored goal, and would hopefully help our model learn.

We then designed two general reward strategies. First was to maximize positive behaviors, and second was to minimize negative behaviors. We accomplished the first strategy by calculating an exponentially-scaled inverse distance, which leads to greater reward for a shorter distance. We found that this strategy was used in previous work in Rocket League. For the second strategy, we multiplied the linear distance by a negative scalar, which leads to more negative rewards for a further distance. At each timestep, we gave this negative reward. We had not seen the usage of negative reward shaping in previous work, but we designed the negative strategy in order to incentivize the agent to reach a better state as quickly as possible, to avoid accruing more negative rewards. We combined the two rewards and two reward strategies to construct four different reward functions to train on, which are displayed in Table 1. All of the reward strategies still rewarded the final objective of scoring the goal at the end.

4.3.2 Learning Approach

Because there are five possible kickoff starting states in Rocket League, we had to determine how to best train the agent to perform a kickoff from any of these locations. Many papers that we researched in similar topics used a general learning strategy, where the agent was trained on all the start states from the beginning.

We also wanted to investigate the use of curriculum learning in our problem. Curriculum learning is a strategy in reinforcement learning where a larger problem is broken into easier sub-problems given domain knowledge of the problem space. The agent is first trained on the easier problems, and then trained on more difficult scenarios after it has some basic ability. We hoped to use curriculum learning to train the bot on one kickoff position first, and then update that model to learn all of the kickoff positions given its prior experience. Ideally, using curriculum learning would speed up training time and allow the agent to learn an optimal policy faster.

To investigate these two possible learning strategies, we took the best-performing reward function from the previous section, and trained one agent from all start states from scratch, and then one on curriculum learning after learning one kickoff.

4.3.3 Terminal Conditions

Another parameter were able to control is the terminal condition. Meaning, in a given rollout, we could end the rollout early if a certain terminal condition was met. We decided to implement two forms of terminal conditions. First is an event-based terminal condition, and the second is a time-based terminal condition. Within the context of our problem, a natural end condition is scoring a goal - once a goal is scored, we give the agent a large reward and end that particular

training episode. Since we have a fixed timestep that our overall training time out at (usually 1,000,000 timesteps), ending a training episode early leaves more time to do more training episodes.

We also implemented a time-based terminal condition - episode length. This means that after x seconds, the current training episode would terminate. We played with this number initially, as similar to creating the goal-based terminal condition, minimizing the episode timeout condition would allow us to train more, but we also needed to give the ball enough time to score in the first place. After some trial and error, we found that somewhere between 60 and 70 time steps worked best, but this is not a conclusive result. A potential future project might deep dive into these terminal conditions and explore their impact on an agent's training.

4.3.4 Training Time

We trained all of our agents for 1 million timesteps, which is equivalent to about 18 hours of in-game playtime. Through early stages of trial and error, we found that the model tended to converge before 1 million timesteps, and would no longer explore entirely new behavior. This is not a conclusive parameter, however, and more training time might lead to better results.

4.4 Code

The code for our implementation can be found here: `https://github.com/rmalde/rlbot-238`

5 Results and Discussion

Once our agents were trained, we used the following metrics to objectively evaluate their performance: percent of episodes that involved the cars scoring a goal, percent of episodes that involved the car hitting the ball, and the average time that it took to hit the ball (given that the ball was hit).

5.1 Positive vs. Negative Reward Functions

In our first set of rewards, we're going to compare the results between bots that we trained on a positive reward and bots that we trained on a negative reward. We explained the reward functions and why we chose them in section 3.3.1.

Here we have graphs showing how a bot following each reward function trained, as a function of reward per time step over the entire training duration. The Y-Axis of each graph is the reward gained, and the X-Axis is the timestep.

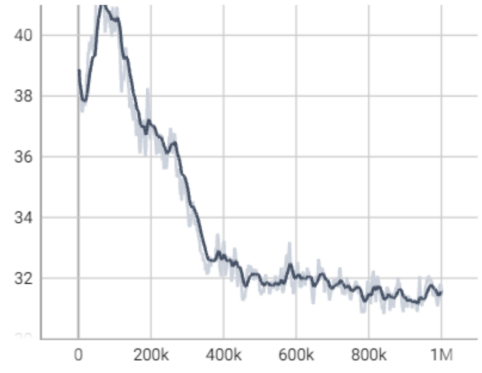

Figure 3: Reward over time: Player Distance to Ball Reward

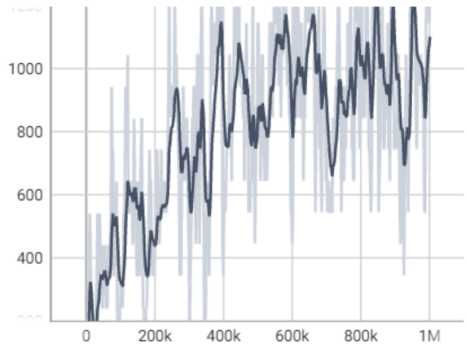

Figure 4: Reward over time: Player Distance to Ball + Ball Distance to Goal Reward

From Figure 4, we see that the negative reward strategy started seeing improvement much earlier in training compared to the positive reward strategy. This would lead us to conclude that the negative reward strategy leads to more efficient and faster training, particularly for straightforward, goal-oriented tasks like scoring the kickoff. Table 2 also shows that the negative reward strategy performed 25% higher on percent goals scored for the player to ball reward function, and 32% higher goal scored on the combination reward function. The negative rewards also led to a much faster average time to hit the ball, likely because the punishing of negative rewards at each time step incentivizes reaching a more optimal state faster.

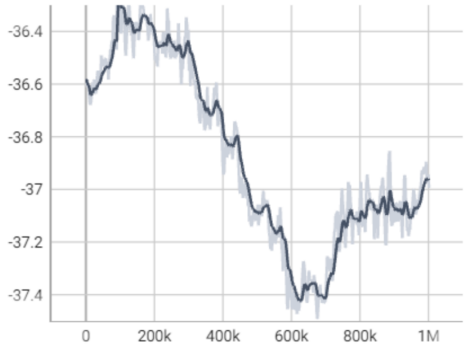

Figure 5: Reward over time: Negative Player Distance to Ball Reward

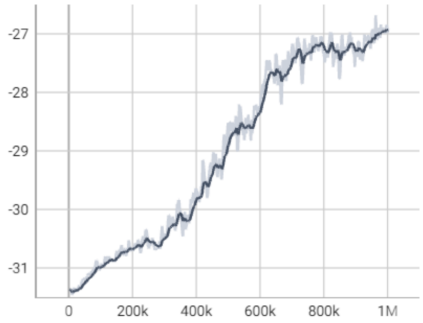

Figure 6: Reward over time: Negative Player Distance to Ball + Ball Distance to Goal Reward

Reward	% Score Goal	% Hit Ball	% Avg Time to Ball
Positive Player Distance to Ball + Positive Goal	0	10	3.1s
Positive Player Distance to Ball + Ball Distance to Goal + Positive Goal	2	25	2.9s
Negative Player Distance to Ball + Positive Goal	27	72	2.3s
Negative Player Distance to Ball + Ball Distance to Goal + Positive Goal	34	90	2.3s

Table 2: Performance of Positive vs. Negative Rewards

Between the two rewards, of player to ball and the combination reward function, we saw slight improvement for using the combination reward function. While there was a 250% increase in how often the ball was touched, the actual goal scoring percentage only increased by 2%. We also manually assigned weights to each of these rewards, so these weights could definitely be tuned as hyperparameters to increase model performance. Several of the agents trained on various reward functions performed better than the baseline in several of the metrics.

5.2 General vs. Curriculum Learning Approaches

In our second set of rewards, we take the reward function that performed best (which was the negative reward function that had the distance of player to ball and distance ball to goal components), and then explore how we can translate our results from one kickoff position to all of the kickoff positions. As discussed in section 3.3.3, we took two different approaches - general learning and curriculum learning.

Here we have graphs showing how a bot following general learning and curriculum learning trained, as a function of reward per time step over the entire training duration. The Y-Axis of each graph is the reward gained, and the X-Axis is the timestep.

While general learning saw a gradual improvement in average reward across training, as seen in figures 7 and 8, we see that curriculum learning failed to optimize the reward as well as the general reward. In fact, it performed worse than the general learning approach. To examine this further, we viewed the agent interacting with the environment live, and found that the agent seemed to be stuck in completely inefficient local optima when trying to apply curriculum learning from it's previous optimal policy. The agent would flip towards the ball, but then veer off in another direction and rarely ever actually hit the ball. For future work, we would have to investigate how to effectively use curriculum learning without the dangers of finding local optima policies that are suboptimal.

Reward	% Score Goal	% Hit Ball	% Avg Time to Ball
General Learning	29	83	2.4s
Curriculum Learning	15	61	2.5s

Table 3: Performance of Curriculum vs. General Learning

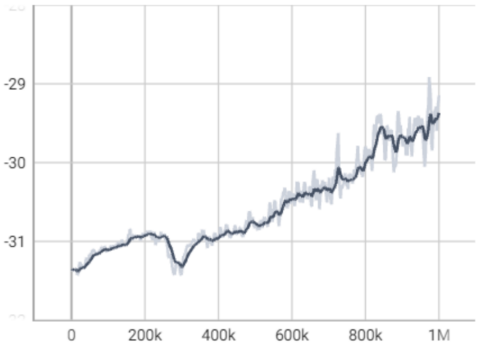

Figure 7: Reward over time: General Learning

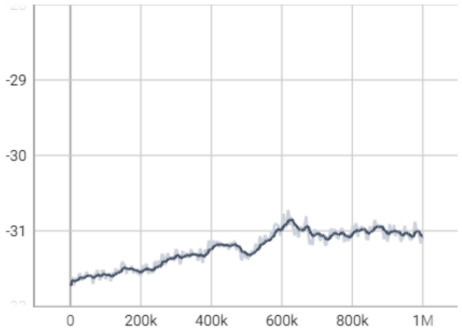

Figure 8: Reward over time: Curriculum learning

5.3 Additional Findings

We were also able to qualitatively analyze the trained agent behaviors when viewing them perform the kickoffs in real-time in Rocket League. In our best-performing bot, with our best reward function and a tighter timeout condition of 60 timesteps, we found that it had learned an advanced mechanic called the "speed flip" to reach the ball quicker. This mechanic is extremely difficult to pull off in-game, requiring several quick joystick movements within a short amount of time, is only used by higher-level players, and was actually only discovered 3 years after the game was released. It is extremely fascinating, then, that our agent learned this advanced mechanic during training. While we may still be far away from having a Rocket League bot that can beat professional players, given unique scenarios reinforcement learning could be used to find new and optimal mechanics within the game that have not been discovered yet.

The bot might have been forced to reach the ball as fast as possible and shoot the ball as hard as possible because of the negative reward shaping and the tight terminal condition, which barely allowed any room for error. A great area for further investigation would be to adjust the episode length and the weights for reward shaping, to see even greater optimization for the kickoffs.

The video of our best performing, all kickoffs bot can be found here: `https://youtu.be/vx8clBZe5Yc`

6 Conclusion

In this work, we set out to use Reinforcement Learning in order to train a bot to play Rocket League. Given the challenges associated with training an agent to play in such a large, 3D environment, we constrained our problem to just the initial kickoff phase of the game. Our goal was to create a bot that, given a random kickoff starting position, could score a goal (with no opponents on the field). We looked at current literature and tried approaches that hadn't been tried yet, including using Negative Rewards and employing Curriculum training (to try and generalize one kickoff starting position to all others). After training numerous bots with varying reward functions and training methods, we found the following. Negative rewards significantly improved our bot's performance, allowing our bot to reliably score a goal from a single kickoff position. However, the curriculum training approach was not quite as successful, performing far worse than a general training method when training the bot on all kickoff positions. Thus, our best bot was one that was trained on a Negative Reward function but was trained using a random sampling of the kickoff positions.

7 Next Steps

Although we made significant progress during the course of our project, we have only scratched the surface in the endeavor to create a Rocket League bot that can outperform the best players in the world. Our first step is to take our approach to train a Rocket League kickoff bot, and generalize it to other aspects of the game. If we were to continue our research, we would next look at training a bot to defend their own goal, while facing an opponent who is trying to score a goal. Additionally, we would like to explore how the dynamics of the problems change as you add more players, both opponents, and teammates. Questions of communication between teammate bots would be especially interesting.

Finally, for a general game-playing agent, the prospect of training the bot from scratch to play the entire game would require an obscene amount of resources. To amend this, we would also like to further explore curriculum learning, and see if it can be applied to learning other aspects of the game without degrading the overall quality of the agent, like what happened in our tests.

References

[1] Max Jaderberg, Wojciech M. Czarnecki, and Iain Dunning. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[2] SaltieRL. Saltie. `https://github.com/SaltieRL/Saltie`, 2019.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[4] Oriol Vinyals, Igor Babuschkin, and Wojciech M. Czarnecki. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, October 2019.