ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seer: Reinforcement Learning in Rocket League

Master Thesis

Neville Walo

Wednesday 24th August, 2022

Supervisor: Prof. Dr. Fernando Perez Cruz
Advisor: Dr. Nathanaël Perraudin

Swiss Data Science Center (SDSC)
Department of Computer Science, ETH Zürich

Abstract

Reinforcement learning studies how intelligent agents should act in an environment to maximize the notion of cumulative reward. Reinforcement learning has been used to achieve superhuman performance in many board games such as chess and Go. Only recently has reinforcement learning made its way into the world of e-sport, and has already achieved great success by defeating the world champions in Dota 2 and mastering StarCraft 2. In this thesis, we investigate how reinforcement learning can be used to create an intelligent agent that plays Rocket League. We created *Seer*, a Rocket League agent that plays 1v1 and is better than about 50 percent of the Rocket League player base. Additionally, we show that it is possible to learn from human replays to accelerate training. This work may be a step on the road to creating a superhuman agent in Rocket League.

Acknowledgements

Uppermost, I would like to thank my advisor, Dr. Nathanaël Perraudin, for the support throughout the last months and for providing guidance and feedback during this project. Furthermore, I want to thank Prof. Dr. Fernando Perez Cruz for supervising my thesis. I would also like to thank the entire Rocket League bot community, without whom this thesis would not have been possible.

Contents

Chapter 1

Introduction

Reinforcement learning has been used to achieve superhuman performance in many board games such as Backgammon [41], Shogi [35], Chess [35], and Go [34, 36]. Reinforcement learning has also made its way into the world of video games and is trying to outperform humans here as well. Many older Atari games could already be mastered [21]. In 2019, OpenAI Five defeated the world champions in Dota 2 [6] and AlphaStar was able to master StarCraft 2 [3]. Dota 2 and StarCraft 2 present immense challenges for reinforcement learning due to long time horizons, partial observability, high dimensionality of observation and action spaces and complex rules. Both games were mastered by training giant networks over multiple months using large batch sizes, enormous computational effort and distributed training.

In this thesis, we use supervised methods and reinforcement learning to create an agent that plays Rocket League. Rocket League differs from most other games, as Rocket League is a physics based esports, thus creating new challenges. The key to victory in Rocket League is not only choosing the right strategy, but also understanding the physics behind the game and acquiring the mechanical skills to maneuver the car in the arena. Many actions per second are required to play the game, and just one miscalculated action can make the difference between winning or losing.

To potentially use less computational resources to train Seer, we use data from human replays. Inputs and outputs of human replays were extracted to train the network in a supervised manner to give the model a prior. By learning from human replays, we hoped to spend fewer resources training an agent than if we relied solely on reinforcement learning. We show that giving the model a prior can indeed speed up training in the early stages, but shows diminishing returns in later stages.

Seer has learned to play Rocket League all by itself using self-play, without

the need for scripted actions or other hard-coded events. Seer trained for about a month and a half, during which Seer played Rocket League for about 20 years. The final version of Seer is roughly in the Platinum I rank (see Sec. 2.2) and thus in the top 50 percent of the Rocket League player base in 1v1.

Chapter 2 provides the reader with the theoretical background for this thesis. In Chapter 3, we describe the methods used and how Seer was trained. In Chapter 4, we present our evaluation methods and evaluate Seer. Chapter 5 provides a conclusion of this thesis.

1.1 Related Work

Lately, much research has been conducted on the application of reinforcement learning to master e-sport video games. Most notably are the successes of OpenAI Five in Dota 2 [6] and AlphaStar in StarCraft 2 [3]. Reinforcement Learning has also been used to master other games, such as Atari [21], Backgammon [41], Shogi [35], Chess [35], and Go [34, 36].

Simultaneously with this work, a lot of effort was made by the Rocket League community to create a superhuman agent. Most agents are still training and not yet published, but the current versions can usually be found in the RLBotPack[1]. Most notable are the agents *Necto* and its successor *Nexto*[2], which are currently the best Rocket League bots. Both *Necto* and *Nexto* use a transformer-like architecture to handle a variable number of players on the field. Attempts have also been made to create a Rocket League clone in Unity to speed up training and perform a sim-to-sim transfer [24].

[1] https://github.com/RLBot/RLBotPack
[2] https://github.com/Rolv-Arild/Necto

Chapter 2

Background

This chapter provides the background knowledge about reinforcement learning and Rocket League for this thesis.

2.1 Reinforcement Learning

Reinforcement learning is one of the three fundamental paradigms of machine learning, along with supervised learning and unsupervised learning. Reinforcement learning studies how intelligent agents should act in an environment to maximize the notion of cumulative reward.

Although the literature on reinforcement learning is vast and many algorithms exist, we will mainly focus on the derivation of the Proximal Policy Optimization (PPO), the algorithm used in this work. We first introduce Markov Decision Processes (MDPs) (Sec. 2.1.1), the origin of reinforcement learning. Then, we introduce policy gradient methods (Sec. 2.1.2), Advantage Actor-Critic (Sec. 2.1.3) methods, Importance Sampling (Sec. 2.1.4), Generalized Advantage Estimation (GAE) (Sec. 2.1.5), Entropy Regularization (Sec. 2.1.6) and finally arrive at the Proximal Policy Optimization (Sec. 2.1.7) algorithm.

2.1.1 Markov Decision Process (MDP)

A Markov Decision Process is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision-making in situations where outcomes are partially random. It is used in many disciplines, including robotics, automatic control, economics, and machine learning.

We denote a Markov Decision Process as $(S, A, P, p_0, r, \gamma)$ where

- S is the discrete state space,
- A is the discrete action space,

- $P : S \times A \times S \to [0, 1]$ is the state transition probability,
- $p_0 : S \to [0, 1]$ is the initial state distribution,
- $r : S \times A \to \mathbb{R}$ is the reward function,
- $\gamma \in [0, 1)$ is the discount factor.

The optimization objective is to find a stochastic policy $\pi : S \times A \to [0, 1]$ which optimizes the expected discounted cumulative reward

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{2.1}$$

where r_t is the reward received at the time step t in the trajectory $(s_0, a_0, r_0, s_1, ..., s_t, a_t, r_t, ...)$ with $s_0 \sim p_0$, $s_t \sim P(\cdot|s_{t-1}, a_{t-1})$, $a_t \sim \pi(\cdot|s_t)$, $r_t = r(s_t, a_t)$. Fully observed Markov Decision Processes can be solved in polynomial time using for example value iteration, policy iteration or linear programming [38].

Planning in partial or unknown MDPs is referred to as reinforcement learning (RL). There are two basic approaches for solving reinforcement learning problems: Model-based RL and model-free RL. With model-based methods (e.g., R-Max [7]), one attempts to learn the underlying MDP and determine the transition probabilities and reward function. Once the MDP is sufficiently explored, the same methods as for fully observed MDPs can be used. With model-free methods, one tries to learn the optimal policy directly without knowing the underlying MDP. This can be achieved using for example policy gradient methods or learning the value function $V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$ (TD-learning [37]) or the Q-function $Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) * V^\pi(s')$ (Q-learning [45]) directly to find the optimal policy.

Reinforcement learning algorithms can be further divided into off-policy and on-policy algorithms. In an on-policy algorithm, data gets collected using the current version of the policy, which means that the agent has implicit control over which data to collect. This means that the agent can decide whether to gather more data to avoid missing out on a potentially large reward (exploration) or stick with the current knowledge and build an optimal policy for the data the agent has seen (exploitation). In an off-policy algorithm, there is no control of how the data gets collected, which means that the algorithm cannot trade off exploitation and exploration and also that data might get collected using a different policy.

2.1.2 Policy Gradient

One of the challenges with traditional MDPs is scaling to large state and action spaces. For example, if one is dealing with a continuous state or action

space, or with Partially Observable Markov Decision Processes (POMDP) that have an exponentially growing state space, conventional methods are impractical. One way to scale to a large state space is to learn a parameterized value function $V_\theta^\pi(x)$ or a parameterized Q-function $Q_\theta^\pi(s,a)$ (DQN [22], DDQN [42]), which can be optimized using gradient descent. To scale to large action spaces, one can learn a parameterized policy π_θ which can be optimized by obtaining the policy gradient $\nabla_\theta J(\pi_\theta)$ of the expected discounted cumulative reward with respect to the policy parameters θ. Although is not possible to compute the policy gradient $\nabla_\theta J(\pi_\theta)$ directly, the REINFORCE trick [46] allows estimating the policy gradient by Monte Carlo sampling a trajectory τ of length T using the current policy (on-policy) and then computing the REINFORCE update

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[G_\tau \nabla_\theta \log \pi_\theta(a_\tau|s_\tau)\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]\end{aligned} \tag{2.2}$$

where $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ denotes the discounted reward after time step t.

2.1.3 Advantage Actor-Critic

The vanilla REINFROCE update (Eq. 2.2) suffers from high variance and noisy gradients, and therefore poor sample efficiency. It is possible to reduce the variance of the REINFORCE update by introducing a baseline b (e.g. $b = \frac{1}{T}\sum_{t=0}^{T} G_t$)

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} (G_t - b) \nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{2.3}$$

To further improve the sample efficiency, one can rewrite $\nabla_\theta J(\pi_\theta)$ according to the policy gradient theorem [39]

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} Q^\pi(s_t,a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{2.4}$$

which leads to Q Actor-Critic methods. In Q Actor-Critic, it is again possible to use a baseline b to improve the sample efficiency. Choosing the baseline as the value function $b = V^\pi(s)$, gives rise to Advantage Actor-Critic methods (A2C/A3C [20])

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} A^\pi(s_t,a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{2.5}$$

where $A^\pi(s_t,a_t) = Q^\pi(s_t,a_t) - V^\pi(s_t) = r_{t+1} + \gamma * V^\pi(s_{t+1}) - V^\pi(s_t)$ is the advantage function.

As shown in equation 2.5, performing the policy gradient update using Advantage Actor-Critic requires two parameterized function, the value function and the policy function. Based on Actor-Critic methods, many other ways of performing gradient updates have been developed to further improve sampling efficiency (e.g. TRPO [30], PPO [32], SAC [10]).

2.1.4 Importance Sampling

The update in general Advantage Actor-Critic methods (Eq. 2.5) requires the data to be sampled on-policy. This means that new trajectories needs to be sampled for every gradient step to perform correct gradient updates. This also means that every collected sample can only be used once, which leads to poor sample efficiency. Sample efficiency can be improved by introducing importance sampling, which provides an unbiased estimate of the policy gradient using off-policy samples of an older policy $\pi_{\theta_{old}}$:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_{\theta_{old}}}\left[\sum_{t=0}^{T}\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A^\pi(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{2.6}$$

The importance-sampled policy gradient can be interpreted as optimizing the surrogate objective

$$L^{IS}(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta_{old}}}\left[\sum_{t=0}^{T} w_t(\theta)A^\pi(s_t,a_t)\right] \tag{2.7}$$

with $w_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$.

Using importance sampling, data efficiency can be improved by performing multiple gradient updates using the same data. Theoretically, it is possible to estimate the policy gradient using data collected from any policy with sufficient support, but using trajectories that are too old (or simply too different) can again lead to high variance and poor sample efficiency.

2.1.5 Generalized Advantage Estimation (GAE)

Using the raw rewards collected during sampling provides an unbiased estimate of the expected return in a given state. However, because of the stochasticity of the environment and the policy, each reward may be a random variable, the sum of which can lead to a high variance in the estimate of expected return. To balance variance and bias in the estimate of the expected return, an n-step return can be defined

$$R_t^{(n)} = \gamma^n * V(s_{t+n}) + \sum_{i=0}^{n-1}\gamma^i r_{t+i} \tag{2.8}$$

where the sum is truncated after n steps and the rest is approximated using the value function. Choosing $n = 1$ results in a 1-step return $R_t^{(1)} = r_t + \gamma V(s_{t+1})$ (high bias but low variance) commonly used in Q-learning [22], while using $n = \infty$ recovers the original return (unbiased but high variance). Therefore, n acts as a trade-off between bias and variance.

Another way to trade off variance and bias in the estimate of the expected return is to use a λ-return [40]

$$R_t(\lambda) = (1 - \lambda) \sum_{i=1}^{\infty} \lambda^{i-1} R_t^{(i)} \tag{2.9}$$

which is an exponentially weighted average of the n-step returns with decay parameter λ. Choosing $\lambda = 0$ gives $R_t^{(1)}$, while choosing $\lambda = 1.0$ recovers the original return. Therefore, λ acts as a trade-off between bias and variance.

Updating the value function using the temporal difference computed with the λ-return results in the $TD(\lambda)$ algorithm [40]. Similarly, estimating the advantage with the λ-return, yields the Generalized Advantage Estimator $GAE(\lambda)$ [31].

2.1.6 Entropy Regularization

Using reinforcement learning, an agent may get stuck in a local optimum prematurely. This could happen because the agent learns early on that a particular action yields a relatively high reward, and from then on, only performs that action, missing out on a potentially even greater reward and thus missing the global optimum.

In policy gradient methods, to prevent the agent from getting stuck in a local optimum and to trade-off exploration and exploitation, the policy loss can be augmented using entropy regularization:

$$H(\pi(\cdot|s_t)) = -\sum_a \pi(a|s_t) \log(a|s_t). \tag{2.10}$$

By adding $H(\pi(\cdot|s_t))$ to the loss: $L_{new} = L_{old} + \alpha H(\pi(\cdot|s_t))$, the agent will favor actions with higher entropy therefore favoring exploration. Exploration and exploitation can be balanced by choosing the hyperparameter α accordingly.

2.1.7 Proximal Policy Optimization (PPO)

Proximal Policy Optimization [32] is a state-of-the-art actor-critic reinforcement learning algorithm. The key idea in PPO is that the next policy should

remain close to the previous policy. In addition, PPO uses generalized advantage estimation $GAE(\lambda)$ to balance bias and variance, uses entropy regularization to balance exploration and exploitation and uses a replay-buffer in combination with importance sampling to improve sample efficiency.

The PPO algorithm alternates between sampling data through interaction with the environment and optimizing a surrogate objective function using stochastic gradient ascent. During the roll-out phase, PPO collects fixed-length trajectories and stores the trajectories in a replay buffer. Afterwards, the replay buffer is used to perform multiple epochs of mini-batch updates to minimize the surrogate loss.

As mentioned several times, policy gradient methods suffer from high variance and noisy gradients, which means that the gradient updates performed are not always well-behaved. To counteract the problem of taking large gradient steps in the wrong direction, one can introduce the concept of a *trust region*. The trust region is intended to prevent the policy from changing too much in just one gradient update, i.e., leaving the trust region. The first algorithm to use the concept of a trust region is Trust Region Policy Optimization (TRPO [30]) which includes an additional KL-divergence constraint to prevent the behavior of the current policy from deviating too far from the previous policy,

$$\mathbb{E}_\tau \left[KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] < \delta \tag{2.11}$$

where δ is a hyper-parameter. TRPO has been successfully applied to solve a wide variety of RL problems, however, ensuring that the constraint (Eq. 2.11) is satisfied can be difficult and computationally expensive. PPO also uses the concept of a trust region, but PPO replaces the hard constrained from TRPO with a surrogate loss. There exist two versions of PPO, one which uses a clipped objective

$$L^{CLIP}(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \left[\sum_{t=0}^{T} \min \left[w_t(\theta) A^\pi(s_t, a_t), w_{clip}(\theta) A^\pi(s_t, a_t) \right] \right] \tag{2.12}$$

where $w_{clip}(\theta) = clip(w_t(\theta), 1-\varepsilon, 1+\varepsilon)$ and ε is a hyper-parameter, and one which uses an adaptive KL penalty

$$L^{KL}(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \left[\sum_{t=0}^{T} w_t(\theta) A^\pi(s_t, a_t) - \beta KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \tag{2.13}$$

where β is a hyper-parameter. We will focus on the version which uses clipping, as it more widely used, and the one used in this thesis. To prevent policies from diverging too far from each other, $clip(w_t(\theta), 1-\varepsilon, 1+\varepsilon)$ sets the gradient to zero whenever the policy has diverged too much from the previous policy. Thus, $w_{clip}(\theta)$ has a similar function as the KL divergence

constraint in TRPO. Then, the minimum is taken between the clipped and unclipped advantages to create a lower bound.

To summarize, the loss of PPO is given by

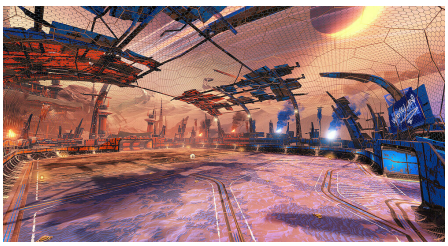
$$L^{PPO}(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 H(\pi_\theta) \tag{2.14}$$

where c_1, c_2 are hyperparameters, $L^{CLIP}(\theta)$ is the loss of the policy function, $L^{VF}(\theta)$ is the squared-error loss of the value function and $H(\pi_\theta)$ in an entropy bonus.

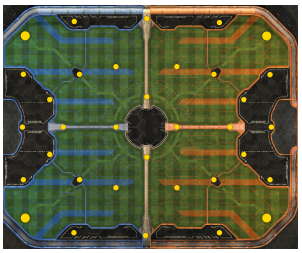
2.2 Rocket League

Rocket League is a vehicular soccer video game developed and published by Psyonix in 2015. It has up to eight players assigned to each of the two teams, one team has the team color blue, the other team has the team color orange. The players use rocket-powered cars to play soccer and score points over the course of a match. The game is played in a symmetric arena with a goal on each side, see Fig. 2.1a. Matches are five minutes long, with a sudden death overtime if the game is tied at that point. After each goal, the players are reset to a specific kick-off position.

The cars can drive, jump, dodge and fly to hit the ball while in the air. Each car can store up to 100 boosts that can be used to move across the field more quickly or to propel itself forward in flight. Players can alter their car's orientation while midair, which combined with midair boosting allows for a controlled flight. To collect boosts, players can drive over 34 boost pads that are spread across the field, see Fig. 2.1b. If one drives into an opponent with enough speed, it is possible to demolish the opponent, who will then be revived after 3 seconds on its side of the field.



(a) Example Arena: Wasteland. [44]



(b) Boost pad locations in Rocket League.

Figure 2.1: **Rocket League Arena and Boost Pads:** Rocket League is played in a symmetric arena with 34 boost pads spread across the field.

2.2.1 Ranking System

Rocket League has its own ranking system to determine players' skills in competitive online play. The skill level is determined by a single number called MMR (Matchmaking Rank). Fig. 2.2 shows the associates emblem for each rank. Each rank (except Supersonic Legend) is divided into 3 sub-ranks (e.g. Bronze I, Bronze II and Bronze III) and each sub-rank is then further subdivided into 4 divisions (e.g. Bronze I Division 1, Bronze I Division 2, Bronze I Division 3, Bronze I Division 4). Fig. A.2 shows the approximate rank distribution of the player base in the 1v1 game mode.

Figure 2.2: **Competitive Ranks in Rocket League:** Ranks from left to right: Bronze, Silver, Gold, Platinum, Diamond, Champion, Grand Champion, Supersonic Legend. [26]

2.2.2 Cars and Hitboxes

The players interact with the game using a car. While there are over 100 cars to choose from, the underlying physics engine only makes use of 6 different hitboxes. The 6 hitboxes are: Breakout, Dominus, Hybrid, Merc, Octane, Plank. See Fig. 2.3 for an example of a hitbox. The hitboxes differ in shape (length, width, height), handling, boost consumption, ground height and inclination. The exact physical properties of the hitboxes are largely unknown, the differences have mostly only been verified through testing.

Figure 2.3: **Octane with Octane Hitbox:** Rocket League is played using a car. There are 6 different hitboxes and over 100 cars to choose from. [33]

2.2.3 Frameworks

RLBot To interact with the game, we use the RLBot framework [27]. The RLBot framework enables the creation of custom bots for Rocket League by providing an API to connect to the game. The API uses sockets to communicate game state and actions between the game and a custom bot. The action space used by the RLBot framework consists of 8 actions, see Table 2.1, with 3 boolean actions and 5 continuous actions in the interval $[-1, 1]$. Note that all actions can be used simultaneously and independently. However, some actions have no effect in certain scenarios, e.g. using the action *boost* does nothing if the car has no boost. The game state provided by the API contains essentially all information about the current game at a given time step. This includes the physical state of the ball and cars, information about the boost pads, the remaining time and more. For an example game tick packet, see A.1. This game tick packet will be provided to the bot up to 120 times per second (the engine's internal physics tick rate), and the bot can return an action for each packet.

Table 2.1: **Actions in RLBot:** 8 different actions are available to control a bot in Rocket League. There are 5 continuous actions in the interval $[-1, 1]$ and 3 boolean actions. There is no dependency between the actions.

Action	Domain
Throttle	$[-1.0, 1.0]$
Steer	$[-1.0, 1.0]$
Pitch	$[-1.0, 1.0]$
Yaw	$[-1.0, 1.0]$
Roll	$[-1.0, 1.0]$
Jump	$\{0, 1\}$
Boost	$\{0, 1\}$
Handbrake	$\{0, 1\}$

RLGym While the RLBot framework (Sec. 2.2.3) allows custom bots in Rocket League, the RLBot frameworks lacks the ability to run the game faster than real-time, making it impractical to train a reinforcement learning agent. To train Seer, we use the RLGym framework [29], which allows the game to be treated like an Openai Gym-like environment. RLGym uses a BakkesMod plugin [4] to control the game and run Rocket League faster than real-time. In addition, the RLGym framework allows multiple instances of the game to be used simultaneously to increase the batch size. While RLGym's action space is the same as RLBot's, RLGym's observation space is missing some observation, such as game information and boost pad timers.

Chapter 3

Methods

This chapter presents how Seer is trained. We created two versions of Seer, one version which started learning with a random prior (random parameter initialization) and one version which has a prior from supervised learning from human data. This chapter describes all the details of the reinforcement learning procedure and how Seer learned a prior from human data.

The reinforcement learning training system of Seer is shown in Fig. 3.1. Interaction with the game is done using a BakkesMod plugin. The BakkesMod plugin connects to the RLGym framework, which allows Rocket League to be treated like an OpenAI Gym environment. Seer receives observations from the environment and return actions to advance to the next step. Observations, rewards, actions, and advantages are stored in a replay buffer which is used by PPO to update the policy and value function.

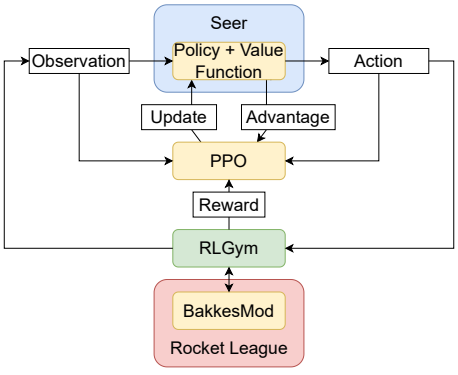


Figure 3.1: **System Overview of Seer:** The RLGym framework is connected to Rocket League via a BakkesMod plugin to communicate actions and observations with the game. The PPO implementation receives observations, rewards, actions, and advantages to update the policy and value function of Seer. Seer receives observations and provides actions and advantages.

3.1 Optimizing the Policy

The goal is to find a policy that maximizes the probability of winning the game. In practice, we optimize a reward function (see Sec. 3.6). The reward function was designed at the start of the project and kept mostly fixed, some minor tweaks were made whenever we observed that Seer may get stuck in a local optimum.

The policy is optimized using Proximal Policy Optimization [32] (see Sec. 2.1.7). The implementation is based on the Recurrent PPO version of stable-baseline 3 [25], to which some improvements were made. The used software can be found in A.3. Seer uses a single neural network to model both the policy and the value function (see Sec. 3.4). The neural network uses a shared LSTM layer that feeds into separate fully connected layers producing policy and value function outputs. The Adam optimizer [18] is applied to optimize the policy and value function using truncated backpropagation through time [47] for 32 epochs per rollout. The rollouts are collected using self-play (see Sec. 3.9), where 20% of the games are played against older versions and 80% of the games are played against the current version.

Seer performs 15 actions per second and has a reaction time of $8 - 75$ ms (see Sec. 3.7). For each time step, Seer receives 159 observations about the game (see Sec. 3.2) and returns one of 1800 actions (see Sec. 3.3). Unavailable actions are masked out to reduce the noise that Seer experiences during training (see Sec. 3.5).

3.1.1 Episode and Rollout

A Rocket League game usually consists of several goals. Instead of selecting the entire game as an episode, each goal itself is selected as an episode. It would be impractical to select the entire game as an episode, as there is a short replay and goal celebration after each goal, which would have to be masked out or otherwise result in noise. It also allows keeping the episodes relatively short ($10s - 120s$ vs. $600s$), which makes learning easier. As a result of this design decision, Seer has no understanding of game time and thus no understanding of certain tactics, such as playing for time.

An overview of how rollouts are collected is shown in Fig. 3.2. The Rocket League physics engine runs at 120 ticks per second. Seer uses a frameskip [16] of 8, which means that one policy time step consists of 8 physics ticks, yielding about 66.6 ms per policy time step. One sample is created by combining 16 time steps that are jointly optimized using truncated backpropagation through time. A rollout then consists of 32 samples or 512 time steps, corresponding to about 34.13 seconds of game time.

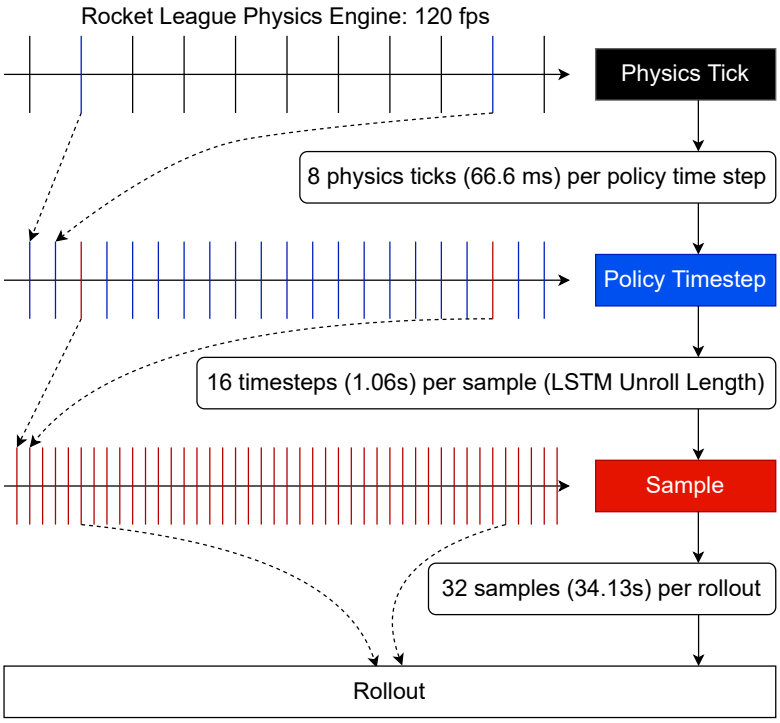


Figure 3.2: **Timescales:** The breakdown of a rollout. A rollout consists of 32 samples, which accounts for about 34.13 seconds of game time. A sample is optimized together using truncated backpropagation through time and consists of 16 policy time steps. A policy time step is equal to the length of 8 physics ticks of Rocket League.

3.1.2 Hyperparameters

All the hyperparameters used to train Seer are shown in Table 3.1. Almost all hyperparameters were kept fixed during training. The ones that were modified are the reward weight for scoring a goal (see Sec. 3.6), the time horizon (see Sec. 3.8), the entropy coefficient (see Sec. 3.10) and the learning rate. The initial learning rate was set to 1e-5 and was then continuously lowered to 5e-6 during training, see Fig. 3.3. An overview of when and how the parameters were modified is shown in Fig. A.1.

A large batch size is usually the key to success when training a reinforcement learning agent (see [6]). With the resources available during this project, we were able to run a maximum of 30 Rocket League instances simultaneously. Since 20% of the games are played against older versions (see Sec. 3.9) and each Rocket League instance produces 32 samples (512 time steps) per rollout, a batch size of 1728 samples (27 648 time steps) was used to train Seer.

Table 3.1: **Hyperparameters:** All the hyperparameters used in training Seer. Most parameters were fixed for the whole experiment, those which were modified during training are indicated $x \leftrightarrow y$.

Parameter	Value
Frameskip	8
LSTM Unroll Length	16
Rollout Length	512
Batch Size (Samples)	1728
Batch Size (Timesteps)	27648
Epochs	32
Time Horizon	$10s \leftrightarrow 20s$
GAE λ	0.95
PPO Clipping	0.2
Value Loss Weight	1.0
Entropy Coefficient	$0.01 \leftrightarrow 0.005$
Learning Rate	1e-5 $\leftrightarrow$ 5e-6
Adam β_1	0.9
Adam β_2	0.999
Past Opponents	20%
LSTM Max Gradient Norm	0.5
Goal Scored Weight	$1.25 \leftrightarrow 1.45$

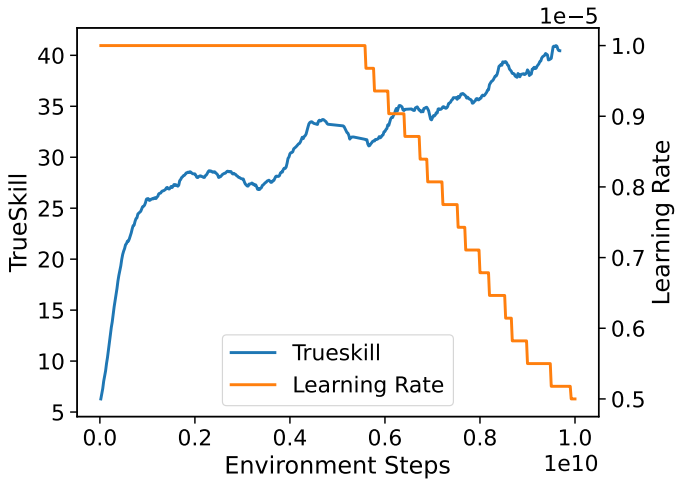


Figure 3.3: **Learning Rate during Training:** The learning rate was initialized to 1e-5 and continuously lowered during training to 5e-6 to help optimization and generalization.

3.2 Observation Space

At each time step, Seer observes 159 inputs about the game, see Table 3.2. The observation contains information about the game state, such as player encodings, the encoding of the ball and boost pad encodings. Most of the information in the observation is provided directly by the frameworks used (RLBot, RLGym), others were computed ourselves e.g. norms. The observation consists mostly of float values, which are treated as such. All boolean observations are encoded as 0.0 or 1.0, and the previous action is encoded using a one-hot encoding. All observations are scaled to be in the range $[-1, 1]$ before feeding them into the neural network.

Table 3.2: **Observation:** Seer receives 159 observations at each time step. The observation consists mostly of physical information about the player and the ball, as well as information about the boost pads. (uu = unreal units)

Game	41	Per Player	54	Per Boost Pad	68
Ball Position $[uu]$	3	Position $[uu]$	3	Active [`bool`]	1
Ball Velocity $[uu/s]$	3	Velocity $[uu/s]$	3	Timer $[s]$	1
Ball Velocity L2 Norm $[uu/s]$	1	Velocity L2 Norm $[uu/s]$	1		
Ball Angular Velocity $[rad/s]$	3	Angular Velocity $[rad/s]$	3		
Previous Action Encoding	19	Rotation (Euler Angles)	3		
Players Position diff. $[uu]$	3	Is Supersonic [`bool`]	1		
Players Position diff. L2 Norm $[uu]$	1	Distance to Ball $[uu]$	3		
Players Velocity diff. $[uu/s]$	3	Distance to Ball L2 Norm $[uu]$	1		
Players Velocity diff. L2 Norm $[uu/s]$	1	Velocity to Ball $[uu/s]$	3		
		Velocity to Ball L2 Norm $[uu/s]$	1		
		Demo Timer $[s]$	1		
		Alive [`bool`]	1		
		Boost [`float`]	1		
		Wheel Contact [`bool`]	1		
		Has Flip [`bool`]	1		

While humans play the game through a screen, Seer receives the information necessary to play the game in a series of data arrays. It is theoretically possible to train an agent using images as input, but impractical in practice, since it would multiply the computational resources manyfold and make it more difficult for the agent to access information. Using data arrays instead of images gives Seer a small advantage, as Seer has nearly complete information about the game at all times. For example, when playing the game as a human, the opponent's boost level is unknown. However, we do not believe that these discrepancies give Seer an unfair advantage, as experienced players can accurately estimate the hidden values.

Seer takes advantages of the fact that Rocket League is played on a symmetric field (see Sec. 2.2) by mirroring all observations if Seer is playing as the orange player. Mirroring the observations if Seer is playing on the orange team significantly reduces the variance of the training data and basically doubles the batch size. This means that Seer will always perceive the game as if Seer were playing as a blue player.

3.3 Action Space

The action space required to play Rocket League consists of 5 continuous actions in the interval $[-1, 1]$ and 3 boolean actions, see Table 2.1. Continuous actions are a good choice when playing the game with a controller, as continuous actions map well to a controller's trigger and joystick, but continuous actions are not the preferred choice when creating a neural network to play the game. As described in [17], there are significant drawbacks to using continuous actions for reinforcement learning in video games, such as longer training time and lower final performance. We therefore discretized the original action space and use a multi-discrete action space.

The action space of Seer is shown in Table 3.3. In total, Seer can perform 1800 different actions. The objective when designing the action space was to keep the action space as small as possible without limiting Seer's ability to play the game. Only 3 bins are used for *throttle* and *roll*, while 5 bins are used for *pitch* and *yaw*. Since Seer can switch between actions relatively quickly (see Sec. 3.7), Seer can perform basically all required actions regarding accelerating and rolling using only 3 bins each. As *pitch* and *yaw* are used to determine the dodge direction, we felt that using only 3 bins might limit the performance of Seer.[1] To save one softmax activation, the same action is used for *steer* and *yaw*.[2]

Table 3.3: **Action Space of Seer:** Instead of using continuous actions to control the car, Seer uses a discretized action space to facilitate learning.

Action	Domain
Throttle	$\{-1, 0, 1\}$
Steer/Yaw	$\{-1, -0.5, 0, 0.5, 1\}$
Pitch	$\{-1, -0.5, 0, 0.5, 1\}$
Roll	$\{-1, 0, 1\}$
Jump	$\{0, 1\}$
Boost	$\{0, 1\}$
Handbrake	$\{0, 1\}$

Many Rocket League agents use scripted actions to perform the kickoff, as this is one of the most important mechanics to get right when playing the game at a high level. Seer does not use scripted actions, but Seer learned everything itself.

[1] There is a huge debate in the Rocket League community whether this is true or not, as keyboard and mouse players only have 3 bins (8 dodge directions), while controller players have many more.

[2] Using the same action for *steer* and *yaw* is possible since *steer* can only be used on the ground and *yaw* only in the air. It is also how humans control the car.

3.4 Neural Network Architecture

To take advantage of transfer learning, Seer uses an architecture in which the value function and action policy share a network and share gradients, see Fig. 3.4. Additionally, a Long Short-Term Memory (LSTM) [12] layer is used to give the network the ability to remember previous events. The combined policy and value network uses 2 015 127 parameters.

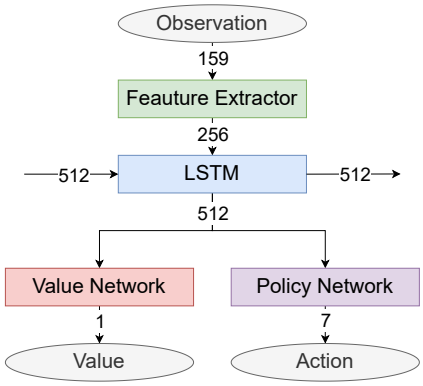
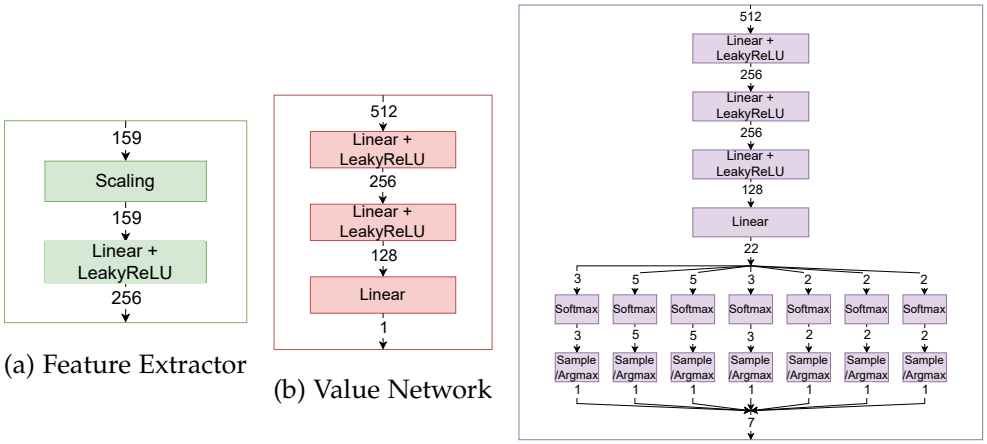


Figure 3.4: **Overview Joint Policy and Value Network:** Seer uses a joint policy and value network that shares layers and gradients. The input is processed into a single vector, which is then passed to a single-layer LSTM. The output of the LSTM is fed into separate policy and value networks.

First, the observations are pre-processed by the feature extractor into a single vector that summarizes the state. Then, the vector is processed by a single-layer LSTM. The output of the LSTM is then fed into the policy and value network to predict value and action.

The feature extractor receives as input observations as described in Table 3.2. The observation has already been augmented to always correspond to the blue player's view (see Sec. 3.2). The observation is first scaled to be in the interval $[-1, 1]$ and then fed into a simple feedforward neural network, see Fig. 3.5a. The output of the feature extractor is then fed into a single-layer LSTM of size 512. The LSTM additionally receives the hidden state and the cell state of the previous time step as input in each time step. The output of the LSTM is then fed into the policy and value network. The value network is just a simple feedforward neural network, see Fig. 3.5b. The policy network first feeds the LSTM output through a feedforward neural network and then divides the output into different actions, having an output for each action, see Fig. 3.5c. A softmax is then applied to each action, resulting in a multicategorical distribution. One can then either sample from this multicategorical distribution or choose the action with the highest probability, which results in the 7 actions needed by Seer to play Rocket League.



(a) Feature Extractor

(b) Value Network

(c) Policy Network

Figure 3.5: **Details of the Network Architecture:** The feature extractor first scales the input to be in the interval $[-1, 1]$, and then feeds into a single fully connected layer. The value network consists of 3 fully connected layers. The policy network first feeds into 4 fully connected layers, afterwards the input is split into multiple actions and for each action a softmax is applied.

3.5 Masking

As described in Section 3.3 and 3.4, Seer uses 7 discrete actions to play Rocket League. In total, there are 22 different action values to choose from (some actions have two, three or five possible values). However, not all possible action values are available at all times. For example, the action *boost* is only available if the car has any boost at all. Following [13], to reduce the noise experienced during training, all unavailable actions are masked out at each time step.

To mask out unavailable actions, the logits in the final linear layer are replaced with a large negative value (e.g. $-1e8$), see [13]. Table 3.4 shows all the actions that are masked out at each time step. The actions *pitch* and *roll* only have an effect if the car is in the air; therefore, all non-zero actions are masked out if the car is on the ground. The *boost* action is masked out if the car has no boost. The *jump* action is masked out if jumping is not available. Seer can only use the handbrake if the car is on the ground. Finally, the *throttle* action does not have an impact if the car is in the air[3], therefore *throttle* is a good candidate for masking. However, jumping and accelerating are the only possibilities to flip the car around if the car is stuck upside down. Since we wanted to give Seer the possibility to do so, it is not possible to

[3]This is actually not true, since using *throttle* provides small acceleration even when in the air. But it is mostly negligible.

mask out both actions. In the end, the *jump* action was masked out.

Table 3.4: **Masked Actions:** To reduce the noise experienced during training, unavailable actions are masked out. Some actions are only available in the air or on the ground, while some actions require certain conditions.

Action	Domain	Condition
Throttle	$\{-1\}$	On Ground
Pitch	$\{-1, -0.5, 0.5, 1\}$	In Air
Roll	$\{-1, 1\}$	In Air
Jump	$\{1\}$	Has Flip
Boost	$\{1\}$	Has Boost
Handbrake	$\{1\}$	On Ground

While the masking defined above helps Seer to learn faster, the masking also prevents Seer of performing some of the more advanced mechanics in Rocket League. For example, it is sometimes useful to use the handbrake even if only one of the four wheels touches the ground to perform a *wavedash*. Furthermore, masking the *jump* action disables the *turtle* mechanic and only enabling *pitch* in the air disables *wall dashing*. Of course, it is also possible that Seer is unable to perform some yet unknown mechanics. The objective when deciding which action to mask was to mask as many actions as possible without limiting Seer's ability to play the game. If masking limits Seer's performance by preventing Seer from performing some specific actions, it is possible to remove the masking once Seer is appropriately skilled and let Seer learn the missing mechanics.

3.6 Reward Weights

The ultimate goal of Seer is to win the game. To simplify the credit assignment problem and speed up learning, a detailed reward function is used. Table 3.5 shows all the rewards that Seer can earn with their associated weights. The rewards are divided into conditional rewards, which are received only when a certain condition is met, and continuous rewards, which are received at each time step. At each time step, each reward is multiplied by its associated weight to obtain the reward for the agent. Furthermore, it is ensured that the reward is *zero sum* by subtracting from each agent's reward, the reward of the opponent agent. Finally, the reward is normalized using a running mean and variance.

When designing the reward function, we tried to give Seer rewards for actions that are considered to be good, weighted in such a way that Seer should learn quickly to play the game but does not learn a suboptimal pol-

Table 3.5: **Reward Weights**. At each time step, Seer receives a reward that is a weighted sum of 16 individual rewards. Some rewards are conditional (top part), while others are received at each time step (bottom part).

Name	Weight	Range
Goal Scored	$1.25 \leftrightarrow 1.45$	$[0, 1.5]$
Boost Difference	0.1	$[0, 1]$
Ball Touch	0.1	$[0, 2]$
Demo	0.3	$[0, 1]$
Distance Player Ball	0.0025	$[0, 1]$
Distance Ball Goal	0.0025	$[0, 1]$
Facing Ball	0.000625	$[-1, 1]$
Align Ball Goal	0.0025	$[-1, 1]$
Closest to Ball	0.00125	$[0, 1]$
Touched Last	0.00125	$[0, 1]$
Behind Ball	0.00125	$[0, 1]$
Velocity Player to Ball	0.00125	$[-1, 1]$
Kickoff	0.1	$[-1, 1]$
Velocity	0.000625	$[0, 1]$
Boost Amount	0.00125	$[0, 1]$
Forward Velocity	0.0015	$[-1, 1]$

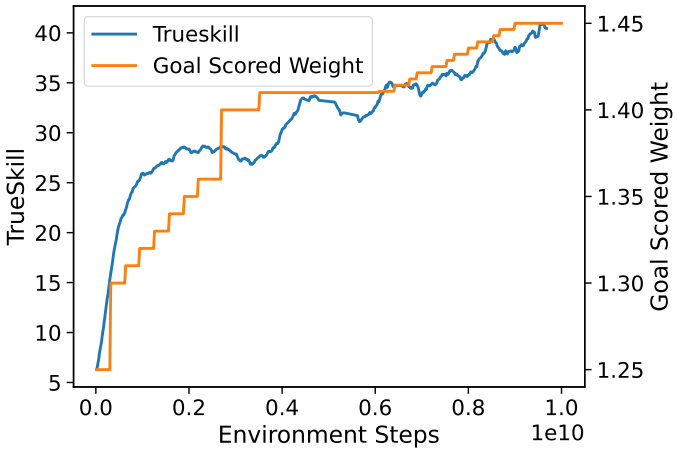


Figure 3.6: **Goal Scored Weight during Training:** The weight for the reward for scoring a goal was initialized to 1.25 and continuously increased during training to 1.45 to ensure that Seer does indeed place the highest priority on scoring.

icy. Initially, we set the reward for scoring a goal and then designed all other rewards in comparison to this reward. During training, the weight of the reward for scoring a goal was gradually increased to ensure that Seer does indeed place the highest priority on winning the game, see Fig. 3.6. Note that there is no reward for winning the game, as the episode ends after each goal (see Sec. 3.1.1).

3.6.1 Details

This subsection describes all the rewards from Table 3.5 in more detail.

Goal Scored This reward is given to the agent if it scores a goal. The agent receives a reward of 1 for scoring a goal and can receive an additional reward of up to 0.5 depending on the ball's speed when scoring the goal. The bonus is calculated using linear interpolation between the ball speed v_{ball} and the maximum ball speed $v_{ball_{max}}$.

$$R_{goal\_scored} = 1.0 + 0.5 * \frac{||v_{ball}||_2^2}{v_{ball_{max}}} \tag{3.1}$$

Boost Difference This reward is given to the agent if it collects or uses boost. The reward height is based on the difference of the boost amount between time step t and time step $t-1$. The boost amount is scaled to be in the range $[0, 1]$ and also scaled by the square root. In this way, the fact that collecting boost is more valuable if the agent does not have much boost is taken into account.

$$R_{boost\_difference} = f(boost^t) - f(boost^{t-1}), f(x) = \sqrt{\frac{x}{100}} \tag{3.2}$$

Ball Touch This reward is given to the agent if it touches the ball. The agent gets a reward of ≈ 1.0 for touching the ball on the ground and can get an additional reward of up to ≈ 1 depending on the ball height pos_{ball_z} during the touch scaled by the ball radius r_{ball}. To make dribbling without scoring disadvantageous, the reward is reduced if the agent touched the ball in the immediate past.

$$\begin{aligned} R_{ball\_touch} &= \lambda^t * \left(\frac{pos_{ball_z} + r_{ball}}{2 * r_{ball}} \right)^{0.2836} \\ \lambda^t &= \begin{cases} \max(0.1, \lambda^{t-1} * 0.95) & \text{touched ball} \\ \min(1.0, \lambda^{t-1} + 0.013) & \text{else} \end{cases} \end{aligned} \tag{3.3}$$

Demo A reward of 1 is given to the agent it demolishes the opponent.

Distance Player Ball This reward is given to the agent at every time step. The agent receives an exponentially distributed reward between 0 and 1 depending on the distance between the the player position pos_{car} and the ball position pos_{ball}. This reward is inspired by [19].

$$R_{distance\_player\_ball} = \exp\left(-0.5 * \frac{||pos_{car} - pos_{ball}||_2^2 - r_{ball}}{v_{car_{max}}}\right) \tag{3.4}$$

Distance Ball Goal This reward is given to the agent at every time step. The agent receives an exponentially distributed reward between 0 and 1 depending on the distance between the ball position pos_{ball} and the opponent's net pos_{net}. This reward is inspired by [19].

$$\begin{gathered} R_{distance\_ball\_goal} = \exp\left(-0.5 * \frac{||pos_{ball} - pos_{net}||_2^2 - c}{v_{ball_{max}}}\right) \\ c = pos_{net_y} - pos_{backwall_y} + r_{ball} \end{gathered} \tag{3.5}$$

Facing Ball This reward is given to the agent at every time step. The agent receives 1 reward if its front $car_{forward}$ points in the direction of the ball, and -1 reward if its front points in the opposite direction, in between the reward gets interpolated.

$$R_{facing\_ball} = car_{forward} \cdot \frac{pos_{ball} - pos_{car}}{||pos_{ball} - pos_{car}||_2^2} \tag{3.6}$$

Align Ball Goal This reward is given to the agent at every time step. The agent gets a high reward if it is in the right position when attacking or defending. When attacking, the agent gets a higher reward if it is behind the ball and the ball is between the car and the opponent's goal. When defending, the agent gets a high reward if it is positioned between the ball and its net.

$$\begin{aligned} R_{align\_ball\_goal} = {} & 0.5 * cosine\_similarity(pos_{ball} - pos_{car}, pos_{car} - pos_{net_{self}}) \\ & + 0.5 * cosine\_similarity(pos_{car} - pos_{ball}, pos_{net_{opponent}} - pos_{car}) \end{aligned} \tag{3.7}$$

Closest to Ball This reward is given to the agent at every time step. The agent receives a reward of 1 if it is closest to the ball, measured by L_2 distance.

Touched Last This reward is given to the agent at every time step. The agent receives a reward of 1 if the last touch of the ball was made by itself. This reward should represent the concept of *possession*.

Behind Ball This reward is given to the agent at every time step. The agent receives a reward of 1 if it is positioned behind the ball (between the ball and its net).

Velocity Player to Ball This reward is given to the agent at every time step. The agent receives a reward between -1 and 1 depending on whether its current velocity v_{car} is into the direction of the ball.

$$R_{velocity\_player\_ball} = \frac{v_{car}}{||v_{car}||_2^2} \cdot \frac{pos_{ball} - pos_{car}}{||pos_{ball} - pos_{car}||_2^2} \tag{3.8}$$

Kickoff This reward is the same as $R_{velocity\_player\_ball}$, but this reward is only applied during kickoff, as long as the ball stays in the center position. It should give strong incentive to learn a good kickoff.

$$R_{kickoff} = \begin{cases} R_{velocity\_player\_ball} & pos_{ball} = 0 \\ 0 & \text{else} \end{cases} \tag{3.9}$$

Velocity This reward is given to the agent at every time step. The agent receives a reward between 0 and 1 depending on its current velocity.

$$R_{velocity} = \frac{||v_{car}||_2^2}{v_{car_{max}}} \tag{3.10}$$

Boost Amount This reward is given to the agent at every time step. The agent receives a reward between 0 and 1, depending on how much boost it currently has.

$$R_{boost\_amount} = \sqrt{\frac{boost}{100}} \tag{3.11}$$

Forward Velocity This reward is given to the agent at every time step. This reward is similar to $R_{velocity}$, but this reward does only reward velocity if it is in the forward direction of the car and punishes backwards velocity. This reward was added because the agent sometimes learned first to drive backwards, which is a local optimum.

$$R_{forward\_velocity} = v_{car_{forward}} \cdot \frac{v_{car}}{v_{car_{max}}} \tag{3.12}$$

3.7 Reaction Time

The internal Rocket League physics engine runs at 120 ticks per second, which means that Seer could theoretically perform up to 120 actions per second. To use the computational resources more efficiently and possibly improve the performance of Seer, a *frameskip* [16] is used. Seer uses a frameskip of 8 frames, which means that Seer performs 15 actions per second and submits an action approximately every 66.6 ms. The action is then continuously applied during the next frameskip. It is not viable to apply the action for only one tick, since most actions must be applied continuously, e.g. steering, boosting. The action is submitted as soon as possible, which means that the game will receive the action performed by Seer on the next physics tick, see Fig. 3.7.

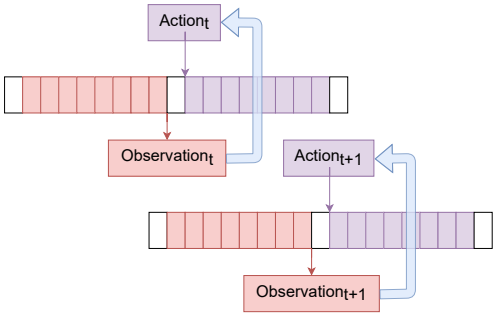


Figure 3.7: **Reaction Time:** Seer uses a frameskip of 8 and submits the action as soon as possible. This means that Seer performs 15 actions per second and has a reaction time ranging from approximately $8 - 75$ ms.

To summarize, Seer performs 15 actions per second and has a reaction time ranging from approximately $8 - 75$ ms, depending on when new information becomes available during the frameskip. For comparison, the average human reaction time is about 250 ms [14], while professional esport athletes can achieve an improved reaction time of up to about 150 ms [15]. Seer has a clear advantage when comparing reaction times to humans. It is difficult to compare the number of actions per second performed by Seer with those of a human, since Rocket League is mainly played with a controller that performs continuous actions. However, we believe that Seer has a small advantage when comparing actions per second to humans.

3.8 Long Term Credit Assignment

Although Rocket League is a very fast-paced game that requires many actions per second to play, decision-making is still critical. Thus, agents must carry out plans over many time steps to be successful. In training Seer, short-

term and long-term rewards are balanced by choosing the discount factor γ [31] as follows:

$$\gamma = \exp\left(\frac{\log(0.5)}{T * A}\right) \tag{3.13}$$

where $A = 15$ is the number of actions performed per second, and T is the time horizon (the time after which a reward is worth half).

The time horizon was initialized to $T = 10$ ($\gamma \approx 0.9954$) to facilitate learning an initial strategy. The time horizon was then continuously increased during training up to $T = 20$ ($\gamma \approx 0.9977$) to ensure that Seer performs proper long-term planning to win the game, see Fig 3.8.

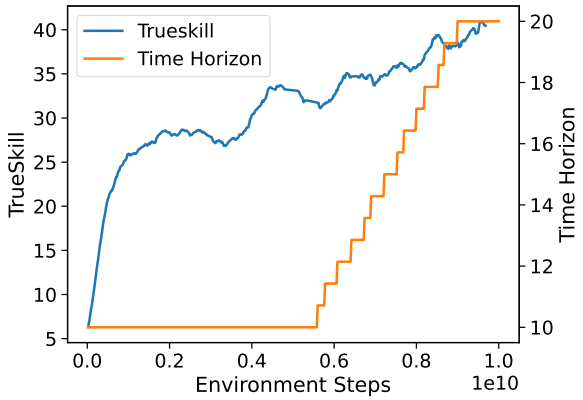


Figure 3.8: **Time Horizon during Training:** The time horizon was initialized to $T = 10$ ($\gamma \approx 0.9954$) to facilitate learning an initial strategy and continuously increased during training to $T = 20$ ($\gamma \approx 0.9977$) to ensure that Seer performs proper long-term planning to win the game.

3.9 Self-Play

Seer is trained through a self-improvement process named *self-play*. This technique has been successfully used in previous work to achieve superhuman performance in a variety of multiplayer games such as Backgammon [41], Go [34, 36], Chess [35], Shogi [35], Hex [2], StarCraft 2 [43], Poker [8] and Dota 2 [6]. In self-play training, the current best version of an agent continually plays against itself or older versions to optimize for new strategies that can defeat these past and present opponents.

In training Seer, 80% of the games are played against the latest set of parameters, and 20% play against past versions of Seer. Seer plays occasionally against past parameter versions in order to obtain more robust strategies and avoid strategy collapse, in which the agent forgets how to play against

a wide variety of opponents because it only requires a narrow set of strategies to defeat its immediate past version (see [5]).

To decide against which older version to play, only agents with TrueSkill $\mu > \mu_{new} - 10$ (see Sec. 4.1), where μ_{new} is the TrueSkill of the newest agent, are considered. This prevents Seer from playing older agents with a large a skill discrepancy. From the remaining agents, an agent is selected by sampling, weighted by the agent's TrueSkill. This strategy allows Seer to play against a wide variety of past opponents of appropriate skill.

3.10 Exploration

Exploration is a well-known and well-researched problem in the context of reinforcement learning. We encourage exploration in two different ways: by shaping the loss and by randomizing the training environment.

As described in [32], PPO uses entropy bonus to encourage exploration and prevent premature convergence to a suboptimal policy. In training Seer, the entropy coefficient was initialized to 0.01 and continuously lowered to 0.005 during training, see Fig. 3.9.

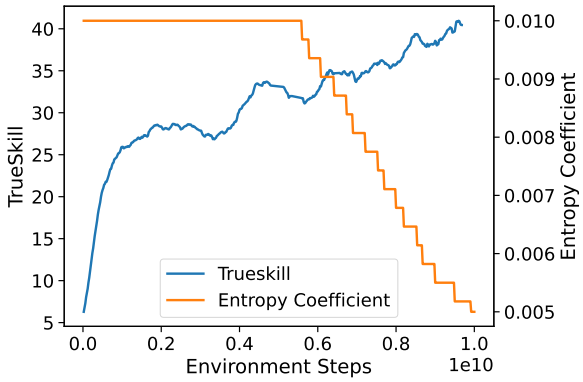


Figure 3.9: **Entropy Coefficient during Training:** The entropy coefficient was initialized to 0.01 at the beginning to favor exploration and continuously increased to 0.005 during training to favor more exploitation.

Further exploration is encouraged through randomization of the environment. The initial state is initialized according to the distribution shown in Table 3.6. Mostly, the state is reset to an actual state from a game. Additionally, some special state setters are used that focus on an important event in Rocket League: Kickoff Training, Wall Training, Goalkeeper Training and Air Training.

Table 3.6: **State Setters:** After each episode, the game state is randomized to encourage exploration. Mostly, the game is reset to a state from a human replay. Furthermore, some special state setters are used that aim to train a specific skill in Rocket League.

Name	Probability
Replay	0.7
Kickoff	0.1
Goalie Practice	0.05
Wall Practice	0.05
Air Practice	0.05
Random	0.05

3.10.1 Details

This subsection explains all the state setters from Table 3.6 in more detail.

Replay The game is reset to a state that is uniformly randomly selected from the Finetuning Dataset (see Sec. 3.12.1).

Kickoff The game is reset to one of the 5 kickoffs in Rocket League. Each kickoff is chosen with the same probability.

Goalie Practice The game is reset to a state in which the ball is flying towards one goal, therefore forcing one player to make a safe. This reset is intended to promote goalkeeper training.

Wall Practice The game is reset to a state in which the ball is rolling on the wall. This reset is intended to promote wall training.

Air Practice The game is reset to a state in which the ball is flying in the air and the cars are positioned on the ground. This reset is intended to promote aerial training.

Random The game is reset to a random state.

3.11 Hitbox

As described in Section 2.2.2, there are 6 different hitboxes in Rocket League, each with different physical properties. To account for all the different hitboxes, it is possible to add the specific hitbox to the player encoding as input to the neural network. However, as shown in [6], it is harder to learn a good

policy if the neural network has to consider multiple play styles. Therefore, we focused on only one hitbox, the *octane* hitbox. Seer is equipped with the octane hitbox, and it is enforced that every opponent Seer encounters during training also uses the octane hitbox. The octane hitbox was chosen because it is the most commonly used, and therefore more replays in which only octane hitboxes occurred could be found (see Sec. 3.12.1).

3.12 Learning a Prior from Human Data

As [6] and [3] show, developing a successful agent in an esport using reinforcement learning requires enormous computational resources and a lot of time, neither of which we had. Hoping to spend less time and resources training an agent, we used supervised learning to learn from human replays to give the network a prior.

Observations and actions were extracted from human replays to create two datasets (see Sec. 3.12.1): A *general* dataset of the best 15.81% players and a *finetuning* dataset of the best 0.05% players. The WebDataset [1] library was used to efficiently store and load the data. The Adam [18] optimizer was used to train the network for approximately 34 million steps using a batch size of 4 samples and an LSTM unroll length of 64. The first 30 million steps were performed using the general dataset using a learning rate of 1e-5, while the remaining 4 million gradient steps were performed using the finetuning dataset with a learning rate of 1e-6. To prevent the weights from diverging too far from zero, a weight decay of 1e-7 was used. The cross-entropy loss was applied to each action in the policy, weighted by the inverse of the frequency of the specific action. Furthermore, the MSE loss was applied to the value network, the final loss to be optimized was the mean of all losses.

3.12.1 Dataset

The *general* dataset consists of 820 958 replays from players with minimum rank Diamond 1 in 1v1 (see Sec. 2.2.1), that is, from the top 15.81% of players in 1v1. The replays were downloaded from ballchasing.com. Of the original 820 958 replays, those that contained errors, and those that contained hitboxes other than the *octane* hitbox (Seer's hitbox, see Sec. 3.11), were removed, as this would have added noise. Furthermore, the parts of the replay that are not part of actual gameplay (goal explosion, kickoff countdown) were also removed to reduce noise. As Seer can play the game only in the role of the blue player (see Sec. 3.2), each replay was mirrored to provide access to the orange player's data, effectively duplicating the training data. Finally, each replay was cut into multiple episodes (the environment is reset after each goal, see Sec. 3.1.1), which resulted in 12 646 104 episodes.

Rocket League replays are stored in a binary format, which has to be decompiled and analyzed to get the input and output features needed. The carball [9] library was used to decompile the binary replays into dataframes. The dataframes contain one row of physics information for each frame in the replay. The remaining features that make up an observation (see Sec. 3.2) were calculated by ourselves.

One of the more difficult features to impute was whether the car's wheels are touching the ground. Eventually, a histogram-based gradient-boosting classification tree (using scikit-learn [23]) was trained to predict whether the car is in contact with the ground. The training data was collected manually while driving around in Rocket League. This classifier also helped to compute some other features, e.g. *has flip*.

Unfortunately, Rocket League replays do not include all actions performed by players at a given time step. Of the 8 actions (see Table 2.1) needed to play Rocket League, only *throttle*, *steer*, *handbrake*, *jump* and *boost* are provided by replays. The *pitch*, *yaw* and *roll* actions were not included and had to be imputed. The carball library provided some rudimentary approximations for the missing actions. In addition to the carball approximations, we took advantage of the fact that all players use the same input for *yaw* and *steer* when playing the game, and keyboard players additionally use the same input for *throttle* and *pitch*.

Finetuning Dataset The *finetuning* dataset was created using the same methods as the general dataset (see Sec. 3.12.1). However, for the finetuning dataset, only replays of players with Supersonic Legend rank, the highest rank in the Rocket League (see Sec. 2.2.1), which corresponds to the top 0.05% of the player base in 1v1, were used. To further increase the quality of the training data, instead of keeping the blue and orange sides of the replay, only the side that scores the goal, i.e., wins the current episode, was kept. This resulted in 376 459 episodes.

Chapter 4

Evaluation

Evaluating the skill of an agent is no easy task. Many games are necessary to get an approximate estimate of an agent's abilities. Furthermore, there are many viable strategies that vary in effectiveness depending on the opponent. Since Psyonix does not allow bots to compete in online play, we had to evaluate Seer ourselves.

4.1 TrueSkill

We use the TrueSkill [11] rating system to evaluate Seer. TrueSkill is a Bayesian skill rating system which uses a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ to model the skill of a player. Whenever two players face each other, the mean and standard deviation of the players' skill are modified accordingly. Our TrueSkill environment uses the parameters $\mu = 25$, $\sigma = \mu/3$, $\beta = \sigma/2$, $\tau = \sigma/100$, $draw\_probability = 0$, which means that a TrueSkill difference of approximately 8.3 corresponds to an 80% winrate.

We first created a league of reference agents (in-game bots and custom bots) with known skills, see Table A.1. The TrueSkill of the reference agents was calculated by playing 281 games between the different reference agents. When calculating the TrueSkill for the reference agents, the matchup that resulted in the largest potential variance reduction was chosen.

To evaluate a new agent, the new agent's TrueSkill is initialized according to the environment and μ is initialized equal to the final μ of the previous agent. The first version of the agent was initialized with $\mu = 0$. The new agent then plays against the reference agents. A new agent is evaluated until a new version is released, which in our case means playing about 20 games to evaluate the agent's skill. As soon as a new version is released, the old agent is added to the reference agents, which means that the new agent is compared against all previous agents and the initial league of reference

agents. Once an agent is added to the reference agents, its TrueSkill remains fixed and only the skill of the agent being evaluated is changed. When evaluating new agents, the matchup with the highest probability of a draw, is chosen.

4.2 Random Prior vs. Human Prior

Using solely supervised learning did not result in an agent that could play Rocket League. The agent had mainly difficulty steering in the correct direction, which makes it impossible to play the game. However, the agent was able to execute some common actions reasonably well, such as kickoffs or dodging into the ball if the ball is directly in front of the agent.

While using supervised learning from replays did not result in an agent that can play Rocket League, we observed faster learning of a good policy using reinforcement learning when using a human prior than with a random prior, see Fig. 4.1. Starting training with a human prior reaches almost immediately a TrueSkill of about 35, while starting training with a random prior starts with TrueSkill around zero. Note, however, that if the agent starts training with a human prior, the agent's TrueSkill will drop off relatively soon after launch and eventually matches the agent's training progress if the agent starts training with a random prior.

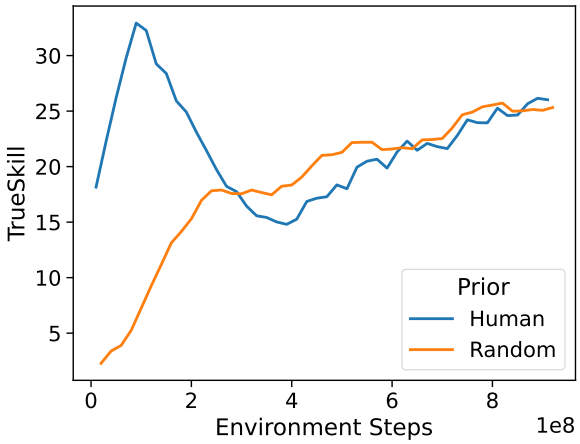


Figure 4.1: **TrueSkill using Different Priors:** We trained two versions of Seer, one version started learning with a random prior and one version started learning with a human prior from human replays. The version that started learning with a human prior shows faster learning in the early stages, but eventually the skills of the two agents converge.

We observed that the initial policy using a human prior has a relatively

low jumping probability, which allows the agent to learn quickly in early training. However, not using the jump button represents a local optimum, and once the agent began to increase the probability of jumping to explore, the agent's skill decreased. We believe that the rewards and losses we have defined favor a certain way of learning how to play Rocket League, and the policy is forced into that way of learning eventually. It may be possible to choose rewards and losses such that the agent's performance does not decrease when using a human prior, we have not had the time and resourced to investigate this phenomenon further.

4.3 Evaluating Seer

Fig. 4.2 shows the TrueSkill of Seer during training. The highest TrueSkill Seer achieved is about 40, which is better than most agents in the reference league and slightly below the best agent in the reference league. Seer is roughly in the Platinum I rank (see Sec. 2.2) and thus in the top 50 percent of the Rocket League player base in 1v1. Seer's play style is mostly focused on ground play, as Seer failed to learn to aerial or to double jump. Therefore, Seer focuses primarily on making outplays by dribbling and by rolling the ball on the wall. Seer also learned to perform a good kickoff, which is essential for the 1v1 game mode. We created a YouTube playlist[1] which documents the training progress and play style of Seer.

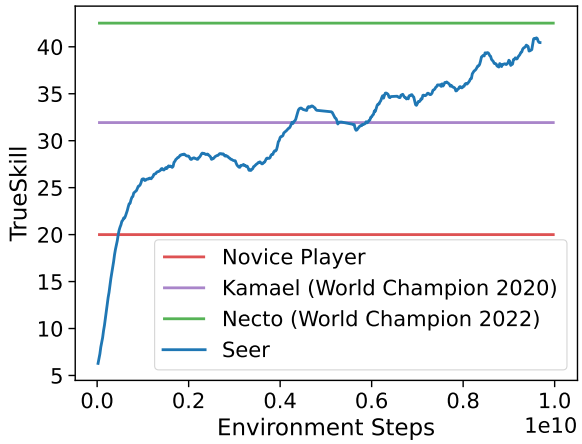


Figure 4.2: **TrueSkill of Seer:** Seer trained for about one and a half month and played about 20 years of Rocket, during which Seer was constantly evaluated. The final version of Seer is roughly in the Platinum I rank and consistently performed better than most agents in the reference league.

[1] https://www.youtube.com/playlist?list=PLsg-iAOpzEaVIV5CdO9R3hbNv6wTOyFVF

We make no claim that Seer is the best performing Rocket League agent, as Seer failed to learn some of the most essential mechanics of Rocket League. However, Seer was able to achieve a high level of play without scripted actions, which sets him apart from most other Rocket League agents, as most agents use scripted kickoffs.

Chapter 5

Conclusion

In this thesis, we created Seer, a Rocket League 1v1 agent. Seer was trained using self-play, where the policy was optimized using Proximal Policy Optimization (PPO). With the goal to accelerate training, we used supervised learning to learn a prior from human replays. We showed that learning from human replays can speed up training in the early stages, but shows diminishing returns in the later stages. The final version of Seer is roughly in the Platinum I rank, and thus in the top 50 percent of the Rocket League player base in 1v1.

5.1 Future Work

The key ingredient to further improve the performance of Seer is to expand the scale of compute used, by increasing the batch size and total training time. It is critical to make use of distributed computing using asynchronous rollout and optimization workers to scale to larger batch sizes. It may also be worthwhile to further explore the approach in which Seer uses a prior from human data.

As it takes a long time to train an agent using reinforcement learning, we make no claim that the architecture, observation space, action space, rewards, losses, hyperparameters etc. used to train Seer are optimal. We believe that there is certainly room for improvement.

This thesis focused only on the 1v1 game mode in Rocket League. Although Seer cannot play other game modes, it is possible to generalize this work to other game modes. Seer currently posses superhuman abilities (e.g. reaction time), a next version could be more handicapped to match human abilities.

Appendix A

Appendix

A.1 Game Tick Packet

```
packet: {
    'game_cars': [
        {
            'physics': {
                'location': {'x': 0.0, 'y': 0.0, 'z': 0.0},
                'rotation': {'pitch': 0.0, 'yaw': 0.0, 'roll': 0.0},
                'velocity': {'x': 0.0, 'y': 0.0, 'z': 0.0},
                'angular_velocity': {'x': 0.0, 'y': 0.0, 'z': 0.0}
            },
            'is_demolished': False,
            'has_wheel_contact': True,
            'is_super_sonic': False,
            'is_bot': True,
            'jumped': False,
            'double_jumped': True,
            'name': 'Jupiter',
            'team': 0,
            'boost': 48.0,
            'hitbox': {'length': 118, 'width': 84, 'height': 36},
            'hitbox_offset': {'x': 13.88, 'y': 0.0, 'z': 20.75},
            'score_info': {
                'score': 340,
                'goals': 2,
                'own_goals': 0,
                'assists': 1,
                'saves': 1,
                'shots': 3,
                'demolitions': 1
            }
        },
        { ... }
    ],
    'num_cars': 2,
    'game_boosts': [
        {
            'is_active': True,
            'timer': 0.0
        },
        { ... }
```

```
    ],
    'num_boost': 36,
    'game_ball': {
        'physics': {
            'location': {'x': 0.0, 'y': 0.0, 'z': 0.0},
            'rotation': {'pitch': 0.0, 'yaw': 0.0, 'roll': 0.0},
            'velocity': {'x': 0.0, 'y': 0.0, 'z': 0.0},
            'angular_velocity': {'x': 0.0, 'y': 0.0, 'z': 0.0}
        },
        'latest_touch': {
            'player_name': 'Beavis',
            'time_seconds': 120.63,
            'hit_location': {'x': 0.0, 'y': 0.0, 'z': 0.0},
            'hit_normal': {'x': 0.0, 'y': 0.0, 'z': 0.0},
            'team': 0,
            'player_index': 0
        },
        'drop_shot_info': {
            'damage_index': 0,
            'absorbed_force': 0,
            'force_accum_recent': 0
        },
        'collision_shape': {
            'type': 1,
            'box': {'length': 153.0, 'width': 153.0, 'height': 153.0},
            'sphere': {'diameter': 184.0},
            'cylinder': {'diameter': 184.0, 'height': 30.0}
        }
    },
    'game_info': {
        'seconds_elapsed': 405.12,
        'game_time_remaining': 34.0,
        'is_overtime': False,
        'is_unlimited_time': False,
        'is_round_active': True,
        'is_kickoff_pause': False,
        'is_match_ended': False,
        'world_gravity_z': -650.0,
        'game_speed': 1.0
    },
    'teams': [
        {
            'team_index': 0,
            'score': 7
        },
        { ... }
    ],
    'num_teams': 2,
}
```

A.2 Reference League

We used version 34 of the RLBotPack [28].

Table A.1: **Reference League:** TrueSkill of custom bots.

Bot	TrueSkill
Necto	42.52 ± 3.29
Self-driving car	40.69 ± 2.20
Wildfire	36.44 ± 2.22
Diablo	35.77 ± 2.17
Botimus Prime	34.89 ± 2.24
Bubo	33.98 ± 2.18
Beast from the East	32.35 ± 2.26
ReliefBot	32.18 ± 2.27
Kamael	31.93 ± 2.22
rashBot	30.34 ± 2.30
AdversityBot	30.29 ± 2.29
PenguinBot	29.77 ± 2.26
Bribblebot	29.30 ± 2.21
Stick	28.14 ± 2.24
BroccoliBot	27.33 ± 2.24
FormularBot 1.5	27.09 ± 2.20
Atlas	26.97 ± 2.25
Leaf	25.68 ± 2.26
Zoomelette	25.45 ± 2.30
Allstar	24.29 ± 2.31
ABot	24.27 ± 2.21
DisasterBot	23.68 ± 2.25
Pro	22.94 ± 2.22
ElkBot	22.35 ± 2.29
Lanfear	21.79 ± 2.27
Air Bud	20.67 ± 2.32
SkyBot	20.60 ± 2.28
FillamentBot	20.39 ± 2.29
St. Peter	19.74 ± 2.28
Codename Cryo	18.19 ± 2.28
NomBot_v1.0	17.30 ± 2.30
VirxEB	16.98 ± 2.26
Rookie	16.33 ± 2.31
RocketNoodles	11.88 ± 2.31

A.3 Software Stack

- Python Version: 3.8.10
- OS: Windows 10
- CUDA: 11.3

A.3.1 Forks

- rlgym: https://github.com/Walon1998/rocket-league-gym (e1f466c1c409f934d41ae029d1f9d6eb2e877a71)
- rlgym-tools: https://github.com/Walon1998/rlgym-tools (1f93d7f3b04be00394007c74d3a40c28ba1fe842)
- sb3-contrib: https://github.com/Walon1998/stable-baselines3-contrib/tree/feat/ppo-lstm (314e7b71837bc2d9ead3d06d9768c6484ff76d20)
- stable-baselines3: https://github.com/Walon1998/stable-baselines3 (803b63eabf93768d542ee778a5eb85b68358b7f3)

A.3.2 Installed Packages

- carball: 0.7.5
- comet-ml: 3.26.0
- gym: 0.21.0
- matplotlib: 3.5.1
- numba: 0.55.1
- numpy: 1.21.6
- nvidia-ml-py3: 7.352.0
- pandas: 1.0.3
- pycuda: 2020.1
- rlbot: 1.63.2
- rlbot-gui: 0.0.126
- rlgym-compat: 1.0.2
- RLUtilities: 0.0.13
- scikit-learn: 1.0.1
- scipy: 1.8.0

- seaborn: 0.11.2
- tensorrt: 8.2.3.0
- torch: 1.10.2+cu113
- torch-tensorrt: 0.0.0
- torchaudio: 0.10.2+cu113
- torchvision: 0.11.3+cu113
- torchviz: 0.0.2
- trueskill: 0.4.5
- webdataset: 0.2.4

A.4 Hyperparamters Change

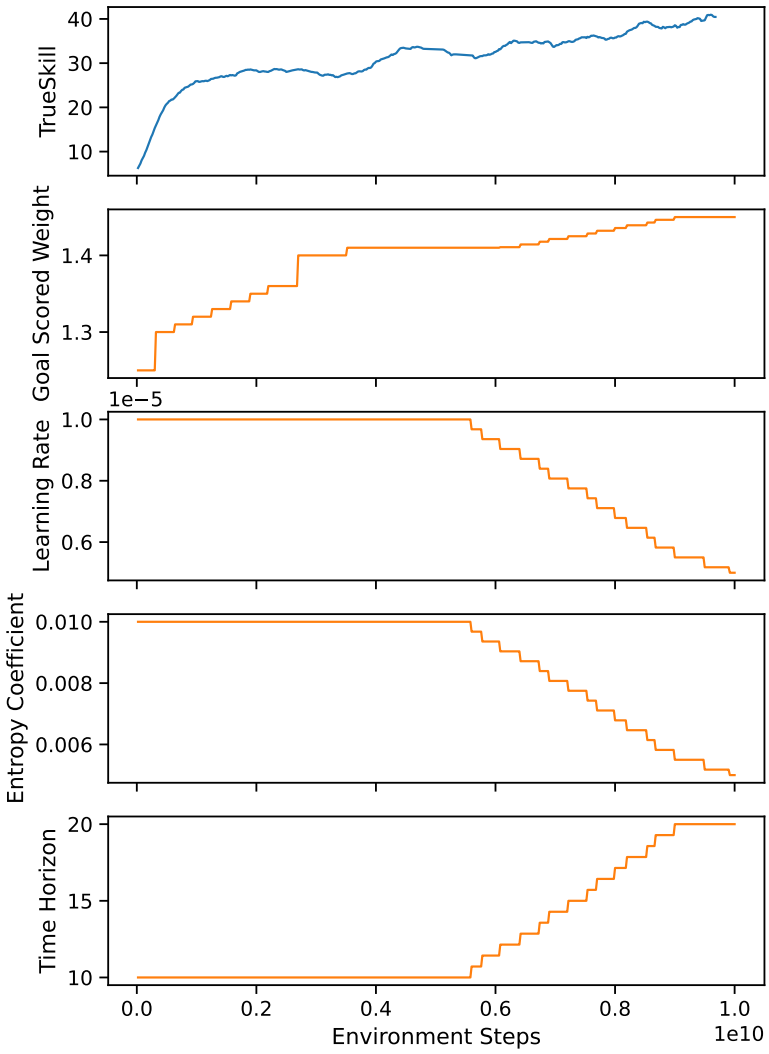


Figure A.1: **Hyperparameters Change during Training:** This figure shows the hyperparameters that were changed during training and how they were adjusted.

A.5 Rocket League 1v1 Rank Distribution

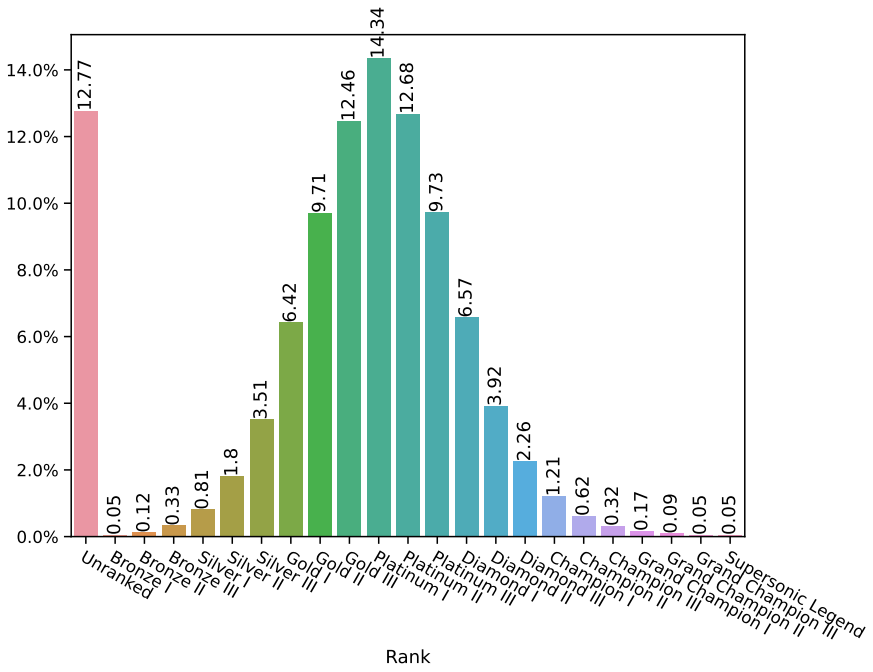


Figure A.2: Approximate rank distribution in 1v1 in Rocket League (Season 7).

Bibliography

[1] Alex Aizman, Gavin Maltby, and Thomas Breuel. High performance i/o for large scale deep learning, 2020.

[2] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *CoRR*, abs/1705.08439, 2017.

[3] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective, 2019. cite arxiv:1902.01724.

[4] Bakkesmod. https://bakkesmod.com/index.php. Accessed: 10.05.2022.

[5] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech M. Czarnecki, Julien Pérolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. *CoRR*, abs/1901.08106, 2019.

[6] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[7] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3(null):213–231, mar 2003.

[8] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365:eaay2400, 07 2019.

[9] Carball. https://github.com/SaltieRL/carball. Accessed: 10.05.2022.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[11] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill(tm): A bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, pages 569–576. MIT Press, January 2007.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[13] Shengyi Huang and Santiago Ontañ ón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, may 2022.

[14] Aditya Jain, Ramta Bansal, Avnish Kumar, and Kd Singh. A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students. *International Journal of Applied and Basic Medical Research*, 5:124 – 127, 2015.

[15] Aditya Jain, Ramta Bansal, Avnish Kumar, and Kd Singh. A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students. *International Journal of Applied and Basic Medical Research*, 5:124 – 127, 2015.

[16] Shivaram Kalyanakrishnan, Siddharth Aravindan, Vishwajeet Bagdawat, Varun Bhatt, Harshith Goka, Archit Gupta, Kalpesh Krishna, and Vihari Piratla. An analysis of frame-skipping in reinforcement learning. *CoRR*, abs/2102.03718, 2021.

[17] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. *CoRR*, abs/2004.00980, 2020.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[19] Siqi Liu, Guy Lever, Zhe Wang, Josh Merel, S. M. Ali Eslami, Daniel Hennes, Wojciech M. Czarnecki, Yuval Tassa, Shayegan Omidshafiei, Abbas Abdolmaleki, Noah Y. Siegel, Leonard Hasenclever, Luke Marris, Saran Tunyasuvunakool, H. Francis Song, Markus Wulfmeier, Paul Muller, Tuomas Haarnoja, Brendan D. Tracey, Karl Tuyls, Thore Graepel, and Nicolas Heess. From motor control to team play in simulated humanoid football, 2021.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Marco Pleines, Konstantin Ramthun, Yannik Wegener, Hendrik Meyer, Matthias Pallasch, Sebastian Prior, Jannik Drögemüller, Leon Büttinghaus, Thilo Röthemeyer, Alexander Kaschwig, Oliver Chmurzynski, Frederik Rohkrähmer, Roman Kalkreuth, Frank Zimmer, and Mike Preuss. On the verge of solving rocket league using deep reinforcement learning and sim-to-sim transfer, 2022.

[25] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[26] What are rocket league competitive ranks? https://support.rocketleague.com/hc/en-us/articles/360054049854-What-Are-Rocket-League-Competitive-Ranks-. Accessed: 09.05.2022.

[27] Rlbot. https://rlbot.org/. Accessed: 10.05.2022.

[28] Rlbotpack. https://github.com/RLBot/RLBotPack. Accessed: 10.05.2022.

[29] Rlgym. https://rlgym.org/. Accessed: 10.05.2022.

[30] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[31] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

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

[33] Rocket Science. Rocket league hitbox visualizations (all cars) [1.82]. https://www.youtube.com/watch?v=99j1mTN1_Vs. Accessed: 10.05.2022.

[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[36] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[37] Richard Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 08 1988.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[39] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[40] R.S. Sutton, R.S.S.A.G. Barto, A.G. Barto, C.D.A.L.L.A.G. Barto, F. Bach, and MIT Press. *Reinforcement Learning: An Introduction*. A Bradford book. MIT Press, 1998.

[41] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[42] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[43] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019.

[44] Starbase arc and wasteland redesigned as standard arenas. https://www.rocketleague.com/news/starbase-arc-wasteland-standard-arenas/. Accessed: 09.05.2022.

[45] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[46] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[47] Ronald Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2, 09 1998.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Seer: Reinforcement Learning in Rocket League

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Walo	Neville

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
26.04.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.