Driving Olympics with Reinforcement Learning

James Herman **Abhinav Gupta** **Bohui Fang** **Ignacio Maronna**

Xinnan Du **Jikai Lu** **Zihang Zhang** **Anirudh Koul**

Jonathan Francis **Siddha Ganju**

Eric Nyberg

Abstract

Our research lies at the intersection of autonomous driving, particularly autonomous racing, and reinforcement learning, a machine learning paradigm where agents seek to maximize their reward in an environment. Agents are incentivized with rewards which define what the agent *should do*, and it is their responsibility to discovering behaviors which result in maximum cumulative reward. As such, reinforcement learning differs from supervised and unsupervised learning since the agent collects its own data through environment exploration. In many ways autonomous racing is a simpler domain than street-legal driving, but is well suited for reinforcement learning research. We began with AWS Deepracer, a toy example with both online and real racing competitions. After early success in this domain, we became the first North American team to join Roborace, an autonomous racing series using full-size electric race cars. While AWS provided their own reinforcement learning environment and training infrastructure, we built our own around Roborace's driving simulator, which is much more realistic than AWS Deepracer's. Our environment is OpenAI gym compliant making it compatible with many public reinforcement learning frameworks. In numerous baseline experiments, we prove that a agent can learn to race without any prior knowledge of driving or racing in this sophisticated environment.

1 Introduction

Reinforcement learning is a learning paradigm where agents explore and exploit the environments they are in to maximize their cumulative reward. Rather than instructing the agent how to receive the reward, reinforcement learning agents must interact with their environment and engage in behavior discovery. This lack of explicit instruction has lead to incredible artificial intelligence achievements in robotics [OpenAI et al., 2018] [OpenAI et al., 2019] and game playing [Bansal et al., 2017] [Silver et al., 2017] [Mnih et al., 2013b].

Given the domains that reinforcement learning has shown success in, we have decided to explore its potential in autonomous racing which lies both in the field of robotics and competitive game play. This has not been done at great depth, and we are far from an agent achieving superhuman performance in a real race car despite success in simplistic virtual racing environments [Fuchs et al., 2020b] [Schwarting et al., 2020].

First, however, we began competing AWS DeepRacer competitions. Compared with similar racing contests, such as ones proposed by Nvidia, DeepRacer is a simplistic environment designed to engage users that are unfamiliar with autonomous driving or reinforcement learning. DeepRacer is a fully managed pipeline with most all of the training infrastructure abstracted from users. This leaves

little room for modification including the choice of learning algorithm which is Proximal Policy Optimization [Schulman et al., 2017]. The AWS DeepRacer environment provides many features to the agent such as speed, position, distance from the center lines, and other information relevant to the agent. In this simplified setting, we designed a complicated reward function, considering many information mentioned above. After several iterations, we achieve top-10% of a monthly scoreboard, inspiring us to further our work in other domains.

Roborace was our next autonomous racing domain. We constructed a fully functional reinforcement learning pipeline around the simulator they provided, incorporating many domains of data science including systems, machine learning, and human-centered data science. Unlike many public reinforcement learning environments which are created specifically for reinforcement learning research, ours is built around a simulator designed for autonomous racing development. In teaching a car how to race itself, we engineered visual features for the agent, applied state-of-the-art reinforcement learning algorithms, monitored training with custom analytical tools, and containerized our environment for automated deployment and distributed training.

2 Related Work

2.1 Roborace & Virtual Racing Simulators

Roborace is one of the first global championships for full-size autonomous race cars, and our capstone team was the first U.S. team to join the series. Roborace builds and owns the vehicles, software-in-the-loop (SIL) and hardware-in-the-loop (HIL) simulators, and a base stack to drive the race car autonomously. Teams are responsible for improving the driving capabilities of the vehicle to navigate increasingly difficult autonomous racing challenges with the hope of wheel-to-wheel competitive reason at the end of Season Beta. Currently, Roborace teams overwhelmingly use classical controls in their vehicles rather than approaching it as an artificial intelligence problem.

CARLA is an open-source simulator for autonomous driving research used primarily for urban environments and street-legal driving. Roborace provides teams with a custom, CARLA-based [Dosovitskiy et al., 2017] SIL simulator which has custom maps of a variety of real-life racetracks. This simulator was built to run on Linux desktops and with an emphasis on accurate physics and vehicle dynamics rather than scalability or compatibility with machine learning applications.

DeepRacer [Balaji et al., 2019] is a platform developed by AWS that provides an end-to-end framework for deploying reinforcement learning algorithms to learn an agent that can autonomously drive a 1/18th scale race car. The platform simplifies the RL algorithms' experimentation process by decoupling the policy update from the model rollouts, which significantly helps in scaling the experiments. The creators have also demonstrated that the learned policy from the virtual simulator can be easily transferred to a real-world autonomous race car without expert relabeling, real-world data, or expensive preprocessing. This is largely due to the relative simplicity of the environment.

TORCS, The Open Racing Car Simulator, is another example of an open-source car racing simulator that provides realistic physics engine and accurate car dynamics. Simulated Car Racing Championship [Loiacono et al., 2013] builds on the TORCS for an international competition where the goal is to develop autonomous racing agents.

2.2 Reinforcement Learning & Autonomous Racing

A recent and promising example of success in autonomous racing using reinforcement learning was achieved in the video game Gran Turismo Sport [Fuchs et al., 2020a]. Researchers built a custom reinforcement learning environment and trained an agent to play the video game at super-human performance, besting the times 50,000 human players in a solo race. This success, however, relied on unrealistic information which told the agent the exact distance to the edge of the track at various angles rather than using data from a camera, LiDAR, or radar sensor like real autonomous vehicles would use. The learned trajectory was described by human experts as being overly risky and only possible due to unrealistic precision.

The Gran-Turismo Sport agent involved only one vehicle on the track at once, however. In contrast, the work presented by [Schwarting et al.], the authors can learn policy to compete with two-player settings with a model-based approach. This work's primary contribution is the ability to learn a

Multi-agent RL policy with just raw-image observations. The model can do so by learning a world model from the ground-truth examples and predicting the opponent's actions in the latent space. Further, it optimizes the agent's behavior by imagined-self play instead of executing the actions in the real world.

One of the earlier successful works of applying Deep RL on Autonomous driving is [Wang et al., 2018]. The authors trained an RL agent in the TORCS simulator that provides various sensor inputs instead of raw images. The authors had to carefully select a set of sensor inputs and design a reward function for the RL algorithm to work. As the agent's actions are continuous, like steering angle, acceleration, and braking, policy-based RL approaches are more suitable than value-based strategies. Therefore, the authors have used DDPG. In a similar work [Ganesh et al., 2016], authors have employed DQN for training an RL agent on the TORCS simulator.

2.3 Reinforcement Learning Algorithms

Recent approaches have succeeded in learning control policies from high-dimensional sensory data. One particular example is an RL agent successfully learning to play Atari games given just the visual inputs [Mnih et al., 2013a]. This model was trained with Deep Q-learning (DQN) which is a model-free, off-policy algorithm. And it inspires us to learn race car's control inputs solely based on the camera feed we receive from the simulator.

Proximal Policy Optimization [Schulman et al., 2017], used by AWS DeepRacer, which builds upon the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) [Schulman et al., 2015], but it does so with only first-order optimizations. Traditional policy gradient approaches perform one gradient update per data sample while PPO can perform minibatch updates. One of the drawbacks of TRPO is that KL divergence in TRPO's objective function is an added constraint and complexity while making the training unstable. On the other hand, PPO forces the policy updates to be conservative in case they diverge too far away from the current policy. Even though PPO is data-efficient, on-policy RL methods typically need new data samples for each gradient step. Furthermore, as the task complexity increases, more gradient steps and samples per each step are needed making these prohibitively expensive. Off-policy methods that use prior experience, such as Deep Deterministic Policy Gradients (DDPG) [Li et al., 2019], can suffer from brittle convergence and hyperparameter sensitivity for problems with continuous state and action space. [Haarnoja et al., 2018]

Soft Actor-Critic is an off-policy method that maximizes entropy to provide greater sample efficiency than on-policy methods while also providing stable convergence properties. SAC has shown to provide good results on very complex, high-dimensional tasks, and therefore it's the algorithm of our choice for this work.

2.4 Asynchronous RL Framework

Asynchronous Gradient Descent for optimization of Deep RL networks has shown promising results in [Mnih et al., 2016]. This work trains several RL algorithms, on-policy, off-policy, value-based and policy-based methods, asynchronously on multiple CPU-threads of a single machine. Training on different parts of the environment parallelly makes all the updates to the parameter-server less correlated to each other compared to a single agent's updates. This parallelism improves the training stability and reduces the training time. On a similar approach, we, too, train our model on a distributed architecture on AWS.

3 Methodology

In this section, we will introduce the methods how we set up the environment for Reinforcement Learning, process the data, and train state-of-the-art models.

3.1 Environment Setting

3.1.1 Racing Simulator & Customized Reinforcement Learning Environment

Arrival Simulator Arrival, a commercial Roborace team, provides other team will access to their racing simulator which is CARLA-based. The simulator includes numerous different race tracks modeled off of their real-life counterparts. Virtual sensors, including IMU, camera, and LiDAR, are used in the simulator to provide the vehicle access to perception features.

To perform reinforcement learning, we needed to send actions, such as steering and acceleration requests, to the environment and then receive observations and rewards from the environment. To do so, we built UDP interfaces to send actions to the environment and used a combination of TCP and UDP interfaces to receive sensor and other observation data from the simulator. Another component to learning is automation, and we did so by building a Websocket interface that communicates with the simulator's API and integrating this into our environment.

Custom Reinforcement Learning Environment Because we did not have access to the source code of the simulator, we had to build our custom reinforcement learning environment from nearly scratch. Using the interfaces described above, we created our environment by discretizing our environment into steps involving an action being taken followed by an observation being received. We modeled the structure of this environment after OpenAI's Gym toolkit [Brockman et al., 2016] primarily to become compatible with a variety of reinforcement learning frameworks. To accomplish this, we implement several common functions of an environment, `make()`, `reset()`, and `step()` by encapsulating the primitive simulator. Then we have well-defined state space, observation space, and action space. Its structure is shown in Figure 1.

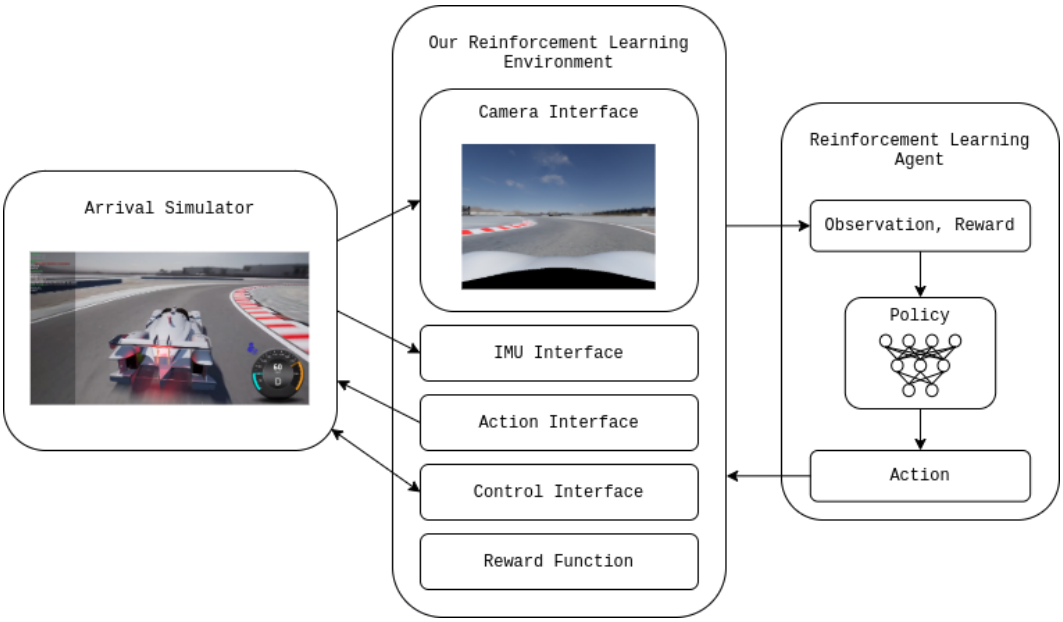

Figure 1: Structure of our Reinforcement Learning Environment.

3.1.2 Coordinate System & Episode Termination

Global Coordinate System (GCS) allows every location on the earth to be described in terms of (x, y, z) coordinates. The (x, y) coordinate are w.r.t to a location defined off the coast of Western Africa, and the z dimension is w.r.t to the center of the earth. For our use case, it's challenging to work with GCS. We are only concerned about a small geographical region limited to the racetrack, and GNU coordinates differ at the 4th decimal place for the entire racetrack.

An alternative to the Global Coordinate System is the East, North, Up (ENU) Local Coordinate system, Figure 2. The ENU system is more suited for tracking coordinates in a small geographical

region where we can approximate the earth's curvature to a flat surface tangential to the earth. Local ENU system is far more intuitive than GCS, and distances can be easily expressed as a euclidean distance in meters. Therefore, racetrack coordinates are represented in the ENU system where the reference point is an arbitrary point close to the racetrack.

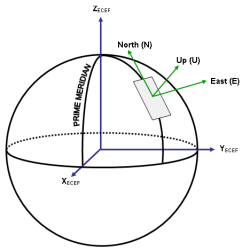

Figure 2: ENU coordinate system with respect to the GCS or Earth-Centered Earth-Fixed (ECEF) system

The data that we receive from the simulator is described in the previous section in the Table 2. The simulator tracks the car's location in the GCS, and we convert that to the ENU system. This helps us in estimating the reward, imitating expert trajectories, or detect racetrack excursions.

While training our RL agent, we want to prevent training on episodes that are suboptimal or incorrect. Therefore we terminate the episode if the car is taking too long to complete one loop, get stuck, or go off the track. We detect if the car has gone off the track by first plotting the track's outer and inner edges based on the ENU coordinates. If any of the wheels go outside the outer edge or inside the inner edge, we terminate the episode.

3.1.3 Basic Feature Processing

With well-defined observation space, we can obtain 30 dimensions of raw data from the simulator, as listed in Table 1. Although the information is truly informative, the underlying pattern may be hard to capture by a neural network model. Thus we process the data every time when we observe.

Dimension	Data
0,1,2	Steering, Gear, Mode
3,4,5	Velocity
6,7,8	Acceleration
9,10,11	Angular Velocity
12,13,14	Yaw, Pitch, Roll
15,16,17	Location Coordinates (x, y, z)
18,19,20,21	Rpm (per wheel)
22,23,24,25	Brake (per wheel)
26,27,28,29	Torq (per wheel)

Table 1: Original Input Format.

We process the data in a way such that we can significantly reduce the total dimension. We list our processed data format as in Table 2. From the perspective of information, the newly calculated feature is definitely less informative than before. However, we discard some rarely used variables (almost unchanged height of a car) and average some information among 4 wheels.

By this operation, we reduce almost 2/3 of the data dimension, while we keep enough data to analyze and do basic driving. Still, it is noteworthy that if we need the car to learn a much higher level of driving skill, we may consider a better method to process the discarded data.

3.1.4 Visual Features - VAE

We encode image data captured by onboard camera with variational autoencoder proposed in [Kingma and Welling, 2014]. As shown in Figure 3, the model encodes the given input image x into latent

Dimension	Data
0,1,2	Steering, Gear, Mode
3	Normalized Velocity
4	Normalized Acceleration
5	Normalized Angular Velocity
6	Yaw
7,8	Location Coordinates without z
9	Average Rpm (per wheel)
10	Average Brake (per wheel)
11	Average Torq (per wheel)

Table 2: Processed Input Format.

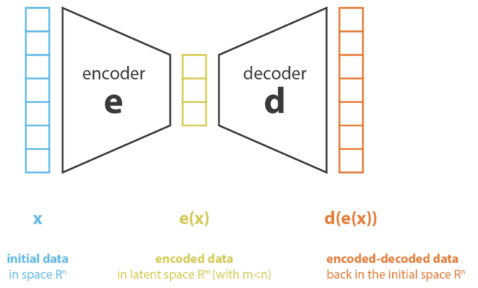

Figure 3: Variational Autoencoder Architecture

vector e(x) of reduced dimension, then it decodes the hidden vector to reconstruct image d(e(x)). The main purpose of the model is to find the best encoder-decoder pair to maximize the embedded information and minimize the reconstruction error after decoding, that is

$$(e*, d*) = argminL(x, d(e(x)))$$

In our project, we use Euclidean loss and latent vector of 16 dimensions to generate the visual feature vector. We first deploy our VAE on artificial projected images of road boundaries. Compared with the left input image shown in left half in Figure 4, the right reconstructed image effective learns the road boundaries, which demonstrates the few information loss in the latent vector. To transfer the model to the much more complex real racing environment, we crop the top part of the image to reduce distraction from irrelevant information like sky or sunlight so that to make neurons focus on learning the track shape and boundary. The final result is shown in right half of Figure 4.
The 16-dimension latent vector is used as visual feature input in our reinforcement learning agent.

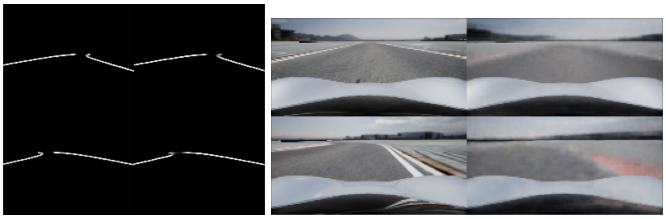

Figure 4: Image Reconstruction

3.1.5 Reward Function

We have designed an utterly complicated reward function back in our journey for AWS DeepRacer Contest which rewarded our agent in many ways such as being near the center of the track. We learned, however, that more desirable behaviour emerges even with simplistic incentives. Our goal is an agent that can complete laps in minimal time. Because of the sparsity of lap completion, we instead reward progression down the track.

Thus we designed a reward function as

$$Reward = C_{center} * (r_{progress} + r_{speed}) - P_{oob}$$

, where

$$r_{progress} = index_{current} - index_{last\_time} - P_{no\_moving}$$

and

$$r_{speed} = C_{speed} * speed$$

. The coefficients C and the penalties P are all positive, and *oob* stands for "out-of-bounds". This means we reward two things: high speed and more track finishing. In the meantime, we will significantly penalize the "out-of-bound" and slightly penalize the "not moving" state. By this definition, we know that a policy which can drive further through tracks with a higher speed can achieve the highest reward. It is consistent with our goal, and smooth enough to start with a randomly initialized policy. Specially, the P_{no_moving} is necessary to add. Otherwise, a "smart" learning progress can converge to a *seemed* good policy where it carefully uses a slowest speed (possibly 0) to stay on the track as long as possible to avoid an easy P_{oob}.

3.2 Training Framework - SAC

We did research on different reinforce learning models, and decided to use Soft Actor-Critic (SAC) as our main model [Haarnoja et al., 2018]. In our autonomous driving scenario, SAC gives a more stable training results compared with other algorithms such as DDPG, A3C.

SAC incorporates ideas from both soft Q-learning and TD3. The most important feature of SAC compared with other RL algorithms is entropy regularization. It tries to maximize a trade-off between entropy and expected return, which is similar to the exploration-exploitation trade-off.

The key equation for SAC is:

$$J(\pi) = \mathbb{E}_{\pi}\left[\sum_{t} r\left(\mathbf{s}_t, \mathbf{a}_t\right) - \alpha \log\left(\pi\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right)\right]$$

To maximize this objective function, we need to consider not only the expected return but also the entropy term.

There are a lot of resources online with off-the-shelf models for us to use. We selected a library called [Stable Baselines], which includes a set of improved implementations of RL algorithms based on OpenAI baselines. We have successfully trained our car to complete the race without rushing out of the track using our SAC model.

3.3 Distributed Training

In terms of training infrastructure, we wanted to be able to run multiple experiments in a distributed fashion so that the agent could drive for a million miles and best learn how to find its way. However, we needed for the project to be portable. Even though we used AWS for our experiments, we wanted to be able to run them on different Cloud providers, locally, or even on private clusters.

Therefore, the simulator and trainer were dockerized. This allowed for them to be run on any hardware, and enabled the use of kubernetes for their deployment. We implemented two modes of deployment: multiple parameters-single environment, and single parameter-multiple environments.

3.3.1 Multiple Parameters - Single Environment

In this configuration, each set of parameters has its own independent simulator. The orchestrator - in this case an EC2 instance (could be any properly configured virtual machine or local computer), launches one training pod for each parameter. Each pod has its own independent trainer and simulator, where simulations are run and logged. The docker images are pulled from AWS ECR, but again, could be any docker registry.

The advantage of this configuration is that it allows to run multiple experiments for different parameters configurations. However, the speed at which they can run is capped by the speed of a single simulator.

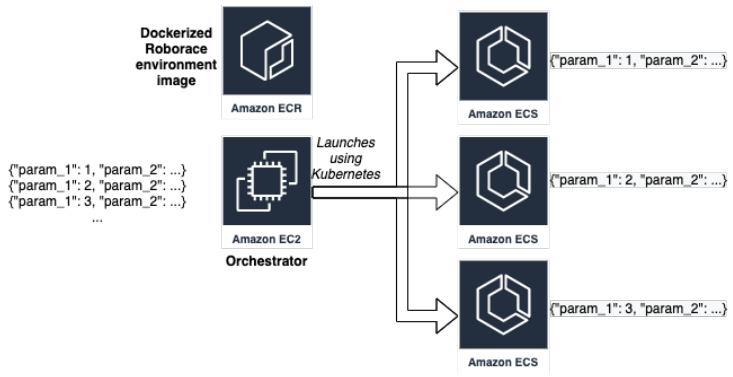

Figure 5: Multiple Parameters - Single Environment architecture

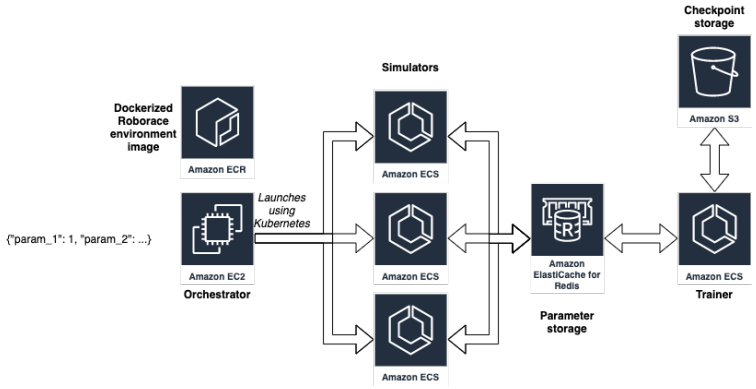

Figure 6: Single Parameter - Multiple Environments

3.3.2 Single Parameter - Multiple Environments

In this configuration, one set of parameters has multiple simulators. The orchestrator launches one training pod, a communication pod, and multiple simulators pulling the images from a docker registry. Each simulator runs one iteration, and relays the results through a Redis publisher-subscriber channel to the trainer. The trainer updates the model, relays the updated weights to the simulators, and the process repeats. What is more, checkpoints are periodically saved to s3 for reproducibility.

The advantage of this configuration is that multiple simulations can be run simultaneously thus, not being limited by the speed of a single trainer. However, at the expense of system complexity.

3.4 Online Visualization Monitoring Tool

The purpose of the analytic system is to provide a unified monitoring tool to analyse the training process and evaluate how good the car is learning to drive. It is capable of displaying multiple training instances and detailed information such as car path, rewards, speed, etc as shown in Figure 7. The analytic tool mimics the functions of tensorboard and provide useful insights for future works.

The Roborace analytic system is made of three major components as illustrated in Figure 8:

3.4.1 Analytic Logger

The analytic logger is a simple class that logs important information from the observations. For each episode of training, it accumulate observations in a buffer and flushes to a log file when the agent was reset. The content of log files will used by the proxy server to provide structured data for front-end user interface.

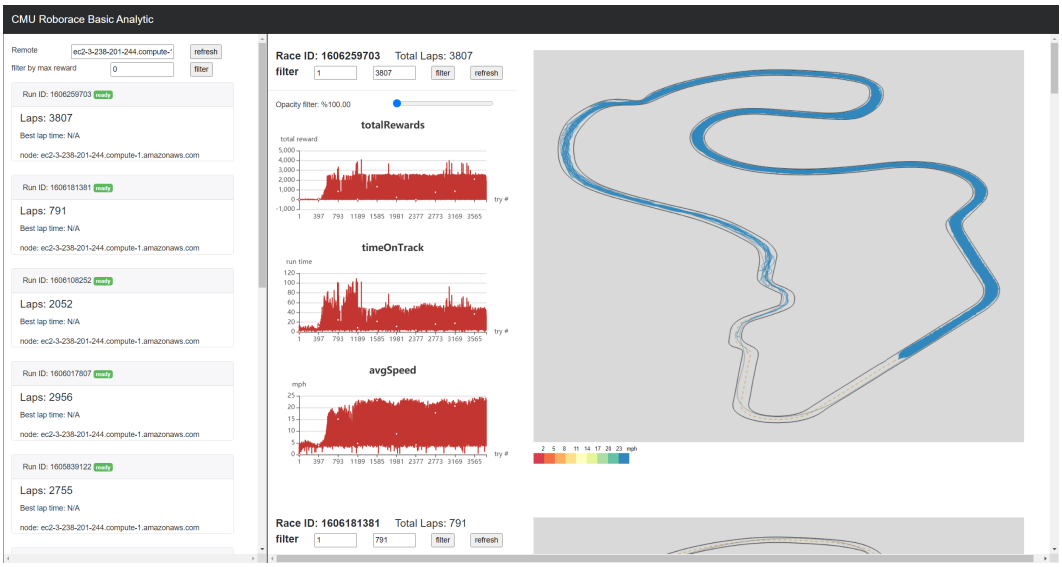

Figure 7: User Interface of Roborace analytic system

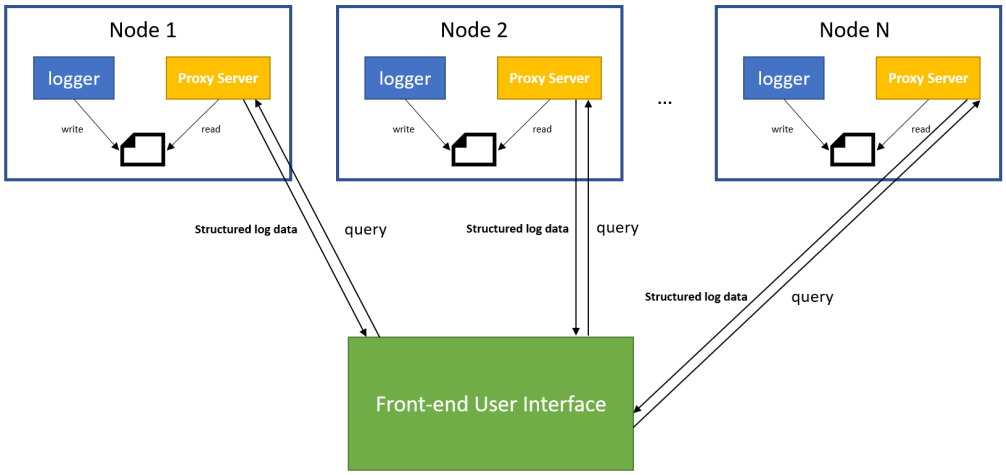

Figure 8: Roborace analytic system structure

3.4.2 Proxy Server

The proxy server is a light weight python http server that runs on the node that produces the log files. It processes and produces all the necessary data for the front end including track data, car paths, lap time, rewards, speeds, etc.

3.4.3 User Interface

The front end interface is a web application developed using vue.js, echarts and d3.js, the front-end interface could be launched anywhere and is able to connect to multiple nodes in the same time and support multiple functions for analytic purposes. The front end is composed of the following major pieces:

Control panel: The control panel locates at the left side of the panel and displays general information of each training instances including training ID, node name, best lap time, and number of episode. On the top of the control panel, we can input a list of remote servers that runs the training process, the front end will automatically connect to the praoxy servers and load data. The data loading process

is asynchronous and a green button will show up once loading is completed. We can also filter the training process by setting a threshold of maximum observed rewards, which can help us identify good training parameters.

Detail panel: On the right hand side of the user interface displays the detailed information of the training process. It contains 4 visualizations:

(1) **Average rewards:** the average rewards graph is a line chart with x-axis represents number of episode and y-axis represents average rewards of that training episode.
(2) **Time on track:** the time on track graph is a line chart with x-axis represents number of episode and y-axis encodes time spent on track during that episode.
(3) **Average speed:** similar to previous two, the average speed graph is a line chart with x-axis represents number of episode and y-axis represents average observed speed during that episode.
(4) **Trajectory graph:** the trajectory graph is a canvas with track and trajectories of each episode plotted on it. This graph is very useful and can help us understand the car's behavior on the track.

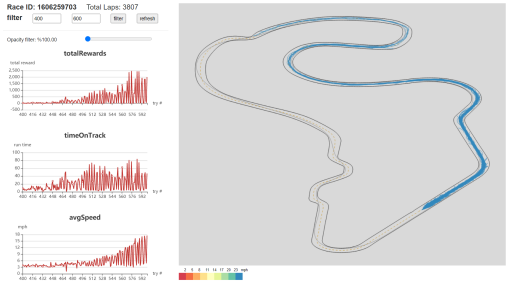

Figure 9: User Interface after filtering

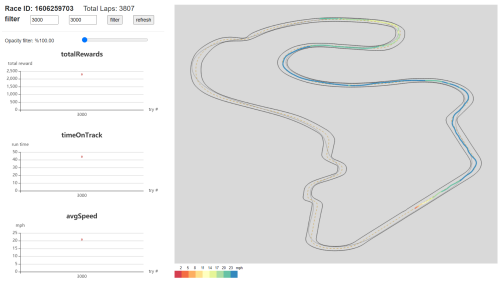

Figure 10: Car path with speed

The detail panel also support filter by episode which provides a much more fine grained analytic as shown in Figure 9. When we zoom in to show only one episode, the car path's color will encode its speed as shown in Figure 10, it can provide more detail about the car's behavior and can help us identify issues with the car.

4 Experiments

4.1 Basic Info with/without Trajectory Info

First we try to verify the trajectory information is necessary for training. Even with all features of the car itself, it is almost impossible to train a RL model in practical. Theoretically speaking, the agent can implicitly generate the track information by memorizing the points where it rush out of the track. However, this requires tons of training episodes to be well learned by a neural network.

Thus we design a comparison experiment between these two settings, to prove trajectory information is necessary, as shown in Table 3. In this experiment, the state in State1 are all features shown in previous Table 2, while the State2/State3 are the same features with additional trajectory information, which are 10 points along the human-expert/center path with rotated coordinates.

Table 3: Experiments with/without Trajectory Info.

State1	State2	State3
{Processed Feature}	{Processed Features + Center Path}	{Processed Features + Expert Path}

4.2 Basic Info with/without Visual Info

Another meaningful experiment would be related with the importance of the visual information. If we explicitly feed the trajectory to the racing car, it is much less challenging compared with direct image information. Thus we would like to add an experiment, as shown in Table 4, to measure the improvement caught simply by the image information, and whether we can drive based on only image features.

Table 4: Experiments with/without Visual Info.

State1	State2	State3
{Processed Feature}	{VAE Embeddings}	{Processed Features + VAE Embeddings}

4.3 Neural Network Sizes

If things work well, we would like to explore the capability of our neural network model. In some cases, wider and deeper models lead to a relatively better performance. Thus we compare different sizes of neural networks, as shown in Table 5, to measure whether our model's performance can be further improved.

Table 5: Experiments with Different NN Sizes.

Exp4 NN Size	Exp5 NN Size	Exp6 NN Size
32×32×32	64×64×64	256×256

5 Results & Analysis

5.1 Basic Info with/without Trajectory Info

We first show the results whose features are only basic features, as in Figure 11. Obviously, the car cannot even learn to achieve the first curve, with sufficient training episodes. This is consistent with our hypothesis that this is essentially impossible in practical.

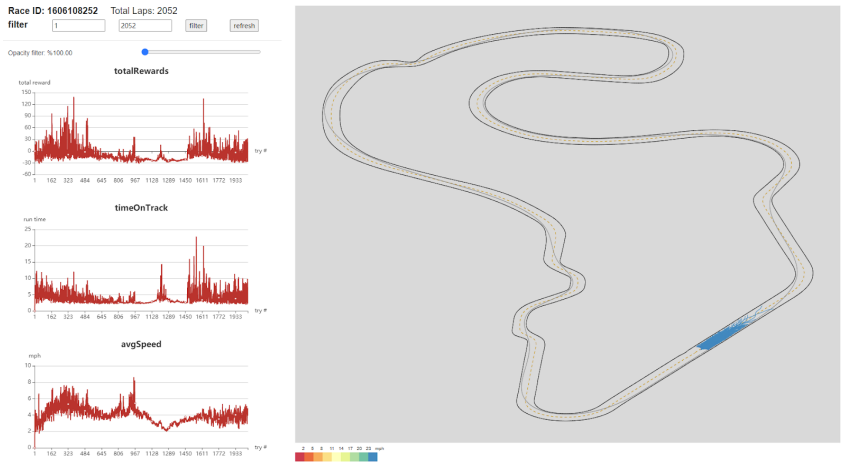

Figure 11: Basic Features Only.

Then we compare the performance between ones with center/expert trajectory info, as shown in Figures 12 and 13. With the similar training time, we can tell the center info is easier to utilize by the agent, but the expert line leads to a more sophisticated but shorter driving path eventually.

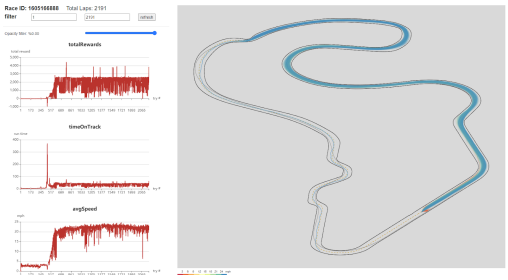

Figure 12: Basic + Center Info.

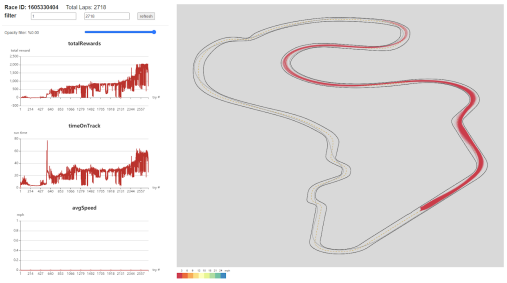

Figure 13: Basic + Expert Info.

5.2 Basic Info with/without Visual Info

In this part, we compare the state consisting of VAE only and the combination of Basic and VAE. From the result, we observe that VAE embeddings are enough for learning to drive. However, if we evaluate more carefully, we find that there exists many zig-zag behaviors during driving. This is because the agent never knows its current speed and acceleration. Thus the car will prefer to drive back to the center line as soon as possible, which is the zig-zag behavior.

Although the combined feature in Figure 15 is slower to learn, caused by higher dimensions of features, the final results are more stable and smarter than following center lines, as in the left Figure 14.

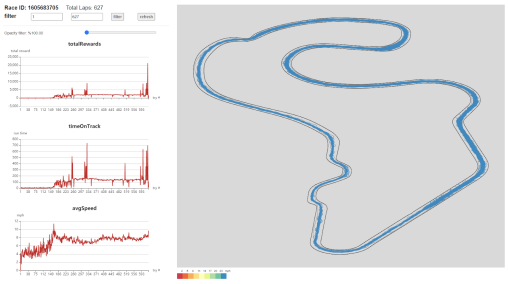

Figure 14: VAE Embeddings.

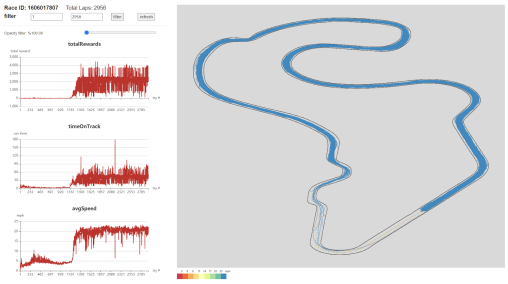

Figure 15: Basic + VAE Embeddings.

5.3 Neural Network Sizes

In this set of experiments, we explore how different layers and units affect the training process. From the Figures 16, 17 and 18, we can see that $32 \times 32 \times 32$ is slightly weaker than the other 2 during training. However, this gap is not quite huge. Thus we can claim in this magnitude of features, the NN sizes we use are pretty sufficient.

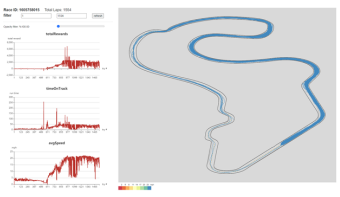

Figure 16: NN Size: 32*32*32.

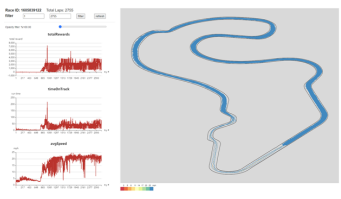

Figure 17: NN Size: 64*64*64.

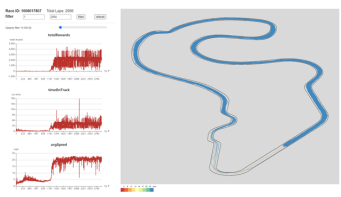

Figure 18: NN Size: 256*256.

6 Conclusion & Future Work

In our journey of using reinforcement learning to race, we explored multiple domains including AWS DeepRacer and Roborace. In the prior, we learned the fundamentals using a training pipeline that was fully managed and abstracted by AWS. In the latter, we built the entire training pipeline ourselves around Arrival's highly realistic racing simulator used to test real, full-size autonomous race cars. Our pipeline includes an OpenAI gym compliant reinforcement learning environment, web-based visualization tools, containerization automated deployment, and the ability to train in a distributed manner. We also developed baseline models in the Roborace environment which utilize visual feature embeddings and other pose data and prove that reinforcement learning is a feasible approach to realistic autonomous racing.

Here we propose several directions for future work. From a learning perspective, different learning algorithms and experimental conditions could be introduced. Our baseline agents have only been exposed to one race track under one set of physical parameters which severely limits its ability generalize. We also believe that more sophisticated perception capabilities will be critical in the future. Our environment could be improved on numerous fronts with the primary being integrated it into a robotics stack that can actually interface with the real Roborace vehicle. Furthermore, we believe that massive scale will be needed to get a vehicle on the real track.

References

Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, et al. Deepracer: Educational autonomous racing platform for experimentation with sim2real reinforcement learning. *arXiv preprint arXiv:1911.01562*, 2019.

Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017. URL http://arxiv.org/abs/1710.03748.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Duerr. Super-human performance in gran turismo sport using deep reinforcement learning. *arXiv preprint arXiv:2008.07971*, 2020a.

Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Duerr. Super-human performance in gran turismo sport using deep reinforcement learning, 2020b.

Adithya Ganesh, Joe Charalel, Matthew Das Sarma, and Nancy Xu. Deep reinforcement learning for simulated autonomous driving, 2016.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

Shihui Li, Yi Wu, Xinyue Cui, Honghua Dong, Fei Fang, and Stuart Russell. Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4213–4220, 2019.

Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *arXiv preprint arXiv:1304.1672*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013a.

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, 2013b. URL http://arxiv.org/abs/1312.5602.

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 *International conference on machine learning*, pages 1928–1937, 2016.

OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018. URL http://arxiv.org/abs/1808.00177.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. URL http://arxiv.org/abs/1910.07113.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Wilko Schwarting, Tim Seyde, Igor Gilitschenski, Lucas Liebenwein, Ryan Sander, Sertac Karaman, and Daniela Rus. Deep latent competition: Learning to race using visual control policies in latent space.

Wilko Schwarting, Tim Seyde, Igor Gilitschenski, Lucas Liebenwein1, Ryan Sander, Sertac Karaman, and Daniela Rus. Deep latent competition: Learning to race using visual control policies in latent space, 2020. URL https://www.gilitschenski.org/igor/publications/202011-corl-deep_latent_competition/corl20-deep_latent_competition.pdf.

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. URL http://arxiv.org/abs/1712.01815.

Stable Baselines. Stable Baselines-Docs-SAC. https://stable-baselines.readthedocs.io/en/master/modules/sac.html.

Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*, 2018.