Reinforcement Learning in TrackMania

Guanchen Liu
College of Computer, Mathematical, and Natural Sciences
University of Maryland – College Park
College Park, MD
gliu0529@umd.edu

***Abstract*— The project aimed to train agents in TrackMania Nations Forever, a racing game allowing vehicles to traverse through courses and compete for the best lap time. We created agents, trained using reinforcement learning to simulate humans playing the game with the aim of completing the course with the shortest time possible. Our implementation of the vehicle with Deep Q-Learning showed lap times that were comparable to human experts and showed generalization characteristics that enabled the agents to complete courses without prior knowledge and showed stability in performance that could not be rivaled by humans.**

Keywords—Reinforcement Learning, Autonomous Driving, Racing, DQN

I. Introduction

With the rising popularity and advancements in self-driving cars, questions arise regarding how well models can perform in a closed-circuit racing environment, where the agent is optimized to maximize the pace to complete a course. This project aimed to explore AI's performance in a racing circuit. There were no available sim-racing games that enabled access to live telemetry data ported out of the game. If we implemented an agent this way, we would have needed multiple CNN models to gather information on the screen. However, this would have introduced conflicts between gathering track information and vehicle information, as the screen used to gather vehicle information was not suitable for gathering track information simultaneously. Thus, we implemented the agent in the TrackMania world. TrackMania is an arcade-type racing game that has its unique physics and simulation characteristics. However, it is available to port live telemetry out of the game for us to train our reinforcement learning models. For our purposes, we saw the TrackMania environment just like the real world, and our objective for the agent was to traverse the track as fast as possible. This project did not use any computer vision techniques to simulate lidar or laser tools to gather environmental information; instead, it simulated it using the vehicle position relative to the track. This ensured faster processing and lower computational costs for processing.

II. Literature Survey

There were a few attempts to train reinforcement learning in the TrackMania environment. This section of the paper reviews some of the approaches and methods used to complete the task.

A. *Tmrl* [1]

Tmrl was a distributed reinforcement learning framework in TrackMania. The project did not attempt to gather data directly in-game; instead, it used screen capture to collect all observation space for the agent. It employed a CNN model to detect the edges of the track as well as the vehicle's speed. Previous input actions were used to determine vehicle inputs, and that was all the agent received in determining its actions. For reinforcement training, the framework used rtgym to elastically constrain the times at which actions and observations were sent or retrieved. This approach was likely chosen due to the runtime complexity of a CNN model. The CNN model may not have been able to compute each observation in time for the next iteration; thus, an elastic observation time approach was used to eliminate this issue and provide better observations for the agent to learn. The framework employed SAC and REDQ methods in training the agent.

The methodology of this approach severely limited the information available to the agent, and the results they achieved showed poor performance and a significant gap between its capabilities and those of a human expert.

B. *TrackMania AI* [2]

TrackMania AI approached the problem differently compared to TMRL. What makes TrackMania AI special is that it was able to convert the in-game binary track file into a human-interpretable format, which was then used as the agent's observation. With the in-game map file, they were able to create a top-down view of the vehicle in relation to the track. From there, they could simulate the lidar observation that the previous team had implemented, but this time without using computer vision. This approach again used the SAC approach, and the agent showed similar performance to the first team. However, the reward function that they implemented could be improved. Since they had access to the in-game map file, it was not ideal that their best-performing agent still maximized the reward for speed by crashing into the walls in corners.

C. *Trackmania_rl_public* [3]

This project takes advantage of real-time telemetry in the game and combines it with computer vision. From the code of the project, it is unclear how the captured video was used. The project uses the IQN algorithm, which is a distributional generalization of DQN. The model achieved very good results; however, it is worth questioning the reproducibility of these results. Nevertheless, this project employs many approaches that we can reference.

The three projects mentioned above use different approaches to achieve the same goal: training agents to complete the course. There are many novel approaches among these projects. For our project, we will not use any computer vision methods and will construct the observation space of the agent entirely with the numerical data provided by the game. Two of the projects use SAC, while another uses a derivative of DQN. We will explore both algorithms in our project and compare the differences between the two.

III. Methodology

In this section, we will discuss the environment and simulation setup, agent observation and action space, as well as our approach to agent rewards and shaping.

A. *Environment Setup*

The environment used for this project is TrackMania Nations Forever [4], a free-to-play game. To gather real-time information in the game, we made use of TMInterface [5]. TMInterface is a custom launch client for TrackMania Nations Forever, originally created for the speedrun community to reproduce world record runs by recording and replaying commands for each map. This naturally requires in-game data, which we leverage to train our agent. By launching the game through TMInterface and installing its Python library, we were able to capture vehicle statistics in real time. To control the vehicle, we also used TMInterface, sending commands through its Python API, consisting of four discrete variables: left, right, up, and down. The version of TrackMania we worked with lacked analogue support for controllers, so having discrete inputs from the agent did not impact the vehicle's performance in navigating through the course.

We set up a Python class called TMInterfaceManager, responsible for receiving game data and sending control commands. The received game data was processed and sent to the simulation as observational space. The TMInterfaceManager class included functions such as rollout, get_obs, setup, input, and respawn:

- The rollout function checked if the game was configured and connected, calling get_obs and returning observations to the simulation environment. If the game was not connected to Python, it called the setup function to establish the connection.
- The input function processed the action space of the agent and sent it to the game through the interface.
- The respawn function, when called from the simulation as the current episode ended, resets the vehicle's position to its initial stationary position, ready to start the next episode.

B. *Track Information*

To train an agent to navigate through the course, it is imperative that the agent has some knowledge of the track, even if it's not the entire track. Knowing the immediate position of the vehicle in relation to the track is sufficient. In the TrackMania environment, the track has a fixed width of 20 units, regardless of the type of track. This allows us to represent track position through vehicle position, where we can implicitly refer to the track position by manually traversing the course.

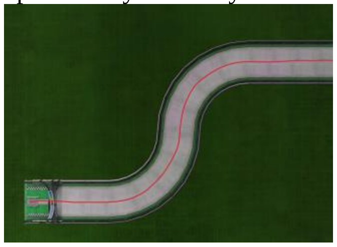

Fig. 1. Visualization of outline of track

By closely following the centerline of the track as we travel through the course manually, we store the position of the centerline at fixed intervals. Connecting these points together, we create an outline of the track.

Each pair of points on the centerline represents a zone. As the vehicle travels through the course, the TMInterfaceManager determines which zone the vehicle is in.

C. *Simulation Setup*

We prepared our simulation environment using Gymnasium [6]. Gymnasium is a fork of the OpenAI Gym environment since Gym is deprecated, and Gymnasium is supported by more libraries. We created our custom environment using Gymnasium.

We designed our own TMGymInterface class and implemented custom constructors, reset, and step functions. Our agent had 12 different metrics in the observation space and 12 discrete actions.

- The step function called TMInterfaceManager with the agent's action, retrieved the observation space from TMInterfaceManager, calculated the reward based on the received observation, and processed the simulation while determining the values of the variables "terminated" and "truncated."
    - "Terminated" was set to true if the vehicle information received indicated that it had reached the end of the course.
    - "Truncated" was set to true if the vehicle:
        - Crashed
        - Was going backward
        - Stopped on the track

D. *Reinforcement Learning Algorithm*

We will use Stable Baseline3 to train our agent [7]. Stable Baseline3 is a Python library that offers multiple reinforcement learning algorithms.

Given the constraints of our simulation environment, which does not allow the launch of multiple instances simultaneously, we have excluded algorithms that depend on parallel environments for training, such as A2C. Since running the game in parallel is not possible, we have focused on off-policy algorithms that incorporate a replay buffer mechanism to enhance sample efficiency.

We will specifically consider two algorithms: SAC (Soft Actor-Critic) and DQN (Deep Q-Network). Each of these algorithms has distinct characteristics that make them suitable candidates for our task.

SAC is suitable for environments with continuous action spaces, but in our case, we will adapt it to work with a discrete action space. We achieve this by employing the argmax function on the algorithm's output, effectively simulating a discrete action space. SAC is special as it is trained to maximize the trade off between expected return and randomness.

On the other hand, DQN is designed for environments with discrete action spaces. While it lacks the sample efficiency of SAC, it is designed to generalize the model to new states and actions. By comparing the performance of these two algorithms in our unique setup, we aim to determine which one is more suited to the specific challenges presented by our racing simulation environment. This evaluation will help us better understand the strengths and weaknesses of each approach and guide our choice for the most effective algorithm.

E. *Observation and Action Space*

The agent has a total of 12 observation spaces. These spaces encompass various aspects of the vehicle's state, including its velocity, turning rate, lateral velocity, and acceleration.

Additionally, the observation space includes the relative positions of the vehicle with respect to the track. This information comprises the angle between the vehicle and the centerline of the track, the angle between the vehicle and the centerlines of the next three zones, the distance from the vehicle to the centerline, and the distance to the centerlines of the next three zones.

Calculating the angles between the vehicle and the track involves using the vehicle's orientation quaternion. Quaternion values represent the rotational position in four-dimensional space. These values are first converted into a rotation matrix and then further converted into a vector. For each point or zone along the centerline, a vector is formed between the two points that define the zone. The angle is then calculated between the zone vector and the vehicle's orientation vector.

F. *Rewards*

We implemented two policies for the agent to learn. The first policy aimed to train the AI to travel through the course while following the centerline of the track. The second policy focused on the agent maximizing its speed while adhering to the racing line. Both policies shared some similar reward policies, as well as some differences.

1) Basic Policies: The agent received rewards based on its relative velocity (1).

$$rew \mathrel{+}= velocity \tag{1}$$

If the agent was moving but traveling backward, a penalty was subtracted from the reward (2) before considering the truncated rule. This was done to encourage the model to learn that moving backward would be penalized.

$$rew \mathrel{-}= 100 \tag{2}$$

If the agent reached a truncated state, indicating it ended the episode prematurely due to crashes, a penalty was deducted from the reward (2). For agents that successfully completed a course, an end-of-episode reward was added for the episode (3). Race time was recorded in milliseconds, and this reward policy favored runs that finished in the shortest time possible.

$$rew \mathrel{+}= 500 \times \left(\frac{15000}{racetime}\right)^3 \tag{3}$$

2) Centerline Policy: To encourage the agent to learn to travel through the center of the course, we penalized the agent when the vehicle's orientation deviated from the centerline. The amount of the penalty depended on the degree of deviation from the centerline.

$$rew \mathrel{-}= |angle\ to\ centerline| \times 10 \tag{4}$$

The "angle to centerline" values ranged between 0 and 1, resulting in a penalty ranging from 0 to 10. This penalty encouraged the agent to stay aligned with the centerline while navigating the course.

3) Racing Line Policies: To teach the agent to follow the racing line, we needed to represent the racing line numerically. Since all available data was through in-game variables without the aid of computer vision to calculate the optimal racing line strategy, we needed a numerical representation of the angle difference between the vehicle's orientation and the racing line.

To represent the racing line, we denoted it as a combination of the angle of the vehicle to the centerline as well as the angles to the next three zones. The formula for this representation was as follows:

$$rew \mathrel{-}= \frac{\left|angle\ to\ centerline + \frac{angle\ to\ next\ zone}{2} + \frac{angle\ to\ next\ next\ zone}{6} + \frac{angle\ to\ next\ next\ next\ zone}{8}\right|}{4} \times 100 \tag{5}$$

Equation (5) considered four different points ahead of the vehicle and combined them to produce a racing line strategy. This equation ensured that no matter how tight a corner was, the agent was aware of what lay ahead and would be penalized if it failed to anticipate the corner. For instance, in the case of a hairpin corner, which is extremely tight, if the agent continued to follow the centerline while traveling on the straight before the corner, it would be heavily penalized. This was because the racing line policy considered the angles between the vehicle and the next few zones. The angle difference between the vehicle and the third closest zone would be significant in this scenario.

Effectively, by adhering closely to the racing line policy, the agent always aimed to hit the geometric apex of the corner, which is the line that minimizes cornering time while maximizing speed and acceleration.

In addition to the racing line penalty, the policy also incorporated a reward mechanism for maximizing speed. Among the 12 discrete action space inputs, rewards were granted if the forward key was pressed. Among these actions with the forward key pressed, the action with only the forward key received the most significant reward. A severe penalty was imposed if the action did not involve the forward or backward keys, i.e., using only directional keys or doing nothing. This ensured that the agent consistently maximized acceleration. Even when a corner approached, the agent continued to maximize speed and only decelerated by braking when necessary. No action was wasted during the timestamp, ensuring that the vehicle neither remained at a constant speed nor experienced unnecessary deceleration.

G. *Training and Testing Track*

All models were trained on the training track Fig. 2. The training track was designed to teach the agent the fundamentals of vehicle commands. It started with a low-speed left-hand corner, followed closely by a right-hand corner. A short straight section followed, intended for the vehicle to increase speed before tackling a high-speed right and left corner. The track ended with a long straight section. The corners were placed in pairs to reduce the risk of the model overfitting, learning how to turn in one direction but not the other. Different speed corners were also incorporated into the design to encourage the model to generalize cornering skills. This prevented the vehicle from overfitting and allowed it to learn how to navigate slow corners while not overestimating its ability to do so in high-speed corners, where it might crash into barriers. The long straight at the end of the track allowed the vehicle to learn how to maximize speed by only accelerating when there were no corners ahead.

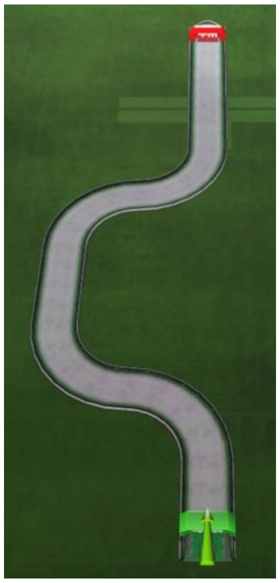
Fig. 2. Train track

After a model was successfully trained and able to reliably complete the course, it was then evaluated on our test track Fig. 3. The test track included all corner scenarios that were not present in our training track. It consisted of long curves, hairpin turns of varying speeds, high-speed sweeping curves, and combinations of short left and right curves. If the model could complete our test course, we considered it to be generalizable, demonstrating its ability to adapt to a range of corner scenarios.

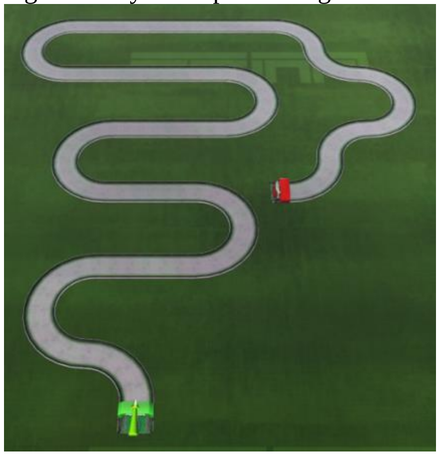
Fig. 3. Test track

IV. Results

A. *SAC with Centerline Policy*

Since SAC outputs a continuous action space, we trained the model by converting its output into a discrete action space. However, this approach yielded poor results. When the model initially started training, the largest value in the action space might not be unique. This led us to interpret the model's action as the first action that shared the duplicate value. Consequently, the model became stuck in a local minimum, hampering its ability to explore effectively.

B. *DQN with Centerline Policy*

The DQN model underwent training for 3 hours, accumulating a total of 1.4 million timesteps. This trained model demonstrated promising results as it was able to complete our training course in under 19 seconds. During training, the model closely followed the racing line, and it also completed our test track without any issues. However, its completion time on the test track was over a minute, whereas a human expert can complete the track in approximately 36 seconds.

Upon investigating the model's actions, it became evident that the model prioritized following the centerline over maximizing speed and completing the course in the shortest time. This behavior was attributed to our reward policy setup, where the model had found a balance between speed, route, and time. It sought to minimize the penalty incurred by deviating from the centerline, while ensuring that the episodic reward did not overshadow the time-step reward provided for each command.

With this proven model as our baseline, we proceeded to fine-tune the hyperparameters. Employing grid search and utilizing reward and network loss graphs, we carefully adjusted the hyperparameters to promote stable and consistent network learning.

TABLE I. HYPERPARAMETERS

Learning Rate	0.00064
Gamma	0.99
Batch Size	128
Update Interval	10,000
Training Frequency	4
Gradient Steps	1
Exploration Rate	0.05
Neural Network Structure	256x256

C. DQN with Racing Line Policy

The new DQN model was trained using the tuned hyperparameters for a total of 20 hours, accumulating 12.2 million time steps. This trained model was able to complete our training course in under 12 seconds, while a human expert could complete it in under 11 seconds. The model navigated with a degree of margin around the track, not hugging the barriers closely. This approach aimed to ensure the model's generalization to other courses. Subsequently, the same model was continuously trained for 50 hours, during which it managed to complete the training track in under 10 seconds. The model now used the entire track, pushing the limits of navigation. However, this extended training period led to overfitting on the training course, as the model essentially memorized each corner and scenario.

When evaluated on the test course, this overfitted model performed poorly, struggling to handle the tightness of the first corner. Consequently, we opted to use the generalized model for evaluation on the test course. The generalized model completed the test course in 39 seconds, a favorable performance given that the model had never been trained on this course. In fact, the model's performance was comparable to, if not better than, that of a human expert playing the course for the first time.

TABLE II. RESULTS

Track	**Human Expert**	**DQN Center Line**	**DQN Racing Line**
	seconds		
Train	10.5	18.8	11.9
Test	37.3	> 60	39.72

V. CONCLUSION

In this project, we successfully trained reinforcement learning agents to navigate in the challenging racing environment of TrackMania. Our approach, which emphasized a balance between route accuracy and speed optimization, yielded impressive results. The DQN model with a racing line policy, showcased great performance. It completed the training and testing track with impressive times and demonstrated its ability to generalize to new scenarios.

In the future, we aim to further enhance our agent's performance and capabilities. We would like to teach the agent to perform in game mechanics, such as sliding. This requires a careful setup of reward policy to tread a fine line between maximizing speed and exploiting sliding. Key of this is to teach the agent on straights.

In addition, instead of using track information from the midpoint, we can gather track one-time track information of the track using computer vision or to create a visualization of the track using our track centerline data in closer intervals. Instead of just having the checkpoint positions, giving more environmental information to the agent can potentially improve model performance. The agent will take into account of what's up next on the track and consider its current position relative to the track and come up with the holistic strategy in the position of the car on the track to maximize the speed while minimizing the track time.

REFERENCES

[1] "TMRL," *GitHub*, Oct. 26, 2023. https://github.com/trackmania-rl/tmrl

[2] GobeX, "TrackMania_AI," *GitHub*, Nov. 21, 2023. https://github.com/AndrejGobeX/TrackMania_AI (accessed Dec. 15, 2023).

[3] pb4git, "Trackmania AI with Reinforcement Learning," *GitHub*, Dec. 14, 2023. https://github.com/pb4git/trackmania_rl_public (accessed Dec. 15, 2023).

[4] "TrackMania Forever • The Most Popular Online PC Racing Game," *www.trackmaniaforever.com*. https://www.trackmaniaforever.com/

[5] "TMInterface," *donadigo.com*. https://donadigo.com/tminterface/ (accessed Dec. 15, 2023).

[6] "Gymnasium Documentation," *gymnasium.farama.org*. https://gymnasium.farama.org/

[7] "Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 2.2.1 documentation," *stable-baselines3.readthedocs.io*. https://stable-baselines3.readthedocs.io/en/master/# (accessed Dec. 15, 2023).