TORONTO METROPOLITAN UNIVERSITY
FACULTY OF ENGINEERING, ARCHITECTURE AND SCIENCE
DEPARTMENT OF AEROSPACE ENGINEERING

Soft Actor-Critic for Autonomous Race Car Control

Bryan Kikuta

AER870 Aerospace Engineering Thesis - Final Report
Faculty Advisor: Dr. Reza Faieghi
April 10th, 2025

Acknowledgements

I would like to express my sincere gratitude to my faculty advisor, Dr. Reza Faieghi, for his invaluable guidance, expertise, and support throughout this thesis. His insightful feedback and encouragement were instrumental in the successful completion of this research.

I am deeply grateful to the Department of Aerospace Engineering at Toronto Metropolitan University for providing me with the opportunity to pursue this thesis and for creating an environment conducive to research and academic growth.

I would also like to extend my appreciation to BeamNG for generously providing a free license for BeamNG.tech, which was essential for implementing the simulation environment required for this thesis. Their support has significantly contributed to the quality and depth of this research.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout my academic journey.

Abstract

This thesis implements a reinforcement learning approach for optimizing the trajectory of a race car about a given track in real-time provided observations of the state of the car, in addition to the state of the racetrack with respect to the car. A Soft Actor-Critic (SAC) model operating in a continuous space implemented with Stable Baselines 3 was trained and tested on a fork of BeamNG's Gymnasium wrapper repository. The Soft Actor-Critic agent was provided vehicle state information such as the velocity, angle relative to the centerline, revolutions per minute (RPM), and simulated light detection and ranging (LiDAR) readings of the edges of the racetrack to provide an observation closely approximating the current state of the car. A custom reward function was generated prioritizing large track progress rates with a penalty for excessive steering rates to minimize oscillatory movement of the car keeping it minimal to avoid generating results which do not align with the main objective of optimizing the lap times. A training pipeline was developed where the model was initially trained on a small percentage of the racetrack, which was incrementally increased until the model was being trained on 100% of the racetrack. The performance of the agent was evaluated for the average lap completion percentage in addition to the average lap time if applicable. The agent was also evaluated on different vehicle types to study the transferability of the SAC model.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Research Objective . . . 2
- 1.3 Report Organization . . . 2

2 Literature Review **3**
- 2.1 Fundamentals of Reinforcement Learning . . . 3
  - 2.1.1 Reinforcement Learning Policy . . . 4
  - 2.1.2 Value Functions and Bellman Equations . . . 4
- 2.2 Overview of Soft Actor-Critic Algorithm . . . 5
  - 2.2.1 Soft Actor-Critic Policy . . . 5
- 2.3 Simulation Platforms for Autonomous Race Cars . . . 5
  - 2.3.1 Simulation Platform Overviews . . . 6
  - 2.3.2 Selected Platform . . . 8

3 Methodology **8**
- 3.1 Observation and Action Spaces . . . 8
  - 3.1.1 Observations . . . 9
  - 3.1.2 Actions . . . 10
- 3.2 Reward Function Design . . . 11
  - 3.2.1 Speed Reward . . . 11
  - 3.2.2 Steering Rate Punishment . . . 12
  - 3.2.3 Boundary Punishments . . . 12
- 3.3 RL Training System . . . 14
  - 3.3.1 Randomization . . . 14
  - 3.3.2 Training Curriculum . . . 15

4 Implementation **15**
- 4.1 System Configuration and Setup . . . 15
- 4.2 Gymnasium Environment Wrapper . . . 16
  - 4.2.1 Environment Architecture . . . 16
  - 4.2.2 Dataclasses for Configuration . . . 17
  - 4.2.3 Simulation Environment . . . 17
  - 4.2.4 Track Representation . . . 19
  - 4.2.5 Core Environment Methods . . . 19
  - 4.2.6 Simulated Sensors . . . 20
- 4.3 Soft Actor-Critic Implementation . . . 20
  - 4.3.1 Training Pipeline Design . . . 20
  - 4.3.2 Training Workflow . . . 21

5 Results & Discussions **21**
- 5.1 Training Results . . . 21
- 5.2 Model Performance Evaluations . . . 25

5.3 Path Comparisons 27
5.4 Vehicle Performance Comparisons 42
5.5 Challenges & Short-comings 44
5.5.1 Reward Function Design Challenges 44
5.5.2 Transferring Skills Between Track Sections 44
5.5.3 Training Time Constraints 44
5.5.4 Observation Space Complexity 44

6 Conclusions **46**

A Software Implementation **49**
A.1 Gymnasium Environment Wrapper 49
A.2 Progressive Training Pipeline 65

Nomenclature

API	Application Programming Interface
BeamNG.tech	High-fidelity vehicle simulation platform
CUDA	Compute Unified Device Architecture (Nvidia's parallel computing platform)
cuDNN	CUDA Deep Neural Network library
ETK 800	Vehicle model used as primary training vehicle
GPU	Graphics Processing Unit
Gymnasium	API standard for reinforcement learning environments (successor to OpenAI Gym)
LiDAR	Light Detection and Ranging (sensor technology for distance measurements)
MDP	Markov Decision Process
MlpPolicy	Multi-layer Perceptron Policy (neural network architecture used in SAC)
RL	Reinforcement Learning
ROS	Robot Operating System
RPM	Revolutions Per Minute
SAC	Soft Actor-Critic (reinforcement learning algorithm)
SB3	Stable Baselines 3 (reinforcement learning framework)
TORCS	The Open Racing Car Simulator
venv	Virtual Environment (Python)
α	Entropy regularization coefficient in SAC
γ	Discount factor for future rewards
π_θ	Policy with parameters θ
τ	Trajectory (sequence of states and actions)
θ	Relative angle between vehicle orientation and track centerline
$\dot{s}$	Steering rate (change in steering value over time)

List of Figures

2.1 The reinforcement learning cycle [1]. 3
2.2 Gazebo Simulator [2]. 6
2.3 BeamNG.tech Simulator [3]. 7
2.4 F1TENTH Simulator [4]. 8
3.1 Visualization of LiDAR rays at equal intervals from $-135°$ to $+135°$. 9
3.2 Vehicle relative angle from centerline. 10
3.3 Optimal racing line (red) versus track centerline (dashed). 11
4.1 Implementation architecture of the wrapper for BeamNG.tech. 17
4.2 Racetrack in West Coast USA used for training in BeamNG.tech [5]. 18
4.3 Track layout showing the shape and key features of the West Coast USA racetrack. 18
4.4 The ETK 800 sedan used as the primary development vehicle. 19
5.1 Stage 0 (5% track) training episode rewards. 22
5.2 Stage 1 (10% track) training episode rewards. 22
5.3 Stage 3 (50% track) training episode rewards. 23
5.4 Stage 4 (70% track) training episode rewards. 23
5.5 Stage 5 (90% track) training episode rewards. 24
5.6 Stage 6 (100% track) training episode rewards. 24
5.7 Stage 7 (100% track) training episode rewards. 25
5.8 Average lap times across models trained on different track percentages. . . . 26
5.9 Vehicle path during stage 0 model testing. 27
5.10 Vehicle path through chicane during stage 0 model testing. 28
5.11 Vehicle path during stage 1 model testing. 29
5.12 Vehicle path through chicane during stage 1 model testing. 30
5.13 Vehicle path during stage 2 model testing. 31
5.14 Vehicle path through chicane during stage 2 model testing. 32
5.15 Vehicle path during stage 3 model testing. 33
5.16 Vehicle path through chicane during stage 3 model testing. 34
5.17 Vehicle path during stage 4 model testing. 35
5.18 Vehicle path through chicane during stage 4 model testing. 36
5.19 Vehicle path during stage 5 model testing. 37
5.20 Vehicle path through chicane during stage 5 model testing. 38
5.21 Vehicle path during stage 6 model testing. 39
5.22 Vehicle path through chicane during stage 6 model testing. 40
5.23 Vehicle path during stage 7 model testing. 41
5.24 Vehicle path through chicane during stage 7 model testing. 42

List of Tables

1 The Observation Space 9
2 The Action Space 10
3 Training Curriculum Stages 15
4 System Configuration 16
5 Model Performance Metrics for Models Trained on Different Track Percentages 26
6 Vehicle Performance Metrics for Models Trained on Different Vehicles 43

1 Introduction

1.1 Background

Reinforcement learning is a branch of machine learning where an agent learns to make decisions for a specific task through interaction and exploration of its environment. Reinforcement learning differs from other branches of machine learning such as supervised learning as it does not require a dataset with labels to be trained on. Instead, reinforcement learning takes advantage of reward and punishment signals which indicate the quality of the actions that were performed by the agent in the environment. The rewards and punishments enable the agent to develop a policy for determining future actions to take to optimize the total accumulated reward, making it well-suited for complex tasks where the correct decision is not easy to determine, such as navigating and minimizing the lap times of a vehicle around a racetrack.

As revolutionary as it sounds, the concept of reinforcement learning has been around for a considerable amount of time, tracing back to the 1950's with Richard Bellman's work on dynamic programming [6], but a significant amount of interest has grown around the topic when DeepMind successfully showcased a system that was capable of surpassing humans in performance with games like Atari [7]. Since then reinforcement learning has continued to grow and become a center of interest for major companies like Nvidia with dedicated platforms for training reinforcement learning models [8].

At the core of reinforcement learning frameworks is the concept of a Markov Decision Process (MDP), where at each time step, an agent can observe the current state, choose an action out of an action space, and transition to a following state with a probability that is influenced by the action taken by the agent [7]. The agent's objective is to learn a policy that maximizes the expected cumulative reward over time [6]. In autonomous racing applications, the state typically encapsulates the vehicle's position, velocity, orientation, and sensory information about the track boundaries as seen in existing implementations [9][10]. The actions represent control inputs such as steering, throttle, and braking, while the reward function incentivizes faster lap times while discouraging behaviors like collisions or going off-track.

Soft Actor-Critic (SAC) is a recent and advanced reinforcement learning algorithm that was introduced by Haarnoja et al. in 2018 [11]. What distinguishes Soft Actor-Critic from earlier reinforcement learning approaches is the introduction of entropy maximization, which leads to exploration and stability [11]. The maximum entropy framework has multiple advantages to it over traditional reinforcement learning models, such as improved exploration, robustness to variations in environments, and better convergence properties by avoiding premature convergence [1]. The algorithm's architecture features two neural networks working in tandem: an actor network that outputs optimal actions for given states, and a critic network that evaluates the expected future rewards of state-action pairs, creating a robust framework for learning complex control tasks.

Racing simulators and games are a good application of the Soft Actor-Critic algorithm and in general for reinforcement learning due to the complexity of determining the optimal path to traverse the race track in the shortest time possible. Implementations of Soft Actor-Critic with racing simulators are available, ranging from implementations with The Open

Racing Car Simulator (TORCS) as described by Tong et al [9] to implementations with Gran Turismo Sport showcasing exceptional performance as described by Fuchs et al [10].

High-fidelity simulation environments like BeamNG.tech which is utilized as the training environment in this thesis provide realistic physics models to accurately capture and simulate the performance of vehicles on a racing track, offering numerous advantages. The use of simulation environments enables for a cost effective and safe approach for training reinforcement learning agents, in addition to the acceleration of training as in simulation, time can be accelerated.

1.2 Research Objective

The objective of this thesis is to develop a Soft Actor-Critic reinforcement learning agent to autonomously navigate a race car around a provided track as fast as possible while remaining within the confines of the racetrack and not taking on any damage to the car during the completion of the lap. Once the SAC agent is trained, its performance is evaluated by studying its trajectory around the racetrack in addition to computing the average lap times and the average lap percentage completed. In addition, the performance of the agent will be evaluated with different vehicle models to analyze the transferability of a trained agent to vehicles with different characteristics. The main components of this thesis will be:

- Implement a Gymnasium wrapper that can be utilized with Stable Baselines 3 to train a SAC model.
- Identify an observation space that will provide sufficient information for the agent to determine the current state of the car.
- Identify an action space that will provide sufficient control of the car.
- Develop a reward function with the objective of minimizing the lap-times of the race car.
- Analyze the performance of the SAC driven agent in navigating the racetrack and minimizing the lap times.
- Evaluate the performance of the trained agent when controlling different vehicles on the racetrack.

1.3 Report Organization

The organization structure of the report is as follows.

1. Literature Review – This section of the report focuses on the fundamentals of reinforcement learning, how the SAC algorithm works, in addition to the advantages of it, and covers different simulation platforms for reinforcement learning.
2. Methodology – This section of the report describes the observation and action spaces that the agent would work with, in addition to the formulation and reason behind the reward function, and the general training process for the agent.

3. Implementation – This section of the report discusses how the items covered in the methodology section of the report were put to practice by analyzing the techniques utilized for developing the Gymnasium wrapper, in addition to implementing a SAC model.

4. Results & Discussions – This final section of the report covers various topics such as the performance of the different models based on the lap percentage they were trained on, in addition to a comparison of the performance of the model with different vehicle types.

2 Literature Review

2.1 Fundamentals of Reinforcement Learning

To build a foundation for understanding the Soft Actor-Critic algorithm, the fundamentals of reinforcement learning will be covered. In reinforcement learning (RL), there are two main parts at play, these being the agent and the environment [1]. In RL, the agent interacts with the environment through a set of actions described by the action space, and determines the state of the environment through a set of observations described in the observation space of the agent. Actions are typically denoted using a_t where t denotes the current step of a given trajectory τ, and the state is denoted by s_t. While the above does provide enough information for the agent to interact with the environment it has been placed in, an additional input to the agent is required, which is the reward received from the environment so that the agent can evaluate the actions it takes. This constant cycle of interaction between the agent and the environment is seen in the figure below.

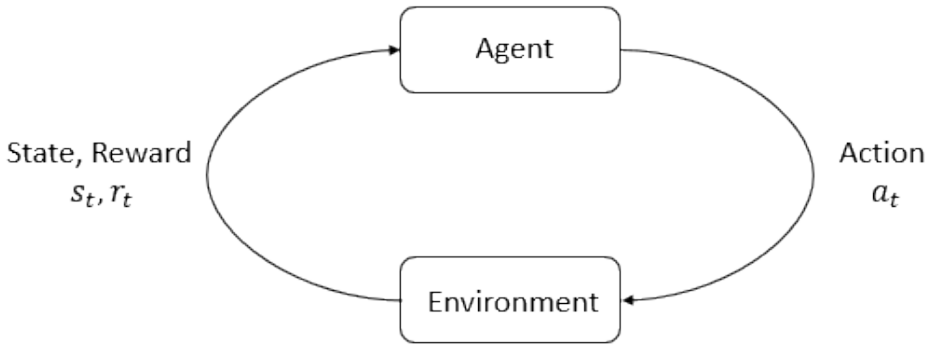

Figure 2.1: The reinforcement learning cycle [1].

Having covered the surface level architecture for reinforcement learning, a more in depth analysis of the key components and mathematical foundations that make up RL can now be covered. The following sections will explore the fundamental elements that make up an RL system and allow for feedback and decision making for the agent of the system.

2.1.1 Reinforcement Learning Policy

The policy of an agent can be thought of as a map for making decisions (those being actions), or more accurately, a map of probabilities of decisions if it is stochastic policy. The equation for stochastic policy can be seen below.

$$\pi_\theta(a|s) = P[A_t = a|S_t = s, \theta] \tag{1}$$

where π_θ denotes a policy with parameters θ [1]. The policy is the probability that $A_t = a$ if $S_t = s$ where a is an action in the agent's action space, and s is a state represented in the environment and t is a step of trajectory τ [6]. This can be intuitively understood through an example, where the number of grids an agent can move forward in a fictional grid environment are defined by a roll of a dice. In this example, there are six possible actions a that can be taken, and given that a roll of a dice is not influenced by where an agent would be standing, the probabilities of each action would be equivalent, and as such, each action is equally likely to be selected for a given state.

Stochastic policies are often seen as advantageous over deterministic policies for the reason that deterministic policies do not encourage any exploration, instead, the agent will always follow the policy which can yield non-optimal results. Exploration from stochastic policy allows the agent to try different actions and learn from the different outcomes improving results in complex environments with uncertainty [12].

2.1.2 Value Functions and Bellman Equations

In order to assess the value of a state or a state with a provided action, value functions can be utilized. Before exploring the value functions, it is important to understand how the expected return is computed for a given policy, and how the optimal policy is to be selected. The expected return equation is shown in equation 2 [1].

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{2}$$

where $J(\pi)$ is the expected return under a given policy. It is determined by computing the product of the probability of a given trajectory τ under the policy π with the infinite-horizon discounted return which is shown in equation 3.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3}$$

where γ is the discount factor discounting future rewards, and r_t is the reward for a given step of the trajectory τ [1].

Now that a way of computing the expected reward has been determined, the optimal policy can be computed with equation 4.

$$\pi^* = \arg\max_\pi J(\pi) \tag{4}$$

where the policy π which generates the maximum expected return is selected.

To recursively express the optimal value received for a provided state, or a provided state and action pair, the Bellman Equations can be utilized, which are seen below in equations 5 and 6.

$$V^*(s) = \max_a \mathbb{E}_{s' \sim P} \left[r(s,a) + \gamma V^*(s') \right] \tag{5}$$

$$Q^*(s,a) = \mathbb{E}_{s' \sim P} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right] \tag{6}$$

Equation 5 describes the optimal value Bellman function, while equation 6 describes the optimal action-value Bellman function [1]. In the optimal value Bellman function, the value is computed by determining what action a provides a maximum reward for each possible next state that the agent could end up on next. It is important to note that the function is recursive in nature, as the Bellman function is called again while being discounted by the factor γ. The optimal action-value Bellman function is similar, but operates on the basis of being provided an initial state, and an initial action of which the optimal value is then determined.

2.2 Overview of Soft Actor-Critic Algorithm

The Soft Actor-Critic algorithm is a maximum entropy reinforcement learning framework that not only aims to maximize rewards, but also maximizes the entropy [11]. Soft actor-critic is an off-policy framework meaning it aims to reuse past experiences for training gradient steps avoiding poor sample efficiency challenges experienced with on-policy frameworks [11]. The entropy optimization component enables the algorithm to enable stability and exploration during the training process [11], which is critical for training an autonomous race car where exploration is required to determine the optimal paths of the racetrack for the fastest time.

2.2.1 Soft Actor-Critic Policy

Given that soft actor-critic is an entropy-regularized reinforcement learning, the optimal policy equation can be updated with an additional entropy term. The entropy is a function of the probability of an action occurring under a given policy, multiplied by the negative logarithm of the probability. The optimal policy under entropy-regularized reinforcement learning is shown in equation 7 [1].

$$\pi^* = \arg\max_\pi \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)) \right) \right] \tag{7}$$

where $H(\pi(\cdot|s_t))$ represents the entropy of the policy at state s_t, and α is the entropy regularization coefficient that determines the relative importance of the entropy term against the reward [1] encouraging exploration.

2.3 Simulation Platforms for Autonomous Race Cars

To safely and cost effectively train a reinforcement learning model, a simulation platform is required to allow the agent to explore the environment, and to develop an effective policy

for navigating a racetrack fast. Moreover, a simulation platform allows for repeatability, in addition to the ability to speed up training by modifying the simulation time factor. Currently, there are numerous different simulation platforms that would allow with some modifications the ability to train an autonomous race car. These platforms consist, but are not limited to the following:

- Gazebo Simulator
- BeamNG.tech
- F1TENTH Gym

2.3.1 Simulation Platform Overviews

Gazebo simulator is a widely used open-source robotics simulator. It is built around Robot Operating System (ROS) thus providing easy integration for ROS based projects and the ability to test them in simulation before implementing in the real world. Gazebo simulator has a strong community with lots of educational resources for learning development. However, resources regarding reinforcement learning are more challenging to come by. Additionally, a scenario for training the autonomous race car would have to be built from scratch.

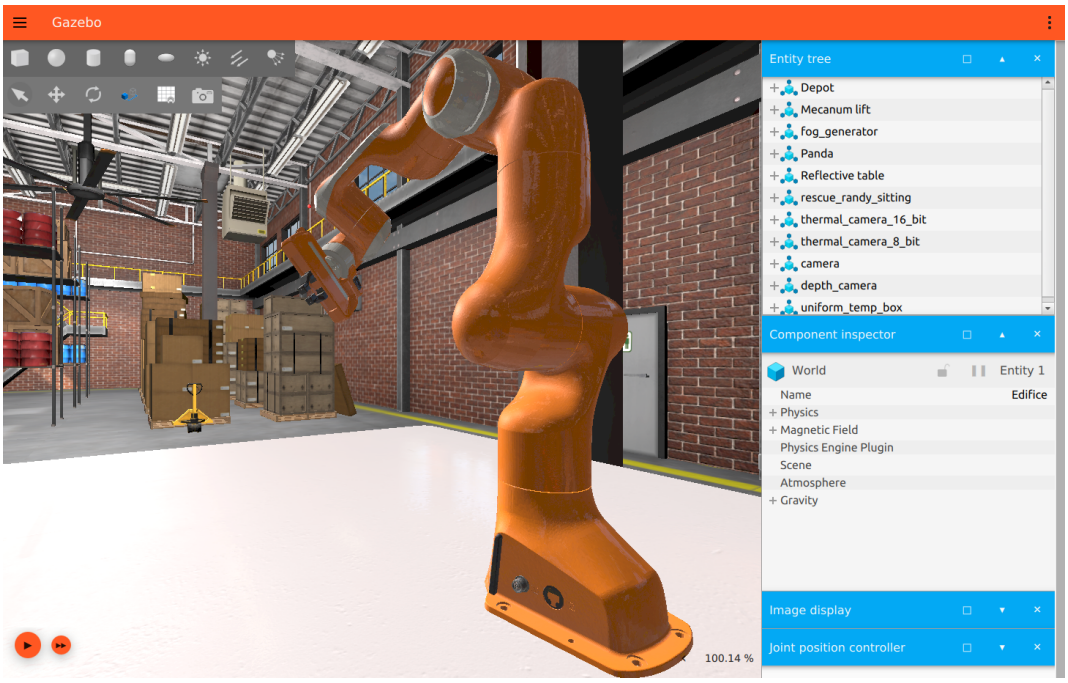

Figure 2.2: Gazebo Simulator [2].

BeamNG.tech is a simulator based on the car simulator BeamNG.drive with realistic soft-body physics [3]. Vehicles are simulated using a node-mass system where components are modeled with mass points connected by beams, providing highly accurate physics and realistic damage modeling [3]. The simulator offers comprehensive API support through

BeamNGPy library for autonomous vehicle testing and research. Moreover, BeamNG.tech has a sample Gymnasium wrapper repository demonstrating how to implement a gym for RL purposes offering a great starting point.

Figure 2.3: BeamNG.tech Simulator [3].

F1TENTH Gym is an open-source simulation environment specifically designed for small scale autonomous racing cars [13]. This environment provides a lightweight 2D simulator that models the dynamics of 1:10 scale RC cars, making it ideal for training and testing reinforcement learning algorithms for autonomous racing applications. The environment follows the OpenAI Gym interface standard, enabling integration with popular RL libraries.

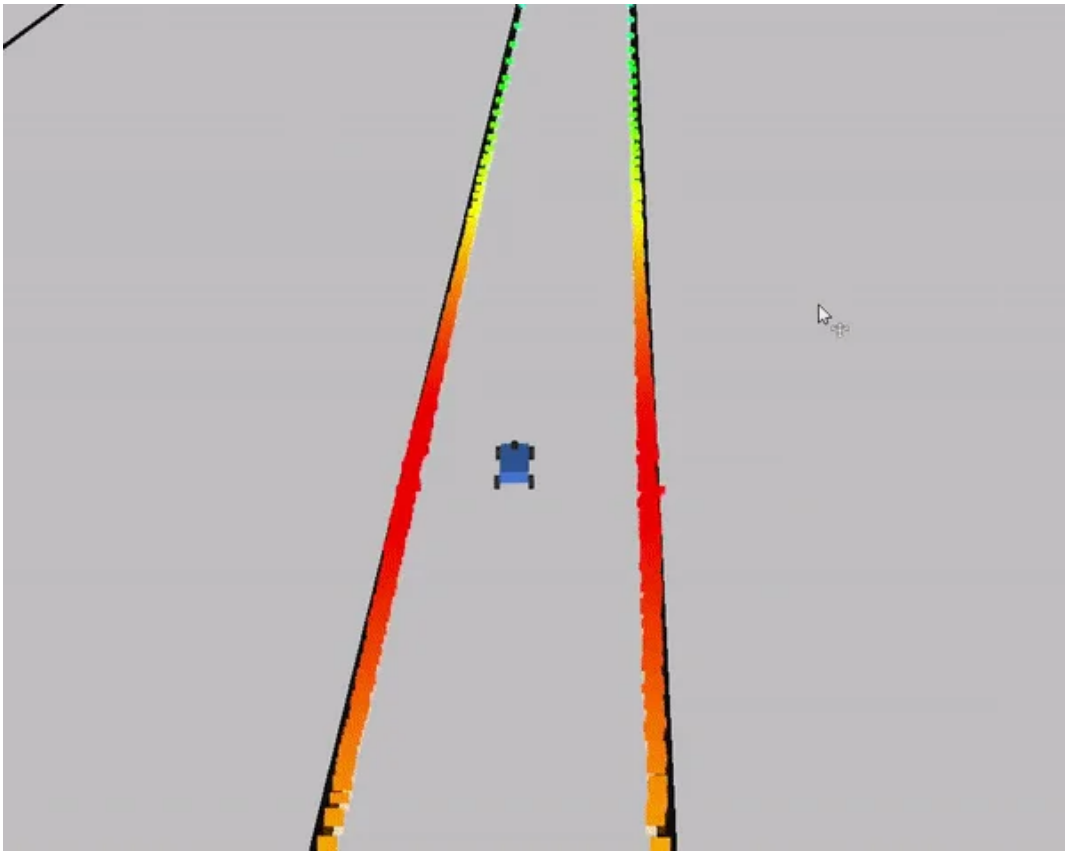

Figure 2.4: F1TENTH Simulator [4].

2.3.2 Selected Platform

While each of the simulators listed above provide a competitive starting point for training RL models, BeamNG.tech was selected as the simulation environment for this thesis. BeamNG.tech's easy integration with Python without having to utilize ROS in addition to the informative documentation for it made it an appealing choice over Gazebo simulator. Moreover, BeamNG.tech did not suffer from outdated libraries resulting in compatibility issues with Stable Baselines 3 that were experienced with the F1TENTH gym wrapper. BeamNG.tech also provides a Gymnasium wrapper coded in Python making customization and testing simple.

3 Methodology

3.1 Observation and Action Spaces

For the agent to be able to interact and interpret the environment it is in, an observation space and an action space are required. The observation space is approximately equivalent to the current state the agent is in, and provides information about the environment, while the action space provides the actions that are available for the agent to make. Each of these is described as a single one dimensional array, and the agent has no understanding of what

they actually represent, instead, it discovers over time what actions at given states maximize the reward.

3.1.1 Observations

The observation space for the autonomous race car agent consists of various sensor readings and state information that provide the agent with necessary data to make informed decisions. The following observations in table 1 are included in the observation space.

Table 1: The Observation Space

Name	**Observation space**	**Dimensions**
d_{lidar}	LiDAR distance measurements	$\mathbb{R}^m$
θ	Relative angle between vehicle orientation and track centerline	$\mathbb{R}^1$
θ_{elev}	Elevation angle of the vehicle	$\mathbb{R}^1$
v	Vehicle speed	$\mathbb{R}^1$
$v_{\text{centerline}}$	Centerline speed of the vehicle	$\mathbb{R}^1$
w_k	Wheel speed	$\mathbb{R}^1$
n_k	Rotation per minute (RPM) of the vehicle engine	$\mathbb{R}^1$
g_k	Gear index	$\mathbb{R}^1$
a	Throttle value	$\mathbb{R}^1$
b_k	Brake pressure	$\mathbb{R}^1$
s	Steering value	$\mathbb{R}^1$

In the observation space, the LiDAR distance measurements are at equal intervals from $-135°$ to $+135°$ ensuring there is vision of the racetrack along the sides and the front of the car, and the number of LiDAR rays is denoted by m. The LiDAR distance measurements are the primary means for the vehicle to navigate the track, relaying information to the agent about how far the car's center is from different parts of the racetrack.

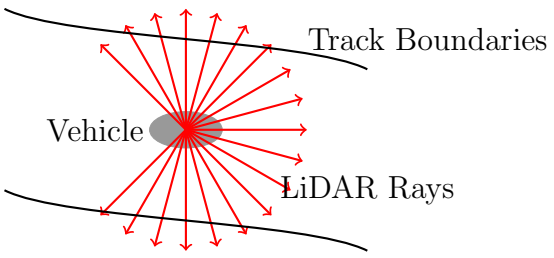

Figure 3.1: Visualization of LiDAR rays at equal intervals from $-135°$ to $+135°$.

The angles in the observation space consist of the elevation angle, and the relative angle of the vehicle. The elevation angle is given due to the nature of the racetrack having a variety

of different inclines. The inclines would affect the ability of the car to accelerate to certain speeds, in addition to brake appropriately before a turn. Providing this value gives the agent information to accommodate for those challenges. The relative angle between the vehicle's orientation and the track centerline is the only indicator for the car that it is traveling in the right direction to progress the race.

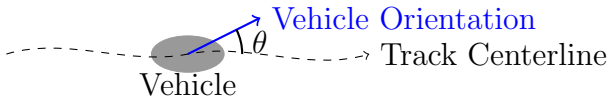

Figure 3.2: Vehicle relative angle from centerline.

When $\theta = 0$, the vehicle is perfectly aligned with the track centerline. Positive values indicate the vehicle is oriented to the left of the centerline (cartesian coordinate system is utilized), while negative values indicate orientation to the right.

There are two different rate values provided to the vehicle. These are the vehicle's speed, and the centerline speed of the vehicle. The vehicle's velocity is straight forward, it is the magnitude of the vehicle's velocity vector. The centerline speed is a metric to inform the agent about how rapidly it is progressing along the racetrack. The centerline speed is computed by projecting the vehicle's velocity vector on the tangent of the track's centerline. The centerline speed can be mathematically expressed as:

$$v_{centerline} = \vec{v} \cdot \hat{t} \tag{8}$$

where $\vec{v}$ is the vehicle's velocity vector and $\hat{t}$ is the unit tangent vector to the track's centerline at the point closest to the vehicle.

3.1.2 Actions

The action space for the autonomous race car agent consists of two continuous control variables that allow the agent to control the vehicle's movement. The following actions in table 2 are included in the action space.

Table 2: The Action Space

Name	**Action space**	**Dimensions**
s	Steering control (range: -1.0 to 1.0)	$\mathbb{R}^1$
t	Throttle/brake control (range: -1.0 to 1.0, where positive values represent throttle and negative values represent brake)	$\mathbb{R}^1$

The action space is defined as a continuous space with values ranging from -1.0 to 1.0 for both steering and throttle/brake controls. This is implemented in the Gymnasium environment wrapper as a Box space with shape (2,), allowing the agent to output precise control values for smooth vehicle operation.

3.2 Reward Function Design

The reward function utilized to train an agent is incredibly important for ensuring convergence upon the optimum, in addition to avoiding convergence upon local optimums and avoiding non-optimal results. The philosophy behind the reward function utilized in this thesis was to maintain simplicity in order to avoid rewarding actions that do not improve lap times, or punishing actions which do improve lap times. To better illustrate this, often times reward functions will punish deviations from the centerline of the racetrack proportional to the distance offset. This punishment can be beneficial in speeding up the training process as it encourages the race car to follow a path, which is the centerline of the track. However, such a punishment could create a non-optimal result. Typically, race cars stick to the inner edges of racetracks when driving around corners to reduce the distance that must be traveled by them, something that would be punished by such a reward. This can be seen in the figure of an optimal path of a race car around two curves.

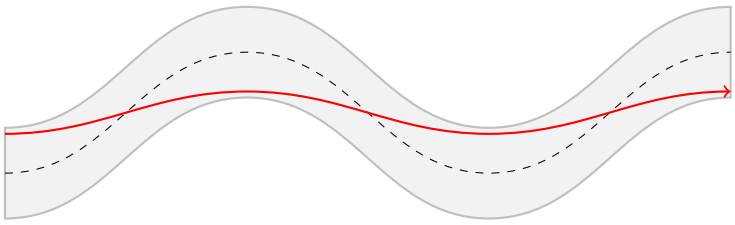

Figure 3.3: Optimal racing line (red) versus track centerline (dashed).

The reward function in this thesis was designed to work on any track length and configuration, focusing on three key components: progress rewards, steering rate penalties, and boundary violations.

3.2.1 Speed Reward

The main reward utilized for training the agent was a speed-based reward. A speed-based reward was selected instead of a time-based continuous punishment reward to avoid converging at a local optimum. Originally, experimentation was conducted with a time-based punishment, but it resulted in the agent crashing the car immediately to end the simulation as the punishment was greater than the reward, which was only attained at the end of the race.

The track speed reward utilized is intended to minimize the lap time of the race car. It was selected so that the race car could function on any length of a racetrack, as the reward is not related to the length of the track, but rather to the centerline velocity of the car. While rewarding proportionately for racetrack progress over each time step is a starting point, this reward function will not guarantee the race car attempts to complete the race as fast as possible, as integrating this reward function for the entire racetrack if it is always a constant length will result in the same reward. Therefore, an additional component was implemented, where the centerline velocity is squared, ensuring that the faster the vehicle progresses, the more reward is received to incentivize the agent to not only progress along the racetrack,

but to do so rapidly. If the agent progressed in the wrong direction, the reward was negated into a punishment.

Another crucial advantage of this reward function is that it provides continuous feedback at each time step, rather than only giving a reward upon completion of the race. This continuous signal helps the agent learn more effectively by immediately reinforcing desirable behaviors. Unlike a sparse reward that might only be given at the end of a successful lap, this approach ensures the agent receives meaningful feedback throughout training. Without such continuous signals, the agent would struggle to associate early actions with delayed outcomes, making the training process substantially more difficult, especially for complex tasks like navigating an entire racetrack.

The speed reward function can be mathematically expressed as:

$$r_{\text{speed}} = \begin{cases} \alpha \cdot (v_{\text{centerline}})^2, & \text{if } v_{\text{centerline}} > 0 \\ -\alpha \cdot (v_{\text{centerline}})^2, & \text{if } v_{\text{centerline}} < 0 \end{cases} \tag{9}$$

where $v_{\text{centerline}}$ represents the speed along the track centerline at time step t, and α is a positive scaling factor for both forward progress and backward movement penalties. The squared term for forward progress ensures that higher velocities along the centerline are rewarded disproportionately more, encouraging the agent to maximize its speed while maintaining progress of the track.

3.2.2 Steering Rate Punishment

The steering rate punishment was added to de-incentivize the agent from making aggressive turns at high velocities. This is to reduce the likelihood of the agent causing the car to swerve out of control or oscillate at high speeds. It is important to note that the addition of this punishment could result in the agent achieving a non-optimal result regarding minimization of lap times. However, it would not make sense for the car to achieve lower lap times by swerving on straight portions of the track at high speeds.

The steering rate punishment can be mathematically expressed as:

$$r_{\text{steer}} = -|r_{\text{speed}}| \cdot \beta \cdot \frac{\dot{s}}{\dot{s}_{\max}} \tag{10}$$

where r_{speed} is the speed reward, β is a scaling factor, $\dot{s}$ is the steering rate, and $\dot{s}_{\max}$ is the maximum steering rate. The punishment is linear with respect to the steering rate and proportional to the speed reward. This design ensures that the punishment never exceeds the speed reward, so there is always a positive incentive for the agent to progress along the track, which prevents convergence to local optima where the agent might stop moving to avoid punishment.

3.2.3 Boundary Punishments

The final component of the reward function is the boundary punishments. For simplicity purposes, the crash, out of bounds, wrong way, and out of time punishments will all be described by a singular equation. The crash punishment is self-explanatory, it is not ideal

to have a race car that recklessly collides with the walls of the racetrack, or drives in such a manner that damage is inflicted upon the vehicle. The boundary punishment prevents the vehicle from driving off the track, something that is punishable in real world racing. The wrong way punishment is there to speed up the training process, as driving the wrong way most certainly won't improve the lap time. Thus, for each of these conditions, the state of the vehicle is reset when the vehicle crashes, goes out of bounds, drives the wrong way, or runs out of time with the necessary punishment added.

The crash penalty, which serves as a base component for most boundary punishments, can be mathematically expressed as:

$$r_{\text{crash}} = \frac{|\theta|}{(\pi/2)} \cdot \gamma \cdot v^2 \tag{11}$$

where θ is the relative angle between the vehicle and the centerline, v is the vehicle speed, and γ is a speed scaling factor. This penalty scales with both the angle and velocity squared, ensuring that high-speed crashes or crashes at extreme angles result in more severe penalties.

An important feature of this punishment design is that it scales proportionally with the absolute value of the relative angle $|\theta|$. This means that head-on collisions (where the vehicle is perpendicular to the centerline, $|\theta| \approx \pi/2$) are penalized much more severely than glancing collisions or wall scrapes (where $|\theta|$ is small). While all contact with barriers is discouraged, this proportional punishment approach acknowledges that not all collisions have the same severity or implications for vehicle performance. The angle-based scaling ensures that catastrophic crashes receive appropriately harsh penalties, while more minor contacts, though still undesirable, are penalized less severely. Without this distinction, the agent might learn to maintain an overly cautious distance from walls, potentially resulting in sub-optimal lap times.

Additionally, the quadratic scaling with velocity (v^2) in the crash penalty is crucial for preventing a local optimum problem. Without this scaling, the agent might learn that the accumulated rewards from high-speed travel outweigh a fixed crash penalty, leading it to barrel into sharp turns at maximum speed rather than slowing down appropriately. By making the penalty proportional to velocity squared, high-speed crashes become expensive in terms of reward, effectively teaching the agent that proper braking before turns is necessary. This design ensures that the punishment for a high-speed head-on collision exceeds any temporary reward gained from the preceding straight-line acceleration, preventing the agent from converging on dangerous driving strategies that might appear optimal in the short term but are catastrophic for overall performance.

The complete boundary punishment system is implemented as a series of conditions that trigger episode termination with appropriate penalties:

$$r_{\text{damage}} = \delta + r_{\text{crash}} \qquad \text{if damage} > \text{damage}_{\text{max}} \tag{12}$$

$$r_{\text{bounds}} = \beta + r_{\text{crash}} \qquad \text{if vehicle position} \notin \text{track polygon} \tag{13}$$

$$r_{\text{wrong-way}} = \omega + r_{\text{crash}} \qquad \text{if } v_{\text{centerline}} < -5 \tag{14}$$

$$r_{\text{timeout}} = \tau \qquad \text{if remaining time} \leq 0 \tag{15}$$

where δ, β, ω, and τ are the base penalty values for damage, out of bounds, wrong way, and timeout conditions respectively. Each of these boundary violations ends the training episode

with the exception of the timeout condition, which is treated differently as it indicates the agent simply took too long rather than making a catastrophic error.

3.3 RL Training System

The process for training a model is also of importance just like the reward functions and the action/observation spaces. The following techniques of randomization, and training increments were utilized to ensure a robust results throughout the training process.

3.3.1 Randomization

When training an RL agent, exploration is often beneficial for allowing the agent to learn more about its environment. This exploration can be introduced through stochastic policy [12] where there is the probability that an agent takes non-optimal actions with the means of learning more about the environment. These actions can allow the agent to discover better trajectories that bring forth a greater optimum value.

Given that a racetrack is comprised of numerous different path types, from large gradual turns to sharper corners to long straightaways, an agent has to develop a policy that works for all of these to successfully navigate the entire racetrack. If an agent always starts at the same location with the same heading, the agent will develop a policy that works well for that specific region, but struggles when introduced to others. For example, imagine the agent starts training continuously on a long straightaway, the ideal policy for a straightaway is to go as fast as possible, as there is minimal need to steer other than for steering corrections thus not requiring low speeds. However, issues arise once the agent is required to complete a turn. Under the current policy, it is going to attempt to travel at a speed as fast as it possibly can, which would result in the agent crashing the car. The optimal solution for the agent is to hit the brakes before entering the turn, and to gradually start turning the car into the turn, the issue however is that such a process requires a lot of trial and error to achieve, as it requires the agent to sequentially perform actions prior to the turn. As a result, training times could be excessive for such a scenario, or the agent might learn a local optimum, where increasing speeds increases rewards, leading the agent to crash into the boundaries at even higher speeds.

To combat the challenge described above, randomization of the track starting location can be utilized. This would allow the agent to learn during early training phases different geometry of the racetrack, exposing it to turns and straightaways improving its ability to progress along the racetrack.

Another element of randomization could be added for the starting state of the vehicle. To further enhance the exploration and develop policy for different situations, the lateral position on the racetrack could be randomized, in addition to the initial heading of the race car so long as it is within the boundaries of the racetrack, and does not direct the race car into a position where there are no future moves possible.

3.3.2 Training Curriculum

Training an agent to successfully navigate an entire racetrack can be a challenging endeavour alone. Environments with variations that affect the difficulty of the task can require a training system that allows the agent to progressively acquire the required skills it needs [14]. The described approach is referred to Curriculum Learning where a series of custom designed curriculum are utilized to train the agent [14]. For training the agent to navigate a car around the racetrack as fast as possible, the training process was broken down into a series of steps defined by the percentage of the racetrack the agent would be required to navigate.

Table 3 outlines the curriculum learning approach used in this project, showing how the agent was trained incrementally on increasing portions of the track before graduating to the full track.

Table 3: Training Curriculum Stages

Stage	**Track Percentage**	**Training Steps**	**Vehicle**
0	5%	50,000	ETK 800
1	10%	50,000	ETK 800
2	30%	50,000	ETK 800
3	50%	50,000	ETK 800
4	70%	50,000	ETK 800
5	90%	50,000	ETK 800
6	100%	50,000	ETK 800
7	100%	50,000	ETK 800

The training begins with only 5% of the track, allowing the agent to quickly learn basic controls in a simplified environment. As the agent's performance improves, the track percentage gradually increases until the agent is training on the complete track. The final stages consist of additional training on the full track to refine the agent's policy and improve performance. This approach ensures that the agent progressively builds skills required for successful navigation, starting with basic control and an introduction to the different geometry of the track, which is further expanded in the later stages as the agent is exposed to all the parts of the track in one lap.

4 Implementation

4.1 System Configuration and Setup

The development and training environment for this research project consisted of specific hardware and software configurations to ensure optimal performance for reinforcement learning tasks. Table 4 summarizes the key components of the system setup.

Table 4: System Configuration

Component	Specification
Hardware	Intel Core i7-11700, 32GB RAM, NVIDIA RTX 3060 Ti (8GB VRAM)
Operating System	Windows 11
Simulation Environment	BeamNG.tech v0.34
CUDA Version	CUDA 11.7 with cuDNN 8.5
Python Version	Python 3.11.4
Key Libraries	PyTorch, Stable Baselines3, Gymnasium, NumPy

The hardware configuration accommodated the computational demands of both the physics-based simulation and reinforcement learning training. The NVIDIA GPU was essential for accelerating training during experimentation with different architectures and hyperparameters.

BeamNG.tech served as the simulation environment due to its realistic physics engine and vehicle dynamics. The simulator runs at 2000 physics steps per second (0.0005 seconds per step), while the reinforcement learning agent was configured to interact with the environment at 5 Hz (0.2 seconds per step).

CUDA and cuDNN were installed to enable GPU acceleration for the deep learning components. The software stack was built on Python 3.11.4, chosen for compatibility with Stable Baselines3, which implemented the SAC algorithm and was compatible with the Gymnasium API. A virtual environment using Python venv managed dependencies and ensured a consistent development environment.

4.2 Gymnasium Environment Wrapper

For the implementation of the Gymnasium environment wrapper that would be utilized to train the agent, BeamNG.tech's BeamNG.gym repository [15] was forked and modified to fit the needs of this thesis. The primary environment class, `WCARaceGeometry`, was developed to provide a standardized interface between the BeamNG.tech simulator and the reinforcement learning framework, following the Gymnasium API conventions.

4.2.1 Environment Architecture

The environment wrapper consists of several key components as illustrated in Figure 4.1. At its core, the wrapper manages the communication between the simulator and the reinforcement learning agent, handling observation collection, action application, and reward calculation.

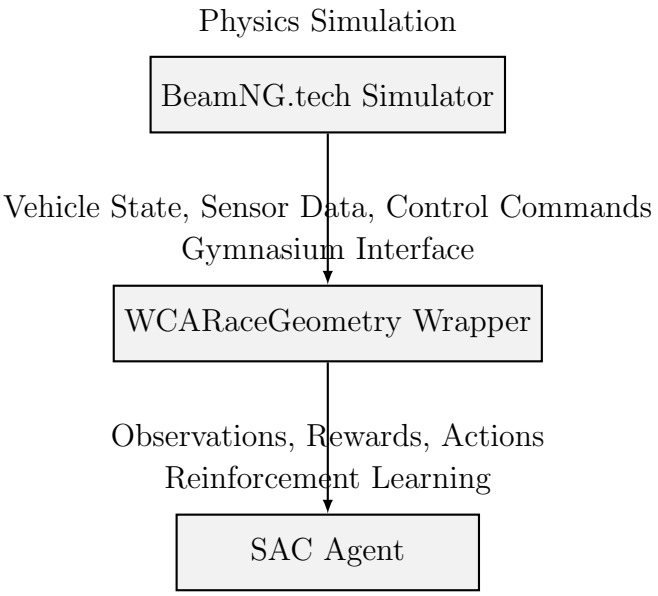

Figure 4.1: Implementation architecture of the wrapper for BeamNG.tech.

4.2.2 Dataclasses for Configuration

To maintain a clean and modular design, the environment uses Python dataclasses to encapsulate related settings:

- `LidarSettings`: Configures the simulated LiDAR sensor parameters, including ray angles, number of rays, and maximum detection distance.
- `RewardSettings`: Stores parameters related to the reward function, including various penalty values and scaling factors for the speed and steering components.
- `EnvironmentState`: Tracks the current state of the environment, including vehicle position, orientation, velocity, and sensor readings.

4.2.3 Simulation Environment

The environment wrapper utilizes the racetrack located in BeamNG.tech's West Coast USA level. The West Coast USA map is a 2048 x 2048 meter environment [5], but for this thesis, only the dedicated racetrack portion was used. This racetrack features a mix of challenging corners, elevation changes, and straightaways, making it ideal for training and evaluating the reinforcement learning agent.

Figure 4.2: Racetrack in West Coast USA used for training in BeamNG.tech [5].

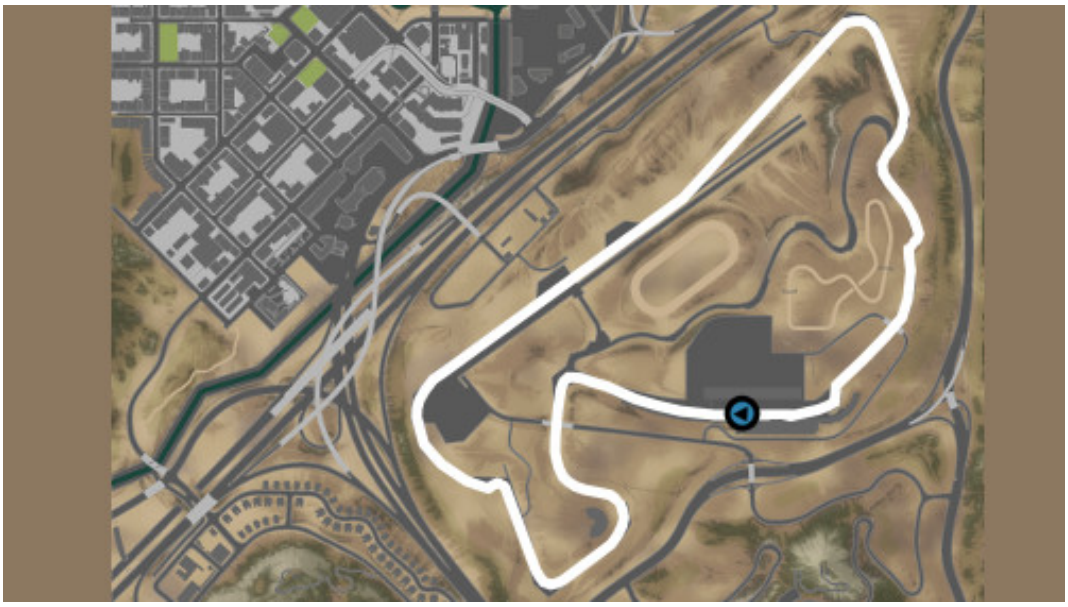

Figure 4.3: Track layout showing the shape and key features of the West Coast USA racetrack.

Multiple vehicle models were available for testing the agent's performance, with the primary development work conducted using the ETK 800 sedan seen in figure 4.4. This vehicle was selected for its balanced handling characteristics and moderate power output, providing a stable platform for agent training. Other available vehicles included the Bruckell LeGran, Bruckell Bastion, Gavril H-Series, Gavril T-Series, Wentward DT40L bus, and the high-performance ETK K-Series race car which were later used to evaluate the model.

Figure 4.4: The ETK 800 sedan used as the primary development vehicle.

4.2.4 Track Representation

A key component of the environment is the representation of the racetrack. The track is constructed using geometrical objects from the Shapely library, creating a precise mathematical model of the racing surface. Three primary components are used:

- `spine`: A `LinearRing` representing the centerline of the track, used for measuring progress and determining the vehicle's relative angle.
- `l_edge` and `r_edge`: `LinearRing` objects defining the left and right boundaries of the track.
- `polygon`: A `Polygon` representing the drivable area of the track, used to detect out-of-bounds conditions.

This geometric representation enables precise calculations of the vehicle's position relative to the track boundaries, which is essential for both reward calculation and providing meaningful observations to the agent.

4.2.5 Core Environment Methods

The environment wrapper implements the standard Gymnasium interface methods:

- `__init__`: Initializes the environment, configures the simulator, and sets up the racetrack representation.
- `reset`: Resets the environment to a starting state, with optional randomization of the vehicle's initial position and orientation.

- `step`: Applies an action to the environment, advances the simulation, and returns the resulting observation, reward, and termination information.
- `_get_obs`: Collects sensor data and environment state to construct the observation vector.
- `_get_reward`: Calculates the reward based on the current state, implementing the reward function described in Section 3.2.

4.2.6 Simulated Sensors

The environment implements several simulated sensors that provide information to the agent. The most complex of these is the LiDAR system, which casts rays from the vehicle's position to detect track boundaries. The implementation uses a ray-casting approach:

1. The vehicle's position and orientation are used as the origin for the rays.
2. Multiple rays are cast at angles ranging from -135° to +135° relative to the vehicle's heading.
3. Each ray is represented as a `LineString` and tested for intersection with the track boundaries.
4. The distance to the closest intersection point is calculated for each ray.
5. These distances form a significant portion of the observation vector provided to the agent.

This approach provides the agent with spatial awareness of the track boundaries, which is essential for learning to navigate the racetrack.

4.3 Soft Actor-Critic Implementation

The implementation of the Soft Actor-Critic (SAC) algorithm focuses on the technical aspects of training an agent with curriculum style learning. The training pipeline was implemented in Python using the Stable Baselines3 library and integrated with the BeamNG.tech simulation environment through the BeamNG.tech Gymnasium wrapper fork.

4.3.1 Training Pipeline Design

The training system follows a progressive approach with several key components:

- Stage-Based Progression – Training occurs in predefined stages, each with specific track percentages (from 5% to 100%) and fixed step counts (50,000 steps per stage).
- Knowledge Transfer – Models trained in earlier stages serve as starting points for later stages, enabling efficient learning of complex behaviors by building upon previously acquired skills.

- Environment Configuration – Each stage dynamically adjusts parameters like track percentage, with randomized starting positions enhancing the agent's ability to generalize.
- Resource Management – The system implements automatic GPU detection, model checkpointing, and comprehensive logging through TensorBoard to track training metrics.

The ETK 800 sedan was used as the primary training vehicle due to its balanced handling characteristics, though the system supports other vehicle models including the Bruckell LeGran, Bruckell Bastion, Gavril H-Series, Gavril T-Series, Wentward DT40L bus, and the high-performance ETK K-Series race car.

4.3.2 Training Workflow

The execution flow proceeds systematically through each stage:

1. A new SAC model with MlpPolicy architecture is initialized for the first stage (5% track length)
2. For subsequent stages, the previous stage's model is loaded to continue training
3. Each stage trains for 50,000 timesteps with environment parameters adjusted to the current track percentage
4. Trained models are saved with unique identifiers that include the stage, vehicle type, and track percentage
5. The system automatically detects completed stages to allow resuming training after interruptions

5 Results & Discussions

5.1 Training Results

The results of training with different lap percentages and building upon the previous model can be seen in the reward vs. step figures seen below. As is expected, the episode reward increased on average with incremental steps/number of episodes taken throughout the training process. Moreover, with each stage, the maximum reward would increase, which could be attributed for two main reasons, the primary reason being that the agent is capable of traveling a longer distance allowing for more reward to be accumulated, and the secondary reason being the agent is getting better at navigating the track at faster speeds.

The most notable figure is figure 5.1 where the agent is receiving negative rewards at the start. This indicates that the agent had not accumulated enough speed reward to offset the collision or time penalty received. During the early steps of the first stage training, the agent would momentarily converge onto either remaining completely stationary, or immediately

crashing into the walls yielding non-optimal results. After approximately 15,000 steps, the agent had developed a policy that started to yield improvements in the reward.

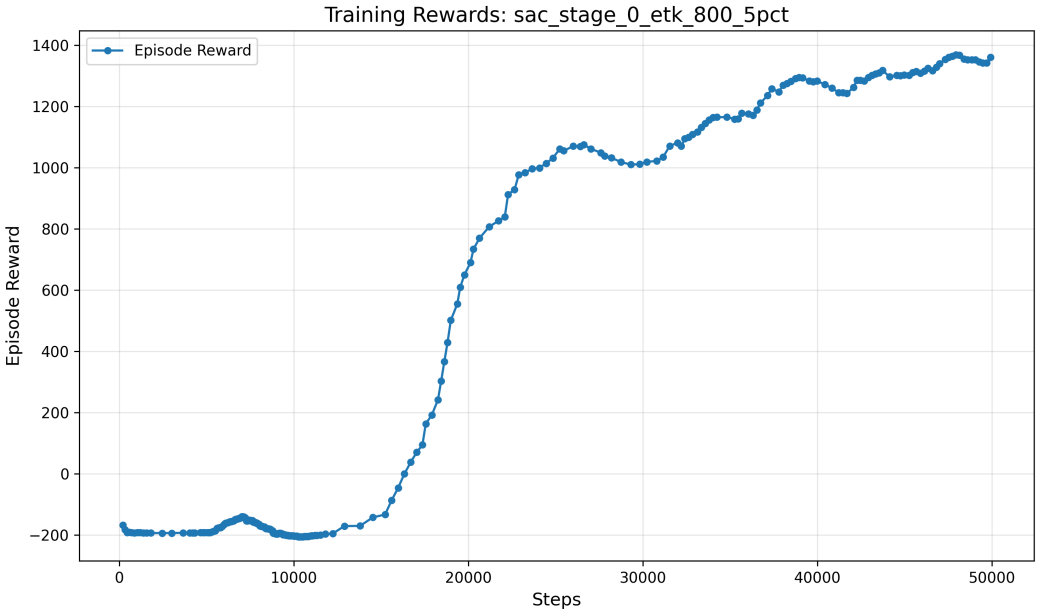

Figure 5.1: Stage 0 (5% track) training episode rewards.

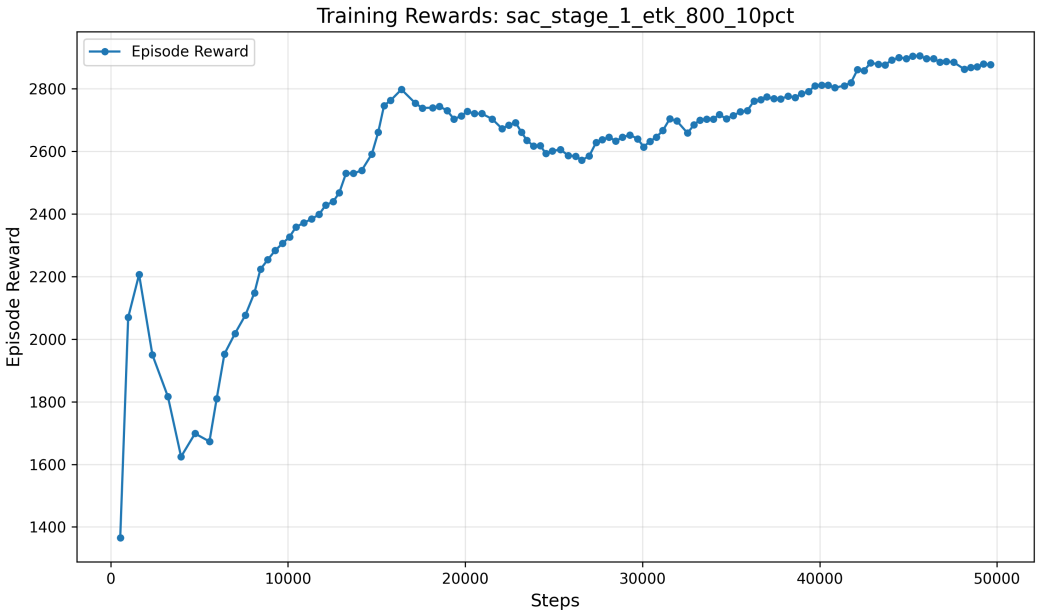

Figure 5.2: Stage 1 (10% track) training episode rewards.

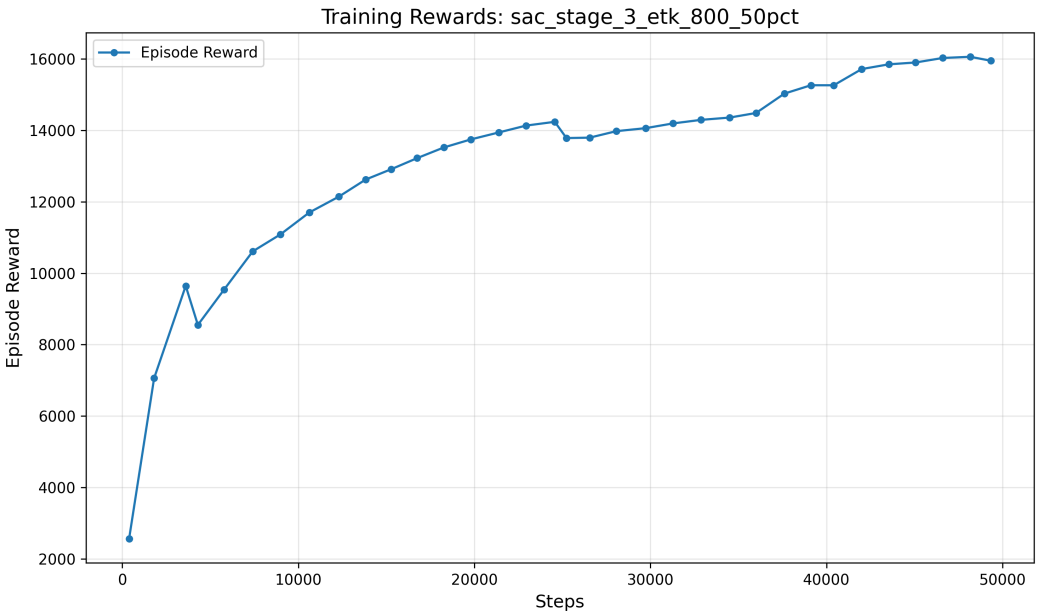

Figure 5.3: Stage 3 (50% track) training episode rewards.

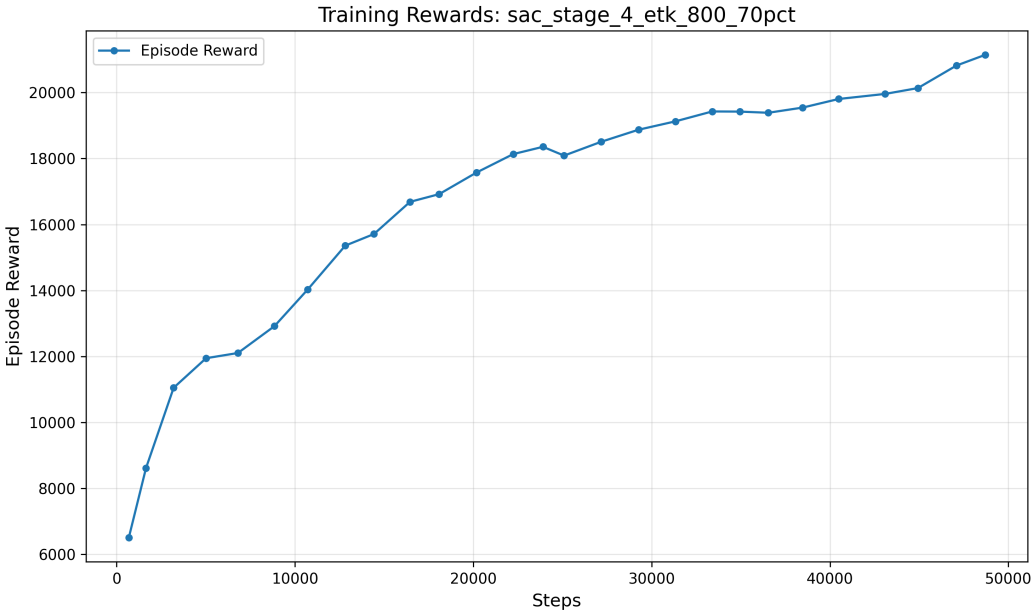

Figure 5.4: Stage 4 (70% track) training episode rewards.

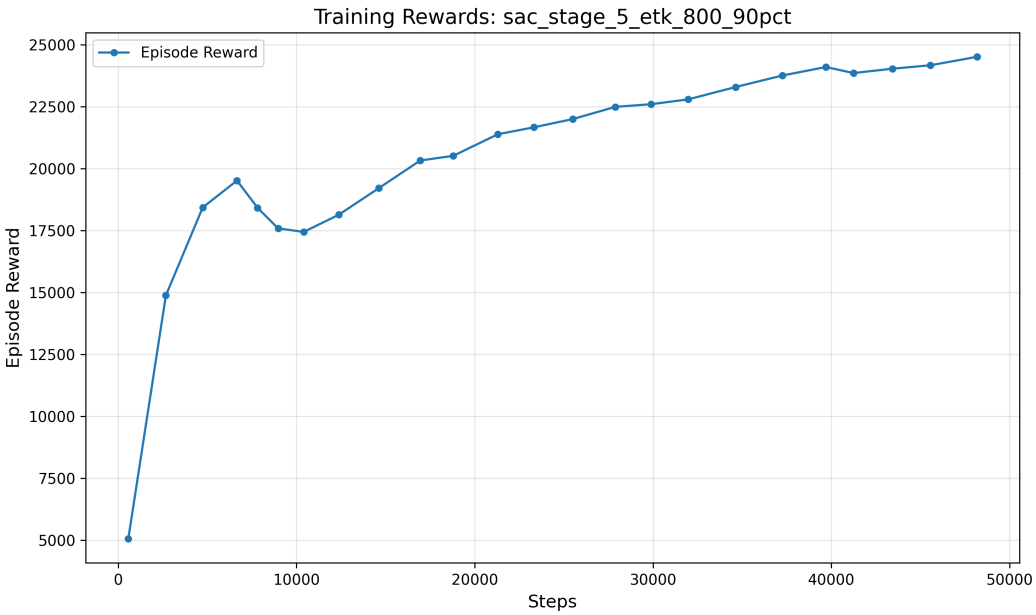

Figure 5.5: Stage 5 (90% track) training episode rewards.

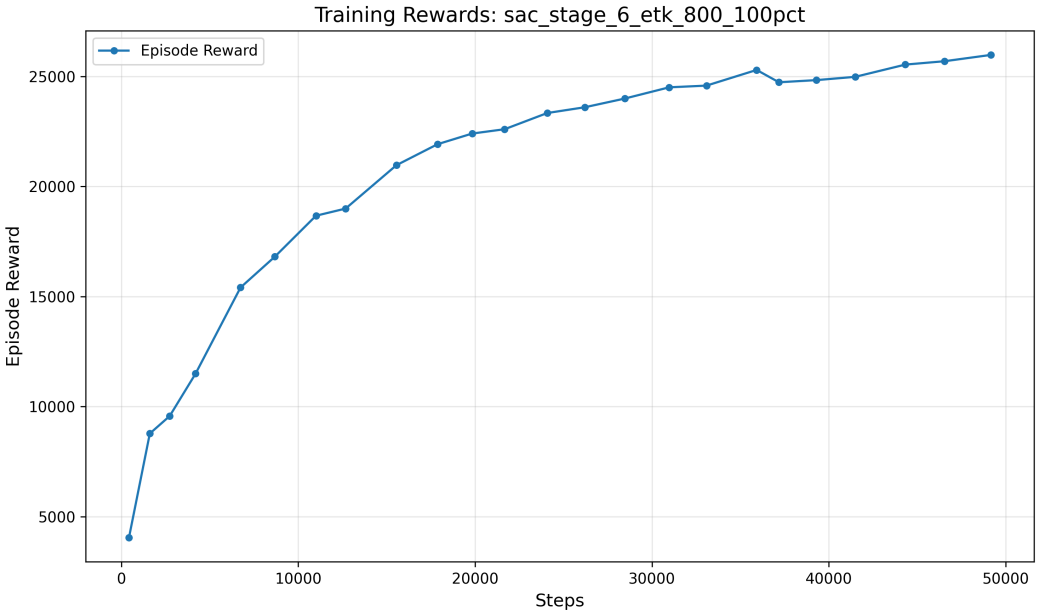

Figure 5.6: Stage 6 (100% track) training episode rewards.

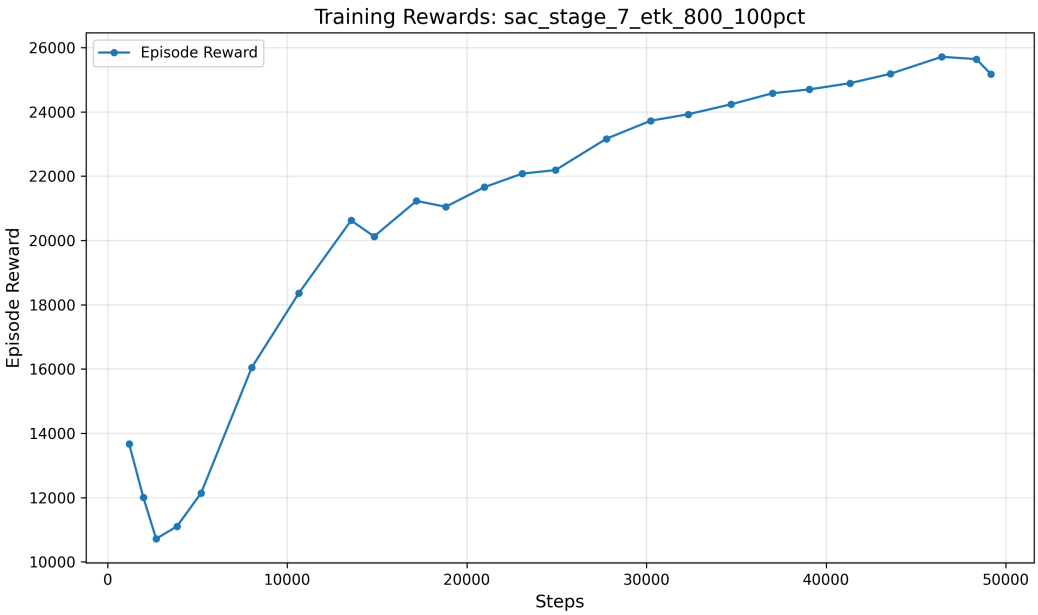

Figure 5.7: Stage 7 (100% track) training episode rewards.

5.2 Model Performance Evaluations

The performance of the trained models was evaluated by testing each model on the complete racetrack (100% length), regardless of the training track percentage. This approach provides insight into how well the models trained on different track lengths could generalize to the full track. The results measure both the completion rate of laps and the average lap times achieved by each model.

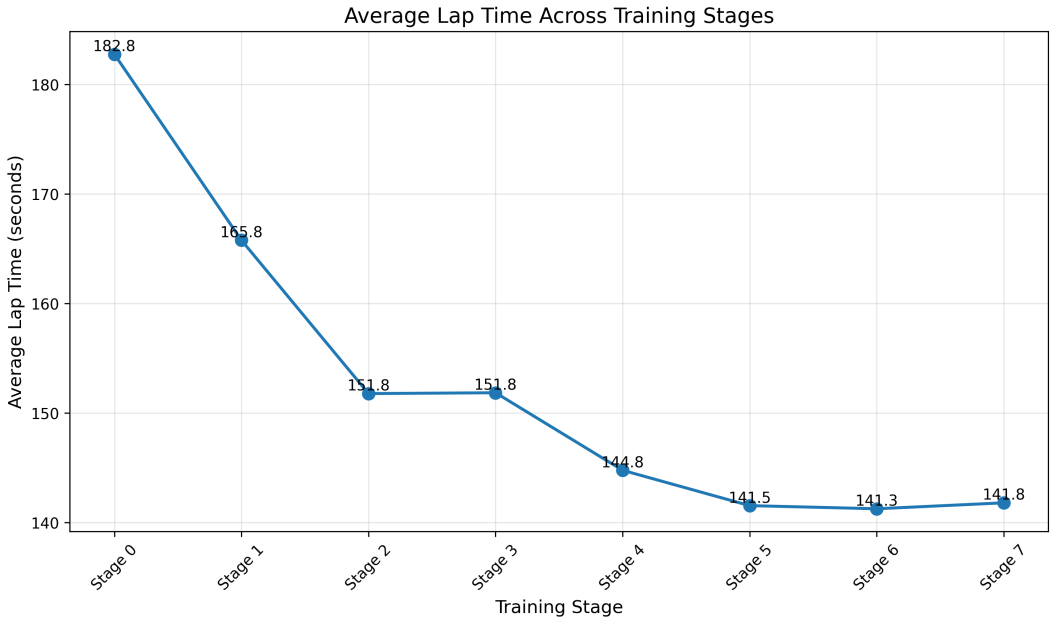

Figure 5.8: Average lap times across models trained on different track percentages.

In figure 5.8 shown above, the greatest improvement in reducing the lap times occurs between stages 0 and 3. Afterwards, the lap times begin to plateau, and finally start to increase after stage 6 indicating overall worse performance compared to stage 6. The lowest average lap time was determined to be approximately 141.3 seconds in total, which occurred at stage 6.

Table 5: Model Performance Metrics for Models Trained on Different Track Percentages

Model Stage	**Average Lap Percentage**	**Average Lap Time (s)**
0	1.000	182.8
1	0.850	165.8
2	0.643	151.8
3	0.445	151.8
4	1.000	144.8
5	1.000	141.5
6	1.000	141.3
7	0.745	141.8

In table 5, the average lap percentages achieved are summarized, in addition to the average lap times for complete race laps. The average lap percentage was shockingly high at 1.0 for the model trained at 5% of the racetrack indicating that the random starting provided sufficient information for the agent to learn to navigate the different components of the racetrack without ever having integrated them together. Notably, the average lap percentages

fluctuated a lot during the training process, indicating that the agent was crashing the car, or veering off the course during the testing process. This could be due to the model having learned a more aggressive policy to increase the rewards, which consequently increased the chances of it going out of the racetrack boundaries.

5.3 Path Comparisons

The performance of the trained models was evaluated by analyzing the paths taken by the SAC agent during testing across different training stages. The following figures illustrate the vehicle's trajectory when testing each model, from those trained on 5% to 100% of the track length, showing both complete track views and close-ups of the challenging chicane section to assess how well each model navigated the course.

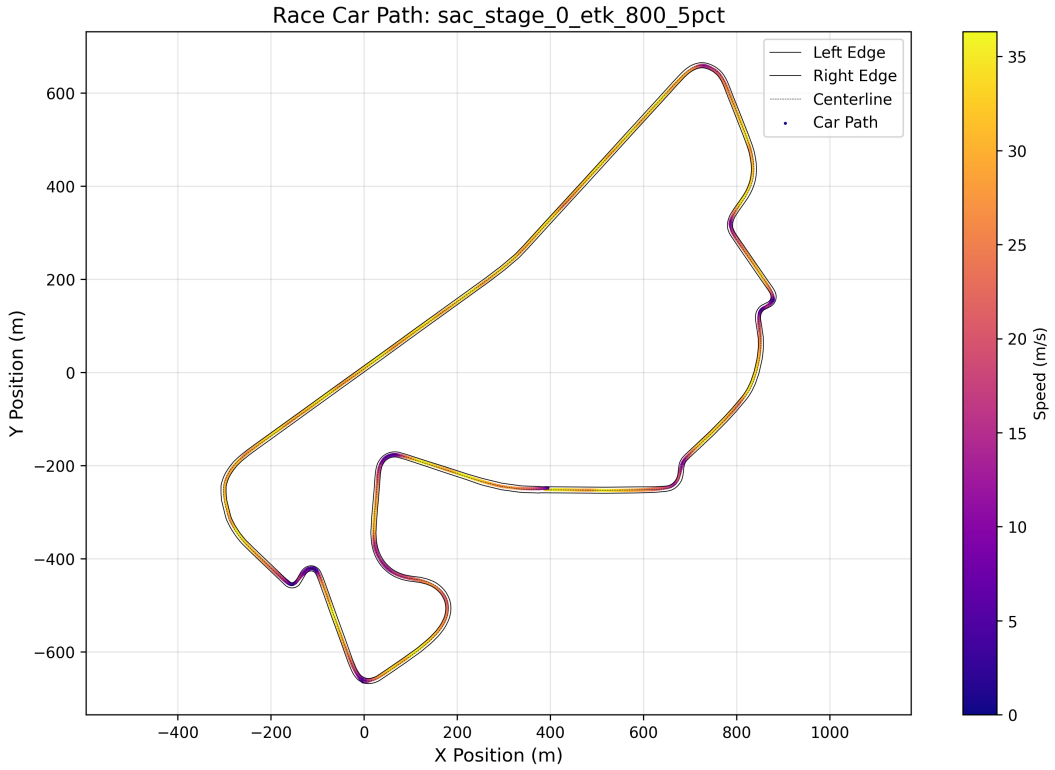

Figure 5.9: Vehicle path during stage 0 model testing.

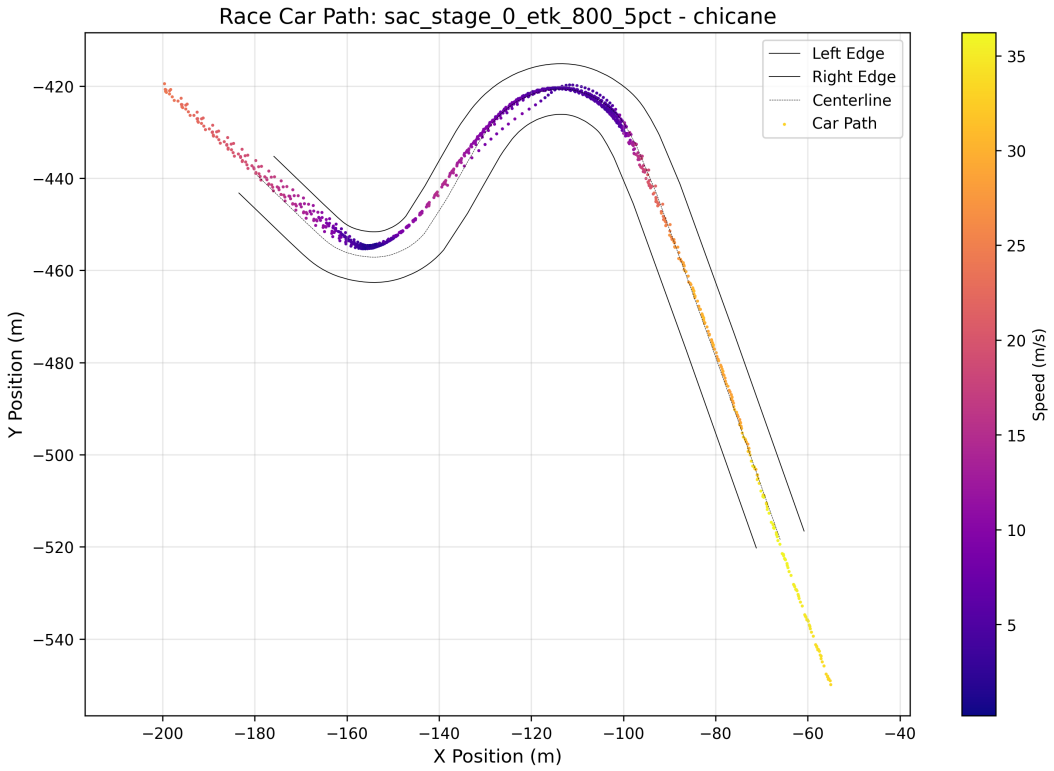

Figure 5.10: Vehicle path through chicane during stage 0 model testing.

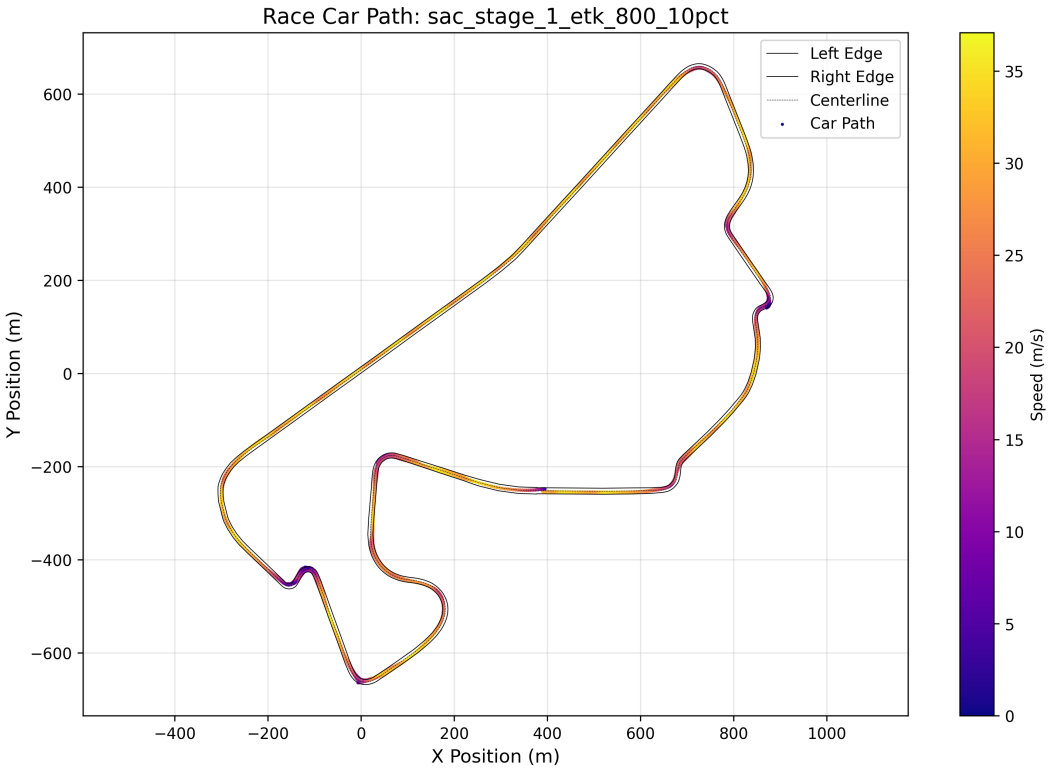

Figure 5.11: Vehicle path during stage 1 model testing.

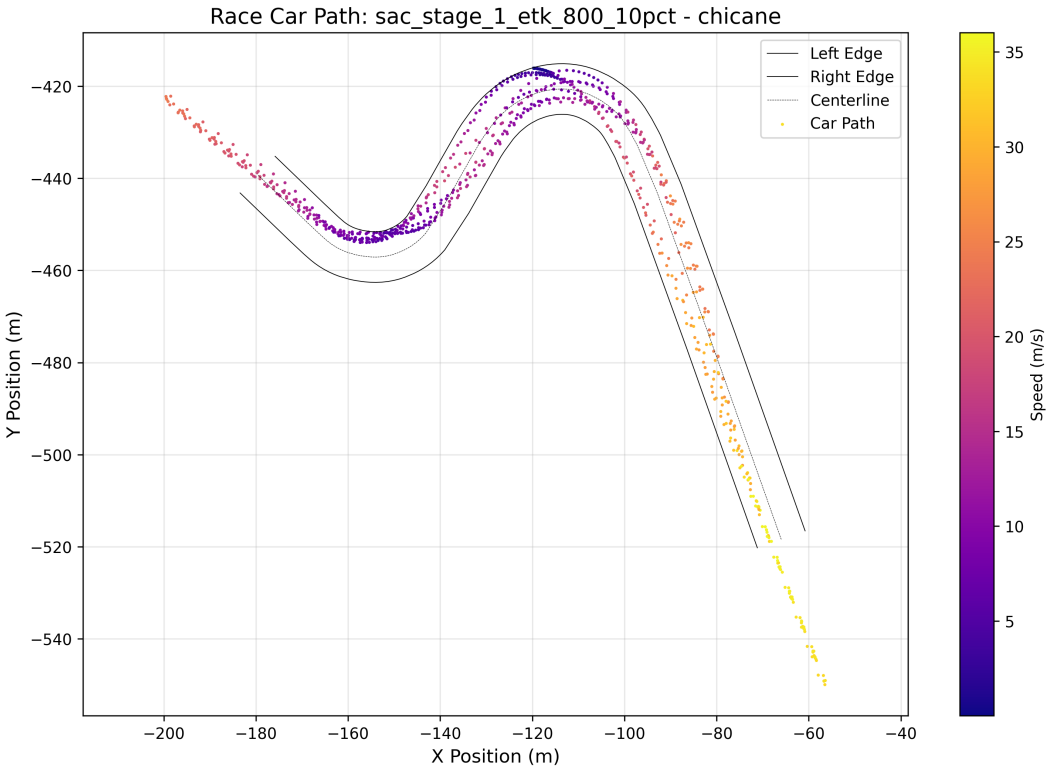

Figure 5.12: Vehicle path through chicane during stage 1 model testing.

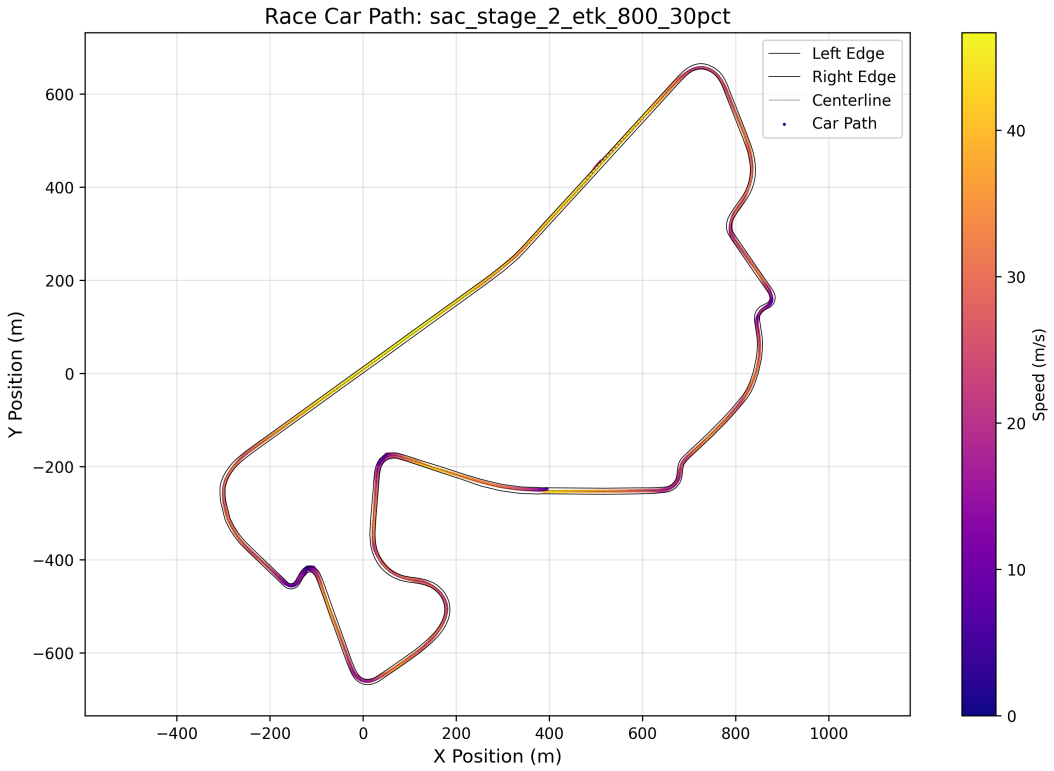

Figure 5.13: Vehicle path during stage 2 model testing.

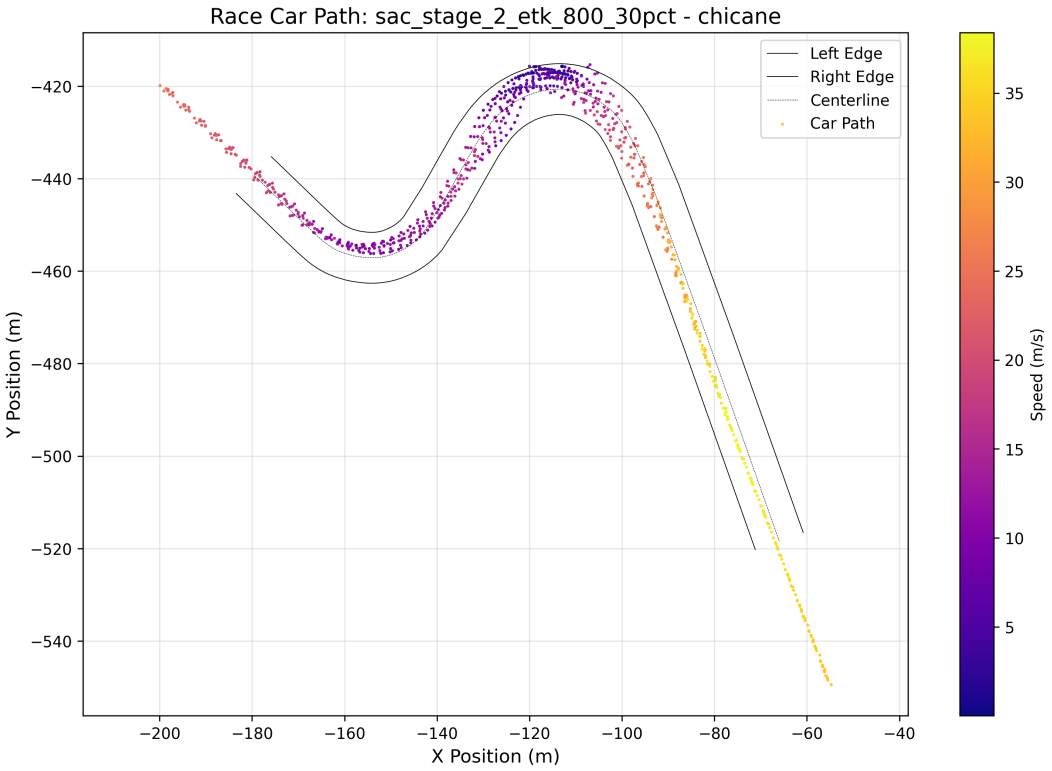

Figure 5.14: Vehicle path through chicane during stage 2 model testing.

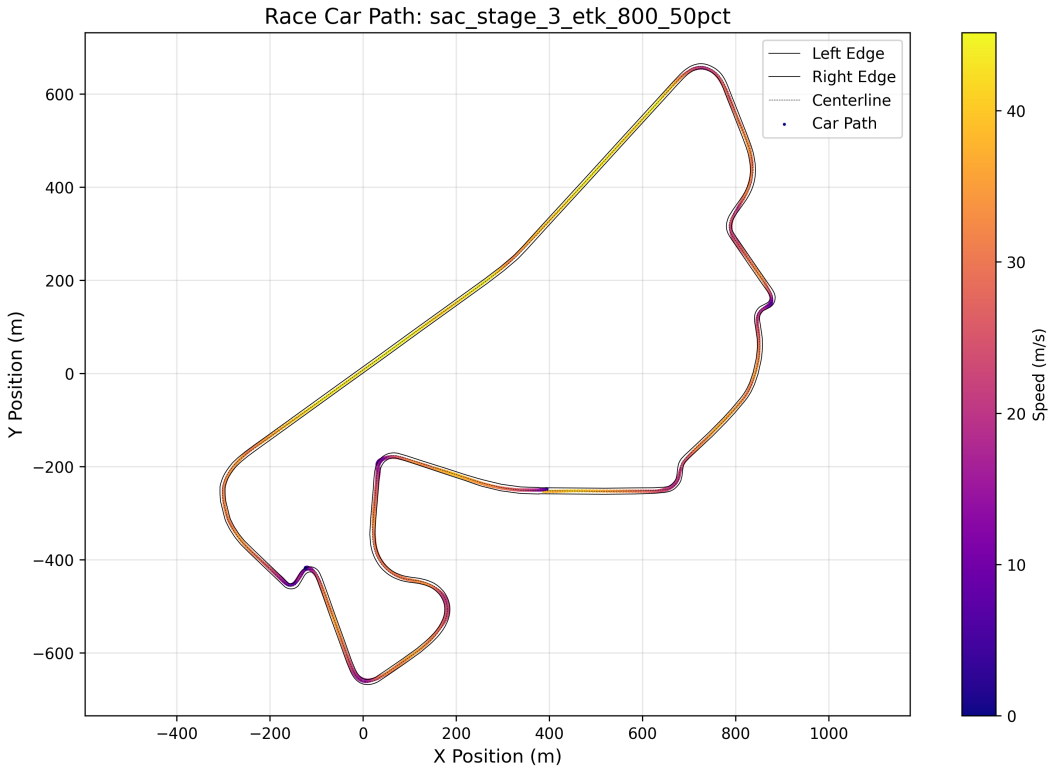

Figure 5.15: Vehicle path during stage 3 model testing.

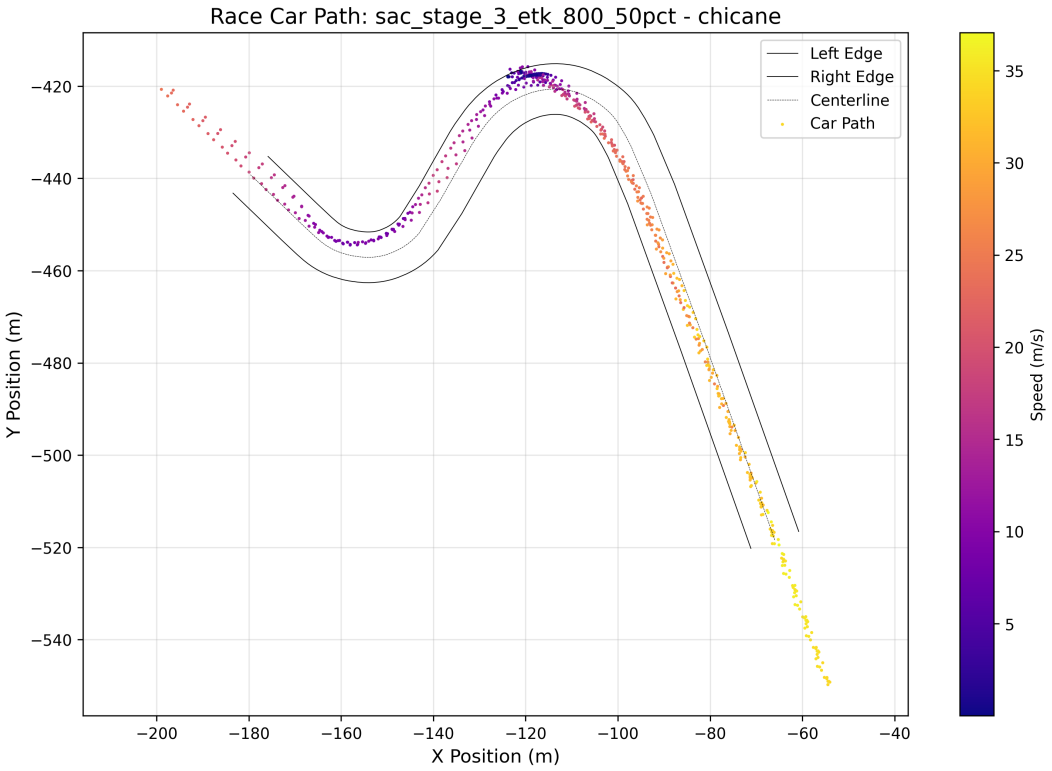

Figure 5.16: Vehicle path through chicane during stage 3 model testing.

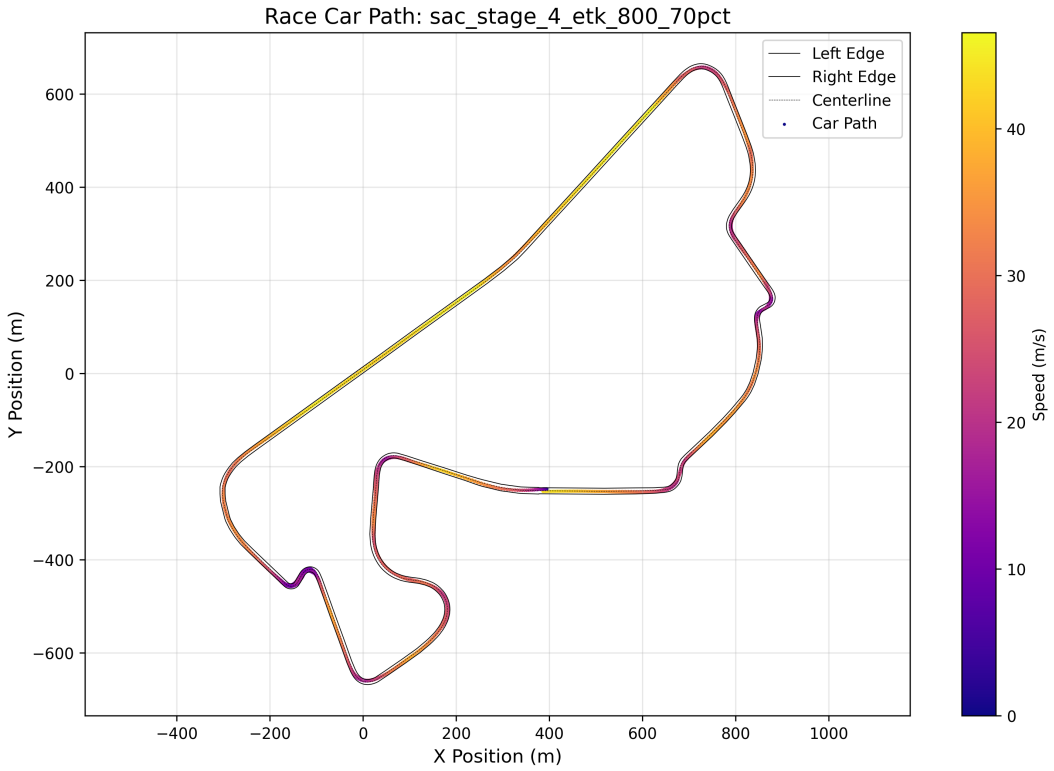

Figure 5.17: Vehicle path during stage 4 model testing.

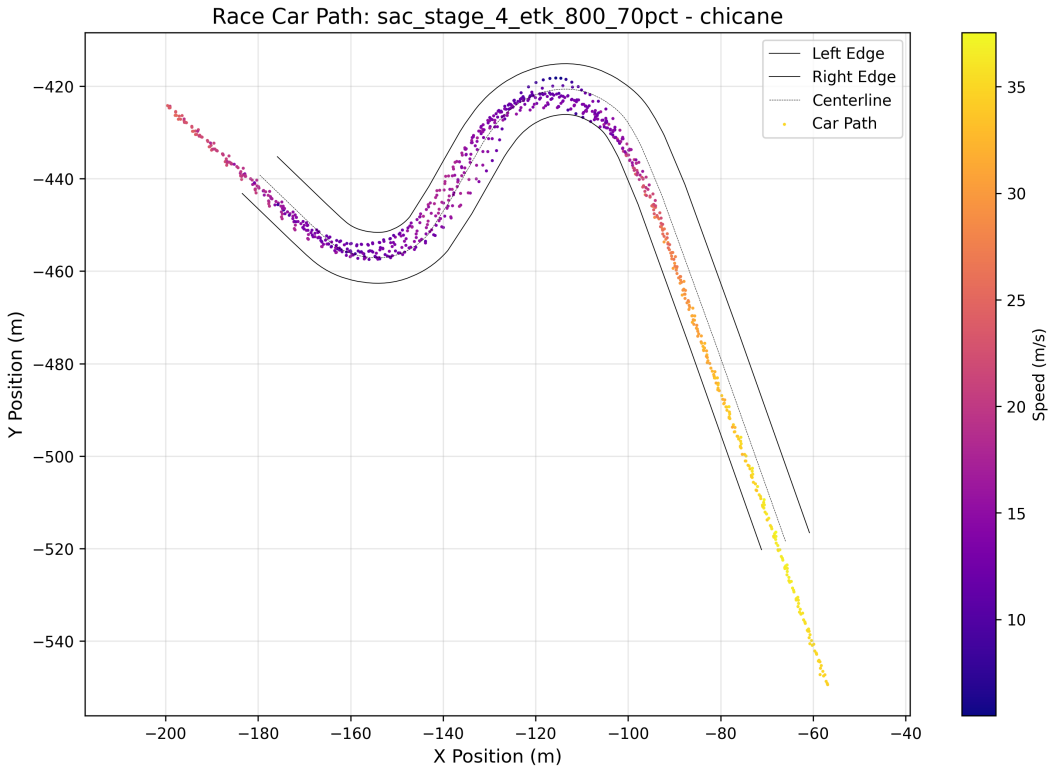

Figure 5.18: Vehicle path through chicane during stage 4 model testing.

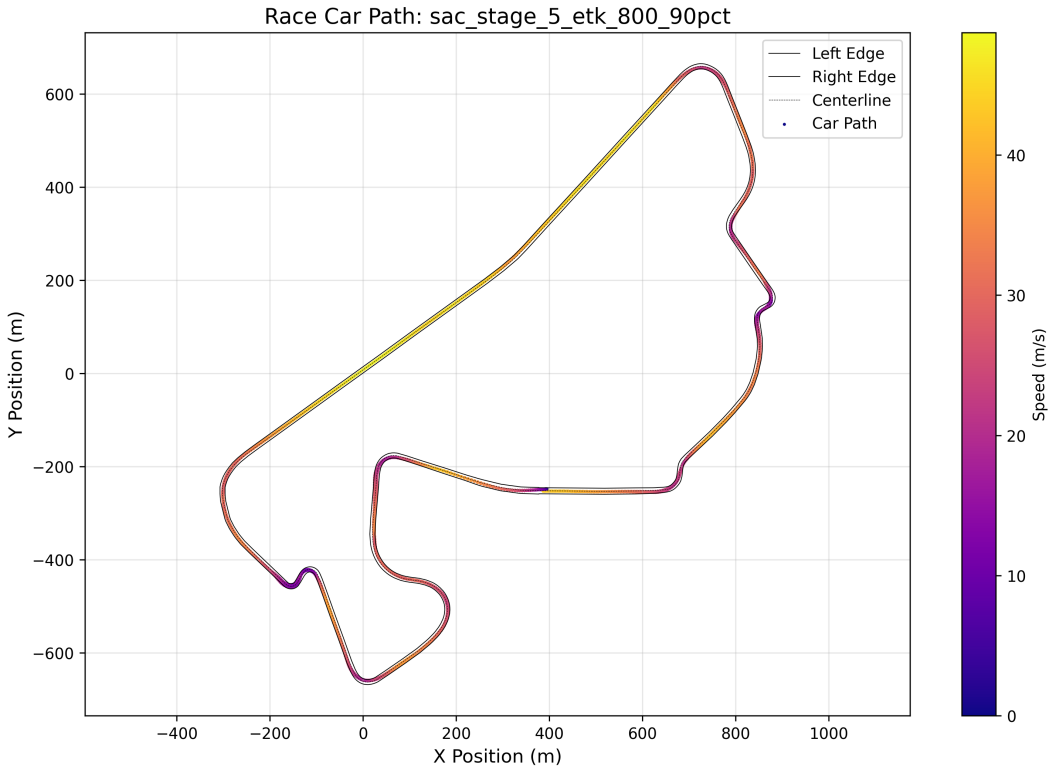

Figure 5.19: Vehicle path during stage 5 model testing.

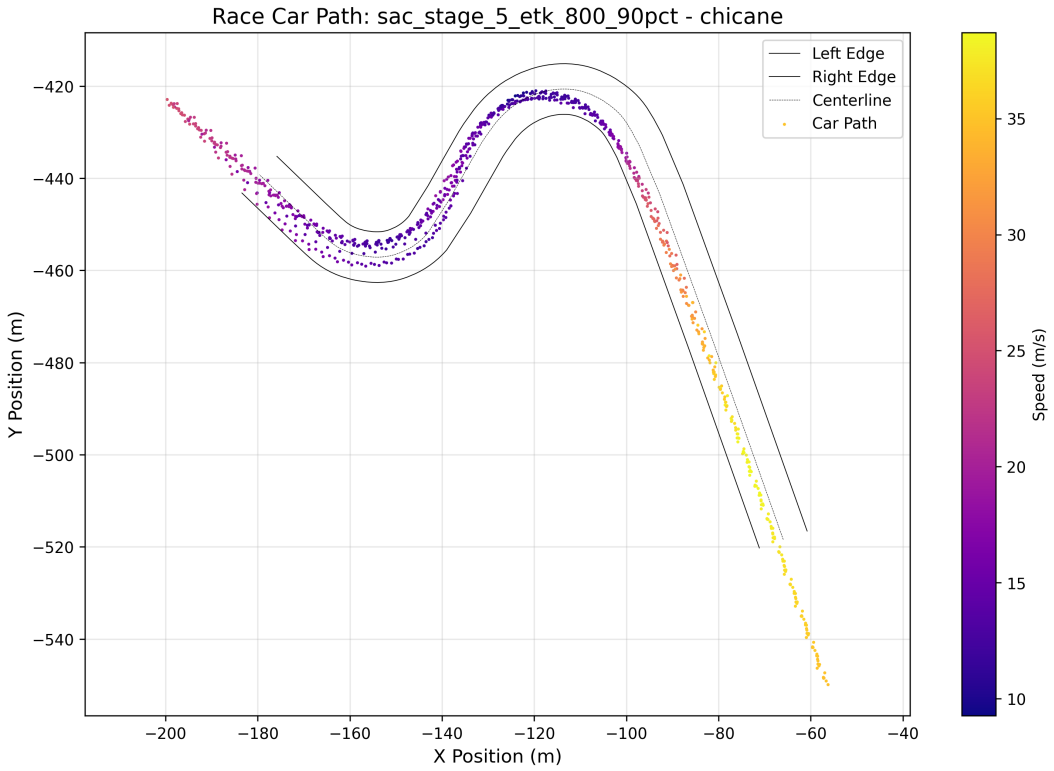

Figure 5.20: Vehicle path through chicane during stage 5 model testing.

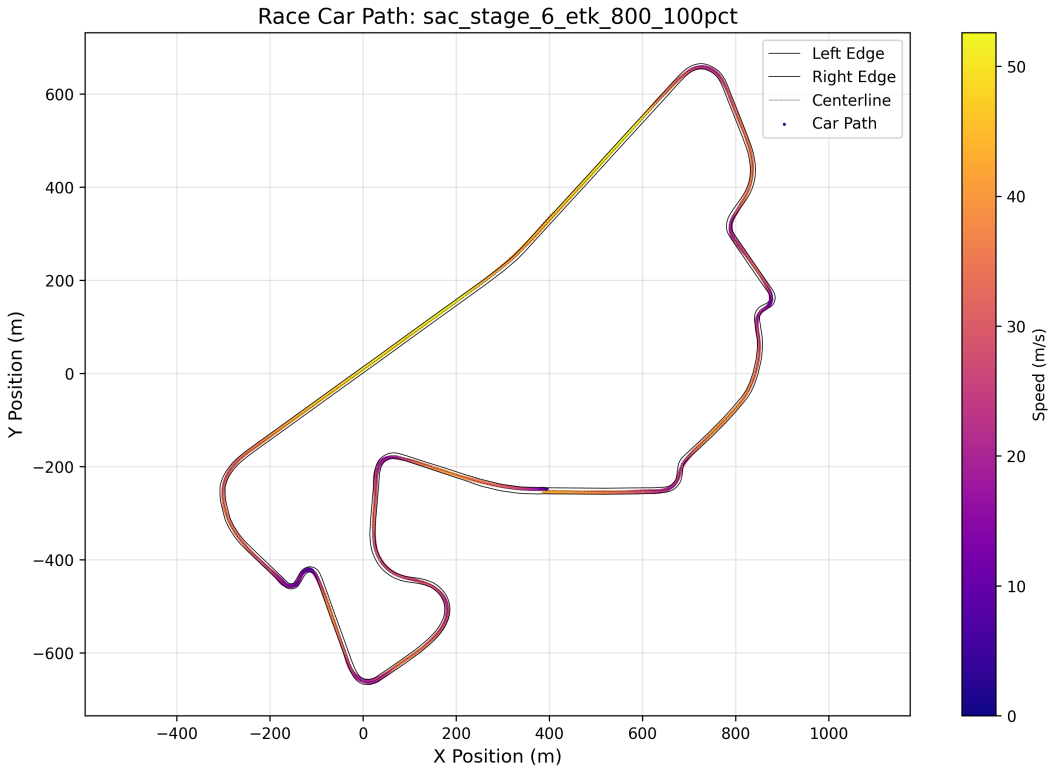

Figure 5.21: Vehicle path during stage 6 model testing.

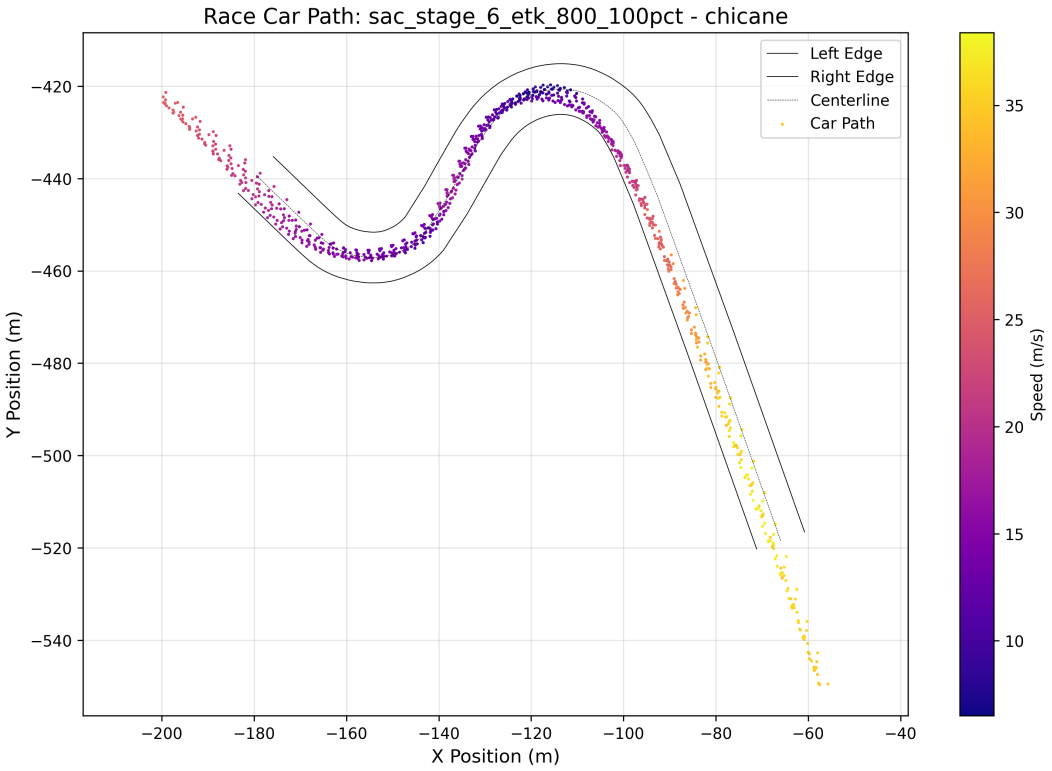

Figure 5.22: Vehicle path through chicane during stage 6 model testing.

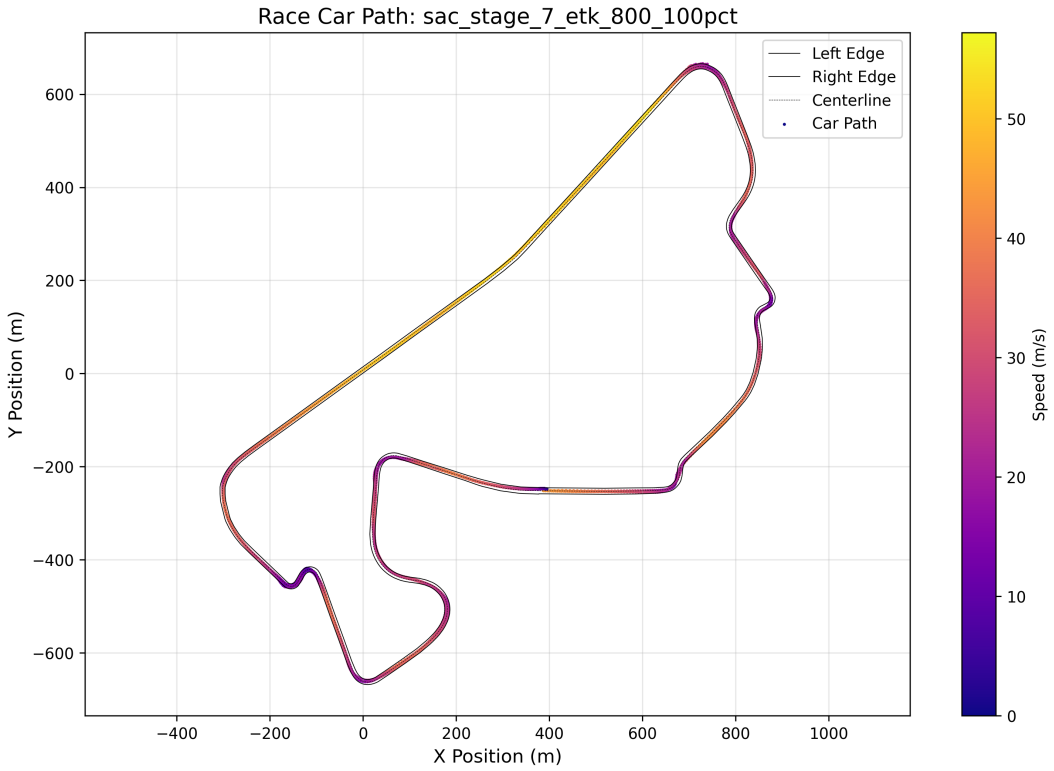

Figure 5.23: Vehicle path during stage 7 model testing.

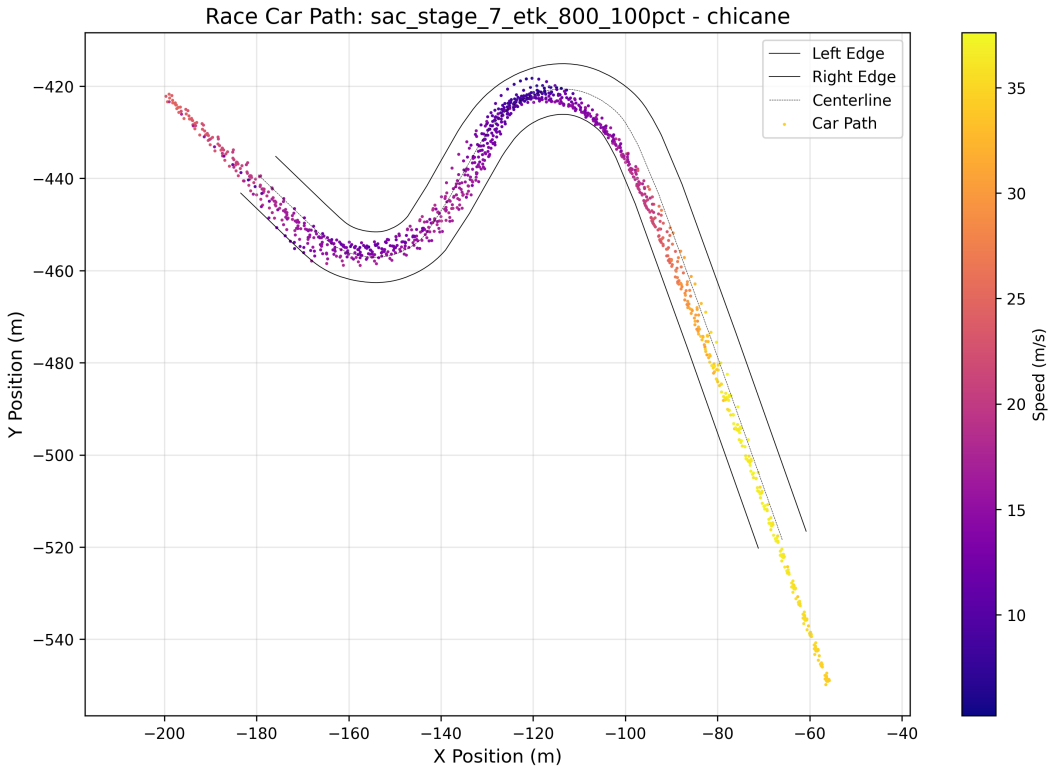

Figure 5.24: Vehicle path through chicane during stage 7 model testing.

From the figures shown above (figure 5.9 to figure 5.24), an understanding of how the agent navigated the courses can be gathered. Looking at the early training stages, it can be seen that on the straight regions of the track, the speed of the car would oscillate faster and slower as indicated by the change in colour on the straight sections. This behaviour was minimized as the agent learned to increase its speed due to the increased reward thus maintaining a constant high speed on the straight portions of the track, while slowing down for the curves as indicated by the darker colour regions. The full racetrack paths indicate that the agent maintained the highest speeds on the straightest parts of the track, while it had the lowest speeds on the sharpest corners of the track, or on the chicane curves where there are two subsequent turns in opposite directions.

The figures depicting the chicane turn of the racetrack were selected due to the complexity of such a turn. The very first stage was successful in the series of turns, but suffered from sharp cornering radii resulting in low speeds about the turns. As the stages progressed, the turns became smoother with larger turning radii allowing for higher speeds through the chicane turn. It can also be seen that a lot of the points end at the first turn of it, indicating the agent crashed or went out of bounds terminating the testing episode.

5.4 Vehicle Performance Comparisons

From the stage training models, the best performing stage was selected for evaluation with different vehicle types, this model was the sixth stage, which was trained on 100% of the

racetrack. This model was tested with 6 different vehicle types making up a diverse range of characteristics. The vehicles that it was tested on are as follows:

- Bruckell LeGran – A front-wheel drive sedan with balanced handling characteristics, representing a typical passenger vehicle with moderate performance capabilities.
- Bruckell Bastion – A high-performance muscle car with powerful acceleration and rear-wheel drive dynamics, optimized for straight-line speed rather than cornering.
- Gavril H-Series – A commercial van with significantly different handling characteristics from passenger vehicles, featuring a higher center of gravity and longer wheelbase.
- Gavril T-Series – A heavy-duty truck with substantial mass and unique handling dynamics, presenting challenges in maneuverability and braking.
- Wentward DT40L – A large city bus with considerable weight and length, featuring distinct handling characteristics due to its size and purpose-built design.
- ETK K-Series – A high-performance sports car with advanced handling capabilities, representing the pinnacle of performance among the test vehicles.

The results from testing the selected model with the different vehicles can be seen in table 6 below.

Table 6: Vehicle Performance Metrics for Models Trained on Different Vehicles

Vehicle	**Average Lap Percentage**	**Average Lap Time (s)**
Bruckell LeGran	0.115	N/A
Bruckell Bastion	1.000	141.5
Gavril H-Series	0.153	N/A
Gavril T-Series	0.015	N/A
Wentward DT40L	0.012	N/A
ETK K-Series	0.636	140.9

From the results seen in table 6, it can be seen that the performance across all vehicles was not incredible, with the exception of the Bruckell Bastion. The agent managed to achieve 100% lap completion with this car, in addition to an average lap time of 141.5 seconds which is almost identical to what the agent achieved with the ETK 800. The next best performing car was the ETK K-Series achieving an average lap percentage of 80% with an impressive average time for the completed laps of 140.9 seconds, surpassing the baseline achieved with the ETK 800. However, the other models which were significantly different such as the Gavril and Wentward vehicles which were much larger in size performed quite poorly, in addition to the Bruckell LeGran which was an older style car likely to have worse handling characteristics than the newer ETK 800 type car.

5.5 Challenges & Short-comings

Several challenges during the development and training of the SAC agent for autonomous racing were encountered. These challenges highlight the complexity of applying reinforcement learning to the domain of autonomous driving and provide valuable insights for future work in this area.

5.5.1 Reward Function Design Challenges

One of the most significant challenges faced was the design of an effective reward function. Early implementations included a constant time-based punishment, intended to encourage faster completion of the track. However, this approach led to an unexpected local optimum where the agent learned that crashing immediately was preferable to taking longer to complete the track. Since the accumulation of time-based penalties would eventually exceed any potential rewards for slow progress, the agent optimized for the "least bad" outcome by ending episodes quickly through deliberate crashes. The solution was to replace the time-based punishment with a speed-based reward that scaled quadratically with the centerline velocity, creating a continuous incentive for faster progress without penalizing the mere passage of time.

5.5.2 Transferring Skills Between Track Sections

Another significant challenge was the agent's difficulty in learning to brake before sharp turns when it had only been trained on straight sections of the track. When initially exposed to a turn after training exclusively on straightaways, the agent would consistently crash at high speeds as it had developed a policy optimized for maximizing speed on straight paths.

This observation motivated the implementation of randomized starting positions for training. By exposing the agent to different segments of the track during early training, including both straight sections and various types of turns, the agent could develop a more general policy capable of handling diverse track features. This approach significantly improved the agent's ability to anticipate the need for braking before sharp turns and adjust its speed accordingly.

5.5.3 Training Time Constraints

The extended training time required for reinforcement learning agents presented another practical challenge. Making changes to the environment, reward function, or model architecture required significant patience as evaluating these changes often took hours or days of training to yield meaningful results. This limitation made iterative development particularly challenging and time-consuming.

5.5.4 Observation Space Complexity

The design of an effective observation space presented additional challenges. Initial experiments included history queues to provide the agent with temporal information about past states, theoretically enabling better decision-making for actions requiring anticipation.

However, these expanded observation spaces dramatically increased the complexity of the learning task, causing the agent to struggle with identifying relevant patterns in the data.

Similarly, experiments with high-resolution LiDAR arrays (using many rays) proved problematic, as the increased dimensionality of the observation space hindered the agent's ability to efficiently learn meaningful representations. Finding the right balance between providing sufficient information and maintaining a manageable observation space dimensionality was a critical aspect of the development process. Ultimately, a reduced number of LiDAR rays proved more effective for learning, demonstrating that more information is not always better in reinforcement learning contexts.

6 Conclusions

With the completion of this thesis, a process for training a reinforcement learning agent with a Soft Actor-Critic model was implemented. The training process involved developing a reward function to encourage high progress speeds, while penalizing going out of bounds and utilizing excessive steering rates. Moreover, an observation space was built to provide sufficient information to the agent about the state of the vehicle, simulating a LiDAR sensor approach for determining the boundaries of the race track. The combination of Soft Actor-Critic and the training process implemented for a self driving race car proved promising for minimizing lap times of the car on the track. The SAC agent was capable of successfully navigating the entire race track with certain models, and minimized the lap times for the majority of the training process. The best average lap time the agent achieved with the default training car was 141.3 seconds, a substantial improvement from the 182.8 second average achieved at the first stage of training. While the lap times consistently dropped, the average lap percentage completion fluctuated inconsistently depending on which training stage model was utilized for testing. The velocity plots across the race track indicated that as the agent was trained more, the speed would oscillate less on the straight regions of the race track indicating that the agent was not utilizing the brakes of the car as often when it was not required. Moreover, the agent improved performance through the complicated chicane turn by smoothening out its turns through it with further training allowing the car to travel with a higher speed through the turns. While the performance for the test vehicle proved promising, when the agent was evaluated with different vehicles, the results were mixed. With similar style vehicles, the agent performed well, maintaining low lap times with some vehicles beating the lap times of the original training vehicle. However, when the vehicle was significantly different the training vehicle either in size, or in terms of handling due to it being an older vehicle, the agent struggled significantly, failing to complete a single lap with vehicles such as the truck, bus, and older car. This indicated that further refinement would be required for the SAC model through additional training with similar vehicles to the testing vehicle. Overall, Soft Actor-Critic as a reinforcement learning model for a self driving race car showcased its ability to develop a robust policy for controlling a specific vehicle on a race track, and minimize the lap times provided sufficient training.

References

[1] OpenAI, "Introduction to RL - Part 1: Key Concepts in RL," https://spinningup.openai.com/en/latest/spinningup/rl_intro.html, OpenAI, 2018, accessed: 2024.

[2] Open Robotics, "Gazebo simulator," https://gazebosim.org/home, Open Robotics, 2024, accessed: 2024.

[3] BeamNG GmbH, "Beamng.tech - soft-body physics simulation," https://beamng.tech, BeamNG GmbH, 2024, accessed: 2024.

[4] F1TENTH, "F1tenth simulator documentation," https://f1tenth.readthedocs.io/en/stable/going_forward/simulator/, F1TENTH, 2023, accessed: 2024.

[5] BeamNG GmbH, "Beamng documentation: Levels," https://documentation.beamng.com/official_content/levels/, BeamNG, 2023, accessed: 2024.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[7] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd ed. O'Reilly Media, Inc., 2019.

[8] NVIDIA Corporation, "Reinforcement learning: A robot learning technique to develop autonomous, adaptable, and efficient robots," https://www.nvidia.com/en-us/use-cases/reinforcement-learning/, NVIDIA, 2024, accessed: 2024.

[9] F. Tong, R. Liu, G. Yin, S. Zhang, and W. Zhuang, "Multi-policy soft actor-critic reinforcement learning for autonomous racing," in *2024 IEEE 18th International Conference on Advanced Motion Control (AMC)*, 2024, pp. 1–7.

[10] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Dürr, "Super-human performance in gran turismo sport using deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4257–4264, 2021.

[11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018. [Online]. Available: https://arxiv.org/abs/1801.01290

[12] T. Carr, "Deterministic vs. stochastic policies in reinforcement learning," https://www.baeldung.com/cs/rl-deterministic-vs-stochastic-policies, Baeldung, 2023, accessed: 2024.

[13] F1TENTH, "F1tenth gym environment," https://github.com/f1tenth/f1tenth_gym, F1TENTH, 2024, accessed: 2024.

[14] C. Romac, "(automatic) curriculum learning for rl," https://huggingface.co/learn/deep-rl-course/en/unitbonus3/curriculum-learning, Hugging Face, 2023, accessed: 2024.

[15] BeamNG GmbH, "Beamng.gym: Gymnasium environments for beamng.tech," https://github.com/BeamNG/BeamNG.gym, BeamNG, 2024, accessed: 2024.

A Software Implementation

A.1 Gymnasium Environment Wrapper

The Gymnasium environment wrapper implementation is built upon a fork of the BeamNG gym repository [15], which provides a standardized interface between the BeamNG.tech simulator and reinforcement learning frameworks by utilizing BeamNGpy. The original repository was modified to create a custom racing environment that supports curriculum-based training and vehicle-specific configurations. Key modifications include but are not limited to:

- Implementation of a custom reward function that balances speed optimization with safe driving behavior
- Addition of simulated LiDAR sensors for track boundary detection
- Enhanced logging and visualization capabilities for training analysis

```
from typing import Any, List, Dict, Tuple
import gymnasium as gym
import numpy as np
from beamngpy import BeamNGpy, Scenario, Vehicle
from beamngpy.sensors import Damage, Electrics
from beamngpy.misc.quat import angle_to_quat
from shapely.geometry import LinearRing, LineString, Point, Polygon
from dataclasses import dataclass, field
import random
from collections import deque
import time
import os


@dataclass
class LidarSettings:
    start_angle: float
    end_angle: float
    num_rays: int
    max_dist: float


@dataclass
class RewardSettings:
    max_damage: float
    damage_penalty: float
    out_of_bounds_penalty: float
    wrong_way_penalty: float
    too_long_penalty: float
    speed_factor: float
    steer_factor: float
```

```
    min_speed: float


@dataclass
class EnvironmentState:
    curr_dist: float = 0
    last_proj: float = None
    last_vehicle_pos: Point = None
    steering_rate: float = 0
    last_steering: float = 0
    remaining_time: float = 0
    spine_speed: float = 0
    vehicle_rel_angle: float = 0
    vehicle_elev_angle: float = 0
    vehicle_velocity: float = 0
    rpm: float = 0
    throttle: float = 0
    brake: float = 0
    steering: float = 0
    gear_index: int = 0
    wheelspeed: float = 0
    lidar_distances: np.ndarray = field(
        default_factory=lambda: np.zeros(271, dtype=np.float32)
    )


class WCARaceGeometry(gym.Env):
    def __init__(
        self,
        host="localhost",
        port=25252,
        start_lap_percent=0.05,
        final_lap_percent=1.0,
        lap_percent_increment=0.05,
        learn_starts=0,
        real_time=False,
        vehicle_index=0,
        randomize_start=False,
        enable_logging=False,
        log_path=None,
    ):
        # Progressive difficulty settings
        self.final_lap_percent = final_lap_percent
        self.lap_percent_increment = lap_percent_increment
        self.current_lap_percent = start_lap_percent
        self.learn_starts = learn_starts
        self.total_steps = 0
```

```
        self.randomize_start = randomize_start
        self.enable_logging = enable_logging
        self.log_path = log_path
        self.episode_time = 0

        # Simulation settings
        self.sim_rate = 20 # simulation steps per second
        self.action_rate = 5 # actions per second
        self.steps = self.sim_rate // self.action_rate
        self.real_time = real_time

        # Define default start position and rotation
        self.default_pos = (394.5, -247.925, 145.25)
        self.default_rot = angle_to_quat((0, 0, 90))

        # Spawn settings
        self.random_position_range = (
            2.5  # Maximum random offset in x,y coordinates (meters)
        )
        self.random_angle_range = (
            20.0  # Maximum random offset in rotation angle (degrees)
        )
        self.spawn_height_offset = 0.6 # Height offset from track surface (meters)

        # Define available vehicles
        self.vehicles = [
            {
                "name": "ETK␣800",
                "model": "etk800",
                "part_config": "vehicles/etk800/856x_310d_A.pc",
                "color": "white",
            },
            {
                "name": "Bruckell␣LeGran",
                "model": "legran",
                "part_config": "vehicles/legran/sport_se_v6_awd_facelift_A.pc",
                "color": "white",
            },
            {
                "name": "Bruckell␣Bastion",
                "model": "bastion",
                "part_config": "vehicles/bastion/sport_gt_A.pc",
                "color": "black",
            },
            {
                "name": "Gavril␣H-Series",
                "model": "van",
```

```
        "part_config": "vehicles/van/h15_ext_vanster.pc",
        "color": "white",
    },
    {
        "name": "Gavril␣T-Series",
        "model": "us_semi",
        "part_config": "vehicles/us_semi/tc82s_cargobox.pc",
        "color": "white",
    },
    {
        "name": "Wentward␣DT40L",
        "model": "citybus",
        "part_config": "vehicles/citybus/highway.pc",
        "color": "white",
    },
    {
        "name": "ETK␣K-Series",
        "model": "etkc",
        "part_config": "vehicles/etkc/kc6x_trackday_A.pc",
        "color": "white",
    },
]

# Validate and set vehicle index
if vehicle_index < 0 or vehicle_index >= len(self.vehicles):
    print(f"Invalid␣vehicle␣index␣{vehicle_index},␣using␣default␣vehicle␣0
        ")
    self.vehicle_index = 0
else:
    self.vehicle_index = vehicle_index

# LiDAR settings
self.lidar_info = LidarSettings(
    start_angle=-np.deg2rad(135),
    end_angle=np.deg2rad(135),
    num_rays=31,
    max_dist=500,
)

# Reward and penalty settings
self.reward_settings = RewardSettings(
    max_damage=1000,
    damage_penalty=-200,
    out_of_bounds_penalty=-200,
    wrong_way_penalty=-200,
    too_long_penalty=-200,
    speed_factor=0.05,
```

```
            steer_factor=0.5,
            min_speed=0,
        )

        # Track and vehicle setup
        self.max_lap_time = 5 * 60
        self.spine = None
        self.l_edge = None
        self.r_edge = None
        self.polygon = None
        self.max_steer_rate = 0.2

        # Initialize state
        self.state = EnvironmentState()

        # Define action and observation spaces
        self.action_space = self._action_space()
        self.observation_space = self._observation_space()

        # Initialize BeamNG
        self.bng = BeamNGpy(host, port)
        self.bng.open()
        self.vehicle = self._setup_vehicle()
        self.scenario = self._setup_scenario()
        self._build_racetrack()

        # Start and configure simulation
        self.bng.scenario.start()
        self.bng.control.pause()

        # Initialize simulation
        self._initialize_simulation()

        # Logging setup
        if self.enable_logging:
            self._log_data()

    def __del__(self):
        if hasattr(self, "log_file") and self.log_file:
            self.log_file.close()
        self.bng.close()

    def close(self):
        if hasattr(self, "log_file") and self.log_file:
            self.log_file.close()
        if hasattr(self, "bng"):
            self.bng.close()
```

```
def _initialize_simulation(self):
    # Set simulation speed
    self._configure_simulation()

    # Set states
    self.state.remaining_time = self.max_lap_time * self.current_lap_percent +
        30
    self.state.curr_dist = 0 # Reset current distance

    # BeamNG config
    self.vehicle.control(throttle=0.0, brake=0.0, steering=0.0)
    self.bng.scenario.restart()
    self.vehicle.recover()
    self.bng.control.pause()
    self.vehicle.set_shift_mode("realistic_automatic")
    self.vehicle.control(gear=2)
    self.vehicle.sensors.poll()

def _setup_vehicle(self) -> Vehicle:
    vehicle_config = self.vehicles[self.vehicle_index]
    print(f"Using␣vehicle:␣{vehicle_config['name']}")

    vehicle = Vehicle(
        "racecar",
        model=vehicle_config["model"],
        license="BEAMNG",
        color=vehicle_config["color"],
        part_config=vehicle_config["part_config"],
    )
    vehicle.sensors.attach("electrics", Electrics())
    vehicle.sensors.attach("damage", Damage())
    return vehicle

def _setup_scenario(self) -> Scenario:
    scenario = Scenario("west_coast_usa", "wca_race_geometry_v0")
    scenario.add_vehicle(
        self.vehicle,
        pos=self.default_pos,
        rot_quat=self.default_rot,
    )
    scenario.make(self.bng)
    self.bng.scenario.load(scenario)
    return scenario

def _configure_simulation(self):
    self.bng.control.queue_lua_command("be:setPhysicsDeterministic(true)")
```

```
        if self.real_time:
            self.bng.control.queue_lua_command(f"Engine.setFPSLimiter({self.
                sim_rate})")
            self.bng.control.queue_lua_command("Engine.setFPSLimiterEnabled(true)"
                )
        else:
            self.bng.control.queue_lua_command("Engine.setFPSLimiterEnabled(false)
                ")

    def _action_space(self):
        return gym.spaces.Box(low=-1.0, high=1.0, shape=(2,), dtype=np.float32)

    def _observation_space(self):
        # Flatten the observation space
        return gym.spaces.Box(
            low=np.array(
                [
                    -np.pi, # Relative angle
                    -np.pi, # Elevation angle
                    -np.inf, # Vehicle velocity
                    0,  # RPM
                    0,  # Throttle
                    0,  # Brake
                    -1.0,  # Steering
                    -1,  # Gear index
                    0,  # Wheelspeed
                    *[0] * self.lidar_info.num_rays, # LiDAR
                    -np.inf, # Spine speed
                ],
                dtype=np.float32,
            ),
            high=np.array(
                [
                    np.pi, # Relative angle
                    np.pi, # Elevation angle
                    np.inf, # Vehicle velocity
                    np.inf, # RPM
                    1.0,  # Throttle
                    1.0,  # Brake
                    1.0,  # Steering
                    8,  # Gear index
                    np.inf, # Wheelspeed
                    *[self.lidar_info.max_dist] * self.lidar_info.num_rays, # LiDAR
                    np.inf, # Spine speed
                ],
                dtype=np.float32,
            ),
```

```
        dtype=np.float32,
    )

def _build_racetrack(self):
    roads = self.bng.scenario.get_roads()
    RACETRACK_PID = "064a5d03-61d1-4ed7-9136-905b40928f01"
    track_id, _ = next(
        filter(lambda road: road[1]["persistentId"] == RACETRACK_PID, roads.
            items())
    )
    track = self.bng.scenario.get_road_edges(track_id)
    l_vtx, s_vtx, r_vtx = [], [], []
    for edges in track:
        r_vtx.append(edges["right"])
        s_vtx.append(edges["middle"])
        l_vtx.append(edges["left"])

    self.spine = LinearRing(s_vtx)
    self.r_edge = LinearRing(r_vtx)
    self.l_edge = LinearRing(l_vtx)
    self.polygon = Polygon([v[0:2] for v in l_vtx], holes=[[v[0:2] for v in
        r_vtx]])

    # Log track geometry if enabled
    if self.enable_logging:
        # Create log directory if it doesn't exist
        if self.log_path:
            os.makedirs(self.log_path, exist_ok=True)
            spine_file = os.path.join(self.log_path, "track_spine.csv")
            left_file = os.path.join(self.log_path, "track_left_edge.csv")
            right_file = os.path.join(self.log_path, "track_right_edge.csv")
        else:
            spine_file = "track_spine.csv"
            left_file = "track_left_edge.csv"
            right_file = "track_right_edge.csv"

        # Log spine points
        with open(spine_file, "w") as f:
            f.write("x,y,z\n")
            for point in s_vtx:
                f.write(f"{point[0]},{point[1]},{point[2]}\n")

        # Log left edge points
        with open(left_file, "w") as f:
            f.write("x,y,z\n")
            for point in l_vtx:
                f.write(f"{point[0]},{point[1]},{point[2]}\n")
```

```
        # Log right edge points
        with open(right_file, "w") as f:
            f.write("x,y,z\n")
            for point in r_vtx:
                f.write(f"{point[0]},{point[1]},{point[2]}\n")

def _log_data(self):
    if not self.enable_logging:
        return

    vehicle_state = self.vehicle.state
    vehicle_pos = vehicle_state["pos"]
    vehicle_dir = vehicle_state["dir"]
    vehicle_angle = np.arctan2(vehicle_dir[1], vehicle_dir[0])

    # Create log directory if it doesn't exist
    if self.log_path:
        os.makedirs(self.log_path, exist_ok=True)
        log_file = os.path.join(self.log_path, "race_track_data.csv")
        episode_log_file = os.path.join(self.log_path, "episode_results.csv")
    else:
        log_file = "race_track_data.csv"
        episode_log_file = "episode_results.csv"

    # Initialize log files if they don't exist
    if not hasattr(self, "log_file"):
        self.log_file = open(log_file, "w")
        self.log_file.write("step,x,y,z,angle,speed,spine_speed\n")
        self.episode_log_file = open(episode_log_file, "w")
        self.episode_log_file.write("steps,lap_time,lap_percentage\n")

    self.log_file.write(
        f"{self.total_steps},{vehicle_pos[0]},{vehicle_pos[1]},{vehicle_pos
            [2]},"
        f"{vehicle_angle},{self.state.vehicle_velocity},{self.state.
            spine_speed}\n"
    )
    self.log_file.flush()

def _log_episode_results(self, terminated, truncated):
    if not self.enable_logging:
        return

    if terminated or truncated:
        lap_percentage = self.state.curr_dist / self.spine.length
        self.episode_log_file.write(
```

```
            f"{self.total_steps},{self.episode_time},{lap_percentage}\n"
        )
        self.episode_log_file.flush()

    def step(self, action):
        self._update(action)
        self.total_steps += 1 # Increment total steps counter
        self.episode_time += 1 / self.action_rate # Increment episode time by time
            step
        observation = self._get_obs()
        reward, terminated, truncated = self._get_reward(observation)
        self._log_data() # Log data after each step
        self._log_episode_results(
            terminated, truncated
        )  # Log episode results if episode ended
        print(f"Reward:␣{reward:.2f},␣Terminated:␣{terminated},␣Truncated:␣{
            truncated}")
        return observation, reward, terminated, truncated, {}

    def reset(self, seed: int | None = None, options: dict[str, Any] | None = None
        ):
        super().reset(seed=seed, options=options)
        self.episode_time = 0 # Reset episode time

        if self.randomize_start:
            # Get random point on spine
            self.state.last_proj = random.uniform(0, self.spine.length)
            self.state.last_vehicle_pos = self.spine.interpolate(self.state.
                last_proj)

            # Get angle of spine at point
            start_angle = self._get_spine_angle(self.state.last_vehicle_pos)

            # Random offset
            random_pos_offset = (
                random.uniform(-self.random_position_range, self.
                    random_position_range),
                random.uniform(-self.random_position_range, self.
                    random_position_range),
                0,
            )
            random_angle_offest = random.uniform(
                -self.random_angle_range, self.random_angle_range
            )

            # Randomize starting position
            start_pos = (
```

```
            self.state.last_vehicle_pos.x + random_pos_offset[0],
            self.state.last_vehicle_pos.y + random_pos_offset[1],
            self.state.last_vehicle_pos.z + self.spawn_height_offset,
        )
        rot_quat = angle_to_quat(
            (0, 0, np.rad2deg(-start_angle) - 90 + random_angle_offest)
        )
    else:
        # Use the fixed starting position from _setup_scenario
        start_pos = self.default_pos
        rot_quat = self.default_rot

        # Initialize self.state.last_proj and self.state.last_vehicle_pos
        self.state.last_vehicle_pos = Point(*start_pos)
        self.state.last_proj = self.spine.project(self.state.last_vehicle_pos)

    # Teleport the vehicle to the starting position
    self.vehicle.teleport(pos=start_pos, rot_quat=rot_quat)
    self._initialize_simulation()
    return self._get_obs(), {}

def _update(self, action):
    action = [*np.clip(action, -1, 1)]
    action = [float(act) for act in action]
    throttle, self.state.steering_rate = (
        float(action[0]),
        float(action[1] * self.max_steer_rate),
    )
    brake = -throttle if throttle < 0 else 0
    throttle = max(throttle, 0)

    # Compute steering angle given a steering rate
    steering = float(
        np.clip(self.state.last_steering + self.state.steering_rate, -1, 1)
    )
    self.state.last_steering = steering
    self.vehicle.control(steering=steering, throttle=throttle, brake=brake)
    self.bng.step(self.steps, wait=True)
    self.state.remaining_time -= 1 / self.action_rate

def _get_obs(self) -> np.ndarray:
    self.vehicle.sensors.poll()
    electrics = self.vehicle.sensors["electrics"]
    vehicle_state = self.vehicle.state
    vehicle_pos = Point(*vehicle_state["pos"])
    vehicle_dir = vehicle_state["dir"]
```

```
        # Update state with current observation values
        self.state.vehicle_rel_angle = self._get_vehicle_rel_angle(
            vehicle_pos, vehicle_dir
        )
        self.state.vehicle_elev_angle = self._get_vehicle_elev_angle(vehicle_dir)
        self.state.vehicle_velocity = self._get_vehicle_velocity(vehicle_pos)
        self.state.rpm = electrics["rpm"]
        self.state.throttle = electrics["throttle"]
        self.state.brake = electrics["brake"]
        self.state.steering = electrics["steering"]
        self.state.gear_index = electrics["gear_index"]
        self.state.wheelspeed = electrics["wheelspeed"]
        self.state.lidar_distances = self._get_lidar_distances(
            vehicle_pos, np.arctan2(vehicle_dir[1], vehicle_dir[0])
        )
        self.state.spine_speed = self._get_spine_speed(vehicle_pos)

        # Ensure other values stay within observation bounds
        elev_angle = np.clip(self.state.vehicle_elev_angle, -np.pi, np.pi)
        brake = np.clip(self.state.brake, 0, 1.0)
        gear_index = np.clip(self.state.gear_index, -1, 8)

        # Ensure LiDAR values are within bounds
        lidar_distances = np.clip(
            self.state.lidar_distances, 0, self.lidar_info.max_dist
        )

        # Flatten the observation
        return np.array(
            [
                self.state.vehicle_rel_angle,
                elev_angle,
                self.state.vehicle_velocity,
                self.state.rpm,
                self.state.throttle,
                brake,
                self.state.steering,
                gear_index,
                self.state.wheelspeed,
                *lidar_distances,
                self.state.spine_speed,
            ],
            dtype=np.float32,
        )

    def _get_vehicle_rel_angle(self, vehicle_pos, vehicle_dir) -> float:
        spine_angle = self._get_spine_angle(vehicle_pos)
```

```
        vehicle_angle = np.arctan2(vehicle_dir[1], vehicle_dir[0])
        rel_angle = (vehicle_angle - spine_angle + np.pi) % (2 * np.pi) - np.pi
        return rel_angle

    def _get_spine_angle(self, position) -> float:
        spine_proj_dist = self.spine.project(position)
        spine_fwd = self.spine.interpolate((spine_proj_dist + 1) % self.spine.
            length)
        spine_bwd = self.spine.interpolate((spine_proj_dist - 1) % self.spine.
            length)
        spine_angle = np.arctan2(spine_fwd.y - spine_bwd.y, spine_fwd.x -
            spine_bwd.x)
        return spine_angle

    def _get_vehicle_elev_angle(self, vehicle_dir) -> float:
        return np.arctan2(vehicle_dir[2], np.linalg.norm(vehicle_dir[0:2]))

    def _get_vehicle_velocity(self, vehicle_pos) -> float:
        if not self.state.last_vehicle_pos:
            self.state.last_vehicle_pos = vehicle_pos
            return 0
        delta_time = 1 / self.action_rate
        velocity_vector = np.array(
            [
                (vehicle_pos.x - self.state.last_vehicle_pos.x) / delta_time,
                (vehicle_pos.y - self.state.last_vehicle_pos.y) / delta_time,
                (vehicle_pos.z - self.state.last_vehicle_pos.z) / delta_time,
            ]
        )
        self.state.last_vehicle_pos = vehicle_pos
        return np.linalg.norm(velocity_vector)

    def _get_spine_dist(self, vehicle_pos) -> float:
        curr_proj = self.spine.project(vehicle_pos)
        dist = curr_proj - self.state.last_proj
        self.state.last_proj = curr_proj

        # Check if distance is valid
        if abs(dist) > 0.9 * self.spine.length:
            if dist < 0:
                # Passed end point
                dist += self.spine.length
            elif dist > 0:
                # Went behind start
                dist -= self.spine.length

        # Update distance covered
```

```
        self.state.curr_dist += dist

        return dist

    def _get_spine_speed(self, vehicle_pos) -> float:
        dist = self._get_spine_dist(vehicle_pos)
        time_step = 1 / self.action_rate
        spine_speed = dist / time_step
        return spine_speed

    def _ray_distance(
        self,
        point: Point,
        angle: float,
        linear_rings: Tuple[LinearRing, LinearRing],
        max_dist: float,
    ) -> float:
        point_2d = Point(point.x, point.y)
        ray_end = Point(
            point_2d.x + max_dist * np.cos(angle), point_2d.y + max_dist * np.sin(
                angle)
        )
        ray = LineString([point_2d, ray_end])
        closest_dist = max_dist
        for ring in linear_rings:
            intersection = ring.intersection(ray)
            if not intersection.is_empty:
                closest_point = (
                    min(intersection.geoms, key=lambda p: point_2d.distance(p))
                    if intersection.geom_type == "MultiPoint"
                    else intersection
                )
                closest_dist = min(closest_dist, point_2d.distance(closest_point))
        return closest_dist

    def _get_lidar_distances(
        self, vehicle_pos: Point, vehicle_angle: float
    ) -> np.ndarray:
        tracks = (self.l_edge, self.r_edge)
        angles = (
            np.linspace(
                self.lidar_info.start_angle,
                self.lidar_info.end_angle,
                self.lidar_info.num_rays,
            )
            + vehicle_angle
        )
```

```
        return np.array(
            [
                self._ray_distance(vehicle_pos, angle, tracks, self.lidar_info.
                    max_dist)
                for angle in angles
            ],
            dtype=np.float32,
        )

    def _get_reward(self, observation: np.ndarray) -> Tuple[float, bool, bool]:
        # Crash penalty
        # Only used if the maximum damage is exceeded, or out of bounds, or wrong
            way
        angle = observation[0]
        velocity = observation[2]
        crash_penalty = 5 * (
            -np.abs(angle)
            / (np.pi / 2)
            * self.reward_settings.speed_factor
            * velocity**2
        )

        # Maximum damage
        if self.vehicle.sensors["damage"]["damage"] > self.reward_settings.
            max_damage:
            print("reset: damage exceeded")
            return self.reward_settings.damage_penalty + crash_penalty, True,
                False

        # Out of bounds
        vehicle_pos = Point(*self.vehicle.state["pos"])
        if not self.polygon.contains(vehicle_pos):
            print("reset: out of bounds")
            return (
                self.reward_settings.out_of_bounds_penalty + crash_penalty,
                True,
                False,
            )

        # Wrong way
        spine_speed = observation[-1]
        if spine_speed < -5:
            print("reset: wrong way")
            return self.reward_settings.wrong_way_penalty + crash_penalty, True,
                False

        # Too long
```

```
if self.state.remaining_time <= 0:
    print("reset: too long")
    return self.reward_settings.too_long_penalty, False, True

# Spine speed reward
# Quadratic function utilized as linear would result in the same reward
    for
# a given track regardless of the car's spine speed.
speed_reward = (
    self.reward_settings.speed_factor * np.sign(spine_speed) * spine_speed
        **2
)

# Steering rate punishment
# Punishment is linear wrt steering rate, and proportional to speed reward
    .
# Designed to not exceed speed reward so that there is always a reward for
# progressing along the track, stops converging at local optimum.
steer_punishment = (
    -np.abs(speed_reward)
    * self.reward_settings.steer_factor
    * (self.state.steering_rate / self.max_steer_rate)
)

# Sum of rewards and punishment
total_reward = speed_reward + steer_punishment

# Race complete
if self.state.curr_dist > self.spine.length * self.current_lap_percent:
    print(f"race complete at {self.current_lap_percent:.3f} lap percent")

    # Increase lap percent for progressive difficulty only after
        learn_starts
    if (
        self.current_lap_percent < self.final_lap_percent
        and self.total_steps > self.learn_starts
    ):
        self.current_lap_percent = min(
            self.current_lap_percent + self.lap_percent_increment,
            self.final_lap_percent,
        )
        print(f"increasing to {self.current_lap_percent:.3f} lap percent")
    else:
        if self.total_steps <= self.learn_starts:
            print(
                f"still in learning phase ({self.total_steps}/{self.
                    learn_starts}), not increasing difficulty"
```

```
                )

            return total_reward, True, False

        return total_reward, False, False
```

A.2 Progressive Training Pipeline

```
import beamnggym
import gymnasium as gym
import torch
from stable_baselines3 import SAC
from stable_baselines3.common.logger import configure
import os
import time

# Learning starts above this value
learn_starts = 100

# Training stages with their respective lap percentages, timesteps, and vehicle
    indices
training_stages = [
    (0.05, 50_000, 0), # 5% of track, ETK 800
    (0.1, 50_000, 0), # 10% of track, ETK 800
    (0.3, 50_000, 0), # 30% of track, ETK 800
    (0.5, 50_000, 0), # 50% of track, ETK 800
    (0.7, 50_000, 0), # 70% of track, ETK 800
    (0.9, 50_000, 0), # 90% of track, ETK 800
    (1.0, 50_000, 0), # 100% of track, ETK 800
    (1.0, 50_000, 0), # Additional training on full track
]

# Vehicle names for reference
vehicle_names = [
    "ETK␣800", # 0
    "Bruckell␣LeGran", # 1
    "Bruckell␣Bastion", # 2
    "Gavril␣H-Series", # 3
    "Gavril␣T-Series", # 4
    "Wentward␣DT40L", # 5
    "ETK␣K-Series", # 6
]

# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using␣device:␣{device}")
```

```
# Initialize model as None
model = None

# Train through each stage
for stage_idx, (lap_percent, timesteps, vehicle_index) in enumerate(
    training_stages):
    # Generate model name from stage information
    vehicle_name = vehicle_names[vehicle_index].lower().replace("␣", "_")
    model_name = f"sac_stage_{stage_idx}_{vehicle_name}_{int(lap_percent*100)}pct"

    # Check if model already exists
    save_path = os.path.join("models", model_name)
    if os.path.exists(save_path + ".zip"):
        print(f"Model␣{model_name}␣already␣exists,␣skipping␣training")
    else:
        # Set up logger for this stage
        log_path = os.path.join("logs", model_name)
        logger = configure(log_path, ["stdout", "csv", "tensorboard"])

        # Create BeamNG gym with current lap percentage and vehicle
        env = gym.make(
            "BNG-WCA-Race-Geometry-v0",
            start_lap_percent=lap_percent,
            final_lap_percent=lap_percent,
            lap_percent_increment=0.0,
            learn_starts=learn_starts,
            real_time=False,
            vehicle_index=vehicle_index,
            randomize_start=True,
            enable_logging=False,
        )

        if stage_idx == 0:
            # First stage - create new model
            model = SAC(
                "MlpPolicy",
                env,
                verbose=1,
                device=device,
                learning_starts=learn_starts,
                seed=42,
            )
        else:
            # Load previous model for next stage
            prev_vehicle_name = (
                vehicle_names[training_stages[stage_idx - 1][2]]
```

```
            .lower()
            .replace("␣", "_")
        )
        prev_lap_percent = training_stages[stage_idx - 1][0]
        prev_model_name = f"sac_stage_{stage_idx-1}_{prev_vehicle_name}_{int(
            prev_lap_percent*100)}pct"

        load_path = os.path.join("models", prev_model_name)
        print(f"Loading␣previous␣model␣from␣{load_path}")
        model = SAC.load(load_path, env=env, verbose=1, device=device)

    # Set logger and train
    model.set_logger(logger)
    model.learn(total_timesteps=timesteps)

    # Save model
    model.save(save_path)
    print(f"Saved␣model␣to␣{save_path}")

    # Close BeamNG connection
    env.close()
```