F1TENTH to DreamerV3
Development Guide

Author: Hyunsu Lim (hyunsul2@uci.edu)

4th February 2025
Version: 1.2.1

Informatics and Computer Science
University of California, Irvine
Irvine, CA, US

CONTENTS

Chapter 1

Background & Objectives

This project addresses the challenge of enabling autonomous navigation for an F1Tenth racing car using the DreamerV3 reinforcement learning framework and LiDAR data. DreamerV3, a model-based reinforcement learning algorithm, has demonstrated impressive performance in various domains, but its typical use with image-based inputs presents a significant hurdle when dealing with the LiDAR range data provided by the F1Tenth's primary sensor. Our work is motivated by the need for robust and efficient autonomous racing solutions on resource-constrained platforms where image processing may be computationally expensive or impractical.

1.1 Background

- DreamerV3 and MBRL: Studied DreamerV3 architecture, training, and performance. Researched model-based RL principles and related algorithms.
- F1Tenth and ROS2: Investigated F1Tenth platform, ROS2, message types, and external simulations. Explored existing F1Tenth projects.
- Existing Implementations: Analyzed DreamerV3 code repositories and similar projects using image-based inputs or other robotic platforms. Researched data representation for neural networks.
- Motivation: Driven by interest in robotics and AI, particularly autonomous racing challenges. Motivated to adapt DreamerV3 to LiDAR inputs and contribute to efficient autonomous racing solutions.

1.2 Analysis

Our background research revealed that directly applying DreamerV3 to the F1Tenth platform with its 1D LiDAR input presents two key challenges. First, DreamerV3's world model is designed to process image-like data through convolutional neural networks (CNNs), making it incompatible with the 1D range data from the LiDAR. Second, the

real-time requirements of autonomous racing necessitate an efficient integration with the ROS2-based F1Tenth simulator.

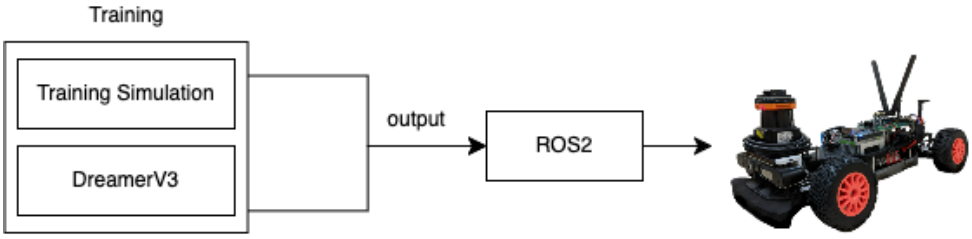

Figure 1.1: Deployment stages

Our analysis of these challenges led us to decompose the problem into four main tasks:

- LiDAR Data Adaptation: We needed to bridge the gap between the 1D LiDAR data and DreamerV3's input requirements. Two primary approaches emerged:
  - Image-like Conversion: Transforming the 1D range data into a 2D representation resembling an image, suitable for input to a CNN. This could involve polar coordinate transformations or creating a simple occupancy grid.
  - MLP-based Adaptation: Modifying DreamerV3's world model to directly accept 1D input using a multi-layer perceptron (MLP) instead of a CNN.
- Custom Environment Development: A custom Gymnasium environment was required to provide a standardized interface between DreamerV3 and external gym simulators. This environment would handle data transfer, reward calculation, and interaction with the simulated car.
- Training Simulation: Either through a multi-core training environment or any other training options, we aim to create reliable software to train our model that maximizes computational efficiency.
- ROS2 Integration: Seamless integration with the ROS2 framework is crucial for real-time control. This involves establishing communication channels between DreamerV3 and the simulator using ROS2 topics and messages.

Our core objective is to pursue either the image-like conversion and the MLP-based adaptation approaches based on efficient code structures, or attempt to convert the sensor data into an image-based observation. While the image-like conversion offers the advantage of leveraging DreamerV3's existing CNN architecture, we recognize that it might introduce information loss or require complex transformations. The MLP-based adaptation, while requiring more extensive code modifications, offers the potential for more direct and efficient processing of the LiDAR data 1.1.

1.3 Process

This project follows an iterative and incremental development process, adapted from the agile methodology. While a full-fledged agile framework with sprints and formal reviews isn't strictly necessary for this project's scale, the core principles of iterative development,

Figure 1.2: Program schema

continuous integration, and close collaboration seem fitting. Our plan can be broken down into the following phases :

1. Planning and Requirements Gathering (Initial Phase): This initial phase involved defining the project scope, identifying key objectives, and analyzing the technical requirements. We conducted background research, explored existing implementations, and analyzed the F1Tenth platform and its ROS2 environment 1.2.

2. Design and Implementation (Iterative Cycles): This phase forms the core of our development process. We adopt an iterative approach, focusing on developing and testing individual components before integrating them into the larger system. Each iteration typically involves:

   – Component Development: Implementing a specific module, such as the custom Gymnasium environment, the data preprocessing module, or a modification to the DreamerV3 codebase.
   – Unit Testing: Thoroughly testing each component in isolation to ensure its functionality and identify potential bugs early on.
   – Integration: Integrating the developed component with the existing system.
   – System Testing: Testing the integrated system to ensure that the components work together correctly.

3. Testing and Evaluation (Continuous): Testing is an ongoing process throughout the project. We employ unit tests for individual components and system tests for integrated modules. As the project progresses, we will conduct more extensive testing in the simulated F1Tenth environment to evaluate the performance of the trained agents. This continuous testing allows us to identify and address issues early in the development cycle.

4. Documentation and Reporting (Ongoing): Documentation will be maintained throughout the project, with regular updates to this document, code comments, and a final project report. This ensures that the project's progress, design decisions, and implementation details are clearly recorded.

This iterative and incremental process will provide high efficiency for this project. It allows us to break down the complex task of integrating DreamerV3 with the F1Tenth platform into smaller, manageable units. The continuous testing and integration enables us to identify and address issues early, preventing them from escalating into larger problems later on. The flexibility of this approach also allows us to adapt to unforeseen challenges and adjust our plans as needed.

Chapter 2

Environment Set Up

Setting up a consistent and reproducible development environment is crucial for a project of this nature. To ensure that all team members have access to the necessary software and dependencies, and to simplify the deployment process, we have opted to use package manager UV and a virtual environment set up using the Python3 standard library. While this ensures a simpler and more stable production environment, it will require the use of local hardware resources, which are prone to compatibility issues. However, we are in active development to eliminate potential incompatibility for mainstream operating systems.

This document serves as a comprehensive guide for contributing to the DreamerV3-F1Tenth project. Whether you're interested in adding new features, fixing bugs, improving documentation, or simply sharing your expertise, this guide will provide you with the necessary information and procedures to contribute effectively.

2.1 List of Packages

We aimed to utilize the latest version of every software to enhance the compatibility among devices and reduce potential deprecation. The list of packages are extensive and continuously updated; therefore, the list will only contain the core packages that are necessary for development.

- PyTorch: The deep learning framework used by DreamerV3.
- racecar_gym: main gym environment forked from CPS_TUWien.
- Gymnasium: OpenAI's gym framework used to create custom environments and standardize Machine Learning interfaces.

For more insight in installed packages, please reference uv.lock file in the root directory.

2.2 Github

This project leverages GitHub for version control, collaboration, and issue tracking. Understanding our GitHub workflow is essential for contributing effectively. We utilize a branching model based on Gitflow, adapted for our project's needs, to manage code changes and ensure a stable main branch.

- main Branch: This branch represents the stable, production-ready version of the codebase. Only reviewed and tested code is merged into main.
- Feature Branches: These branches are created for developing new features, implementing improvements, or experimenting with new ideas. They branch off from main and are eventually merged back into main after review. Feature branch names should be descriptive and indicate the purpose of the branch (e.g., feature/lidar-integration, feature/mlp-world-model).
- Bugfix Branches: These branches are specifically for addressing bug fixes. They also branch off from main and are merged back into main after the bug is resolved. Bugfix branch names should clearly identify the bug being fixed (e.g., bugfix/incorrect-reward-calculation).
- Pull Requests (PRs): PRs are the mechanism for proposing changes to the main branch. When you've completed your work on a feature or bugfix branch, you create a PR to request that your changes be reviewed and merged.
- Code Review: Before a PR is merged, it undergoes a code review by project maintainers. This process helps ensure code quality, identify potential issues, and promote knowledge sharing.

2.3 Development Setup

This section details the process for configuring the development environment for the DreamerV3-F1Tenth project. This setup provides a pre-configured environment containing all the necessary dependencies, including uv and the F1Tenth Gym simulator, accessible on your local machine. This approach eliminates environment inconsistencies and simplifies the onboarding process for new contributors.

2.3.1 uv Package Manager

uv is a Python package and project manager that wraps the standard pip functionalities. We utilize uv to enhance the environment reliability. To install uv follow the guide: uv installation guide

2.3.2 Initial Set Up

1. Clone the repository using SSH (HTTP will not grant you access for direct commits):

git clone "git@github.com:uci-f1tenth/uci_f1tenth_workshop.git"

2. Install dependencies by syncing the packages:

   uv sync

3. To run a file, run:

   uv run path\to\file

2.3.3 Code Scheme

The codebase is organized to clearly separate the Gym environment integration from the core DreamerV3 implementation. This modular design facilitates maintainability and allows for independent development of each component. The key directories and files are described below:

- dreamer_node/agents/dreamer_agent.py: This is the primary file responsible for interfacing with the ROS2 environment. It handles the subscription to the scan topic for laser scan data and the publishing of control commands (steering and speed) to the drive topic. This file contains the logic for receiving sensor data, preprocessing it as needed, feeding it to the DreamerV3 agent, and translating the agent's actions into ROS2 messages for the F1Tenth car.
- dreamer_node/dreamer/dream.py: This file serves as our entry point for training, orchestrating the interaction between the environment, the DreamerV3 agent, and the F1Tenth car's actuators. The dreamer directory houses the DreamerV3 algorithm itself, allowing for focused development and modification of the core RL component. This separation of concerns ensures a well-structured and easily maintainable codebase.
- dreamer_node/env/: This directory contains the various Gymnasium environment that includes the observation space, actions space, reward functions, and step functions.
- dreamer_node/util/constants.py: This file contains all the necessary constants and the configuration for the agent.

2.3.4 Linters and Formatters

By default, our Github workflow contains continuous integration that checks for correct style and formatting. We integrated ruff as our primary linter. If the file is not correctly formatted, any merging will be blocked by default.

- To check any flags, use:

uv run ruff check

- You can also fix and error in checks with:

  uv run rugg check --fix

- To auto-format the code base, use:

  uv run ruff format

Chapter 3

DreamerV3

This chapter presents a simplified and mathematically detailed implementation of the DreamerV3 algorithm, drawing inspiration from the paper "DreamerV3: Mastering Diverse Robotic Control by Dreaming". We aim to provide a clear and understandable representation of the algorithm's core components and their interactions.

3.1 World Model: The Agent's Internal Simulator

The world model 3.1, denoted by parameters ϕ, learns to predict the future based on current observations and actions. It's the agent's internal simulator, allowing it to "imagine" the consequences of its choices. We use a Recurrent State-Space Model (RSSM) for this purpose.

3.1.1 RSSM Components

1. Encoder: Takes the current observation (sensory input) x_t and the previous hidden state h_{t-1} and produces a latent state z_t. This is a probabilistic mapping:

$$z_t \sim q_\phi(z_t|h_{t-1}, x_t) \tag{1}$$

We can think of q_ϕ as a neural network (e.g., a multilayer perceptron or a convolutional network if x_t is an image) parameterized by ϕ.

Figure 3.1: Basic functions of DreamerV3

2. Recurrent Dynamics Model: Predicts the next hidden state h_t based on the previous hidden state h_{t-1}, the previous latent state z_{t-1}, and the action taken a_{t-1}:

$$h_t = f_\phi(h_{t-1}, z_{t-1}, a_{t-1}) \tag{2}$$

   f_ϕ is another neural network (e.g., an RNN or LSTM).

3. Dynamics Predictor: Predicts the next latent state $\hat{z}_t$ based on the current hidden state h_t:

$$\hat{z}_t \sim p_\phi(\hat{z}_t|h_t) \tag{3}$$

4. Reward Predictor: Predicts the reward $\hat{r}_t$ the agent expects to receive:

$$\hat{r}_t \sim p_\phi(\hat{r}_t|h_t, z_t) \tag{4}$$

5. Continue Predictor: Predicts whether the episode will continue ($\hat{c}_t = 1$) or terminate ($\hat{c}_t = 0$):

$$\hat{c}_t \sim p_\phi(\hat{c}_t|h_t, z_t) \tag{5}$$

6. Decoder: Reconstructs the observation $\hat{x}_t$ from the latent state z_t and hidden state h_t:

$$\hat{x}_t \sim p_\phi(\hat{x}_t|h_t, z_t) \tag{6}$$

3.1.2 World Model Loss

The world model is trained by minimizing the following loss function:

$$L_{WM}(\phi) = \sum_{t=1}^{T} \left[L_{pred}(t) + L_{dyn}(t) + \lambda_{rep} L_{rep}(t)\right] \tag{7}$$

where λ_{rep} is a weighting factor (e.g., 0.1). The individual loss terms are:

$$L_{pred}(t) = -\log p_\phi(x_t|z_t, h_t) - \log p_\phi(r_t|z_t, h_t) - \log p_\phi(c_t|z_t, h_t) \tag{8}$$
$$L_{dyn}(t) = \max\left(1, KL\left(q_\phi(z_t|h_{t-1}, x_t)||p_\phi(z_t|h_t)\right)\right) \tag{9}$$
$$L_{rep}(t) = \max\left(1, KL\left(q_\phi(z_t|h_{t-1}, x_t)||p_\phi(z_t|h_t)\right)\right) \tag{10}$$

The KL divergence term encourages the approximate posterior q_ϕ to be close to the prior p_ϕ. The $\max(1, \cdot)$ term implements the "free bits" technique.

3.2 Critic: Evaluating the Agent's Performance

The critic, parameterized by ψ, learns to estimate the expected cumulative reward (return) from a given state. The critic network $v_\psi(s_t)$ takes the state $s_t = (h_t, z_t)$ as input and outputs a prediction of the return. It is trained to predict the distribution of returns.

The critic is trained using the following loss function:

$$L_C(\psi) = \sum_{t=1}^{T} -\log p_\psi(R_t^\lambda | s_t) \tag{11}$$

where R_t^λ is the λ-return, calculated as:

$$R_t^\lambda = r_t + \gamma c_t \left((1-\lambda) v_\psi(s_t) + \lambda R_{t+1}^\lambda \right) \tag{12}$$

$$R_T^\lambda = v_\psi(s_T) \tag{13}$$

γ is the discount factor, and λ controls the balance between bootstrapping and using the actual returns.

3.3 Actor: Choosing the Optimal Actions

The actor, parameterized by θ, learns to select actions that maximize the expected return. The actor network $\pi_\theta(a_t|s_t)$ takes the state s_t as input and outputs a probability distribution over actions. The actor is trained using the following loss function:

$$L_A(\theta) = \sum_{t=1}^{T} -\frac{R_t^\lambda - v_\psi(s_t)}{\max(1, S)} \log \pi_\theta(a_t|s_t) + \eta H(\pi_\theta(a_t|s_t)) \tag{14}$$

where S is a running average of the range of returns (between the 5th and 95th percentiles), and η is the entropy bonus coefficient. This loss function combines maximizing returns with encouraging exploration.

3.4 Symlog Transformation and Twohot Encoding

DreamerV3 uses the symlog transformation to handle rewards and observations with different scales. The symlog function is defined as:

$$\text{symlog}(x) = \text{sign}(x) \log(|x| + 1) \tag{15}$$

Its inverse, the symexp function, is:

$$\text{symexp}(x) = \text{sign}(x)(\exp(|x|) - 1) \tag{16}$$

For rewards and returns, DreamerV3 uses a categorical distribution with exponentially spaced bins and twohot encoding. This allows the model to predict continuous values while maintaining stable gradients.

This simplified mathematical description provides a more accessible understanding of the DreamerV3 algorithm. It clarifies the roles of each component and their interactions, highlighting the key mathematical operations involved. This foundation should be helpful for implementing and adapting DreamerV3 for various robotic control tasks.

Chapter 4

Understanding F1Tenth ROS Control

This chapter provides a detailed explanation of how the F1Tenth autonomous racing car interacts with ROS 2 using a simple wall following example written in Python. We will dissect the code, explaining the purpose of each section and how it relates to the overall ROS 2 communication and control process.

4.1 Code Overview

The provided Python code implements a wall following algorithm for the F1Tenth car. It uses a Proportional-Integral-Derivative (PID) controller to maintain a desired distance from a wall. The code interacts with the simulator through ROS 2 topics such as 4.1.

4.1.1 ROS 2 Node and Topic Setup

The code defines a ROS 2 node named wall_follow_node. This node subscribes to the /scan topic, which publishes LaserScan messages containing LiDAR data, and publishes to the /drive topic, which accepts AckermannDriveStamped messages to control the car's steering and speed.

lidarscan_topic = '/scan'
drive_topic = '/drive'

self.sub_scan = self.create_subscription(LaserScan, ...
self.pub_drive = self.create_publisher(AckermannDriveStamped, ...

These lines of code set up the communication channels. self.create_subscription creates a subscriber that listens for LaserScan messages on the /scan topic and calls the scan_callback function whenever a new message arrives. self.create_publisher creates a publisher that sends AckermannDriveStamped messages on the /drive topic to control the car.

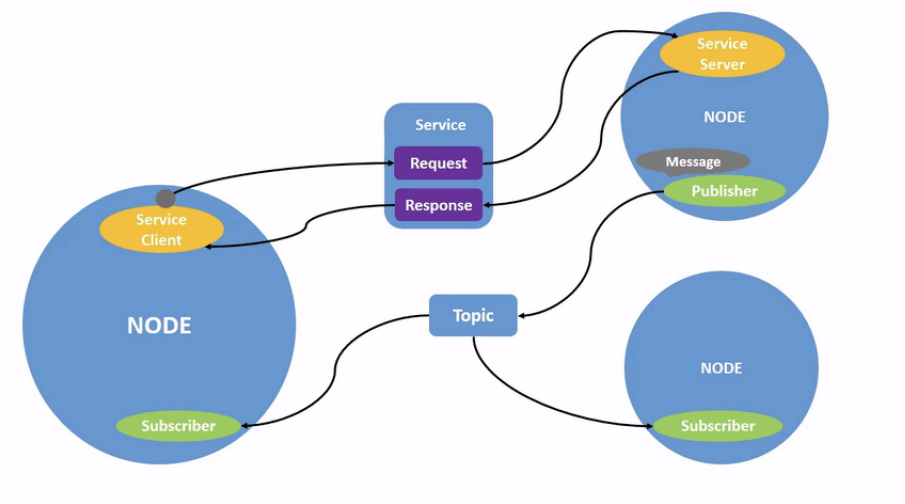

Figure 4.1: Basic ROS2 workflow

4.1.2 Helper Functions

The code includes helper functions to process the LiDAR data and calculate the error.

4.1.2.1 get_range(range_data, angle)

This function takes the raw LiDAR data and an angle as input and returns the corresponding range measurement at that angle. It handles potential NaN (Not a Number) and inf (infinity) values in the LiDAR data.

4.1.2.2 get_error(range_data, dist)

This function calculates the error, which is the difference between the desired distance to the wall (dist) and the actual distance calculated from the LiDAR data. It uses geometric calculations based on LiDAR readings at specific angles to estimate the distance to the wall.

4.1.3 PID Control

The pid_control function implements the PID controller. It takes the calculated error, desired velocity, time difference (delta_t), and some additional LiDAR beam measurements as input. It calculates the steering angle based on the PID gains (kp, ki, kd) and publishes the control command to the /drive topic. The code also includes a simple obstacle avoidance mechanism based on the front LiDAR beam.

self.integral += self.ki * self.error * delta_t
angle = self.kp * self.error + np.clip(self.integral, -1, +1) ...

drive_msg.drive.speed = velocity

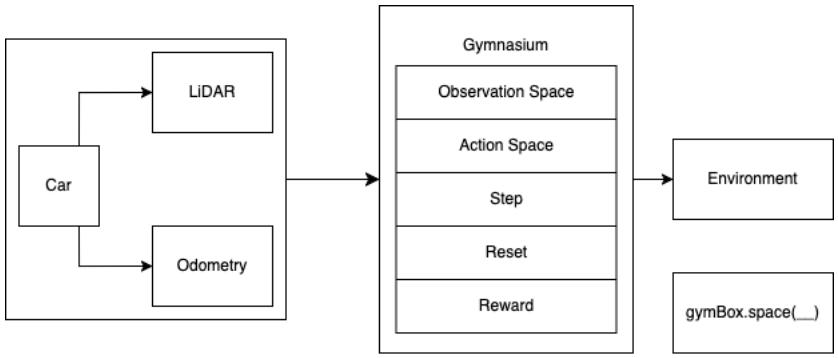

Figure 4.2: Structure of the program

drive_msg.drive.steering_angle = angle
self.pub_drive.publish(drive_msg)

This section calculates the PID output (steering angle) and publishes it along with the desired speed as an AckermannDriveStamped message.

4.1.4 Scan Callback

The scan_callback function is the heart of the wall following algorithm. It is called every time a new LaserScan message is received. This function performs the following steps:

1. Calculates the time elapsed since the last message.
2. Calls get_error to calculate the error.
3. Calculates the desired car velocity based on the current heading angle.
4. Calls pid_control to calculate the steering angle and publish the drive message.
5. Updates the previous time.

4.1.5 Main Function

The main function initializes the ROS 2 node, creates the WallFollow object, and starts the ROS 2 event loop using rclpy.spin(). This keeps the node alive and processes incoming messages. The final result of the program should look something like this: 4.2

4.2 ROS 2 Communication Flow

The LiDAR sensor data is published on the /scan topic. The wall_follow_node subscribes to this topic, processes the data, and publishes control commands on the /drive topic. The F1Tenth car's actuators subscribe to the /drive topic and execute the commands.

This example demonstrates a basic ROS 2 node that interacts with sensor data and publishes control commands. It showcases the fundamental concepts of ROS 2 communication and provides a foundation for more complex autonomous driving algorithms.

Chapter 5

Implementation Details

This chapter details the core logic behind integrating the F1tenth autonomous race car with the DreamerV3 reinforcement learning agent. It explains the structure of the DreamerV3 codebase, how we adapt it for the F1tenth platform using ROS2 Humble, the importance of data type compatibility and normalization, and outlines the high-level implementation goals and timeline.

5.1 DreamerV3 Codebase Structure

The DreamerV3 codebase is designed with a hyperparameter-driven approach, with its main function located in dream.py. It utilizes OpenAI Gymnasium environments to define the observation space, action space, the step() function (which interacts with the environment), and the reset() function (to reset the environment). Wrappers are employed to configure these environments and perform necessary data preprocessing.

A typical DreamerV3 environment definition looks like this (example from the Crafter environment):

```
import gym
import numpy as np

class Crafter:
    # ... (metadata and initialization) ...

    @property
    def observation_space(self):
        # ... (defines the structure of the observation) ...

    @property
    def action_space(self):
        # ... (defines the structure of the action) ...

    def step(self, action):
        # ... (interacts with environment based on action) ...
```

```
def reset(self):
    # ... (resets the underlying environment) ...
```

Wrappers, like the ‘NormalizeActions‘ wrapper in the DreamerV3 examples, can be used to modify the environment (like normalizing the action space).

5.2 Adapting DreamerV3 for F1tenth with ROS2 Humble

Our primary goal is to replace the standard DreamerV3 environment (like Crafter) with an interface to the F1tenth car within the ROS2 Humble framework. This involves creating a custom Gymnasium environment, racerenv.py, that acts as a bridge between DreamerV3 and ROS2. This environment will not directly handle ROS2 communication. Instead, a separate ROS2 node, dreamer_node.py, will manage all ROS2 interactions (subscribing to LiDAR and Odometry data, publishing motor commands) and pass the processed data to racerenv.py.

5.2.1 Data Flow and Compatibility

The dreamer_node.py will subscribe to the relevant ROS2 topics:

- /scan: For LiDAR data.
- /odom: For Odometry data.
- /drive: To publish drive commands.

The dreamer_node.py will then process the received ROS2 messages. The LiDAR data (a 1D array of floats) and Odometry data will be extracted and passed to the racerenv.py's step() and reset() methods. The actions generated by the DreamerV3 agent will be passed from racerenv.py to dreamer_node.py, which will then publish them as messages to the /drive topic.

5.2.2 Data Normalization

Data normalization is crucial for stable and effective reinforcement learning. Raw sensor data (LiDAR, Odometry) often has a wide range of values, which can make it difficult for the agent to learn. We normalize the LiDAR data to a range of [0, 1] within the racerenv.py environment. The action space (steering and speed) is also normalized to a range of [-1, 1]. This ensures that both the input and output of the DreamerV3 agent are within a consistent and manageable range. The denormalization of the actions (from [-1,1] to the cars physical limits) happens in the ‘step‘ function before being published to the car.

5.3 Implementation Goals and Timeline

Our primary goals for this project are:

- Create a robust and efficient interface between the F1tenth car and DreamerV3.
- Train the DreamerV3 agent to successfully navigate the F1tenth car in various simulated environments.
- Evaluate the performance of the trained agent using appropriate metrics.
- Document the project thoroughly.

The project timeline is divided into the following phases:

1. Environment and ROS2 Integration: Implementing racerenv.py and dreamer_node.py, establishing ROS2 communication.
2. DreamerV3 Adaptation: Integrating racerenv.py with the DreamerV3 training loop, adapting networks.py, implementing the reward function.
3. Isaac Lab Migration: Migration to Isaac Lab since F1Tenth Gym is proved to be unreliable.
4. Training and Evaluation: Training the agent, tuning hyperparameters, evaluating performance in simulation.
5. Testing and Refinement: Testing in more complex scenarios, refining the code, and documenting the project.

This timeline is subject to change based on project progress and any unforeseen challenges. Regular team meetings and communication will be essential to ensure the project stays on track.