GENG4412 Engineering Research Project Part 2
Final Report

Deep Learning for Mobile Robots

Noah M. Student
23356892
School of Engineering, University of Western Australia

Supervisor: Thomas B. Supervisor
School of Engineering, University of Western Australia

Word count: 6685

School of Engineering
University of Western Australia

Submitted: 15th October 2025

Project Summary

This project demonstrates the first successful implementation of a deep reinforcement learning (RL) control system on the University of Western Australia's EyeBot 8 mobile robot platform. The primary objective was to address the limitations of traditional, manually coded controllers by developing a Python-based training pipeline to teach an agent complex navigation tasks within the EyeSim simulator and subsequently transfer the learned policy to a physical robot.

Two distinct control models were developed using the Proximal Policy Optimisation (PPO) algorithm: an Angular Model for steering and a Linear Model for speed control in response to traffic signs. The models were trained in custom-designed simulated environments with carefully designed reward functions. A key technical contribution was the establishment of a robust PyTorch-to-ONNX pipeline, which enabled the deployment of the trained models onto the EyeBot's resource-constrained Raspberry Pi hardware.

While the project successfully validated the core hypothesis that simulation-trained RL agents could be deployed on the physical EyeBot, the investigation revealed a significant sim-to-real performance gap. Discrepancies in physical dynamics and visual perception led to issues such as control instability, where a minor "wobble" in the simulated steering became greatly amplified on the physical robot. The results showed that a purely learned solution was insufficient for reliable real-world performance. The most effective outcomes were achieved through a hybrid approach, integrating the learned RL policy with a classical Proportional-Derivative (PD) controller to improve stability. The project concludes that while RL is a viable and powerful paradigm for the EyeBot platform, successful real-world deployment on low-cost hardware hinges on pragmatic hybrid solutions that bridge the gap between simulation and reality.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Thomas Braunl, for his invaluable guidance and mentorship throughout this research project. His expertise, high-level direction, and the foundational concept for this work were instrumental to its success. I am deeply appreciative of his support and the academic freedom he provided me to explore this complex topic.

I would also like to extend my thanks to my fellow researcher, Ben Nicholson. Our regular discussions provided a crucial source of peer feedback, and his collaboration in constructing the physical track was essential for the project's real-world testing phase. The journey was made significantly easier and more enjoyable thanks to his support and camaraderie.

Finally, I owe a special thank you to my partner, Rhiannon Mills. Her unwavering patience, encouragement, and support have been my foundation throughout this entire process. Thank you for being a constant source of motivation, especially during the most challenging times.

Table of Contents

List of Figures

1 Introduction

The standard approach for programming the EyeBot 8 robots [1] at the University of Western Australia has been manual coding. This traditional method, where developers write deterministic rules to map inputs to outputs, has been a reliable foundation for teaching fundamental robotics concepts. However, this method has inherent limitations that cap the platform's potential for more advanced research. Hard-coded controllers are often too constrained as they perform well in the specific environment for which they were designed but fail to adapt when faced with dynamic conditions, a well-documented limitation of non-adaptive control systems [2].

This rigidity poses a direct challenge to the academic and engineering community utilising the EyeBot platform. The significant time and effort required to manually reprogram robots for new tasks creates a bottleneck, slowing down experimental research and limiting the complexity of problems that can be addressed. As a result, the platform's utility as a tool for exploring autonomous systems is constrained.

Reinforcement Learning (RL) presents a compelling solution to this challenge, as its trial-and-error learning framework is particularly well-suited for developing complex control policies in robotics [3]. Instead of being explicitly programmed, an RL agent learns optimal behaviours through direct interaction with its environment. This project aims to investigate the feasibility of implementing modern RL algorithms within the EyeBot ecosystem. The primary objective is to develop and train an agent in the EyeSim simulator [4] to perform a complex navigation task and subsequently transfer this learned policy to a physical EyeBot.

By demonstrating that an EyeBot can acquire a sophisticated skill autonomously, this work seeks to establish a new, more powerful paradigm for controlling these robots. A successful implementation would not only solve a specific control problem but also serve as a blueprint for future research, enhancing the EyeBot platform's capabilities and enabling a more advanced robotics curriculum. This would empower students and researchers to tackle more ambitious projects in autonomous navigation and machine learning.

1.1 Background

1.1.1 Research Summary

RL provides a mathematical framework for agents to learn optimal behaviours through trial-and-error interactions with their environment [5]. Deep Reinforcement Learning (DRL) extends this framework by using deep neural networks to approximate policies and value functions, enabling

effective learning in complex, high-dimensional domains such as gaming and autonomous driving [5].

Among DRL algorithms, Deep Q-Networks (DQN) exemplify value-based methods, offering strong performance but exhibiting sensitivity to hyperparameter tuning and reduced stability in continuous control tasks [6], [7]. Actor-critic methods such as Advantage Actor-Critic (A2C) and Proximal Policy Optimisation (PPO) improve learning stability by combining value estimation and policy optimisation [5], [8]. PPO constrains policy updates via a clipped surrogate objective, achieving a balance between adaptability and robustness across a variety of environments [6]-[8].

In autonomous navigation, DRL enables end-to-end learning of control policies from sensory inputs. El Sallab *et al.* showed that while DQN performs adequately for discrete actions, continuous-action algorithms such as Deep Deterministic Actor-Critic (DDAC) produce smoother, more realistic control [9]. However, deploying such models on real robots faces challenges due to the *sim-to-real gap*, which domain randomisation techniques aim to mitigate by varying simulation parameters to enhance generalisation [10].

A comprehensive literature review detailing these algorithms, comparative analyses, and sim-to-real transfer methods is provided in Appendix A: Literature Review.

1.1.2 EyeBot 8

The EyeBot 8 as seen in Figure 1-1, is an embedded controller system built upon a Raspberry Pi 4 board [11]. This central processor runs the Debian 11 operating system [12], on top of which the EyeBot's proprietary software operates. The Raspberry Pi is connected to a custom EyeBot input/output (I/O) board, which manages all sensor and actuator interfaces.

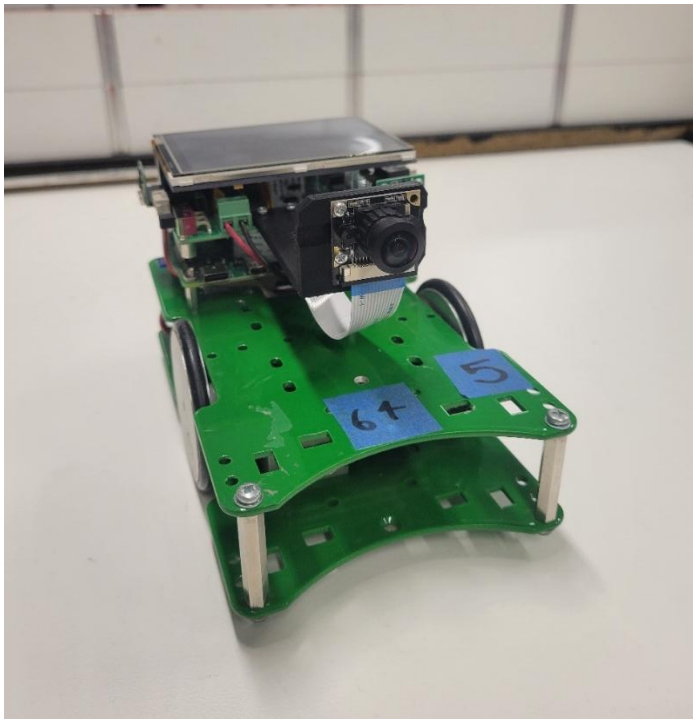

Figure 1-1 Eyebot-8 Robot

The EyeBot platform runs on a custom operating system called RoBIOS [13] (Robot Basic Input/Output System). RoBIOS is a comprehensive C/C++/Python library that provides a high-level Application Programming Interface (API) for interacting with the robot's hardware. This API abstracts the low-level details of hardware control, allowing developers to easily write code for motor control, sensor readings, and image acquisition without direct hardware manipulation.

1.1.3 EyeSim

A critical component of the EyeBot ecosystem is the EyeSim simulator, a native application for Windows, macOS, and Linux that provides a virtual testing ground for the physical robots. Its key strength lies in its high-fidelity simulation, which accurately models the EyeBot's camera, motors, distance sensors, and the physics of its environmental interactions. Additionally, the simulator supports customisable worlds, allowing users to create specific challenges and scenarios, such as tracks with defined lines, obstacles, and colours.

The most significant advantage of EyeSim is its shared Application Programming Interface (API) with the physical EyeBot. This seamless integration means that code developed and debugged in the virtual environment can be deployed on the real robot with little to no modification, dramatically accelerating the development cycle.

1.2 Objectives

This research is guided by the central hypothesis that a control policy for a mobile robot, when trained exclusively in a simulated environment using reinforcement learning, can be successfully transferred to a physical robot with minimal performance degradation, provided the simulation accurately models the real-world operational environment.

To validate this hypothesis, the project is structured around two specific, measurable objectives:

1. **Develop a Simulation-Based RL Training Pipeline:** To design and implement a complete, Python-based reinforcement learning pipeline utilising the EyeSim simulator. The primary goal for this pipeline is to train an agent to learn a complex task combining lane-following and speed control. Success will be quantified by the agent achieving a high completion rate across a variety of simulated track layouts with differing curvatures and conditions.
2. **Evaluate Sim-to-Real Policy Transfer:** To deploy the trained RL models onto the physical EyeBot hardware and conduct a performance evaluation. The objective is to verify that the robot can replicate the simulated behaviour in a real-world replica of the training environment. The key performance metric will be the degree of performance loss, with a target of retaining the basics observed in the simulation.

1.2.1 Advancement of the State of the Art

This work directly addresses a significant gap in the current utilisation of the EyeBot platform. While reinforcement learning is an established and powerful tool in the broader field of mobile robotics, its application within the EyeBot ecosystem is novel. This project will build upon the foundational work of Wege (2020), who successfully applied imitation learning for a lane-following task on the EyeBot platform [14]. The primary contribution of this thesis is the implementation and evaluation of a complete Reinforcement Learning (RL) pipeline, pioneering a new control paradigm for the platform.

1.2.2 Significance and Benefits for Stakeholders

The successful achievement of these objectives will deliver significant benefits to the University of Western Australia's robotics program and its students, representing a tangible return on the research investment. This project provides a foundation for more advanced studies using the EyeBots, laying the groundwork for future projects in areas such as multi-agent collaboration, obstacle avoidance, and more complex sim-to-real transfer research. It elevates the platform's capability from a tool for teaching introductory robotics to one suitable for Artificial Intelligence (AI) research.

2 System Design

2.1 Choice of Reinforcement Learning Algorithm

The Proximal Policy Optimisation (PPO) algorithm was chosen to train the lane-following agent due to its strong performance in continuous control tasks, its training stability, and its ability to generalise which is a crucial factor for sim-to-real transfer.

To implement this, the Gymnasium [15] library was used define a standardised environment API and Stable-Baselines3 (SB3) [16] for its reliable and optimised PPO implementation. This standard toolchain allowed us to focus on experimental design rather than debugging fundamental RL components. The standardised API ensured seamless integration, with the SB3 `learn()` method managing the agent-environment interaction loop.

2.1.1 Hyperparameters and Environment Optimisation Strategy

A strategic decision was made to prioritise environment design and reward shaping. Despite a thorough search across a range of hyperparameters, including the learning rate and discount factor, preliminary observations showed that tuning failed to produce any significant improvement over the baseline. In contrast, refining the reward function led to more immediate gains in the agent's performance.

Consequently, an iterative refinement methodology was adopted for model improvement. This approach involved systematically introducing a change to the reward function, evaluating its impact on the agent's performance, and retaining the modification if it proved beneficial. This iterative loop of proposing, testing, and validating changes proved to be a more direct and impactful means of guiding the agent toward an optimal policy.

2.1.2 Input Imagery and Computational Efficiency

A critical preprocessing step was the conversion of the standard RGB camera feed into a binary image format. This transformation was implemented to achieve two key objectives.

First, by reducing the image to its fundamental components, it serves as a form of feature selection. This process removes irrelevant visual information, compelling the agent's policy to focus exclusively on the most necessary features for navigation. Second, this conversion yields computational benefits as reducing the input data from three channels to a single channel lowers the computational load on the convolutional neural network (CNN) policy. This optimisation is

particularly important for deployment on the resource-constrained Eyebot platform, as it will lower the memory usage and thus increase inference times, which are essential for real-time operation.

To enhance the agent's perceptual capabilities, a fisheye lens was implemented within the simulation environment for training both angular and linear control models. The lens's wide field of view proved critical for navigating sharp turns, where a standard rectilinear lens would typically lose sight of the inside lane marker (as illustrated in Figure 2-1). By maintaining continuous visibility of both lane boundaries, the fisheye lens provides the necessary visual data for the agent to accurately estimate its state and localise its position within the lane.

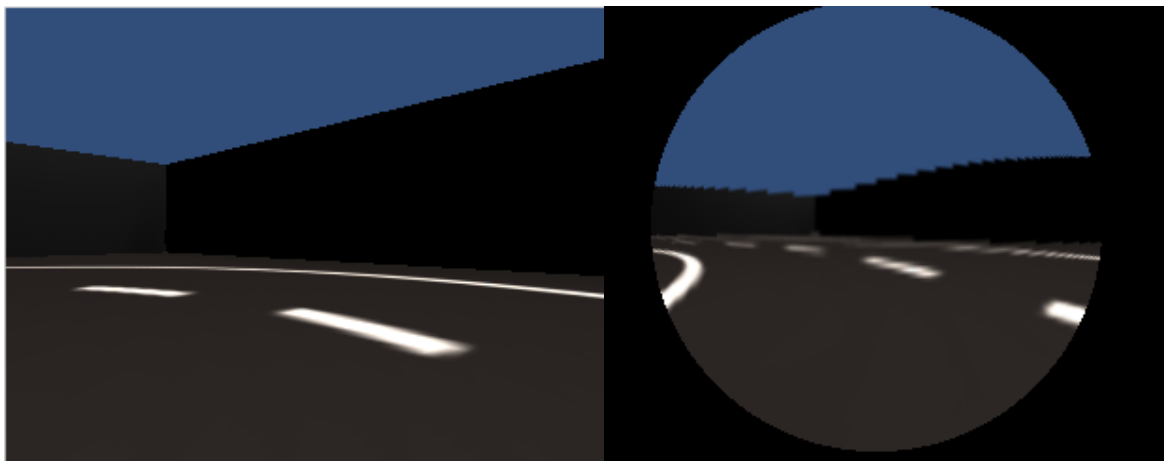

Figure 2-1 Comparison Between Rectilinear Lens and Fisheye Lens in Eyesim

2.2 Angular Model

2.2.1 State Spaces

The agent's perception of the environment is derived from the forward-facing camera feed provided by the EyeSim simulator. To create an efficient and relevant state representation, the binary data undergoes cropping in which the resulting image only retains only the lower half.

The upper portion of the image contains no critical information and the camera's fixed position ensures that the most relevant road information for navigation is always contained within this lower portion of the view. As shown below the four figures show that the top half of the image can be removed and binarisation can be applied without losing detail of the road markings.

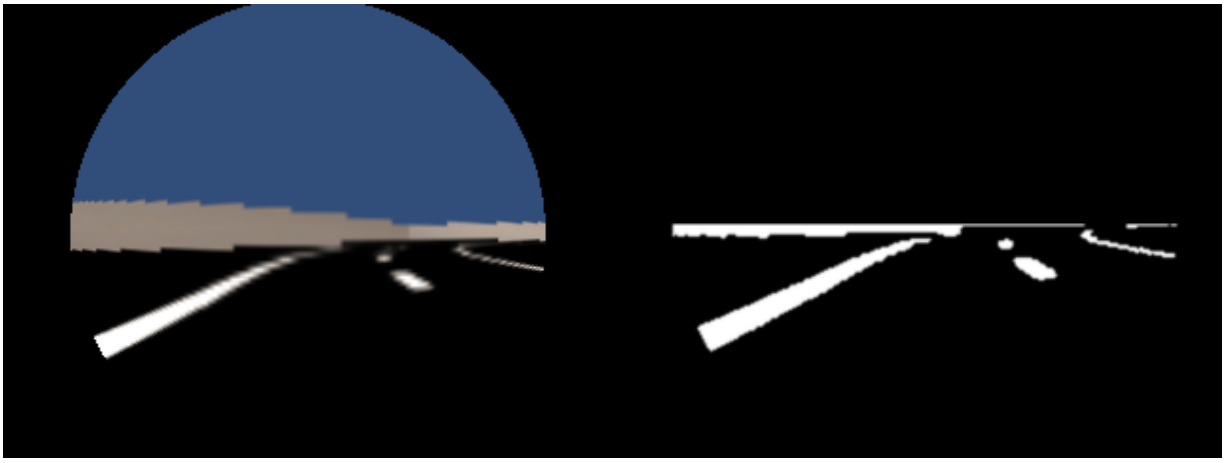

Figure 2-2 RGB Image and Binary Image (cropped) at a Simple Straight Section

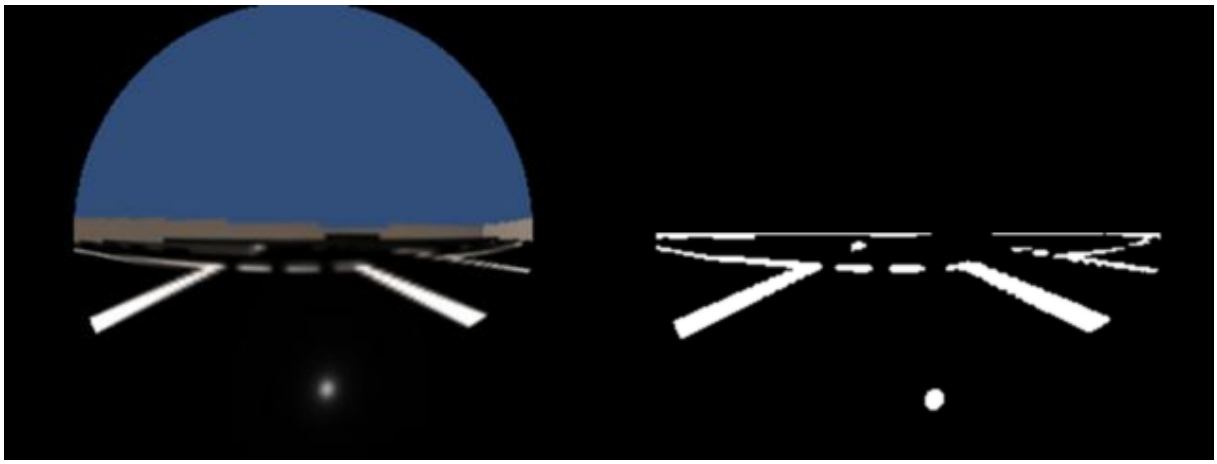

Figure 2-3 RGB Image and Binary Image (cropped) at a Give way for an Intersection

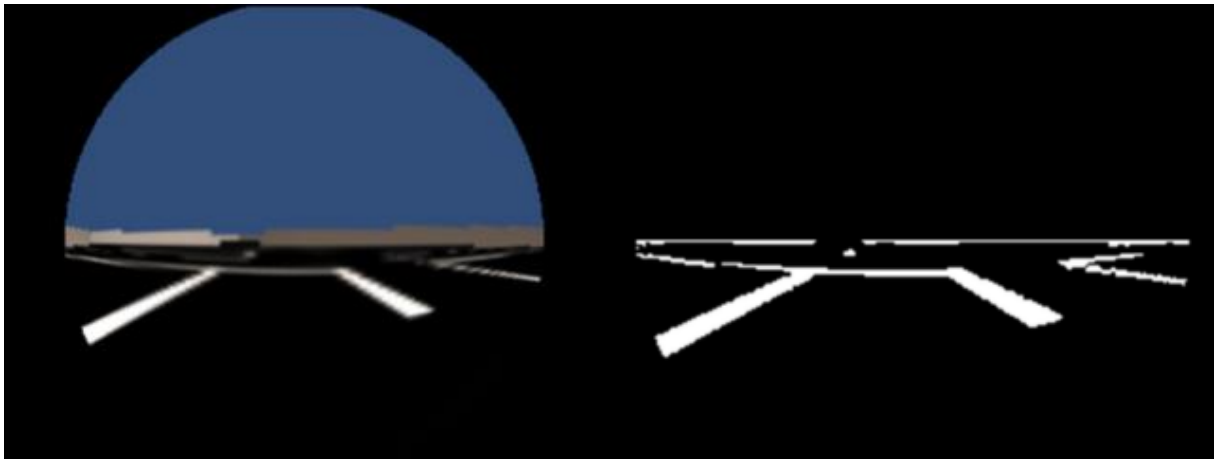

Figure 2-4 RGB Image and Binary Image (cropped) at a Stop for an Intersection

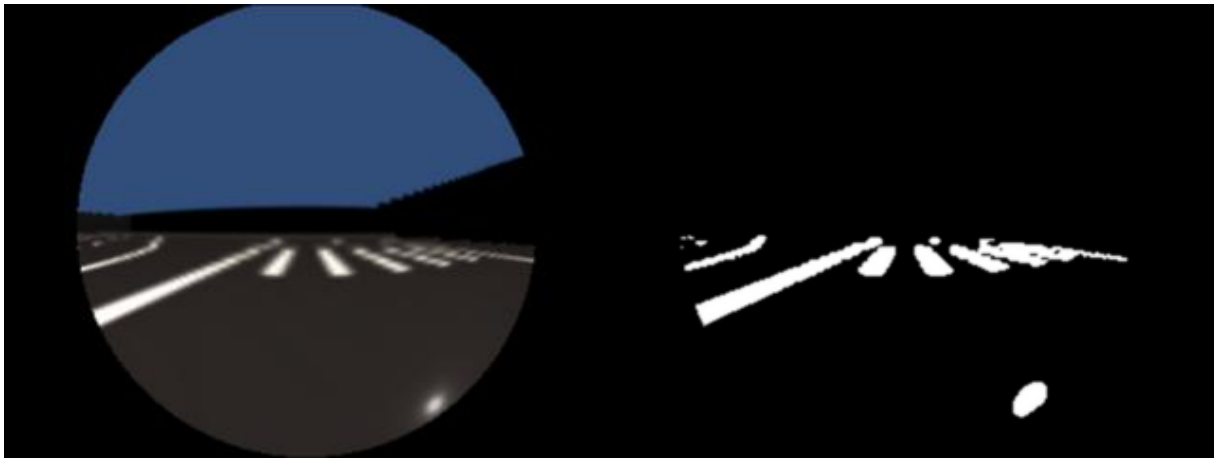

Figure 2-5 RGB Image and Binary Image (cropped) at a Street Crossing

2.2.2 Action Space

This action is represented by a single floating-point number bounded within a normalised range of (-1.0, 1.0). A value of -1.0 corresponds to a maximum left turn, 1.0 corresponds to a maximum right turn, and 0.0 results in straight-line motion.

2.2.3 Training Environment

To train the angular control model, two distinct simulated tracks were developed. The design logic was to first teach the agent foundational turning and navigation skills on a simple track before introducing it to a more complex track to encourage policy generalisation.

2.2.3.1 Track 1: Foundational Circuit

The initial training environment is a simple circuit featuring three long straight sections and a single 4 way intersection (Figure 2-6). This track was designed to teach the agent basic lane-following and turning manoeuvres.

A key limitation of this symmetrical design is the lack of turn diversity within a single path. When navigating the left side of the track, the agent exclusively encounters right-hand turns, while the right side presents only left-hand turns. While effective for initial learning, this track alone is insufficient for developing a policy capable of handling varied turn sequences.

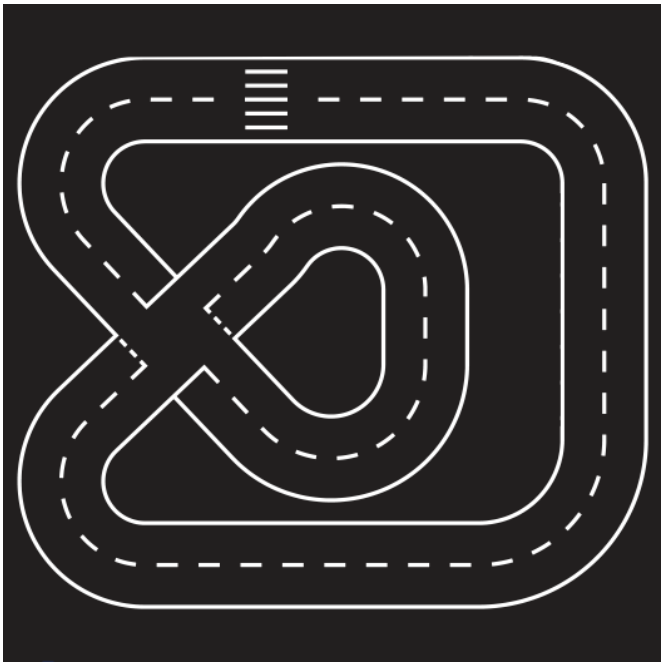

Figure 2-6 Track 1 Image

2.2.3.2 Track 2: Complex Navigation Circuit

To address the limitations of the first track, a second, more complex track was created (Figure 2-7). This track is larger and was specifically designed to provide a greater variety of driving scenarios. It incorporates multiple intersections along with additional curves with varying radii and features sequences of both left and right turns, ensuring the agent is exposed to a more diverse set of states.

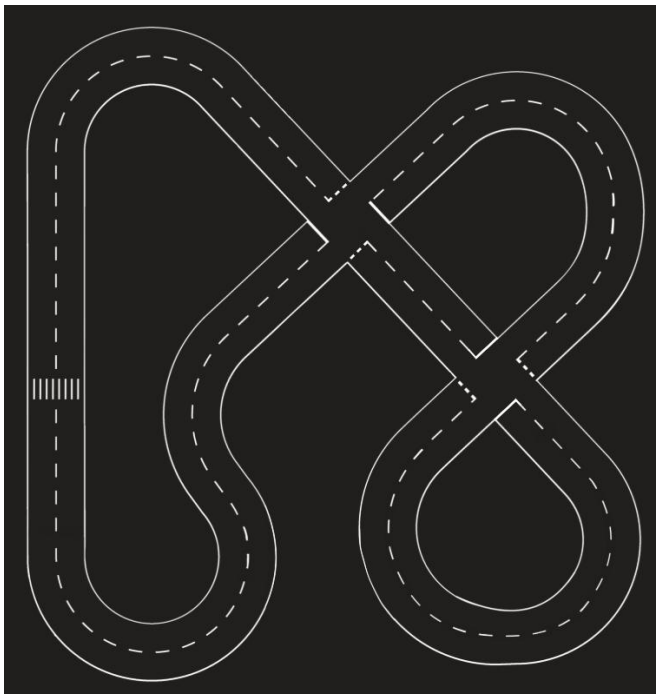

Figure 2-7 Track 2 Image

2.2.3.3 Environment Style and Episode Progression

Both tracks share a consistent high-contrast visual style, with a black base surface and white road markings. This design simplifies the visual processing task for the agent. The intersections on both tracks feature both "stop" and "give way" markings. This was a practical design choice informed by the space constraints of the eventual physical track, which would only allow for a single, multi-purpose intersection.

The training process follows a structured progression. The robot begins on one side of Track 1, after completing a loop, it is reset to the opposite side. Once that loop is complete, the agent is moved to Track 2, where it follows a similar progression before the entire cycle repeats. This ensures the agent is consistently exposed to both simple and complex environments throughout its training.

2.2.4 Reward Function

2.2.4.1 Reward Function Design

The reward function for the angular model is designed to guide the agent in following the track by providing discrete rewards for progressing through a predefined path. The system is based on dividing the track into a sequence of invisible checkpoints, an example of this is shown below in Figure 2-8.

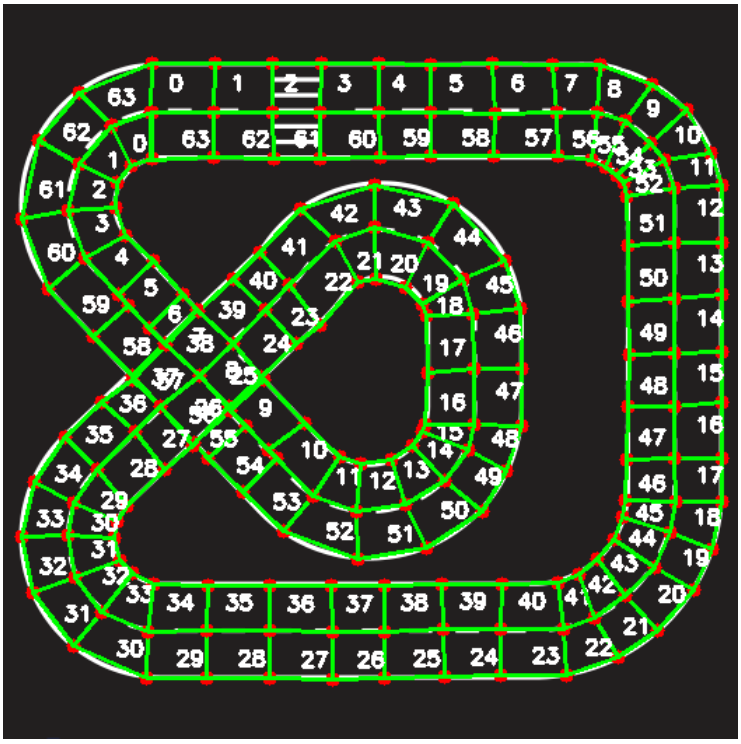

Figure 2-8 Track 1 Divided into 128 Polygons for Position Tracking

2.2.4.2 Track Discretisation with Checkpoints

To establish a baseline for the robot's position, the tracks were manually divided into a series of continuous polygonal regions, hereafter referred to as checkpoints. Each track is represented by an ordered sequence of these checkpoints, forming a defined path for the agent to follow.

The agent's position is continuously monitored relative to these checkpoints. At the start of an episode, the agent is in the first checkpoint (P_0) and its objective is to reach the next one in the sequence (P_1).

2.2.4.3 Reward and Penalty Logic

The reward signal is sparse and event-driven, based on the agent's transitions between checkpoints:

- **Successful Progression (+5 Reward):** If the robot successfully moves from its current checkpoint (P_i) into the next sequential checkpoint (P_{i+1}), it receives a positive reward of +5. The system then updates the current checkpoint to P_{i+1} and the target to P_{i+2}. This update mechanism prevents the agent from exploiting the system by repeatedly re-entering a checkpoint.
- **Stagnation (0 Reward):** No reward is given if the robot remains within its current checkpoint (P_i) without progressing.
- **Track Deviation (-10 Penalty):** If the robot's position is detected outside of both the current checkpoint (P_i) and the immediate next checkpoint (P_{i+1}), it is considered to have left the track. In this event, a large negative reward of -10 is applied.

A penalty twice the magnitude of the reward was chosen to strongly discourage the agent from leaving the track, making path-following the clearest objective during exploration.

2.2.4.4 Targeted Reset Mechanism

Following a track deviation event, the robot is reset to the position where it first entered the last successfully cleared checkpoint. This targeted reset mechanism forces the agent to repeatedly practice the specific track segment where it failed. This allows for focused learning on difficult sections without restarting the entire track.

2.3 Linear Model

2.3.1 State Space

The state, or observation, provided to the agent is designed to give it a sense of memory and motion, which is critical for interpreting speed limit signs and maintaining consistent speed. The state is a composite of two components: a history of recent visual observations and a history of recent speeds. The primary visual input is a stack of the 5 most recent camera frames, as this was found to be a good compromise, providing some memory for the agent to perceive motion without adding excessive computational overhead. Each frame is a binary image using specific threshold values to isolate the track and traffic signs while minimising background noise (Figure 7-2). This history of frames allows the agent to perceive motion and to temporarily "remember" a sign it has just passed, which is essential for learning to maintain a new speed limit. To supplement the visual data, the state also includes a history of the 5 most recent linear velocities of the robot. This provides the agent with a memory of its recent dynamic behaviour, encouraging smoother and more stable speed control.

2.3.2 Action Space and Speed Control

The action space is a continuous, one-dimensional vector representing a desired acceleration, with values ranging from -1.0 to +1.0. Instead of directly setting the robot's speed, this raw output is scaled down and then added to the robot's current speed. This updated value becomes the new target speed for the robot, ensuring smoother changes in velocity. By having the agent output acceleration instead of absolute speed, we prevent jarring and unrealistic behaviour, such as instantaneously changing from a complete stop to maximum speed.

2.3.3 Training Environment

The training environment was designed to teach the agent to recognise and react to three distinct traffic signs. It consists of three parallel, linear tracks, each featuring a specific sign and task.

2.3.3.1 Track Configuration and Tasks

The environment is composed of the following three tracks:

1. **Stop Sign Track (**Figure 7-1**):** The robot's objective is to come to a complete stop upon identifying the stop sign.
2. **10 km/h Speed Limit Track (**Figure 7-1**):** The robot must adjust its speed to match the 10 km/h limit.

3. **30 km/h Speed Limit Track (**Figure 7-1**):** The robot must adjust its speed to match the 30 km/h limit.

On all tracks, the signs were physically angled at 45 degrees toward the robot's path to improve their visibility to the camera (Figure 2-9). The agent's control is restricted to linear motion with its angular speed fixed at zero with slight adjustments to account for imperfections in the simulated motor control. Upon passing the sign on one track, the environment resets the robot onto the next track for a new episode, where it will repeat this process.

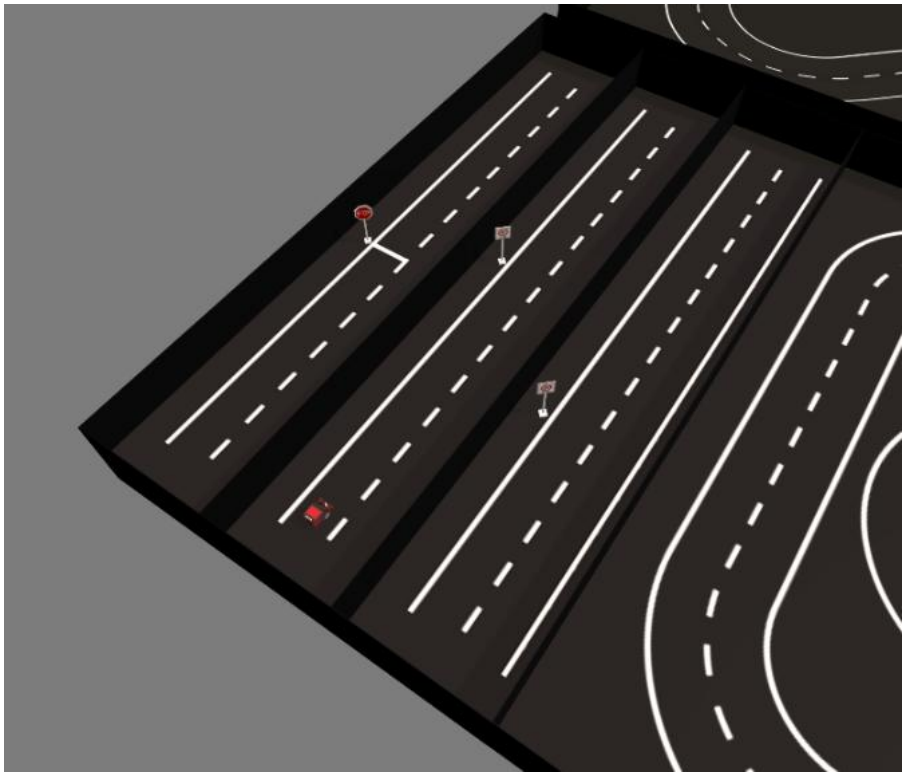

Figure 2-9 Straight Track Environment Showing Position of Signs and Robot

2.3.3.2 Randomisation for Generalisation

To prevent the agent from simply memorising to change speed based upon its position along the track, randomisation was implemented. For the stop sign track, the robot's starting position was randomised along the length of the track preceding the sign. Unlike the two speed limit tracks, where the robot started at a fixed position, but the location of the speed limit sign was randomised along the track.

This methodology introduces variation to the agent's visual input. However, it is a key limitation that the most critical learning events are concentrated in the visual frames immediately before and after the sign, regardless of the randomisation.

In addition to positional randomisation, the robot's initial speed for each episode was sampled from a uniform distribution between a lower and upper speed limit. This sampled speed was then used to populate the entire speed history buffer provided to the agent at the beginning of an episode. This approach was intended to reinforce the agent's ability to learn speed control by providing it with a history of the target speed.

2.3.4 Reward Function

2.3.4.1 Reward Function Design

To effectively teach the agent precise speed control, a multi-feature reward system was created. This system is composed of a continuous reward function based on a Gaussian distribution, a technique that has proven effective for encouraging goal-oriented progress in other complex robotics domains like multi-step manipulation [17].

2.3.4.2 Reward Function: Gaussian Function

The primary reward signal is calculated at each timestep using a Gaussian function, as shown in the equation below.

$$R(v_{current}) = A * \exp\left(\frac{-\left(v_{current} - v_{target}\right)^2}{2\sigma^2}\right) + C$$

Where the parameters were configured as follows for this setup:

- **A (Height):** The height of the Gaussian curve, this scales the peak of the reward curve.
- $\boldsymbol{v_{target}}$ **(Mean):** This value is dynamic, and changes based on the robot's position relative to upcoming signs.
- $\boldsymbol{\sigma}$ **(Standard Deviation):** The width of the curve, a smaller value creates a sharper peak, meaning the agent is more strictly penalised for small deviations from the target speed.
- **C (Vertical Shift):** The vertical shift of the entire reward curve, its purpose is to control the reward range, specifically the minimum and maximum reward the agent can receive.

The parameters of this function were adjusted to meet key design criteria aimed at stabilising the learning process. The function imposes bounded penalties, where rewards diminish for speeds far from the target to prevent destabilising negative values. It also maintains a symmetrical reward structure and a normalised range between the maximum positive reward (when $v_{current} = v_{target}$)

and the minimum reward. The full implementation can be found in the project's code repository (see Linear_Control_PPO.py in Appendix C: Code Repository).

2.3.4.3 Dynamic Target Speed

The target speed (v_{target}) used in the Gaussian function is not static. Instead, it changes dynamically based on the robot's position relative to an upcoming sign. This process is designed to encourage smooth acceleration and deceleration. When the robot is beyond a set distance from a sign, the target speed is set to the randomised speed. Once the robot enters a predefined "transition distance" from a sign (designed to coincide with the sign becoming clearly visible) the target speed begins to linearly change from the previous speed limit to the new speed limit required by the sign. After passing a speed limit sign, the target speed remains fixed at the new value for a short distance, giving the agent an opportunity to learn how to maintain a constant speed before the episode resets. This logic is modified for the stop sign, where the interpolation to is calculated to complete before the robot reaches the sign, compelling it to stop ahead of the line. To further accelerate learning, a discrete, one-time bonus reward of +2.0 is given the first time the robot's speed matches the target speed within a small tolerance.

2.4 Model Transfer

2.4.1 Open Neural Network Exchange (ONNX)

To transfer the trained models to the physical EyeBot, the Open Neural Network Exchange (ONNX) [18] format was used. This approach was vital for bridging the deployment gap, as it provided a lightweight, framework-independent model representation. By converting the PyTorch-based SB3 model, it allowed the use of ONNX Runtime on the EyeBot's Raspberry Pi, bypassing the need to install heavy dependencies like the full PyTorch [19] library.

2.4.2 Real Life Testing Environment

A physical track was constructed in the UWA robotics lab to replicate the simulated environment as seen in Figure 2-10. However, inherent differences of the track contributed to the sim-to-real gap. The physical track had surface imperfections, including minor bumps and greater friction from painted lines that were slightly raised. Additionally, the lab setting introduced significant visual noise lighting differences and from surrounding equipment, a stark contrast to the sterile, uncluttered background of the simulation.

Figure 2-10 Physical Environment in the UWA Robotics Lab

To prevent any hardware damage during testing, a safety override using the EyeBot's infrared distance sensors [20] was implemented. This system halted the robot's motors when the distance in front of the Eyebot was too low. This was used to prevent collisions with the walls, a necessary safeguard that was not required for the simulation.

3 Results & Discussion

The project successfully demonstrated the feasibility of using RL to control the low-cost EyeBot mobile robot platform, representing a significant addition to the platform's control repertoire. While the core objective of deploying and testing simulation-trained models on the physical robot was met, the investigation revealed a sim-to-real performance gap that was substantially larger than anticipated.

3.1.1 Validation of the Core Hypothesis

The primary hypothesis, that an RL agent could learn control policies in a simulated environment and be successfully deployed on the resource-constrained EyeBot was validated in principle.

The project achieved its fundamental goals by successfully developing and testing both an Angular Model (steering) and a Linear Model (speed/sign response) on the physical EyeBot. This confirmed that policies learned in a high-level Python training environment (PyTorch) could be converted and executed on the embedded system via the ONNX Runtime.

The final design, which included the ONNX conversion pathway and necessary hardware/software modifications, was successful in meeting the objective of real-world deployment. The EyeBots were controlled using the learned models, confirming the possibility of this control methodology.

3.1.2 Evolution of the Hypothesis

The original expectation that transferring the model from simulation to the real world would result in a minor, manageable performance drop was not met. The actual results necessitated a refinement of the hypothesis, shifting the measure of success from performance replication to a proof-of-concept validation.

The performance degradation was significant, particularly in the Angular Model, where a "wobble" in the model's predicted actions caused the physical robot to struggle to stay on the track. This demonstrated that while Reinforcement Learning is a viable control method, its ability to match simulated performance is highly constrained by the simulation's fidelity, the platform's embedded limitations, and the model's learned behaviour. As a result, the hypothesis evolved to accept the use of engineering heuristics as a necessary bridge across the sim-to-real gap. The final working design was not a pure end-to-end system but a hybrid one, incorporating classical control to improve stability and reliability in the physical environment

3.1.3 Comparison with the Pre-existing State of the Art

This project establishes the first demonstration of Reinforcement Learning-based control for the EyeBot platform. This work provides a documented foundation for applying adaptive, reward-based learning to future, more complex tasks on the system. The successful implementation of the PyTorch-ONNX pipeline is a crucial procedural addition. It provides a lightweight, and repeatable methodology for deploying computationally intensive, deep learning models onto resource-constrained, ARM-based embedded systems.

3.1.4 Improvement Over Existing Approaches

The new design procedure improves on existing approaches by offering a more scalable and less brittle method for generating more complex behaviours. Unlike more traditional methods which require developers to explicitly code rules for every scenario, the RL approach allows the agent to autonomously discover an optimal policy based on a high-level reward function. It proves that complex behaviours like track navigation can be learned through reward-based optimisation. While limitations forced the inclusion of heuristics, the core proof remains, RL is a viable, alternative path for developing control systems on the EyeBot.

3.1.5 Angular model

The angular model was developed to handle the robot's steering control. The results demonstrate a clear success in fundamental path-following, but also highlight significant challenges related to the sim-to-real transfer process.

3.1.5.1 Performance and Successes

The primary objective for this model was achieved; the agent successfully learned a policy to navigate custom-designed tracks with varying lengths and turn complexities. It consistently completed laps in the simulation, validating the checkpoint-based reward system and the agent's ability to learn a navigation strategy within the confines of the simulated environment.

3.1.5.2 Limitations and Challenges

A significant and unexpected issue arose from the learned control policy. The agent developed a tendency to use maximal turning actions generated a “wobble” (Figure 7-3, Figure 7-4). While this resulted in successful completion of the track, it would prove problematic during physical deployment.

3.1.6 Linear model

The linear model was designed for the more complex task of controlling longitudinal speed in response to traffic signs. The training process proved challenging, and while a fully functional model was not achieved, the results provide valuable insights into the difficulties of learning from visual cues.

3.1.6.1 Performance and Partial Successes

The model demonstrated clear evidence of learning, particularly in response to the stop sign where it could bring the robot to a halt successfully. For the 10 and 30 km/h speed limit signs, the agent showed a tendency to come to a complete stop like it does for the stop sign rather than changing speeds. Given that the visual details on the signs were clearly perceptible (as shown in Figure 7-2), this failure to differentiate suggests either insufficient training time or a suboptimal reward function. This indicates that while the agent learned to react to a sign, it did not fully learn to classify it from the visual input alone.

3.1.6.2 Limitations and Challenges

The primary challenge was enabling the agent to act based on past events, a classic problem of partial observability. An ideal solution would involve a network with explicit memory, such as SB3's Recurrent PPO (RPPO), but its high computational demand made it infeasible for both the available training hardware and deployment on the resource-constrained EyeBot.

Consequently, a pragmatic workaround was implemented by creating a manual frame stack, a history of images and speeds, within the standard PPO agent. This approach, where historical observations are stacked to provide the agent with a form of memory, aligns with established techniques for addressing partial observability in visual RL tasks. While this provided a functional, low-overhead form of memory, it is considerably less effective than a true recurrent architecture and, therefore, only partially resolves the underlying issue.

Beyond the challenge of memory, task complexity also necessitated a simpler solution. The initial goal for the stop sign included a learned two-second pause before resuming motion; however, this temporal task proved too complex for the agent. A more reliable solution was therefore implemented using a hard-coded wait command triggered by a zero-velocity action.

3.1.7 Model Transfer to the Eyebot Platform

A critical objective of this project was to transfer the models trained in simulation to the physical EyeBot robots. The successful deployment was ultimately achieved using the ONNX format, but the process presented significant technical challenges that had a major impact on the project's timeline and outcomes.

3.1.7.1 The Deployment Challenge: Bridging the Sim-to-Real Gap

The core challenge lay in executing a model trained on a powerful desktop computer within a high-level Python environment (Stable-Baselines3, PyTorch) on the resource-constrained Raspberry Pi of the EyeBot. A systematic investigation was undertaken to find a viable deployment pathway.

Several methodologies were attempted, with most proving infeasible due to the hardware and software limitations of the embedded platform:

1. **Direct Environment Replication:** The initial approach was to install the SB3 and Gymnasium libraries directly onto the Raspberry Pi. This failed due to a lack of official support and incompatible dependencies for the ARM architecture of the device.
2. **Full Framework Installation (PyTorch):** The next attempt was to install the underlying PyTorch framework. This was also unsuccessful, as the storage footprint and memory requirements of the full PyTorch library exceeded the hardware capabilities of the EyeBot.
3. **Cross-Compilation:** A more advanced strategy involving cross-compiling a custom PyTorch installation was explored. However, this process proved to be exceedingly complex and resulted in numerous failed builds.

3.1.7.2 The Successful Solution: ONNX Runtime

The successful deployment was finally achieved using the ONNX format. This involved a two-step process where the trained PyTorch model was converted into a self-contained .onnx file on the development computer. Then the lightweight ONNX Runtime was installed on the EyeBot. This high-performance inference engine is specifically designed to run ONNX models efficiently on a wide range of hardware, including ARM-based systems like the Raspberry Pi.

3.1.7.3 Impact on Project Timeline and Outcomes

The investigation into a viable deployment strategy was a significant and unforeseen time expenditure. The technical complexities and iterative trial-and-error process consumed a substantial

portion of the project's schedule. This directly impacted the development and refinement of the learning agents.

As a direct consequence of these delays, the linear model for speed control did not reach a satisfactory stage of performance, in which the model could not be physically deployed on the Eyebots. This outcome validates the initial concerns regarding time management and highlights a critical lesson in robotics research. The lesson being that challenges of deployment are not trivial and must be considered a primary factor in project planning.

3.1.8 Sim-to-Real Performance on EyeBot

While the models were successfully transferred, their real-world performance was initially insufficient, revealing a sim-to-real gap created and exacerbated by design choices made to optimise the simulation environment. This gap manifested in two primary areas: a perception gap stemming from the simplified visual inputs, and a dynamics gap related to the unmodelled physics of the physical robot and trained model deficiencies.

3.1.8.1 Challenge 1: The Perception Gap (Visual Discrepancies)

Initial tests highlighted severe inconsistencies between the agent's perception in the simulator and the real world. Two key adaptions were made the first being a hardware redesign of the camera mount**.** The original camera mount on the EyeBot was off-centre and positioned in a way that the robot's own chassis obstructed the lower portion of the view (Figure 3-1). This was a major deviation from the simulated camera. A new 3D model was designed and printed that positioned the camera further forward and more centralised, which more closely replicated the agent's viewpoint in the simulation (Figure 3-2).

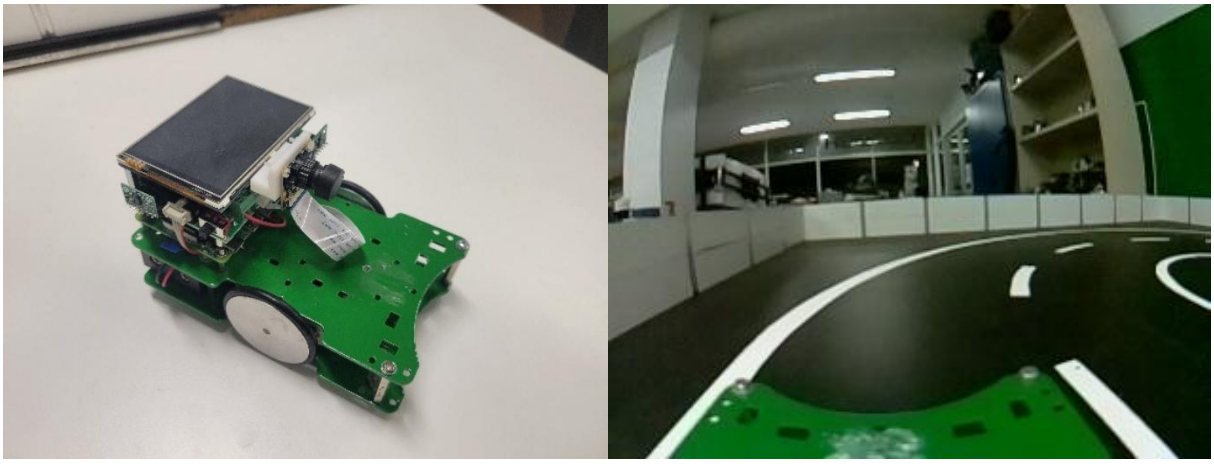

Figure 3-1 Old Camera Mount Attached to Eyebot and Old Camera Image

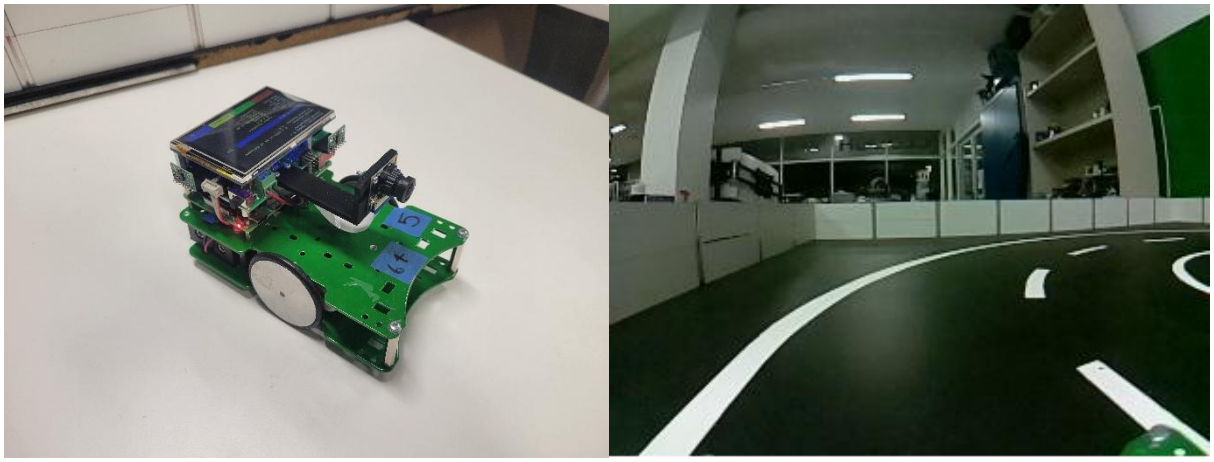

Figure 3-2 New Camera Mount Attached to Eyebot and New Camera Image

Despite these advancements, a persistent issue was the discrepancy in lens effects. The simulated fisheye lens produced a cropped, circular image (Figure 2-1), whereas the physical fisheye camera lens [22] on the Eyebot provided a full rectangular frame. This meant the model was processing images that were fundamentally different in shape from its training data. This produced more erratic behaviour of the model when tested due to the large amount of unexpected data given to the model. To remedy this situation a simple image masking code was implemented to black out the unexpected data to closely match the simulated image inputs (**Error! Reference source not found.**). However, this solution only provides a way to ensure the model receives the correct shape of data it does not ensure all the necessary data is transferred into that circular area.

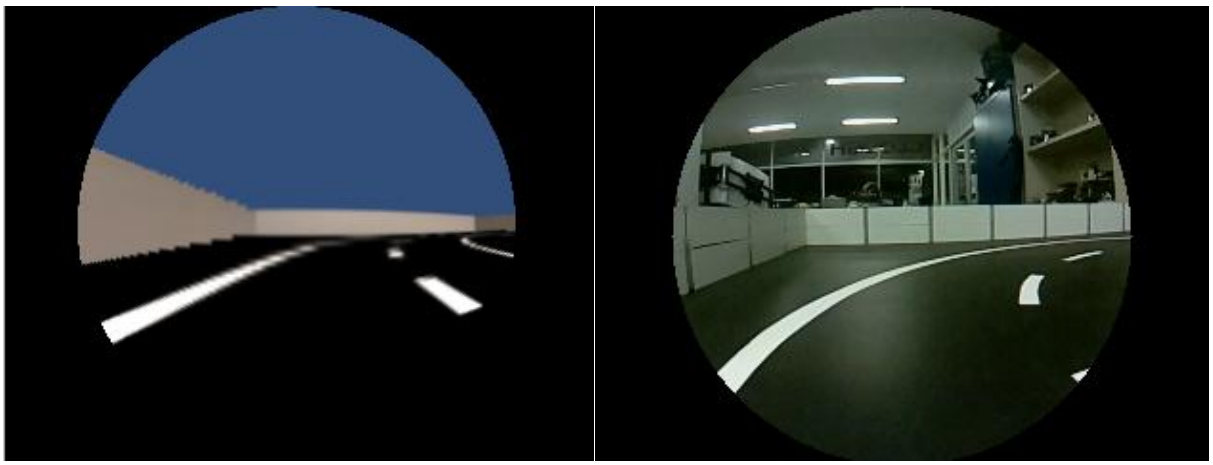

Figure 3-3 Comparison Between Simulated Image and Masked Real Image

3.1.8.2 Challenge 2: The Dynamics Gap (Control Issues)

A significant dynamics gap was evident in the robot's physical movement. The "wobble" behaviour from the angular model, while successful in simulation, became greatly amplified on the physical robot, causing severe instability that often drove it off the track. This issue was critical because the model lacked a trained recovery policy, meaning any deviation from the track resulted in a complete task failure. This is a classic sim-to-real gap problem, a well-documented challenge in robotics where discrepancies between simulated and real-world dynamics degrade performance [23].

Initial attempts to mitigate this instability by introducing a reward penalty for large changes in angular velocity proved unsuccessful, failing to yield a stable policy even after a substantially longer training period. A more effective solution was the implementation of a Proportional-Derivative (PD) [24] controller to post-process the agent's raw steering commands. A PD controller was specifically chosen for its capacity to dampen high-frequency oscillations. The proportional (P) term acts to correct the immediate steering error provided by the RL agent, while the derivative (D) term counteracts the rate of change of that error. This derivative action directly opposes the agent's rapid, jerky outputs, thereby smoothing the aggressive policy and stabilising the physical robot's response. This hybrid architecture combines the high-level pathfinding intelligence of the RL model with the classical stability afforded by the controller. While this approach showed evidence of improving the robot's performance by reducing overshoot, it only treated the symptoms of the "wobble" rather than resolving the underlying instability of the learned policy.

4 Conclusions & Future Work

4.1 Summary of Results

This project successfully served as a proof-of-concept, demonstrating for the first time the feasibility of implementing a near end-to-end Reinforcement Learning (RL) control system on the UWA EyeBot platform. The primary objectives were met RL agents were trained in simulation, and a viable PyTorch-to-ONNX deployment pipeline was established to run these models on the EyeBot's resource-constrained hardware. This represents a novel and significant addition to the platform's capabilities, moving beyond traditional control methods.

However, the investigation revealed a significant sim-to-real performance gap that challenged the initial hypothesis that a simulation-trained model would transfer with minimal degradation. The performance drop upon transfer was substantial; for instance, the Angular Model's instability was significantly amplified on the physical robot. This indicates that the discrepancies in dynamics and perception between the EyeSim simulator and the physical EyeBot are too large to be overcome by the current learning algorithm alone.

Consequently, the most effective solution was not purely learned but were hybrid systems. Real-world stability and reliability were only achieved by integrating the learned RL policy with classical control elements, such as a PD controller. This finding suggests that for low-cost robotic platforms, the optimal approach is often a blend of learned behaviours and traditional solutions.

Finally, the technical hurdles of deploying a deep learning model onto the embedded system proved to be a major, unforeseen bottleneck. While the successful ONNX pipeline is a key contribution of this project, the time dedicated to solving this challenge directly limited the time available for refining the RL agents themselves.

4.2 Recommendations for Future Work

This project has established a strong foundation and highlighted several clear paths for future investigation to build upon its successes and address its limitations.

4.2.1 Sim-to-Real Transfer

Bridging the sim-to-real gap is the most critical area for improvement. To address the dynamics gap and mitigate control instability, future work should focus on system identification. By more accurately measuring the physical EyeBot's motor characteristics and calibrating the EyeSim

simulator to match, this gap could be significantly reduced. Simultaneously, the perception gap must be addressed by harmonising the sensory data between the simulation and the physical robot.

The most straightforward solution is to reconfigure the simulation's camera to produce a full rectangular frame, mirroring the physical camera's output and ensuring the agent's training data is visually identical to its real-world operational data. A similar approach involves transforming the real-world images to conform to the simulation's circular format. Instead of masking the rectangular image, which discards corner data, the entire frame could be compressed into the circular area, thereby preserving all visual information.

4.2.2 The Angular Model

The immediate priority is to improve the model's real-world reliability by teaching it a recovery policy. The training environment should be modified to allow the agent to learn how to navigate back to the track after driving off, rather than simply ending the episode. To further enhance the model's generalisation, additional tracks can be created with a greater density of intersections and crossings, increasing the agent's exposure to these scenarios. Concurrently, an additional reward function should be designed to encourage more stable control. This function would penalise large, sharp changes in angular velocity, and reward smaller, smoother turning actions.

4.2.3 The Linear Model

To enhance the training process, a more structured methodology such as curriculum learning [25] is recommended. This approach would involve incrementally increasing task complexity. For instance, the agent could first be trained to respond to a singular speed sign. Once a satisfactory performance level is reached, additional signs would be introduced, systematically building the model's capabilities for as many features as required.

To improve learning, the agent's training should transition from basic straight-line paths to a guided exploration of the full track. By using a pre-existing angular control model or a simple program to handle navigation, the robot can be guided through complex scenarios like turns, curves, intersections, and crossings. This allows the agent to focus entirely on the more complex challenge of correctly identifying and reacting to traffic signs within a relevant context.

Finally, to address the challenge of resuming the correct velocity after a stop, the state representation should be augmented. Specifically, it should be modified to include a memory of the last observed speed limit, enabling the agent to recall and reinstate the appropriate speed after executing a stop command.

Exploring more advanced recurrent architectures like Recurrent PPO (RPPO) could provide a more reliable solution to the memory challenges faced by the Linear Model. This would require access to powerful hardware and a substantial upgrade to the EyeBot's computational hardware but would represent a significant leap in the platform's capability for learning complex.

5 References

[1] T. Bräunl, “EyeBot 8 Mobile Robot Controller,” *Roblab.org*, 2025. https://roblab.org/eyebot/

[2] T. Owen, “Introduction to Robotics: Mechanics and Control by John J. Craig AddisonWesley Publishing Company, Massachusetts, USA, 1986, (£17.95, Student Hardback edition),” *Robotica*, vol. 6, no. 2, pp. 164–165, 1988, doi: https://doi.org/10.1017/S0263574700004045.

[3] J. Kober, B. J. Andrew, and J. Peters, “Reinforcement Learning in robotics: a Survey,” *INT J ROBOT RES*, vol. 32, no. 11, pp. 1238–1274, 2013, doi: https://doi.org/10.1177/0278364913495721.

[4] “EyeSim - Mobile Robot Simulator,” *Uwa.edu.au*, 2025. https://robotics.ee.uwa.edu.au/eyebot5/doc/sim/sim.html

[5] Z. Xiao, *Reinforcement Learning : Theory and Python Implementation*, First edition. Singapore: Springer, 2024. doi: https://doi.org/10.1007/9789811949333.

[6] Neil and D. A. Vidal, “A Comparative Study of Deep Reinforcement Learning Models: DQN Vs PPO Vs A2C,” 2024, doi: https://doi.org/10.48550/arxiv.2407.14151.

[7] D. Ma, Y. Qiu, J. Zhang, and J. Chen, “Reinforcement Learning and Autonomous driving: Comparison between DQN and PPO,” Melville: American Institute of Physics, 2024. doi: https://doi.org/10.1063/5.0215570.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Openai, “Proximal Policy Optimization Algorithms,” Aug. 2017.

[9] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, “End-to-End Deep Reinforcement Learning for Lane Keeping Assist,” 2016, doi: https://doi.org/10.48550/arxiv.1612.04340.

[10] A. Shakerimov, T. Alizadeh, and Varol, Huseyin Atakan, “Efficient Sim-to-Real Transfer in Reinforcement Learning through Domain Randomization and Domain Adaptation,” *Access*, vol. 11, pp. 1–1, 2023, doi: https://doi.org/10.1109/ACCESS.2023.3339568.

[11] Raspberry Pi, “Raspberry Pi 4 Model B Specifications,” *Raspberry Pi*. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[12] The Debian Project, “Debian -- Debian ‘Bullseye’ Release Information,” *Debian.org*, 2024. https://www.debian.org/releases/bullseye/

[13] T. Bräunl, R. Keat, and M. Pham, “RoBIOS - Mobile Robot Library,” *Roblab.org*, 2025. https://roblab.org/eyebot/robios.html (accessed Oct. 14, 2025).

[14] F. Wege, P. Göttsch, Examiner, H. Werner, and T. Bräunl, “Domain Adaptation and Meta-Learning for End-to-End Autonomous Driving,” 2020. Accessed: Oct. 14, 2025. [Online]. Available: https://roblab.org/theses/2020-End2End-Wege.pdf

[15] Stable Baselines3, “Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations - Stable Baselines3 1.3.1a2 Documentation,” *stable-baselines3.readthedocs.io*. https://stable-baselines3.readthedocs.io/en/master/index.html

[16] Farama Foundation, “Gymnasium Documentation,” *gymnasium.farama.org*, 2025. https://gymnasium.farama.org/index.html

[17] S. Kumra, S. Joshi, and F. Sahin, “Learning Robotic Manipulation Tasks via Task Progress Based Gaussian Reward and Loss Adjusted Exploration,” *LRA*, vol. 7, no. 1, pp. 534–541, 2022, doi: https://doi.org/10.1109/LRA.2021.3129833.

[18] The Linux Foundation, “ONNX | Home,” *onnx.ai*, 2019. https://onnx.ai/

[19] PyTorch, “PyTorch,” *Pytorch.org*, 2023. https://pytorch.org/

[20] Pololu, “Pololu - VL53L0X Time-of-Flight Distance Sensor Carrier with Voltage Regulator, 200cm Max,” *www.pololu.com*, 2025. https://www.pololu.com/product/2490

[21] K. Jiang, Q. Wang, Y. Xu, and H. Deng, “ObservationTimeAction Deep Stacking Strategy: Solving Partial Observability Problems with Visual Input,” in *IJCNN*, IEEE, 2024, pp. 1–8. doi: https://doi.org/10.1109/IJCNN60899.2024.10650736.

[22] Seeed Studio, “OV5647-160 FOV Camera Module for Raspberry Pi 3B+4B, Suitable for Large or Night Landscape surveillance, Support IR,” *Seeedstudio.com*, 2025. https://www.seeedstudio.com/OV5647-160-FOV-IR-Camera-module-for-Raspberry-Pi-3B-4B-p-5485.html (accessed Oct. 15, 2025).

[23] W. Zhao, J. P. Queralta, and T. Westerlund, “SimtoReal Transfer in Deep Reinforcement Learning for Robotics: a Survey,” *arXiv.org*, 2021, doi: https://doi.org/10.48550/arxiv.2009.13303.

[24] BräunlT., *Embedded robotics : from mobile robots to autonomous vehicles with Raspberry Pi and Arduino*, Fourth edition. Singapore: Springer, 2022. doi: https://doi.org/10.1007/9789811608049.

[25] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, “Curriculum Learning: A Survey,” *Int J Comput Vis*, vol. 130, no. 6, pp. 1526–1565, 2022, doi: https://doi.org/10.1007/s1126302201611x.

6 Appendix A: Literature Review

Reinforcement Learning (RL) has become a cornerstone of modern artificial intelligence, providing a mathematical framework for learning optimal behaviour through trial-and-error. At its core, RL involves an agent (the decision-maker) interacting with an environment by taking actions in various states and receiving rewards in return [5]. The agent's goal is to learn a policy, a mapping from states to actions, that maximises its cumulative reward over time [5]. A foundational resource in this domain is Zhiqing Xiao's book *Reinforcement Learning: Theory and Python Implementation*, which systematically explores both classical and deep reinforcement learning (DRL) techniques, emphasising their mathematical underpinnings [5].

6.1 Deep Reinforcement Learning Algorithms

DRL algorithms leverage deep neural networks to approximate the components of the RL problem, allowing them to handle complex, high-dimensional environments like video games or autonomous driving simulations [5]. Key DRL methodologies include value-based, policy-based, and actor-critic methods.

6.2 Value-Based Methods: Deep Q-Networks (DQN)

Value-based algorithms focus on learning a value function, which estimates the expected return from a given state or state-action pair [5]. Deep Q-Networks (DQN) are a prime example of this approach. This method uses a deep neural network to approximate the optimal action-value function which represents the maximum expected future reward achievable by taking an action a in state s [5]. By learning an accurate Q-function, the agent can derive a policy by simply selecting the action with the highest Q-value in any given state [5]. A key innovation in DQN is the use of an experience replay buffer, which stores past transitions and samples from them to break correlations in the data and stabilise training [5].

A recent comparative study analysing DRL performance in the *BreakOut* Atari environment found that DQN has high performance potential, but its effectiveness can be sensitive to hyperparameter tuning [6]. Similarly, another study on autonomous driving notes that while DQN can learn effectively, its performance and stability can be lower than policy-based methods in complex driving tasks [7].

6.3 Policy-Based & Actor-Critic Methods: A2C and PPO

In contrast, policy-based methods learn the policy directly without first learning a value function [5]. A popular hybrid of these approaches is the Actor-Critic framework. These models use two networks [5]:

- The Actor is the policy, which decides which action to take.
- The Critic is a value function, which evaluates the states visited by the actor, providing feedback to improve the policy.

Advantage Actor-Critic (A2C) is an efficient, synchronous implementation of this framework. It is noted for its computational speed but can sometimes exhibit higher variability and instability during training due to its direct and continuous policy updates [6].

Proximal Policy Optimisation (PPO) builds upon this actor-critic foundation and has become a widely adopted algorithm due to its stability and performance [8], [5]. As detailed in Schulman et al.'s seminal paper, "Proximal Policy Optimisation Algorithms" [8], PPO's key innovation is its method for constraining policy updates. It uses a clipped surrogate objective function, which discourages the policy from changing too drastically in a single update step by clipping the probability ratio between the new and old policies [8]. This simple but effective mechanism prevents destructive, large updates and ensures more stable and reliable learning [8]. This conceptual advantage translates directly to practical performance; one comparative study concludes that PPO offers a balance between adaptability and stability [6], while another driving simulation found PPO to have a superior ability to generalise to unseen traffic patterns [7]. PPO's combination of performance, ease of implementation, and sample efficiency has made it a benchmark for DRL research [8].

6.4 Application to Autonomous Navigation

Applying these DRL algorithms to mobile robot navigation, specifically for tasks like lane following, has become a significant area of research. In this context, the RL agent learns a control policy that maps sensor inputs, such as images from a forward-facing camera, directly to control outputs like steering angles or wheel velocities [9]. The reward function is typically designed to encourage the robot to stay within lane boundaries, maintain a certain speed, and drive smoothly [7].

In their work on end-to-end learning for lane keeping, El Sallab et al. demonstrated the feasibility of this approach by applying DRL to the task of autonomous manoeuvring in a simulated racing

environment [9]. The study highlights a critical distinction in algorithm choice based on the nature of the vehicle's control system: the action space [9].

For discrete action spaces, where the agent chooses from a limited set of commands (e.g., "turn left," "go straight," "turn right"), algorithms like DQN are suitable [9]. However, the study found that this approach can lead to abrupt, jerky movements as the agent switches between distinct actions [9].

For continuous action spaces, which allow for fine-grained control over outputs like a specific steering angle, the authors employed a Deep Deterministic Actor-Critic (DDAC) algorithm [9]. This method produced significantly smoother and more effective control, particularly on curved sections of the track, proving to be better suited for realistic driving tasks [9].

This research underscores that while various DRL algorithms can solve navigation tasks, the choice of algorithm, particularly whether it handles discrete or continuous actions, has a profound impact on the quality and smoothness of the resulting driving behaviour [9].

6.5 The Sim-to-Real Transfer Challenge

A primary motivation for using simulators is that training RL agents on physical robots is often costly, time-consuming, and can risk damaging the hardware [10]. Simulators provide a safe and efficient way to collect the vast amount of data needed for training. However, policies trained exclusively in simulation often fail when transferred to a physical robot [10]. This issue, known as the "sim-to-real gap" arises from discrepancies between the simulated environment and the real world, which can include differences in sensor noise, lighting conditions, and the physics of motors and friction [10].

To bridge this gap, researchers have developed several techniques. One of the most effective and widely used methods is domain randomisation. This technique involves training the agent in a simulation where key environmental and physical parameters are randomised during each training episode [10]. As described by Shakerimov et al., this can include randomising dynamics parameters like the mass or length of system components, as well as visual properties like lighting, textures, and colours [10]. By exposing the agent to a wide variety of simulated conditions, the resulting policy becomes less likely to overfit to the specific characteristics of any single simulated environment [10]. This process forces the agent to learn the essential features of the task rather than superficial details of the simulation, thereby improving its ability to generalise to the real world [10]. However, a key challenge of domain randomisation is that it requires expert knowledge to

define appropriate randomisation ranges; if the range is too narrow, the agent may not generalise, and if it is too wide, the policy may become overly conservative or fail to converge [10].

7 Appendix B: Additional Images

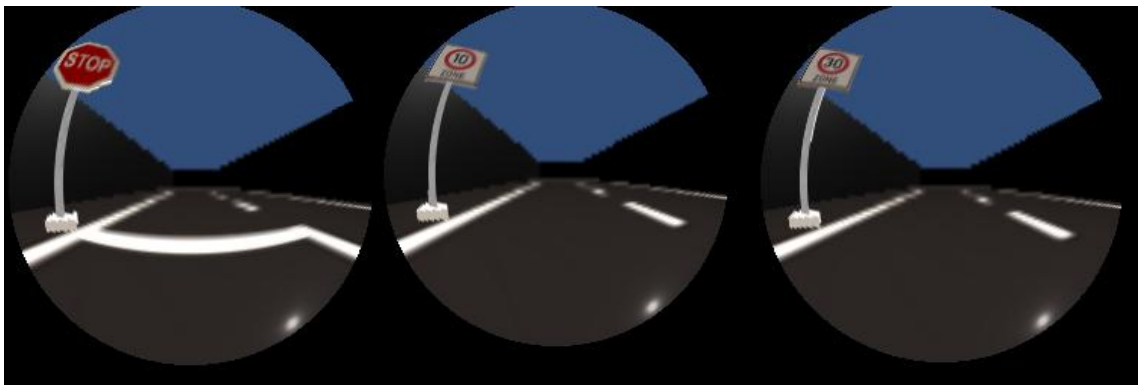

Figure 7-1 Simulated Signs in RGB Format

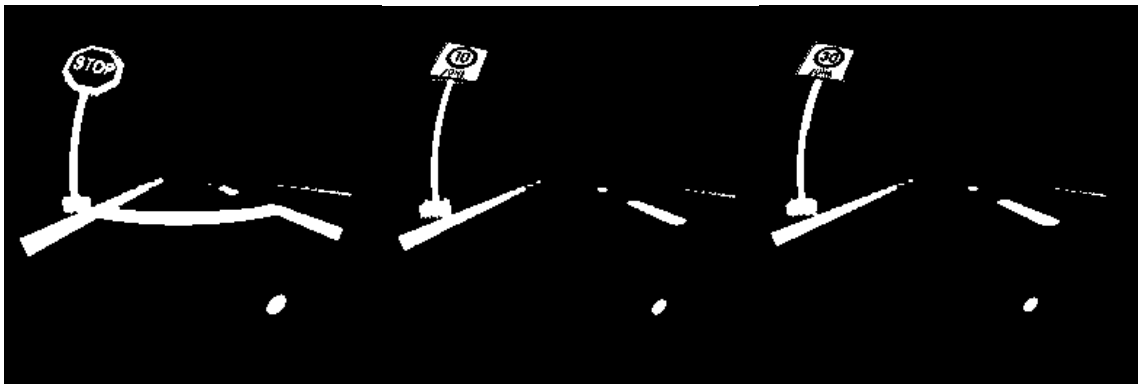

Figure 7-2 Simulated Signs in Binary Format

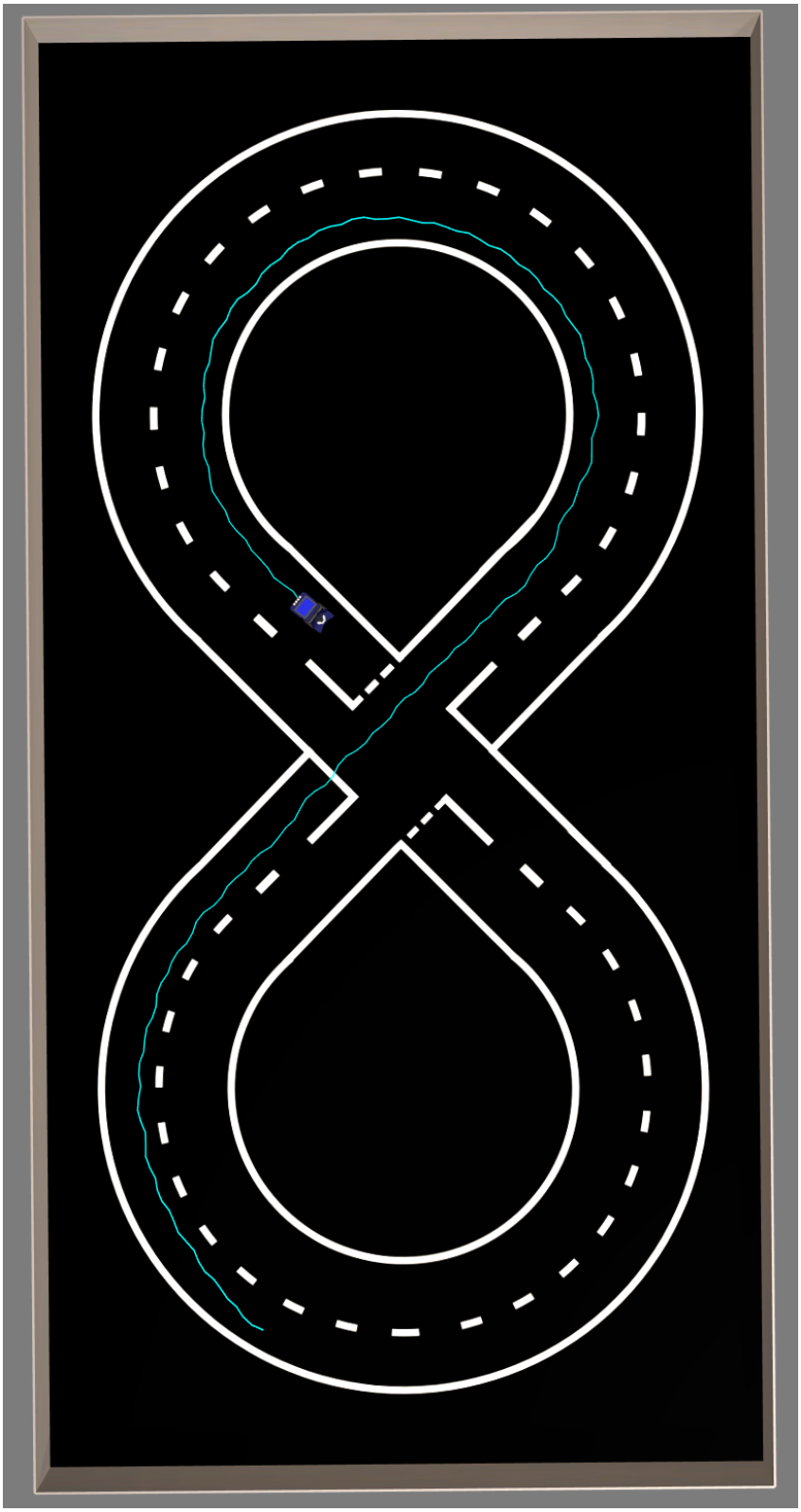

Figure 7-3 Angular Model Running in Eyesim Showing "wobble" during Driving of the Right Lane

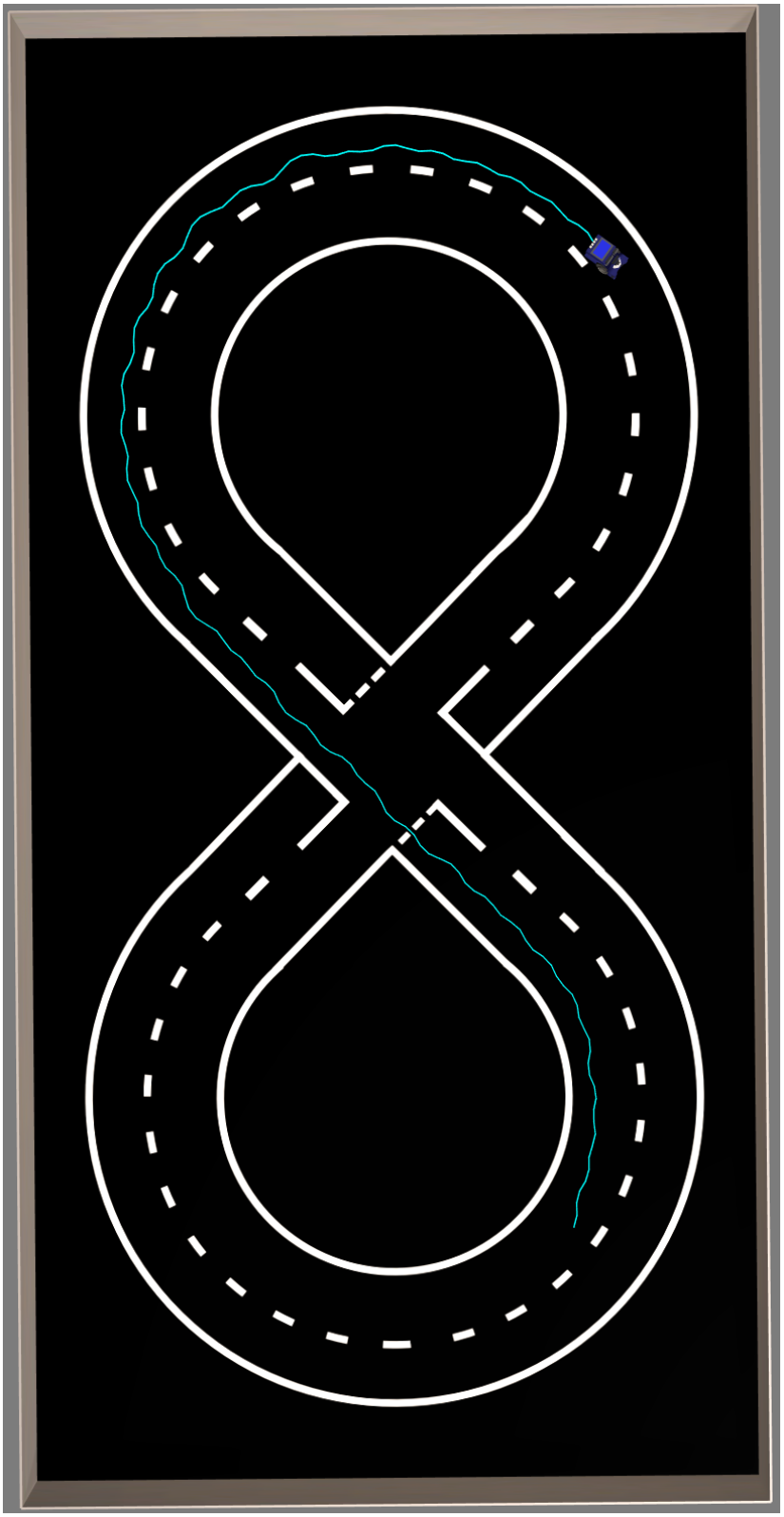

Figure 7-4 Angular Model Running in Eyesim Showing "wobble" during Driving of the Left Lane

8 Appendix C: Code Repository

The complete source code for this project, including the custom Gymnasium environments, training scripts, model conversion utilities, and the final deployment code for the EyeBot, is publicly available.

The repository is hosted on GitHub to ensure accessibility and support the reproducibility of this research.

URL: https://github.com/NoahWMueller/Eyesim