GENG4412 Engineering Research Project Part 2
Final Report

Deep Learning for Mobile Robots

Bennett B. Nicholson

23368495

School of Engineering, University of Western Australia

Supervisor: Thomas Bräunl

School of Engineering, University of Western Australia

Word count: 6760

School of Engineering

University of Western Australia

Submitted: 18/10/2025

Synopsis

This project investigates whether a compact, on-policy deep reinforcement learning (DRL) controller can learn end-to-end lane keeping from monocular grayscale images in simulation and how far that competence transfers to an EyeBot-8 mobile robot. The study is motivated by educational and research needs for reproducible autonomy stacks that run on modest hardware, and by the practical question of how computation limits, sensing noise, and actuation non-idealities shape reliability on small platforms. Rather than seeking to surpass carefully tuned classical baselines, the work maps the boundary conditions under which a PPO-trained vision policy is viable and identifies the design choices that most affect that boundary.

Lane keeping is formalised as a partially observed Markov decision process with a single continuous steering action at fixed forward speed. The course is marked with progress boxes, and the robot only advances when it enters the next box. This gives clear feedback while treating intersections as unmarked areas to avoid mistakes.

Evaluation prioritises the fraction of boxes traversed per episode (normalised progress), reported over a catalogue of spawn poses and multiple seeds. In EyeSim, policies learn straight-segment and curve tracking reliably and begin to traverse the unsigned intersection straight-through in both directions. Failures cluster in the box immediately before the intersection, consistent with the metric definition and with the policy's tendency to "chase" a vanishing curb when entering the unmarked region. After export to ONNX, simulated behaviour improves at the junction while performance at the pedestrian crossing degrades, likely due to small closed-loop timing and numeric differences that smooth rapid turns in one context and exacerbate stripe-induced overcorrections in the other. Median progress rises modestly, but crosswalk failures remain the dominant residual mode.

The move to the real EyeBot-8 didn't go well. The robot often couldn't keep a steady path because of: (i) slower and less predictable processing on the robot's CPU, (ii) differences in how the robot steers compared to the simulation, (iii) differences in lighting and camera between the simulation and real world, and (iv) the simple control system that doesn't allow slowing down for stability. These issues caused the robot to act unpredictably on the hardware, even though it performed decently in simulation.

A practical constraint runs through the study: EyeSim executes a single instance at approximately real time. Even well-tuned runs require 4–6 hours to reach useful competence, and many exceed 24 hours. This throttles ablations, reduces seed counts, and encourages conservative, sample-efficient choices over broader exploration. Building the PPO pipeline from scratch minimised the deployment footprint but delayed experimentation relative to mature libraries that would have accelerated the path to analysis.

Future work should migrate to a machine-learning-first stack such as Gym-Duckietown for simulation (parallel, faster-than-real-time rollouts) and Duckiebot hardware with embedded GPU acceleration, enabling low-jitter inference at meaningful control rates. Experimentally, results should be reported with sufficient seeds, targeted resets that concentrate experience on difficult segments, and systematic ablations of reward weights, steering limits, and observation processing. Taken together, these steps would convert the present partial success in simulation into reliable, reproducible on-robot lane keeping and provide a durable baseline for subsequent cohorts.

Acknowledgements

I would like to express my thanks to my supervisor Professor Thomas Bräunl for trusting me with this project. My experience has grown immensely thanks to you. I am also extremely grateful to Professor Norbert Oswald for providing helpful and relevant advice. Furthermore, a debt of gratitude is owed to Noah Mueller – without whom I would have had to wrangle the EyeSim libraries by myself.

This project would never have been completed without my parents supporting me, you mean the world to me.

Last but certainly not least, I would like to express my thanks, love, and gratitude to my girlfriend – Paola, for supporting and encouraging me throughout the entirety of university.

Table of Contents

DECLARATION OF CONTRIBUTION ii

Synopsis iii

Acknowledgements iv

List of Figures vii

Nomenclature vii

1. Introduction 1

  1.1 Background 2

  1.2 Project Objectives and Anticipated Benefits 2

2. Design Approach 4

  2.1 Problem Formalisation and Model Derivation 4

  2.2 Simulation Platform and Configuration 5

  2.3 Observation Space and Preprocessing 7

  2.4 Action Space and Execution Limits 8

  2.5 Reward Shaping and Penalties 8

    2.5.1 Geometric Shaping 8

    2.5.2 Control Effort 9

    2.5.3 Boundaries and Progress 9

  2.6 Policy Network Architecture 10

  2.7 Training Procedure and Hyperparameters 10

  2.8 Software Tools and Export path 11

  2.9 Evaluation Metrics and Experimental Design 11

  2.10 Deployment Tests and Validation 12

  2.11 Health, Safety, and Ethical Considerations 13

  2.12 Limitations 13

3. Results & Discussion 14

  3.1 Evaluation Metric and Headline Simulation Behaviour 14

  3.2 Attempted Sim-to-Real Transfer on EyeBot-8 via ONNX 16

  3.3 Training Throughput, Stability, and the Implications of EyeSim's execution model 17

  3.4 Comparison with Prior Work and with Alternative Platforms 18

  3.5 Limitations, Arbitrary Choices, and Sensitivity 19

  3.6 What the Results Mean for the Original Objectives 20

4. Conclusions & Future Work 22

References 25

Appendix A. Extended Literature Review 29
A.1 Scope and Aim 29
A.2 Search Strategy and Data Collection 29
A.3 Principal Sources and Inclusion Rationale 29
A.4 Screening and Evaluation 29
A.5 Thematic synthesis 30
A.5.1 Classical Lane Keeping 30
A.5.2 Behavioural Cloning to Deep RL 30
A.5.3 PPO for Image-to-Control 30
A.5.4 Reward Shaping and Evaluation for Long-Horizon Navigation 31
A.5.5 Visual Design on Small Robots 31
A.5.6 Simulation Platforms 31
A.5.7 Sim-to-Real 32
A.5.8 Embedded Inference 32
A.5.9 Multi-Robot RL 32
A.6 Implications for Method and Evaluation 32
Appendix B. torchTraining.py 34
Appendix C. modelConversion.py 50
Appendix D. runONNXModel.py 52

List of Figures

Figure 2.1: Carolo Cup Simulation in EyeSim. 5
Figure 2.2: Carolo Cup Simulation in EyeSim with markers used to determine box vertices in outer lane, a simulated EyeBot is inside Box 1. 6
Figure 2.3: Carolo Cup Simulation in EyeSim with markers used to determine box vertices in inner lane, a simulated EyeBot is inside Box 1. 6
Figure 2.4: Image of an EyeBot in simulation on the Carolo Cup course, the image visible to the EyeBot is displayed on the LCD. 7
Figure 2.5: Visualisation of the end-to-end neural network. 10
Figure 3.1: Graph showing the percentage of episodes in which the ONNX model failed per box. 14
Figure 3.2: Graph showing the percentage of episodes in which the PyTorch model failed per box. 15

Nomenclature

PPO	Proximal Policy Optimization
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
GAE	Generalised Advantage Estimation

1. Introduction

Reliable lane keeping is a foundational capability for autonomous ground vehicles, underpinning safety, throughput, and energy efficiency in structured environments from public roads to factories and campuses. On compact, education-oriented platforms, the problem is sharpened by sparse sensing, modest onboard processing power, and limited actuation fidelity. These constraints make it impractical to deploy heavyweight perception stacks or multi-sensor fusion while still demanding predictable behaviour under variable conditions. The present study examines whether a modern on-policy deep reinforcement learning (DRL) agent can learn effective, end-to-end lane keeping in simulation and how far that competence transfers to a physical EyeBot-8 robot with a Raspberry-Pi microcontroller. The intent is not to claim superiority over well-tuned classical baselines on identical hardware, but to identify the boundary conditions under which a PPO-trained, end-to-end controller is viable, and to document the changes that most affect that boundary.

This project is relevant to multiple stakeholder groups. For engineering education programs and student teams, a compact, reproducible learning-based controller that runs on commodity hardware can enable credible lab exercises and competitions without bespoke tuning for each track. Considering researchers interested in sim-to-real transfer, the setting provides a tightly scoped testbed to study observation shift, control-loop latency, actuator non-idealities, and runtime effects introduced by model export and inference backends. For organisations considering small autonomous ground vehicles – such as warehouse, factories, or the mining industry – the findings help delineate what learning-based policies require to remain stable under compute and sensing limits, when to combine learned perception with classical control, and where model compression or runtime selection becomes the dominant constraint.

The motivation for attempting an end-to-end learning approach rather than a classical lane-detection-plus-controller pipeline is twofold. First, an end-to-end agent can, in principle, internalize the coupling between perception and control under the platform's specific optics, lighting, and dynamics, which often dominate failure modes in small robots. Second, the study aims to expose and measure precisely where such a policy fails under real-world constraints, thereby informing whether and how to combine learned and classical elements in future iterations.

1.1 Background

The key findings from the accompanying literature review (See Appendix A) are as follows. First, compact convolutional encoders with grayscale input often suffice for lane keeping at miniature scales, provided that the reward landscape strongly couples lateral error and heading. Second, PPO is a pragmatic algorithmic choice for on-policy training in simulation because it tolerates short rollouts and modest batch sizes while handling continuous actions without complex target networks. Third, completion-centric evaluation is preferable to absolute reward when reward terms are heterogeneous and shaped, and it aligns with competition-style success criteria such as those of the Carolo Cup. Fourth, sim-to-real transfer on low-cost platforms is dominated by sensing and execution effects – camera photometrics, timing jitter, actuator friction and saturation – so pre-deployment measures should prioritise photometric augmentation, latency profiling, and closed-loop tests with the exact deployment runtime. These points inform the method and evaluation choices detailed in the subsequent sections and delimit the claims that can reasonably be made from the evidence gathered here.

1.2 Project Objectives and Anticipated Benefits

The project set out to design, train, and evaluate an end-to-end reinforcement-learning controller for lane keeping in EyeSim on a Carolo Cup track, then export the trained policy to a lightweight runtime for execution on an EyeBot-8 embedded mobile robot. The overarching objective was to maximise lap completion in simulation under non-deterministic physics while maintaining a compact network that could plausibly run on constrained hardware. Success was defined and measured primarily by completion percentage across repeated episodes drawn from diverse spawn locations, with secondary indicators including intersection and pedestrian-crossing segment success, lane-departure incidence, and failure location.

The five key objectives of the project were to:

1. Implement a reproducible PPO training pipeline that consumes grayscale camera observations of shape 1×120×160 (NCHW) and outputs a continuous steering command.
2. Shape the reward so that long-horizon completion became learnable without compromising policy validity.
3. Export the trained PyTorch model to ONNX and execute it on the robot using a lower-weight runtime, thereby measuring and analysing the simulation-to-hardware performance gap.

4. Situate the chosen algorithm among viable alternatives and justify the selection in the face of platform and problem constraints.
5. Define, track, and report a small set of decision-relevant metrics for stakeholders who may wish to reuse or extend the work.

These objectives advance the state of practice for EyeSim-based lane keeping in two ways. First, they consolidate a completion-centred methodology that treats reward as a means to a task-level end rather than as an end in itself, thereby avoiding misleading comparisons based on shaped reward sums. Second, they explicitly include the export-and-deploy step to a severely resource-limited platform, which surfaces the practical obstacles that often go unreported in simulation-only studies and frames a concrete agenda for closing the gap. The project therefore lays groundwork for continuing studies that can incorporate stronger photometric augmentation, domain and dynamics randomisation, latency-aware control policies, and hardware-in-the-loop training, with the ultimate goal of reliable on-robot lane keeping.

From the perspective of benefits analysis, the consequences of meeting these objectives are material for several entities. A teaching laboratory gains a low-cost, reproducible experiment that immerses students in the full stack from perception to control to deployment, improving attainment of program-level outcomes in machine learning for control and embedded systems. The EyeSim and EyeBot developers obtain a benchmarked reference that highlights where simulator fidelity and runtime tooling most affect task success, aiding prioritisation of development resources. Student teams preparing for competitions with Carolo-style rules benefit from quantitative guidance on reward shaping, steering-limit scheduling, and the hazards of inference-time mismatch between training and deployment frameworks. Conversely, the negative consequences of failing to address the objectives – most notably the export-time and runtime mismatches – include wasted integration effort on hardware, poor reproducibility across labs, and misleading expectations based on simulation-only results. By articulating these objectives and the associated metrics, the work supports more efficient allocation of time and equipment in future investigations and provides a defensible baseline against which incremental improvements can be assessed.

2. Design Approach

The problem addressed is continuous steering from monocular grayscale video on a Carolo Cup miniature road course that includes straights, curves, an intersection, and a pedestrian crossing. The agent receives a single 160×120 image per control step and outputs a normalized steering command, while the longitudinal speed is held constant to isolate lateral control. Training is conducted in EyeSim, a physics-based simulator distributed with the EyeBot ecosystem. EyeSim introduces stochasticity through contact dynamics and sensor modelling rather than running as a strictly deterministic state machine; its perturbations are milder than those of the physical robot but sufficient to create non-repeating trajectories and occasional near-misses or stalls under the same nominal seed. This non-determinism is essential context for both the training design and the interpretation of results.

2.1 Problem Formalisation and Model Derivation

Lane keeping was cast as a partially observed Markov decision process. At each control step t, the agent received a grayscale forward camera image $o_t \in [0,1]^{1\times120\times160}$ and produced a single continuous steering command $a_t \in \mathbb{R}$ expressed as a wheel-angle in degrees. The environment returned a shaped scalar reward r_t and transitioned to o_{t+1}. Episodes terminated on collision, leaving the drivable corridor, or after a fixed horizon. The EyeSim dynamics are stochastic due to contact modelling and sensor noise, so returns are random variables rather than deterministic functions of the policy.

The decision was made to implement a PPO from scratch to reduce the number of libraries that would need to be run on the EyeBot. PPO was selected because it offers stable on-policy updates with clipped probability ratios and advantages computed via Generalised Advantage Estimation (GAE). A squashed Gaussian parameterisation suited to bounded steering actuators was used as the policy. The network predicted a pre-squash mean $\mu_u \in \mathbb{R}$ and a global log-standard-deviation parameter $\log(\sigma_u \in \mathbb{R}$ for a one-dimensional normal $u \sim \mathcal{N}(\mu_u, \sigma_u^2)$. The executed command was $a = \text{limit}_t \cdot \tanh(u)$, where limit_t is a training-time steering bound scheduled from 45° to 90°. During optimisation the correct change-of-variables term for the tanh transformation was included in the log-likelihood used by PPO, and value-function regression shared the convolutional trunk. This "squashed-Gaussian PPO" is a small but material modification of the textbook PPO parameterisation, introduced here to align the policy's support with the actuator's physical range and to avoid pathological exploration at large angles.

2.2 Simulation Platform and Configuration

As the EyeBot-8 is developed inhouse at UWA, it was chosen as the hardware platform to test the final agent on. EyeSim was therefore chosen as the physics and rendering engine because it models the EyeBot-8 platform and exposes RoBIOS calls, enabling the same control interface in simulation and on hardware.

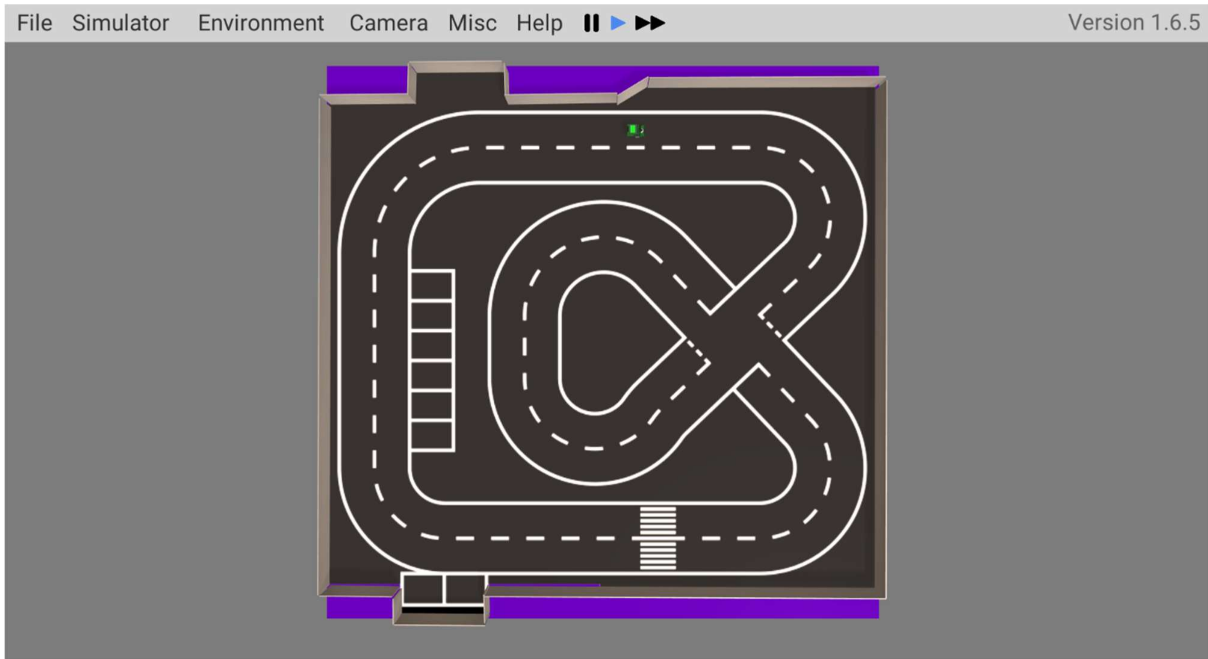

Figure 2.1: Carolo Cup Simulation in EyeSim.

The closed track used for training was the "Carolo Cup" map bundled with the EyeSim examples on roblab.org, see figure 2.1. The simulator introduces non-determinism through its physics and sensor models but is gentler than the real platform in how quickly small errors compound into failures.

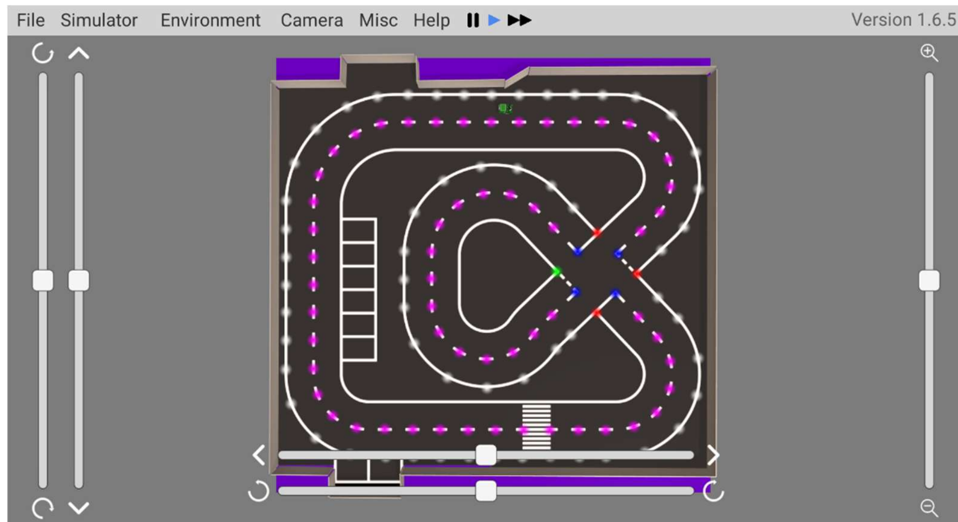

Figure 2.2: Carolo Cup Simulation in EyeSim with markers used to determine box vertices in outer lane, a simulated EyeBot is inside Box 1.

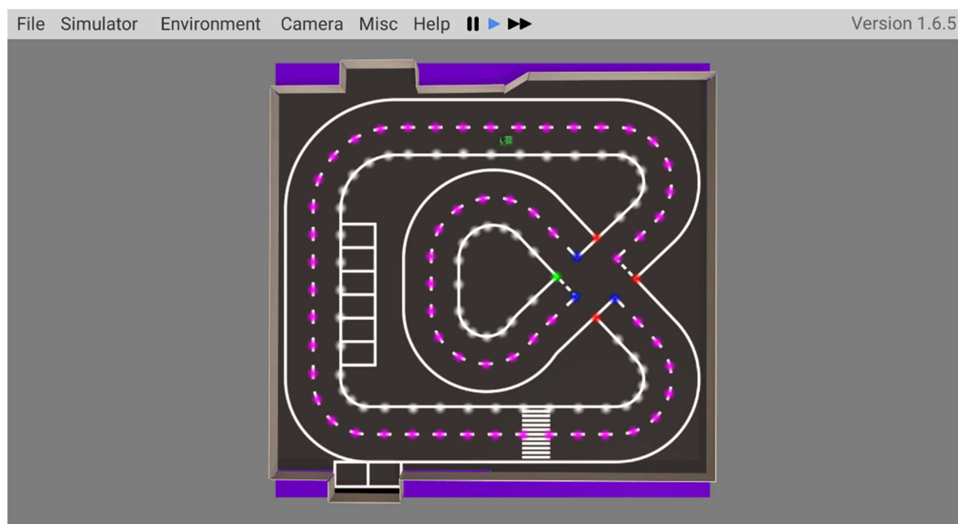

Figure 2.3: Carolo Cup Simulation in EyeSim with markers used to determine box vertices in inner lane, a simulated EyeBot is inside Box 1.

To make progress measurable and the reward dense, the track was tiled with an ordered set of convex “progress boxes” (see figures 2.2 and 2.3) stored in CSV files and loaded at runtime. At any time, the agent occupied at most one box. Progress advanced only when the agent entered the next box in sequence, so the progress index traced a monotonically increasing path around the circuit for successful runs. A small catalogue of spawn states was defined on both the outer and inner lanes to diversify early experience and to make evaluation reproducible. Each spawn specified (x, y, ϕ) and

the box identifier local to the lane. Episodes were capped at 3,000 simulation steps to prevent rare but uninformative long tails whilst still allowing for enough time to complete the entire course.

The control loop maintained a fixed forward speed, with steering as the only learned action. This choice reduced the dimensionality of the policy and stabilised learning in the early stages, at the cost of limiting the attainable curvature in tight features such as the pedestrian crossing. The consequence of this design decision is analysed in the discussion; here the methodological priority was to isolate lane-keeping from speed control to keep the derivation crisp and the deployment footprint small.

2.3 Observation Space and Preprocessing

Input images were captured in grayscale to match the on-board camera configuration and to minimise bandwidth and memory on the EyeBot-8.

Figure 2.4: Image of an EyeBot in simulation on the Carolo Cup course, the image visible to the EyeBot is displayed on the LCD.

Frames were captured in QQVGA (120x160 (height × width)) with a field of view of 120° (see figure 2.4), normalised to [0,1], and laid out in NCHW order as [1,1,120,160] at inference time. No histogram equalisation or contrast normalisation was applied, to ensure that the deployed network saw the same photometric statistics as in simulation.

2.4 Action Space and Execution Limits

The policy controlled a single continuous steering degree-of-freedom. To reduce early over-steering and the associated spin-up failures seen in bring-up experiments, the executable steering bound was linearly ramped from 45° to 90° over the first 500 episodes, while the network was always trained against the action executed. Exploration noise was injected in pre-tanh space with a standard deviation annealed from $\sigma_{u,\text{init}} = 0.35$ to $\sigma_{u,\text{min}} = 0.10$ over 800 episodes. This schedule maintained sufficient exploration to discover the intersection trajectory while steadily reducing variance to consolidate lane keeping.

2.5 Reward Shaping and Penalties

Training uses discounted return with γ = 0.995, augmented by GAE with λ = 0.95. These values were chosen close to 1 to maintain the value of long-term rewards (completion of a successful lap). The per-step reward is the sum of three components: a bounded shaping term for geometry, small regularization on the control effort, and explicit penalties for undesirable events. The design prioritizes stable gradients and clear semantics over marginal gains in reward magnitude.

2.5.1 Geometric Shaping

Two error measures drive the shaping: lateral deviation from the lane centerline and heading error relative to the local tangent.

$$score = MAX_SHAPE \cdot \exp(-SHAPE_KD|d|) \cdot \frac{1}{1+\left(\frac{\Delta\psi}{HEAD_E0_DEG}\right)^2} \tag{2.1}$$

The lateral deviation section of the equation in equation (2.1) is an exponential decay equation, multiplying the weight `SHAPE_KD=0.0035` by the lateral distance from the robot position to the center of the lane. This causes the reward contribution to smoothly decrease the further the robot is from the center of the lane. Orientation of the robot contributes to the heading term of equation (2.1), which is structured as a Lorentzian centered at zero degrees. The width of the Lorentzian is set by `HEAD_E0_DEG=15°`, meaning that every 15° of error in the heading of the robot from the centerline halves the shaping factor for the reward (i.e. reward factor of 1 at 0° error, reward factor of 0.5 at 15° of error). A Lorentzian was chosen to promote close alignment to the centerline of the

lane – whilst allowing some room for misalignment as the lane tangent was not always perfectly aligned – avoiding the algorithm from being punished for slight error.

The values chosen for the constants were designed so that a positive signal close to +0.60 is granted for near-zero deviation and small heading error. This value was chosen to balance providing a reward enticing enough for the algorithm to learn smooth and central lines, whilst not so large as to oversaturate the neural network's rewards. The intent is to create a basin that rewards being centered and aligned but avoids plateaus that would starve the critic of gradient away from the center.

2.5.2 Control Effort

To discourage limit-cycle oscillations and spinning, a small quadratic cost on steering is subtracted every step (see equation (2.2) and equation (2.3)) where δ is the executed steering and δ_{max} is the current angle limit. This term is dimensionless and remains an order of magnitude smaller than the maximal positive shaping – ensuring that turning is not disincentivized, but needless steering is.

$$-ANGLE_L2_COEF \left(\frac{\delta}{\delta_{max}}\right)^2 \tag{2.2}$$

$$ANGLE_L2_COEF = 0.03 \tag{2.3}$$

2.5.3 Boundaries and Progress

Leaving the drivable corridor yields a per-step penalty of -1 for 15 steps to tolerate transient excursions on tight curves. If the agent enters a progress box out of order, a penalty of −2.0 is applied. A +5 reward is given for entering the next box in the sequence. Stagnation is detected by a short-horizon stuck condition that penalizes low displacement over 8 steps by -3.25 per step. A separate step penalty can be set but is currently at zero as adding any step penalty disrupted training and produced poor quality results.

This shaping is deliberately sparse in large negative events and modest in its positive basin. In preliminary runs, larger positive shaping caused the agent to exploit stationary configurations that scored well locally; increasing the control cost without the grace period caused the policy to cut corners and clip the outer lane on sharp bends. The configured values reflect empirical calibration against those failure modes.

2.6 Policy Network Architecture

The convolutional encoder comprised three strided layers followed by a two-layer multilayer perceptron, with shared features feeding both policy and value heads. For a grayscale input tensor of shape $1 \times 120 \times 160$ (NCHW), the stack followed a three-block $5 \times 5 \rightarrow 5 \times 5 \rightarrow 3 \times 3$ stack with strides of two and ReLU activations, feeding a 19,200-dimensional flatten into a 256–64 tanh MLP trunk, with linear heads for policy mean, value, and a learned log standard deviation. The tahn squashing was applied outside the network when converting u to a physical steering command using the current angle limit.

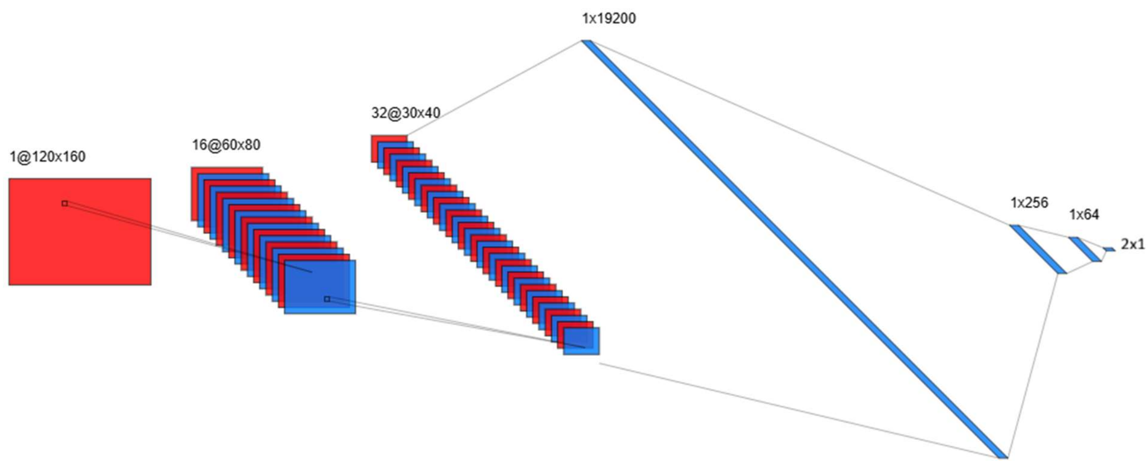

Figure 2.5: Visualisation of the end-to-end neural network.

This architecture was chosen to keep the compute footprint small for deployment while retaining sufficient receptive field to capture long-range lane cues. ReLU was used in the convolutional stack for gradient flow, and Tanh in the fully connected layers to bound hidden activations in support of the squashed output distribution.

2.7 Training Procedure and Hyperparameters

Training used PPO with clipped surrogate loss, value-function clipping, and an adaptive KL check. The discount factor was $\gamma = 0.995$, and GAE used $\lambda = 0.95$. The clipping parameter was $\epsilon = 0.2$. Each policy update consumed 512 environment steps per rollout, split into mini-batches of size 128, and ran for 5 epochs. Entropy regularisation used coefficient 0.01 to preserve early exploration

without preventing consolidation. The value function was clipped with VCLIP = 0.25, and an approximate KL target of 0.05 was monitored to detect overly aggressive updates. The optimiser was Adam at a learning rate of 3×10^{-4} with default betas. Fixed forward speed was set to 150 in EyeSim units, matching the deployment baseline on hardware.

The final configuration emerged from a sequence of failures and adjustments rather than a single sweep. Initial runs without a control-effort term and with a 90° steering bound from the outset produced attractive early returns but suffered from frequent spin-up and oscillation, particularly following brief lane departures. Introducing the angle-squared cost at 0.03 and ramping the steering bound from 45° to 90° over 500 episodes helped prevent persistent spinning without discouraging decisive turns. A second failure mode appeared as stagnation near the start boxes, where the agent learned to oscillate within a small region to harvest shaping reward. Adding the no-progress watchdog and wrong-box penalty broke this mode by making genuine forward motion the only way to sustain returns. The non-deterministic physics meant that apparent improvements over a handful of episodes were unreliable, so each change was judged over hundreds of seeds.

2.8 Software Tools and Export path

The training stack was implemented in Python using PyTorch for model definition and optimisation (see Appendix B). EyeSim provided the physics, rendering, and RoBIOS-compatible robot interface. Logging and diagnostics were written to episode-level CSV files and a rotating textual log, with explicit capture of progress indices, lane-departure counts, stuck events, and returns. The trained PyTorch model was exported to ONNX for deployment on the EyeBot-8 (see Appendix C), which runs a Raspberry Pi-class processor that cannot host the full Torch runtime. The exported graph preserved the inference contract as $[1,1,120,160]$ float32 in $[0,1]$, and outputs "mu_u", "std_u", and "value" as $[1,1]$. At run time on the robot, the ONNX outputs were post-processed by applying the same $\tanh$ squashing and current steering limit as in simulation (see Appendix D). This parity was important to ensure that any discrepancy between simulation and hardware reflected sensing and actuation differences rather than software divergence.

2.9 Evaluation Metrics and Experimental Design

The primary evaluation metric was normalised progress, defined as the fraction of progress boxes traversed in sequence before termination. A complete lap scored 1.0. Completion rate, reported as the

fraction of episodes reaching the terminal box, was retained as a secondary headline number for comparability with prior work but not used to draw fine-grained conclusions. Additional diagnostics included the count and duration of lane departures, the frequency of stuck terminations, and segment-specific progress around the intersection and pedestrian crossing. Because EyeSim is stochastic, all numbers were aggregated over the fixed spawn set and several random seeds. Plots reported medians and interquartile ranges; where appropriate, empirical cumulative distribution functions were used to compare training variants without assuming unimodal performance.

The training-validation split followed a standard reinforcement-learning pattern. Policies were trained for a fixed budget of environment steps, with periodic evaluation sweeps run under evaluation mode (no exploration noise) from the spawn catalogue. Hyperparameter decisions were made based on the evaluation distributions rather than the training rollouts to avoid overfitting to the idiosyncrasies of a particular random seed. Convergence was assessed qualitatively by the stabilisation of the progress distribution and quantitatively by the decay of policy-update KL divergences and the flattening of the value-loss curve.

2.10 Deployment Tests and Validation

Two validation phases were executed. First, the ONNX graph was exercised inside EyeSim to isolate export-related numerical differences from platform effects. In this configuration, the ONNX model reproduced the Torch model’s behaviour within the noise of the simulator and, interestingly, achieved slightly higher normalised progress through the intersection in the outer lane while performing worse at the pedestrian crossing (see figures 3.1 and 3.2). This suggested that the modest graph-level differences (for example, fused convolution kernels and activation ordering) were not the dominant factor behind the later hardware shortfall.

Second, the ONNX policy was run on the EyeBot-8. Performance degraded substantially: the robot frequently failed to align with the lane and exhibited sluggish or inconsistent steering. The potential reasons for this are discussed in section 3.2.

2.11 Health, Safety, and Ethical Considerations

Most experimentation occurred in simulation, eliminating human-subject or environmental risk. Hardware tests were performed briefly on an enclosed laboratory replica course at low speed to observe qualitative behaviour and to avoid damage to the robot or surroundings. The project's principal safety control was therefore scope limitation: the policy never left the supervised lab context and was not tested on a large scale. These choices were consistent with the project's objective, which was to study sim-to-real lane keeping rather than to achieve field-ready autonomy.

2.12 Limitations

Methodologically, the chief limitation is the single-camera, fixed-speed formulation, which simplifies the action space but forces a brittle trade-off between turning authority and stability. The grayscale assumption and minimal preprocessing reduce deployment complexity but increase sensitivity to camera photometry differences. The EyeSim physics model is non-deterministic but still more forgiving than reality, especially with respect to accumulation of small tracking errors, so policies that appear stable in simulation can sit close to instability when deployed. Finally, the ONNX runtime's execution model on the Raspberry Pi introduces nontrivial latency and jitter, which were not explicitly budgeted during training. These limitations are not flaws of the methodology so much as the design constraints under which it operated, and they motivate the future work set out in the conclusions.

3. Results & Discussion

3.1 Evaluation Metric and Headline Simulation Behaviour

Progress was measured using a sequence of axis-aligned boxes tiled along the lap. Critically, the intersection itself is modelled as an unsigned region: while the robot is inside the intersection, the progress index is not advanced or reset, and the agent is considered to remain in the last pre-intersection box until it fully crosses to the corresponding post-intersection box on the far side. Only then is progress updated. This convention prevents overlapping progress boxes in the intersection causing false ‘wrong order’ flags.

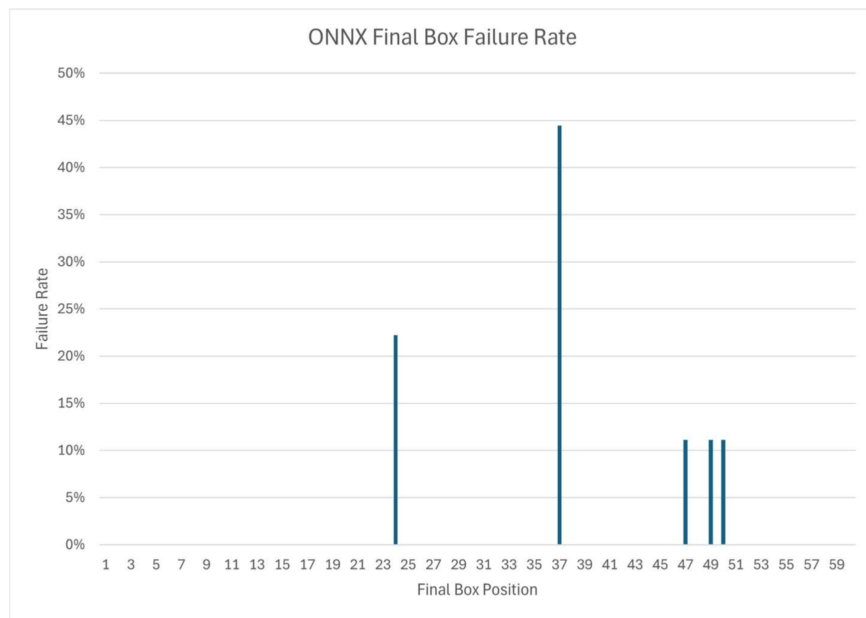

Figure 3.1: Graph showing the percentage of episodes in which the ONNX model failed per box.

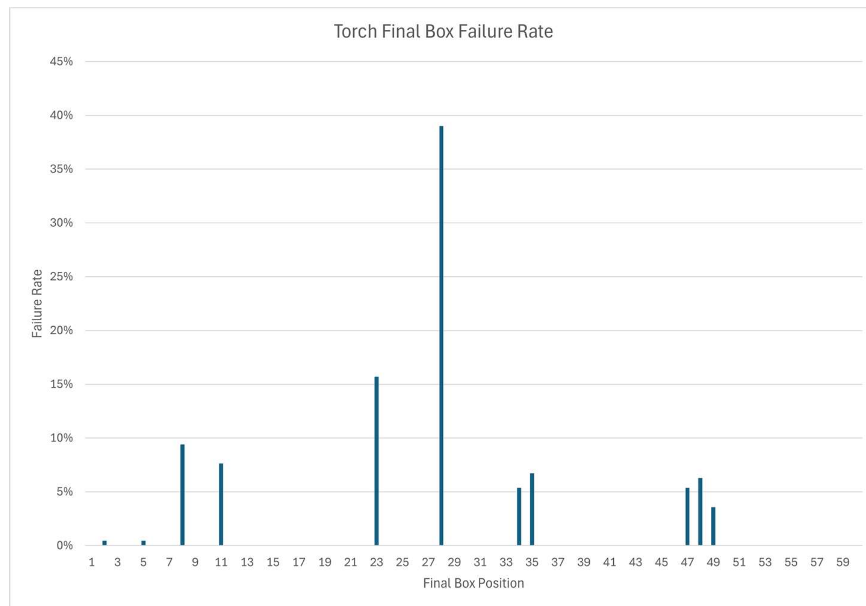

Figure 3.2: Graph showing the percentage of episodes in which the PyTorch model failed per box.

With this metric, the best simulated policies achieved consistent progress on straights and bends and showed a characteristic pattern at the junction. Under the Torch runtime, failures clustered immediately before the intersection. After conversion to ONNX and execution with the lighter runtime, simulated behaviour improved at the junction – fewer mid-junction oscillations and fewer aborted entries – while failures at the pedestrian crossing increased greatly (see figures 3.1 and 3.2). The net effect was a modest increase in median progress per episode and a shift of the failure histogram away from the first post-intersection box toward the pedestrian-crossing segment. This is consistent with small changes in closed-loop latency and activation numeric that smooth rapid steering changes across the unsigned region but can exacerbate over-correction under high-contrast visual flicker.

Learning curves exhibited the typical S-shape: an early phase dominated by lane-keeping on marked straights, a middle phase where junction traversals began to register as occasional advances past the unsigned band, and a late phase in which successful agents consistently cleared the first post-intersection box and stabilised progress through the remainder of the lap. Failure histograms, plotted against box index, consistently showed a sharp mode at pre-intersection box rather than within the intersection itself, reflecting the metric design. Qualitatively, policies that succeeded through both straight-through traversals did so by carrying a consistent heading across the unmarked centre and issuing earlier, lower-amplitude corrections immediately upon reacquiring lane markings on exit.

Policies that failed typically committed to a sharp turn to follow the edge of the road, resulting in loss of correct heading and ultimately leaving the track.

An improvement to the training protocol could be trialling a new method of resetting on failures. As per the current method, a failure on a tricky section on the track results in the end of the episode. With the random spawn locations, the agent may have to traverse the majority of the course to reach a difficult location again – greatly reducing the time spent learning once basic lane following is already achieved. A potential improvement to this would be keeping track of the pose that the robot possesses when entering each new box – and then resetting it to its position one or two boxes prior upon failure. This would force repeated attempts at difficult sections of the track – such as the intersection or the pedestrian crossing – thereby providing the agent with more data and an improved success rate for those sections.

3.2 Attempted Sim-to-Real Transfer on EyeBot-8 via ONNX

Deploying the ONNX model on the EyeBot-8 revealed a stark sim-to-real gap. Despite reasonable simulation behaviour, the physical robot struggled to follow lanes and often failed to establish a stable trajectory at all. There are several interacting causes, each traceable to platform constraints and modelling assumptions.

First, the EyeBot's compute budget is not designed for learned perception-control policies. PyTorch could not be installed due to memory limits, necessitating an ONNX export and a lighter runtime. Even with the lighter runtime, inference latency on the Raspberry Pi-class CPU was materially higher than in the simulation workstation, and the timing jitter was larger. In a closed-loop steering task at fixed speed, tens of milliseconds of extra latency, especially when variable, can push the controller from critically damped to underdamped behaviour, amplifying lateral deviations that would have been corrected earlier in simulation.

Second, the physical robot's low-level control and mechanics differ from the simulated model in ways that matter under a learned policy. Small differences in wheelbase, tread, friction, motor deadband, and controller discretization collectively change the mapping from action to lateral motion. In a hand-tuned classical controller, one retunes gains. In an end-to-end policy trained in a single simulator,

these unmodeled discrepancies appear to the network as "unseen dynamics" and produce out-of-distribution observations as the robot drifts, which the policy has not been trained to correct.

Third, the visual domain gap is substantial. Even with grayscale inputs, the real camera introduces sensor noise, lens distortion, and lighting variability that were not represented in the EyeSim observations. The pedestrian crossing failures in simulation foreshadowed this sensitivity. Without explicit domain randomization or augmentation during training, these differences reduce the effective SNR of lane features in the early convolutional layers and produce brittle downstream control. The literature is unambiguous on the value of domain randomization for bridging such gaps; policies trained on aggressively varied textures, lighting, and slight camera pose perturbations transfer more reliably from sim to real in vision-based tasks.

Finally, the single-action interface assumes that speed is exogenous and benign. On the real platform, speed interacts with actuation lag, motor quantization, and floor friction, which resulted in noticeable speed variation over the course of just a few seconds. This mismatch suggests rethinking the action space for real deployment so that the policy can trade speed for stability when needed.

In short, the poor real-world performance is consistent with a system whose closed-loop latency, actuation mapping, and visual statistics all fall outside the training distribution, and whose compute platform cannot execute the network with sufficiently low and stable latency.

3.3 Training Throughput, Stability, and the Implications of EyeSim's execution model

The central practical constraint shaping the entire study was EyeSim's run-time model. EyeSim executes a single simulation instance at a time and advances the world at (approximately) real-time. This constraint meant that even a modest training run required several hours to accumulate sufficient experience for policy updates, and any broad hyperparameter sweep or ablation became prohibitively time consuming. In well-tuned runs, the shortest end-to-end training cycles required roughly 4–6 hours to produce a policy that could complete the majority of the track; many runs took substantially longer, with several extending past 24 hours. Because EyeSim is also non-deterministic due to its physics and random error sources, replicate runs at the same seed sometimes diverged meaningfully in learning curves and final performance. The practical result is that progress was gated far more by

data collection time than by gradient computation time, creating a training regime that is throughput-limited rather than compute-limited.

This execution model has two second-order effects. First, it increases the variance of empirical conclusions because the number of independent runs that can be completed within a fixed project timeline is small. Second, it pushes the design toward reward shaping and architecture choices that maximize sample efficiency over time, even when those choices might not be globally optimal in a more data-rich setting. The project therefore prioritized a compact convolutional feature extractor and a single-action continuous control head, tuned rewards for dense progress signals, and adopted PPO because of its well-documented stability and sample efficiency relative to earlier policy gradient methods. PPO's clipped objective and advantage normalization are specifically designed to moderate destructive policy updates and maintain steady improvement in on-policy settings, which is a useful property when every additional hour of real-time simulation is expensive. This rationale aligns with the original arguments for PPO's robustness in environments with noisy returns and limited samples.

3.4 Comparison with Prior Work and with Alternative Platforms

The initial choice to implement PPO from scratch successfully produced a minimal inference footprint, but it delayed the transition to genuine experimentation. Much of the labour was consumed by infrastructure work – advantage estimation, numerically stable log-probabilities, data pipelines, and diagnostics – that mature libraries such as Stable-Baselines3 provide out of the box (PPO — Stable Baselines3 1.0 Documentation, 2021). In retrospect, training with Gymnasium and Stable-Baselines3 and exporting the trained network to ONNX for deployment would have preserved the runtime advantages on the robot while compressing the path to meaningful ablations in simulation. This observation does not change the measured outcomes, but it explains why hyperparameter tuning remained shallow and why some promising directions (e.g., curriculum schedules and stronger augmentation) were under-explored.

Against the objectives, the study succeeded in training PPO policies that advanced reliably in EyeSim and in diagnosing their failure modes, but the policies did not transfer effectively to the EyeBot hardware. In contemporary practice, sim-to-real transfer in vision-based lane following is routinely achieved when the training stack includes: (i) a simulator that supports high-throughput data collection and parallel instances, (ii) domain randomization or photorealistic rendering, and (iii) target

hardware with sufficient acceleration to sustain low-jitter inference. The Duckietown ecosystem is designed around these needs. The Gym-Duckietown simulator adheres to the OpenAI Gym interface and is explicitly described as supporting ML workflows for both imitation and reinforcement learning (Simulation in Duckietown — the Duckietown Developer Manual - Ente, 2018). These properties enable parallel data collection and, on suitable hardware, faster-than-real-time rollouts that shrink the real time needed for ablation and validation.

On the hardware side, Duckiebots in the DB21M family are configured to carry embedded accelerators, including NVIDIA Jetson Nano variants, providing GPU support for on-board inference (Duckietown Technology: A Complete Platform to Learn Autonomy, 2020). This compute headroom is central when deploying convolutional policies at 20–30 Hz with low jitter, as required for stable closed-loop lane keeping under realistic disturbances.

More broadly, the Duckietown project was created to make autonomy experimentation accessible and reproducible across education and research, with standardized tracks, signage, and a well-documented software stack that shortens the path from a research idea to a controlled experiment. Those characteristics map directly onto the bottlenecks encountered here.

The “best episode” performance achieved in EyeSim therefore represents an encouraging simulation milestone but should be interpreted as a stepping stone rather than an endpoint. The transfer failure is not anomalous given the platform’s constraints; rather, it is a strong signal about where the engineering leverage lies for future work.

3.5 Limitations, Arbitrary Choices, and Sensitivity

Beyond the platform-imposed limits, several project choices were necessarily arbitrary but were tracked and, where possible, stress-tested.

The observation pipeline used grayscale images at 1×120×160 (NCHW) with fixed cropping and normalization to [0,1]. This choice balanced computational simplicity against representational power, but it also removed colour cues that could have disambiguated crosswalk patterns and lane markings in difficult lighting. The convolutional architecture values (see section 2.6) were selected to keep the

parameter count tractable under the inference constraints while preserving receptive field growth across the feature maps. Alternative choices – e.g., adding batch normalization, using leaky ReLU, or introducing a small spatial attention module – were not explored due to throughput limits, and could alter the balance between intersection decisiveness and crosswalk robustness.

Reward weights were tuned empirically to reduce oscillations without stalling. The final weighting emphasized forward progress and bounded but decisive heading corrections. While these weights produced the best progress-per-hour curves within the EyeSim budget, they also reflected the incentive landscape that made the pedestrian crossing a stubborn local optimum: the visually high-contrast pattern occasionally yielded spurious centerline proxies, and the policy learned to overcorrect rather than reduce speed or "wait out" the transient because speed was not in the action space. Sensitivity tests with slightly different heading penalty slopes showed minimal change in aggregate progress, suggesting that the binding constraint was not the heading weight but the observation robustness at that feature scale.

Finally, model export from PyTorch to ONNX introduced a small but measurable change in simulated behaviour. Without precise profiling of per-stage latency and numerical parity across operators, these differences are difficult to attribute uniquely, but the consistent pattern – smoother intersection traversal, worse crosswalk robustness – points to closed-loop timing and numerical activation differences rather than a fundamentally different policy representation. That hypothesis could be tested by replaying identical observation logs through both runtimes and comparing action traces, which is recommended as future diagnostic work.

3.6 What the Results Mean for the Original Objectives

The original objective was to train a policy capable of robust lane keeping on a Carolo Cup–style track in simulation and to explore the feasibility of transferring that policy to identical robot hardware. The simulation objective was met in the sense that the progress metric improved reliably and the best ONNX-exported policy achieved strong performance through most of the course, with clear, localized failure at the pedestrian crossing. The transfer objective was not met: the policy did not yield acceptable lane keeping on the physical EyeBot. Interpreted in context, these outcomes are consistent with the state of the art: sim-to-real transfer for end-to-end vision policies typically demands higher-throughput simulation, deliberate domain randomization, and on-board acceleration, all of which are

integral to platforms like Duckietown. The present results therefore do not just document a shortfall; they delineate a migration path for future researchers that addresses the two dominant impediments identified here.

4. Conclusions & Future Work

This project demonstrated that a compact, grayscale vision policy trained with PPO can learn lane-keeping behaviours in EyeSim and intermittently traverse an unsigned four-way intersection, but it remained brittle at the pedestrian crossing and did not generalise to the EyeBot hardware. The results confirm that end-to-end learning is viable in principle on the Carolo-style track, yet they also show that simulator throughput, reward sensitivities, and inference-time constraints strongly shape the attainable performance. The most robust empirical signal was the median number of progress boxes traversed per rollout. That metric rose steadily during training, stabilised on straight segments, improved at the intersection under ONNX inference, and plateaued with persistent failures at the pedestrian crossing.

Relative to the objectives, the simulation goal – completing the Carolo Cup track whilst maintaining lane following – was partially achieved. Lane following was consistent when evaluating policies, but still experienced issues with the pedestrian crossing and intersection. Policies reached and crossed the intersection with increasing frequency late in training, and ONNX inference reduced over-steering at entry. However, consistent full-circuit completion across seeds was not realised, and pedestrian-crossing robustness did not match straight-segment competence. The deployment goal – credible on-robot lane keeping using the exported ONNX model – was not met. On hardware, the controller struggled to stabilise within a few boxes, reflecting compounded latency on a CPU-only pipeline, actuation mismatches, photometric shifts between the simulator and camera, and more punitive real-world dynamics. These outcomes align with expectations from the literature on sim-to-real transfer without aggressive domain and dynamics randomisation, and they are consonant with known sensitivities of on-policy vision controllers to runtime and sensing changes.

The project's main contributions are threefold. First, it provides a defensible, box-based evaluation protocol that exposes where and how failures cluster on the track, rather than reducing performance to a binary success measure. Second, it identifies two distinct closed-loop failure modes – texture-induced instability at the pedestrian crossing and 'panicked' responses to the lack of a lane border at the intersection – and shows that minor changes in the inference engine can move performance in opposite directions on these modes. Third, it makes explicit how a single-environment, real-time simulator constrains the science that can be done: long real time runs suppress seeds and ablations, blur cause and effect, and favour high-stakes sequential tuning over principled exploration. Each

contribution is practical and immediately useful to subsequent researchers working with EyeSim and the EyeBot platform.

Future work should prioritise three themes: platform, perception and control, and experimental depth. At the platform level, moving to a machine-learning-first stack such as Gym-Duckietown for simulation and Duckiebot hardware for deployment would address the structural issues identified in sections 3.3 and 3.4 by removing the throughput bottleneck and enabling GPU-accelerated, real-time inference. Parallel, faster-than-real-time simulators allow proper seed counts, systematic ablations, and curriculum studies that are infeasible under EyeSim's one-at-a-time regime. The paired hardware, with embedded GPU acceleration, matches the computational profile assumed during training and reduces the latency variance that undermined EyeBot performance. The feasibility of this migration is supported by prior demonstrations on Duckietown of end-to-end reinforcement learning performing lane following and obstacle avoidance (Chen, 2024).

In perception and control, several targeted changes follow directly from the observed failure modes. Photometric alignment between training images and robot captures should be enforced through calibrated camera models, histogram matching, and strong photometric augmentation. Observation-space randomisation should be complemented by dynamics randomisation of steering gains, latency injections, and slip to immunise the policy against the very changes that degraded it on hardware. The encoder can be modestly strengthened without sacrificing runtime by incorporating temporal context – e.g., a light-weight recurrent layer or frame-stacked 3D convolutions – to dampen stripe-induced oscillations at the pedestrian crossing while preserving responsiveness at the intersection. On the action side, latency-aware training that places the tanh-limited command through an explicit delay and first-order motor model during rollouts will better match the EyeBot's actuation, while a shallow, classical stabiliser (e.g., a centreline-seeking PID layer) under a learned perception policy remains a credible hybrid alternative if strict end-to-end control continues to show sensitivity on hardware.

In experimental depth, the work should move from one-off improvements to principled studies. Reward-shaping weights should be ablated systematically against the boxes-traversed metric, with pedestrian-crossing success and intersection success reported separately to avoid Simpson's-paradox masking. The degree limit on the tanh-squashed steering, the density of progress boxes, and the episode time budget should be scanned under fixed seeds to localise the regimes where learning stalls.

Where feasible, imitation learning from a robust classical lane follower can warm-start the policy before switching to PPO fine-tuning, shortening the time to competent behaviour. All results should be reported with seed counts sufficient to estimate uncertainty bands, and raw logs and model artefacts should be archived to enable exact reproduction.

In closing, the project delivers a clear picture of what does and does not work today. A compact PPO policy can learn useful structure from grayscale images in EyeSim, and ONNX export changes behaviour in ways that matter for intersection handling. Yet the same policy fails to carry over to the EyeBot, primarily because the computational and physical assumptions in training do not hold on the robot. The most leveraged next steps are to adopt a simulator and hardware stack designed around machine learning throughput and latency, to harden the vision and control loops against the specific shifts observed here, and to replace anecdotal tuning with proper ablation and uncertainty reporting. Doing so will convert the partial simulation success into repeatable, on-robot lane keeping and will turn the present diagnostic insights into a durable, extensible baseline for subsequent cohorts.

References

Bojarski, M., Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). *End to End Learning for Self-Driving Cars*. https://arxiv.org/pdf/1604.07316

Paull, L., Tani, J., Ahn, H., Alonso-Mora, J., Carlone, L., Cap, M., Chen, Y. F., Choi, C., Dusek, J., Fang, Y., Hoehener, D., Liu, S.-Y., Novitzky, M., Okuyama, I. F., Pazis, J., Rosman, G., Varricchio, V., Wang, H.-C., Yershov, D., & Zhao, H. (2017, May 1). *Duckietown: An open, inexpensive and flexible platform for autonomy education and research*. IEEE Xplore. https://doi.org/10.1109/ICRA.2017.7989179

Dickmanns, E. D., & Mysliwetz, B. D. (1992). Recursive 3-D road and relative ego-state recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *14*(2), 199–213. https://doi.org/10.1109/34.121789

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. https://arxiv.org/pdf/1801.01290

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep Reinforcement Learning That Matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1). https://doi.org/10.1609/aaai.v32i1.11694

Hüttenrauch, M., Šošić, A., & Neumann, G. (2018, July 17). *Deep Reinforcement Learning for Swarm Systems*. ArXiv.org. https://arxiv.org/abs/1807.06613

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., & Shah, A. (2019). *Learning to Drive in a Day*. https://arxiv.org/pdf/1807.00412

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274. https://doi.org/10.1177/0278364913495721

Kulhánek, J., Derner, E., Bruin, T.D., & Babuška, R. (2019). Vision-based Navigation Using Deep Reinforcement Learning. *2019 European Conference on Mobile Robots (ECMR)*, 1-8.

Ye, F., Cheng, X., Wang, P., Chan, C.-Y., & Zhang, J. (2020). *Automated Lane Change Strategy using Proximal Policy Optimization-based Deep Reinforcement Learning*. ArXiv.org. https://arxiv.org/abs/2002.02667

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). *CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING*. https://arxiv.org/pdf/1509.02971

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., Kumaran, D., & Hadsell, R. (2017). Learning to Navigate in Complex Environments. https://arxiv.org/abs/1611.03673

Mnih, V., Badia, Adrià Puigdomènech, Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). *Asynchronous Methods for Deep Reinforcement Learning*. ArXiv.org. https://arxiv.org/abs/1602.01783

Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018, May 1). *Sim-to-Real Transfer of Robotic Control with Dynamics Randomization*. IEEE Xplore. https://doi.org/10.1109/ICRA.2018.8460528

Pomerleau, D. A. (1989). ALVINN: An autonomous land vehicle in a neural network. *Proceedings of the 2nd International Conference on Neural Information Processing Systems (pp. 305–313)*

Randløv, J., & Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. *Proceedings of the Fifteenth International Conference on Machine Learning (pp. 463-471)*

Ross, S., Gordon, G., Bagnell, J., & Learning, M. (2011). *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. https://arxiv.org/pdf/1011.0686

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. https://arxiv.org/pdf/1707.06347

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., & Markey, C. (2006). Stanley: The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics*, *23*(9), 661–692. https://robots.stanford.edu/papers/thrun.stanley05.pdf

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*. https://arxiv.org/pdf/1703.06907

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, *105*(12), 2295–2329. https://doi.org/10.1109/jproc.2017.2761740

PPO — Stable Baselines3 1.0 documentation. (2021). Readthedocs.io. https://stable-baselines3.readthedocs.io/en/v1.0/modules/ppo.html

Duckietown Technology: a complete platform to learn autonomy. (2020, October 17). Duckietown - Learning Robotics and AI like the Professionals. https://duckietown.com/platform/

Simulation in Duckietown — The Duckietown Developer Manual - ente. (2018). Duckietown.com. https://docs.duckietown.com/ente/devmanual-software/intermediate/simulation/index.html

Chen, G. (2024). *Autonomous robot car with Deep Reinforcement Learning* [Review of *Autonomous robot car with Deep Reinforcement Learning*]. Https://Tud.qucosa.de/; Technische Universität Dresden. https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-945829

Appendix A. Extended Literature Review

A.1 Scope and Aim

This review surveys methods and evidence relevant to vision-based, end-to-end lane keeping for miniature mobile robots trained with reinforcement learning (RL). It proceeds from classical perception-and-control pipelines to deep RL (DRL) with image inputs, and then to reward shaping, evaluation practice, simulation platforms, sim-to-real transfer, embedded deployment, and multi-robot considerations. The objective is not to reproduce encyclopaedic breadth, but to justify specific design choices made in this project and to mark the limits within which findings should be interpreted.

A.2 Search Strategy and Data Collection

Sources were gathered through iterative searches on IEEE Xplore, ACM Digital Library, Scopus, Web of Science, and Google Scholar, supplemented by arXiv for influential preprints later published or frequently cited. Queries combined task, method, and platform terms, for example: "autonomous lane keeping vision end-to-end," "PPO pixels continuous control," "reward shaping navigation potential-based," "domain randomization sim-to-real," "ONNX embedded inference robot," "Duckietown Gym-Duckietown PPO," and "EyeBot EyeSim RoBIOS." Backward and forward snowballing extended the corpus from anchor papers. DRL and sim-to-real works were drawn primarily from 2015 onward, while foundational items in classical vision and reward shaping were included regardless of date.

A.3 Principal Sources and Inclusion Rationale

The review inspects peer-reviewed studies in robotics and machine learning (ICRA, IROS, RSS, CoRL, NeurIPS, ICML) and journals that set methodological baselines for embodied learning. Platform documentation (e.g., Duckietown) and textbooks or monographs were consulted but used primarily to establish system interfaces and constraints rather than to ground empirical claims. Practitioner blogs and code repositories were excluded unless they filled an otherwise undocumented implementation gap. Preference was given to studies that specify observation and action spaces, report task-level metrics rather than reward alone, and provide ablations or sensitivity analyses.

A.4 Screening and Evaluation

Screening proceeded in two passes. First-pass filters removed items that did not address end-to-end navigation, continuous control, reward shaping for long-horizon tasks, or sim-to-real transfer. A second pass assessed methodological clarity, reproducibility signals (e.g., code, parameter tables), and relevance to small robots with constrained compute. Studies whose conclusions hinged on compute scales unavailable to the present platform were down-weighted. Classical lane-keeping

papers were retained as historical baselines because they illuminate why end-to-end approaches have appeal at embedded scales.

A.5 Thematic synthesis

A.5.1 Classical Lane Keeping

For three decades, lane keeping has been framed as perception followed by control. Canonical pipelines detect edges, fit line models with Hough transforms, and use inverse perspective mapping to estimate geometry, which is then tracked by controllers such as PID or Stanley (Dickmanns & Mysliwetz, 1992; Thrun et al., 2006). These systems can be fast and interpretable, yet they are brittle under shadows, worn paint, glare, and exposure drift. The problem intensifies on small robots with inexpensive lenses and rolling-shutter cameras, where changes in illumination and lens distortion are proportionally larger relative to the scene. Pomerleau's early end-to-end effort, ALVINN, already hinted that learned representations could absorb some of this variability, though compute and data were limiting at the time (Pomerleau, 1989).

A.5.2 Behavioural Cloning to Deep RL

End-to-end learning collapses feature engineering and control into a single function from images to actions. Supervised behavioural cloning showed that convolutional networks can map images to steering, but covariate shift remains a central failure mode: once the policy drifts off the expert manifold, mistakes compound (Bojarski et al., 2016; Ross et al., 2011). DRL replaces imitation objectives with reward maximization, at the cost of more challenging exploration and credit assignment. Asynchronous actor–critic methods (A3C/A2C), deterministic off-policy methods (DDPG), entropy-regularized algorithms (SAC), and on-policy proximal methods (PPO) provide different Pareto points in stability, sample efficiency, and implementation burden (Mnih et al., 2016; Lillicrap et al., 2016; Haarnoja et al., 2018; Schulman et al., 2017; Kober et al., 2013). For partially observable visual navigation, auxiliary tasks (e.g., depth prediction) and memory modules (e.g., LSTMs) have improved data efficiency and robustness in indoor benchmarks (Mirowski et al., 2017; Kulhánek et al., 2019). Whether such additions pay for themselves on embedded targets depends as much on timing and numerical stability as on accuracy.

A.5.3 PPO for Image-to-Control

PPO's minibatch, multiple-epoch updates over short rollouts reduce variance and make it straightforward to integrate with convolutional encoders. In driving tasks, PPO has repeatedly shown stable learning from shaped rewards where off-policy methods can be more sensitive to hyperparameters avoids the brittle replay dynamics that sometimes undermine off-policy methods with non-stationary visual observations (Schulman et al., 2017; Ye et al., 2020; Küttel et al., 2020).

This combination has made PPO a reliable default in vision-based control where developers value predictable behaviour over peak sample efficiency (Schulman et al., 2017). In short-run experiments constrained by simulator throughput, this stability is often decisive.

A.5.4 Reward Shaping and Evaluation for Long-Horizon Navigation

Lane completion is a long-horizon objective with sparse natural reward. Potential-based shaping augments the signal with dense, state-dependent terms while preserving optimal policies in theory (Ng et al., 1999). In practice, effective shaping for lane keeping couples lateral error and heading alignment within a bounded basin and adds event penalties for lane departure, stagnation, and regress (Randløv & Alstrøm, 1998; Kendall et al., 2019). A consistent warning in the literature is not to conflate shaped reward with task performance. Evaluation should use task-level metrics – lap completion, distance to goal, or progress over waypoints – because the relative weights in a shaped reward are design choices, not ground truth (Henderson et al., 2018). Progress-by-boxes (waypoints) is a practical compromise in small-scale simulators because it yields dense feedback without precise global pose, and it localizes failure to interpretable segments of track.

A.5.5 Visual Design on Small Robots

On small platforms, low-resolution grayscale input paired with compact convolutional encoders is often sufficient when markings are high-contrast and the reward emphasizes geometry (Bojarski et al., 2016; Paull et al., 2017). The binding constraint is not representational capacity but the determinism of preprocessing and the constancy of timing in the control loop. Small architectural additions – coordinate channels, shallow attention – can help at intersections and crossings, though each addition must justify its compute and latency cost. In many cases, gains from improved normalization, exposure control, and lens calibration outweigh those from deeper networks.

A.5.6 Simulation Platforms

Simulation is essential for RL, but its characteristics determine what can be learned. General-purpose platforms such as Gazebo and Webots offer rich physics and sensor models. Duckietown's Gym-Duckietown adds gym-compatible interfaces, fast simulation steps, and straightforward parallelization, and it links to physical Duckiebots with NVIDIA Jetson modules for accelerated inference (Paull et al., 2017; Chevalier-Boisvert et al., 2018). EyeSim sits at the other end of the trade-off: it offers API coherence with EyeBot hardware via RoBIOS, simplifying code transfer, but it executes a single simulation at real-time speed. For RL, the difference matters. Where the research question is learning dynamics, parallel rollouts and faster-than-real-time stepping dominate. Where the question is code portability and teaching, API fidelity has primacy.

A.5.7 Sim-to-Real

Transferring policies from simulation to hardware remains a core challenge. Visual domain shift (lighting, exposure, lens distortion), dynamics mismatch (friction, backlash, latency), and runtime differences between training and deployment stacks can all degrade performance. Strategies in the literature span extensive photometric augmentation, domain randomisation, dynamics randomisation or system-identification loops, and architectural or runtime choices that limit latency and numerical drift between frameworks (Tobin et al., 2017; Peng et al., 2018). For constrained hardware, model export to ONNX or similar runtimes is typical, but even minor preprocessing differences or quantisation can alter action distributions – an effect reported in several sim-to-real studies and directly observed in this project's comparison between PyTorch and ONNX inference (Huang et al., 2021).

A.5.8 Embedded Inference

Without GPUs, viable policies are compact and predictable. Exporting trained networks to ONNX removes the training framework at deployment and helps reduce memory overheads, but it raises a new requirement: the deployment stack must replicate training-time preprocessing and numeric scaling exactly. Compression techniques – pruning and quantization – can be effective, but they must be validated in closed loop because even small phase delays or amplitude changes in steering can produce oscillations. Empirical accounts consistently place frame-rate regularity and low jitter ahead of marginal model accuracy for lateral control (Sze et al., 2017).

A.5.9 Multi-Robot RL

Although this project focuses on a single vehicle, multi-robot RL is instructive in highlighting representation and scalability pressures. Permutation-invariant encodings and centralized-training/decentralized-execution schemes handle variable team sizes, but real-world deployments remain sensitive to bandwidth limits and embedded compute budgets (Hüttenrauch et al., 2018). The lesson for single-robot lane keeping is conservative: keep stacks lean, eliminate unnecessary nondeterminism, and privilege timing stability over architectural ambition.

A.6 Implications for Method and Evaluation

Four conclusions follow. First, compact grayscale encoders suffice for miniature lane keeping when shaping links lateral and heading errors tightly; the limiting factor is not representational power but runtime determinism (Bojarski et al., 2016; Paull et al., 2017). Second, PPO is a pragmatic algorithm under limited simulator throughput because it tolerates short rollouts and modest batch sizes while retaining stable updates in pixel-to-control settings (Schulman et al., 2017; Küttel et al., 2020). Third, evaluation should foreground task-level metrics – completion rate and boxes traversed – and use shaped reward strictly as a training device (Henderson et al., 2018). Fourth, successful sim-to-real

transfer on low-cost platforms depends more on photometric robustness and tight control of timing and preprocessing than on ever larger networks or exotic losses (Tobin et al., 2017; Peng et al., 2018). These points underpin the choices taken in the project and explain the observed divergence between competent simulation behaviour and limited on-robot performance.

Appendix B. torchTraining.py

```
#!/usr/bin/env python3
# ============ UNBUFFERED DEBUG / LOGGING ============
import os, sys, logging, faulthandler, math, random, csv, argparse
from datetime import datetime

def RAW(msg: str):
    try: os.write(1, (msg + "\n").encode("utf-8"))
    except Exception: pass

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(message)s",
    handlers=[logging.FileHandler("run.log", mode="a", encoding="utf-8")],
    force=True,
)
LOG = logging.getLogger("ppo")

faulthandler.enable()
# faulthandler.dump_traceback_later(10, repeat=True)


RAW("[boot] script start")
RAW(f"[boot] python={sys.executable}  {sys.version}")
LOG.info("[boot] logger online")

# ============ IMPORTS ============
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from eye import *   # RoBIOS/EyeSim API
RAW("[boot] after eye import")

# ============ CAMERA / INPUT ============
USE_GRAY_CAMERA = True
USE_HUE_SINCOS = True   # ignored if grayscale

# ============ HYPERPARAMETERS ============
CAMWIDTH  = 160
CAMHEIGHT = 120

GAMMA       = 0.995
GAE_LAMBDA  = 0.95
CLIP_EPS    = 0.2
PPO_EPOCHS  = 5
ENTROPY_COEF= 0.01
LR          = 3e-4
ROLLOUT_STEPS = 512
MINIBATCH_SIZE = 128
VCLIP       = 0.25
KL_TARGET   = 0.05

# Squashed policy settings
BASE_ANG_LIMIT_DEG = 90.0       # theoretical action bound
EARLY_ANG_LIMIT    = 45.0       # execution limit for early training
ANG_LIMIT_RAMP_END = 500        # episodes to reach full 90°
```

```
EXEC_CMD_SCALE     = 1.0

# Exploration (pre-tanh) std anneal
U_STD_INIT    = 0.35            # ~30° after tanh*limit
U_STD_MIN     = 0.10
U_STD_ANNEAL_EPISODES = 800

# Files
MODEL_PATH = "ppo_lane_20250924_043048.pt" if USE_GRAY_CAMERA else
"ppo_lane_squash.pt"
#MODEL_PATH = "ppo_lane_squash_gray.pt" if USE_GRAY_CAMERA else
"ppo_lane_squash.pt"
BACKUP_DIR  = "backups"
LOG_CSV     = "episode_stats.csv"
GRAPH_CSV   = "graph_data.csv"
BOXES_CSV = os.path.join(os.path.dirname(__file__), 'boxes.csv')
BOXES_INNER_CSV = os.path.join(os.path.dirname(__file__), 'boxes_inner.csv')

# ============ LAYOUT (camera left, map right) ============
MAP_X, MAP_Y = CAMWIDTH + 2, 0
MAP_W, MAP_H = CAMWIDTH, CAMHEIGHT
MAP_MARGIN   = 4

# ============ TASK / DYNAMICS (bring-up friendly) ============
EVAL_MAX_STEPS       = 1500
OUTSIDE_STEP_PENALTY = -1.0
OUTSIDE_GRACE_STEPS  = 15
WRONG_BOX_PENALTY    = -2.0
DEBUG_RENDER_EVERY   = 5
LINE_SHAPING_REWARD  = True
STEP_PENALTY         = 0

# Shaping (stronger for bring-up)
MAX_SHAPE     = 0.60
SHAPE_KD      = 0.0035
HEAD_E0_DEG   = 15.0

# Stuck/spin detection (tighter)
STUCK_DIST_EPS   = 3.0
STUCK_MAX_STEPS  = 8
STUCK_PENALTY    = -3.25

# Steering-only phase
FIXED_LIN_SPEED = 150

# Anti-spin: tiny control cost on steering
ANGLE_L2_COEF = 0.03  # per-step: -coef * (ang/limit)^2

# No-progress nudge
NO_PROGRESS_STEPS   = 20
NO_PROGRESS_PENALTY = -1.0

# Spawns (x, y, phi, spawn_box_id, is_inner flag appended later)
OUTER_SPAWNS = [
    (2645, 4400, -359, 0),
    (300, 1570, -89, 47),
    (300, 3180, -89, 52),
    (3635, 540, -179, 35),
    (4165,  2245, -317, 28),
    (1690,  1920, -105, 18),
```

```
]
INNER_SPAWNS = [
    (2646, 4086, -179, 0),
    (600, 3158, -270, 9),
    (600, 1580, -270, 14),
    (3645, 882,    0, 26),
    (4153, 1817, -131, 32),
    (1956, 2611, -270, 42),
]
ALL_SPAWNS = [(*s, False) for s in OUTER_SPAWNS] + [(*s, True) for s in
INNER_SPAWNS]
os.makedirs(BACKUP_DIR, exist_ok=True)

# ============ GEOMETRY ============
def point_in_poly(x, y, poly):
    inside = False
    n = len(poly)
    for i in range(n):
        x0, y0 = poly[i]
        x1, y1 = poly[(i+1) % n]
        if ((y0 > y) != (y1 > y)) and (x < (x1 - x0) * (y - y0) / (y1 - y0 +
1e-9) + x0):
            inside = not inside
    return inside

class Box:
    def __init__(self, box_id, corners, is_intersection):
        self.box_id = box_id
        self.corners = corners
        self.is_intersection = bool(is_intersection)
    def contains(self, px, py): return point_in_poly(px, py, self.corners)

def load_boxes(csv_path):
    normal, inter = [], None
    with open(csv_path, "r", newline="") as f:
        reader = csv.reader(f); header = next(reader, None)
        for row in reader:
            box_id = int(row[0])
            x1, y1 = float(row[1]), float(row[2])
            x2, y2 = float(row[3]), float(row[4])
            x3, y3 = float(row[5]), float(row[6])
            x4, y4 = float(row[7]), float(row[8])
            is_int = int(row[9])
            box = Box(box_id, [(x1,y1),(x2,y2),(x3,y3),(x4,y4)], is_int)
            if is_int == 1: inter = box
            else: normal.append(box)
    normal.sort(key=lambda b: b.box_id)
    return normal, inter

# ============ SQUASHED GAUSSIAN UTILS ============
def atanh_safe(x, eps=1e-6):
    x = torch.clamp(x, -1 + eps, 1 - eps)
    return 0.5 * (torch.log1p(x) - torch.log1p(-x))

def squash_action(u, limit):  # limit can be float or tensor [B,1]
    return limit * torch.tanh(u)

def log_prob_squashed(dist_u, a, limit):
    """
```

```
    Per-sample squashed Gaussian log-prob with per-sample limit and optional
actuator scaling absorbed.
    a:     [B,1] executed continuous action (degrees), before int()
rounding.
    limit: [B,1] or scalar tensor — the exact limit used when executing
action
           (include EXEC_CMD_SCALE by folding it into 'limit' if scaled).
    """
    if not torch.is_tensor(limit):
        limit = torch.tensor(float(limit), dtype=a.dtype,
device=a.device).view(1,1).expand_as(a)
    eps = 1e-6
    a_scaled = torch.clamp(a / limit, -1 + eps, 1 - eps)   # back to tanh-
space
    u = atanh_safe(a_scaled)
    base_log = dist_u.log_prob(u).sum(dim=1)
    corr = 2.0 * (u + F.softplus(-2.0*u) - math.log(2.0))
    corr = corr.sum(dim=1)
    return base_log - torch.log(limit.squeeze(1)) - corr

def entropy_base(dist_u):  # proxy entropy of base Normal
    return dist_u.entropy().sum(dim=1)

# =========== NETWORK ===========
class ActorCritic(nn.Module):
    def __init__(self, input_shape):
        super().__init__()
        c, h, w = input_shape
        self.conv = nn.Sequential(
            nn.Conv2d(c, 16, kernel_size=5, stride=2, padding=2), nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=5, stride=2, padding=2),
nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
nn.ReLU(),
            nn.Flatten()
        )
        with torch.no_grad():
            conv_out = self.conv(torch.zeros(1, c, h, w)).shape[1]
        self.fc = nn.Sequential(
            nn.Linear(conv_out, 256), nn.Tanh(),
            nn.Linear(256, 64),       nn.Tanh()
        )
        self.mu_u       = nn.Linear(64, 1)  # pre-squash mean
        self.log_std_u  = nn.Parameter(torch.log(torch.tensor([U_STD_INIT],
dtype=torch.float32)))
        self.value_head = nn.Linear(64, 1)

    def forward(self, x):
        z = self.fc(self.conv(x))
        mu_u  = self.mu_u(z)
        std_u = self.log_std_u.exp().expand_as(mu_u)
        val   = self.value_head(z)
        return mu_u, std_u, val

# =========== PPO AGENT ===========
class PPOAgent:
    def __init__(self, input_shape):
        self.net = ActorCritic(input_shape)
        self.optimizer = optim.Adam(self.net.parameters(), lr=LR)
```

```
        self.buffer = []   # dict(state, a_exec, limit_used, old_logp,
value, reward, done, next_value)
        self.total_steps = 0
        self.episode_count = 0
        if os.path.exists(MODEL_PATH):
            self._load()

    def _save(self):
        data = {
            'model_state_dict': self.net.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
            'episode_count': self.episode_count
        }
        torch.save(data, MODEL_PATH)
        RAW(f"[PPOAgent] checkpoint saved {MODEL_PATH} (ep
{self.episode_count}).")
        if self.episode_count % 10 == 0:
            ts = datetime.now().strftime("%Y%m%d_%H%M%S")
            backup = os.path.join(BACKUP_DIR, f"ppo_lane_{ts}.pt")
            torch.save(data, backup); RAW(f"[PPOAgent] backup saved
{backup}.")

    def save_success_backup(self, ep_idx: int):
        """Save a special backup whenever an episode completes a lap
successfully."""
        try:
            ts = datetime.now().strftime("%Y%m%d_%H%M%S")
            path = os.path.join(BACKUP_DIR, f"EP{ep_idx}_SUCCESS_{ts}.pt")
            data = {
                'model_state_dict': self.net.state_dict(),
                'optimizer_state_dict': self.optimizer.state_dict(),
                'episode_count': self.episode_count
            }
            torch.save(data, path)
            RAW(f"[PPOAgent] SUCCESS backup saved {path}")
        except Exception as e:
            RAW(f"[PPOAgent] SUCCESS backup failed: {e}")

    def _load(self):
        try:
            ck = torch.load(MODEL_PATH, map_location="cpu")
        except Exception as e:
            RAW(f"[PPOAgent] no checkpoint ({e})"); return
        model_state = self.net.state_dict()
        matched = {k:v for k,v in ck.get('model_state_dict',{}).items()
                   if k in model_state and v.shape == model_state[k].shape}
        model_state.update(matched)
        self.net.load_state_dict(model_state, strict=False)
        RAW(f"[PPOAgent] Loaded params {len(matched)}/{len(model_state)}")
        if 'optimizer_state_dict' in ck:
            try:
                self.optimizer.load_state_dict(ck['optimizer_state_dict'])
                RAW("[PPOAgent] Optimizer state loaded.")
            except Exception as e:
                RAW(f"[PPOAgent] skip opt state ({e})")
        self.episode_count = int(ck.get('episode_count', 0))
        RAW(f"[PPOAgent] ep_count={self.episode_count}")

    def current_exec_limit(self):
        # ramp execution limit from EARLY_ANG_LIMIT to BASE_ANG_LIMIT_DEG
```

```
        if self.episode_count >= ANG_LIMIT_RAMP_END:
            return BASE_ANG_LIMIT_DEG
        frac = max(0.0, min(1.0, self.episode_count /
float(ANG_LIMIT_RAMP_END)))
        return EARLY_ANG_LIMIT + frac * (BASE_ANG_LIMIT_DEG -
EARLY_ANG_LIMIT)

    def value_of(self, state_np):
        with torch.no_grad():
            s = torch.FloatTensor(state_np).unsqueeze(0)
            _, _, v = self.net(s)
            return v.squeeze().item()

    def get_action(self, state_np, deterministic=False):
        state_t = torch.FloatTensor(state_np).unsqueeze(0)
        mu_u, std_u, val = self.net(state_t)
        dist_u = torch.distributions.Normal(mu_u, std_u)

        if deterministic: u = mu_u
        else:             u = dist_u.rsample()

        # Policy's base limit is BASE_ANG_LIMIT_DEG, EXECUTE with a
(possibly smaller) per-episode limit,
        # and (optionally) an actuator scale. Must train on the EXACT
executed action to stay on-policy.
        limit_exec = self.current_exec_limit()
        limit_tensor = torch.tensor([[limit_exec]], dtype=u.dtype)

        a_cont = squash_action(u, limit_tensor)             # continuous deg
within ±limit_exec
        a_cont_exec = a_cont * EXEC_CMD_SCALE               # if scaling,
fold it into executed action
        ang_exec = int(torch.clamp(a_cont_exec[0,0], -
limit_exec*EXEC_CMD_SCALE, limit_exec*EXEC_CMD_SCALE).item())

        # log-prob of the executed continuous action (with the SAME limit
incl. scale)
        limit_for_log = limit_tensor * EXEC_CMD_SCALE
        old_logp = log_prob_squashed(dist_u, a_cont_exec,
limit_for_log).squeeze(0).detach()

        return ang_exec, old_logp, val.squeeze().detach(),
a_cont_exec.detach(), float(limit_exec*EXEC_CMD_SCALE)

    def add_transition(self, tr):
        self.buffer.append(tr)
        self.total_steps += 1

    def maybe_update(self, force=False):
        if (len(self.buffer) >= ROLLOUT_STEPS) or (force and
len(self.buffer) > 0):
            self._update_from_buffer()
            self.buffer.clear()
            # anneal exploration std in pre-tanh space
            with torch.no_grad():
                hi = torch.tensor(U_STD_INIT)
                lo = torch.tensor(U_STD_MIN)
                frac = min(1.0, self.episode_count /
float(U_STD_ANNEAL_EPISODES))
                target = hi + frac * (lo - hi)
```

```
                self.net.log_std_u.data.copy_(torch.log(torch.clamp(target,
min=U_STD_MIN)))

    def _update_from_buffer(self):
        RAW(f"[ppo] update enter buf={len(self.buffer)}
ep={self.episode_count}")
        LOG.info("[ppo] update enter buf=%d ep=%d", len(self.buffer),
self.episode_count)

        states  = torch.FloatTensor(np.stack([t['state'] for t in
self.buffer]))
        actions = torch.stack([t['a_exec'] for t in self.buffer]).view(-1,1)
# executed continuous deg
        limits  = torch.FloatTensor([t['limit_used'] for t in
self.buffer]).view(-1,1)
        old_logp= torch.stack([t['old_logp'] for t in self.buffer]).view(-1)
        rewards = torch.FloatTensor([t['reward'] for t in self.buffer])
        dones   = torch.FloatTensor([float(t['done']) for t in self.buffer])
        values  = torch.FloatTensor([t['value'] for t in self.buffer])
        next_values = torch.FloatTensor([t['next_value'] for t in
self.buffer])

        # GAE
        returns, advs = self._compute_gae(rewards, values, next_values,
dones)
        advs = (advs - advs.mean()) / (advs.std() + 1e-8)

        N = len(self.buffer)
        idxs_all = torch.arange(N)

        with torch.no_grad():
            total_before = sum(p.abs().sum().item() for p in
self.net.parameters())

        RAW(f"[PPO][pre] N={N} adv_mean={advs.mean().item():.4f}
adv_std={advs.std().item():.4f} "
            f"ret_mean={returns.mean().item():.4f}
ret_std={returns.std().item():.4f}")

        last_mean_kl = 0.0
        clip_fracs, p_losses, v_losses, ents = [], [], [], []

        for _ in range(PPO_EPOCHS):
            idxs = idxs_all[torch.randperm(N)]
            approx_kls = []
            for start in range(0, N, MINIBATCH_SIZE):
                b = idxs[start: start + MINIBATCH_SIZE]
                bs = states[b]; ba = actions[b]; blim = limits[b]
                bold = old_logp[b]; bret = returns[b]; badv = advs[b]
                v_old = values[b]

                mu_u, std_u, v = self.net(bs)
                dist_u = torch.distributions.Normal(mu_u, std_u)

                new_log = log_prob_squashed(dist_u, ba, blim)
                ratio = (new_log - bold).exp()

                s1 = ratio * badv
                s2 = torch.clamp(ratio, 1 - CLIP_EPS, 1 + CLIP_EPS) * badv
                p_loss = -torch.min(s1, s2).mean()
```

```
                v_pred = v.squeeze(1)
                v_clipped = v_old + (v_pred - v_old).clamp(-VCLIP, VCLIP)
                v_loss1 = (v_pred - bret).pow(2)
                v_loss2 = (v_clipped - bret).pow(2)
                v_loss = 0.5 * torch.max(v_loss1, v_loss2).mean()

                ent = entropy_base(dist_u).mean()

                loss = p_loss + v_loss - ENTROPY_COEF * ent

                self.optimizer.zero_grad()
                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.net.parameters(), 0.5)
                self.optimizer.step()

                with torch.no_grad():
                    approx_kls.append((bold - new_log).mean().item())
                    clip_frac = ((ratio - 1.0).abs() >
CLIP_EPS).float().mean().item()
                    clip_fracs.append(clip_frac)
                    p_losses.append(p_loss.item())
                    v_losses.append(v_loss.item())
                    ents.append(ent.item())

            last_mean_kl = float(np.mean(approx_kls)) if approx_kls else 0.0
            if last_mean_kl > KL_TARGET:
                RAW(f"[PPO] Early stop epoch due to KL={last_mean_kl:.4f} >
{KL_TARGET}")
                break

        with torch.no_grad():
            total_after = sum(p.abs().sum().item() for p in
self.net.parameters())
        delta_param = total_after - total_before

        clip_frac = float(np.mean(clip_fracs)) if clip_fracs else 0.0
        p_loss_mean = float(np.mean(p_losses)) if p_losses else 0.0
        v_loss_mean = float(np.mean(v_losses)) if v_losses else 0.0
        ent_mean = float(np.mean(ents)) if ents else 0.0

        RAW(f"[PPO][upd] Δ|θ|={delta_param:.4f} KL={last_mean_kl:.4f}
clip_frac={clip_frac:.3f} "
            f"p={p_loss_mean:.4f} v={v_loss_mean:.4f} H={ent_mean:.4f}
episodes={self.episode_count+1}")
        LOG.info("[PPO][upd] dTheta=%.4f KL=%.4f clip_frac=%.3f p=%.4f
v=%.4f H=%.4f episodes=%d",
                 delta_param, last_mean_kl, clip_frac, p_loss_mean,
v_loss_mean, ent_mean, self.episode_count + 1)

        self.episode_count += 1
        self._save()
        RAW(f"[ppo] update done ep={self.episode_count}")
        LOG.info("[ppo] update done ep=%d", self.episode_count)

    def _compute_gae(self, rewards, values, next_values, dones):
        N = len(rewards)
        advs = torch.zeros(N)
        gae = 0.0
        for t in reversed(range(N)):
```

```
            mask = 1.0 - dones[t]
            delta = rewards[t] + GAMMA * mask * next_values[t] - values[t]
            gae = delta + GAMMA * GAE_LAMBDA * mask * gae
            advs[t] = gae
        returns = advs + values
        return returns, advs

# ============ ENV ============
class LaneDrivingEnv:
    def __init__(self):
        RAW("[boot] LaneDrivingEnv.__init__")
        self.outer_normal, self.outer_intersection = load_boxes(BOXES_CSV)
        self.inner_normal, self.inner_intersection =
load_boxes(BOXES_INNER_CSV)

        self.current_step = 0
        self.visited = set()
        self.next_box_idx = 0
        self.max_visited = 0
        self.done = False
        self.outside_steps = 0

        self.prev_pos = None
        self.stuck_steps = 0

        self._last_progress_step = 0
        self._last_progress_idx = 0

        # NEW: episode analytics fields
        self._spawn_side_out = 0   # 1 if OUTER set, 0 if INNER set
        self._spawn_box_id   = -1
        self.last_box_id     = -1

        self._choose_side_and_setup(is_inner=False, spawn_box_id=None)

        CAMInit(QQVGA)  # 160x120
        LCDClear()
        LCDImageStart(0, 0, CAMWIDTH, CAMHEIGHT)
        if USE_GRAY_CAMERA: LCDImageGray(CAMGetGray())
        else:               LCDImage(CAMGet())

        in_channels = 1 if USE_GRAY_CAMERA else (4 if USE_HUE_SINCOS else 3)
        self.agent = PPOAgent(input_shape=(in_channels, CAMHEIGHT,
CAMWIDTH))

        # CSV headers (create if missing)
        if not os.path.exists(LOG_CSV):
            with open(LOG_CSV, 'w', newline='') as f:

csv.writer(f).writerow(["Episode","Steps","Progress","LapComplete","TotalRew
ard","AvgReward"])
        if not os.path.exists(GRAPH_CSV):
            with open(GRAPH_CSV, 'w', newline='') as f:

csv.writer(f).writerow(["Episode","SignedSpawnSide","Progress","spawnBox","f
inalBox","LapComplete"])

        RAW("[boot] LaneDrivingEnv ready")

    # ----- map transform -----
```

```
    def _recalc_map_transform(self):
        xs = [p[0] for b in self.normal_boxes for p in b.corners]
        ys = [p[1] for b in self.normal_boxes for p in b.corners]
        if self.intersection_box is not None:
            xs += [p[0] for p in self.intersection_box.corners]
            ys += [p[1] for p in self.intersection_box.corners]
        self.WMINX, self.WMAXX = min(xs), max(xs)
        self.WMINY, self.WMAXY = min(ys), max(ys)
        sx = (MAP_W - 2*MAP_MARGIN) / max(1.0, (self.WMAXX - self.WMINX))
        sy = (MAP_H - 2*MAP_MARGIN) / max(1.0, (self.WMAXY - self.WMINY))
        self.MAP_S = min(sx, sy)

    def _w2s(self, px, py):
        sx = int(MAP_X + MAP_MARGIN + (px - self.WMINX) * self.MAP_S)
        sy = int(MAP_Y + MAP_H - MAP_MARGIN - (py - self.WMINY) *
self.MAP_S)
        return sx, sy

    # ----- setup helpers -----
    def _choose_side_and_setup(self, is_inner, spawn_box_id):
        if is_inner:
            self.normal_boxes     = self.inner_normal
            self.intersection_box = self.inner_intersection
        else:
            self.normal_boxes     = self.outer_normal
            self.intersection_box = self.outer_intersection

        self.num_normal    = len(self.normal_boxes)
        self.box_id_to_idx = {b.box_id: i for i, b in
enumerate(self.normal_boxes)}

        self.visited.clear()
        if spawn_box_id is not None:
            self.visited.add(spawn_box_id)
            idx = self.box_id_to_idx.get(spawn_box_id, 0)
            self.next_box_idx = (idx + 1) % self.num_normal
            self.max_visited = 1
        else:
            self.next_box_idx = 0
            self.max_visited = 0

        self._recalc_map_transform()

    def _centroid(self, box):
        xs = [p[0] for p in box.corners]; ys = [p[1] for p in box.corners]
        return (sum(xs)/4.0, sum(ys)/4.0)

    def _angle_diff_deg(self, a, b):
        d = (a - b) % 360.0
        if d >= 180.0: d -= 360.0
        return d

    def _signed_lateral_distance(self, px, py, ax, ay, bx, by):
        vx, vy = bx - ax, by - ay
        wx, wy = px - ax, py - ay
        seg_len2 = vx*vx + vy*vy + 1e-9
        t = max(0.0, min(1.0, (wx*vx + wy*vy)/seg_len2))
        cx, cy = ax + t*vx, ay + t*vy
        sign = np.sign(vx*(py - ay) - vy*(px - ax))
        d = sign * math.hypot(px - cx, py - cy)
```

```
        return d, (cx, cy), t

    def _shaping_score(self, x, y, yaw_deg, ax, ay, bx, by):
        d, _, _ = self._signed_lateral_distance(x, y, ax, ay, bx, by)
        seg_yaw = math.degrees(math.atan2(by - ay, bx - ax))
        heading_err = abs(self._angle_diff_deg(seg_yaw, yaw_deg))
        dist_factor = math.exp(-SHAPE_KD * abs(d))
        head_factor = 1.0 / (1.0 + (heading_err / HEAD_E0_DEG)**2)
        score = MAX_SHAPE * dist_factor * head_factor
        return score, dist_factor, head_factor

    # small pixel cross
    def _draw_cross(self, x, y, size=2, color=YELLOW):
        scr_w = MAP_X + MAP_W
        scr_h = max(MAP_H, CAMHEIGHT)
        for dx in range(-size, size+1):
            xx = x + dx
            if 0 <= xx < scr_w and 0 <= y < scr_h: LCDPixel(xx, y, color)
        for dy in range(-size, size+1):
            yy = y + dy
            if 0 <= x < scr_w and 0 <= yy < scr_h: LCDPixel(x, yy, color)

    def _draw_overlays(self, ax, ay, bx, by, rx, ry):
        # frame
        LCDLine(MAP_X, MAP_Y, MAP_X+MAP_W, MAP_Y, WHITE)
        LCDLine(MAP_X+MAP_W, MAP_Y, MAP_X+MAP_W, MAP_Y+MAP_H, WHITE)
        LCDLine(MAP_X+MAP_W, MAP_Y+MAP_H, MAP_X, MAP_Y+MAP_H, WHITE)
        LCDLine(MAP_X, MAP_Y+MAP_H, MAP_X, MAP_Y, WHITE)
        # boxes
        for box in self.normal_boxes:
            pts = [self._w2s(px, py) for px, py in box.corners]
            for i in range(4):
                x0, y0 = pts[i]; x1, y1 = pts[(i+1) % 4]
                LCDLine(x0, y0, x1, y1, WHITE)
        if self.intersection_box is not None:
            pts = [self._w2s(px, py) for px, py in
self.intersection_box.corners]
            for i in range(4):
                x0, y0 = pts[i]; x1, y1 = pts[(i+1) % 4]
                LCDLine(x0, y0, x1, y1, RED)
        # target segment
        axs, ays = self._w2s(ax, ay); bxs, bys = self._w2s(bx, by)
        LCDLine(axs, ays, bxs, bys, GREEN)
        # robot cross
        rxs, rys = self._w2s(rx, ry)
        self._draw_cross(rxs, rys, size=2, color=YELLOW)

    # ----- core API -----
    def reset(self):
        self.current_step = 0
        self.done = False
        self.outside_steps = 0
        self.prev_pos = None
        self.stuck_steps = 0
        self._last_progress_step = 0
        self._last_progress_idx = self.next_box_idx
        self.last_box_id = -1

        x, y, phi, spawn_box_id, is_inner = random.choice(ALL_SPAWNS)
        SIMSetRobot(0, x, y, 0, phi)
```

```
        self._choose_side_and_setup(is_inner, spawn_box_id)

        # Record spawn fields
        self._spawn_box_id = int(spawn_box_id)
        self._spawn_side_out = 0 if is_inner else 1

        LCDClear()
        LCDImageStart(0, 0, CAMWIDTH, CAMHEIGHT)
        if USE_GRAY_CAMERA:
            g = CAMGetGray(); LCDImageGray(g); state =
self._process_frame(g)
        else:
            img = CAMGet(); LCDImage(img); state = self._process_frame(img)

        side_str = 'inner' if is_inner else 'outer'
        RAW(f"[reset] side={side_str}, start_box={spawn_box_id} at
({x},{y},{phi})")
        try:
            m,s,mn,mx = float(state.mean()), float(state.std()),
float(state.min()), float(state.max())
            RAW(f"[DBG][obs] mean={m:.4f} std={s:.4f} min={mn:.4f}
max={mx:.4f}")
            LOG.info("[DBG][obs] mean=%.4f std=%.4f min=%.4f max=%.4f",
m,s,mn,mx)
        except Exception as e:
            RAW(f"[DBG][obs] stat fail: {e}"); LOG.info("[DBG][obs] stat
fail: %r", e)
        return state

    def _process_frame(self, frame):
        if USE_GRAY_CAMERA:
            g = np.reshape(frame, (CAMHEIGHT, CAMWIDTH)).astype(np.float32)
/ 255.0
            return g[None, ...]
        h, s, i = IPCol2HSI(frame)
        h = np.reshape(h, (CAMHEIGHT, CAMWIDTH)).astype(np.float32) / 360.0
        s = np.reshape(s, (CAMHEIGHT, CAMWIDTH)).astype(np.float32) / 255.0
        i = np.reshape(i, (CAMHEIGHT, CAMWIDTH)).astype(np.float32) / 255.0
        if USE_HUE_SINCOS:
            hcos = (np.cos(2*np.pi*h) * s)
            hsin = (np.sin(2*np.pi*h) * s)
            arr = np.dstack((hcos, hsin, s, i))
        else:
            arr = np.dstack((h, s, i))
        return arr.transpose(2,0,1)

    def _which_box(self, x, y):
        for box in self.normal_boxes:
            if box.contains(x, y): return "normal", box.box_id
        if (self.intersection_box is not None) and
self.intersection_box.contains(x, y):
            return "intersection", self.intersection_box.box_id
        return None, None

    def step(self, ang_action):
        ang = ang_action
        VWSetSpeed(FIXED_LIN_SPEED, ang)
        OSWait(100)

        LCDImageStart(0, 0, CAMWIDTH, CAMHEIGHT)
```

```
        if USE_GRAY_CAMERA:
            g = CAMGetGray(); LCDImageGray(g); next_state =
self._process_frame(g)
        else:
            img = CAMGet(); LCDImage(img); next_state =
self._process_frame(img)

        x_raw, y_raw, _, phi_raw = SIMGetRobot(0)
        x = x_raw.value if hasattr(x_raw,'value') else x_raw
        y = y_raw.value if hasattr(y_raw,'value') else y_raw
        phi = phi_raw.value if hasattr(phi_raw,'value') else phi_raw

        # stuck detection (spin / tiny motion)
        if self.prev_pos is None:
            self.prev_pos = (x, y)
        else:
            dx = x - self.prev_pos[0]; dy = y - self.prev_pos[1]
            dist = math.hypot(dx, dy)
            if dist < STUCK_DIST_EPS and FIXED_LIN_SPEED > 50:
                self.stuck_steps += 1
            else:
                self.stuck_steps = 0
            self.prev_pos = (x, y)

        reward = 0.0
        done = False

        if self.stuck_steps >= STUCK_MAX_STEPS:
            reward += STUCK_PENALTY + STEP_PENALTY
            self.current_step += 1
            return next_state, reward, True,
{'box_type':'stuck','box_id':None,'visited_count':len(self.visited)}

        box_type, box_id = self._which_box(x, y)

        if box_type is None:
            reward += OUTSIDE_STEP_PENALTY + STEP_PENALTY
            self.outside_steps += 1
            done = self.outside_steps >= OUTSIDE_GRACE_STEPS
            if DEBUG_RENDER_EVERY and (self.current_step %
DEBUG_RENDER_EVERY == 0):
                cur_idx = (self.next_box_idx - 1) % self.num_normal
                ax, ay = self._centroid(self.normal_boxes[cur_idx])
                bx, by =
self._centroid(self.normal_boxes[self.next_box_idx])
                self._draw_overlays(ax, ay, bx, by, x, y)
            self.current_step += 1
            return next_state, reward, done,
{'box_type':None,'box_id':None,'visited_count':len(self.visited)}

        self.outside_steps = 0

        # record last seen box id (normal or intersection)
        if box_id is not None:
            self.last_box_id = int(box_id)

        if box_type == "normal":
            idx = self.box_id_to_idx[box_id]
            if idx == self.next_box_idx:
                reward += 5.0
```

```
                self.visited.add(box_id)
                if len(self.visited) % 20 == 0: reward += 100.0
                self.next_box_idx = (self.next_box_idx + 1) %
self.num_normal
                self._last_progress_step = self.current_step
                self._last_progress_idx = self.next_box_idx
            elif box_id in self.visited:
                pass
            else:
                reward += WRONG_BOX_PENALTY
            if len(self.visited) > self.max_visited:
                self.max_visited = len(self.visited)

        # direct per-step shaping
        if LINE_SHAPING_REWARD:
            cur_idx = (self.next_box_idx - 1) % self.num_normal
            ax, ay = self._centroid(self.normal_boxes[cur_idx])
            bx, by = self._centroid(self.normal_boxes[self.next_box_idx])
            score, _, _ = self._shaping_score(x, y, phi, ax, ay, bx, by)
            reward += score
            if DEBUG_RENDER_EVERY and (self.current_step %
DEBUG_RENDER_EVERY == 0):
                self._draw_overlays(ax, ay, bx, by, x, y)

        # angle L2 control cost (discourage “spin”)
        # NOTE: penalize based on executed angle range actually allowed
(agent clamps & scales)
        cur_exec_limit = self.agent.current_exec_limit() * EXEC_CMD_SCALE
        norm_ang = (float(ang) / max(1e-6, cur_exec_limit))
        reward += -ANGLE_L2_COEF * (norm_ang ** 2)

        # no-progress nudge
        if (self.current_step - self._last_progress_step) >=
NO_PROGRESS_STEPS:
            reward += NO_PROGRESS_PENALTY
            self._last_progress_step = self.current_step  # throttle

        reward += STEP_PENALTY
        self.current_step += 1
        if len(self.visited) == self.num_normal:
            reward += 1000.0; done = True
        if self.current_step >= 3000:
            done = True

        return next_state, reward, done, {
            'box_type': box_type,
            'box_id': box_id,
            'visited_count': len(self.visited),
            'x': x, 'y': y, 'phi': phi
        }

    def run_episode(self, episode_idx, mode="train"):
        self.mode = mode
        state = self.reset()
        total_reward = 0.0
        step_count = 0
        lap_complete = 0

        while True:
            deterministic = (mode == "eval")
```

```
            ang, old_logp, val, a_cont_exec, limit_used =
self.agent.get_action(state, deterministic=deterministic)

            next_state, reward, done, info = self.step(ang)
            LOG.info("ep=%d step=%d ang=%d a_exec=%.2f limit=%.1f r=%.3f
box=%s visited=%d/%d",
                     episode_idx, step_count, ang,
float(a_cont_exec.squeeze().item()),
                     limit_used, reward, str(info.get('box_id','N/A')),
len(self.visited), self.num_normal)

            total_reward += reward

            if mode == "train":
                next_v = self.agent.value_of(next_state)
                self.agent.add_transition({
                    'state': state,
                    'a_exec': a_cont_exec.squeeze().view(1),   # executed
continuous action (deg)
                    'limit_used': limit_used,                  # per-sample
limit (incl. scale)
                    'old_logp': old_logp.view(1),
                    'value': val.item(),
                    'reward': reward,
                    'done': float(done),
                    'next_value': next_v
                })
                self.agent.maybe_update(force=False)

            state = next_state
            step_count += 1

            if reward >= 1000.0: lap_complete = 1
            if mode == "eval" and step_count >= EVAL_MAX_STEPS: done = True
            if done: break

        if mode == "train":
            self.agent.maybe_update(force=True)

        avg_reward = total_reward / step_count if step_count > 0 else 0.0
        with open(LOG_CSV, "a", newline="") as f:
            csv.writer(f).writerow([episode_idx, step_count,
self.max_visited, lap_complete, total_reward, avg_reward])

        # Write graph_data.csv row
        final_box = int(self.last_box_id) if self.last_box_id is not None
else -1
        with open(GRAPH_CSV, 'a', newline='') as f:
            csv.writer(f).writerow([
                episode_idx,
                int(self._spawn_side_out),            # 1 outer, 0 inner
                int(self.max_visited),                # Progress
                int(self._spawn_box_id),              # spawnBox
                final_box,                            # finalBox (or -1)
                int(lap_complete),
            ])

        # Success backup on lap completion (both train & eval)
        if lap_complete == 1:
            self.agent.save_success_backup(episode_idx)
```

```
        RAW(f"[Episode {episode_idx} Summary] Steps={step_count}, "
            f"Progress={self.max_visited}/{self.num_normal},
LapComplete={lap_complete}, "
            f"TotalReward={total_reward:.2f}, AvgReward={avg_reward:.4f}")
        LOG.info("[ep_summary] ep=%d steps=%d total=%.2f avg=%.4f
progress=%d/%d",
                 episode_idx, step_count, total_reward, avg_reward,
self.max_visited, self.num_normal)
        VWSetSpeed(0, 0)

# ============ MAIN ============
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--mode", choices=["train", "eval"], default="eval")
    parser.add_argument("--episodes", type=int, default=5000)
    args = parser.parse_args()

    env = LaneDrivingEnv()

    LCDMenu("train", "eval", "", "exit")
    reading = True
    while reading:
        key = KEYRead()
        if key == KEY1:
            for ep in range(args.episodes):
                env.run_episode(episode_idx=ep, mode="train")
        if key == KEY2:
            for ep in range(args.episodes):
                env.run_episode(episode_idx=ep, mode="eval")
        if key == KEY4:
            LCDClear()
            reading = False
```

Appendix C. modelConversion.py

```
#!/usr/bin/env python3
import argparse, os, sys
import torch
import torch.nn as nn

# ---- must match training net from torchTraining.py ----
U_STD_INIT = 0.35  # same as in training

class ActorCritic(nn.Module):
    def __init__(self, input_shape=(1, 120, 160)):
        super().__init__()
        c, h, w = input_shape
        self.conv = nn.Sequential(
            nn.Conv2d(c, 16, kernel_size=5, stride=2, padding=2), nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=5, stride=2, padding=2),
nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
nn.ReLU(),
            nn.Flatten()
        )
        with torch.no_grad():
            conv_out = self.conv(torch.zeros(1, c, h, w)).shape[1]
        self.fc = nn.Sequential(
            nn.Linear(conv_out, 256), nn.Tanh(),
            nn.Linear(256, 64),       nn.Tanh()
        )
        self.mu_u       = nn.Linear(64, 1)
        self.log_std_u  = nn.Parameter(torch.log(torch.tensor([U_STD_INIT],
dtype=torch.float32)))
        self.value_head = nn.Linear(64, 1)

    def forward(self, x):
        z = self.fc(self.conv(x))
        mu_u  = self.mu_u(z)
        std_u = self.log_std_u.exp().expand_as(mu_u)
        val   = self.value_head(z)
        return mu_u, std_u, val
# --------------------------------------------------------------

def load_checkpoint_into_model(model, ckpt_path):
    ck = torch.load(ckpt_path, map_location="cpu")
    # support both {'model_state_dict': ...} or raw state_dict
    src = ck.get("model_state_dict", ck)
    dst = model.state_dict()
    matched = {k: v for k, v in src.items() if k in dst and v.shape ==
dst[k].shape}
    dst.update(matched)
    model.load_state_dict(dst, strict=False)
    print(f"[convert] Loaded params {len(matched)}/{len(dst)} from
{os.path.basename(ckpt_path)}")

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--model-file", default=None,
                    help="ignored (kept for compatibility with earlier
instructions)")
    ap.add_argument("--ckpt",   required=True, help="Path to trained .pt
checkpoint")
```

```
    ap.add_argument("--onnx",   required=True, help="Output .onnx path")
    ap.add_argument("--opset",  type=int, default=17, help="ONNX opset
version")
    ap.add_argument("--height", type=int, default=120, help="Input image
height")
    ap.add_argument("--width",  type=int, default=160, help="Input image
width")
    ap.add_argument("--channels", type=int, default=1, help="Input channels
(1 for grayscale)")
    args = ap.parse_args()

    input_shape = (args.channels, args.height, args.width)
    model = ActorCritic(input_shape=input_shape).eval()
    load_checkpoint_into_model(model, args.ckpt)

    dummy = torch.zeros(1, *input_shape, dtype=torch.float32)
    torch.onnx.export(
        model, dummy, args.onnx,
        input_names=["obs"],
        output_names=["mu_u", "std_u", "value"],
        dynamic_axes={"obs": {0: "batch"}, "mu_u": {0: "batch"}, "std_u":
{0: "batch"}, "value": {0: "batch"}},
        opset_version=args.opset,
        do_constant_folding=True
    )
    print(f"[convert] Wrote ONNX -> {args.onnx} (opset {args.opset})")

if __name__ == "__main__":
    main()
```

Appendix D. runONNXModel.py

```
#!/usr/bin/env python3
# runONNXModel.py — ONNX inference on EyeBot with Pause/Exit + PSD(front)
failsafe
import os, sys, time, glob, argparse
import numpy as np

# Quiet onnxruntime logs a bit; and force CPU
os.environ.setdefault("ORT_LOG_SEVERITY_LEVEL", "3")

try:
    import onnxruntime as ort
except Exception as e:
    print("[run] onnxruntime not available:", e)
    sys.exit(1)

# RoBIOS / EyeBot API
from eye import *  # noqa

# ------------ constants (match your training) ------------
CAMWIDTH, CAMHEIGHT = 160, 120
FIXED_LIN_SPEED     = 150
BASE_ANG_LIMIT_DEG  = 45.0
EXEC_CMD_SCALE      = 1.0
EVAL_MAX_STEPS      = 1500
FRAME_DELAY_MS      = 100  # ms

# PSD failsafe
DEFAULT_FRONT_THRESHOLD_MM = 150  # pause below this

# ------------ helpers ------------
def frame_to_obs_gray(frame):
    """Convert CAMGetGray() to [1,1,H,W] float32 in [0,1]."""
    if isinstance(frame, (bytes, bytearray, memoryview)):
        arr = np.frombuffer(frame, dtype=np.uint8)
    else:
        arr = np.array(frame, dtype=np.uint8)
    arr = arr.reshape(CAMHEIGHT, CAMWIDTH).astype(np.float32) / 255.0
    return arr[np.newaxis, np.newaxis, :, :]  # NCHW

def pick_action(mu_u, std_u, stochastic: bool, limit_deg: float) ->
tuple[int, float]:
    """μ/σ are pre-tanh. Returns (int steering cmd, continuous deg before
int)."""
    u = mu_u + std_u * np.random.randn() if stochastic else mu_u
    a_cont = np.tanh(u) * limit_deg
    ang_cmd = int(np.clip(a_cont * EXEC_CMD_SCALE,
                          -limit_deg * EXEC_CMD_SCALE,
                          +limit_deg * EXEC_CMD_SCALE))
    return ang_cmd, float(a_cont)

def load_session(onnx_path: str):
    sess = ort.InferenceSession(onnx_path,
providers=["CPUExecutionProvider"])
    outs = {o.name for o in sess.get_outputs()}
    if not {"mu_u", "std_u", "value"}.issubset(outs):
        raise RuntimeError(f"[run] Unexpected ONNX outputs in {onnx_path}:
{outs}")
    return sess
```

```
def discover_models(initial: str | None):
    """Return (models_list, start_index). If initial is a file, prefer its
folder."""
    if initial is None:
        models = sorted(glob.glob("*.onnx"))
        return models, 0
    if os.path.isdir(initial):
        models = sorted(glob.glob(os.path.join(initial, "*.onnx")))
        return models, 0
    folder = os.path.dirname(initial) or "."
    models = sorted(glob.glob(os.path.join(folder, "*.onnx")))
    if not models:
        models = [initial]
    try:
        idx = models.index(initial)
    except ValueError:
        models.insert(0, initial)
        idx = 0
    return models, idx

def set_speed(lin: int, ang: int):
    VWSetSpeed(int(lin), int(ang))

def draw_menu_idle():
    LCDMenu("train", "eval", "model", "exit")

def draw_menu_running_eval(paused: bool):
    LCDMenu("pause" if not paused else "play", "", "", "exit")

def draw_menu_running_train():
    LCDMenu("running", "", "", "exit")

def front_blocked_psd(threshold_mm: int) -> bool:
    """True if front PSD reports a valid distance below threshold."""
    try:
        d = PSDGet(PSD_FRONT)  # mm; requires PSD calibrated in HDT
        if isinstance(d, (list, tuple)):
            d = d[0] if d else 0
        d = int(d)
        # Treat <=0 as invalid/no return; only block on positive values
        return (d > 0) and (d < int(threshold_mm))
    except Exception:
        # If PSD not available, don't block (or you could default to safe
stop)
        return False

# ------------ main control loop ------------
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--onnx", default=None,
                    help="Path to initial ONNX (if omitted, first *.onnx in
current folder is used)")
    ap.add_argument("--limit-deg", type=float, default=BASE_ANG_LIMIT_DEG,
                    help="Steering limit in degrees")
    ap.add_argument("--episodes", type=int, default=9999999,
                    help="How many episodes to run when a mode starts")
    ap.add_argument("--front-threshold-mm", type=int,
default=DEFAULT_FRONT_THRESHOLD_MM,
                    help="PSD(front) pause threshold in mm")
```

```
    args = ap.parse_args()

    # Camera & LCD
    CAMInit(QQVGA)  # 160x120
    LCDClear()
    LCDImageStart(0, 0, CAMWIDTH, CAMHEIGHT)

    # ONNX model discovery/load
    models, model_idx = discover_models(args.onnx)
    if not models:
        print("[run] No .onnx model found. Place one next to this script or
pass --onnx.")
        return
    current_model = models[model_idx]
    sess = load_session(current_model)

    # State
    running = False
    mode = None             # "train" or "eval"
    user_paused = False     # manual pause in eval
    draw_menu_idle()

    # -------- outer UI loop --------
    while True:
        if not running:
            # Idle menu: wait for a key
            key = KEYRead()
            if key == KEY1:  # train (stochastic μ+σ·ε)
                running = True
                mode = "train"
                user_paused = False
                draw_menu_running_train()
            elif key == KEY2:  # eval (deterministic μ)
                running = True
                mode = "eval"
                user_paused = False
                draw_menu_running_eval(paused=user_paused)
            elif key == KEY3:  # switch model
                if len(models) == 1:
                    folder = os.path.dirname(models[0]) or "."
                    more = sorted(glob.glob(os.path.join(folder, "*.onnx")))
                    if more:
                        models = more
                model_idx = (model_idx + 1) % len(models)
                current_model = models[model_idx]
                sess = load_session(current_model)
                draw_menu_idle()
            elif key == KEY4:
                LCDClear()
                set_speed(0, 0)
                break
            continue

        # -------- running loop (train/eval) --------
        steps = 0
        episodes_to_run = args.episodes
        while running and episodes_to_run > 0:
            steps = 0
            while running and steps < EVAL_MAX_STEPS:
                # Key handling every cycle
```

```
            key = KEYRead()
            if key == KEY4:  # immediate stop & return to idle menu
                set_speed(0, 0)
                running = False
                mode = None
                user_paused = False
                draw_menu_idle()
                break

            if mode == "eval" and key == KEY1:
                # Toggle manual pause/play
                user_paused = not user_paused
                draw_menu_running_eval(paused=user_paused)
                if user_paused:
                    set_speed(0, 0)

            # Camera frame -> LCD -> obs
            g = CAMGetGray()
            LCDImageGray(g)
            obs = frame_to_obs_gray(g)

            # PSD(front) failsafe (auto-pause when blocked; auto-resume
when clear)
            auto_paused = front_blocked_psd(args.front_threshold_mm)
            effective_paused = (user_paused if mode == "eval" else
False) or auto_paused

            if effective_paused:
                set_speed(0, 0)
                OSWait(FRAME_DELAY_MS)
                continue  # don't advance step budget while paused

            # Inference
            mu_u, std_u, _ = sess.run(["mu_u", "std_u", "value"],
{"obs": obs})
            mu = float(mu_u[0, 0]); std = float(std_u[0, 0])
            ang_cmd, _a_cont = pick_action(mu, std, stochastic=(mode ==
"train"),
                                           limit_deg=args.limit_deg)

            # Drive
            set_speed(FIXED_LIN_SPEED, ang_cmd)
            OSWait(FRAME_DELAY_MS)
            steps += 1

        # Episode ended (budget exhausted or aborted)
        set_speed(0, 0)
        episodes_to_run -= 1

        if not running:
            break

        # For long-running modes, continue unless stopped
        if mode == "eval":
            draw_menu_running_eval(paused=user_paused)
        else:
            draw_menu_running_train()

    if running and episodes_to_run <= 0:
        running = False
```

```
            mode = None
            user_paused = False
            draw_menu_idle()

if __name__ == "__main__":
    main()
```