Reinforcement Learning for Steering Control in the Formula Student Driverless Simulator: Structural Failure Modes of Soft Actor-Critic and the Case for PPO

Daniel Ortiz Valencia

Department of Electrical and Computer Engineering, Colorado State University

Fort Collins, CO, USA | Jose.Ortiz-Valencia@colostate.edu

Abstract: We study the replacement of a hand-coded reactive steering controller with a learned policy in the Formula Student Driverless Simulator (FSDS). The work has three parts: a validated perception and control baseline that fuses LiDAR clustering with a YOLO camera detector, a reinforcement-learning (RL) environment that exposes a 41-dimensional cone observation and a single steering action, and a sequence of training experiments. To separate control from perception, the policy is trained against ground-truth cone positions and warm-started with behavior cloning. We evaluate Soft Actor-Critic (SAC) across six configurations (Tests 9 to 13). All six runs ended in collapse. We attribute this to three structural properties of the algorithm rather than to tuning: replay-buffer poisoning, automatic-entropy collapse, and catastrophic forgetting after reward changes. We then adopt Proximal Policy Optimization (PPO), whose on-policy formulation, trust-region clipping, and fixed entropy coefficient remove each of these failure modes. The result is a reproducible FSDS training environment and a catalogue of failure modes with recognizable training-log signatures.

Index Terms: Reinforcement learning, autonomous racing, Soft Actor-Critic, Proximal Policy Optimization, behavior cloning, Formula Student Driverless, reward shaping.

I. Introduction

Formula SAE (FSAE) is a collegiate engineering competition in which student teams design, build, and validate open-wheel prototype vehicles. The Driverless class requires the human driver to be replaced by an onboard autonomous stack that performs perception, localization, and motion planning in an unmapped, cone-delimited environment [1]. The Formula Student Driverless Simulator (FSDS) provides a physics-based virtual environment, built on Unreal Engine 4 and AirSim, in which such a stack can be validated before deployment on hardware [2].

This paper addresses one question: can a reinforcement-learning agent learn to steer an FSAE car around a cone track at least as well as a hand-coded controller, and what does that require in practice. The main cost was infrastructure correctness and the fit between the learning algorithm and the problem, not model design.

We restrict the learning problem to steering. Longitudinal control is handled by a proportional speed controller, so the agent makes one decision per step: how far to turn. A single-axis action keeps the reward design tractable and makes the learned behavior easier to analyze.

Contributions.

- A reproducible FSDS RL environment (41-D observation, 1-D steering, a three-term reward), validated one component at a time, with the simulator and reward bugs documented.
- An empirical study of SAC across six configurations showing that its failures on single-environment continuous control are structural and have recognizable training-log signatures.
- A justification for moving to PPO that maps each SAC failure mode to a PPO design property that removes it.

II. Background and Related Work

RL design choices follow eight axes. FSAE racing maps to a continuous action, a continuous and partially observable state, stochastic dynamics, an episodic setting, dense reward when shaped properly, a single agent, and cheap simulation. That combination points to actor-critic continuous-control methods such as SAC, TD3, or PPO behind a perception front-end [3, 4, 5].

Published FSAE RL is limited. Merton *et al.* [6] is the closest validated reference: the only peer-reviewed work that applies SAC to a cone-delimited FSAE track with real-world transfer (87.5 percent completion), converging in 735 episodes and ablating three reward designs. The Technion FSTD effort reaches a first lap in about one hour of AirSim wall-clock time, but it relies on a VAE image encoder that is unnecessary when a perception stack already produces structured cone detections. The dominant Formula Student programs, including AMZ (ETH Zürich) [1], KIT, MIT Driverless, and Oxford Brookes, have not adopted RL as a primary approach; their main results use classical perception, SLAM, planning, and control. The field is early, which is what makes documented negative results useful.

On representation, the literature agrees that processed cone detections converge faster than raw pixels when a perception stack exists. On bootstrapping, warm-starting RL with a few expert demonstrations is reported to cut convergence time by factors of 3 to 30 [7]. Both findings shaped the design used here.

III. System Architecture and Perception Baseline

The Python client connects over TCP using `fsds.FSDSClient()`; calling `enableApiControl(True)` transfers full control to the API. Inputs are ground-truth kinematics, RGB and depth images, LiDAR point clouds, and GPS or IMU odometry. Outputs are normalized throttle in $[0, 1]$,

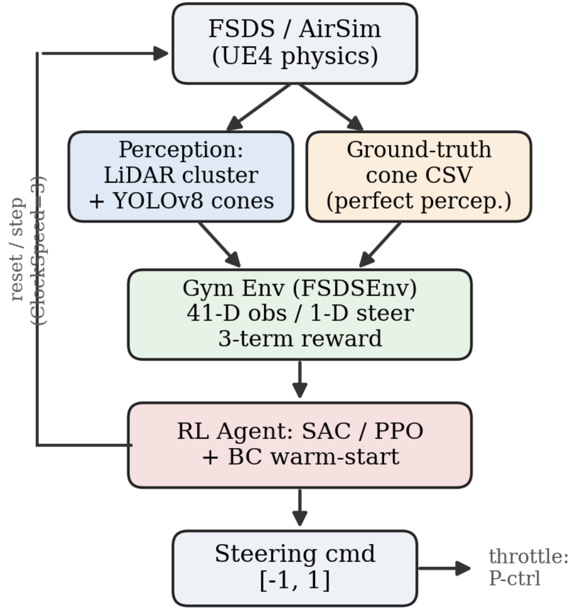

Figure 1. End-to-end pipeline. The simulator is wrapped as a Gymnasium environment. During RL training the perception block is bypassed in favor of ground-truth cone positions to isolate the control problem; the validated LiDAR and camera perception stack remains the deployment path. Throttle is handled by a proportional controller so the agent learns steering alone.

steering in $[-1,1]$, and brake in $[0,1]$.

Before any learning, we built and validated a reactive controller that drives laps without learning. A LiDAR-only baseline clusters the point cloud in $O(n)$ by exploiting the angular scan order: inter-cone gaps that exceed a 0.1 m threshold, calibrated to the roughly 30 cm regulation cone base, partition the sorted array without a full DBSCAN or k-NN search. The car steers from the mean lateral cone position. This baseline completes laps but weaves on straights because it lacks boundary-color information. We therefore added a forward-facing camera. A YOLO detector trained on the FSOCO dataset classifies cone color, and a pinhole ground-plane projection,

$$t = \frac{h}{\mathrm{ray}_y}, \quad x_{\mathrm{g}} = t + x_{\mathrm{off}}, \quad y_{\mathrm{g}} = -\mathrm{ray}_x\, t, \tag{1}$$

converts pixel detections to metric vehicle-frame coordinates ($h = 1.1$ m, $x_{\mathrm{off}} = -0.3$ m). Nearest-neighbor matching fuses camera color with LiDAR depth. Calibration required correcting three coordinate-frame errors found through debug logging: a 1.3 m X-axis reference mismatch, swapped projection axes, and incorrect default camera pitch and offset. The fused proportional controller removed the straight-line oscillation of the LiDAR-only baseline. This controller is reused as the demonstrator for behavior cloning in Section VI.

IV. Infrastructure: Connection, Speed, and Hidden Bugs

Training a network requires hundreds of thousands of steps. At the default 30 Hz, one million steps costs about 9 hours. AirSim exposes a `ClockSpeed` setting. Setting it to 3 and disabling the render viewport (`NoDisplay`) raised the step rate past 1,200 iterations per second, more than 40 times baseline, against a pass criterion of 75 iterations per second, so one million steps complete in under an hour. The speedup has a limit. At a higher clock the physics timestep dt grows and collision detection degrades, so $\times 3$ (d$t \approx 30$ ms) is the safe ceiling and all sensor reads and action writes must be synchronous to avoid stale data.

Two undocumented bugs cost several days. First, at `ClockSpeed` 3 the car sometimes spawned with effectively locked wheels; a mandatory 2 s pause after each reset lets the suspension settle. Second, the car ignored throttle because it spawned in neutral gear, so every control command must explicitly set the gear flag to automatic. Neither was documented; both were found by watching the car refuse to move and reasoning backward. Establishing that the simulator connects, runs fast, and moves on command was the foundation for everything that followed.

V. The Reinforcement-Learning Environment

We wrap FSDS as a Gymnasium environment with three responsibilities: observation, action, and reward.

A. Observation (41-D)

At each step the agent sees the six nearest blue (left) and six nearest yellow (right) cones as (x, y, color), which is 36 values, plus forward speed, lateral speed, yaw rate, previous steering command, and target speed (Fig. 2). This mirrors the information the hand-coded controller uses. Six cones per side give about 15 m of look-ahead at 4 to 8 m/s, enough for one full corner, and the previous-action channels let the network condition on its last output rather than infer it from observation deltas. During training, the environment transforms ground-truth cone positions from a track CSV into the car frame each step. This isolates the control problem from sensor noise so that we can first establish whether the agent can steer at all.

B. Action and Reward

The action is a single scalar in $[-1,1]$ scaled to the physical steering limit. The reward sums three terms (Fig. 3): a velocity reward that rises with speed up to a cap, a centerline reward shaped as a Gaussian that peaks when the car is centered between the cone walls, and a small per-step smoothness penalty on jerky steering changes. Two termination penalties apply: -10 for a cone collision and -5 for losing sight of all cones for three consecutive steps.

Two reward bugs cost real time. The first hardcoded throttle at 40 percent regardless of speed, which let the car reach 17 m/s before doing anything useful; replacing it with the proportional speed controller fixed it. The second was subtle: the centerline

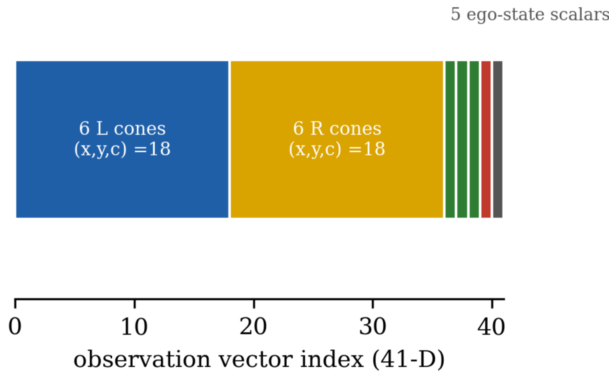

Figure 2. The 41-dimensional observation: 18 values per cone wall plus five ego-state scalars. Cones are sorted by distance and zero-padded.

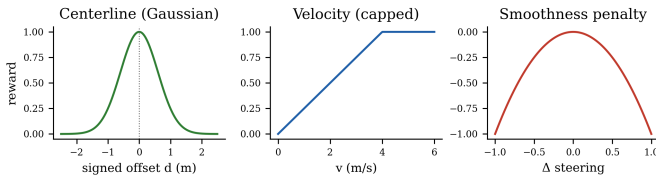

Figure 3. The three reward components. The centerline term must use the absolute offset. An early signed-offset bug rewarded drifting to one side and looked like a tuning problem.

offset was signed, but the reward consumed it without an absolute value, so the agent was rewarded for drifting left. After both fixes, a validation run with the hand-coded controller confirmed a live, varied signal: the centerline component had a standard deviation above 0.05, the smoothness term was always negative, and the mean step reward was about 1.17.

VI. Behavior Cloning: A Warm Start

Random initialization in a racing environment crashes the car within a second of every episode, which makes early learning very slow. We therefore recorded the hand-coded controller driving laps and trained the policy by supervised regression to imitate it, producing `sac_bc_pretrained.zip`. The demonstrator itself was iterated. A Gaussian-weighted lookahead caused phase lag at 75 Hz, a geometric pure-pursuit variant ran off-track at corner exits, and lowering the speed target made matters worse by weakening the proportional correction. At fast loop rates, tuning the gain mattered more than changing the algorithm. The final demonstrator was simple: take the midpoint of the three nearest cones per side and apply a proportional correction with a softer divisor. It hit about one cone per lap, an imperfect but adequate teacher.

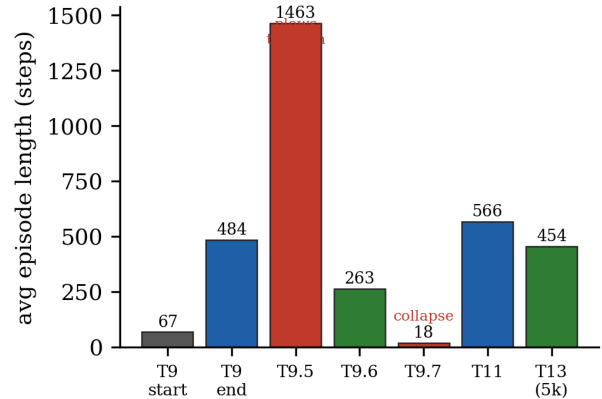

Figure 4. Average episode length across SAC runs. Each apparent success carried a hidden pathology. The long episodes in Test 9.5 came from driving through cones, and Test 9.7 collapsed within 3k steps after a reward change.

VII. Soft Actor-Critic: Six Experiments

With the environment validated and a warm-start policy in hand, we trained with SAC, an off-policy actor-critic method suited to continuous control. The runs are summarized in Fig. 4 and Table 1.

Test 9 (50k steps, BC warm-start) learned, with episode length growing from 67 to 484, but it crashed at the same hairpin every run. Because every episode ended there, the replay buffer held no data past the corner, a condition we call early-termination starvation. Test 9.5 made cone hits a -10 penalty instead of a terminal event; episodes grew to 1463, but the $+50$ lap bonus plus the velocity reward outweighed the penalties, so the agent learned to drive fast through cones, hitting more than 30 per episode. Test 9.6 raised the penalty to -25 and removed a cooldown window; the agent stopped hitting cones and became the cleanest model (263 steps, zero hits), though it still drifted sideways. Test 9.7 reloaded Test 9.6 and increased the centerline weight to fix the drift; episode length collapsed from 131 to 18 in 3k steps and never recovered, because the critic had been calibrated to the old reward. Test 10 was spent fixing evaluation: the lap counter read cumulative referee state, fixed with manual deltas, and the geofence misfired on curves because averaging asymmetric nearest cones gave readings 4 to 5 m off, fixed by using the single closest cone per side. Test 11 multiplied velocity by centerline quality so that a fast but off-center car earns little; centerline quality climbed from 0.45 to 0.74 and one episode reached 566 steps before an early-stopping bug fired at 15k. Test 13 added RPC retries, collapse detection, and frequent checkpoints; it ran well from 5k to 11k steps (reward about $+0.71$), then entropy collapsed, auto-recovery failed three times, and the pruning script deleted the best checkpoint.

Table 1. SAC experiment outcomes.

Test	Avg. len	Outcome and cause
9	67→484	Learns, then crashes at the hairpin every run: the replay buffer never sees the second half (termination starvation).
9.5	1463	Cone hits made non-terminating; the agent learns to treat cones as speed bumps.
9.6	263	Cone penalty raised to -25; cleanest model, zero cone hits, but a consistent sideways drift.
9.7	18	Reloaded Test 9.6 with a stronger centerline weight; collapsed in 3k steps (catastrophic forgetting).
11	566	Multiplicative reward improved centerline quality from 0.45 to 0.74; an early-stopping bug killed the run at 15k steps.
13	454	Strong from 5k to 11k steps, then entropy collapse; auto-recovery failed and pruned the best checkpoint.

VIII. Why SAC Kept Failing

Across Tests 9 to 13, three failure modes recurred. They are properties of the algorithm, not tuning errors.

Replay-buffer poisoning. SAC re-learns from a large memory of past experience. Bad episodes, such as driving through 41 cones, persist in the buffer for thousands of future updates with no mechanism to flush or filter them, so the agent keeps re-learning behavior it should abandon.

Entropy collapse. The automatic entropy tuner can drive the exploration bonus toward zero, making the actor deterministic. Once deterministic, a small systematic bias, for example steering $+0.1$ too far, is reinforced by the critic until the policy outputs full steering lock. This is the pattern observed at step 11k in Test 13.

Catastrophic forgetting after reward changes. Because the critic predicts future reward, its calibration is tied to one reward shape. Changing a single weight after loading a model desynchronizes critic predictions from reality; the agent then optimizes a mispredicted target and collapses. This ended Test 9.7.

In combination, these modes meant that almost any SAC experiment was one bad batch, one entropy step, or one reward change away from collapse, and none of them could be fully prevented by tuning.

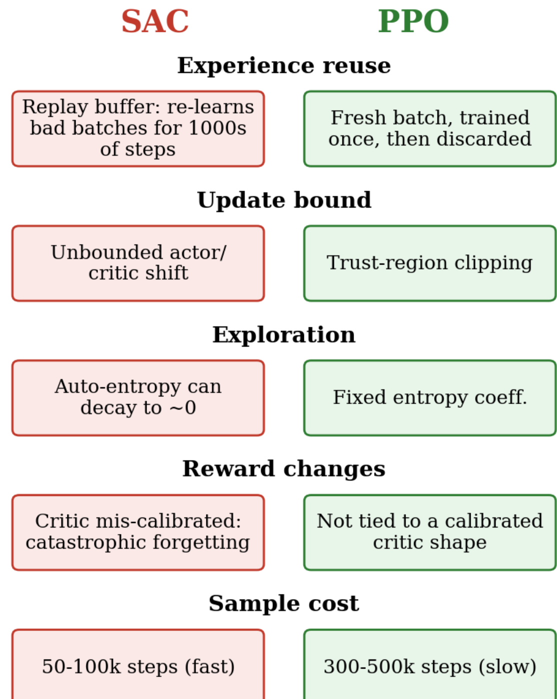

Figure 5. Structural comparison. PPO removes each SAC failure mode by design rather than by tuning, at the cost of more environment interactions.

IX. The Move to PPO

After five collapses across distinct configurations, we concluded that SAC's failure was structural and adopted Proximal Policy Optimization (PPO). Fig. 5 maps the correspondence. PPO has no replay buffer: it collects a fresh batch, trains on it once, and discards it, so there is no contaminated history to re-learn. Trust-region clipping bounds how far the policy can move in one update, which makes catastrophic collapse much less likely. A fixed entropy coefficient prevents silent exploration decay. Fine-tuning is also safer because the policy is not tied to a separately calibrated critic shape.

The cost is sample efficiency. PPO needs roughly 300k to 500k interactions, against SAC's 50k to 100k, which is 8 to 10 wall-clock hours rather than 1 to 2 at the rate of 12 to 15 steps per second measured here. The governing point from every collapsed SAC run is that a slow algorithm that finishes is more useful than a fast one that collapses. A SAC run that collapses produces no usable output, while a PPO run that completes, even slowly, produces a model that can be evaluated and improved.

X. Results, Assets, and Next Steps

This phase produced three SAC reference models: the BC baseline, which drives without crashing immediately; the Test 9.6 model, the cleanest at 263 average steps with zero cone hits but a sideways drift; and the Test 13 step-5000 checkpoint, the

longest-driving at 454 average steps with occasional cone hits. None are competition-ready, but all are useful reference points.

The more durable result is a validated training infrastructure: a correct environment wrapper, a verified reward signal, a working speed-curriculum callback, and a stable telemetry pipeline. The expensive debugging, covering the wheel-lock spawn, the gear flag, the centerline sign error, the lap counter, and the corner distance metric, is complete and does not need to be repeated. The next experiment is PPO training on this same environment and BC starting policy; only the algorithm changes. Open questions remain worth pursuing: a clean wall-clock-to-baseline benchmark on FSDS, whether SAC's sample-efficiency advantage trades against cross-track generalization, whether residual RL on top of the reactive controller transfers to FSDS, and how much domain randomization closes the cone-detection sim-to-real gap.

XI. Conclusion

A learned steering policy for an FSAE car in FSDS is achievable. The main cost of the project was infrastructure correctness and the fit between algorithm and problem, not model novelty. SAC's repeated collapses followed from off-policy replay, automatic entropy tuning, and a critic whose calibration is tied to the reward function. PPO addresses each of these. We report both the successes and the failures so that other teams can reuse the validated infrastructure and avoid the same failure modes.

Acknowledgment

The author thanks Dr. Panahi for advising this project and the RAM Racing EV team for the surrounding FSDS autonomous pipeline.

References

[1] J. Kabzan *et al.*, "AMZ Driverless: The full autonomous racing system," *J. Field Robotics*, 2019.

[2] M. Tigchelaar *et al.*, "FSDS: Formula Student Driverless Simulator," Formula Student Team Delft, open-source project.

[3] T. Haarnoja *et al.*, "Soft Actor-Critic: Off-policy maximum entropy deep RL with a stochastic actor," *ICML*, 2018.

[4] S. Fujimoto *et al.*, "Addressing function approximation error in actor-critic methods (TD3)," *ICML*, 2018.

[5] J. Schulman *et al.*, "Proximal policy optimization algorithms," arXiv:1707.06347, 2017.

[6] M. Merton *et al.*, "Soft Actor-Critic for cone-delimited Formula Student tracks with sim-to-real transfer," 2024.

[7] A. Rajeswaran *et al.*, "Learning complex dexterous manipulation with deep RL and demonstrations (DAPG)," *RSS*, 2018.

[8] University of Auckland FSAE perception dataset, arXiv:2308.13088, 2023.