AWS DeepRacer PPO Agent: Architecture and Training

Adrian S. Kazi

April 2025

1 Introduction

This report outlines the design and training workflow of a custom PPO (Proximal Policy Optimization) agent developed for AWS DeepRacer. The agent learns to drive using inputs from a front-facing camera, trained through reinforcement learning to complete laps on several virtual tracks.

The project is implemented in PyTorch and modularized into core components: **agents.py:** a convolutional actor-critic model that maps visual observations to action logits and value predictions. **buffer.py** collects on-policy experience and computes GAE-based returns and advantages. **transforms:** reshapes and normalizes the front camera input before feeding it into the network. **agentParams.json:** defines how the agent interacts with the environment — including its action space (steering and speed combinations), the sensor it uses (front-facing camera), and its neural network architecture. It also specifies the type of action space (discrete) and the version of the agent spec. **hyperParams.yaml:** control the agent's learning behavior, including training setup, optimization settings, and how experience is collected and processed. **rewardFunction.py** reward functions for 3 set ups: Raw Tracks, Obstacle Avoidance and Head-to-Head. **train.py:** orchestrates the training loop, tracks progress across episodes, and handles PPO updates.

The objective is to train an agent that can consistently complete laps across three tracks: `reInvent2019_wide`, `reInvent2019_track`, and `New_York_Track`. A track is considered mastered once the agent completes five full laps in a row. The next sections describe the agent's architecture, training setup, CNN and the PPO update mechanism.

2 Main Components of Model

2.1 CNN

In DeepRacer, the agent receives raw pixel input from a front-facing camera. These visual observations are high-dimensional and unstructured, so we use a Convolutional Neural Network (CNN) to extract useful spatial features before making decisions.

Model Structure

The CNN processes the input image through a series of convolutional layers that learn to detect patterns such

as edges, corners, and gradually more abstract structures like road boundaries or curves. Instead of treating the image as a flat vector, it preserves spatial information and builds a hierarchy of features relevant for driving. After extracting spatial features, the CNN compresses them into a lower-dimensional representation that captures the most important aspects of the scene. This compact feature vector is then passed to the policy and value networks to decide what action to take and how good the current state is.

Why this setup?

Using a CNN is essential in visual navigation tasks like DeepRacer because it allows the agent to interpret the road layout and react to it without hand-crafted inputs. It balances generalization and efficiency, letting the agent focus on what matters visually — not on memorizing pixels, but learning structure that transfers across tracks.

2.2 PPO

We use PPO (Proximal Policy Optimization) because DeepRacer is not about theoretical elegance — it's about getting a car to drive. We want fast learning, stable updates, and something that won't break after one bad episode.

Intuition

We can think of PPO like traction control in a race car. We want to push hard (learn aggressively), but not so hard that you spin out (destabilize the policy). PPO watches how much the policy changes after each update. If it changes too much, it clips the update. This keeps learning steady and prevents sudden disasters.

How it actually works

Instead of just blindly pushing gradients like in plain REINFORCE, PPO adds a simple guardrail:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min(r_t A_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon) A_t) \right] \tag{1}$$

r_t is how much the new policy changed compared to the old one, A_t is the advantage — how much better or worse an action was, If r_t is too far from 1, we clip it. Simple.

The full loss we use is:

$$L^{\text{PPO}} = L^{\text{CLIP}} - c_1 (V_\theta(s_t) - V_t)^2 + c_2 \cdot \text{Entropy}[\pi_\theta] \tag{2}$$

This means: learn a stable policy, learn a value function that predicts returns, and keep exploring (don't get stuck driving the same way).

In Practice

In DeepRacer, the reward signal is noisy. One lap might go great, another not. PPO handles that well. It won't overfit to one lucky lap. And since we train in episodes and switch tracks often, stability matters more

than theoretical optimality.

PPO is like coaching a driver — we let them try bold moves, but if they jerk the wheel too hard, we bring them back in line gently. That's what makes it perfect for DeepRacer.

3 Project Execution Plan

The training process is structured into three phases, each covering a different race type: Time-Trial, Obstacle-Avoidance, and Head-to-Head. For each race type, we execute the following pipeline: 1. Train the agent from scratch on all three tracks (A to Z Speedway, Smile Speedway, Empire City). 2. Once the agent completes 5 consecutive laps on each track, we move to metric evaluation. 3. After all metrics are verified, we generate a demo video for visual confirmation. All environment setups are handled via changes in `environment_params.yaml`, e.g., switching `WORLD_NAME` or modifying `OBSTACLES` and `BOT_CARS` fields.

Detailed Execution Steps

Phase I: Time-Trial

- Train on all tracks with `WORLD_NAME=reInvent2019_wide, reInvent2019_track, New_York_Track`.
- Evaluate with `python src.utils.evaluate(agent)`.
- Generate video using `python src.utils.demo(agent)`.

Phase II: Obstacle-Avoidance

- Update config with `OBSTACLES=6, BOT_CARS=0`.
- Repeat training, metrics, and video steps for all tracks.

Phase III: Head-to-Head

- Update config with `OBSTACLES=0, BOT_CARS=3`.
- Repeat training, metrics, and video steps for all tracks.

Each transition to the next step is gated by agent performance: it must complete 5 full laps in a row before moving on.

4 Set up

4.1 Phase I

This section details the design choices made in Part I of the project: from network architecture and action space to reward shaping and hyper-parameters. Everything is tuned for stability, interpretability, and sample efficiency in a simulation-first setup like AWS DeepRacer.

Action Space

We defined a custom discrete action space with 31 actions, each specifying a unique pair of steering angle and speed. Steering ranges from $-30°$ to $30°$, while speed increases symmetrically toward the center. This gives the agent fine-grained control near the center line (high speed) and slower speeds on sharper turns — encouraging faster straight-line motion but caution in curves.

Neural Network / Function Approximation

We used a shallow convolutional neural network with 3 Conv2D blocks, batch norm, and ReLU, followed by adaptive pooling to an 8×8 spatial resolution, which is enough for bot to actually see good representation of track. The output is flattened and passed to two heads — policy and value. This structure gives us a compact representation of the front-facing image without blowing up parameters. It's lightweight, generalizes well, and doesn't overfit to visual noise.

We also added a Dropout layer in the policy head and scaled the logits with temperature ($T = 1.5$) to encourage broader exploration — especially in early episodes.

RL Algorithm

We implemented PPO using a custom PyTorch agent with separate policy and value heads. Below we explain key design decisions made during implementation. **Custom CNN-based agent:** Our policy is parameterized by a convolutional neural network followed by fully connected layers. It takes in the unflattened front-facing camera image, processes it spatially, and outputs both action logits and state values. **Temperature = 1.5:** After computing logits, we divide them by 1.5 before sampling from the distribution. This softens the probabilities, reducing the gap between the most and least likely actions. In practice, it makes the agent explore more — instead of always picking the top action, it samples from a more uniform set. That's useful early in training or when switching between tracks. **Dropout(0.3):** We add dropout before the output layer of the policy head to prevent overfitting. Since visual input varies across tracks, we wanted the network to learn robust patterns, not memorize camera-specific noise. **Entropy coefficient = 0.5:** We set this deliberately high to encourage exploration. Entropy acts as a regularizer that favors broader action distributions. If entropy is too low, the agent gets overconfident and collapses to one mode (e.g., driving straight). With multiple tracks, we need the agent to stay flexible and exploratory longer. **Clip ratio = 0.1:** PPO updates are clipped to prevent large changes in policy. A lower clip value like 0.1 gives more conservative updates, which stabilizes learning — especially important in environments with sparse or high-variance rewards. **Value loss coefficient = 0.5:** Balances how much we care about value prediction error versus policy improvement. We use a moderate value here to let the critic learn accurately without dominating the loss. **Advantage normalization:** Advantages are implicitly normalized by the PPO surrogate loss. We didn't add extra normalization, as clipping already keeps the update size in check. **Gradient clipping (max_norm = 0.5):** Helps avoid exploding gradients in early training, especially when the CNN outputs unstable features.

These tweaks (temperature, entropy, dropout) were chosen to make PPO more robust in a multi-track, image-based setting — where exploration, generalization, and stability are critical.

Reward Function

Reward function was built based on the example published by Bahman Javadi on LinkedIn[1]. Reward is shaped to encourage the agent to stay close to the center line and to drive fast. We added reward bands based on distance from center (tighter = higher reward), and combined that with a speed bonus. This way, the agent learns that hugging the center while maintaining good speed is best.

If the car is slow (below threshold speed), we penalize slightly. If it's fast and on track — we boost the reward.

Hyper-parameters and Tuning

`seed = 42` — We fixed the seed for reproducibility. `total_timesteps = 1024` — We limited each episode's budget to stay memory-efficient and favor shorter, faster rollouts — ideal when training across multiple tracks. `batch_size = 256` — Large enough for good gradient estimates, small enough to keep updates frequent. We wanted the model to adapt quickly within each training loop. `n_epochs = 8` — This gives multiple passes over the same batch. PPO benefits from more reuse of experience, as long as the updates are clipped — which we do. `learning_rate = 2e-3` — Slightly higher than typical defaults ($\sim 1e$–4). We tested higher LR to encourage faster learning early on. PPO's clipping helps absorb instability. `gamma = 0.999` — A higher discount factor prioritizes long-term lap completion over short-term tricks. It nudges the agent to value track-level planning instead of greedy turns. `lam = 0.95` — Standard for GAE (Generalized Advantage Estimation). This balances bias and variance. Lower lambda makes advantages more reactive, but we preferred smoother learning. `entropy_coef = 0.5` — Intentionally increased to encourage more exploration. Since we switch tracks often, we didn't want the agent to converge too quickly on one strategy.

This setup favors quick generalization across multiple tracks, while keeping the model light, responsive, and interpretable. Most tweaks (entropy, dropout, temperature, gamma) were made to promote exploration and stability in a changing visual landscape.

4.2 Phase II

In Phase II we've used agent's checkpoint trained on A to Z Speedway (wide), all hyperparameters and set-ups remain the same, the only difference lies in reward function, which we updated with penalties or incentives specific to given Phase II problem.

Obstacles Avoidance

For Obstacle Avoidance, we introduced a strict penalty. If the car **collides** with any obstacle or goes **off the track**, it receives a minimal reward (close to zero). This signals a failed attempt and discourages reckless behavior. The intention is to teach the agent that any collision or off-track event leads to a failed episode, so it must prioritize clean, uninterrupted driving.

Head-to-Head

For the Head-to-Head setup, we added two simple reward rules. If the car **collides** with a bot vehicle, it receives a strong penalty. This discourages aggressive or careless driving. If the car **stays on the track and avoids any collisions**, it earns additional reward proportional to its progress. The further it goes safely, the

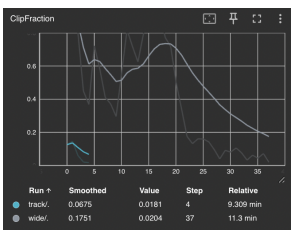
more it gains.

This encourages the agent to drive responsibly: avoid collisions and maintain clean forward motion to be rewarded.

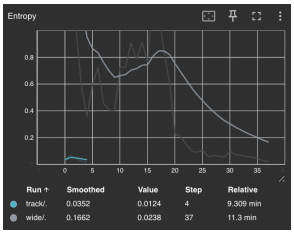
5 Training

5.1 Phase I

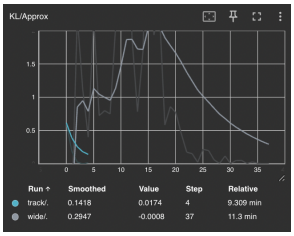
A to Z Speedway (wide) and Smile Speedway (track) - What Went Well?



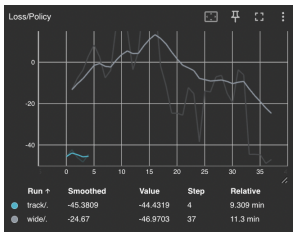
(a) Clip Fraction



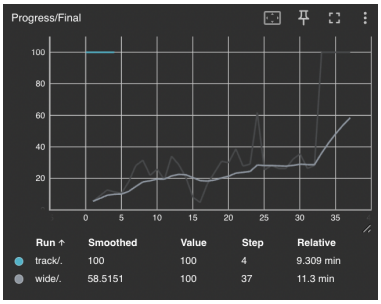
(b) Entropy



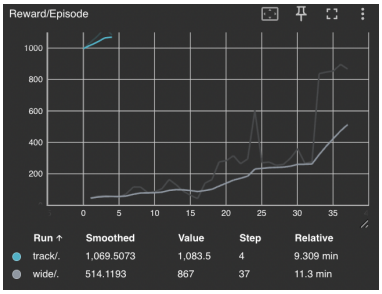
(c) KL



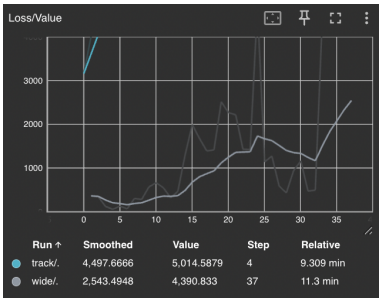
(d) Policy Loss



(e) Progress



(f) Rewards



(g) Value Loss

Figure 1: Comparison of results across A to Z Speedway (wide) and Smile Speedway (track)

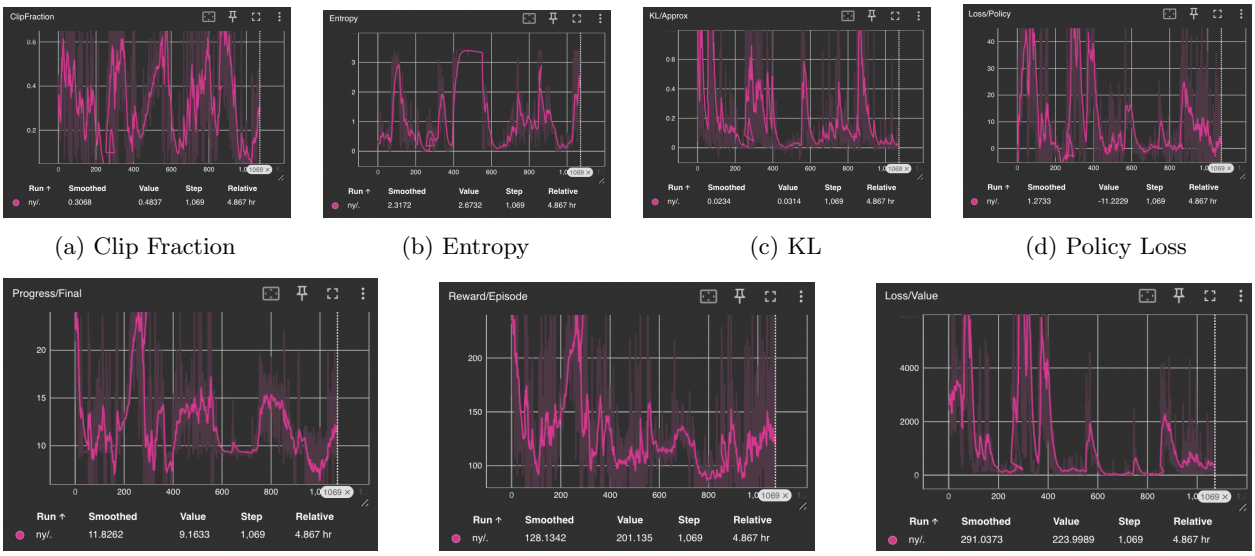
To sort things out, we've used 1 model which was trained primarly on A to Z Speedway (wide), then after 5 consecutive laps it was fine-tuned on Smile Speedway (track), then after 5 consecutive laps it was fine-tuned on Empire City Training (New York), after 5 consecutive laps on last track model would be considered as trained. However, unfortunately set up outlined in the report wasn't able to complete 5 consecutive laps on last track Empire City Training (New York). Let's try to understand what went correctly and what needs improvements.

Here is an interpretation of the TensorBoard plots comparing the PPO agent's behavior on A to Z Speedway (wide), (gray) and Smile Speedway (track), (blue). **Clip Fraction:** On the `wide` track, the clip fraction starts relatively high, indicating more frequent large policy updates early on. Over time, it stabilizes as the policy converges. On the `track` environment, the clip fraction is low and stable from the beginning, suggesting the policy required fewer adjustments, possibly due to prior familiarity or a simpler structure. **Entropy:** The entropy on `wide` begins higher and decays gradually, showing the agent was initially exploring a wide range of actions before settling into a policy. On `track`, entropy is low from the start, indicating the agent didn't

need to explore much because it likely already had a useful policy for this environment. **KL Divergence (KL/Approx):** The KL divergence on `wide` is higher at the beginning, reflecting significant changes to the policy, but then it drops as the updates become smaller. On `track`, KL is lower and stable, suggesting the policy remains close to its previous version, requiring minimal changes. **Policy Loss:** On `wide`, the policy loss decreases steadily, indicating ongoing policy improvements. On `track`, the loss is negative and relatively stable, supporting the idea that the policy was already effective and required less refinement. **Progress (Progress/Final):** On `wide`, the agent starts with lower progress and gradually improves, showing it is still learning. The agent on `track` immediately achieves 100% lap completion, indicating the task is already solved. **Rewards:** `Track` yields high and consistent rewards early on, confirming optimal behavior. `Wide` starts with lower cumulative rewards and sees a gradual increase, reflecting progressive learning. **Value Loss:** The value loss is more volatile on `wide`, indicating the agent's value function is still adapting. On `track`, value loss is smoother, implying a more stable estimation of state values.

Overall, the agent behaves as expected when transferring from `track` to `wide`. The `wide` track initially requires more exploration and adaptation, as evidenced by higher entropy and KL divergence, while the `track` shows efficient exploitation of existing knowledge.

Empire City Training (New York) - What Didn't Go Well?



(a) Clip Fraction (b) Entropy (c) KL (d) Policy Loss

(e) Progress (f) Rewards (g) Value Loss

Figure 2: Results of Empire City Training (New York)

We analyze PPO training performance for the New York Track using seven key metrics. Below are the insights derived from the TensorBoard logs. **Clip Fraction:** Initially oscillates between 0.2 and 0.6, indicating frequent clipping. This suggests the policy updates are often large. Over time, the clip fraction fluctuates but does not converge, implying instability or ongoing exploration. This may be due to the difficulty of adapting from a pretrained model to a new track. **Entropy:** High entropy spikes (up to 3.0) occur repeatedly across training, especially early on and around step 400–600. This indicates that the agent is consistently forced to explore, potentially due to struggling with generalization on the new track. Unlike reInvent2019_track, where entropy stayed low (as the agent was already confident), here the agent doesn't yet know how to act

effectively. **KL Divergence:** KL stays low (mostly up to 0.6) throughout training, despite high entropy. This suggests that while the agent samples actions broadly, it doesn't diverge far from the old policy. This might be a sign of ineffective learning or an overly cautious PPO update step. **Policy Loss:** Initially high and positive, with bursts above 40, before decreasing to around zero and turning slightly negative. This trend reflects unstable updates where the policy might be trying to aggressively change but is later regularized. The convergence remains erratic. **Progress:** Peaks early (23%) then steadily decreases and fluctuates below 15% — the agent is not improving. Compared to reInvent2019_wide where progress consistently increased, the New York agent stagnates. It may not have adapted well from prior training or the environment dynamics differ too much. **Reward per Episode:** Mirrors the progress chart — early peaks (above 200), followed by consistent drops and plateaus around 100–150. This suggests the agent exploits a suboptimal strategy but fails to generalize or optimize it further. **Value Loss:** Initially extremely high (up to 4000), it reduces and plateaus around 200–1000. The critic is struggling to estimate returns reliably, especially due to unstable or noisy reward signals from failed laps.

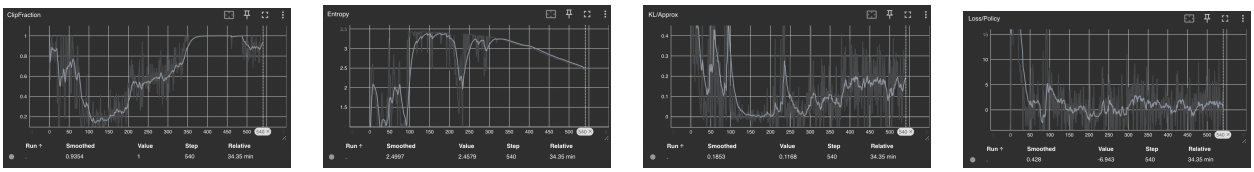
Overall, the model trained on `reInvent2019_wide` likely transferred poorly to New York Track. The most obvious reason is the difference between two first tracks and Empire City Training track surroundings. In third track we have tunel and buildings which distracts agent. To perform better in New York, we would have to adjust our CNN network to be more robust to spatial distractions and visual complexity, allowing it to focus on lane-related features rather than being misled by irrelevant background textures like tunnels or buildings. The high entropy and unstable progress suggest that more targeted fine-tuning or reward shaping is necessary to adapt. Importantly, the contrast with the reInvent2019_track shows how a pretrained agent may not need to explore as much when already familiar with the dynamics, whereas here exploration spikes repeatedly as the agent searches for an effective policy.

5.2 Phase II

In Phase II we've used agent's checkpoint trained on A to Z Speedway (wide), all hyperparameters and set-ups remain the same, the only difference lies in reward function, which we updated with penalties or incentives specific to given Phase II problem.

Obstacles Avoidance

I didn't have enough time to let this model train fully, but based on current plots it still shows promising signals. Here's what we noticed. **Entropy:** Starts high, drops steadily — which is good. It means the agent is exploring a wide range of actions at first, and gradually focusing on a consistent strategy. We didn't hit zero entropy, so the policy didn't collapse — that's a green flag. **Policy Loss:** The loss oscillates early on but stabilizes around zero. That usually means the policy is not being heavily updated anymore, and PPO clipping is doing its job. It could use more training to refine, but no obvious divergence is visible. **Value Loss:** Starts high, drops sharply, and flattens. That's expected — the critic is learning to predict future returns and gets better with more data. Again, the trend is good. **Clip Fraction:** Initially drops (as updates are large and frequent), then slowly climbs back up and stabilizes around 0.9–1.0 — suggesting clipping is engaged often. That's not unusual when the policy is updating cautiously. **Progress and Rewards:** The agent is getting consistent, though modest, rewards. Progress isn't impressive yet — under 11% — but stable. It shows the agent isn't failing outright; it just hasn't figured out a path past the first obstacles.

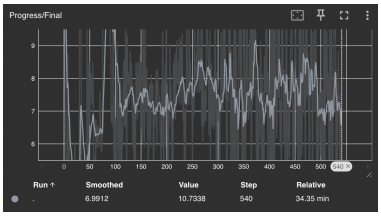


(a) Clip Fraction

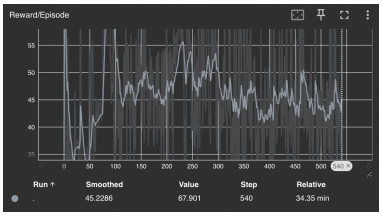
(b) Entropy

(c) KL

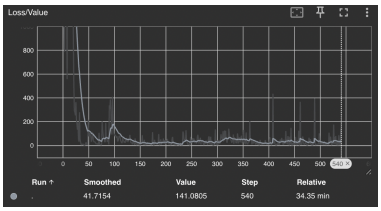
(d) Policy Loss



(e) Progress



(f) Rewards

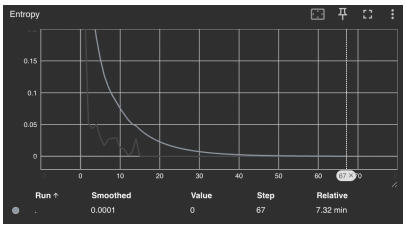


(g) Value Loss

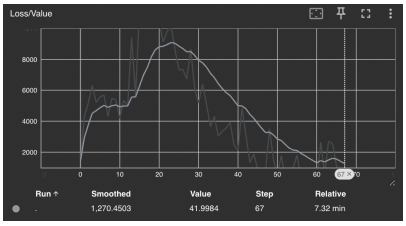
Figure 3: Results of Empire City Training (New York)

Conclusion: This model didn't have enough training to succeed yet — but it shows no signs of overfitting or collapse. With longer runs, better reward shaping and maybe small architecture tweaks, this one likely has a solid chance to learn obstacle avoidance effectively.

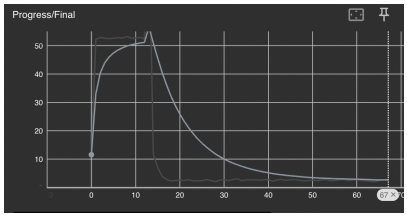
Head-to-Head



(a) Entropy



(b) Entropy



(c) KL

Figure 4: Training metrics for obstacle avoidance (Empire City)

During Head-to-Head training, we observed a complete collapse in exploration — entropy dropped to 0.0 early on and stayed there. This indicates that the policy became overconfident and deterministic too quickly. As a result, the agent failed to explore alternative behaviors, leading to poor progress and ineffective learning. We stopped training at this point since further updates would reinforce a stuck, suboptimal policy.

6 Evaluation

6.1 Phase I

We evaluated the final agent on all three tracks using saved models trained on `reInvent2019_wide` and `reInvent2019_track`. Results show that the agent generalizes well on these two — consistently achieving 100% progress and lap times improving across episodes. On `New_York_Track`, however, performance was poor: the agent failed to complete any lap and often got stuck early. This confirms that training on wide

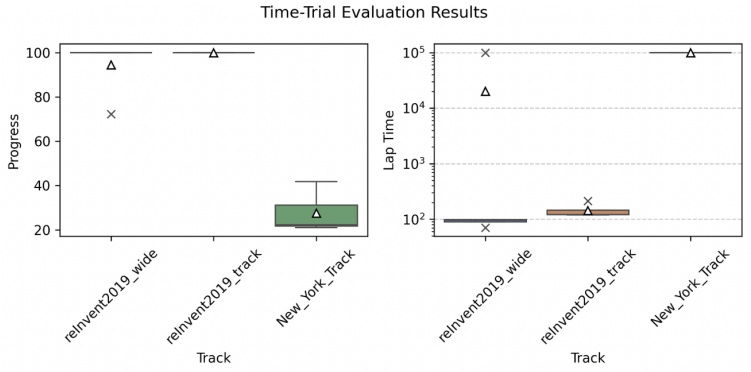


Figure 5: Phase I Evaluation

and curved layouts transfers decently between similar tracks but doesn't generalize to narrow, unseen ones without dedicated fine-tuning.

Lap times on the wide track varied slightly **(best: 70.5 steps which is 7.05 seconds)**, while on the standard track they improved steadily **(final: 119.6 steps which is 11.96 seconds)**. The New York layout likely requires explicit training due to its complexity and tighter turns.

6.2 Phase II

Since we didn't manage to train (or rather fine-tune) properly our agent for Obstacles Avoidance and Head-to-Head evaluation on those 2 set-ups doesn't make sense. Since that I've decided to don't overuse attention of the reader.

7 Conclusion

This project demonstrates the feasibility of training a custom PPO agent using raw front-facing camera input for autonomous driving tasks in AWS DeepRacer. Our architecture, centered around a lightweight CNN and stabilized PPO loss, proved effective on A to Z Speedway and Smile Speedway.

While the agent generalized well between similar tracks, Empire City (New York) revealed limitations in visual generalization, suggesting the need for more robust architectures or data augmentation.

Phase II experiments on obstacle and bot scenarios showed early promise, but limited training time and entropy collapse hindered full convergence. Future work could improve generalization through curriculum learning, attention-based models, or multi-agent coordination strategies.

Despite not completing all tasks, the agent showed solid performance and provides a strong base for further research and development in sim-to-real driving agents.

References

[1] Bahman Javadi. *Sample Reward Functions for AWS DeepRacer*. LinkedIn, 2020. https://www.linkedin.com/pulse/samples-reward-functions-aws-deepracer-bahman-javadi