Wild PPO TuxKart Driver

Huaiyuan Ma
BESE
KAUST
huaiyuan.ma@kaust.edu.sa

Abstract

This work details the development and training of a Reinforcement Learning (RL) agent to play the game "SuperTuxKart" using Proximal Policy Optimization (PPO). The agent employs an actor-critic framework with a pre-trained ResNet-18 backbone for visual feature extraction and an aim-point prediction mechanism for policy generation. The study investigates the impact of pre-training different components of the model, the design of a specialized reward function, and a custom data rollout method. Key findings demonstrate successful agent training, the ability to generalize to unseen tracks, and the critical importance of an aim-point-based reward for effective navigation. Comparisons are drawn among agents with pre-training, without pre-training and a baseline planner.

1 Introduction

1.1 RL

Reinforcement Learning (RL) has significantly impacted the field of game playing, with a landmark moment being the 'Playing Atari with Deep Reinforcement Learning' paper by Mnih et al. (2013) from DeepMind Volodymyr Mnih et al. [2013]. This pioneering work demonstrated that an RL agent, specifically a deep Q-network (DQN), could learn to play a range of Atari 2600 games directly from raw pixel inputs, without any prior knowledge of the game rules Volodymyr Mnih et al. [2013]. The DQN model, a convolutional neural network, was trained using a variant of Q-learning and an experience replay mechanism to stabilize learning. This approach achieved human-level or even superhuman performance on several of the tested Atari games, marking a significant breakthrough in applying deep learning to RL for complex control tasks.

Building on such foundational work in RL, Proximal Policy Optimization (PPO) has emerged as a highly effective and widely used RL algorithm in game playing and other domains. PPO is a policy gradient method that aims to train an agent's policy—its strategy for making decisions—in a stable and efficient manner Schulman et al. [2017]. Unlike some earlier methods that could suffer from overly large policy updates leading to performance collapse, PPO introduces a "clipping" mechanism in its objective function. This clipping restricts the extent to which the new policy can deviate from the old one, ensuring smoother and more reliable learning. PPO is recognized for striking a favorable balance between sample complexity, simplicity of implementation, and wall-clock time, making it a popular choice for training agents in complex environments, including Atari games and simulated robotic control. Its design allows for multiple epochs of minibatch updates on the same batch of collected experiences, improving sample efficiency without causing destructive updates. This combination of stability, performance, and relative ease of use has contributed to PPO's widespread adoption in both research and practical applications of game AI Schulman et al. [2017].

In this work, we will use an algorithm with architecture based on PPO learning to learn to play a game called "Super Tux Kart". We will demonstrate how we build this network, how we train and test this network, and the detours we have along the way.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

1.2 Environment

SuperTuxKart is a 3D open-source arcade racing game designed to be fun for all ages, emphasizing enjoyable gameplay over realistic physics. It features a diverse cast of characters, many of whom are mascots from various open-source projects, a wide array of tracks, and multiple game modes.

For training neural networks, a significantly modified version of SuperTuxKart, its python wrapper PySTK, is utilized. This version is specifically adapted for sensorimotor control experiments. Many of the elements that contribute to SuperTuxKart's entertainment value, such as sound, networking for general multiplayer, and the standard user interface, have been removed. Instead, PySTK offers a highly efficient Python interface to the game's rendering engine and assets. This allows for a controlled and customized setup for RL.

2 RL setup

2.1 Model

In this paper, we utilize a deep neural network for predicting value and action. Our model employs a actor-critic framework, designed for end-to-end training with Proximal Policy Optimization (PPO). It decomposes the decision-making process into visual feature extraction, a aim-point prediction for policy generation, and state value estimation.

The perceptual foundation of PPOPlanner is the ImageDecoder. This module utilizes a pre-trained ResNet-18 backbone (resnet18.fb_swsl_ig1b_ft_in1k) to transform raw image inputs into rich, 512-channel feature maps. These features are spatially interpolated to a fixed resolution (24x32) and further refined by a convolutional layer that reduces the dimension of the heatmap to (12x16), providing a robust and consistent representation for downstream processing. This leveraging of pre-trained visual knowledge is crucial for efficient learning in visually complex environments.

The policy, or actor, component is realized through our AimPointPredictor. Taking the features from the ImageDecoder, this deep convolutional network generates a spatial heatmap. This heatmap represents the desirability of potential "aim points," guiding the agent's actions by identifying salient regions rather than directly outputting low-level controls. The form of that heatmap is shown as in image 1. Concurrently, a value_head (critic) processes the same shared visual features to produce a scalar estimate of the current state's value. This critic network also employs a series of convolutional layers, incorporating 2D Layer Normalization for training stability, and culminates in an adaptive average pooling and linear layer to output the value. The integrated PPOPlanner thus learns to perceive, decide on intermediate goals, and evaluate states for effective visual control.

The reason why we use LayerNorm instead of batchnorm in the critic head is that while Batch Normalization (BN) is a common technique for accelerating training and improving generalization in deep networks, we observed a critical issue related to its behavior when transitioning from training to single-instance trajectory inference.

Batch Normalization operates by normalizing the activations within a mini-batch during training, using the mean and variance of that specific mini-batch. For inference, BN typically switches to using population statistics—running averages of mean and variance accumulated across many training batches. However, a significant discrepancy arises in reinforcement learning scenarios where, during trajectory generation or deployment, the model often processes a single image (a batch size of one) at each time step. In this single-instance inference mode, the concept of a "mini-batch" mean and variance is ill-defined. The network must rely solely on the aggregated population statistics. If the distribution of activations for single inference instances significantly diverges from the average distributions encountered during training with larger mini-batches, the normalization applied by BN can be suboptimal or even detrimental, leading to a performance gap between training and inference and potentially causing training instability as the agent learns with one set of normalization statistics but acts with another.

This observed sensitivity of Batch Normalization to differing batch sizes between training and inference, particularly the extreme case of single-image inference, motivated a careful consideration of normalization strategies within our PPOPlanner. While our AimPointPredictor (which uses nn.BatchNorm2d) might still be susceptible if its pre-trained weights relied heavily on specific batch statistics, for the value_head which is trained concurrently with the PPO algorithm, we

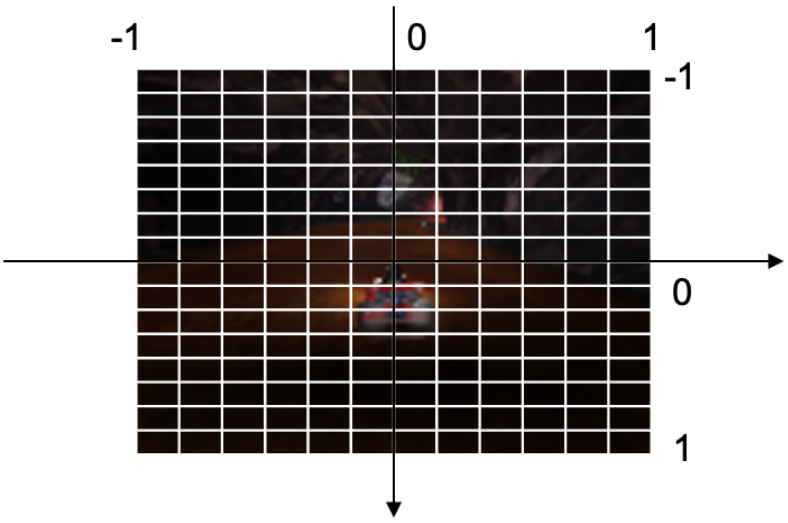

Figure 1: An image is treated as a heatmap, each coordinate is represented by a element in the heatmap produced by AimpointPredictor.

opted for 2D Layer Normalization (LayerNorm2d). Layer Normalization, by contrast, normalizes activations across the channel dimension for each individual data sample (and each spatial location independently), making its behavior consistent regardless of the batch size. This choice aims to mitigate the distribution shift issues associated with Batch Normalization during single-image inference, contributing to more stable learning and more consistent performance between training and deployment phases of our visual control agent.

2.2 Rollout Method

Although an initial rollout method was provided, to effectively implement and train our PPO agent, we recognized the need for a more specialized data collection mechanism that could capture the specific types of information required by the PPO algorithm. While the provided rollout function was capable of general simulation and potentially for supervised data collection via its data_callback, it didn't directly provide all the components needed for an actor-critic RL algorithm like PPO. Therefore, building upon the principles and structure of the provided rollout function, we developed our own ppo_rollout method with several key enhancements tailored for PPO.

Our primary goal in implementing ppo_rollout was to create a seamless data pipeline for our PPO training loop. A crucial modification we introduced was the direct integration of our PPO policy (our PPOPlanner model) into the data collection process. In each step of the simulation, our ppo_rollout function queries this policy with the current game image. Unlike the original rollout which might focus on just obtaining an aim point, our version retrieves not only the action (which, for our heatmap-based policy, is the selected pixel index representing the aim point) but also two other critical pieces of information: the log probability of that chosen action and the state value estimate (V(s)) generated by our policy's critic component.

To manage these PPO-specific data elements, we incorporated dedicated lists within ppo_rollout to store the sequence of actions (as pixel indices), their corresponding log probabilities, and the state values estimated at each timestep, alongside the observations, rewards, and done flags. We also designed our ppo_rollout to accept a custom reward_fn as an argument. This provided us with the flexibility to experiment with different reward shaping strategies to guide our agent's learning, a common practice in reinforcement learning. Furthermore, recognizing the importance of Generalized Advantage Estimation (GAE) in PPO, especially for episodes that are truncated (i.e., end due to reaching max_frames rather than task completion), we implemented logic to calculate a final_bootstrap_value. This involves taking the very last observed state of a truncated episode and obtaining its value estimate from our policy, which is then used in the GAE calculation.

Finally, the output of our ppo_rollout function was structured as a dictionary containing all these collected trajectories as NumPy arrays (obs, actions, log_probs, rewards, values, dones), along with the last_value (the bootstrap value). This format is designed to directly feed into our PPO agent's training buffer, making the data collection and learning process more efficient and organized. These enhancements to the data collection process were instrumental in enabling us to effectively train our PPO agent for the SuperTuxKart navigation task.

2.3 Reward Function

This reward function meticulously shapes the agent's behavior to encourage efficient and successful race completion by combining several weighted incentives and penalties. At its core, it utilizes a helper function, abs_reward(loc_diff, center), which is instrumental in guiding both aiming and positioning. This helper function penalizes deviations (loc_diff) that exceed a specified center threshold and rewards deviations that are below it, effectively creating a target zone; the reward is typically maximized when loc_diff (representing deviation from an ideal state, like the track's true center line) is zero.

The most dominant incentive in the main calculate_reward function is a substantial reward for progress_diff, scaled by a factor of 80, strongly pushing the agent to make consistent forward progress. To ensure the agent doesn't become too cautious, a penalty is applied if its velocity falls below 20% of a predefined target_velocity, discouraging overly slow movement. A small, constant penalty of -0.02 is applied at every step the episode is not done, subtly encouraging quicker race completion. More significant penalties are reserved for undesirable events: triggering a rescue action incurs a hefty -8 penalty, strongly discouraging situations where the agent gets irretrievably stuck or crashes.

Crucial for effective navigation, the agent's steering is guided by an aim point mechanism. Using the abs_reward function with aim_point_diff (the deviation of the kart's projected aim point from the track's center line 15m ahead) and a center value of 0.4, the system rewards aiming close to the true track center (achieving a maximum of +0.8 for perfect central aim) while penalizing aiming further than 0.4 units off this line. Similarly, the kart's physical position on the track is managed through location_diff (the kart's current lateral deviation from the track center). This also employs abs_reward with a center of 5.0, strongly rewarding the agent for staying near the track's center (up to +5 for being perfectly centered) and penalizing it for deviations greater than 5 units.

Finally, the episode's conclusion is marked by large terminal rewards: a +30 reward if done (signifying a successful race completion) and a -30 penalty if finished is true separately (likely indicating a timeout or other unsuccessful end to the episode). To maintain stability during training, the total calculated reward for any given step is clipped to a range between -30 and +30.

2.4 PPO Algorithm

When we set out to train our agent for SuperTuxKart, we opted for Proximal Policy Optimization (PPO) due to its reputation for stable and sample-efficient learning in complex environments. Our implementation focused on robustly estimating how good each action was and then carefully updating our agent's policy based on that information, an example structure of PPO is shown in figure 2.

The first critical piece we implemented was a reliable way to evaluate actions. For this, we chose Generalized Advantage Estimation (GAE), as detailed in our compute_gae function. We knew that simply using one-step TD errors could be noisy. So, for each trajectory collected, our process began by calculating the TD error, δ_t, at every timestep t. This was done using the formula

$$\delta_t = r_t + \gamma V(s_{t+1})(1 - d_t) - V(s_t) \tag{1}$$

where r_t is the observed reward, $V(s_t)$ is our critic's estimate of the state value, γ is our discount factor, and d_t handles episode termination. If an episode was truncated rather than naturally ending, $V(s_{t+1})$ for the last step was replaced by a bootstrapped value from our critic for that final observed state (last_value). With these TD errors in hand, we then worked backward from the end of each trajectory to compute the GAE advantage, $\hat{A}_t^{GAE}$, for each step. This was calculated recursively as

$$\hat{A}_t^{GAE} = \delta_t + \gamma\lambda(1 - d_t)\hat{A}_{t+1}^{GAE} \tag{2}$$

(where $\hat{A}_T^{GAE} = \delta_T$), which is equivalent to the sum $\sum_{k=0}^{T-t-1}(\gamma\lambda)^k\delta_{t+k}$. The λ parameter here allowed us to balance between lower-bias (higher λ) and lower-variance (lower λ) advantage estimates.

Finally, the targets for updating our value function, the returns R_t, were simply these GAE advantages added back to the original value estimates: $R_t = \hat{A}_t^{GAE} + V(s_t)$.

With these advantages and returns computed, we moved on to the core PPO update logic within our PPO class's update method. Before anything else, to help stabilize the learning process, we normalized the batch of GAE advantages, $\hat{A}_t$, to have zero mean and unit standard deviation. A central quantity in PPO is the probability ratio,

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3}$$

, which compares the likelihood of an action under the current policy π_θ versus the policy $\pi_{\theta_{old}}$ that generated the data. We calculated this in log space as $r_t(\theta) = \exp(\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{old}}(a_t|s_t))$ to maintain numerical stability.

The innovation of PPO lies in its clipped surrogate objective, which we designed our policy loss around. To ensure our policy updates weren't too drastic, we implemented the clipped objective

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{4}$$

Our actual policy_loss variable was the negative of this, as our optimizer performs minimization: policy_loss = torch.max(-ratio * adv_squeezed, -torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * adv_squeezed).mean(). This clipping, controlled by clip_param (ϵ), effectively creates a pessimistic lower bound on the policy improvement and prevents the ratio $r_t(\theta)$ from pushing the policy too far, too quickly.

Alongside the policy update, we trained our value function (the critic). Its loss,

$$L^{VF}(\theta) = \hat{\mathbb{E}}t \left[(V\theta(s_t) - R_t)^2 \right] \tag{5}$$

was a straightforward mean squared error between the critic's current value predictions $V_\theta(s_t)$ and the target returns R_t we computed earlier using GAE. To encourage our agent to explore diverse actions and avoid premature convergence to a suboptimal deterministic policy, we also incorporated an entropy bonus, $S(\pi_\theta(\cdot|s_t))$. For our categorical policy (sampling pixels from a heatmap), this was the mean entropy of the predictive distribution.

These individual loss components were then combined into a single objective for our optimizer. The final loss we minimized was

$$L_{PPO}(\theta) = -L^{CLIP} + c_1 L^{VF}(\theta) - c_2 S(\pi_\theta(s_t)) \tag{6}$$

where policy_loss is our negated clipped surrogate objective, c_1 is value_coef, and c_2 is entropy_coef. To make efficient use of the collected experience, we performed this update over several epochs for each batch of trajectory data, shuffling the data and processing it in mini-batches. To further guard against destructively large gradients, we applied gradient clipping by norm using nn.utils.clip_grad_norm from pytorch. As a final stability measure, we monitored an approximation of the KL divergence between the policy before and after the mini-batch update:

$$\hat{KL} \approx \frac{1}{2}\hat{\mathbb{E}}_t[(\log r_t(\theta))^2] \tag{7}$$

If this $\hat{KL}$ value exceeded our target_kl_heuristic, we would break out of the update loops for that particular data batch early, preventing the policy from deviating too much from the behavior that generated the successful experiences. This iterative process of advantage estimation and carefully constrained policy updates allowed our agent to learn effectively.

3 Train Setup

Our training process for the PPO agent is orchestrated by the train function in ppo_train.py. The process begins by setting up the computational device and enabling anomaly detection in PyTorch for debugging.

First, we initialize our PPOPlanner model, which contains the actor and critic networks, and move it to the selected device. An important step here is to set any Batch Normalization layers within our

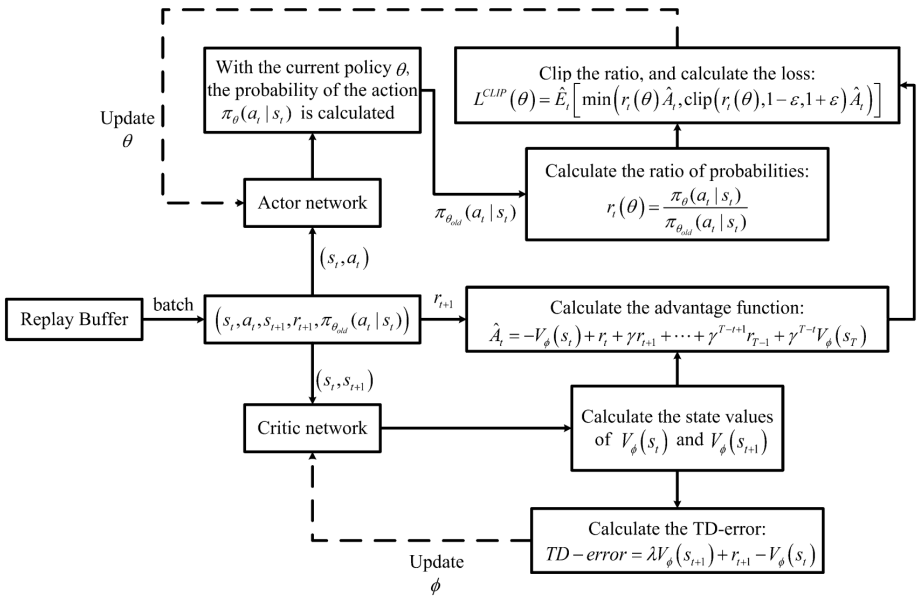

Figure 2: An example PPO architecture Shi et al. [2021]

policy to evaluation mode. This is crucial because during rollouts, data is often collected with a batch size of 1 per step, which can destabilize Batch Norm's running statistics. If the –continue_training flag is set, we attempt to load a previously saved model state (ppo_planner.th) to resume training.

Next, we set up the Adam optimizer for our policy's parameters, using the specified learning rate of 1e-5 commonly used for the Adam optimizer. Following this, our PPO agent is instantiated. This agent class encapsulates the core PPO update logic, taking the policy, optimizer, and various PPO-specific hyperparameters like clip_param, value_coef, entropy_coef, max_grad_norm, and target_kl (which are taken from the args parsed from command-line arguments or their defaults).

A TensorBoard logger (train_logger) is set up, allowing us to monitor various metrics during training. This is essential to RL training, as the progress and the change of various variables in the agent can be utilized as an indication of success and help us to improve our network and its hyperparameters. We then initialize our PPODataCollector. This class, as detailed in ppo_pytux_adapter.py, is responsible for interacting with the SuperTuxKart environment. The main training then enters a loop that runs for a total of 100 epochs (as per your -n 100 argument). Within each epoch, we iterate through a predefined list of game tracks (e.g., 'zengarden', 'lighthouse', etc., specified by args.tracks). Notice that during training we have not included cocoa_temple, for we also want to see if

4 Training Process

For training, we roughly distinguish it into 2 groups; the critic of the first group is pretrained aiming at the track center 15m ahead, and the second group is RL only (except for the ImageDecoder that is pretrained on the Instagram dataset). We want to compare the performance of these two groups and to observe their ways of converging to show how pretraining might affect RL training.

4.1 Pretrained Group

The training process appears to go through distinct phases, reflecting the adaptation of the pre-trained components to the reinforcement learning objective and the subsequent learning and refinement of the control policy. The plot from this group is shown as in figure 3.

Initial Adaptation and Rapid Learning Phase (Approx. 0 - 100k episodes): Entropy: Starts very high (around 4.6) and decreases sharply until about 75k-100k inferences. This indicates that initially, the policy is exploring broadly, with actions being quite random or uniformly distributed over the

possible aim points. The pre-trained AimPointPredictor, while providing a starting point, might not immediately translate to an optimal stochastic policy for PPO, hence the high initial entropy. The rapid decrease suggests the agent is quickly learning some fundamental aspects of the task or that the pre-trained features from ImageDecoder are immediately useful, allowing the policy to quickly become less random and more decisive as it identifies initially rewarding actions or patterns.

Policy Loss: Is highly volatile during this phase, oscillating significantly around zero. This is expected as the policy is making large adjustments based on the initial experiences and advantages calculated. The pre-trained policy head is being reshaped by the RL objective.

Value Loss: Starts relatively low (around 20-22) but increases dramatically to a peak around 100k-150k inferences (reaching values of 40-43). This is a key indicator. The pre-trained ImageDecoder features, while useful for the supervised task, feed into a value head that now needs to learn to predict the expected cumulative reward (returns) specific to the PPO task and its reward function. The initial low value loss might reflect some correlation between the pre-training objective (e.g., being near the track center) and some components of the PPO reward, but as the agent explores and experiences the full PPO reward signal and GAE-computed returns, the critic realizes its initial predictions are insufficient for this new, more complex objective, leading to a necessary increase in loss as it tries to fit these new targets.

Exploitation, Refinement, and Renewed Exploration Phase (Approx. 100k - 300k episodes): Entropy: After the initial sharp drop, the entropy reaches a minimum around 150k-200k inferences (around 2.5-2.7). This suggests a period where the policy becomes more deterministic, exploiting the strategies it has learned. Interestingly, after this minimum, the entropy begins to trend upwards again, albeit with large oscillations, settling into a range of roughly 3.0-3.5. This subsequent increase could signify that the agent is discovering more nuanced strategies that require a bit more stochasticity, or the entropy bonus in the PPO objective is effectively preventing the policy from collapsing and encouraging continued exploration to escape local optima.

Policy Loss: Continues to be highly oscillatory around zero, which is typical for PPO as it continuously refines the policy based on batches of new experiences. The magnitude remains relatively small.

Value Loss: Remains very high and volatile throughout this phase, though it might start a very slow, almost imperceptible decline towards the end of this period. The critic is still working hard to predict the returns from an evolving policy that is itself exploring and changing. The high variance in returns from a policy that is still learning and exploring makes the critic's job difficult.

Convergence and Stabilization Phase (Approx. 300k - 450k episodes): Entropy: Seems to stabilize in a range of 3.0-3.5, still with considerable variance, suggesting a balance has been struck between exploiting known good actions and maintaining some level of exploration.

Policy Loss: Remains oscillatory as expected.

Value Loss: Shows a more distinct and consistent downward trend in this phase, decreasing from highs of around 35-40 down to around 20-25. This is a positive sign, indicating that the critic is becoming significantly better at predicting the expected returns. This often happens when the policy itself starts to stabilize and perform more consistently, leading to more predictable return patterns for the critic to learn. The value loss returning to levels similar to or even below its initial values suggests the critic has now successfully adapted to predicting the PPO-specific returns.

4.2 Pure RL Group

This group shows a learning trajectory characteristic of an agent learning a complex task without task-specific pre-training, shown in figure 4.

Extended Initial Exploration & Critic's Initial Learning (Approx. 0 - 250k steps): Entropy: Starts very high (around 5.2, which is typical for a near-uniform distribution over many actions) and remains very high for an extended period, only beginning a noticeable decline around 200k-250k steps. This prolonged high entropy phase indicates that, with a randomly initialized policy head, the agent spends a significant amount of time exploring the action space very broadly. The Instagram

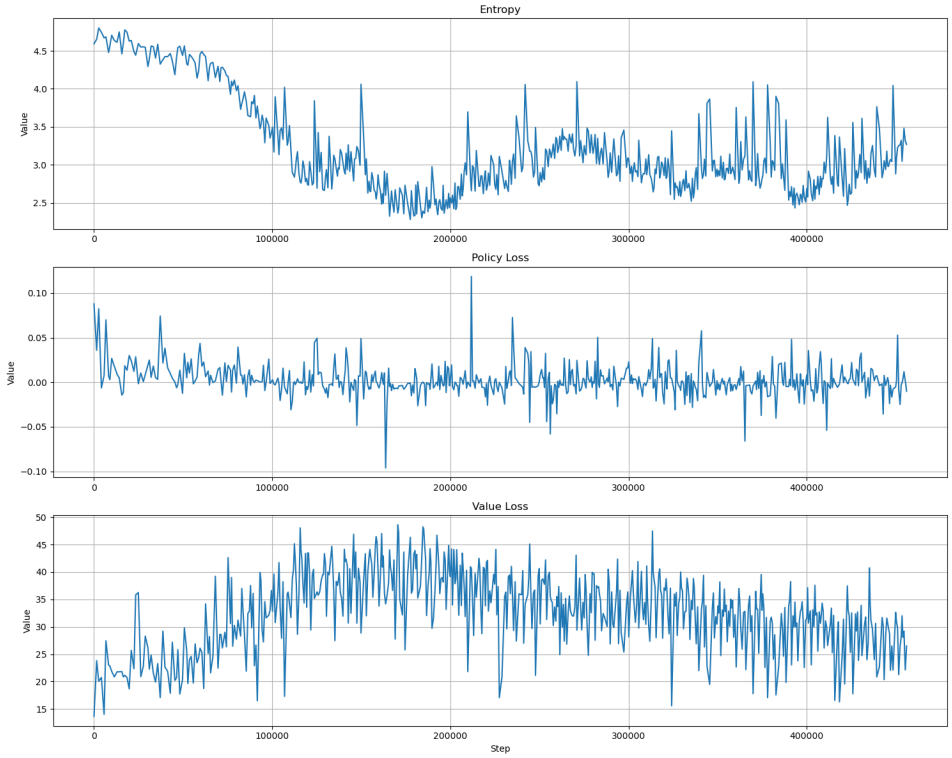

Figure 3: Entropy, policy loss, and value loss plotted against number of episode, logged from pretrained group.

pre-training on the ImageDecoder provides good general visual features, but the policy itself has no initial bias towards useful actions for the racing task and must discover them through trial and error.

Policy Loss: Remains very close to zero with minimal oscillations for the first 100k steps. This suggests that during the very early, highly random exploration phase, the calculated advantages are likely small, noisy, or average out, leading to minimal updates to the policy. As the agent starts to stumble upon slightly more structured experiences (beyond 100k steps), the policy loss begins to show more activity and oscillations.

Value Loss: Starts relatively low (around 5-15) for the first 50k steps. With a randomly initialized value head and a policy acting randomly, the initial returns are also likely very noisy and lack strong patterns. The critic's initial low loss might reflect it predicting values close to the mean of these noisy returns. However, from around 50k steps onwards, the value loss begins a steady and significant climb, peaking around 250k-300k steps (reaching values of 35-40). This mirrors the previous scenario in that the randomly initialized value head must learn to predict the actual, task-specific cumulative rewards. As the policy, even through random exploration, starts to generate trajectories with more discernible differences in outcomes, the critic's initial predictions are found to be poor, causing the loss to rise as it attempts to fit these emerging patterns of returns.

Active Policy Learning & Peak Critic Effort (Approx. 250k - 450k steps): Entropy: Shows a sharp and significant decline from around 250k steps, reaching a minimum around 450k steps (dropping to values as low as 1.5-2.0). This is the core phase where the policy rapidly learns and becomes much more decisive. The agent has gathered enough experience to identify actions and sequences that lead to better outcomes according to the reward function. The minimum entropy reached is notably lower than in the previous scenario, suggesting the policy might have become more deterministic or 'peaked' at this stage.

Policy Loss: Becomes highly active and oscillatory during this period, with larger swings. This aligns with the rapid changes in the policy (decreasing entropy) as the agent exploits the learned information and makes significant updates based on the advantage function.

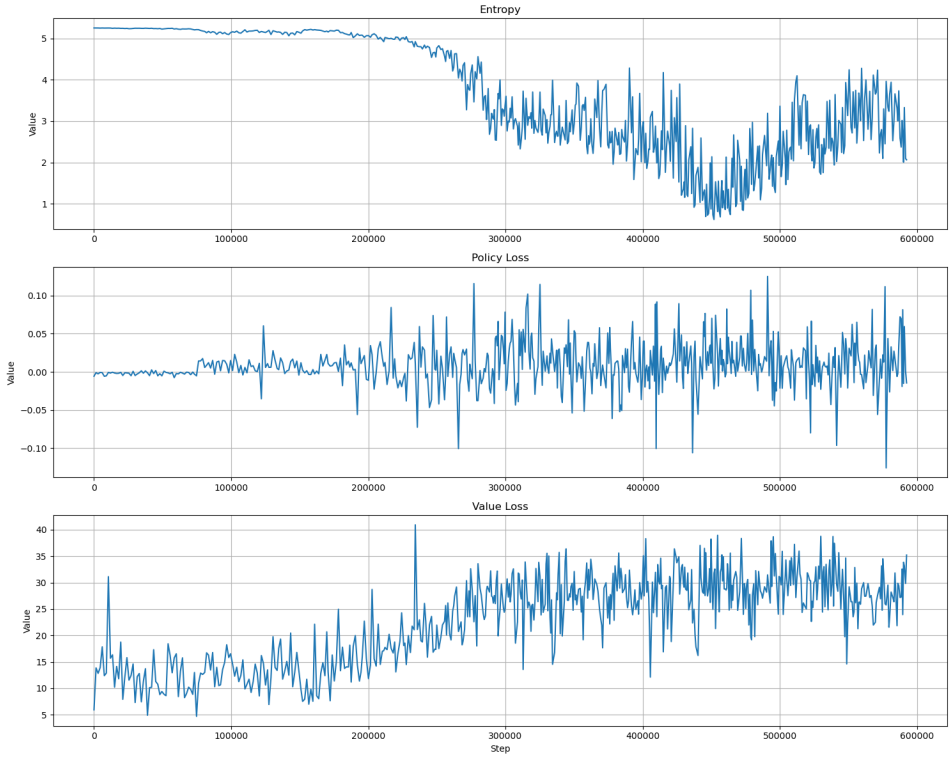

Figure 4: Entropy, policy loss, and value loss plotted against number of episode, logged from pure RL group.

Value Loss: Remains very high and highly volatile throughout this phase, hovering near its peak. The critic is working intensely to predict the values of states under a policy that is itself changing rapidly and dramatically. The target for the value function is constantly shifting as the policy improves, making the critic's learning task very challenging.

Later Stage Refinement and Stabilization Attempts (Approx. 450k - 600k steps): Entropy: After reaching its minimum, the entropy shows a tendency to increase again slightly and then appears to try and stabilize, oscillating in a range perhaps between 2.0 and 3.5. This could indicate the policy is exploring again (possibly due to the entropy bonus or escaping a local optimum of being too deterministic) or is settling into a more complex stochastic strategy after the initial sharp exploitation phase.

Policy Loss: Continues to be active and oscillatory, reflecting ongoing adjustments to the policy.

Value Loss: Remains high and very volatile, without a clear, sustained downward trend within this 600k step window. While it might not be increasing further, it hasn't shown the significant reduction seen in the previous scenario where the policy head also had pre-training. This suggests that learning an accurate value function from scratch (even with good features from ImageDecoder) is a prolonged process, and the critic is still actively trying to adapt to the policy's behavior and the environment's reward structure.

4.3 Train Process Summary

We analyzed two sets of TensorBoard training plots. The first set, where the actor's feature extractor and policy head had some pre-training, showed a rapid initial decrease in entropy and policy adaptation, but a significant period where the critic had to re-learn, evidenced by a surge and subsequent recovery in value loss. The second set, representing a "pure RL" approach with only the feature extractor pre-trained on the Instagram dataset, exhibited a much longer initial phase of random

exploration (sustained high entropy), a delayed onset of policy learning, and a value function that struggled more and for longer to accurately predict returns within the observed training window.

4.4 Test Result

The results are shown in table 1, with the raw planner being the fastest, followed by the pretrained and pure RL groups. We also added a break penalty (penalizing the agent for breaking when it's already slow), and removed the aimpoint reward to see how they perform.

Generally, the results offer significant insights into the agents' capabilities. A key finding is the generalization ability of the learning-based agents: both the "Pretrained" (80.6s) and "Pure RL" (91.6s) configurations successfully completed the Cocoa Temple track without prior training on it. This demonstrates their capacity to adapt learned visual features and policies to entirely novel environments. Notably, the "Pretrained" agent, with its supervised pre-training for the policy head, achieved a faster completion time on this unseen track, suggesting that such pre-training can provide a beneficial inductive bias for quicker adaptation in zero-shot scenarios.

When compared to the "Raw Planner" baseline, the learning-based agents ("Pretrained," "Pure RL," and "W/ Break Penalty") generally approached its ability to complete tracks but typically did so with a higher time cost. The "Raw Planner," when successful, was often the fastest. This indicates that while the RL and pre-trained agents learn robust and generalizable navigation strategies, they may not always match the raw speed of a more specialized or direct planner on familiar terrain. Their strength lies more in their adaptability and broader applicability, including to unseen tracks.

Finally, and critically, the experiments underscore the importance of incorporating the aimpoint into the reward structure. The "W/O Aim Point" configuration's widespread failure, resulting in an inability to complete most tracks, starkly illustrates that this reward signal is fundamental. It provides essential guidance for the learning process, enabling the agent to develop coherent and successful driving policies. Without it, the agent struggles to understand the task and navigate effectively, regardless of other factors like feature pre-training or other reward components.

Table 1: Agent Performance: Time Cost (seconds)

	Zengarden (s)	Lighthouse (s)	Hacienda (s)	Snowtuxpeak (s)	Cornfield (s)	Scotland (s)	Cocoa Temple (s)
Raw Planner	40.8	54.2	58.1	53.4	66.1	63.0	N/A
Pretrained	45.5	46.6	58.5	55.8	72.8	63.2	80.6
Pure RL	46.9	47.4	64.4	69.1	72.7	62.9	91.6
W/ Break Penalty	59.7	47.4	58.5	85.5	75.7	68.4	N/A
W/O Aim Point	Failed	83.9	Failed	Failed	Failed	Failed	N/A

5 Log of Mistakes

5.1 False Dimension

In a torch tensor, adding two vectors with different shapes (e.g. (1000,) and (1000,1)) will be parsed as a "dot add," which will result in a matmul-like process and return another matrix, with the same dimension as their dot product. Basically, treating them as a row vector and a column vector (which is mathematically correct, but different from the handling of the same two arrays in numpy). The error occurred at the GAE return computation step, resulting in a return of size (1000, 1000) for initially I trained on a trajectory length of 1000. This makes the value loss almost completely meaningless because all it does is add the first advantage to every value predicted before, and the model never converges.

5.2 BatchNorm

As I have also mentioned in the previous sector, namely the model part, using Batchnorm in a scenario that requires 2 different batch sizes would cause some shift in distribution because it relies on accurate

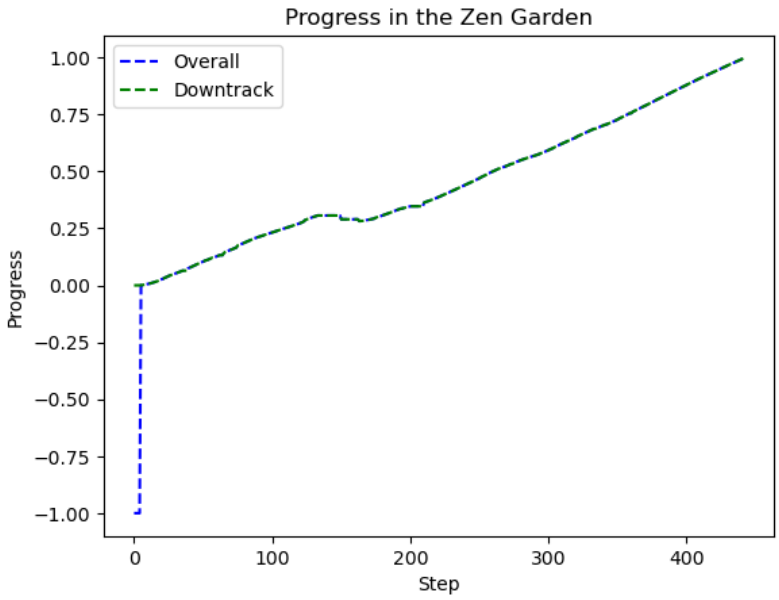

Figure 5: Progress provide by kart.overall_distance and kart.distance_down_track.

batch mean and batch variance for accurate normalization. This is particularly true for PPO, which computes KL divergence between trajectory inference output and batched agent update, resulting in early stopping in every first minibatch. This limits the number of updates and makes the training process very unstable. We finally solve this by replacing Batchnorm with Layernorm in the Value head and turning Batchnorm in the Policy head and Image Decoder to evaluation mode.

5.3 kart.overall_distance or kart.distance_down_track

This is not particularly a mistake, but more of an improvement of the original rollout method. Originally, the progress was monitored with kart.overall_distance. However, when logging the change of kart.overall_distance and kart.distance_down_track, I found out that kart.distance_down_track takes a distance of negative length of the whole track. For example, shown in figure 5, the kart.overall_distance starts at -1.0, which is obviously incorrect. This makes the kart start with a very high reward when rewarding by the change of progress between the start and end of an action like we do in our rewarding function. This can also make the training unstable.

6 Discussion

The results presented in this report demonstrate the successful application of PPO for training an agent to navigate complex tracks in pytux. A key finding is the generalization capability of the learning-based agents, particularly their ability to complete the Cocoa Temple track without prior training on it. This highlights the robustness of the learned visual features and policies.

The comparison between the "Pretrained" group (with a pre-trained policy head) and the "Pure RL" group revealed interesting trade-offs. While the "Pretrained" agent adapted faster on the unseen track, suggesting a beneficial inductive bias from supervised pre-training for zero-shot scenarios, both approaches eventually learned to drive. The "Pure RL" group, relying only on a pre-trained visual backbone, exhibited a longer initial exploration phase, underscoring the challenge of learning complex control policies from scratch.

The experiments also underscored the critical role of the reward function design, specifically the aimpoint mechanism. The failure of the agent "W/O Aim Point" to complete most tracks starkly illustrates that this reward signal is fundamental for guiding the learning process towards successful driving policies.

While the learning-based agents demonstrated adaptability, they generally did not match the raw speed of the "Raw Planner" on familiar tracks. This suggests that while RL agents can learn generalizable strategies, specialized planners might still hold an edge in terms of speed on known terrain. The "Log of Mistakes" section further highlights practical challenges in RL implementation, such as tensor dimension mismatches and the careful handling of normalization layers like BatchNorm, which can significantly impact training stability and performance. Future work could explore further refinements in reward shaping or model architecture to improve both speed and learning efficiency.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv (Cornell University)*, 2013. doi: 10.48550/arxiv.1312.5602. Publisher: Cornell University.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017. URL `http://arxiv.org/abs/1707.06347`.

Wei Shi, Yanghe Feng, Honglan Huang, Zhong Liu, Jincai Huang, and Guangquan Cheng. Efficient hierarchical policy network with fuzzy rules. *International Journal of Machine Learning and Cybernetics*, 13(2):447–459, 2021. doi: 10.1007/s13042-021-01417-2.