# Reinforcement Learning Project SuperTuxKart

Hu Yingcai, Félix Bos, Xu Shuo

December 2025

## 1 Introduction

Reinforcement learning (RL) provides a natural framework for sequential decision-making problems with delayed feedback and continuous control. In this project, we study an RL approach for autonomous kart racing in the SuperTuxKart environment, where an agent must learn to control steering and acceleration to complete a race efficiently.

This task presents several challenges typical of practical RL problems, including a high-dimensional structured observation space, a hybrid continuous-discrete action space, and delayed reward signals primarily based on progress along the track and race outcome. These characteristics make stable policy learning non-trivial.

To address these challenges, we formulate the task as a Markov Decision Process and adopt a policy-gradient-based method to learn a stochastic control policy from interaction with the environment. The objective of this project is not only to achieve good performance, but also to analyse the learning behaviour and limitations of policy optimisation methods under realistic computational constraints.

## 2 MDP Formulation

The autonomous kart racing task is formulated as a Markov Decision Process (MDP) defined by the tuple $(\mathcal{S}, \mathcal{A}, R, \gamma)$. At each discrete time step $t$, the agent observes a state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$, and receives a scalar reward $r_t$.

The state space $\mathcal{S}$ consists of high-dimensional structured observations provided by the environment, including track-related geometric information, vehicle dynamics, and interaction-related features such as items and other karts. The action space $\mathcal{A}$ is hybrid, combining continuous control commands (steering and acceleration) with discrete binary decisions (e.g., braking, drifting, and item usage).

The reward function is designed to encourage steady progress along the track and successful race completion. At time step $t$, the reward is defined as

$$r_t = \frac{1}{10}(d_t - d_{t-1}) + \left(1 - \frac{\text{pos}_t}{K}\right)(3 + 7f_t) - 0.1 + 10f_t,$$

where $d_t$ denotes the cumulative distance traveled along the track, $\text{pos}_t$ is the position of the kart among the $K$ competitors at time $t$, and $f_t$ is an indicator variable equal to 1 when the kart finishes the race and 0 otherwise. This formulation combines dense progress-based rewards with sparse terminal feedback.

The agent aims to maximize the expected discounted return

$$E\left[\sum_{t=0}^{T} \gamma^t r_t\right],$$

where $\gamma \in (0, 1)$ is the discount factor.

# 3 Moddel Architecture

## 3.1 Overall Architecture

The overall architecture of our reinforcement learning framework is illustrated in Figure 1. The agent interacts with the SuperTuxKart environment in an on-policy manner. At each time step, the environment provides a raw structured observation, which is first processed by an observation wrapper and then encoded into a compact latent representation.

The observation encoder produces a fused state embedding that captures both the ego-vehicle state and its surrounding context. This embedding is shared by the policy and value networks within an actor–critic architecture. The policy network outputs action distributions, while the value network estimates the state value function.

During training, trajectories generated by the current policy are stored in an on-policy buffer and used to update the model parameters using Proximal Policy Optimisation (PPO). Parameter updates are performed using the Adam optimiser. This design ensures a clear separation between observation processing, decision making, and policy optimisation while maintaining an efficient end-to-end learning pipeline.

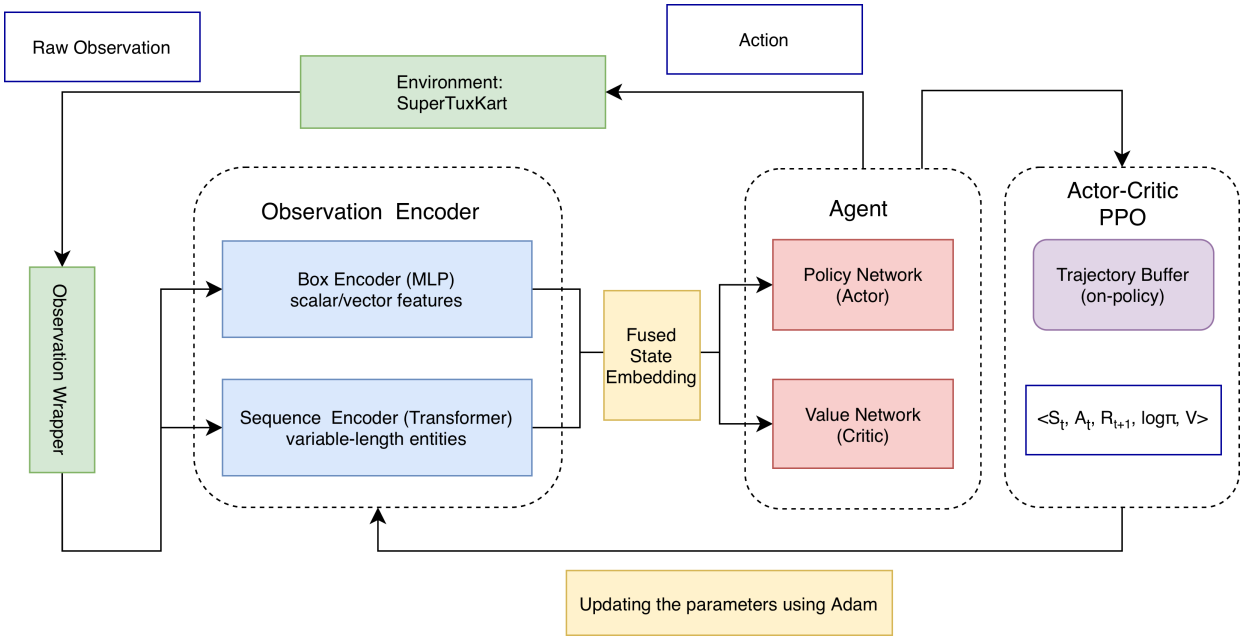


Figure 1: Overall architecture of the proposed actor–critic framework, showing the interaction between the environment, observation encoder, agent, and PPO optimization.

## 3.2 Observation Encoder

The observation provided by the SuperTuxKart environment is a structured dictionary containing heterogeneous components, including scalar values, fixed-size vectors, and variable-length entity sets. Directly flattening such observations may ignore their underlying structure. To address this, we design a structured observation encoder that explicitly separates different observation modalities.

As shown in Figure 2, the observation wrapper first decomposes the raw observation into two main categories: box observations and sequence observations. Box observations correspond to scalar or fixed-dimensional vector features, such as ego-vehicle dynamics and track-related quantities. These features are processed by a multi-layer perceptron (MLP) to produce box embeddings.

Sequence observations correspond to sets of variable-length entities, such as nearby karts, items, or path segments. These entities are encoded using a transformer-based sequence encoder, which naturally handles variable-length inputs and captures interactions among entities of the same type. Type-specific encoders are used to process different categories of entities independently.

The resulting box embeddings and sequence embeddings are then aggregated into a single fused state embedding. This fused representation serves as a compact and expressive summary of the environment state and is used as the input to both the policy and value networks. Note that there are two observation encoders, one is designed

specifically for the policy network, and the other is specifically for the value network. All of the parameters about the precise architectures are presented in the table 1.

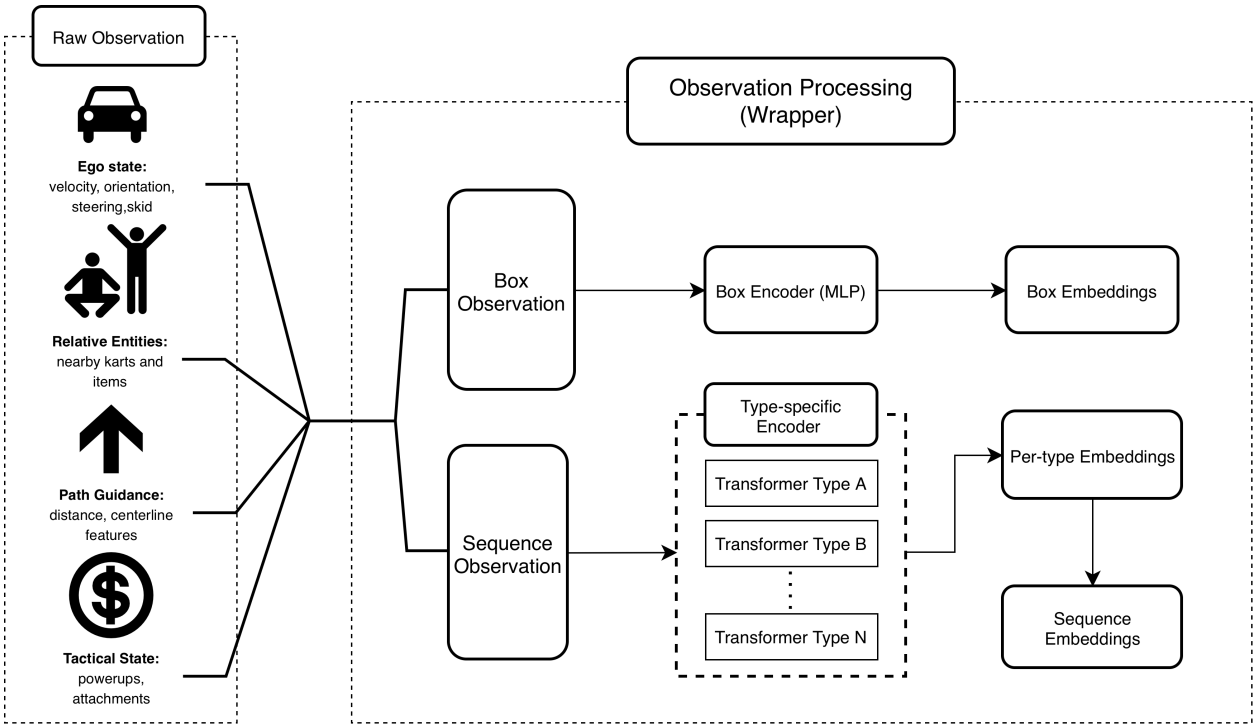


Figure 2: Detailed structure of the observation encoder. Box observations are encoded using an MLP, while sequence observations are processed by transformer-based encoders. The resulting embeddings are fused into a unified state representation.

### 3.3 Value and Policy Heads

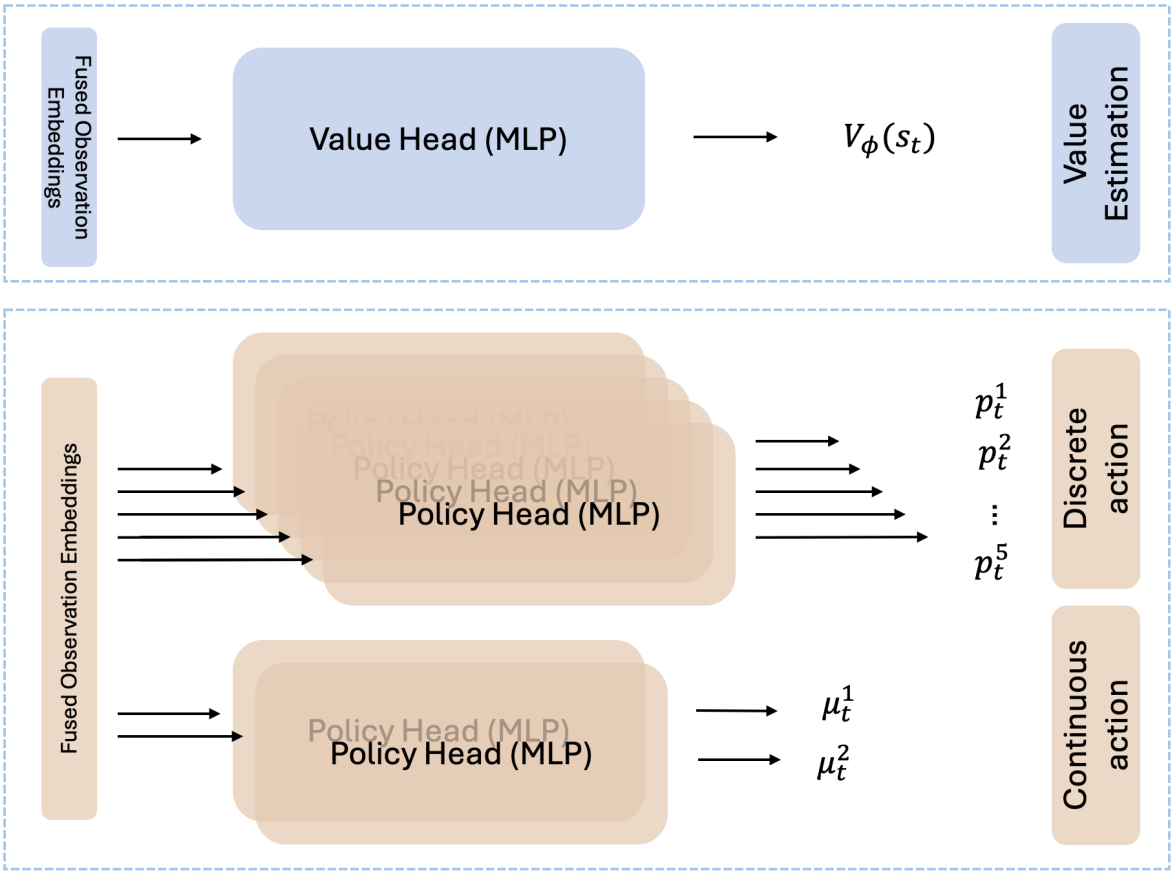


Figure 3: Detailed structure of the networks for value and policy function estimation. The value network computes value estimate at each time step. The policy network receives a policy-specific fused state embedding. The policy heads either give a logit value for a discrete action or a mean for continous action type.

The fused state embeddings will then be passed to a value head and policy heads. The state-dependent values $V(s_t)$ are obtained via a value head, which is a 2-layer MLP to compute the advantage function (Figure 3). A collection of two-layer MLPs is used to obtain the policy. Each policy head receives the same fused observation embeddings from the policy observation encoder to learn a specific action type (Figure 3). The policy head outputs a logit value $p_t$ for a discrete action type, then the action $a_t$ for a specific discrete action value variable will be modelled by a Bernoulli distribution:

$$a_t \sim \text{Bernoulli}(p_t(\theta))$$

On the other hand, we consider the policy $\pi_c$ for continuous action as a parametrised normal distribution. Hence, the policy head will estimate the mean $\mu_t(\theta)$ and a state-independent standard deviation $\sigma$ for the continuous action type:

$$\begin{aligned} a_t &\sim \mathcal{N}(\mu_t(\theta), \sigma) \\ a_t &= tanh(a_t) \end{aligned} \tag{1}$$

### 3.4 Optimization

Here we employ Proximal Policy Optimisation (PPO), where we constrain our policy to be updated in small incremental steps. The optimisation objective for policy network includes a clipped surrogate and entropy regularisation:

$$\mathcal{L}^{\text{PPO}}(\theta) = E_t\Big[\min\Big(r_t(\theta)\,\hat{A}_t,\ \text{clip}\big(r_t(\theta),\,1-\epsilon,\,1+\epsilon\big)\,\hat{A}_t\Big)\Big] \ +\ \beta\,\mathcal{H}(\pi_\theta(\cdot \mid s_t))\,, \tag{2}$$

where $\beta$ is the regularisation parameter controlling the exploration of this high-dimensional state space. The gradient of this loss will be backpropagated to the policy-specific observation encoder. The $r_t$ is the log ratio of the current and the new policy

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

The entropy regularisation is then defined as:

$$\mathcal{H}(\pi_\theta(\cdot \mid s)) = -E_{a\sim\pi_\theta}[\log \pi_\theta(a \mid s)]\,. \tag{4}$$

To ensure stable multi-epoch PPO updates under truncated rollouts, we choose generalised advantage estimation (GAE) $\hat{A}_t$ :

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l\, \delta_{t+l}, \tag{5}$$

Here $l$ represents the length of trajectories we used per update. The reward prediction error $\delta_t$ is dependent on the value function $V(s_t)$:

$$\delta_t = r_t + \gamma(1 - d_t)\, V_\phi(s_{t+1}) - V_\phi(s_t), \tag{6}$$

. In parallel, the value network is optimized by minimizing the $\mathcal{L}^{Value}$ with regression:

$$\mathcal{L}_V(\phi) = E_t\left[\left(V_\phi(s_t) - \hat{G}_t\right)^2\right]. \tag{7}$$

Typically, $\hat{G}_t$ is the estimated return at time $t$:

$$\hat{G}_t = \hat{A}_t + V_{\phi_{\text{old}}}(s_t) \tag{8}$$

# 4 Results

Due to computational constraints (RAM runout), we are unable to train our current model even with reduced sizes.

| Module | Parameter | Value |
|---|---|---|
| All related parameters are presented in table 1. *Value Network* | | |
| ValueNet | Hidden sizes | [64, 32] |
| | Hidden activation | ReLU |
| | Output activation | Identity |
| | Learning rate | $1 \times 10^{-4}$ |
| *Policy Network (Discrete Actions)* | | |
| Policy (disc) | Hidden sizes | [64, 32] |
| | Hidden activation | ReLU |
| | Output activation | Identity |
| *Policy Network (Continuous Actions)* | | |
| Policy (steer) | Hidden sizes | [64, 32] |
| | Hidden activation | ReLU |
| | Output activation | Identity |
| Policy (accel) | Hidden sizes | [64, 32] |
| | Hidden activation | ReLU |
| | Output activation | Identity |
| Policy (all) | Learning rate | $1 \times 10^{-4}$ |
| *Observation Encoder* | | |
| Box encoder | Output dimension | 32 |
| | Hidden sizes | [128, 64] |
| | Hidden activation | ReLU |
| | Output activation | Linear |
| Seq encoder | Model dimension $d_{\text{model}}$ | 64 |
| | Number of heads | 2 |
| | Number of layers | 2 |
| | Dropout | 0.1 |
| | Layer normalization | True |
| *PPO Algorithm* | | |
| PPO | PPO epochs | 5 |
| | Discount factor $\gamma$ | 0.99 |
| | GAE parameter $\lambda$ | 0.95 |
| | Clip parameter $\epsilon$ | 0.2 |
| | Max trajectory length $l$ | 64 |
| | entropy regularization coeffcient $\beta$ | 0.001 |
| *Training* | | |
| Training | Max epochs | 10000 |
| | Update interval | $32 \times 16$ |
| | Batch size | 16 |

Table 1: Hyperparameters and architectural details of the PPO agent. Here the update interval indicates the number of environment steps that the agent collects for PPO update. The number of elements in the hidden size indicates the number of layers, where the elements indicate the size of the linear layer. PPO epochs indicate the number of PPO updates per minibatch of trajectories.