Sorbonne Université

Faculté des Sciences et Ingénierie

UFR d'Ingénierie

Master II   Données, apprentissage, connaissances

Module : *Reinforcement Learning and advanced Deep Learning*

Rapport Projet Super Tux Kart

Réalisé par :

BENDIB   Nazim 21417246

BOUDJENAH Nassim 21418045

Contents

1 Introduction

SuperTuxKart (STK) is a popular open-source kart racing game that presents unique challenges for Reinforcement Learning. These challenges include complex physics, stochastic environments, and the need for decision-making against multiple opponents. In this report, we present a novel approach

that integrates an expert agent with customized training strategies to improve policy learning in STK. Our method involves designing new observation wrappers, implementing an improved expert agent, and employing a hybrid training framework inspired by the DAGGER [1] algorithm. Through these enhancements, we aim to create an RL agent that not only mimics expert behavior but also learns effective recovery strategies, ultimately improving its racing performance.

2 Motivation

The current SuperTuxKart environment is quite slow and difficult to parallelize. As a result, we believe it is not wise to use on-policy [2] algorithms like PPO [3], as PPO is not sample-efficient and requires long training times. A more reasonable approach would be to use Imitation Learning and Behavioral Cloning [4], given that the STK bindings include a built-in expert agent. However, simply performing Behavioral Cloning by fitting a model to the actions of the built-in expert agent led to very poor results despite the expert agent being highly competent. This is because the expert agent is near-perfect and never makes mistakes. Consequently, the model only learns from expert trajectories, but as soon as it makes a mistake, it encounters unseen states and does not know how to recover (often getting stuck against a wall with no way to escape).

One intuitive way to mitigate this issue is to introduce mistakes during training by performing random actions and observing how the expert agent corrects them. This would allow the model to learn appropriate recovery strategies for similar situations. Unfortunately, the current implementation of `pystk2_gymnasium` does not support this approach, as we must either use the expert agent or manually select actions—we cannot mix both modes. Adding this feature in the future would be highly beneficial.

The ability to alternate between expert actions and custom actions would enable more effective Behavioral Cloning algorithms. That's why... *drumroll...* we decided to implement our own expert agent, giving us full control to design our training algorithm as needed.

3 Our method

In the following section, we detailed all the components of our solution, including wrappers, custom expert, training algorithm, and policy network.

3.1 Wrappers

To develop our expert agent, we implemented multiple wrappers to preprocess observations. These wrappers help structure the input data and apply adjustments:

1) ActionTimeExtensionWrapper

This wrapper stores previous actions, as we believe that keeping a history of past actions is crucial for deciding which action to take next.

2) ExpertObservationWrapper

This wrapper produces simpler and more informative observations for the expert agent. Since we take observations from `supertuxkart/full-v0`, we filter them to retain only the necessary information and engineer new observations that simplify decision-making. These new observations include:

- **start_race (bool)**: To detect whether the kart is stuck, we check if its speed is null (or almost null). If the kart is stuck, it should move backward; if it is at the start, it should move forward. However, a stationary kart at the start of the race is normal. This distinction makes this information useful.
- **powerup (int)**: In the `pystk2_gymnasium` implementation, the power-up is represented as `kart.powerup.num`, which always returns a constant value rather than the actual power-up ID. We override this value with `kart.powerup.type.value`, which correctly provides the power-up's ID.
- **obstacle_ahead (bool)**: Indicates whether there is an obstacle ahead of the kart, triggering dodging behavior for obstacles such as bananas or bubblegum.
- **obstacle_position (int[3])**: The (x, y, z) coordinates of the obstacle ahead. If no obstacle is detected, this is set to (-1, -1, -1).
- **target_position (int[3])**: Using observations from `supertuxkart/full-v0`, we compute the furthest point that the kart can reach in a straight line and set it as the target. This is a common strategy in racing bots, and when inspecting the C++ code of `PySTK`, we found that it follows a similar approach for optimal movement.
- **target_distance (int)**: The distance from the kart's current position to the target position.
- **target_angle (int)**: The angle between the kart and the target position.

3.2 Our expert

Our expert can be seen as a pure function that takes a state observation as input and returns the action to take in the form of a dictionary with both continuous actions (speed and steer) and discrete actions (brake, drift, fire, nitro, rescue). For simplicity, we always set drift to 0 and rescue to 1. A pseudo algorithm for the expert is presented in Algorithm 1.

3.3 Training algorithm

Now that we have our custom expert agent, the goal is to train a policy network that mimics its behavior. Note that even our custom expert rarely makes mistakes, so we need to intentionally introduce mistakes in the decision-making to ensure the policy network learns how to recover from them. We achieve this by alternating between expert actions and policy network actions. More specifically, we apply expert actions for 50 steps, then policy network actions for the next 50 steps, and repeat this cycle.

At each step, we store the state and the action selected by the expert in a buffer. This buffer is later used to train the policy network. Initially, the policy network selects random actions, which often lead the kart into undesirable situations (going in circles, stuck against the wall), serving as an exploration phase. However, as training progresses, the policy network improves and gradually shifts toward exploiting its learned behavior.

Our training strategy resembles an existing method called DAGGER [1]. A pseudo-code of the training algorithm is presented in Algorithm 2.

3.3.1 Stratified sampling

One key setup that is important in the training is to use stratified sampling. Because the values of one action type (speed, for example) are not uniformly distributed and often follow a very peaky distribution, which makes it hard to cover all scenarios if we use uniform sampling (as shown in Experiments 4.3). to avoid this, we use stratified sampling, where for each action type we sample values that are evenly spaced and cover the whole range. This way, the policy network will be trained on both very occurring and rare scenarios (like dodging an obstacle, driving backward to escape the wall, etc...).

Algorithm 1 Expert agent function

```
if not start_race and (kart_is_going_backward or kart_is_stuck) then
    speed, steer, brake, fire, nitro ← [0.0, 0.0, 1, 0, 0]
    goto END
end if
if obstacle_ahead then
    if $0 <$ obstacle_position.x $< 2$ and obstacle_position.z $> 2$ then
        speed, steer, brake, fire, nitro ← [0.5, -0.5, 0, 0, 0]
        goto END
    else if $-2 <$ obstacle_position.x $\leq 0$ and obstacle_position.z $> 2$ then
        speed, steer, brake, fire, nitro ← [0.5, 0.5, 0, 0, 0]
        goto END
    end if
end if
steer ← target_angle / 20
speed ← 1
if |target_position.x| > 5 then                                    ▷ Steep turn
    speed ← 0.5
end if
nitro ← 0
if target_distance > 30 then
    nitro ← 1
end if
fire ← 0
if powerup $\neq 0$ then
    if powerup $\in$ {BUBBLEGUM, PARACHUTE, ANVIL} then
        fire ← 1
    else if powerup $\in$ {CAKE, BOWLING, PLUNGER, RUBBERBALL} and kart_is_ahead then
        fire ← 1
    end if
end if
END:
action ← { 'continuous': [speed, steer], 'discrete': [brake, 0, fire, nitro, 1] }
return action
```

Algorithm 2 Imitation Learning with Buffer

```
buffer ← Buffer(max=100000)
steps ← 0
state ← env.reset()
for epoch in range(epochs) do
    for $i = 1$ to 100 do
        expert_action ← expert(state)
        if $(steps \div 50) \mod 2 = 0$ then
            policy_action ← policy_network(state)
            next_state ← env.step(policy_action)
        else
            next_state ← env.step(expert_action)
        end if
        steps ← steps + 1
        buffer.append((state, expert_action))
        state ← next_state
    end for
    for $i = 1$ to training_steps do
        state, expert_action ← buffer.stratified_sample(batch_size)
        policy_action ← policy_network(state)
        loss ← criterion(policy_action, expert_action)
        optimizer(policy_network, loss)
    end for
end for
```

3.4 Policy network

For the policy network, we decided to keep it simple and use an MLP. However, since we have multiple actions that include both continuous and discrete components, we used multiple prediction heads—one for each action. The following figure shows the architecture of our policy network:

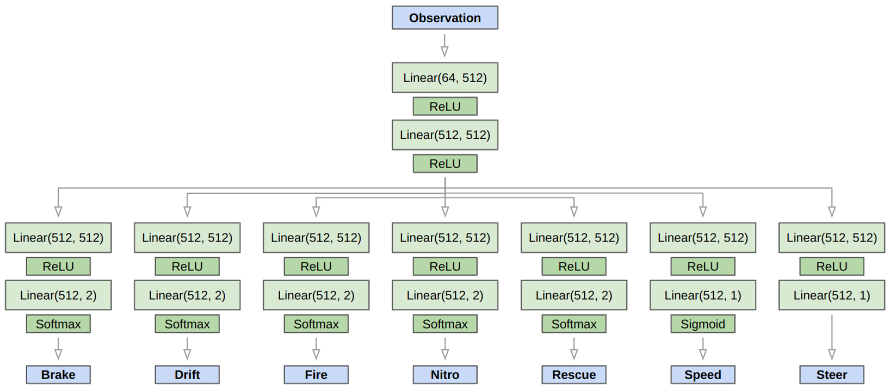

Figure 1: The architecture of our policy network. It starts with a shared backbone and then seven prediction heads for each action.

4 Experiments

In the following, we present the experiments conducted to assess the performance of our method. To compare policies, we run multiple races (10 in total), where the competing policies race alongside 10 bots set to the maximum difficulty (difficulty = 2). We then evaluate performance based on the average reward and average ranking. For training our main policy network, we use the Adam optimizer with a learning rate of 0.001, Mean Absolute Error (MAE) for continuous actions, and Cross-Entropy Loss for discrete actions. The policy network has an input size of 64, corresponding to the size of the observation space.

4.1 Comparaison between our expert and the PySTK built-in bots

As a first experiment, we wanted to evaluate the performance of our custom expert against the PySTK built-in expert. To achieve this, we run multiple races where our expert competes against the built-in bots at maximum difficulty.

Difficulty	**Our Expert's Average Ranking**	**Our Expert's Average Reward**
Difficulty = 1	1.4 ± 1.1	19.1 ± 3.1
Difficulty = 2	3.6 ± 1.7	17.6 ± 2.8
Difficulty = 3	7.4 ± 2.9	11.3 ± 5.1

Table 1: Performance of our expert across different difficulty levels.

Analysis

We can see in Table 1 that our expert achieves results comparable to the built-in bots. Specifically, it ranks well against the easy and medium difficulty levels while achieving an average ranking of 7.4 against the maximum difficulty. This is very good, given that the built-in bots in SuperTuxKart have an advantage due to a starting boost and additional boosts during the race that, to the best of our knowledge, our agent cannot use.

4.2 Comparaison with simple Imitation Learning and PPO

In the following experiment, we compare our method against two policies: One trained using simple Imitation Learning and a second policy trained using PPO [3]. For Imitation Learning, we simply fit the policy network on the actions of the expert by minimizing a loss function between their actions. For PPO, we use a learning rate of 3e-4, a batch size of 64, and train for 10 epochs. The discount factor (γ) is set to 0.99, while the Generalized Advantage Estimation (GAE) parameter (λ) is 0.95. We apply a clipping ratio of 0.2 to stabilize training and use an entropy coefficient of 0.01 to encourage exploration. Additionally, the value loss coefficient is set to 0.5 to balance the policy and value function updates. For the PPO policy network, we use the same architecture shown in Fig 1 and an MLP with four layers of 256 neurons for the value network.

Method	**Average Ranking**	**Average Reward**	**Training time**
Imitation Learning	8.8 ± 0.7	3.1 ± 2.3	16min
PPO	7.1 ± 2.0	9.1 ± 5.5	112min
Ours	**5.5 ± 1.4**	**15.3 ± 2.5**	**13min**

Table 2: Comparison of average ranking, reward and time across different methods.

Analysis

We can see in Table 2 that our method consistently outperforms both Imitation Learning [4] and PPO. We believe the reasons for this are aligned with the motivation of this work: Imitation Learning fails to teach the policy network how to recover from mistakes, while PPO is not very sample-efficient and struggles to produce high-quality policies within our constraints. It likely requires significantly more training iterations and a larger number of parallel environments to achieve better performance. It is also worth noting that our method is much faster than PPO, which is one of the main features of our solution.

4.3 Stratified sampling vs Uniform sampling

In this experiment, we aim to evaluate the effect of stratified sampling by training two policies using our method: one with stratified sampling and the other with uniform sampling.

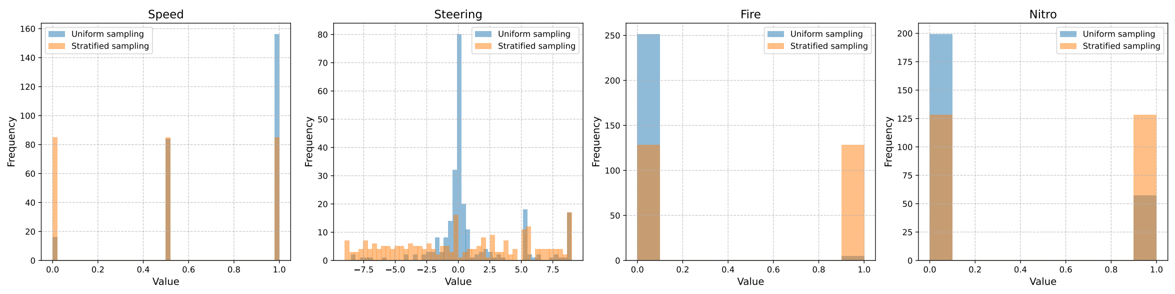

Figure 2: The distribution of the values of the actions (Speed, Steer, Fire, Nitro) when sampled using stratified sampling and uniform sampling.

Method	Average Position	Average Reward
With uniform sampling	6.4 ± 2.3	12.9 ± 4.2
With stratified sampling	**7.6 ± 1.5**	**13.4 ± 5.1**

Table 3: Comparison of average position and reward using uniform and stratified sampling.

Analysis

As shown in Fig.2, the distribution of the values of the actions is very sharp, and simply doing uniform sampling will return samples that are very close to the mode of the distribution, which means that the policy network will less likely be trained on particular situations that doesn't occur much in the race, for example if a race takes 1000 steps to finish, and the kart faces 5 obstacles in the episode, then we will have only 5 samples out of 1000 total samples that represent a situation where we need to dodge an obstacle, which is very important in a race even though it doesn't occur that much. So, stratified sampling forces the policy network to get trained on all kinds of scenarios and give them all the same importance no matter how sharp they are distributed, which yields a better policy, as shown in Table 3.

4.4 Adding expert trajectories form the `PySTK` built-in expert

As an extension of our work, we decided to also include some trajectories from the PySTK built-in expert as we believe it should be better than the simple expert that we implemented. A simple way to achieve this is to initialize the replay buffer with collected trajectories from the PySTK built-in expert.

Method	Average Ranking	Average Reward
With PySTK expert trajectories	8.2 ± 1.5	9.7 ± 3.3
Without PySTK expert trajectories	**6.5 ± 1.2**	**14.0 ± 2.7**

Table 4: Comparison of methods with and without PySTK expert trajectories.

Analysis

We observe in Table 4 that mixing trajectories from the PySTK built-in expert and our custom expert leads to a policy that performs worse than the one trained solely with our expert. Upon inspecting the learned policy's behavior, we noticed that it appears indecisive, trying to follow both strategies at the same time, which results in suboptimal performance.

5 Conclusion

In this work, we have demonstrated an effective approach to training an RL agent for SuperTuxKart. By designing a custom expert agent and implementing modifications to the observation space, we enabled the model to learn more robust policies. Our method improves upon traditional Behavioral Cloning by introducing controlled mistakes, allowing our policy to learn recovery behaviors and enhance generalization. Through a series of experiments, we validated the effectiveness of our approach, showing that our trained policy network outperforms standard Behavioral Cloning models as well as reinforcement learning algorithms like PPO [3], which suffer from high sample inefficiency in this environment. While our results are promising, future work could explore further refinements, such as:

- Developing more performant experts, as the quality of the policy heavily depends on the quality of the expert.
- Exploring the adaptation of our method to multiple experts instead of a single expert while maintaining performance.
- Although we attempted to refine our trained policy network using PPO, it did not improve the results. Further investigation into this could be valuable, as it is expected to yield better outcomes.

References

[1] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.

[2] Satinder Singh, Tommi Jaakkola, Michael Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38:287–308, 03 2000.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] Boyuan Zheng, Sunny Verma, Jianlong Zhou, Ivor W Tsang, and Fang Chen. Imitation learning: Progress, taxonomies and challenges. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.