Optimizing Autonomous Racing in the Donkey Car Simulator: Novel Reward Functions and Regularization Techniques

Albert Fugardo Cortada
albert.fugardo@estudiantat.upc.edu

Margarita Cabrera-Bean
marga.cabrera@upc.edu

Josep Vidal
josep.vidal@upc.edu

Universitat Politècnica de Catalunya

Abstract

Autonomous driving has gained significant popularity in recent years. Many commercial vehicles are now equipped with self-driving capabilities, allowing them to operate with minimal human intervention. One exciting application of this technology is autonomous racing, where self-driving vehicles aim to complete laps on a track as quickly as possible. Significant advancements have been made in this field, exemplified by initiatives like Formula Student and the Abu Dhabi Autonomous Racing League. These developments offer new insights into optimal racing lines and racing dynamics. In this paper, we propose enhancements to Donkey Simulator, a simulated autonomous racing environment. The primary challenge addressed by this paper is achieving fast lap times with minimal training. Our approach is based on a deep reinforcement learning framework, where we develop and implement novel reward functions to improve lap times. Additionally, we significantly reduce the number of training steps required to achieve competitive lap times, thereby enhancing training efficiency. We also introduce several architectural modifications to the model to address these challenges and further optimize performance.

This paper is accompanied by a video of the performance of the car: `https://youtu.be/bgBmoeJbTeQ`

1. Introduction

The goal of autonomous car racing is to complete a lap on a given track in the shortest possible time. Developing a system to achieve this may seem straightforward, but it involves numerous challenges. Firstly, the system must make real-time decisions, making execution speed crucial. Secondly, the environment is often non-stationary, requiring the system to adapt to various changes. Thirdly, a significant amount of data must be collected and properly handled.

Classical approaches to autonomous racing typically involve two main steps. The first step is to obtain a map of the track, which can be done by using a pre-loaded map (which must be generated manually) or generating the map on-the-fly. For the latter method, the car takes slow laps to collect data using reliable sensors, ensuring precise mapping of the track. These sensors must provide high accuracy to precisely locate the car on the map. In the second step, sophisticated algorithms use the track map and vehicle characteristics to generate an optimal trajectory for the car to complete laps as quickly as possible. An example of the usage of this methodology are the driverless Formula Student cars, using it since their inception [18].

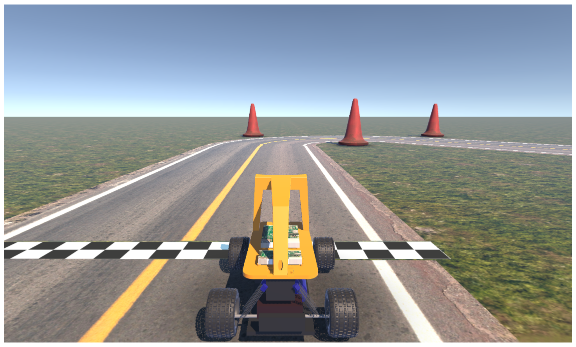
Figure 1. Donkey Simulator framework, in which this work has been developed.

However, classical approaches have several drawbacks. Precise sensors are required to ensure proper mapping. Additionally, once the algorithm is developed, there is no room for performance improvement, as it is restricted by the initial algorithm and does not account for environmental changes. Finally, complex nonlinear optimization methods are often needed. Overall, conventional systems for autonomous driving increase in complexity with the model and do not leverage the vast amounts of data generated by real-world driving.

In this work, we tackle the autonomous car racing problem using Donkey Simulator (Figure 1), a driving simulator aimed at developing self-driving models for both simulation and real-life remote control cars. Donkey Simulator is known for its ease of use and focus on user development.

Our approach employs model-free deep reinforcement learning, a *time-sector* reward function, and a variational autoencoder to train a multi-layer perceptron policy for vehicle control. The key is formulating the minimum lap time problem by dividing the circuit into sectors and optimizing the sector time for each sector individually. Additionally, we utilize a suitable low-dimensional representation of the input image to balance policy efficiency and input detail retention. As a result, this work makes three main contributions:

Development and Analysis of New Reward Functions: We introduce and analyze reward functions not previously used in this framework, including the *time-sector* reward. This reward improves lap times compared to agents trained with the baseline reward, which follows the center line of the track.

Investigation of Regularization Methods: We explore regularization techniques for reinforcement learning in autonomous car racing environments. This includes encoding input image observations using a Variational Autoencoder (VAE) and applying deep learning regularization techniques, such as weight decay and input/reward normalization, to the model networks.

Framework Enhancements for Future Research: We prepare the framework to facilitate further research. Our goal is to provide a foundation for future researchers working with self-driving simulators or real remote control cars. We hope the results of this work will be beneficial for future research endeavors.

2. Related Work

Our work builds upon previous works in simulation autonomous car racing using deep reinforcement learning.

General frameworks: There are numerous frameworks for developing self-driving racing algorithms, and the methodologies applied in some of these can be transferred to our environment.

Most of the works utilize standard reward functions, which typically include the longitudinal speed of the car, the angle between the car's direction and the racing line, and the distance of the car from the track axis. Examples of works including such rewards can be found in [7, 14, 5].

In [7], the RGB input image is used to feed an Asynchronous Actor-Critic (A3C) framework for the WRC6 rally environment simulator.

In [14], vehicle telemetry data serves as the input for the model. This model uses a Deep Deterministic Policy Gradient (DDPG) approach and includes window sampling (feeding the last w states to the actor and critic network, where w is a hyperparameter). The environment used is TORCS (The Open Racing Car Simulator). An innovation in this work is that, apart from considering the distance of the car to the track axis in the reward function, they also propose considering the distance of the car to any axis, such as a given racing line.

In [5], the research is conducted using the F1Tenth simulator. A DDPG is employed, and a reward of 1 is given if a lap is completed, -1 if a crash occurs, and otherwise a reward is based on the variables mentioned at the beginning of this section. Additionally, they incorporate a supervisor, and a reward of -1 is given when it intervenes. The objective of this work is to explore safe reinforcement learning while also driving fast.

In [6], a significant milestone is achieved with state-of-the-art performance. A policy is developed that outperforms the best driver in their dataset of human players in the popular simulator Gran Turismo Sport. The algorithm used is the Soft Actor-Critic (SAC), and the reward function developed maximizes course progress while adding a secondary reward to penalize wall contact. The model inputs include the car's velocity, orientation, and other metrics that are obtained from the simulator.

In [17], the model from [6] is improved, and an algorithm capable of overtaking in Gran Turismo Sport is developed. In addition to the previously mentioned reward, another reward is introduced to avoid collisions and promote overtaking. The Curriculum Soft Actor-Critic is used, training the model in three major stages, each emphasizing a different reward.

Donkey Simulator framework: There has also been some research using the Donkey Simulator environment, although no extensive research has been conducted.

In [21, 20], a Double Deep Q-Network (DDQN) is employed to train a model in the Donkey Simulator. Notably, in [21], the trained model is transferred to a real RC car. The innovation in these works lies in the preprocessing of raw input images to extract features such as track lines before feeding them to the Convolutional Neural Network (CNN). This preprocessing uses line segmentation methods as explained in [2].

In [19], a Variational Autoencoder (VAE) is utilized to encode the raw input images. What this does is reducing the amount of parameters of the Reinforcement Learning algorithm, making it more efficient.

Finally, Raffin has made significant advancements and established foundational work for the Donkey Simulator. In [9], the basics for keeping the car on track are introduced. Furthermore, in [13, 11, 12], additional ideas are presented, including the use of a Variational Autoencoder to encode input images and various hyperparameter tuning techniques.

However, none of these studies introduce a new reward function. As stated in our introduction, developing a new reward function for this framework is one of the aims of this work.

3. Methodology

3.1. Preliminaries

Reinforcement Learning (RL) is a powerful machine learning approach that enables agents to learn optimal behavior through trial and error in dynamic environments. Its strength lies in addressing problems where explicit guidance or labeled data are scarce. RL is particularly effective for tasks requiring sequential decision-making, offering flexibility, adaptability, and the ability to learn directly from interactions with the environment. By balancing exploration and exploitation, RL agents can autonomously discover strategies that maximize cumulative rewards over time, making it a valuable tool in various domains, including racing simulators. An example of a successful RL application is MuZero [15], which achieves superhuman performance in a range of challenging and visually complex domains, without any knowledge of their underlying dynamics.

One of the most popular RL algorithms is the **Actor-Critic** (AC) method. In the AC framework, an agent (the *Actor*) learns a policy to make decisions, and a value function (the *Critic*) evaluates the actions taken by the Actor.

Proximal Policy Optimization (PPO) [16] is an advanced variant of the AC algorithm. It has proven to be one of the most effective RL algorithms, with organizations such as OpenAI adopting it as their default choice. Compared to other algorithms, PPO offers three main advantages: simplicity, stability, and sample efficiency.

Given these advantages, PPO is highly suited for car racing simulation, where interaction with a constantly changing environment and the need for rapid decision-making are critical.

3.2. Minimum time problem and reward function

We aim to find an agent that minimizes the lap time the car is able to do. Given that lap time is a very sparse signal, relying solely on it for rewards is not ideal, as it would create challenges for the agent in learning the optimal trajectory. To address this, we have engineered a new reward, which we call the *time-sector* reward. This reward involves dividing the track into sectors and giving a reward inversely proportional to the time taken to complete each sector. By having numerous sectors, this reward becomes more frequent compared to one based solely on lap time. The chosen sectors, illustrated in Figure 2, are more concentrated around curves, as these are the regions where the car will face the greatest difficulty in learning to drive quickly. Our idea is that with properly developed sectors, minimizing the time to complete each sector approximates to minimizing the overall lap time.

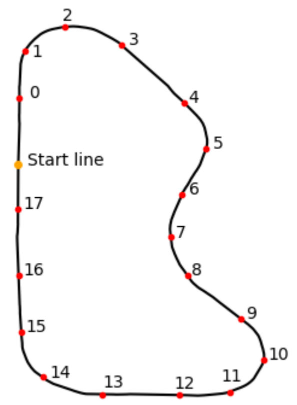

Figure 2. Track sectors chosen (in red).

Since rewards are given upon completing each sector, not every step will receive a reward. That's why a new reward term has to be introduced. In our case, we have opted to introduce the linear speed of the car. This way the car will be encouraged to go faster.

Moreover, we want the car to complete laps without veering off the track. Therefore, we add another term to the reward function that penalizes the car for exiting the track.

Taking all the aforementioned factors into account leads to the final reward function, defined as follows:

$$r_t = \begin{cases} -1 & \text{if off the track} \\ r_{sector} + c_v \cdot v & \text{otherwise} \end{cases} \tag{1}$$

Here, $c_v \geq 0$ is a hyperparameter controlling the trade-off between speed and the sector reward (with $c_v = 1/10$ used in our experiments), v is the forward velocity of the car and

$$r_{sector} = \begin{cases} 1/t_s & \text{if sector completed} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

with t_s being the time taken for the car to travel from the penultimate sector to the last sector crossed.

3.3. Model networks

We represent the driving policy with a deep neural network, utilizing the Proximal Policy Optimization (PPO) network architecture as proposed in [16]. Specifically, we use the implementation from Stable Baselines 3 [3]. The policy follows an Actor-Critic model, which consists of a policy network (actor) and a value network (critic). Both networks have two hidden layers with 256 Tanh nodes each. They also include a final linear layer to produce the desired output (Figure 3). This configuration results in a policy with 149,253 trainable parameters. Next, we provide a more detailed explanation of the network's inputs and its predictions.

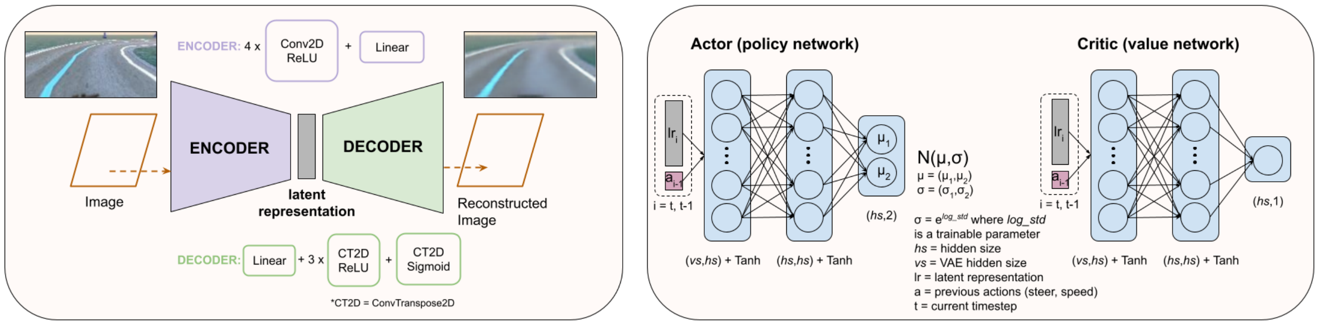

Figure 3. VAE network (left). The decoder part is only used during the training of this network, which is done offline. Actor and Critic networks (right). They receive as input the current and previous observations.

Input features: The car has a camera on top of it, which captures images of size 120x160x3. While these images could be fed directly into the network, learning directly from pixels is not efficient. Therefore, we have opted to use the encoder part of a Variational AutoEncoder [8]. The architecture employed consists of an encoder with four blocks of 2D Convolutions with ReLU activations, which reduce the tensor image of 3x120x160 (channels x height x width) to a tensor of size 128x3x8. This is followed by a final linear layer that reduces the tensor to a vector of a desired dimension, 32 in our case. The decoder mirrors this structure, beginning with a linear layer and followed by four blocks of 2D Transposed Convolutions with ReLU activations (the last layer having a Sigmoid instead) (Figure 3). The VAE is trained offline with images collected from the simulator. During policy training, the observed image passes through the encoder (which is freezed), reducing its size to a vector of 32 elements. These elements contain the most important information of the image, such as the location and position of the track lines, while filtering out unimportant information like background noise. Compressing the input image into such a vector significantly reduces the number of parameters the policy needs.

Additionally, we incorporate as input the previous action, since the image alone does not convey the car's previous steer or speed. Thus, we can define an observation at time t as $s_t = [lr_t, a_{t-1}]$, where lr is the latent representation and a is the action (steer, direction). We also add what we call *history* as input, which consists of the previous step's observations. We therefore represent the input at a given time step t as a vector denoted as $inp_t = [s_t, s_{t-1}]$ (we have chosen a history value of 2, current and past observation, but more observations could be added).
This combination results in an input of size 68 (32+2+32+2).

Network predictions: The actor (policy network) outputs a vector of two values, $\mu = [\mu_1, \mu_2]$. Together with $\sigma = [\sigma_1, \sigma_2]$, which is a trainable parameter, they are used as the parameters of a bivariate Normal distribution $N(\mu, \sigma)$. Sampling a value $x_t = [\alpha_t, v_t]$ from this Normal distribution gives the steer and speed to apply to the car in a given step.

4. Experiments

All the work's experiments have been performed using the Donkey Car simulator, as mentioned previously. The track used has been the *donkey-generated-track-v0*, with the default car and configurations, except for adjustments to the steering and speed limits. The steer limit has been set to 0.5 (the default value was 1.0), the minimum speed to -0.2 (the default value was 0.0) and the maximum speed to 1.5 (the default value was 1.0). Although the maximum speed value has been set to 1.5, the maximum value the simulator can handle is 1.0, so values above it are clipped to 1.0. The reason for setting the maximum speed to 1.5 is because it has been seen to encourage the agent to select values close to 1.0 more frequently. The choice of these parameters is justified by Raffin's study [10].

As mentioned before, the Variational Autoencoder has been trained offline, which means that has been trained before training the policy, with its usual reconstruction loss. Then, the encoder part has been integrated in the policy network, but frozen, meaning that its weights have not been modified during its subsequent training. To collect images for training the VAE, the car has been driven manually at first. Then, once the VAE has been good enough to make the car learn to complete full laps, more images have been collected during the policy testing.

During training, the framework speed has been increased by 4x, accelerating the training process fourfold. This has allowed us the testing of more models. The training framework operates as follows: the car drives on the track following the actor's actions, saving each step's information, such as the observation or reward to a rollout buffer. If the car goes off the track it is reset to the starting line. Once the rollout buffer is full (its size is a hyperparameter), the pol-

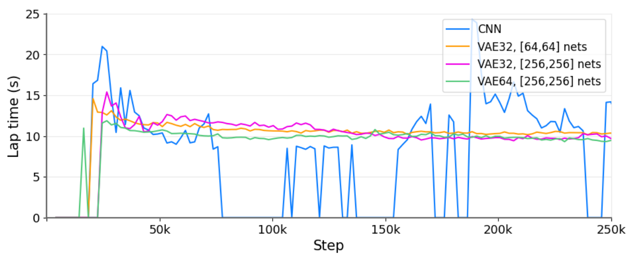

Figure 4. Testing lap time across steps for four different models. Each line is the average of three tests. A lap time of 0 means the car was not able to complete any of the 5 testing laps.

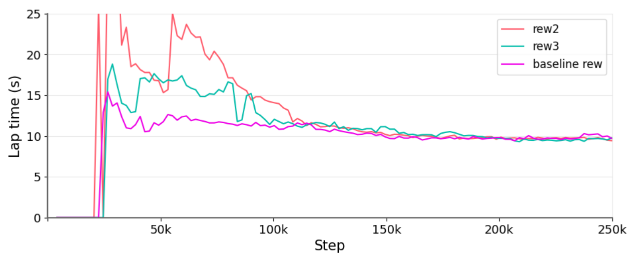

Figure 5. Testing lap time across steps for three equal models having different rewards. rew2 (A5), rew3 (A6) and baseline rew (A3).

icy and value networks are updated through backpropagation using the data in the rollout buffer. After the specified number of training epochs (also a hyperparameter), a test is conducted to evaluate the model over 5 laps, recording the average lap time. This data, together with the training metrics, is logged into Weights And Biases for later analysis. This process repeats until the total number of training steps is reached. In addition to the testing lap time, other performance metrics recorded include the average episode length and average episode reward. Since our objective is to complete a lap as quickly as possible, the testing lap time is the primary metric that we have used to evaluate the performance of the models. Additionally, some tests have been recorded in video, to visually analyze if the car speed and trajectory are optimal or not.

5. Results

In this section, we evaluate our proposed approaches. As mentioned, the primary evaluation metric is the lap time. The baseline used from which comparing some of our approaches is the model that came with the default implementation of the simulator. This model had a reward which incentivized following the center line of the track (A3).

The definition of all the rewards used in this section (except the *time-sector* reward, which has been defined above) and the exact hyperparameters of the tests performed can be found in the Supplementary Material section (A and B).

5.1. Inputs and networks

To compare the performance of encoding the input images with a VAE versus not encoding them, we conducted a series of tests. They can be seen in Figure 4. The first test uses as input the raw images (in this case there are some Convolutional Neural Networks (CNN) substituting the encoder before the policy and value networks, which are also trained online). The second test uses the encoder part of a VAE with hidden size 32 to reduce the dimension of the raw images, with the policy and value networks of size [64,64]. The third test is the same as the second one with the networks of size [256,256]. The fourth test is the same as the third one but with a VAE with hidden size 64.

VAE usage and hidden dimension: As shown in Figure 4, using the encoder part of a VAE to lower the dimension of the input image before feeding it to the networks does have a positive impact on the performance of the model. Its stability is improved a lot, and the performance of the model does not change abruptly from one update to another. Moreover, the agent trained with a VAE stays on track most of the time after some training, while it is not the case for the model trained directly using the observed image. In terms of the VAE hidden dimension, a dimension of 32 is enough, there's no need to increase it, for example to 64.

Networks' sizes: As seen in Figure 4, the networks' sizes tested have been [64,64] or [256,256] for both the policy and value networks ([*hs*,*hs*] meaning two hidden layers of size *hs*). There's no apparent difference between agents with networks of these mentioned sizes, although from 150k steps it seems that bigger networks yield better results in terms of lap time.

5.2. Reward functions

To compare the performance of different reward functions, a fixed model and architecture has been used, varying only the reward. To determine whether the reward needs to include guidance, rewards 2 and 3 (A5 and A6), which only take into account the speed of the car and lap times, have been compared with the *baseline reward* (A3). The tests are shown in Figure 5. Additionally, to evaluate if a reward focused directly on minimizing lap time improves performance, the *time-sector* reward (1) and the two versions of the *progress reward* (A7, A8) have been compared with the *baseline reward*. The results can be seen in Figure 6.

Complex rewards: As shown in Figure 5, a reward that simply incentivizes positive speeds (such as *rew2* and *rew3*) is sufficient for the car to complete full laps without going off the track. There's no need for a reward with complex terms, such as the car's distance to the track's centerline (such as in the *baseline reward*) or the time taken to complete artificial generated sectors. The lap times achieved by

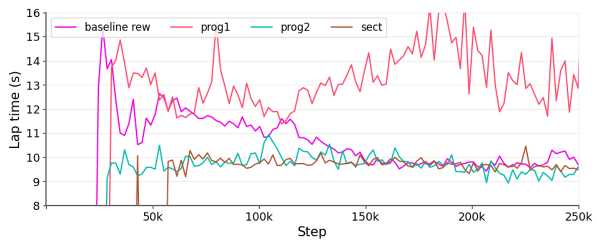

Figure 6. Testing lap time across steps for four equal models having different rewards. baseline rew (A3), prog1 (*progress reward 1*, A7), prog2 (*progress reward 2*, A8) and sect (*time-sector reward*, 1).

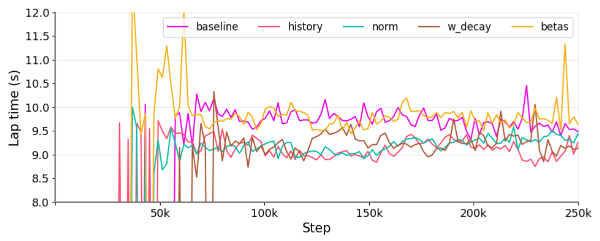

Figure 7. Testing lap time across steps for five models using the *time-sector reward* and different hyperparameters.

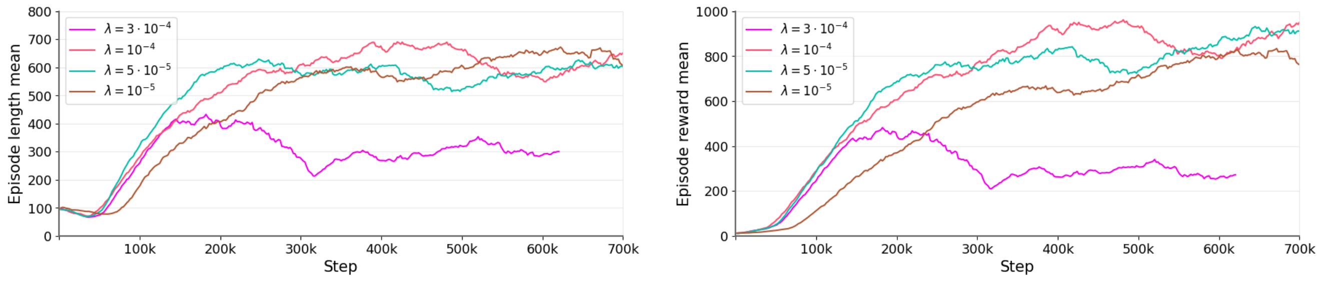

Figure 8. Episode length mean (left) and episode reward mean (right) for four different learning rates λ. $\lambda = 3 \cdot 10^{-4}$ was executed for less than 700k steps. The reward used here is the *time-sector* reward.

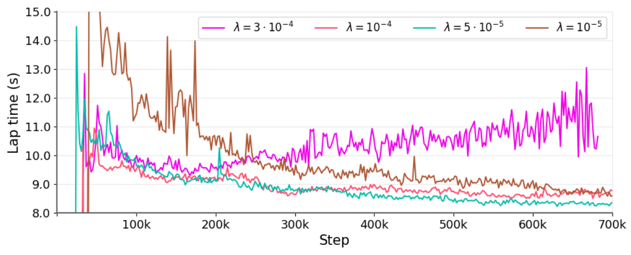

Figure 9. Testing lap time across steps for different learning rates λ. $\lambda = 3 \cdot 10^{-4}$ was executed for less than 700k steps. The reward used here is the *time-sector* reward.

the car after 250k steps are almost the same for all the rewards shown in Figure 5. However, during the first 100k steps, the model using the *baseline reward* performs much better. This is because the reward provides guidance, enabling the car to learn a fairly good trajectory more quickly. In contrast, the models with the other rewards require many more steps to learn an effective trajectory and speed.

However, rewards must be properly shaped. Otherwise, the car may fail to move from the starting position. This issue was observed when we tested *rew1* (A4). In this case, a small reward is given when advancing, while a very negative reward is given for going off the track. As a result, the car barely moved from the starting position, as it prioritized avoiding going off the track at all costs.

Moreover, comparing *rew2* and *rew3* in this same figure we can see that giving a reward as a function of the lap time (as both rewards do), when the car completes a lap, does not affect the performance of the agent. It is a very sparse reward, and because of the discounting factor, only states close to the starting line will benefit from it.

***Time-sector* reward:** As shown in Figure 6, our engineered reward, the *time-sector* reward, is an improvement in comparison to the baseline reward, mostly in the number of steps taken to achieve a competitive time. The performance of *progress reward 2* (A8) is very similar to the *time-sector* reward, although the speed term has been needed in the reward in order to achieve these results (*progress reward 1* (A7) does not use the speed term, and it is much worse, as it can be seen in the figure).

5.3. Hyperparameters and regularization methods

All the results presented have been obtained using default hyperparameters. In the result above we see that the *time-sector* reward is the one with the most potential, although there is still room for performance improvement. Notably, there is minimal improvement in the final training steps. Additionally, when examining the episode rollout mean and episode reward mean it appears that the agent may be experiencing *catastrophic forgetting*, as it can be seen with $\lambda = 3 \cdot 10^{-4}$ in Figure 8 (this is the default value of the learning rate used to train the networks of the PPO). This phenomenon occurs when the agent's networks forget previously learned information upon learning new information, leading to a collapse in performance. To address this issue, adjustments to certain hyperparameters may be necessary. Normalizing the input observations and rewards are some popular methods to overcome it. Moreover, as suggested in [1] and [4], incorporating weight decay can help mitigate catastrophic forgetting. Additionally, when using the Adam optimizer for training (such as in our case), adjusting its betas parameter value (betas are the coefficients used for computing running averages of gradient and its square) can also be beneficial.

In Figure 7 we compare the baseline model, with adding history with a horizon of 2, normalizing input observations and rewards, adding weight decay or changing the betas' value from the optimizer.

In Figure 9 we compare different values of the learning rate λ, keeping all the other hyperparameters constant. Moreover, in Figure 8 we compare the episode reward mean and episode length mean for these same tests.

Hyperparameters and Regularization methods: As shown in Figure 7, using history or normalizing the input observations and rewards leads to better performance compared to the baseline model. A similar improvement is observed with the addition of weight decay; however, in this case, the model requires more steps to complete full laps (approximately around 60k steps, whereas the other two versions start completing laps around 40k steps). Modifying the beta parameters of the Adam optimizer slightly deteriorates performance.

The most significant change comes from adjusting the learning rate. As shown in Figure 9, modifying the learning rate has a clear impact on the model performance. The default learning rate ($\lambda = 3 \cdot 10^{-4}$) becomes ineffective after around 400k steps, exhibiting catastrophic forgetting (as evidenced in Figure 8, where both the episode length mean and episode reward mean decrease after a certain number of steps). Decreasing the learning rate mitigates this issue. Striking a balance between excessively large and small values is crucial. As mentioned, large learning rate values lead to catastrophic forgetting while very small values (such as $\lambda = 10^{-5}$) create difficulty for the model to learn to complete laps initially, resulting in slower laps during the initial steps, as clearly shown in the figure. However, the smaller learning rates ($\lambda = 5 \cdot 10^{-5}$ and $\lambda = 10^{-5}$) are the ones that show greater potential, and that would likely continue to improve model performance with extended training.

6. Conclusions

In this paper, we have investigated novel reward functions and regularization methods for the task of autonomous racing on a track. Our proposed reward, the *time-sector* reward, achieves better lap times compared to the baseline reward, demonstrating its effectiveness in improving performance metrics in the autonomous racing domain.

Furthermore, our experiments validate the usage of using a VAE to encode input image observations, which provides a more robust representation for the learning algorithm. We have also highlighted the importance of adapting certain hyperparameters, particularly the learning rate, to enhance model performance and prevent issues such as catastrophic forgetting.

We hope our training framework will inspire future research to build on the presented knowledge, with a focus on generalizing these methods and transferring them to other environments.

Limitations: The primary limitation of this study has been the limited computational resources available. This constraint has significantly reduced the number of tests we have been able to conduct, thereby limiting the scope and generalizability of our findings. Greater computational power would have allowed for a more extensive exploration of hyperparameters, a broader range of experiments, and validation in more complex and diverse environments. Future research with enhanced computational resources can address these limitations, enabling a more comprehensive investigation and validation of the proposed methods.

Acknowledgements: This research received support during the Introduction to Research in Data Science Engineering course, Bachelor's degree in Data Science and Engineering, Universitat Politècnica de Catalunya.

References

[1] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study. *arXiv preprint arXiv:2006.05990*, 2020.

[2] Laurent Desegur. A lane detection approach for self-driving vehicles, 2017. Supplied as additional material `https://medium.com/@ldesegur/a-lane-detection-approach-for-self-driving-vehicles-c5ae1679f7ee`.

[3] DLR-RM. stable-baselines3. Supplied as additional material `https://github.com/DLR-RM/stable-baselines3`.

[4] Shibhansh Dohare, Qingfeng Lan, and A. Rupam Mahmood. Overcoming policy collapse in deep reinforcement learning. In *Sixteenth European Workshop on Reinforcement Learning*, 2023.

[5] Benjamin D. Evans, Hendrik W. Jordaan, and Herman A. Engelbrecht. Safe reinforcement learning for high-speed autonomous racing. *Cognitive Robotics*, 3:107–126, 2023.

[6] Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Dürr. Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning. *arXiv preprint arXiv:2008.07971*, 2021.

[7] Maximilian Jaritz, Raoul de Charette, Marin Toromanoff, Etienne Perot, and Fawzi Nashashibi. End-to-End Race Driving with Deep Reinforcement Learning. *arXiv preprint arXiv:1807.02371*, 2018.

[8] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[9] Antonin Raffin. Learning to drive smoothly in minutes, 2019. Supplied as additional material `https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb35f4`.

[10] Antonin Raffin. Donkeycar rl - hyperparameters study, 2022. Supplied as additional material `https://wandb.ai/araffin/donkeycar/reports/DonkeyCar-RL-Hyperparameters-Study--VmlldzoxODIyMDQx`.

[11] Antonin Raffin. Donkeycar simulator: Autoencoder training (part 2), 2022. Supplied as additional material `https://www.youtube.com/watch?v=DUqssFvcSOYt=5s`.

[12] Antonin Raffin. Donkeycar simulator: reinforcement learning tweaks (part 3), 2022. Supplied as additional material `https://www.youtube.com/watch?v=v8j2bpcE4Rg`.

[13] Antonin Raffin. Donkeycar simulator: Setup and reinforcement learning training (part 1), 2022. Supplied as additional material `https://www.youtube.com/watch?v=ngK33h00iBEt=8141s`.

[14] Adrian Remonda, Sarah Krebs, Eduardo Veas, Granit Luzhnica, and Roman Kern. Formula RL: Deep Reinforcement Learning for Autonomous Racing using Telemetry Data. *arXiv preprint arXiv:2104.11106*, 2022.

[15] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv preprint arXiv:1911.08265*, 2019.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[17] Yunlong Song, HaoChih Lin, Elia Kaufmann, Peter Dürr, and Davide Scaramuzza. Autonomous Overtaking in Gran Turismo Sport Using Curriculum Reinforcement Learning. *arXiv preprint arXiv:2103.14666*, 2021.

[18] Hanqing Tian, Jun Ni, and Jibin Hu. Autonomous Driving System Design for Formula Student Driverless Racecar. *arXiv preprint arXiv:1809.07636*, 2018.

[19] Ari Viitala. Scale model autonomous driving benchmark for deep reinforcement learning algorithms, 2021. Supplied as additional material `https://aaltodoc.aalto.fi/server/api/core/bitstreams/8856331e-1e9b-4123-adcf-a9c9fc704a4e/content`.

[20] Felix Yu. Train donkey car in unity simulator with reinforcement learning, 2018. Supplied as additional material `https://flyyufelix.github.io/2018/09/11/donkey-rl-simulation.html`.

[21] Qi Zhang, Tao Du, and Changzheng Tian. Self-driving scale car trained by Deep reinforcement learning. *arXiv preprint arXiv:1909.03467*, 2019.

Optimizing Autonomous Racing in the Donkey Car Simulator: Novel Reward Functions and Regularization Techniques

Supplementary Material

In this supplementary material, we show the engineered rewards mentioned in the work (which gave worse results than the *time-sector* reward) in Sec. A. We also provide more details about the tests carried and shown in the figures, in Sec. B.

A. Rewards

The *baseline reward*, which was already implemented in the simulator, is defined in the following way:

$$r_t = \begin{cases} -1 & \text{if off the track} \\ (1.0 - |cte|/max\_cte) \cdot v & \text{if } v > 0 \\ v & \text{otherwise} \end{cases} \tag{3}$$

Here, v is the forward speed of the car and *cte* is the Cross Track Error (distance of the car to the center of the track), which is given by the simulator itself. *max_cte* is the maximum value of *cte* we allow (the higher the more freedom we give to the car to go far from the center of the track, with default value of 8 in our simulator) and $|\cdot|$ is the absolute value.
Reward1, engineered by us, is defined in the following way:

$$r_t = \begin{cases} -100 & \text{if off the track} \\ (100/t_l)^2 & \text{if lap completed} \\ 0.1 & \text{if } v > 0.1 \\ -0.1 & \text{otherwise} \end{cases} \tag{4}$$

Here, t_l is the time it has taken to complete the last lap, and v the forward speed of the car.
Reward2 is a modification of *reward1*, defined as:

$$r_t = \begin{cases} -10 - v & \text{if off the track} \\ (100/t_l)^2 & \text{if lap completed} \\ 1 + v & \text{otherwise} \end{cases} \tag{5}$$

Reward3 is a modification of *reward2*, defined as:

$$r_t = \begin{cases} -10 - v & \text{if off the track} \\ 8 \cdot (20 - t_l) + 50 & \text{if lap completed} \\ 1 + v & \text{otherwise} \end{cases} \tag{6}$$

We also engineered the *Progress reward 1*, inspired by [6]:

$$r_t = \begin{cases} -10 - v & \text{if off the track} \\ p_t & \text{otherwise} \end{cases} \tag{7}$$

Here, p_t is the progress (in %) the car has made from the last step to the current one.

Also *Progress reward 2*, a modification of the reward above:

$$r_t = \begin{cases} -1 & \text{if off the track} \\ p_t \cdot v & \text{otherwise} \end{cases} \tag{8}$$

v being the forward speed of the car and p_t the progress (in %).

B. Tests Details

In this section we will present the different hyperparameters used to carry the tests shown in the figures.

B.1. Figure 4

Model	CNN	VAE32, [64,64] nets	VAE32, [256,256] nets	VAE64, [256,256] nets
use VAE	False	True	True	True
VAE hidden dimension size	-	32	32	64
network architecture	[64,64]	[64,64]	[256,256]	[256,256]

Table 1. Hyperparameters for Figure 4 tests.

The learning rate used is $\lambda = 3 \cdot 10^{-4}$ and the reward the *baseline* reward.

B.2. Figure 5

Model	rew2	rew3	baseline rew
reward used	Reward2 (A5)	Reward3 (A6)	baseline reward (A3)

Table 2. Hyperparameters for Figure 5 tests.

The learning rate used is $\lambda = 3 \cdot 10^{-4}$, a VAE is used with hidden size 32 and the networks have size [256,256].

B.3. Figure 6

Model	baseline rew	prog1	prog2	sect
reward used	baseline reward (A3)	Progress1 (A7)	Progress2 (A8)	*time-sector* reward (1)

Table 3. Hyperparameters for Figure 6 tests.

The learning rate used is $\lambda = 3 \cdot 10^{-4}$, a VAE is used with hidden size 32 and the networks have size [256,256].

B.4. Figure 7

Model	baseline	history	norm	w_decay	betas
use history	False	True	False	False	False
normalize	False	False	True	True	True
weight decay	0	0	0	0.001	0
betas	β	β	β	β	(0.9,0.9)

Table 4. Hyperparameters for Figure 7 tests. $\beta = (0.9, 0.999)$.

The learning rate used is $\lambda = 3 \cdot 10^{-4}$, a VAE is used with hidden size 32, the networks have size [256,256] and the reward used is the *time-sector* reward. When history is used, it is done with a value of 2 (that is, taking the current and past observations as input). The normalization implies normalizing both the input image observations and the rewards.

B.5. Figures 8 and 9

The learning rates used for each model are specified in their names. A VAE is used with hidden size 32, the networks have size [256,256] and the reward used is the *time-sector* reward. History is used with a value of 2, and both the input image observations and the rewards are normalized.

B.6. Constant hyperparameters

Some hyperparameters used by our agents, whose value has remained constant throughout all the tests are:

- batch size = 64
- entropy coefficient = 0
- gamma = 0.99
- GAE (Generalized Advantage Estimation) lambda = 0.95
- clip range = 0.2
- number of epochs = 10
- rollout buffer length (number of steps before performing SGD) = 2048
- value function coefficient = 0.5