University of Belgrade - Faculty of Electrical Engineering

Department of Signals and Systems

THESIS

Machine learning application for autonomous racing in video games

Candidate
Andrej Gobeljić

Mentor
Ph.D Goran Kvaščev, associate professor

Belgrade, *July* 2022.

ABSTRACT

Autonomous racing, as a special type of self-driving, represents an extremely demanding problem in the field of artificial intelligence, because it consists of a large number of subtasks such as lateral control, longitudinal control, path panning, speed and braking control, etc.

This work presents a collection of simplified examples of agents trained by methods of supervised learning and reinforcement learning, which are able to complete a given track.

Some aspects of the learning system, reward functions, and neural network parameters are based on the paper *Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning*, which demonstrated an agent capable of nearly perfect results in a racing simulation [2]. This work, along with an agent based on it, showed that with the help of this approach it is possible to create extremely fast and skilled self-driving cars.

The goal of this work is to develop a miniature version of a self-driving vehicle in a well-known video game. To track progress during training, the agent's success is recorded on different tracks, as well as the duration of each training episode.

CONTENTS

ABSTRACT 2

CONTENTS 3

1 INTRODUCTION 5

2 OVERVIEW OF EXISTING SOLUTIONS 6

3 SYSTEM DESCRIPTION 7

3.1 Trackmania 7

3.2 Software used 9

4 SUPERVISED APPROACH 10

4.1 Inputs and outputs 10

4.2 Perception 11

4.3 Network architectures 12

4.4 Advantages and disadvantages of this approach 14

5 REINFORCEMENT APPROACH 16

5.1 Details of the training algorithm 16

5.2 Environment, inputs and outputs 16

5.3 Reward function 18

5.4 Reward Hacking and Delayed Markov Process 21

5.5 Termination conditions and overfitting avoidance 22

5.6 Environment parameters 23

5.7 Hyperparameters 24

6 RESULTS 26

7 CONCLUSIONS 32

8 REFERENCES 34

1 INTRODUCTION

Today, autonomous driving represents one of the most well-known and common applications of artificial intelligence. From simple driving aids to fully autonomous vehicles, this type of control affects both increased traffic safety and more comfortable transportation of a larger number of passengers.

Given that new technologies are being developed through motorsport, which will later be used in commercial cars, the idea of autonomous racing is emerging as a potential platform for the development of artificial intelligence algorithms. This concept combines the development of the vehicles themselves and their self-driving algorithms and, in this way, represents an ideal field for the development of the entire self-driving system.

The problem that is being solved in this paper is the training of an agent for autonomous racing in a video game called Trackmania in order to achieve the best possible times on the given tracks. The supervised agent is trained using classic gradient descent methods, while the reinforcement agent is trained using the SAC algorithm (Soft Actor-Critic) [1], one of the most popular reinforcement learning algorithms today. In order to better measure the success of the second agent, the distance traveled and the intensity of contact with the walls are measured.

By using the methods described in the later chapters of this paper, solid self-driving algorithms were obtained that are capable of completing the given track configuration, and which can be additionally trained and improved in order to achieve better results. Bearing in mind that this project is a proof of concept and that for much better results, much more training and hyperparameter tweaking is necessary, realistic expectations are that the times achieved by the agent will be significantly lower than the average human driver. Therefore, the aim of this paper is to describe the system for training agents for autonomous racing, review the results of the agents and analyze the possibilities for improvement.

2 OVERVIEW OF EXISTING SOLUTIONS

Previous works in the field of autonomous racing can be grouped based on the applied artificial intelligence algorithms: classical approaches, supervised training approaches, and reinforcement learning approaches [2].

Classical approaches decompose the autonomous racing problem into three parts: perception, path planning and control [2 - 5]. A Model Predictive Control (MPC) represents a promising approach for driving vehicles at high speeds [4 , 6 - 10]. A similar approach called Model Predictive Path Integral Control (MPPI) [3,11,12] can additionally be combined with neural networks. However, although both approaches have shown impressive results, the requirements for high parallelization of computations and the lack of flexibility in designing reward functions represent significant limitations for the application of such methods.

Approaches using neural networks, compared to classical control approaches, have significantly less real-time computation time and do not suffer from the enormous complexity of the model with increasing problem complexity. Additionally, neural networks can unify perception, path planning and vehicle control into a single problem, i.e. these three tasks do not have to be explicitly separated within the solution. With supervised learning, it is possible to create an agent that replicates the actions of a human driver. One of the first autonomous driving systems that uses this approach is ALVINN (Autonomous Land Vehicle in a Neural Network)[13]. Great success has also been achieved with convolutional neural networks for lane tracking [14]. This approach overcomes the problems of classical approaches, but the agents are limited by the quality of the data on which they are trained [2].

Unlike other types of machine learning (Behavioral cloning, Generative-Adversarial Imitation learning, Supervised learning, Neuroevolution), Reinforcement learning often shows better results, for the simple reason that the agent learns on its own, while in other situations it is limited by the capabilities of the human driver from whom it receives data. With this approach, the agent can find better routes that human drivers are unaware of. Applying model-free deep learning with reinforcement, numerous studies have demonstrated the success of such approaches [15]-[19]. Recently, researchers from the University of Zurich, in cooperation with the Sony company, developed the first control system in the game Gran Turismo Sport that achieves superhuman results with the help of reinforcement learning [2] and thereby further confirmed the success of such training techniques.

3 SYSTEM DESCRIPTION

For the reasons stated in the previous chapter, this paper primarily deals with the application of reinforcement learning. However, in this miniature scenario, the results achieved by the supervised agents, the ease of training and the flexibility of processing the training set are not negligible, so this approach will also be addressed.

3.1 Trackmania

All the algorithms explained in this paper were trained and tested on the video game Trackmania [20] because of the faster and easier overall process of learning and recording the results. In addition, Trackmania offers a very good track creator, with which it is possible to create various tracks, which is an ideal training ground for agents and comparison with human drivers.

Figure 1 - View from the game Trackmania

Tracks are created from blocks, which leads to easier construction of various track configurations, easier determination of boundaries and center lines, and efficient tracking of cars while driving. Also, the paths can be very complex, which contributes to the variety of combinations of curves, which is beneficial, because both approaches require a very large number of situations based on which they need to learn. This means that algorithms can be trained without frequent track changes and thus reduce training time, without the risk of overtraining.

For the purposes of training and testing the agents, the track configurations shown in the following images were used.

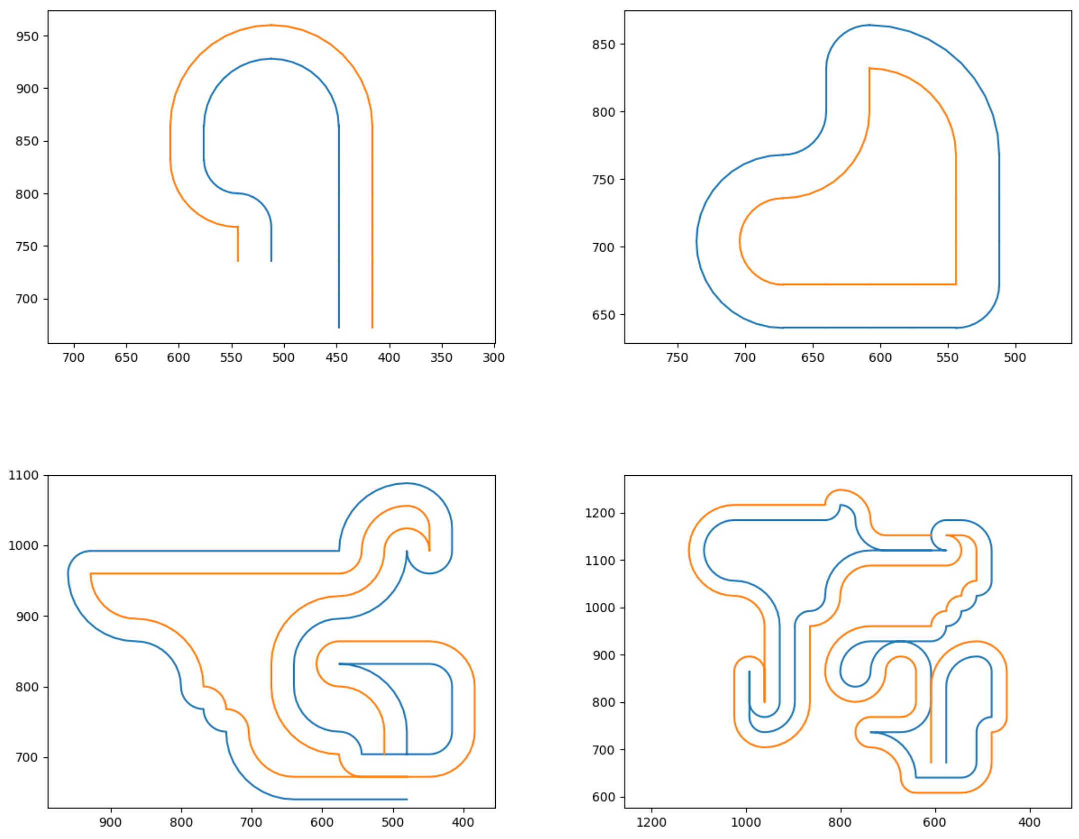


Figure 2- Track view

3.2 Software used

The programming language used is *Python* [21] due to its great flexibility, ease of use and support for machine learning through popular packages (such as *Tensorflow* [22] or *Pytorch* [23]). The platform on which the project was developed is *Windows 10.* The scripts that run the agents simulate keyboard and joystick presses, which provides easier integration with the game (no additional settings are necessary to send actions to the car in the game). However, agents need information about the speed of movement (and the supported agent uses additional input parameters), so they require additional tools that can extract state details from the game itself. The *Openplanet* plugin was used in this project [24] through which it is possible to send the car's current location on the track, current speed and other information to scripts. The open-source tool GBX-Net was used to extract track details from individual files [25].

One of the main advantages of supervised training versus reinforcement training is the complete separation of the training process from the training set collection. The entire set, together with training scripts, can be sent to specialized servers in order to train neural networks faster and easier. In this project, *Google* 's platform called *Google Colab* [26] was used. This platform enables the use of significantly more powerful machines for training agents in the shortest possible time. Unlike supervised networks, reinforced neural networks interact with the environment and gradually improve the results. Therefore, these agents collect a training set during the training itself and run on the same computer where the video game is running. Although the system setup is easier, the work of the neural networks is significantly slowed down and the training process itself takes longer. This can be circumvented by using another device to run the game with an intermediary Python script that would exchange information about actions and current states between devices. It is also possible to implement a distributed environment with a larger number of computers, where for the same period, many more trajectories would be collected than in the case of a single device. This approach is, however, extremely demanding and would represent a project itself, and because of that it is listed here as a good idea for improvement.

4 SUPERVISED APPROACH

Supervised learning is tasked with generalizing input data and finding correlations between different scenarios. In order to apply this method to solve this problem, it is necessary to define the input data so that it reflects the current state of the track as precise as possible. It will be processed in two ways:

- image method using *convolutional neural networks*, and
- sensor method using classical *feedforward* networks.

4.1 Inputs and outputs

None of the mentioned methods require special tools in order to communicate with the game, which makes training even easier. The main method of agents' perception is the visual representation of the situation on the track. Both systems use *screenshots* that are additionally processed before being passed through the neural network. The training set used in this project contains 6130 images of the car's front camera.

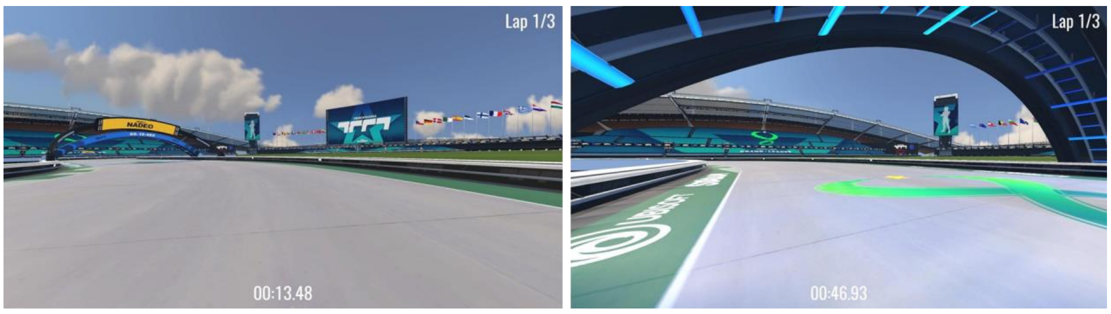


Figure 3- Display of input data before modification

Apart from the input data, it is necessary to define how the neural networks will control the car itself. During the implementation of the supervised approach, the main way to control the car during data collection was via the keyboard. For this reason, along with screenshots, discrete keyboard presses were recorded in the form of four bits:

$$\frac{0101}{\text{right, left, brake, throttle}}$$

In the example above, the action the agent would take is to turn left with full throttle. As neural networks handle real numbers instead of integers, the output of the neural network will be four real numbers in the range $[0, 1)$. These numbers represent the probabilities of pressing a particular key. For simplicity, instead of emulating a joystick (which will be implemented in the reinforcement learning approach), a threshold will be used to determine whether a certain action will be performed or not.

4.2 Perception

The first method (convolutional) compresses the screenshot and eliminates all details except the track limits (*Figure 4*). Although the approach is minimalistic, it has been shown that this kind of network is trained relatively quickly and that the results are acceptable.

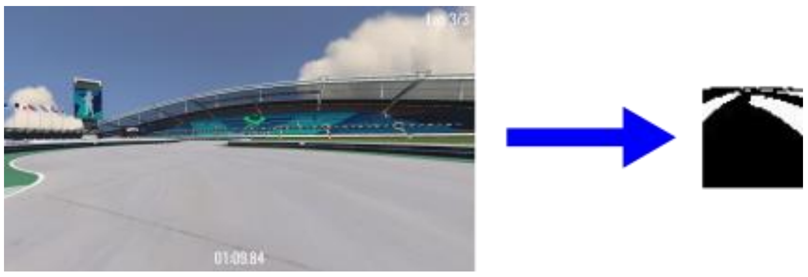


Figure 4- Screenshot transformation representation for convolutional network input

Because the second method cannot accept an image, it is necessary to perform certain measurements that will represent the input. The process is somewhat similar to the previous one, with the additional measurement of distances between the bottom of the image and the first black pixel (*Figure 5*). Unlike the previous method, this method prepares the input much faster, at the cost of less information.

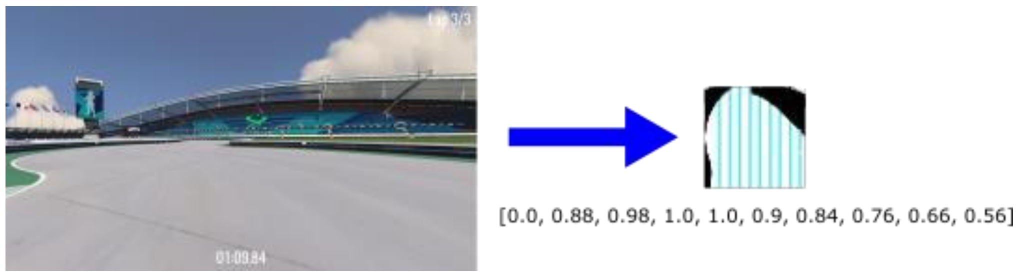


Figure 5- Screenshot transformation view for deep network input

Given that the speed of image processing and passing through the neural network plays a crucial role in system design, it is necessary to make image preprocessing as efficient as possible. If the system has too long a delay between two actions, the agent will not be able to control the car adequately. For this purpose, the *OpenCV* [27] and *Pillow (PIL)* [28] libraries were used, today standardized libraries for programmable image manipulation, which enable easy and fast application of the above-mentioned transformations.

However, the question arises how the agent can conclude with what intensity to turn or brake based on only one image? A car can move at different speeds in the same or similar curves and thus make the training set confusing for the neural network. It is necessary to deliver information to the speed of movement in an adequate way that will not increase the complexity of neural network architectures too much. The simplest method is to simply add another input neuron to the first layer. It is very easy to implement in the second approach, given that it is a classic network, while in the first case it is more complicated. The speed is passed as a single number, which is not possible to fit into the image passing through the convolutional layers. There is a need to make the neural network non-sequential.

4.3 Network architectures

The final architecture used for convolution is shown in the figure (Figure 6). After all the convolutional and regularization layers there is a layer for concatenating two independent parts of the neural network. This ensures that the agent can extract all the necessary details from the image, and at the same time take into account the speed of the car in an efficient way.

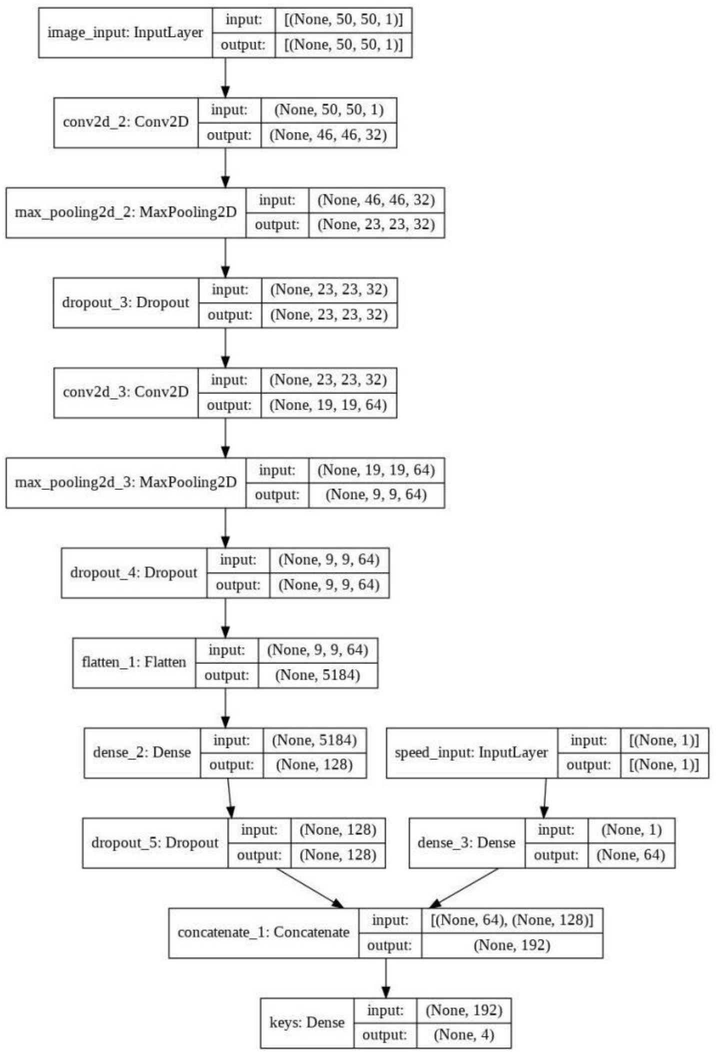

image_input: InputLayer | input: [(None, 50, 50, 1)] | output: [(None, 50, 50, 1)]
conv2d_2: Conv2D | input: (None, 50, 50, 1) | output: (None, 46, 46, 32)
max_pooling2d_2: MaxPooling2D | input: (None, 46, 46, 32) | output: (None, 23, 23, 32)
dropout_3: Dropout | input: (None, 23, 23, 32) | output: (None, 23, 23, 32)
conv2d_3: Conv2D | input: (None, 23, 23, 32) | output: (None, 19, 19, 64)
max_pooling2d_3: MaxPooling2D | input: (None, 19, 19, 64) | output: (None, 9, 9, 64)
dropout_4: Dropout | input: (None, 9, 9, 64) | output: (None, 9, 9, 64)
flatten_1: Flatten | input: (None, 9, 9, 64) | output: (None, 5184)
dense_2: Dense | input: (None, 5184) | output: (None, 128)
speed_input: InputLayer | input: [(None, 1)] | output: [(None, 1)]
dropout_5: Dropout | input: (None, 128) | output: (None, 128)
dense_3: Dense | input: (None, 1) | output: (None, 64)
concatenate_1: Concatenate | input: [(None, 64), (None, 128)] | output: (None, 192)
keys: Dense | input: (None, 192) | output: (None, 4)

Figure 6- Convolutional network architecture

As the second neural network does not suffer from the need to extract image features, but receives information from the distance of the walls, the simplest way to account for speed is to simply add an input neuron. However, the idea is to treat distances differently from speed. The final appearance of the neural network is shown in the picture (*Figure 7*).

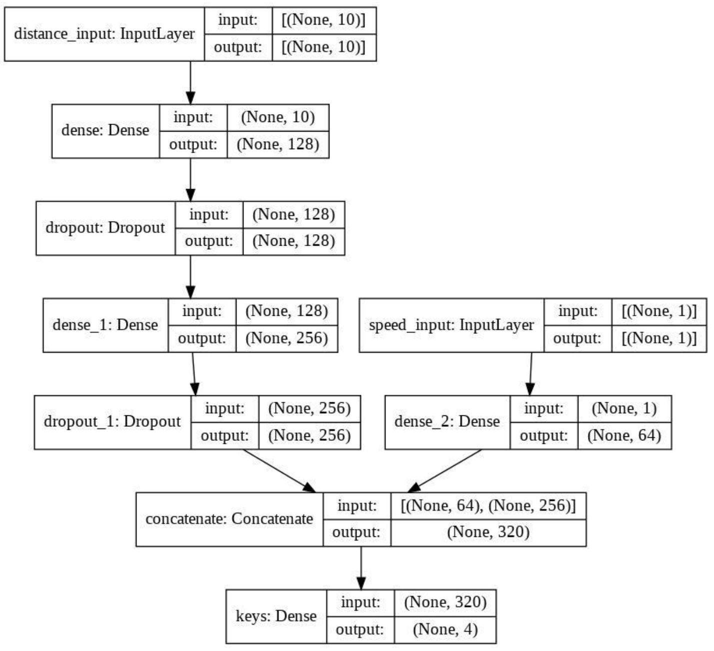


Figure 7- Classic network architecture

This architecture also contains a concatenation layer, followed by an output layer with four neurons.

As stated in this chapter, both networks will output four real numbers that will be discretized afterwards. It has been experimentally determined that a higher threshold gives more stable management and better-quality results. For both networks, a value of 0,9999 was taken. At first glance it seems that this value leads to a decrease in the agent's reactivity. However, it appears that an agent with a lower coefficient applies actions that are not adequate at certain moments. With a high threshold, only actions that the agent is, conditionally speaking, sure to be good will be applied.

4.4 Advantages and disadvantages of this approach

One of the biggest drawbacks of the supervisory approach to training autonomous vehicles is tracking the success of the training. Through training, the agent tries to imitate as best as possible the drivers from which it receives data. Classical supervision problems contain training and test sets that are static and where it is clear how good some output is. In this case, the success of the network on the training and test sets solely shows how well the network has learned the moves of other drivers. During experimentation, it was noticed that networks with a success rate of 60% to 70% on the test set were very good according to the results achieved on the track. Also, human drivers, based on whose drives the training set was created, are not perfect. This causes that in certain cases it is not even desirable for the agent to learn to apply identical moves that can be found in the training set. Moreover, if the set is too large, it becomes impossible, because it can occur that in two almost identical situations two drivers applied different actions. This problem causes the training performance of these networks to be less relevant to measuring driving performance. Clearly, networks with a 60% success rate will be faster than networks with a 30% score because they generalize better. However, determining the relevance limit is an extremely complicated problem that exceeds the scope of this project and will not be addressed further. Neural networks are trained over multiple trials and intermediate states are saved to select the best agents. On average, all neural networks would have a success rate of approximately 90% on the training set and 70% on the validation set over a period of 3000 epochs.

As mentioned, this method imitates the moves of human drivers whose trajectories it uses as training data. This means that the agent always tends to drive fast, which leads to a lot of contact at the edges of the track. However, since in this simulation there is no damage to the vehicle upon contact, the end result is an aggressive driving style that achieves acceptable results. The main drawback of such a system is that the agents are not aware of their location on the track, nor of the next curves. As a downside, the system has no memory to remember the configuration of the track, or at least its portion. Therefore, there are numerous possibilities for improvement.

5 REINFORCEMENT APPROACH

This type of machine learning consists of two parts:

- implementation of the algorithm that will be used to train the agent, and
- implementation *of the gym environment* which formulates the rules by which the agent can function, the rewards it receives during training and the states in which the agent is currently located.

5.1 Details of the training algorithm

For the training algorithm, Soft Actor-Critic was selected as one of today's most famous algorithms of this type of training due to its success in various problems [1], even in autonomous driving [2]. Due to the almost perfect results achieved by this algorithm in the paper *Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning*, this work will base certain elements of the solution on it.

With the help of the *Stable Baselines 3 package* [29] for the Python language, the implementation of the algorithm is very simple, and the adjustment of the training parameters is done with ease. This package is based on *PyTorch*, Facebook*'s* [23] framework for neural networks. The developers of the SB3 package tried to make all the algorithms as close as possible to the original works, so the algorithm used in this project is standardized and adapted to the Python language.

5.2 Environment, inputs and outputs

In reinforcement learning, *actions* and *observations*, along with *the reward function*, are part of the training environment, the second building block of this system. The environment communicates with the video game and delivers the necessary data to the agent. One action-state pair is called *a timestep*, and several steps make up one *training episode.* Training is usually done in a certain number of time steps, and the ultimate goal is to maximize the reward at the end of the episode. The end of an episode is defined separately, and there will be more discussion about this in the following chapters.

Considering that SAC deals with *continuous* actions [1], it is necessary to precisely define how the agent controls the vehicle. The outputs used in the supervised approach can be applied here as well, but there is a problem in determining the turning priority. If the signal outputs for both directions are very small, but still not negligible, it is not clear which signal should be given priority, nor if turning is a good option in that situation. The first idea is to use three outputs: throttle intensity, braking intensity, and steering angle. Based on the research of the driving of human players, it is concluded that the gas and the brake will almost never be used at the same time [2]. Therefore, it is possible to combine the throttle and brake signals into one, resulting in a final output of two real numbers with bounds $[-1, 1]$.

In this case, the input to the neural network is the aforementioned observation from the training environment. The formulation of the state the agent is in should contain enough information for the training to be meaningful. However, too much information can lead to over-sensitivity to certain parameters and ignoring others, even total training failure. The supervised approach was not aware of its position on the track, and this information, along with the prediction of the next curves, can greatly improve the algorithm itself. Using the research from the mentioned paper [2], it can be concluded that the following parameters are the most effective starting point for training:

- normalized measurements of the distance from the car to the walls of the track (seven measurements were used in this project, evenly distributed in $180°$) represented by real numbers in the range $[0, 1]$,
- speed intensity in the range $[0, 1]$,
- the angle between the vector of the direction of movement of the car and the vector of the tangent to the central line of the track at the point of the normal projection in the range $[-\pi, \pi]$,
- steering signal in the previous action in the range $[-1, 1]$,
- wall contact (0 or 1),
- the curvature of the next portion of the track in the range $[0, 1]$.

The number of sensor measurements can be changed to improve performance. Almost all inputs are normalized (reduced to the range $[0, 1]$) because most *reinforcement learning algorithms* rely on a *Gaussian distribution* for continuous actions [29]. If actions are not normalized, there is a danger of bad training, the cause of which is difficult to detect later [29].

Considering that this is a miniature version of a self-driving system, and taking into account the limited computing power of the devices on which the algorithms were run, some input parameters have been simplified. For example, in this work, the curvature is calculated as the inverse radius of a circle defined by three points on the center line of the track, the first of which is the projection of the location of the vehicle onto the center line. The distance of the other two points can be adjusted between two independent trainings. The picture (*Figure 8*) shows the operation of the sensor and finding points on the central line for measuring curvature.

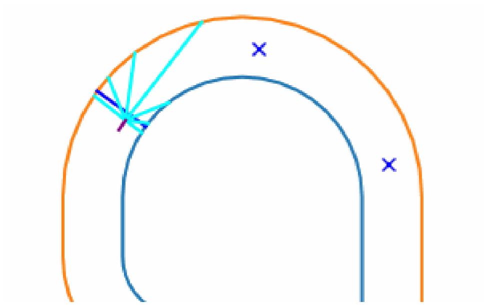


Figure 8- Display of sensors and curvature calculations

The main idea is that curvature provides enough information about subsequent curves, without slowing down and overcomplicating the algorithm. In this way, the algorithm gains awareness of the current situation on the track and can make a better preparation for entering the next curves.

5.3 Reward function

The core of the training environment is *the reward function*. This feature should give the agent insight into how good his currently learned policy is. When an agent takes an action that leads to a solution, it will receive a positive reward. Otherwise, the reward function should return a null value, or negative in case the algorithm makes big mistakes that it should learn to avoid. The process of creating the reward function is called *reward engineering*.

For this project it is necessary to define what exactly is required from the agent. The main goal is, obviously, to complete the course in the shortest possible time. However, the environment cannot provide information about the passing time before the end of the track itself. Such rewards are called *sparse* because they prevent a precise insight into individual mistakes made by the agent during training. As a solution to this problem, it is possible to make more *checkpoints* where intermediate times can be measured and delivered to the agent at more frequent intervals. This view also has a problem, because all agents aim for maximization, while here the goal is to minimize the passing time. A naive approach would be to simply return a negative value of the time, but this leads to the agent not wanting to pass through those points in order to not get negative rewards at all. Although there is room to improve this approach, it is still limited by the number of checkpoints that are distributed along the track. The more points, the slower the system. The fewer points, the more diluted the reward becomes and the harder it is to train. The basic gym environment that is applied for the simplest two-dimensional examples of self-driving vehicles is the following (N indicates the total number of track blocks) [30]:

$$-0.1 + \begin{cases} 1000/\mathrm{N} & \text{if the agent crosses a block} \\ 0 & \text{otherwise} \end{cases}$$

In this way, the agent is stimulated to complete the track as quickly as possible. The first value represents the base penalty, which additionally encourages completing the episode in as few steps as possible. For this example, this reward function is quite sufficient, but for any more realistic scenario, the need for a better definition arises. Following professional drivers, the idea of implementing “racing lines” becomes mandatory. This trajectory, combined with timely braking and acceleration, is theoretically ideal for achieving the best times on a given track. In order for the agent to learn to drive along the racing line and achieve the best results, the reward function should be constructed in a way that best evaluates the short-term and long-term effects of the actions applied by the learned policy.

Finding inspiration in the paper *Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning* and analyzing the tests presented in it, the following idea becomes promising: centerline progress tracking [2]. By measuring the distance between normal projections of the car’s location on the centerline between timesteps, an adequate metric is obtained with which progress on the track can be measured. If the agent approaches the curve optimally, he will cross a larger portion of the central line in a shorter time interval, while if he drives on the outer part of the curve, he will need more time and the reward will be smaller. This feature, however, gives the same reward to all rides that complete the course, leading to slower results. In this paper, the concept of *cutting episodes* is proposed which, instead of an episode spanning the entire track, limits episodes to small sprints of equal time intervals. When not using episode cutting, a school-like concept with a base penalty at each time step was applied [30]. In both cases, the agent gets an incentive to drive faster in order to cover a greater distance in a given time period and thereby increase the reward. The idea of multiplying the reward by speed was also considered, which would make the agent want to both drive fast and approach corners better. The work mentioned in this chapter uses the exponential *discount factor* which later reduces rewards, which also leads to an incentive for higher speeds [2]. However, much experimentation with various coefficients is necessary to obtain a good factor and this parameter will be left at the default value.

After ensuring that the agent tends to drive fast, it is also necessary to ensure that it avoids contact with walls. By cutting episodes and applying a discount factor, it forces the exploitation of short-term rewards, leading to less braking and more crashes. In order to neutralize this phenomenon, a second part of the reward function is added, which gives a *penalty* for each contact. After analyzing the experiments in the mentioned paper, the best way would be to subtract the square of the speed at the moment of collision multiplied by the empirically determined coefficient [2] in case of contact. This can also be formulated as kinetic energy, and the justification for its use in this case is found in the energy-dependent acceleration loss upon collision with the wall. The final reward function looks like this:

$$p_t - p_{t-1} - \begin{cases} cv^2 & \text{if in contact with a wall} \\ 0 & \text{otherwise} \end{cases} + \begin{cases} b & \text{if using base penalty} \\ 0 & \text{otherwise} \end{cases}$$

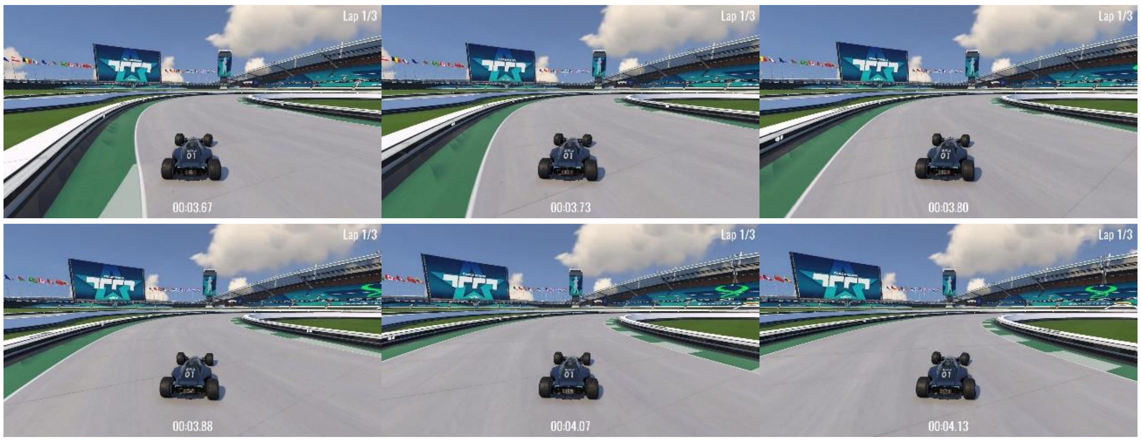


Figure 9- Consequence of reward function, better corner entry

5.4 Reward Hacking and Delayed Markov Process

A problem that often arises in reinforcement learning is finding unexpected ways to obtain high rewards that are inconsistent with the desired goal that the agent needs to achieve. This problem is intuitively called *reward hacking*. Specifically in the autonomous control problem, depending on the formulation of the reward function and the termination conditions, it can occur that the agent stops spontaneously. Due to the excessive penalties that he receives (in contact with the walls or base penalties), he finds a strategy to get as little penalty as possible, that is, to maximize the reward, which boils down to ending the episode as quickly as possible. It happens in some cases that the agent intentionally crashes, in order to cause the restart of the episode and the time spent on the track as short as possible. The reward function that was presented in the previous chapter effectively overcomes these problems, because driving achieves a sufficiently large reward and it is always optimal to drive on the track in a targeted manner, and the coefficients for contacts with the walls are within optimal limits.

Often, problems that are solved by reinforcement learning methods do not take place in real time for the reason that each environment should represent a *Markov decision process*. As the Markov process is a discrete-time process, there is a difficulty in applying these methods to real-time problems. In this case, the concept of the *Delayed Markov* process is considered. Real-time problems, such as autonomous driving, break the *Markov assumption* that the probabilities of future states depend only on the current state. During training, there is a gap between accepting an observation and sending an action, which means that the next state is conditioned by this delay. For turn-based games, this is not a problem, as the environment stagnates while a decision is being made. The delay depends on several parameters, but mostly on the time required to process the observation. As mentioned in the previous chapters, the agent is trained on the same computer where the game is running, which leads to longer input processing times and higher delays. The consequence of this problem is, in addition to slower reactions, the danger of poorly learned associations between states. If the spacing is too large or uneven, the algorithm will get different information by applying the same actions for the same conditions. The simplest solution to this problem is to provide a history of observations with each new observation [29]. Although there are several solutions, considering that in this project the system has short and relatively uniform delays, this problem will be ignored. Although it sounds naive, ignoring delays is one kind of solution, because some algorithms are able to learn to generalize states with delays and thus overcome this problem. In addition, the current speed and previous steering angle are included in the observation frame, which allows the agent to take the appropriate action independently of the delay.

5.5 Termination conditions and overfitting avoidance

In this work, episodes end when an agent successfully completes a path. As mentioned earlier, in order to allow the agent to have an incentive to drive faster, the technique of *cutting episodes* was applied. This means that, except when completing a track, episodes end for a certain amount of time. This period is the same for one agent, in order to track success during training. Also, if the agent stops for any reason (due to noise during information transmission, the car is considered stopped if it has a normalized speed value lower than $0{,}005$), it is necessary to end the episode and start driving again. This is important because the agent does not have the ability to go backwards (the goal is to drive as fast as possible, so the reverse drive is ignored) and in frontal contact with the wall it remains stuck. Also, due to the specific way of training, there is a possibility that the car stops by itself, which prolongs the training and does not lead to optimal results. In order to circumvent these problems, the aforementioned condition for restarting episodes was implemented. It is important to note that, unlike ending an episode because of a crash, ending an episode due to a timeout does not cause the agent to return to the beginning. This allows the agent to still be able to learn the entire track even though in the early stages of training it is not fast enough to reach the later parts of the track. Among other things, applying this method avoids retraining to a certain extent.

Bearing in mind that races in *Trackmania* works in a way that the agent cannot be placed at a random location on the track but will return to the beginning on each restart (there is a way to return to checkpoints, but this requires rebuilding the existing system and takes more time for each episode), there is a great danger of overfitting on the initial parts of the track. Despite the episode cutting described in the previous paragraph, the car still returns to the beginning of the track when it crashes or stops, so overfitting is still present. In the early stages of the project, this problem manifested itself with the agent going through the first curve very well, while on the second it would get completely confused and eventually crash. A trick used to prevent overfitting is to flip the actions and observations on each subsequent episode. This mechanism gives the appearance that the agent is driving on two symmetrical tracks, rather than one. This trick not only prevents overtraining, but effectively doubles the diversity of the training set that the agent collects and thus improves performance. Given that the implementation of this trick is very simple, and that action and observations flips do not take much time, the computational cost is almost non-existent. Therefore, all agents in this project will contain this trick. The problem of overtraining to the beginning of the track will still exist, but to a much lesser extent. Starting the episodes at random locations on the track would solve the overfitting problem entirely, but for the technical reasons mentioned it will be left as an improvement idea.

5.6 Environment parameters

In this project, there are several environment parameters that can be changed to improve training:

- length of episodes,
- the intensity of the penalty coefficient for wall contacts,
- the number of sensors that measure the distances from the car to the walls,
- centerline distance between the points from which the curvature is calculated and
- coefficients by which the reward is multiplied (it can be a speed or a discount factor).

By increasing the duration of the episode, long-term rewards are favored, which is better for the algorithm. This value can be increased during the training itself, so that the algorithm first learns how to drive, and then how to be as fast as possible. On the other hand, too long episode duration can cause reward convergence, which again raises the problem that every episode will have the same return value. For these reasons, it is necessary to determine the optimal duration of episodes. If the episodes are not shortened, but the end of the episode is determined by passing the goal, it is mandatory to define some factor by which the reward will be multiplied. With a sufficiently high factor, the total track time is most closely approximated [2], but a lot of experimentation is still necessary in order to determine its value. If speed is a factor, the algorithm learns very quickly that giving throttle is always a good option, but finer moves and better entrances to curves remain somewhat neglected.

Penalizing contacts with walls is not a mandatory element, because over time the algorithm will realize that there are faster trajectories that avoid collisions and thus learn to drive better. However, the application of penalties greatly speeds up training and forces the algorithm to move away from the edges of the track from the start. This parameter must also be adjusted carefully, because too small coefficients lead to ignoring penalties and wall grinding. On the other hand, too big coefficients cause the agent to have no desire to drive fast in order not to crash and receive a negative reward [2].

As long as the agent has a sufficient number of sensors for a clear perception of the path, it is not necessary to add more. As there is a danger of cluttering the networks with too much information, the number of sensors is left at seven. This does not mean that the agent will not drive better with more sensors, so it is necessary to test multiple drives with different numbers of measurements to determine the optimal number of sensors.

Since, along with the supervised method, this method also does not have any memory mechanism to remember the path, the only way to give any insight into the next curves is through the input parameters, with the key element being the curvature factor. This factor is conditioned by the size of the distance between the points on the central line. The larger the gap, the larger portion of the track will be covered and the algorithm will have a better overview of the track. However, this can lead to looking too far ahead and drawing the wrong conclusions about the curve just ahead. On the other hand, if the gap is too small, the agent will only focus on the configuration immediately ahead and will not have adequate exit from the first curve and preparation for the next curve.

5.7 Hyperparameters

Apart from the environment, the algorithm itself, as part of the *Stable framework Baselines 3* [29], contains parameters that need to be adjusted. These parameters are called *hyperparameters*. Some of them are learning coefficient, entropy coefficient, discount factor, number of training steps and the size of the replay buffer.

The learning coefficient is one of the parameters that have the greatest impact on the training itself. During the experiments, it was noticed that even with a very small coefficient, the algorithm can learn to drive very well. Even when this value is fixed, the agent progresses quite solidly, and the progress is satisfactory. The downside is that training is very slow. If the learning factor increases, the agent in the early stages of training learns the basics faster, but later it starts to stagnate, even losing performance. If a function is passed as a parameter, which gradually reduces the learning coefficient, the algorithm can quickly learn how to move, and later fine-tune the driving technique without sudden losses in results. Of course, this function should also be carefully constructed, because it is unknown at which moments of training the algorithm will start to diverge in results, if the learning factor is too high, and in which moments it will learn too slowly with too little a factor.

The entropy coefficient is the second most important hyperparameter, because it determines the relationship between *exploration* and *exploitation*. In reinforcement learning, this is a very important aspect, because the agent learns by exploring the environment independently. This coefficient determines exactly how much the agent will focus on research and how much on applying what he has learned. A higher coefficient causes more exploration, while a lower one causes more exploitation. This coefficient is often not fixed, and changes during training. In this project, an automatic entropy coefficient was used, where the framework itself tries to optimize the coefficient during training. It is also important when this coefficient changes. In the initial stages of training, it is ideal for the coefficient to be higher, so that the agent gets more information about the environment. As training progresses, the coefficient should be reduced to allow the agent to focus on exploring finer details. Of course, the coefficient should be increased again at some point if it turns out that the agent was trained incorrectly.

Soft Actor-Critic itself contains a built-in discount factor [1]. This means that each subsequent award will be reduced a certain number of times compared to the previous ones. This is present due to environments that have a theoretically infinite number of steps per episode (*infinite horizon*). In this project, that coefficient can be applied when an episode represents a ride from start to finish, and even rides on circular tracks that would have an unlimited number of laps. As episodes with a fixed time duration will also be used, additional changes to this coefficient in this situation are not crucial, but remain one of the important elements to require attention in the future.

The total number of training steps defines the duration of the entire training. This number can be fixed, and a condition can be added to stop training if the agent achieves a certain result. As in supervised problems, in reinforcement learning there is a need to stop training early in cases where the agent experiences a sudden drop in performance. These falls can be caused by, for example, excessive learning and entropy coefficients. A good practice in those cases is to stop training, review the results, and continue training from the previous step with other hyperparameter settings. Therefore, the total number of steps may vary, but it is useful to have an insight into the duration of training in order to compare different algorithms. After the same number of steps, one algorithm can be more successful than the other and can be compared adequately.

6 RESULTS

The following table shows the best results of the trained agents compared to the average human driver. In order to make the comparison fair, the times of the average driver were observed on the first run, without additional practice and track memorization.

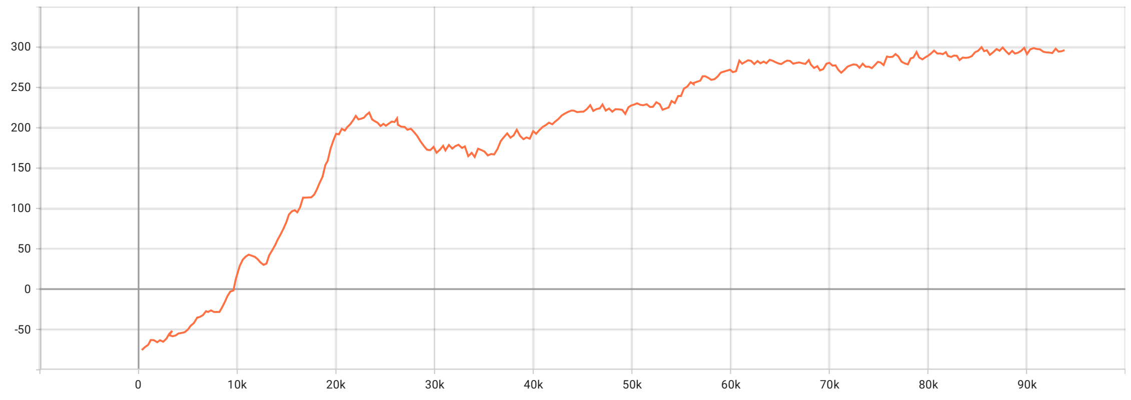
Table 1- Best times by track

Driver	Track 3	Track 4
Average Human Driver (First Attempt)	0:41:107	1:12:882
Convolutional approach	/	1:51:369
Classic supervised	2:01:275	/
Reinforcement with episode cutting	1:09:192	2:08:087
Reinforcement with base penalty	1:11:951	1:57:102

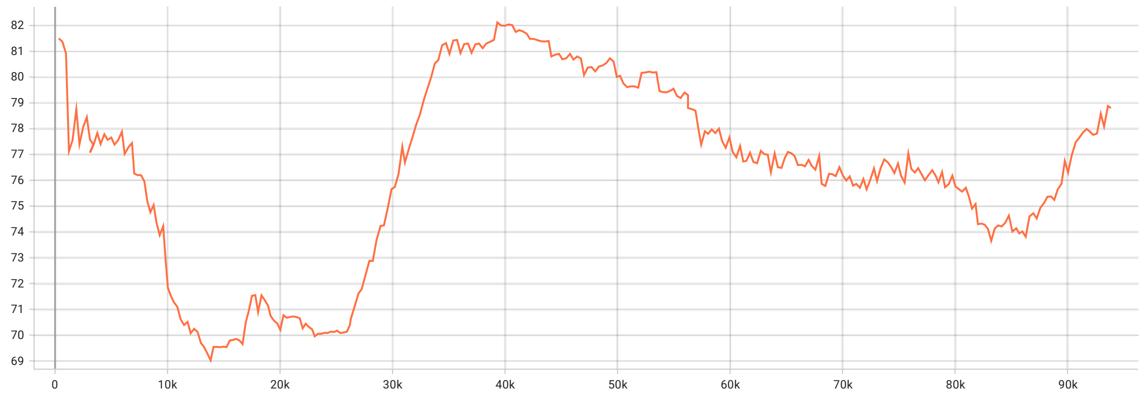
Track 4 is longer and more complex than track 3, so it is expected that the agents' results will be worse on it, compared to the average human driver. An interesting phenomenon is that none of the supervisory approaches was able to complete both tracks. Looking at the results achieved on track 3, the classic supervised approach completes the track with huge difficulty and high number of attempts and achieves very bad times (more than a minute behind the human driver which is more than double). In contrast, the convolutional approach manages to beat both reinforced agents on path 4, which is more demanding, with a significantly lower relative delay compared to the human result. The explanation lies in the better conception of the input data in the convolutional approach, which takes into account more details than the classical approach. Also, the supervised agents are trained on data where actions often include full throttle, which leads to faster driving. Combined with a track 4 configuration that gives room for collision recovery in places where a convolutional agent would often make a mistake, this agent is capable of defeating reinforcement agents by nearly six seconds. However, when looking at driving style, supervisory approaches are no better than reinforced ones. Additionally, supervised agents will never be able to outperform the training set they were trained on, so the advantage of reinforced agents is obvious. However, with supervised learning it is possible to train agents very quickly to a certain level, which could be further trained by reinforcement. This hybrid approach theoretically represents a very good type of training that combines the advantages of both techniques, and its application could bring much better results.

Reinforcement agents demonstrated very careful driving styles, compared to the aggressive steering of the supervisory agents. This leads to fewer collisions, but also a lower average speed. This is another reason for the poorer result on track 4 compared to the convolutional agent. However, it can be seen that the agents behave in accordance with the reward functions, ie. they try to enter curves better and to reduce the number of sudden changes in direction of movement. Still, in order to exploit the full potential of reinforcement learning, it is necessary to spend much more time on the track and collect a larger amount of data. Also, with better tuning of parameters and hyperparameters, much better results can be achieved.

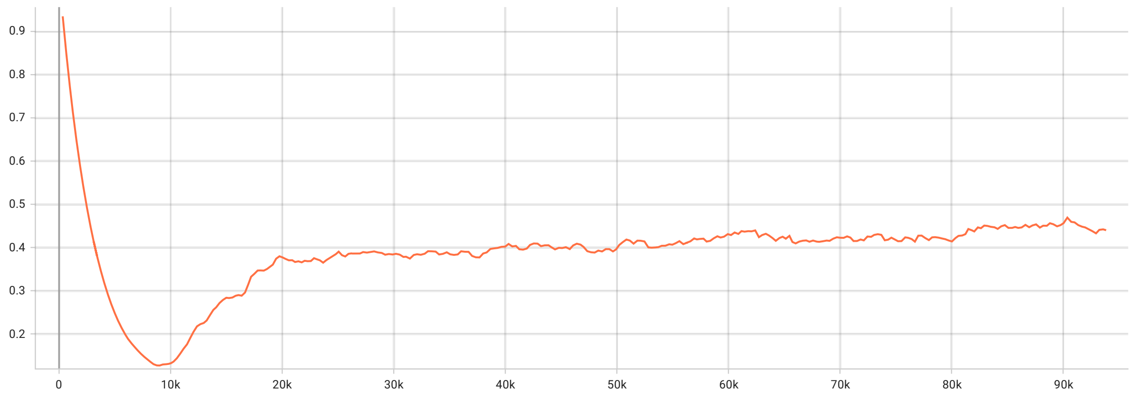
During the training of the agent that uses the method of cutting episodes, parameter changes were made in order to improve performance. Not all parameters are optimal during the entire training period, which further complicates training and recording results. The following images show the progress of this agent in time steps.



Graph 1- Total reward at the end of the episode during training



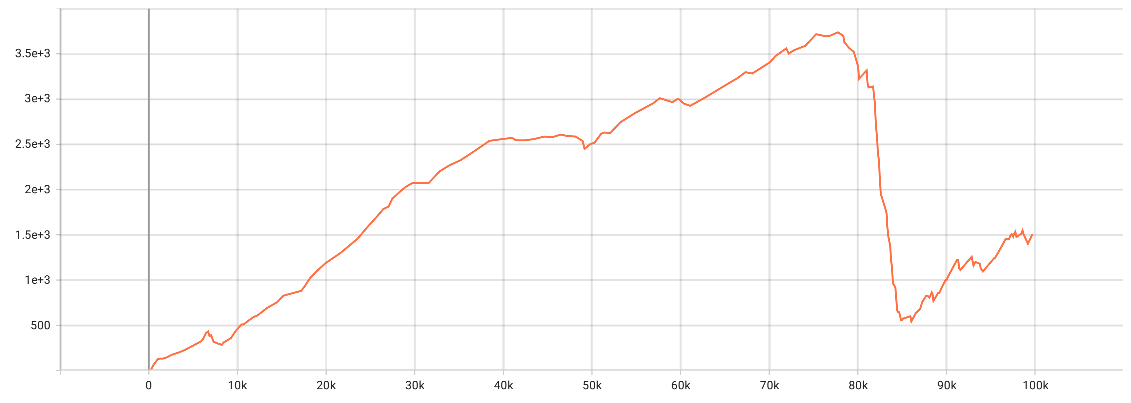
Graph 2- Episode duration during training



Graph 3- Entropy coefficient

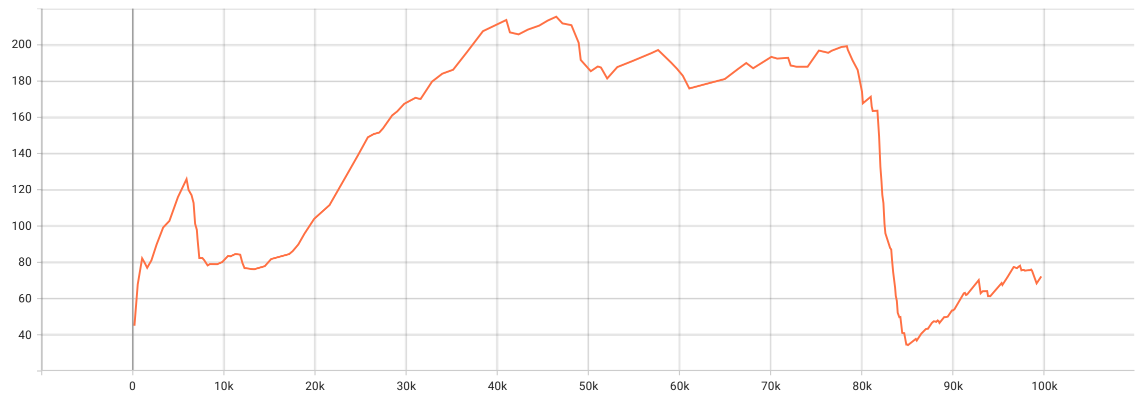
The duration of an episode for this agent is not relevant to determining the success of the agent because all episodes last the same time. This graph, however, can be used to monitor the delay of the actions described in the previous chapters, and for that reason it is important to show these results as well. The graphic showing the reward clearly shows that the agent in the early stages of training is not familiar with the basics of driving and often comes into contact with walls, receiving a negative reward. After ten thousand timesteps, an increasing trend towards higher rewards is observed. By twenty thousand steps, the agent is already capable of driving fast enough and achieving good results. With further training, the agent improves his driving style and continuously receives higher rewards. It follows from this that further training can be expected to improve. Progress is very slow, which is directly related to the small and constant learning coefficient of 0.0003. This is the default value for this parameter, which has been shown to give stable training and clear improvements.

For the agent using the base penalty, without cutting episodes, a linear learning coefficient starting at 0.003 and ending at 0.0003 was used. From the graph of the total reward during training (*Graph 4*), a faster progress can be observed compared to the agent with a fixed learning coefficient.



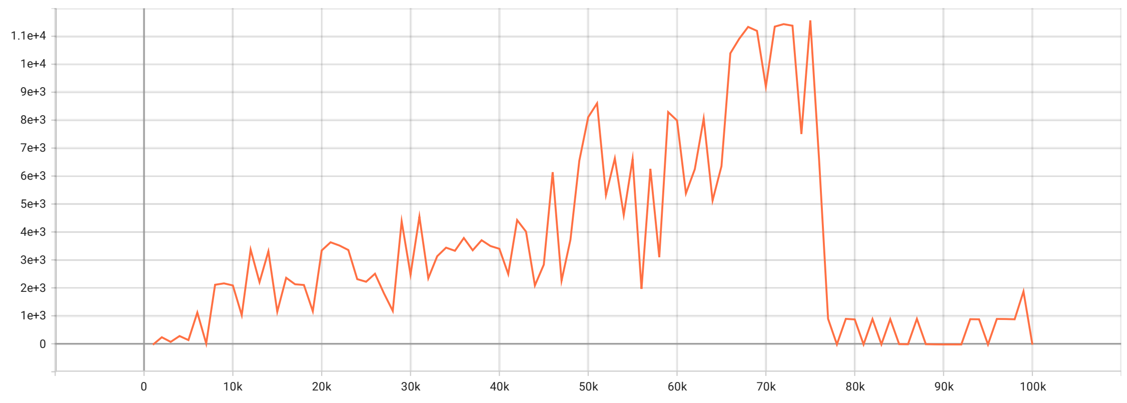
Graph 4- Total reward at the end of the episode during training

For this agent, it makes sense to follow the episode duration graph in parallel with the reward, because with a higher reward, the algorithm should tend towards shorter episodes. Just before fifty thousand timesteps, the agent continues to improve the reward, but reduces the duration of the episode (*Graph 5*).

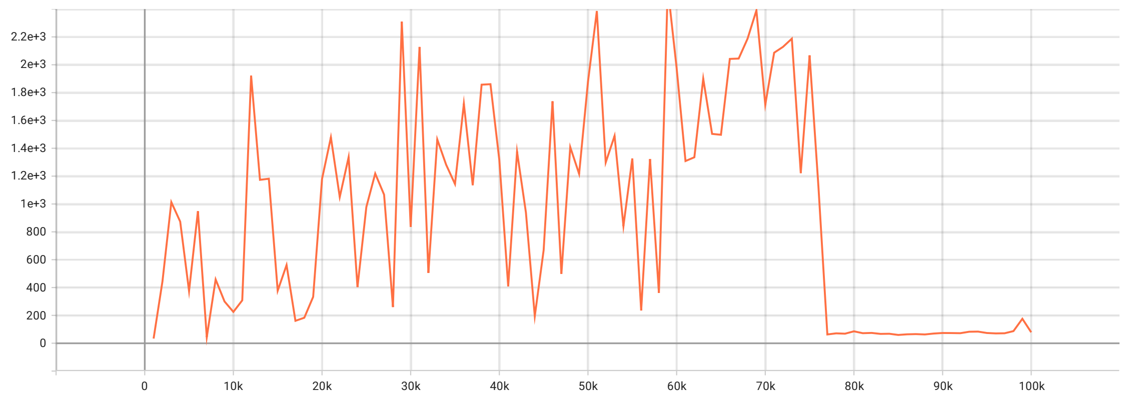


Graph 5- Episode duration during training

Although all agents drive with deterministic evaluation actions, noise and delays in obtaining observations and sending actions lead to large differences in individual drives. In order to show the average success of the agent as well as possible, it is necessary to run a large number of evaluation runs, which leads to significantly slower training. The evaluation graphs (*Graph 6*and *Graph 7*) are shown here for easier comparison of the results that the agent achieves during training (with a certain coefficient of exploration) and during pure driving (without exploration). The following graphs show a huge drop in performance after seventy seven thousand steps.

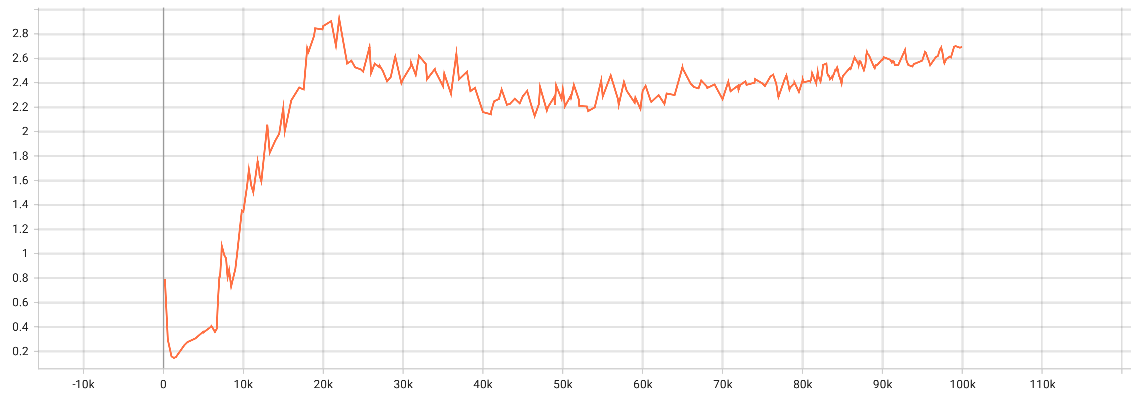


Graph 6- Average reward during evaluation



Graph 7- Average duration of an episode during the evaluation

There are several possible reasons for the sudden deterioration of the results, and one of them is an excessive entropy coefficient. After a certain time, it is necessary to reduce this coefficient so that the algorithm has less deviation from the learned path and thus improves the driving style more finely. In this case (*Graph 8*), the coefficient remained approximately the same during most of the training, which led to overexploration.



Graph 8- Entropy coefficient

As the best agent a checkpoint saved after seventy-five thousand steps was used. According to the results, it can be concluded that this agent is slightly better than the previous one, because it is more than ten seconds faster on track 4, while on track 3 it lags behind by a little less than three seconds. Considering that track 3 is shorter, the results favor the agent with the base penalty.

7 CONCLUSIONS

Based on these results, it can be concluded that relatively good self-management systems can be made very quickly with machine learning. With longer training, more training data, better tuning of parameters and hyperparameters and combining different approaches, there is a huge room for improvement. This miniature project proves that the Soft Actor-Critic algorithm is extremely flexible and applicable to different configurations of the training environment [1]. However, it would be a good practice to apply, after adequately constructing the environment, other algorithms in order to compare the results, both in terms of results, as well as in terms of learning duration and driving styles.

As the final goal of autonomous racing is the application of algorithms on real cars, the next step for the development of this project would be to modify the system to work with miniature, and later on commercial vehicles of the right size. Additionally, during the training period of the supervisory networks, a miniature system was built using a microcomputer with a camera mounted on the front bumper of a remote-controlled car.

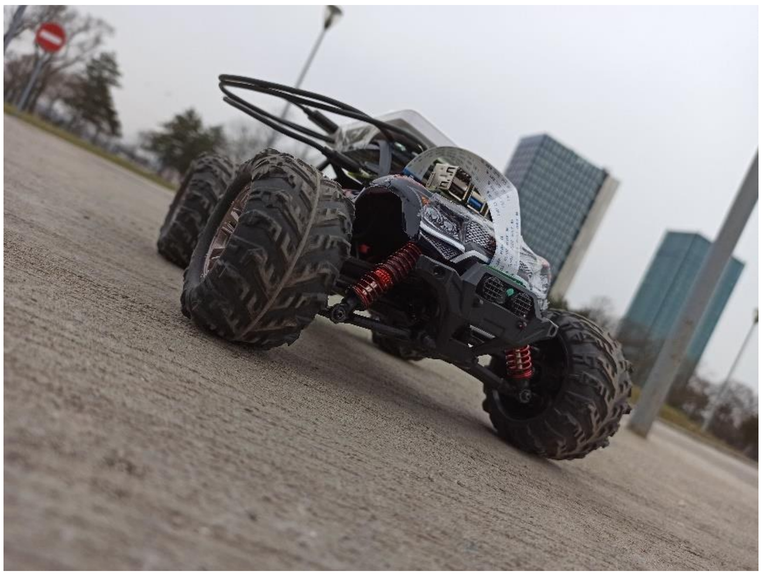


Figure 10- Modified car with microcomputer and camera

The car has been modified to receive signals from a microcomputer rather than a remote control. Since supervised networks are simpler to implement, and the microcomputer did not have enough sensors for better perception at the time of the test, the choice of convolutional and classical agents was justified. Although they had no prior insight into the camera angles of the miniature car, nor the configuration of the track it was driven on, the networks trained in the simulator were able to drive around the simplest parts of the track.

Picture 11- Driving a car on simple tracks

This observation proves the high portability of self-driving algorithms, and with additional training on the car itself, it would lead to even better results. Therefore, one of the next stages in the development of artificial intelligence for autonomous racing is definitely a transfer to physical cars.

This project, as said in the introduction, is *a proof of concept*, which means there is a lot of room for improvement. Given that all algorithms were implemented on home computers that do not have high computing power, these results are quite acceptable. However, with the use of much more powerful devices and the eventual distribution of training environments, agents will definitely have much better results. For all types of machine learning, and especially for reinforcement learning, a very large training set is necessary that contains many different situations. The application of distributed environments would greatly facilitate the entire process of data collection, and thus improve the quality of the agent that would learn from several different situations at once. Also, using a better restart mechanism would solve the retraining problem that is still very noticeable on the displayed agents. Although the reward feature proved to be great in other games [2], it doesn't necessarily mean that it will perform the same in other simulators, especially due to the different ways in which simulators handle car movement and contact with walls. However, for the purposes of this project, this reward function contributed to relatively quick training and relevant results.

8 REFERENCES

[1] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”

[2] F. _ Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P . D ü rr , “ Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning ”

[3] G. Williams, P. Drews, B. Goldfain, JM Rehg, and EA Theodorou, "Aggressive driving with model predictive path integral control"

[4] A. Liniger, A. Domahidi, and M. Morari, "Optimization-based autonomous racing of 1: 43 scale rc cars"

[5] E. Frazzoli, MA Dahleh, and E. Feron, "Real-time motion planning for agile autonomous vehicles"

[6] T. Novi, A. Liniger, R. Capitani, and C. Annicchiarico, "Real-time control for at-limit handling driving on a predefined path"

[7] U. Rosolia and F. Borrelli, "Learning how to autonomously race a car: a predictive control approach"

[8] J. Kabzan, M. dl I. Valls, V. Reijgwart, HFC Hendrikx, C. Ehmke, M. Prajapat, A. B¨uhler, N. Gosala, M. Gupta, R. Sivanesan et al., “Amz driverless: The full autonomous racing system"

[9] CJ Ostafew, AP Schoellig, and TD Barfoot, "Robust constrained learning-based nmpc enabling reliable mobile robot path tracking"

[10] R. Verschueren, S. De Bruyne, M. Zanon, JV Frasch, and M. Diehl, "Towards time-optimal race car driving using nonlinear mpc in real-time"

[11] G. Williams, N. Wagener, B. Goldfain, P. Drews, JM Rehg, B. Boots, and EA Theodorou, “Information theoretic mpc for model-based reinforcement learning”

[12] G. Williams, P. Drews, B. Goldfain, JM Rehg, and EA Theodorou, "Information-theoretic model predictive control: Theory and applications to autonomous driving"

[13] DA Pomerleau, "Alvinn: An autonomous land vehicle in a neural network"

[14] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, LD Jackel, M. Monfort, U. Muller, J. Zhang et al., "End to end learning for self - driving cars"

[15] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, “End-to-end race driving with deep reinforcement learning”

[16] M. Riedmiller, M. Montemerlo, and H. Dahlkamp, “Learning to drive a real car in 20 minutes”

[17] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J. Allen, V. Lam, A. Bewley, and A. Shah, “Learning to drive in a day”

[18] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving"

[19] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu, “High-speed autonomous drifting with deep reinforcement learning”

[20] Trackmania – The ultimate track racing game, link: https://www.trackmania.com/

[21] Python Software Foundation , Python Language Reference, version 3.9, link : http://www.python.org

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, GS Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp , G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke , Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems”, link: https://www.tensorflow.org/

[23] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Kopf, Andreas and Yang, Edward and DeVito, Zachary and Raison, Martin and Tejani, Alykhan and Chilamkurthy, Sasank and Steiner, Benoit and Fang, Lu and Bai, Junjie and Chintala, Soumith, “PyTorch : An Imperative Style, High-Performance Deep Learning Library”, link : https://pytorch.org/

[24] Openplanet, The advanced extension platform for Trackmania and Maniaplanet, link: https://openplanet.dev/

[25] GBX.NET, a C#/.NET parser for Gbx files from Nadeo games, link: https://github.com/BigBang1112/gbx-net

[26] Google Colab, link: https://colab.research.google.com/

[27] Bradski, G., The OpenCV Library, link: https://opencv.org/

[28] Alex Clark and Contributors, forked from Fredrik Lundh and Contributors, link: https://github.com/python-pillow/Pillow

[29] Antonin Raffin and Ashley Hill and Adam Gleave and Anssi Kanervisto and Maximilian Ernestus and Noah Dormann, “Stable-Baselines3: Reliable Reinforcement Learning Implementations”, link: https://stable-baselines3.readthedocs.io

[30] Gym, link: https://www.gymlibrary.ml/