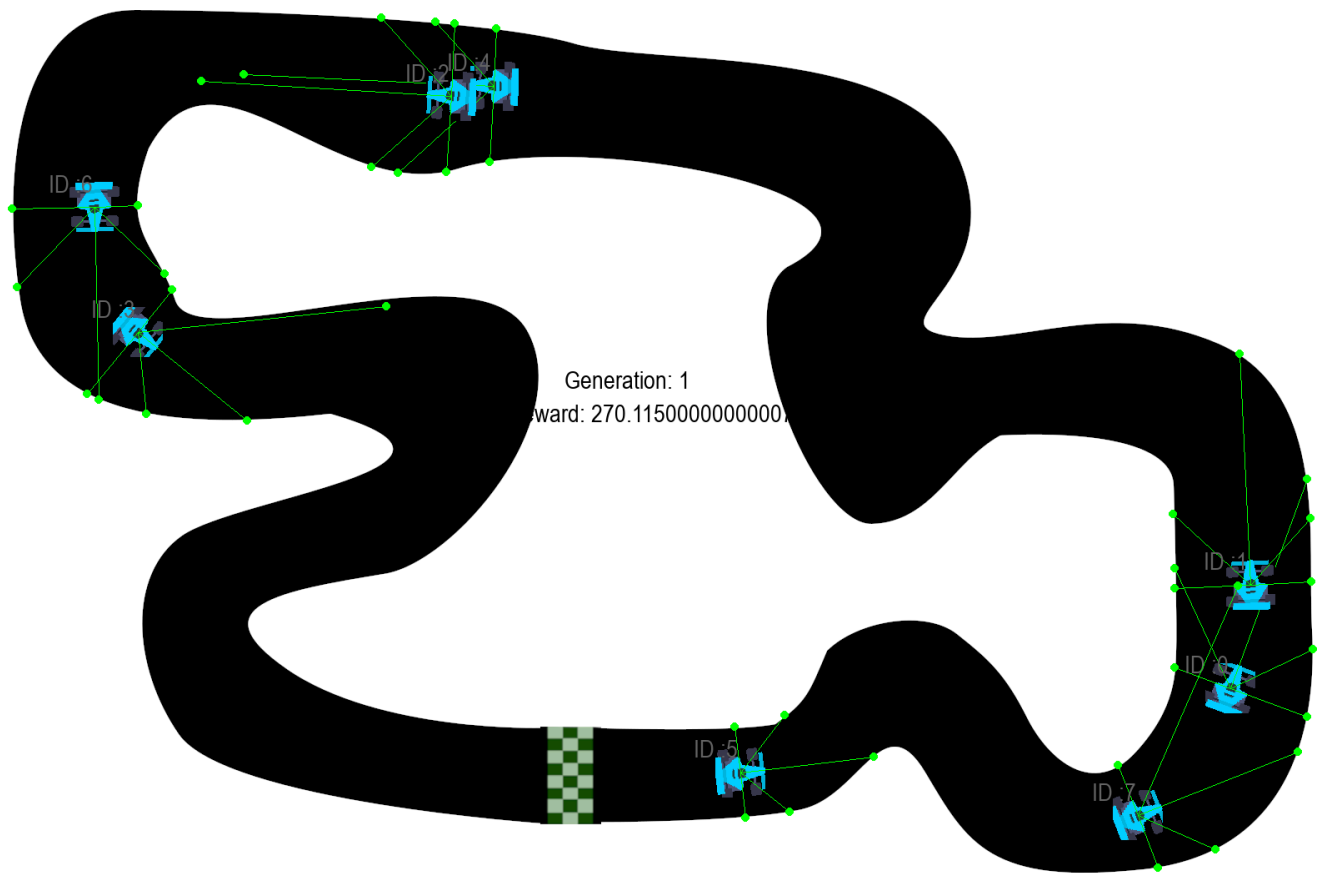


AI Project Report

2D Autonomous Driving



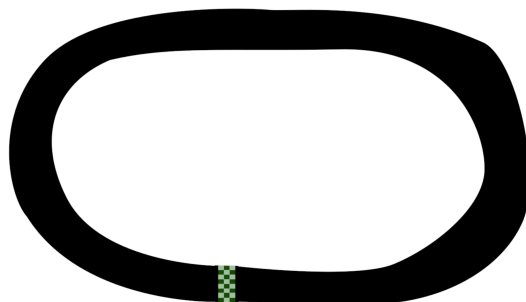
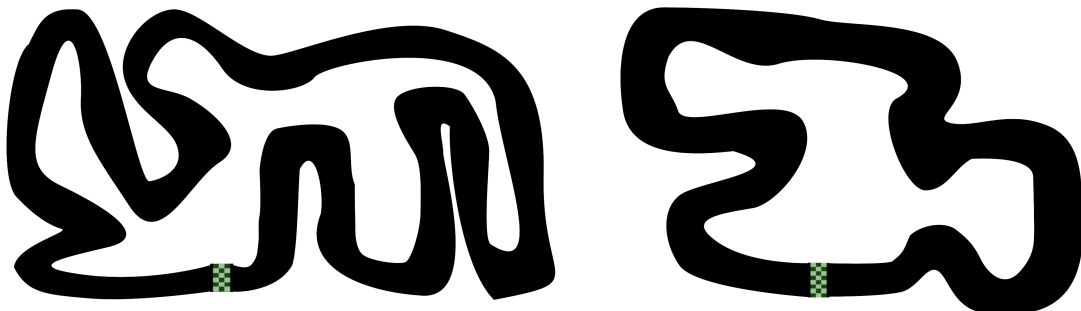
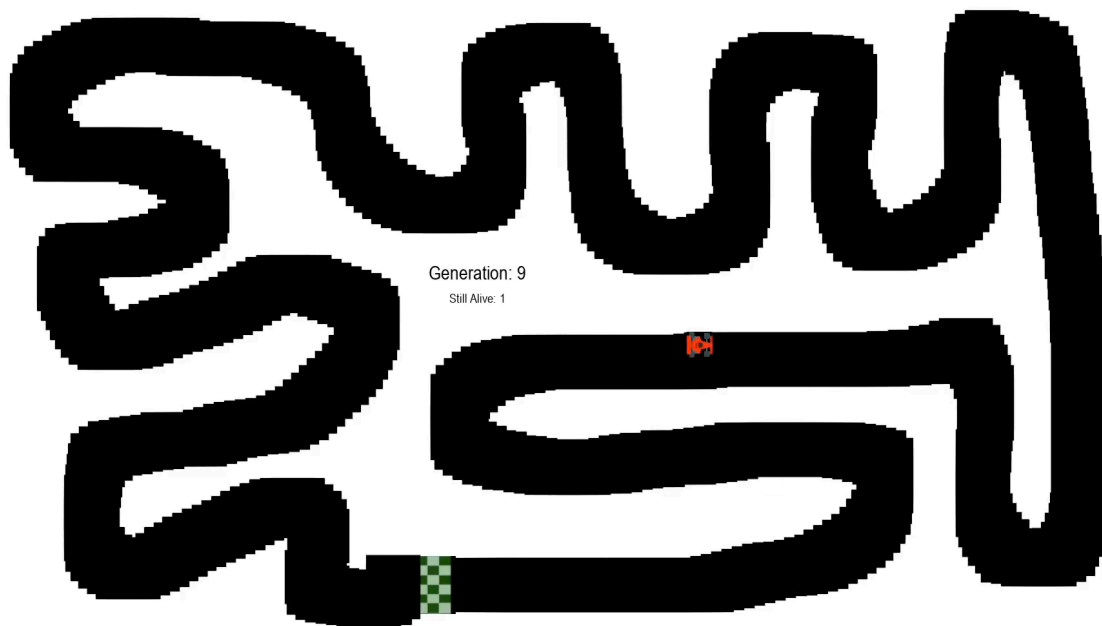
CARDOT Clement
GUILBAUD Maxime
GUAIS Clement

Table of Contents

| | |
|--|-----------|
| Table of Contents | 1 |
| 1. Explanation and formalization of the problem | 2 |
| 2. Description of data | 3 |
| 3. Work Done | 4 |
| 3.1. Analysis of the existing | 4 |
| 3.2. NEAT algorithm | 4 |
| 3.3. Migration vers Stable Baselines | 5 |
| 3.3.1. Algorithms | 6 |
| 3.3.1.1 A2C | 6 |
| 3.3.1.1 PPO | 7 |
| 3.4. Awards | 8 |
| 3.4.1 Sectors | 8 |
| 3.4.2 Optimization of Rewards | 8 |
| 3.5. Le multiprocessing | 9 |
| 4. Results Obtained | 11 |
| 4.1. NEAT algorithm | 11 |
| 4.2. A2C algorithm | 11 |
| 5. Discussion and opening | 14 |
| 5.1. Multi-circuit training | 14 |
| 5.2. Radars | 14 |
| 5.3. 3D environment | 14 |
| 5.3.1 CARLA Simulator | 14 |
| 5.3.2 Donkey Car | 15 |
| 6. References: | 16 |

1. Explanation and formalization of the problem

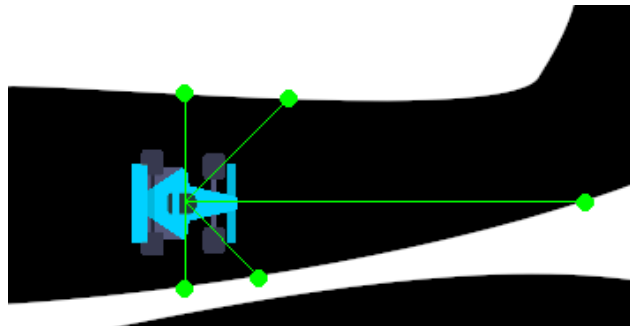
The objective of the project is to teach a car a circuit so that it can do laps of a circuit independently. Subsequently, we can test the model on other more or less similar circuits in order to test the robustness of the learning.



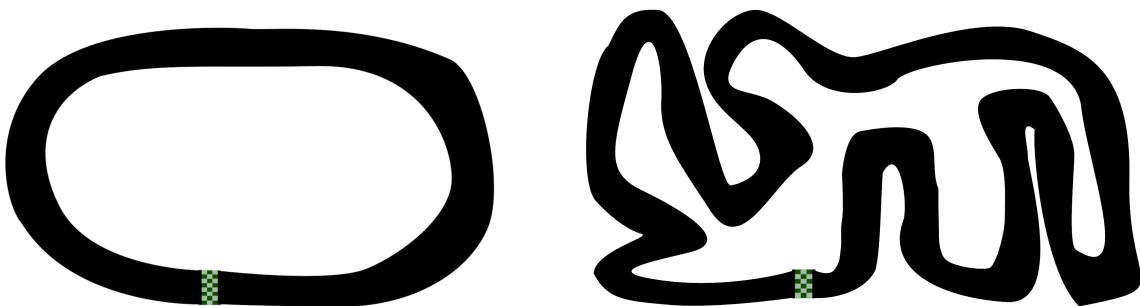
2. Data description

In our case, the input data are those provided by the sensors present on the car. There is in all **5 sensors returning information about the distance to the border**. At each refresh, the sensors calculate the distance between the car and the nearest wall (collision with a white pixel), if the distance is greater than 300 pixels, the distance retained is 300 pixels.

Once this distance has been calculated, in order to **normalize data between 0 and 1**¹ the sensor will therefore divide the measured value by the maximum value: 300 pixels. Thus each sensor generates a variable ranging from 0 to 1 corresponding to the distance from the nearest wall.



We immediately notice that the quality of the data transmitted by the sensors is largely dependent on the card chosen for learning. The choice of map is therefore important for good learning, as it can be more or less complex.



normalize data between 0 and 1¹: This is a good practice when doing reinforcement learning. Indeed, it has been shown that AI performs better when the data is normalized between 0 and 1.

Subsequently we use reinforcement algorithms and from the input data the **output data** will be here **acceleration** (or braking of the car) as well as its **direction**.

3. Made work

3.1. Analysis of the existing

In order to embark on this project, we first needed to **set a training environment**: a car, a circuit, sensors... This is work that can quickly take a lot of time. Since our time during this project is limited and it would be better for us to focus on AI, we decided to **take over an already existing environment**.

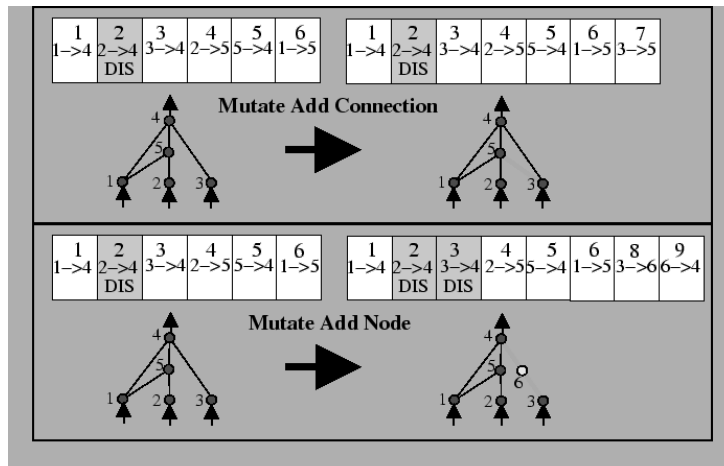
There are many projects on Youtube and GitHub that implement reinforcement learning on cars with a 2D circuit. After going through a number of them, we selected the one from the Youtube and GitHub account "NeuralNine". His project has many advantages: **multiple circuits available, simple sensor system, NEAT algorithm...**

In the remainder of this report, you will see that the initial project has been greatly modified, but the choice of starting with this basis will have allowed us to push our initial objective much further.

3.2. NEAT algorithm

Firstly, we therefore used the algorithm provided in order to be able to train it and test it on the different circuits. A model was created for each circuit and they were tested on all circuits.

The algorithm **NEAT** (NeuroEvolution of Augmenting Topologies) is an evolutionary algorithm used to evolve artificial neural networks. Unlike traditional optimization approaches that simply adjust the weights of connections in a fixed neural network, NEAT is able to **evolve the very structure of the network, including the number and configuration of neurons as well as the connections between them**. NEAT uses the steps of a genetic algorithm, that is to say it will use the steps of **selection, reproduction and mutation** in order to have genetic diversity. In addition, NEAT uses a mechanism of **gradual complexity of neuronal structures** over generations. Innovations that improve performance are more likely to survive and spread through the population.



Performance is evaluated by a reward function. This is determined by the distance traveled by the car but this poses problems discussed more in the section on rewards.

For the training phase:

- The configuration was done by iteration. **Population** should not be too small so as not to interfere with learning. It was set at 100 in order to allow suitable learning. **The number of generations** as for it was adjusted so that the circuit was finished by at least 1 car.
- The number of generations depends mainly on the complexity of the card. Below we find the minimum number of generations and the learning time in order to complete a lap with at least 1 car with a population of 100 cars
- At the end of training, the best genome (the one with the greatest reward) was saved as a model map.

For the testing phase:

- Using a **parameterized program** with the argument of the card played and the model to use. In this way we can test each model on all the cards.

3.3. Migration vers Stable Baselines

In order to challenge this model we decided to migrate to stable-baselines. Stable-Baselines is an open-source library in Python designed for training, evaluating, and deploying reinforcement learning algorithms.

We chose this library because it has very complete documentation and it is one of the easiest machine learning libraries to use. In addition, it has different algorithms which can be interchanged very easily in order to compare their performances.

3.3.1. Algorithms

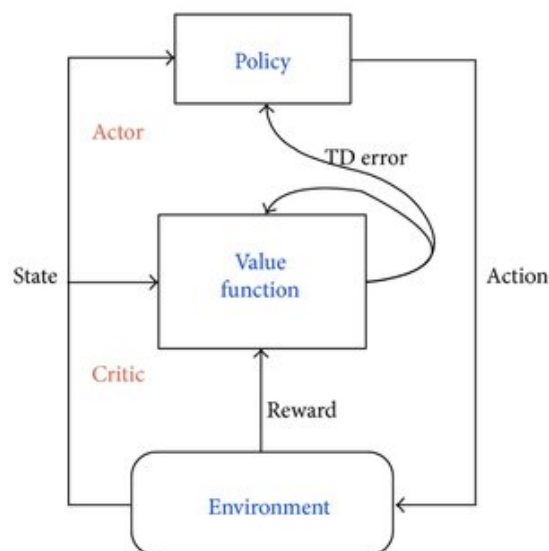
The stable baselines library allows the use of different algorithms allowing us to be able to compare them to use the most efficient one.

3.3.1.1 A2C

The algorithm **A2C** (Advantage Actor-Critic) is a reinforcement algorithm that uses **actors**. The actor is responsible for taking actions in the environment. It takes as input the current state of the environment and generates a probability distribution over possible actions. This results in the speed and rotation actions on the car.

Then there is **criticism**, a phase of evaluating the quality of the actions taken by the actor. This evaluation makes it possible to calculate the value of a state, which represents the expected return from this state. The further the car travels, the higher this state will be.

The advantage will subsequently modify the politics of actors and criticism in order to maximize the value of the state.



For the training phase:

- We use **8 environments in parallel** to maximize CPU usage.
-

- we leave the minimum training time so that at least one car finishes a lap.
- The model at the end is saved in order to be tested on all cards afterwards.

For the testing phase:

- We test the model on the cards and note whether the car can do a trick.

3.3.1.1 PPO

The algorithm **PPO** (Proximal Policy Optimization) is very similar to A2C in how it works. It introduces a measure of proximity between new and old policies to avoid too drastic updates, which contributes to the stability of learning.

Firstly, the agent interacts with the environment which influences the calculated reward.

Then the algorithm performs updates on the agents' policy in order to maximize a clipped objective function. This function uses the probability ratio between old and new policies to limit change from the previous policy.

The clipped objective function:

$$L(\theta) = E[\min(r_t(\theta) \cdot A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \cdot A_t)]$$

Or :

- $r_t(\theta)$ is the probability ratio between new and old policies.
- A_t is the advantage of the action at time t .
- ϵ is a clip hyperparameter, usually a small number.

Its main difference with the A2C algorithm is the update of its policy. A2C uses the standard ascending gradient method, while PPO introduces a proximity form with **batch update to stabilize learning**. This makes the PPO algorithm more stable but slower to learn.

Then, whether for the learning or testing phase, it remains similar to the conditions used for the A2C algorithm.

3.4. Awards

3.4.1 Sectors

To ensure consistency of navigation, we used the **sensors** already existing in order to detect different colors on the circuit and thus set up a system of sectors. The latter allows the vehicle to memorize the sector in which it is located. In the event of reaching an unanticipated sector, protection mechanisms **rewards** or **penalties** are activated to steer the vehicle in the right direction.

Application examples:

- If the car is traveling in the opposite direction, the sector system triggers **penalties**. Consequently,
- If the car appears to be going in circles, it will receive **fewer rewards** than its congeners.

3.4.2 Optimization of Rewards

Initially, the **distance** traveled constituted a relevant metric for evaluating vehicle performance. However, we set out to improve the process. With this in mind, we explored the idea of basing the **reward** on the **time**, thus emphasizing the survivability of the vehicle. However, during the first tests, this choice led to undesirable behavior where the vehicle tended to rotate on itself, thus minimizing the risks.

With the aim of achieving the opposite effect, we sought to balance the rewards and introduce **temporal penalties**, with the intention of promoting the speed of the vehicle. However, this approach proved counterproductive, as the vehicle tended to move as quickly as possible into a wall, highlighting the need for more precise optimization.

For a more detailed evaluation of performance, we finally integrated the **average speed** in the reward process. This modification allowed for a more nuanced assessment, promoting fast and efficient driving while avoiding undesirable behaviors such as excessive rotations or collisions against walls.

These adjustments were essential to increasing the robustness of the AI in the face of varied scenarios, highlighting the crucial importance of precise calibration of rewards to achieve optimal driving behaviors.

Improvement in performance observed to complete 1 round (A2C):

- **Map 1:** From 1 minute to 26 seconds (division by ~2)
- **Map 2:** From 2 minutes 38 seconds to 1 minute 3 seconds (division by ~2.5)
- **Map 3:** From 7 minutes 12 seconds to 2 minutes 35 seconds (division by ~2.8)
- **Map 4:** From 3 minutes 6 seconds to 1 minute 31 seconds (division by ~2)

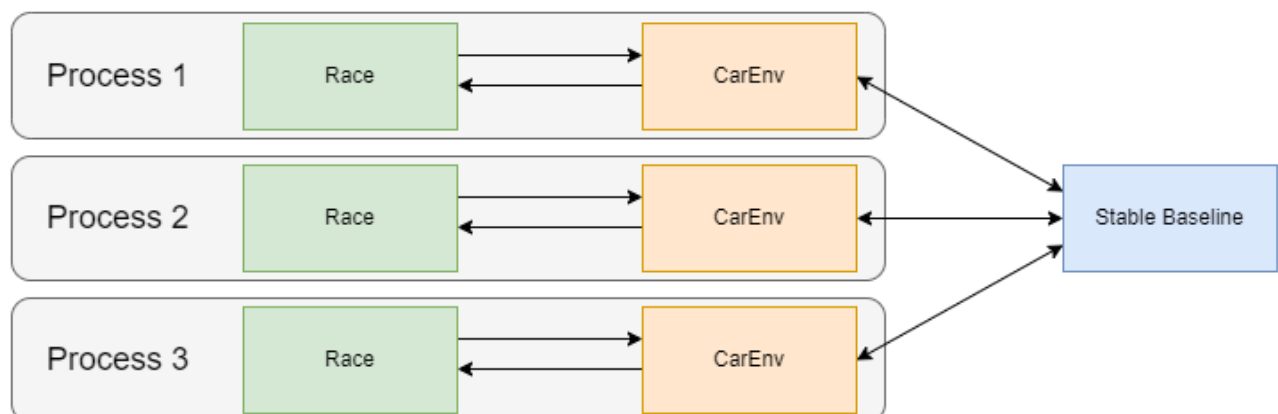
3.5. Le multiprocessing

The basic algorithm with NEAT allowed the simulation of several cars simultaneously in order to create waves comparable to model generations. But all this in **mono-processing** !

With the aim of improving performance during learning, we wanted to set up learning **multi-process** in order to use the full power of the CPU.

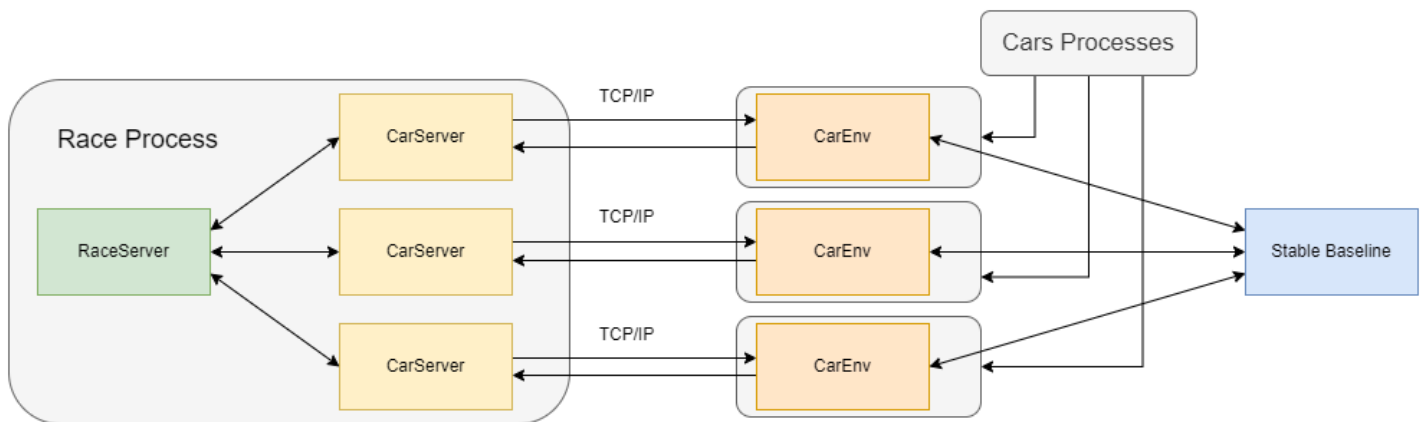
With stable-baselines, we had to change a lot of things about the architecture of our system. Indeed, most machine learning frameworks define an environment on which **An instance** of the model is training. If we want to accelerate learning by using several CPU cores, the framework will generate as many instances of the environment as of the model. In our case, this is not desirable because the environment is relatively heavy and we would prefer to visualize all the cars on the same circuit.

It was therefore necessary to find an alternative. One solution that immediately came to mind was to transform the current single-process system into a multi-process system. But to share the same instance of the circuit it becomes complicated. Since the cars are each on an independent process, the call to the Singleton of the circuit class creates a new instance of the circuit each time. We are therefore left with **as many circuits as cars...** For several processes to share the same circuit, another solution had to be found.



We therefore decided to transform our circuit into a **TCP/IP server** and instances of our models in Client. Therefore, when launching our script, we define a

server with the expected number of clients, then we start training the model with the same number of clients.



The implementation of multiprocessing allows a significant time saving during learning, in fact it allows the AI to train with many more iterations per second. The performance gain can be estimated as a multiplication by the number of CPU cores used.

Example with the duration of the apprenticeship:

1 Core = 5 mins

2 Core = 2 mins 30 secs

4 Core = 1 mins 15 secs

8 Core = 37.5 secs

PS: This is a theoretical estimate because in reality learning still requires a minimum of time in order to correctly complete the entire circuit.

4. Results Obtained

4.1. NEAT algorithm

With the NEAT algorithm the learning time corresponds to the time taken for at least 1 car to finish a lap.

| | Card 1 | Card 2 | Card 3 | Card 4 |
|----------------------|--------|--------|------------|--------|
| Number of Generation | 2 | 5 | 40 | 9 |
| Learning time | ~5sec | ~10sec | 4min 50sec | ~44sec |

The conclusions of the tests are: **the more complex the map with diverse turns** for the learning phase **the more the model is adaptable to other cards**. An example of this is the model driven by card 1. This model trained on a circuit where you only turn left does not know how to turn right when necessary. A second example is the model driven by circuit 4. This model is only trained for one type of turn (90 degree turn) which makes it not very robust when the turns are wider or shorter. Below is a table showing for each model whether the car managed to complete a lap on all maps:

| | Card 1 | Card 2 | Card 3 | Card 4 |
|-----------------|--------|--------|--------|--------|
| Card template 1 | | | | |
| Card template 2 | | | | |
| Card template 3 | | | | |
| Card template 4 | | | | |

4.2. A2C algorithm

With the A2C algorithm the learning time corresponds to the time taken for at least 1 car to finish a lap.

| | Card 1 | Card 2 | Card 3 | Carte4 |
|---------------|--------|--------|--------|--------|
| Learning time | ~25sec | 1min25 | 5min25 | 2min04 |

Then when we test the models trained on the maps for 30 minutes, the car cannot complete a lap on the most complex maps despite the fact that it could do several laps during training. This can be explained by insufficient exploration making the model not adapted to the change in environment. Below is the result of the models trained on each map for approximately 30 minutes:

| | Card 1 | Card 2 | Card 3 | Card 4 |
|-----------------|--------|--------|--------|--------|
| Card template 1 | | | | |
| Card template 2 | | | | |
| Card template 3 | | | | |
| Card template 4 | | | | |

4.3. PPO algorithm

With the PPO algorithm the learning time corresponds to the time taken for at least 1 car to finish a lap. We notice that the minimum time for a car to complete a lap increases quickly depending on the complexity of the map.

| | Card 1 | Card 2 | Card 3 | Carte4 |
|---------------|--------------|---------------|-------------|--------|
| Learning time | 3 min 34 sec | 25 min 10 sec | more than 1 | / |

| | | | | |
|--|--|--|------|--|
| | | | hour | |
|--|--|--|------|--|

4.4. Comparison

The most efficient algorithm in terms of learning speed and robustness among the three used is therefore the NEAT algorithm. Even if during learning the A2C algorithm can finish a lap faster than the NEAT algorithm, the model must have a training time significantly higher for it to be used on the different cards. Finally it takes a training time of several hours minimum so that the PPO algorithm can provide a result but it may be more robust than the other two methods given the stability of the learning.

5. Discussion and opening

5.1. Multi-circuit training

We have seen that the circuit on which the model trains strongly impacts the performance of the model on other circuits. Indeed, the generated model is quickly found in overfitting in order to best stick to the circuit on which he trains. To avoid this bias, we could train the model on several circuits simultaneously. Each car would be connected to a circuit to feed the algorithm with more varied data.

Training of this type should in theory allow the model to be very versatile afterwards.

5.2. Radars

In our simulation, each car has five radars. These radars are comparable to those found on the reversing radars of current cars. For autonomous driving, however, today's vehicles are more often equipped with LIDAR sensors and/or cameras. This allows much better anticipation of the trajectory and detection of obstacles. An area of improvement could therefore be to improve our radar system in the simulation. Note that this type of sensor would lead to a drastic increase in the data to be processed by the AI. It would therefore undoubtedly be necessary to process radar data upstream in order to reduce the load on the AI.

5.3. 3D environment

Our simulator allowed us to generate a model capable of moving in a 2-dimensional environment. In order to get closer to a realistic environment, the next step would therefore be to move to 3 dimensions. Although cars do not fly and therefore technically move in 2 dimensions, the collection of information from sensors must be done in 3 dimensions. Today there are simulators that make it easy to obtain this kind of environment.



5.3.1 CARLA Simulator

CARLA is an Open Source project sponsored in particular by Intel and Toyota. This tool allows you to obtain a 3D environment in order to train AI models to drive vehicles. The environment is generated using the Unreal Engine, which provides extremely realistic images and physics.

5.3.2 Donkey Car

Donkey Car is another Open Source project which is more like a framework for training models capable of driving miniature cars. These cars are generally composed of a Raspberry or an NVidia Jetson nano which allows the model to be loaded directly into the car. However, in order to make the project accessible to as many people as possible, the community created a simulator called DonkeySim. This one is comparable to CARLA, however it runs on the Unity engine and is much less realistic in terms of physics and graphics.

6. References:

- Youtube video from the “NeuralNine” channel
 Self-Driving AI Car Simulation in Python
- Code source de NeuralNine
<https://github.com/NeuralNine/ai-car-simulation>
- Youtube video from the channel “Dave’s Space” clearly explaining the difference between Thread and MultiProcessing
 threading vs multiprocessing in python
- Stable-Baselines ([documentation](#))
- DonkeyCar
<https://www.donkeycar.com>
- CARLA Yes
<http://carla.org/>