Performance Evaluation project:
Optimizing cars' trajectory with AI

Ottavy Macéo, Longatte Mathieu, Mocq Louison

Abstract

The project is divided into five parts:

- Creating a racing car environment to simulate a simple 2D racing car model.
- Implementing Deep Q-Learning and Genetic Algorithms to optimize the behavior of a car on tracks, enabling it to follow the best possible trajectories.
- Evaluating the performance of Deep Q-Learning and Genetic Algorithms and comparing their results.
- Assessing the performance of Deep Q-Learning with respect to different hyperparameters.
- Evaluating the performance of the best car behavior achieved by both algorithms.

You can see the complete project on our public Github page.

Introduction

We focus on solving the problem of optimizing a car's trajectory using a Deep Q-Learning model. The goal is to assess the ability of this model to generalize its experience from a limited number of circuits to new ones. To achieve this, we consider the car's trajectory in a plane under a simplified physics model. The model's performance will be compared to that of a genetic algorithm. Then, we will examine the impact of the chosen hyperparameters on the model's training performance. Finally, we will explore the model's limitations when trained on a large amount of data.

Modeling

Racing environment

Tracks

A track is originally a .png file wich look like the left image of figure 1. Then, the image is converted to a matrix T such that $T[0][0]$ is the bottom left corner. After that, we crop the image, compute the starting point and the lines of track (that will be explained in the reward part) to have a final result which look the right image of figure 1. The white case represent the road, the green point represent the starting point.

Cars' physics

The Car physic is really simple. It is a 2D cartoon-like physics that act as follow:
The car has two main informations: its speed $\in$ [0, MaxSpeed] and its rotation $\in [0, 360]$. The physics acts as follow: at each times step the car move to next coordinates on the direction of the car's rotation and of distance equal to the car's speed.

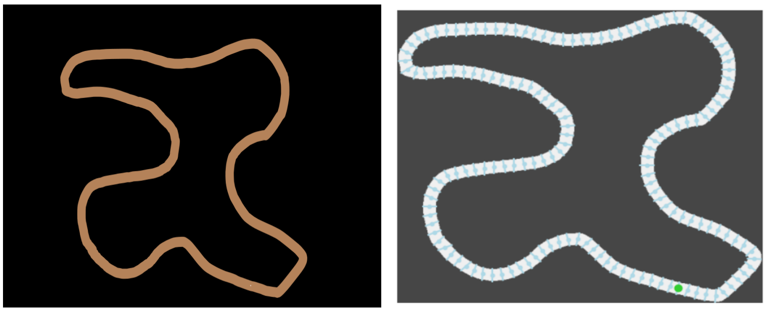


Figure 1: .png and computed track

If the coordinates of the car is (x, y), its speed is s and its rotation is α, then, after a time step, the coordinate of the car will be:

$$(s.cos(\frac{\pi}{180}\alpha) + x,\ s.sin(\frac{\pi}{180}\alpha) + y)$$

Moreover, at each time step, the car can make some actions:

- It can accelerate, this will increase the car's speed by a constant.
- It can brake, this will decrease the car's speed by reduce the car speed by a constant. The car cannot have a negative speed.
- It can turn, i.e. add a constant $\in [\![-K, K]\!]$ to its rotation. K is a constant that is the maximum angle the car can turn per each time step.

The behaviour of the car will need to interact with the track therefore we need to decide what is the state of a car, i.e. how the car see the environment. We could give to our algorithms the track matrices and the informations of the car but this will leed to to many parameters because a track can have size 900×600. Therefore we will need to train on all possible state wich will be at least $2^{900\times 600}$. Therefore, we decided to give a more realistic state wich represent how a car racer see. Then the state of a car is a array T of size 8.

- T_0 is the current speed of the car
- $\forall i \in \{1, ..., 7\}$, T_i is the distance of the car to the next wall in the direction $\alpha + A_{i-1}$ where α is the current rotation of the car and $A = [60, 40, 20, 0, -20, -40, -60]$

Then, the representation looks like figure 2.

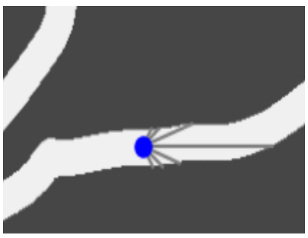


Figure 2: Car state

Technical aspects of the environment

To manipulate our environment, we use the python packages `gymnasium` which provide code convention for those type or environment, i.e. environment where at each time step, you have one action to do. The environment has to have some essential function: `reset()` that reset the environment to be able to do an other simulation, `render()` that render the current state of our environment and the most important one is `step()` that do one step of time, i.e. given an action, the `step()` function figure out is the car has crashed or not, move the car to its next position and return the new state of the car, a reward and if the car has crashed.

Our environment has a variable named `time` which give us the opportunities to discreet more or less the time.

Rewards

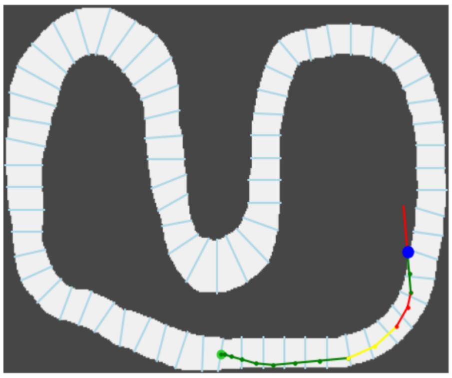
For those type of problem where the AI model has to compute a behavior, the AI model produces something which look like a function f that take a car state and return an action. We need to specifies to our AI model when it produce a good action and a bad action, for instance, if a car crash, we need to punish the AI model.
We do that thanks to a function reward implemented in the function `step()` of our environment. The reward is an integer, the bigger it is the best the action was. The function might be the most important one of all the project because it is thanks to it that our AI model will perform well or not. We try lot of reward function and we finish by using the following one. To punished the car when it do something bad we do:

- If the car crashes, we stop the simulation and return a reward of -500
- If the car is not moving, i.e. has a speed of 0, the reward is -10

For the positive reward, we have automatically computed some track line (represented in right image of figure 1). If the car crosses next line, it has a reward of ($+10\times$ the number of lines it has cross in the good order with this action). If the car cross a line in the wrong order, it means that it has gone backward, therefore, we punished the car with a reward of -200 and we stop the computation.
On top of that, at each time step, we add to the current reward the speed of the car divided by a constant to encourage the car to go fast.

An example of a car on a track

After explaining all of this, here is an example in figure 3. We have plot the trajectory of the car, the green color is when the car has accelerate, red color when it brake and yellow otherwise.



List of rounded reward: `[10, 1, 2, 12, 13, 13, 13, 14, 24, 14, 13, 12, 13, -497]`

Figure 3: Car state

The total reward of a car behaviour is the sum of all reward of a simulation with a car behaviour. For example, the reward of the car behaviour or the figure 3 is -343.

Deep Q-learning

Deep Q-Learning is a reinforcement learning algorithm that combines Q-Learning with Deep Learning to solve complex decision-making problems. It allows an agent to learn how to act optimally in environments with large state spaces by approximating a function, known as the *Q-function*, which evaluates the quality of an action taken in a given state.

Q-function

The Q-function, $Q(s, a)$, represents the expected cumulative reward an agent will receive after taking action a in state s, and then following the optimal policy. The cumulative reward is computed as:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a'),$$

Where:
- r is the immediate reward received after taking action a in state s.
- s' is the next state reached.
- a' is the next action.
- $\gamma \in [0, 1]$ is the discount factor, which balances immediate and future rewards.

Key Techniques

- **Replay Buffer**: A memory that stores past experiences (s, a, r, s'). Randomly sampling experiences from the buffer during training reduces correlations between consecutive samples, improving learning stability.
- **Exploration-Exploitation Balance**: The agent uses an ϵ-greedy policy to choose actions, where it explores randomly with probability ϵ and exploits the best-known action otherwise.

High-Level Workflow

- Observe the current state s.
- Choose an action a using an ϵ-greedy policy.
- Execute the action, observe the reward r and next state s'.
- Store the experience (s, a, r, s') in the replay buffer.
- Sample a mini-batch of experiences from the buffer to train the Q-network.

Genetic algorithms

What are genetic algorithms?

Genetic algorithms (GA) are probabilistic algorithms based on natural selection. Therefore, GA takes some populations which are sets of solutions (here a solution is a car's behavior), select the best solutions thanks to the reward function. Then, it changes the population by adding new random solutions, adding some mutations which are some small variations of a behavior, adding some cross-over which are the equivalent of natural reproduction. We can either repeat this process a fixed number of generations or for a fixed amount of time.

Markov Chain modelisation

We will now introduce a Markov chain modelisation to genetic algorithm. We define a Markov chain $(Y_n)_{n \in \mathbb{N}}$ as following:
- A state of $(Y_n)_{n \in \mathbb{N}}$ is a population.
- Let y_0 be a special state such that if $Y_n = y_0$ then it means that the population of state Y_n contain an optimal solution.

Now, the sequence of population of genetic algorithm can be describe with this Markov chains. Y_n represent the population at generation n. Notice that the state y_0 is an absorbing state. In fact if $Y_n = y_0$ then it mean that the population P_n contain an optimal solution. Since we always keep the best solution of the previous population, it means that $\forall n' > n$, we have that $P_{n'}$ contain an optimal solution. Therefore, $\forall n' > n$, $Y_{n'} = y_0$. Moreover, y_0 is the only absorbing state of $(Y_n)_{n \in \mathbb{N}}$.

If we suppose that our mutation and cross-over are made such that a solution x can reach y by a series of a finite number of those operations. Then, all solutions x can reach an optimal value. Then

every state y_n can reach state y_0. The set of all possible state is finite. Then $\mathbb{P}(Y_n = y_0) \underset{n\to+\infty}{\rightarrow} 1$
Then $\mathbb{P}($ The population P_n contain an optimal solution$) \underset{n\to+\infty}{\rightarrow} 1$
Thus, the genetic algorithm converge toward a global optimal solution. However, we do not know how many time it will take in average.

NEAT

Basic genetic algorithms are not efficient enough to compute an optimize behavior. Therefore, we will use the famous python packages called `NEAT`. It is an optimized generalized genetic algorithms with represent solution as dynamic neural network. By dynamic we mean that the algorithm can add or delete some of the nodes of the neural network. The principle of GA stay the same but we have a lot more hyper parameters.

Simulation

Once we completed the modeling of the environment, the deep Q-learning algorithm and the genetic algorithm, we have to compute some simulation to process evaluation performance. We will compare our two algorithms using various metrics. For this purpose, we choose the following metric: the average reward after training depending of some parameter, we call it the score of a training. We measure this value across different training durations and varying the numbers of tracks used to training the models. To ensure robustness and evaluate potential over-fitting, we test the models on tracks that were not included in the training set.
This approach is applied in the context of an AI project where we train AI agents to complete a racing circuit. The goal is for the cars to complete laps as quickly as possible, and the average reward reflects their performance under these conditions.

- First, we will compare genetic algorithm to Q-learning depending of the training time and the number of tracks used to train the car. We we train our algorithms during 10 to 60 minutes and with 10 to 67 tracks (which represent 80% of the total number of tracks created).
- Then, we will compare the result of deep Q-learning depending of some hyper-parameters which are, the goal here is to be able to find the hyper-parameters that fit the most the situation, and for us this evaluation is only possible by running the algorithms with different parameters.
- Finally, we will evaluate the performance or the best car we found using deep Q-learning with a training of six hours, with the best hyper-parameters.

Experimental

All the computations presented in this section were performed on the Grid5000 infrastructure, which allowed us to ensure the reproducibility of the results and guarantee a consistent comparison of the executions, as they were carried out on machines with equivalent performance and assure stability of our executions.

Comparison between Deep Q-Learning and Genetic Algorithm

Firstly, our goal was to be able to compare the different models we tried to use to train the car, in order to do that, we choose to give a certain time and a certain amount of tracks for the different models. The models were allowed to be training for this amount of time on Grid5000.
We choose to evaluate the models on these metrics because we believed that they were the most important in the training of such AI. Indeed, these metrics allow the users to have an idea on how powerful can the models be.
As said before, we choose to evaluate the average reward on tracks on which the car has not trained, we believed it is the most relevant way to evaluate the training because it allows us to evaluate if there

is over-fitting or not and this represent how much the car is doing well on the track.

Since the training can be long to have satisfying result, we choose to concentrate ourselves on the following durations : $\{10; 40; 60\}$ (mins) and the following number of tracks for the training $\{10; 40; 67\}$ (tracks). We evaluate these trainings on 20 tracks that are not in the training set. The two models are run with specific hyper-parameters for this section that are described in the final section 9. The result are in figure 4.

Figure 4: Comparison between the Genetic algorithm and the Deep Q learning algorithm.

We can notice that the Deep-Q algorithm outperform the Genetic algorithm for every test. We can also notice that sometimes the use of more time or more tracks can lower the performance. This can be due to over-fitting.

To compare the models, we can also look at the global volatility of the solutions, this represent the standard deviation of score of the training. In order for the algorithm to perform efficiently, we want it to to have low standard deviation. The result are shown in figure 5.

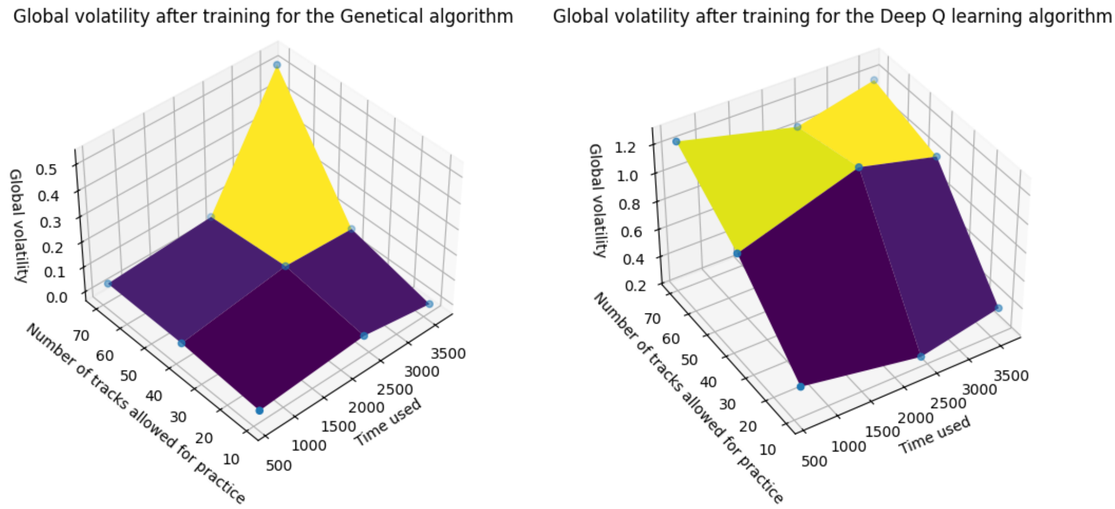


Figure 5: Standard deviation after training

Finally, we can look at the evolution of the reward for the Deep Q learning for 60 minutes of training on 67 tracks to have an idea the possible over-fitting happening for this training (figure 6). We can see that the reward does start decreasing after 650 generations, this can be a sign of over-fitting.

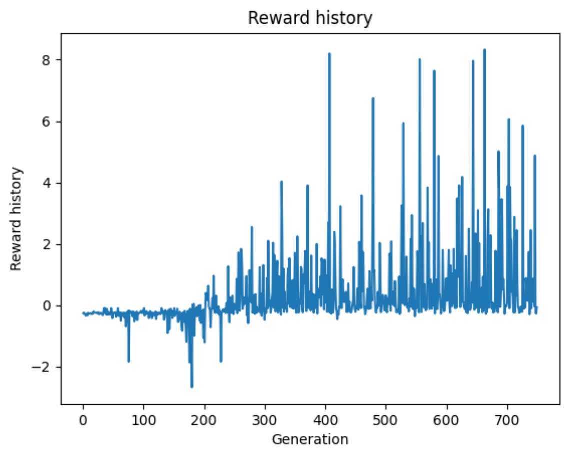


Figure 6: Evolution of the reward

Influence of Hyper-parameters on Deep Q-Learning

Focusing on the Deep Q algorithm, we can ask ourselves what are the best hyper-parameters. We focused on 3 hyper-parameters that we perceived as more important in our model

- Batch size : this is the number of samples that are kept in memory between each step of the training.
- Lr (learning rate) : this is the velocity at which the neural network update the weights.
- Epsilon Decay : this is the probability that the model will try to explore new ways.

We get the following results (figure 7). These graphs allows us to have an idea on how efficient each hyper-parameter tested is. For example for batch size, we want to have a batch size of 60 in order to be efficient.

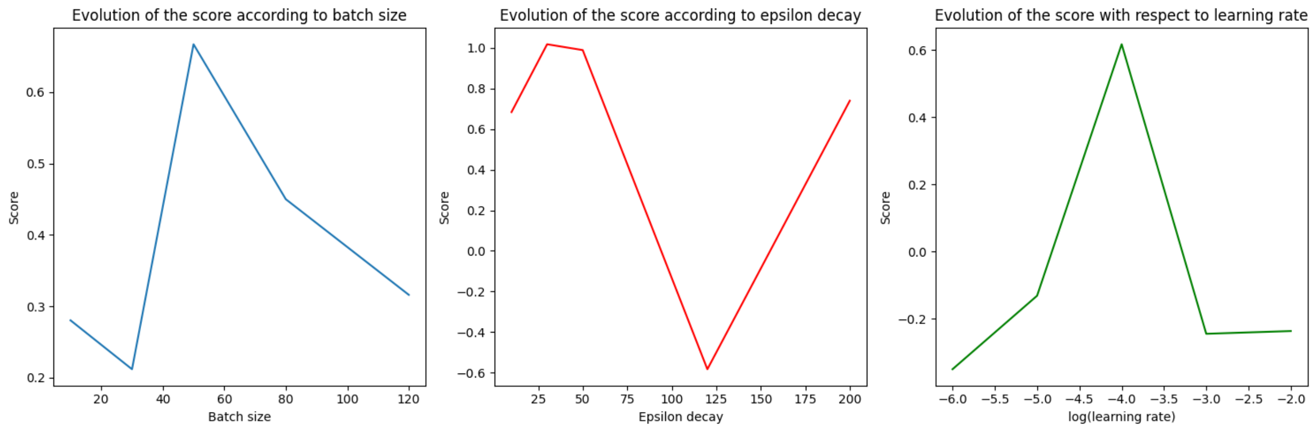


Figure 7: Evolution of the score after fluctuating hyper-parameters

Best Car: performances of a model trained for few hours

We can now look at the performance of the best car, it is chosen on how efficient it is on the tracks. Firstly, as seen in part one, training for too long on a reduced amount of tracks can lead to over-fitting, we have been able to see this also by training a model for 6 hours in the same conditions as the other. This model performs way less than the others. We choose the Hyper-parameters chosen due to part 2 and the time and number of tracks that maximise the score. You can see on figure 8 the evolution of the car on a map that the car has never seen before.

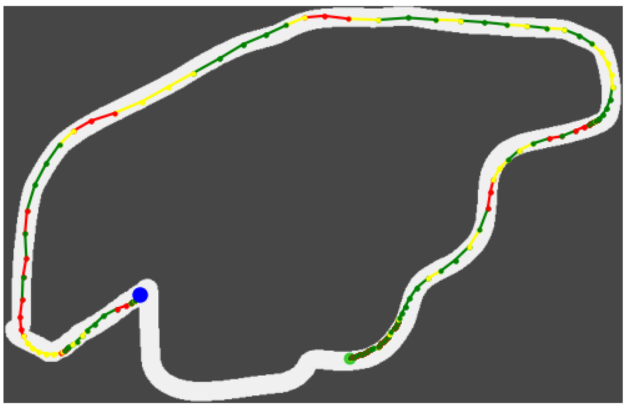


Figure 8: Evolution of the best car

Conclusion

Annexes

Reproducibility

This section store the data necessary to reproduce the experiments:

Comparison between the Genetic algorithm and the Deep Q learning algorithm.

Hyper-parameters for the genetic algorithm for part 1.

```
[NEAT]
fitness_criterion      = max
fitness_threshold      = 10000
pop_size               = 500
reset_on_extinction    = False

[DefaultGenome]
# node activation options
activation_default       = tanh
activation_mutate_rate   = 0.2
activation_options       = sigmoid tanh relu

# node aggregation options
aggregation_default      = sum
aggregation_mutate_rate  = 0.0
aggregation_options      = sum

# node bias options
bias_init_mean           = 0.0
bias_init_stdev          = 1.0
bias_max_value           = 30.0
bias_min_value           = -30.0
bias_mutate_power        = 0.5
bias_mutate_rate         = 0.7
bias_replace_rate        = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob            = 0.5
conn_delete_prob         = 0.3

# connection enable options
enabled_default          = True
enabled_mutate_rate      = 0.01

feed_forward             = True
initial_connection       = full
#full_nodirect

# node add/remove rates
node_add_prob            = 0.2
node_delete_prob         = 0.2

# network parameters
num_hidden               = 0
num_inputs               = 8
num_outputs              = 9

# node response options
response_init_mean       = 1.0
response_init_stdev      = 0.0
response_max_value       = 30.0
response_min_value       = -30.0
response_mutate_power    = 0.0
response_mutate_rate     = 0.0
response_replace_rate    = 0.0

# connection weight options
weight_init_mean         = 0.0
weight_init_stdev        = 1.0
weight_max_value         = 30
weight_min_value         = -30
weight_mutate_power      = 0.5
weight_mutate_rate       = 0.8
weight_replace_rate      = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 15
species_elitism      = 3

[DefaultReproduction]
elitism            = 3
survival_threshold = 0.2
```

Hyper-parameters for the Deep Q learning algorithm for part 1.

```
batch_size = 40
epochs = 5000
max_episode_duration = 1000 * 1/env.env.time
epsilon_max = 1
epsilon_min = 0.01
epsilon_decay = 30.
lr = 1e-4
discount_factor = 0.9
self.model = DQN(400,8,self.n_action)
```