

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 every possible 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 hyper-parameters.
- 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 hyper-parameters 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 as shown by the first image of figure 1. Then, the image is converted to a matrix T such that $T[0][0]$ is the bottom left corner. Finally, we crop the image, compute the starting point, and determine the lines of the track (these will be explained in the reward section) to obtain the final result shown by figure 1.

Car's physics

The car's physics model is quite simple, it follows a 2D cartoon-like set of physical laws :

- The car is characterized by two main properties: its speed $s \in [0, \text{MaxSpeed}]$ and its direction $\alpha \in [0, 360]$
- The physical laws work as follows: at each time step, the car moves in the direction defined by α , covering a distance equal to its speed.

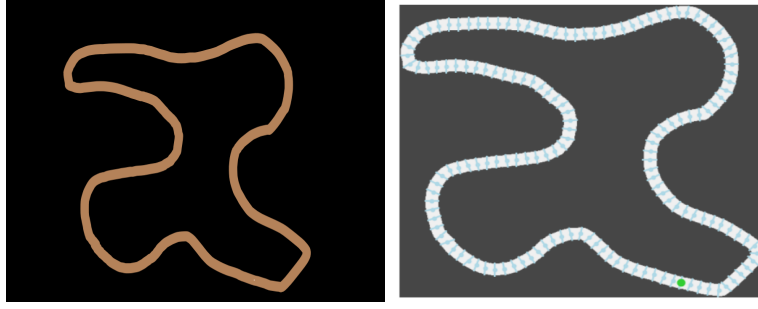


Figure 1: Before and after processing the track : the white squares represent the road, and the green point represents the starting point

- If the current coordinates are (x, y) , its speed is s , and its direction is α , then after one time step, the new coordinates of the car will be:

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

Moreover, at each time step, the car will take an action:

- It can accelerate, this will increase the car's speed by a constant.
- It can brake, this will decrease the car's speed by a constant. Note that the car cannot have a negative speed, it means that it cannot go backward.
- 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 time step.

The car needs to interact with the track, therefore we need to decide what is the state of a car, i.e., what the car knows about the environment. We could model a state by the matrix of the track and the information of the car but this would lead to a state of large dimension because a track can have size 900×600 pixels. Moreover, it will be hard to train from those states because there is a great diversity of possible tracks. To reduce the dimension of the problem, we decided to represent a state as a real car racer would see it. Then, the state of a car is an array V of size 8 :

- V_0 is the current speed of the car
- $\forall i \in \{1, \dots, 7\}$, V_i is the distance of the car to the closest wall in the direction $\alpha + A_{i-1}$ where α is the current direction of the car and $A = [60, 40, 20, 0, -20, -40, -60]$

Then, a typical state is represented by the figure 2.

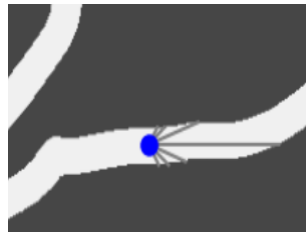


Figure 2: Car's state

Technical aspects of the environment

To manipulate our environment, we use the python package **gymnasium** which provides conventions for Markov decision process environments (i.e. environment in which at each time step you have to take an action and receive a reward). The environment must have some essential methods: **reset()** that resets the environment in order to begin a new simulation, **render()** that renders the current state of our environment, and the **step()** function which moves the car to the next position and return the

new state, a reward and a boolean representing if the car has crashed or not. Finally, the environment has a variable `time` which measures how much time is discretized.

Rewards

Machine Learning algorithms usually try to optimize an objective function that reflects the problem that the real application wants to solve. In our case the objective function will depend on the rewards that the agent will observe when taking actions. Typically, the reward will be low if the car crashes and high if the car is fast. The reward is computed by the method `step()` of our environment. Clearly, the quality of the training highly depends on quality of the reward function. We tried lot of reward functions, some that we invented, other that we saw in other projects and ended up using the following one:

To discourage bad choices we do:

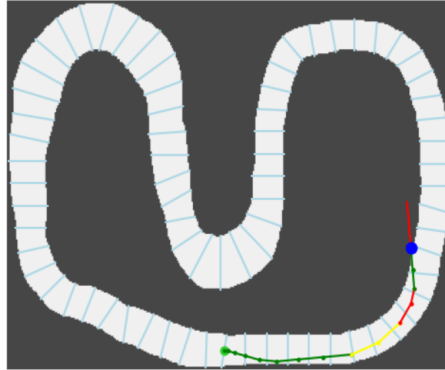
- If the car crashes, we stop the simulation and return a reward of -500 .
- If the car is not moving, (i.e. has a speed of 0) we return a reward of -10 .

For the positive reward, we have automatically computed some track lines thanks to an algorithms we created (represented by the second image of figure 1). If the car crosses next line, it receives a reward of $(+10 \times \text{the number of lines it has crossed with one action})$. If the car crosses 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. Finally, we subtrack the reward by 1 to encourage the car to cross has many lines has possible with the least amout of time steps.

An example of a car on a track

Figure 3 shows an example of a car on a track. We plotted the trajectory of the car : the green color means that the car is accelerate, red color means that the car is braking, and yellow represents all the other actions.



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

Figure 3: Car's trajectory

The total reward of a car behavior is the sum of all reward of a simulation with a car behavior. For example, the reward of the car behavior of the figure 3 is $\sum_{r \in R} r = -343$.

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 Q-function satisfies the Bellman equation:

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

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 modelization

We will now introduce a Markov chain modelization for 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 chain. Y_n represents 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 the state y_0 . The set of all possible state is finite. Then $\mathbb{P}(Y_n = y_0) \xrightarrow{n \rightarrow +\infty} 1$. Then $\mathbb{P}(\text{The population } P_n \text{ contain an optimal solution}) \xrightarrow{n \rightarrow +\infty} 1$.

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 optimized policy in a reasonable time. Therefore, we will use the a python package called **NEAT**. This is an optimized generalized genetic algorithm which represents solutions 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 trained several models and evaluated their performances. We compared our two algorithms using several metrics. For this purpose, we chose the following metric: the average reward after training depending of some parameter, we call it the score of a training. We measured this value across different training durations and varying the numbers of tracks used to train the models. To ensure robustness and evaluate potential over-fitting, we tested the models on tracks that were not included in the training set. The goal of the car is to complete laps as quickly as possible, and the average reward reflects their performance under these conditions.

We will evaluate the performances of the Deep Q-Learning algorithm in three steps:

- First, we will compare genetic algorithm to Deep Q-Learning depending of the training time and the number of tracks used to train the car. We will train our algorithms during 10 to 60 minutes and with 10 to 67 tracks (which represents 80% of the total number of tracks created).
- Then, we will compare the result of Deep Q-Learning depending of some hyper-parameters. The goal here is to be able to find the hyper-parameters that fit the most with our problem, and this evaluation can only be made by running the algorithms with different parameters.
- Finally, we will evaluate the performance of the best model we found using Deep Q-Learning.

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. They were carried out on stable machines with equivalent performances.

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 chose 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 chose to evaluate the models on these metrics because we believe that they are 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 chose to evaluate the average reward on tracks on which the car has not trained, we believe 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 represents how well is the car doing on the track.

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

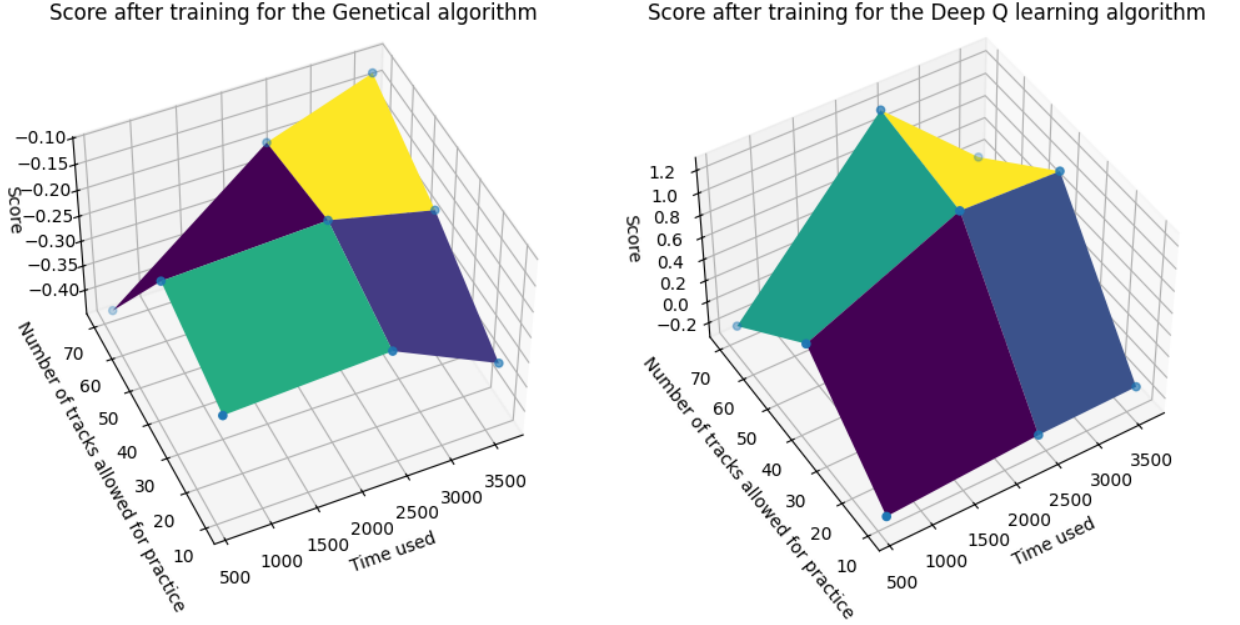
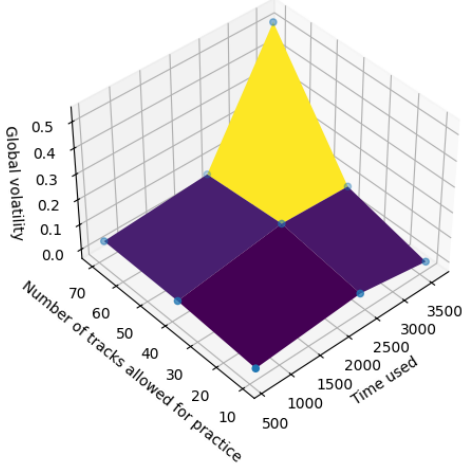


Figure 4: Comparison between the Genetic algorithm and the Deep Q learning algorithm.

We can notice that the Deep-Q algorithm outperforms the Genetic algorithm in every test. Also, we notice that sometimes the use of more time or more tracks can lower the performance of our Deep Q-Learning algorithm. This is surely due to over-fitting.

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

Global volatility after training for the Genetical algorithm



Global volatility after training for the Deep Q learning algorithm

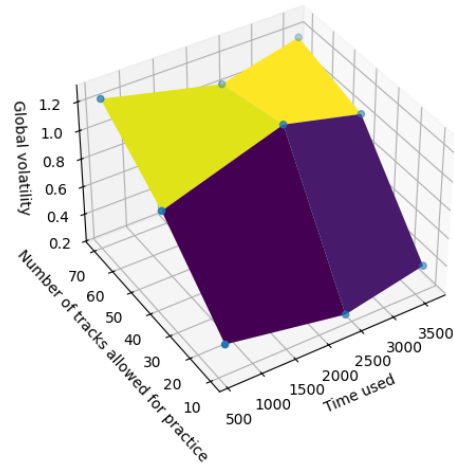


Figure 5: Standard deviation after training

Finally, we can look at the evolution of the reward for the Deep Q-Learning for 60 minutes of training on 67 tracks to estimate when is 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.

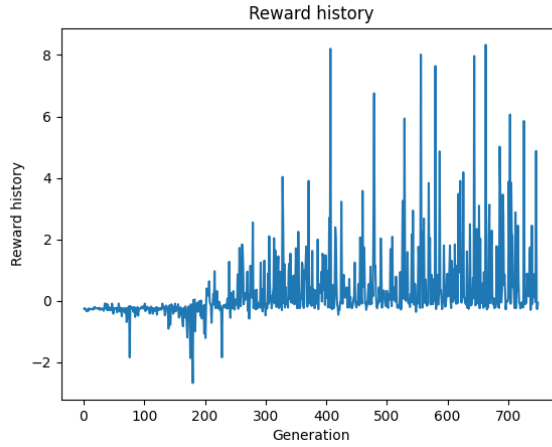


Figure 6: Evolution of the reward

Influence of Hyper-parameters on Deep Q-Learning

Focusing on the Deep Q algorithm, we can wonder 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 used to optimize the model's parameters for each optimization step.
- Lr (learning rate) : this is the velocity at which the neural network updates 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 the batch size, we want to have a batch size of 60 in order to be efficient.

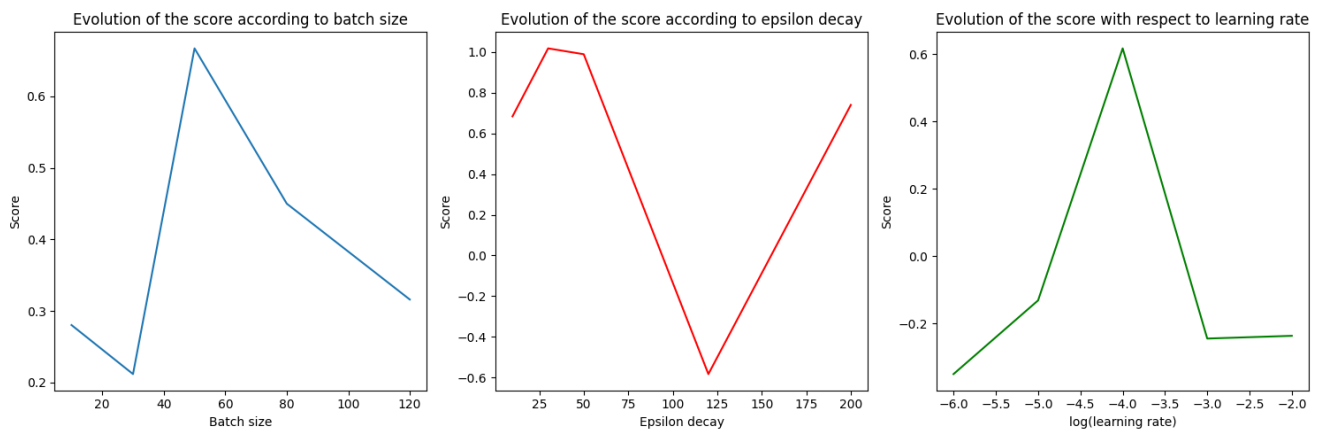


Figure 7: Evolution of the score according to hyper-parameters

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 thanks to part 2 and the time and number of tracks that maximize the score. You can see on figure 8 the evolution of the car on a map that the car has never seen before.

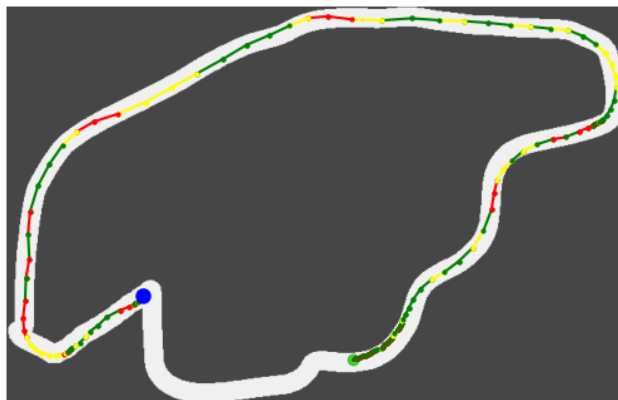


Figure 8: Evolution of the best car

Conclusion

The goal of the project was to train an AI model to choose the best trajectory for a car on random tracks. To carry out this project, we created an environment to model the evolution of a car on different tracks. Then, we experimented with different types of training algorithms. In particular, we selected the genetic algorithm and the Deep Q-learning algorithm. Finding a relevant reward function for the algorithm was a key part of the work. Evaluating the algorithm then enabled us to determine the best hyperparameters for our models, as well as the optimal training time and number of tracks. Our goal was to compare the performance of the algorithms, and we believe that using the score and global volatility were relevant metrics. Initially, we aimed to compare the performance of the algorithms with real-world performances. This would have allowed us to maintain a connection to reality and to assess the relevance of our work. However, this proved to be much more challenging than expected, but it could serve as an interesting extension of our project.

Appendix

Reproducibility

This section stores 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_stddev      = 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_stddev  = 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_stddev    = 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)
```