Route Optimization with Genetic Algorithms in a Car Racing Game

Group B
André Dias (m20210668)
Beatriz Gonçalves (m20210695)
Guilherme Simões (m20211003)

NOVA Information Management School

Data Science and Advanced Analytics

Leonardo Vanneschi, Berfin Sakallıoğlu

Curricular Year - 2021/ 2022

Link for the repository: https://github.com/beatrizctgoncalves/project_cifo

Keywords: generation, neural network, car, lap, operator, method.

Introduction

The goal of the Racing Car Game is to represent in a small scale a certain object (car) driving around a randomly defined track to optimize the time that it takes to complete a full lap. The idea is to analyze the lines that he chooses and improve the overall performance in every generation. The experiment is developed in an already made pygame where we propose to implement various genetic algorithms combinations to see which best fits the desired solution.

Game Development

For this work we used a basic driving car game to explore the implementation of Genetic algorithms. We will generate a population consisting of **50 cars** driving along a specific track with each being a random initialized neural network. They contain an **input layer** with 5 neurons where it is constantly receiving the distances between the different car points and the walls of the track, a **hidden layer** with 6 neurons and an output layer with 5 **output** move instructions (accelerate, brake, turn right and turn left). As the next generation is created, the cars evolve by selecting and then applying genetic algorithm operators.

Implementation of GA in the game

Fitness function

The fitness function was implemented with 2 main variables that increment the fitness value of each car. The first one, increments based on the **velocity of the car** in every frame (30 frames per second). The second one increments the fitness as the cars pass by one of the **checkpoints** placed along the map. The fitness stops increasing if the car hits the wall (velocity = 0) or if the command to generate a new set of cars is activated.
Additionally in this project, we could have implemented a minimization problem by the fact that we could choose the cars with the best fitness according to the shortest time it would take them to do the entire track.

Selection

Our initial approach to selection was based on what we saw after each generation. When all cars complete their runs, we identify the ones who went further. Ideally, we select the best two. This way we will apply a manual **Rank Selection**.

The next approach was by using **Tournament Selection** where we select 10 cars from the population to the tournament. The third method was **Fitness Proportion Selection** (Roulette Wheel Selection) so it would add diversity to the selected cars.

Crossover operators

Regarding the crossovers, each of them will significantly determine the weights and biases of children based on the two selected parents. **Single point crossover** is done by copying the weights and biases from the parents and then selecting a random crossover point from the children array. This point marks the spot from where we swap the genes of the children. For example, if we randomly select point 3, we swap the children's genes from point 3 until the end.

The **arithmetic crossover** uses a formula to modify the weights and biases of the children: *child.weights[i][j][k] = parent.weights[i][j][k] * alpha + (1 - alpha) * parent.weights[i][j][k]* where alpha is a random value between 0 and 1.

While in the arithmetic crossover children's weights and biases are influenced by both parents plus a random alpha, the same doesn't happen for uniform crossover.

The **uniform crossover** treats each gene independently. In other words, each gene has a 50-50 chance of being included in the outputed genome. We biased this chance so that the two parents have the same influence in the children's weights and biases.

Mutation variations

It's important to explain that we first defined a **rate for the mutations**. This rate can be changed while the algorithm is running, but we found that the value that best fits our problem would be 80 mutations per car.

We have implemented 3 types of mutations: **One Gene, Inversion and Swap**. In the swap and inversion mutation we changed the genome weights and biases randomly according to the defined points. In the one gene mutation we select a random gene and then multiply its value by a random one between 0.8 and 1.2, meaning that it will decrease or increase, respectively.

Results

The initial weights and biases given randomly to the neural network will have a deterministic impact on the behavior of the next generation's performance. That's why we tested each combination multiple times so we can have a large enough sample.
The first crossover method tested was the **Single Point Crossover**. At the beginning, we can see that the cars are improving at a very low pace, hitting every wall as they slowly try to go through the track. They pass the first turn at generation 10 (highest pick in Figure 1) and then they get stuck in the second. The process repeats itself for several generations until we no longer see any clear improvement. This happens to be a case of premature convergence.

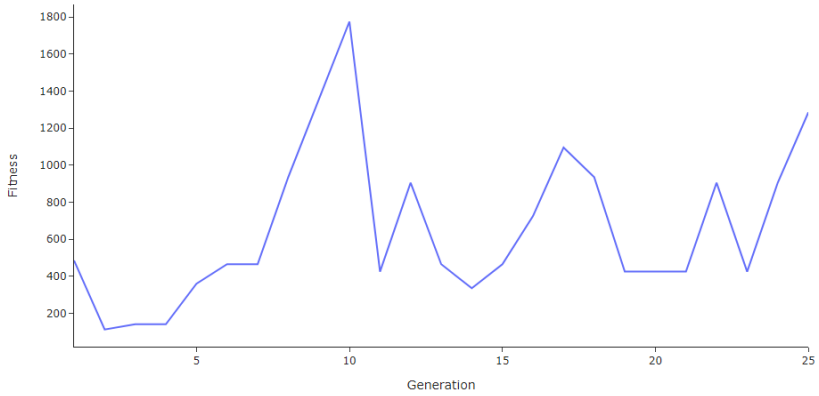

Figure 1 - Tournament + SinglePoint Crossover + OneGene Mutation

We can see in the figure 3 in the annexes, a comparison between different mutations, the **Swap and Inversion Mutations** with the same selection method and crossover operator. In terms of fitness, they both have a similar performance. At close, after seeing the cars behavior, we identify that in both solutions, the majority of them crash immediately against the lateral walls as if the solution was to initially do a turn either right or left.

Given the same CrossOver method (we will assume the Uniform crossover), we compared the **Tournament Selection with the Fitness Proportion Selection** (figure 4). Depending on which car was selected, the arithmetic crossover would create two different groups of cars with similar actions. On the other hand, if the Tournament selection was used, we would have the two cars with the most fitness and we could clearly see one big group of cars.

We got the best combination when using **Rank Selection, Uniform CrossOver and One Gene Mutation**. The arithmetic crossover also completed a full lap, but it took 2 more

generations to do it as shown below in the figure 5. We can see a slight pick in one of the first iterations meaning that the car was gaining fitness by driving very slowly.

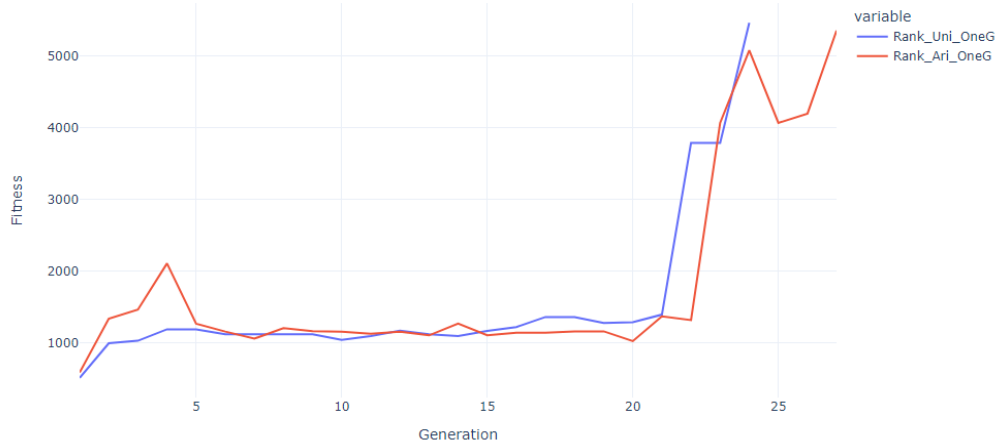

Figure 2 - Rank + Arithmetic Crossover + OneGene Mutation VS Rank + Uniform Crossover + OneGene Mutation

Discussion

In every generation, the weights and biases are all passed to two distinct genomes in the crossover and mutation processes. Depending on which type of operator we choose, the **genome of 60 weights** is changed and finally passed onto the following population.
As we already know, the neural network performance depends largely on the combination of weights and biases in each neuron. This means that if we select the best cars (Rank Selection) as parents for the next generation, it is **most likely that they will perform better**. Having that in mind, we chose two Crossovers that would not have significant impact on the NN configuration, but would instead increase the value on certain weights so that we could accentuate the more deterministic values and consequently their output. The same happened with mutations, where the One Gene Mutation had a better performance compared to the rest, due to the fact that the **majority of the cars followed a similar path according to their parents genome**. Instead of increasing the weights and biases values, the Inversion and Swap mutations would randomly change them resulting in a worse performance. In terms of practical visualization, this resulted in a very sparse population where we would have a variety of cars turning at an early stage when the track has a straight line at the beginning. Nonetheless, **mutation plays a significant role in the evolution of the genetic algorithm**. Without it, the cars would all follow the same path as the parent, and we wouldn't see many changes in the fitness values.

Future Work

Instead of having a population of 50 cars, we could have experimented with a higher number, since it would not only give us more options when selecting the parents cars for the next generation, but also could solve the problem of premature convergence in some scenarios. Despite being more computationally demanding, the algorithm would be able to identify faster the optimal trajectory for the car.
Studying the different routes that the best car is taking in every generation through the drawing of a path line, would help us understand the decision making behind the neural networks decisions. Now that many cars can complete a full lap in a specific track, it would be relevant to experiment with different tracks to compare the model efficiency.
Finally, by adding obstacles along the track or increasing even more the angles of the curves, it would increase the difficulty of the game and subsequently the complexity of the neural network.

Conclusion

Having the possibility to compete with the best car around the track, allows us to understand the efficiency of a neural network behind a certain object. In just a few generations, with only one hidden layer, our best NN will beat us every time with the exact combination of acceleration, breaking and turning depending on the track configuration. The solution implemented focused on route optimization where the goal of completing a full lap was achieved. However, the same principles applied in this situation can also be implemented in similar projects, such as solving different types of mazes or even a snake game. Being able to apply Genetic Algorithms to a real game, allowed us to better understand the concepts while visualizing the different results in real time.

Percentage Work Distribution: André Dias - 33%; Beatriz Gonçalves - 33%; Guilherme Simões - 33%.

References

"Genetic Algorithms - Fundamentals",
https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals.htm

"Crossover Operator — The Heart of Genetic Algorithm", https://medium.com/@samiran.bera/crossover-operator-the-heart-of-genetic-algorithm-6c0fdcb405c0

Source code available at https://github.com/ReadySetPython/Neural-Network-Cars

Annexes

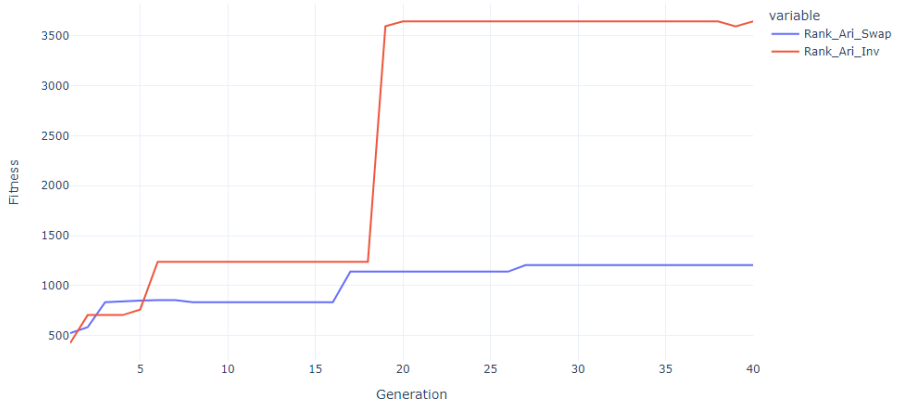

Figure 3 - Rank + Arithmetic Crossover + Swap Mutation VS Rank + Arithmetic Crossover + Inversion Mutation

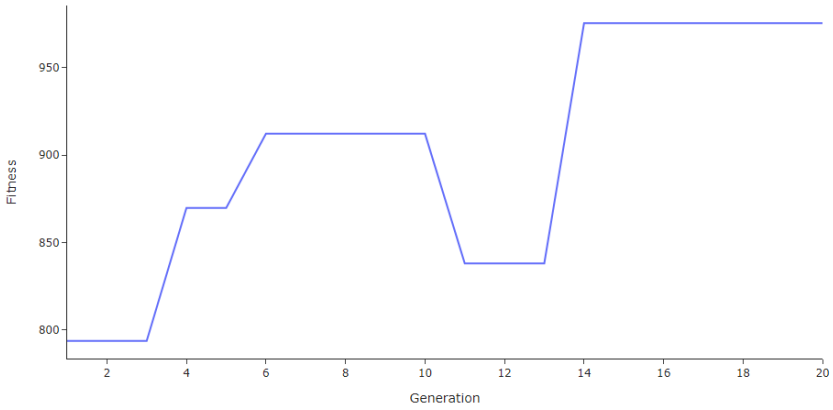

Figure 4 - Tournament + Uniform Crossover + Inversion Mutation

Generation	Rank + Uniform + OneGene	Rank + Arithmetic + OneGene
1st	513.80	587.33
2nd	995.20	1338.03
3rd	1030.00	1463.47
4th	1189.20	2106.67
5th	1189.20	1268.37
...	...	...
20th	1288.00	1024.76
21th	1394.00	1369.37
22th	3787.80	1317.37
23th	3787.80	4065.34
24th	5459.60	5077.03
25th	-	4065.34
26th	-	4194.99
27th	-	5352.19

Table 1 - Best Final Combinations