Beating Snake Through the Power of Evolution

Gustavo Brito (r20170760@novaims.unl.pt), Mariana Albernaz (r20170785@novaims.unl.pt), Marta Santos (r20170770@novaims.unl.pt), Miguel Mateus (r20170752@novaims.unl.pt)

1. INTRODUCTION

The gaming industry has been exploring and taking advantage from several features of intelligent systems. Particularly, from Reinforcement Learning and Genetic Algorithms, based on the theory of evolution of Charles Darwin. In this report, it is shown how a Genetic Algorithm was implemented and tested to create a system capable of proficiently playing an adaptation of the classic Snake Game. The code used for this project can be found here.

2. METHODOLOGY

2.1 Implementation

The adaptations made to the classic Snake Game were few, there are always four apples on the board and we choose that the snake starts with a length of three (including the head), other than that, the snake is free to move around the board, if it eats an apple it grows by one square and if it collides against itself or a wall it dies. The game board size is 10x10, but it can be easily changed in the code by adjusting one parameter. In each playthrough, the snake has up to 100 moves* to eat its next apple, otherwise it dies. This rule was created to prevent an individual, particularly in the early generation, from roaming around the board endlessly.

Each individual in the population is represented by a list of weights of size k, where k is the total number of weights of the chosen neural network for the algorithm †. In the first generation, for each individual, all of the weights are randomly initialized through a uniform distribution within the range [-1, 1]. The neural network chosen to control the snake has two hidden layers, each with 15 neurons and with the Tanh activation function, takes as input a flattened window of size 7x7 around the snake's head (what the snake can see) and produces four binary outputs, which dictate the next move of the snake (either up, down, left or right). The number of weights in each layer and the size of the window around the snake's head are two parameters that can be changed in the code of the project. Each square on the window that dictates the snake's vision has a value of either 1, -1, or 0 (1 if an apple is placed there, -1 if it is either the snake or a square outside of the board and 0 otherwise).

Throughout each game, all of the apples' positions are predetermined. Initially, the apples would appear in random positions, however, no matter which configuration and parameters of the genetic algorithm were experimented, seemingly no evolution or ability to play the game was demonstrated by the algorithm. As such, in order to remove some randomness in the evolution, the choice was made to fix the apples' positions, search for a configuration that could achieve good results and then see if those good results could be transferred back to an implementation with random apple positions. Several fitness functions were experimented, but only two extensively: $F(x) = score + 1$‡ and $F(x) = moves + ((2^{score}) + (score^{2.1}) * 500) - ((score^{1.2}) * (0.25 * moves)^{1.3})$, where score is equal to the number of apples an individual was able to collect in one playthrough of the game and moves is the total number of moves an individual made during one playthrough of the game.

On both fitness functions, a high fitness is desired, meaning we are dealing with a maximization problem, but the code also works for minimization problems by changing the respective parameter.

The first fitness function was chosen due to its simplicity and ease of implementation in the early stages of the development of the project. With this fitness function, there could be various global optima with a known fitness value of 98 (maximum score + 1, where maximum score = 100 – 3 = 97, meaning, the total number of

*a snake will take 100 moves to reach from one corner of the board to the opposite, passing by all the squares once.

†for implementation purposes, in the code the individual's representation is already formatted into an array that organizes the weights according to their specific layer and neuron

‡if fitness was exactly equal to the score, it could be zero, and obvious problems would arise in Fitness Proportionate Selection

squares on the board minus the initial length of the snake plus one). The second fitness function was adapted from another implementation of genetic algorithms to solve the Snake Game, with the purpose of rewarding individuals who explored the game board in the early generations of the algorithm, but penalize such behaviour in the last generations, where presumably more apples would be consumed.

Fitness sharing and some measures to quantify the diversity in the population were implemented, these being phenotypic variance, phenotypic entropy, genotypic variance and genotypic entropy. When evaluating different configurations of parameters, entropy was taken more into account, since different fitness functions with very different fitness ranges were implemented, which consequently influenced the respective variance values and did not permit fair comparisons. Elitism was used, it is possible to decide whether to use elitism and to change the number of elites by changing the respective parameters in the code of the project and three different selection methods were tested, Fitness Proportionate Selection, Tournament Selection and Rank Selection.

2.2 Genetic Operators

Three different crossover and mutation operators were implemented.

2.2.1 Crossover

Arithmetic Crossover – Performs a linear combination of two parents in order to produce two offsprings. Given two parents, [x1, x2, ..., xm] and [y1, y2,..., ym], two offsprings are generated as follows, [α x1 + (1-α)y1, α x2 + (1-α)y2, ... , α xm + (1-α)ym] and [α y1 + (1-α)x1, α y2 + (1-α)x2, ... , α ym + (1-α)xm]. Where α is a number randomly selected from a uniform distribution with a range of [0, 1].

Geometric Crossover – Similar to the latter. Given two parents, [x1, x2, . . . , xm] and [y1, y2, . . . , ym], two offsprings are generated: [r1*x1 + (1-r1)*y1, r2*x2 + (1-r2)*y2, ..., rm*xm + (1-r1)*ym] and [r1*y1 + (1-r1)*x1, r2*y2 + (1-r2)*x2, ..., rm*ym + (1-r1)*xm]. Where, for each i=1,2,. . . ,m, Ri is a number randomly selected from a uniform distribution with a range of [0, 1].

Simulated Binary Crossover – Simulates the single-point crossover operator of binary-coded GA's. Given two parents x and y, a random number u is generated from a uniform distributed with a range of [0,1]. Then, β is calculated as follows:

$$\beta = \begin{cases} (2u)^{\frac{1}{\eta_c+1}}, & \text{if } u \le 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise} \end{cases}$$

where η_c is the distribution index. A large η_c tends to generate children closer to the parents and a small η_c allows children to be generated farther away from the parents. In this project η_c was always equal to 2. Two offsprings are then generated, such that offspring1=0.5*[(1+β)*x + (1- β)y] and offpsring2=0.5*[(1-β)*x + (1+β)*y].

2.2.2 Mutation

Random Mutation – Given one individual [x1, x2, . . . , xm], a new one is generated such that for each i=1,2,. . . ,m, xi has a probability of α of being replaced with a random number taken from a uniform distribution. In this project, the α chosen was 0.1, and the uniform distribution range was set to [-1, 1], equal to the random initialization of the individuals.

Geometric Mutation – Given one individual [x1, x2, . . . , xm], a new individual is generated as follows: [x1+R1, x2+R2, . . . , xm+Rm]. Where, for each i=1,2,. . . ,m, Ri is a number randomly selected from a uniform distribution with a range of [-ms, ms], and mutation step (ms) is a hyperparameter that can be tuned.

Gaussian Mutation – Similar to the previous mutation, the only difference being that, for each i=1,2,. . . ,m, Ri is a number randomly selected from a gaussian distribution with zero average and a standard deviation equal to α, a hyperparameter that can be tuned.

2.3 Evaluation Criteria

In order to compare different configurations of the algorithm, each one was always executed 5 times, with 100 individuals in the population and with 500 generations. Usually, the average fitness of the best individual in each generation across all runs and its standard deviation were analysed, but when required, average phenotypic and genotypic entropies across all generations and runs were also studied.

3. RESULTS

Ideally, at least initially, a complete combination of all possible genetic operators would be experimented, however, due to computational reasons, that was not possible. Instead, some common rules of thumb were followed to decide which operators to start with and which to begin optimizing, and a higher importance was given initially to the stability of the algorithm between the five runs, instead of the highest achieved average fitness.

At first, with the fitness function as $F(x) = score+1$, elitism with one elite, tournament selection (tournament of 20) and geometric mutation (mutation rate of 0.1 and mutation step of 0.25), all three crossover operators were compared.

In terms of average score and its evolution, the different operators presented similar results, however, geometric crossover presented the highest standard deviation, while the other two showed a more consistent score and evolution across all runs. Simulated Binary Crossover seemed to be the most consistent (smallest standard deviation range) and it also showed a clearer improvement in score, specifically in the first 200 generations and therefore for now, that was the chosen crossover operator. It is worth noting that all standard deviations showed some overlap, something that is prevalent in almost all tests performed in this project, meaning this difference in average score and consistency across runs is not statistically significant, and so, it cannot be said that Simulated Binary Crossover is definitively better than the other two options.

Next, the three selection approaches were tested, having all other operators and hyperparameters as before, except for crossover, which was now Simulated Binary Crossover. As expected, rank selection did not behave well, having a high standard deviation. Since, with the first fitness function applied, fitness is essentially equal to the score, it is expected that in the first generations several individuals have a score of 0, 1 or potentially 2, meaning the same fitness. So to rank them and give, by chance, one individual a much higher probability of being selected in comparison to another one with the same fitness does not seem like the best approach. Fitness proportionate selection also showed bad results, between generation 150 and 350 there seemed to have been almost no improvement. So far, tournament selection seemed like the way to go.

Lastly, continuing with the same parameters as before, the three different mutation operators were tested. Geometric Mutation was tested with a mutation step of 0.25 and Gaussian Mutation was tested with a standard deviation of 0.25. Both Random Mutation and Gaussian Mutation showed high standard deviations, in contrast, Geometric Mutation, as we've covered before, performed consistently across all five runs.

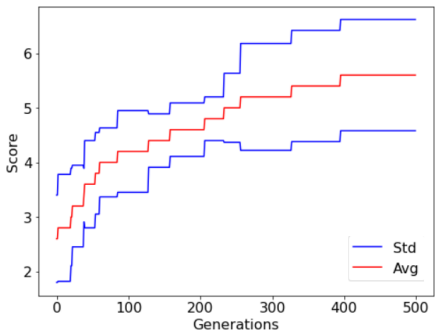


Fig. 1: Algorithm performance after optimization of selection approach and genetic operators [§]

[§]On figures 1, 2, 5, 6 and 7 we are only presenting the score as the fitness used was $F(x) = score + 1$ and so presenting it also would be redundant.

After, continuing with Tournament selection, Simulated Binary Crossover, Geometric Mutation and elitism with one elite, several crossover and mutation rates and mutation steps were tested. Crossover rates were always kept high, between 0.75 and 0.95 and mutation rates were kept low, between 0.05 and 0.25. Mutation steps varied between about 0.05 and 0.7.

The best results were achieved with a crossover rate of 0.85, a mutation rate of 0.2 and a mutation step of 0.3 and can be seen in figure 2. A slightly higher average score was achieved, and the algorithm converged much faster. It also showed consistency across all runs, having a small standard deviation, although increasing as the generations went up. In the last 300 generations there was no significant improvement, the algorithm seemed to be experiencing premature convergence, confirmed by the phenotypic and genotypic entropy across all runs, which had an average across all generation of -1.1 and -6.6 respectively.

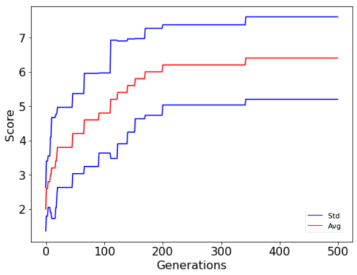
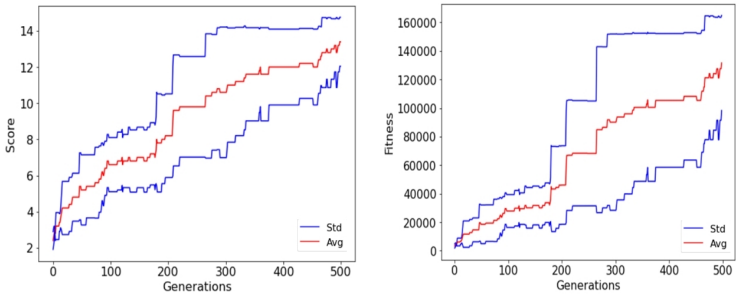


Fig. 2: Algorithm performance after hyperparameter optimization

To solve the problem of premature convergence, we decided to apply fitness sharing and immediately after, we decided to stop using elitism, to ensure that the population is improving as a whole, not being influenced by an individual that happened to have a high fitness in the first generations. Both of these changes combined seemed to have an immediate impact on the algorithm, which showed signs of a reduced standard deviation, a more consistent evolution across generations and a higher average score of the best individual. The phenotypic and genotypic entropy, however, remained about the same as before.

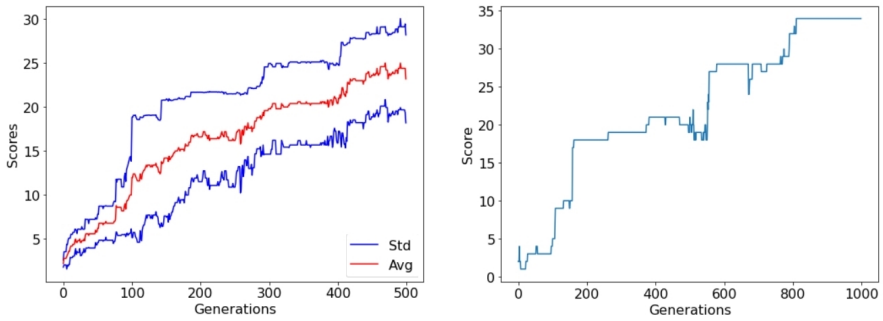
At this point, we tried different fitness functions, to see what effect that would have on the algorithm, and all genetic operators and hyperparameters were re-tested and changed, as the *greedy approach* to determine the current best parameters, did not guarantee that they were undoubtedly the best, and a new fitness function would surely require some change in parameters. Firstly, the fitness function was changed to $F(x) = 2^{(score+1)}$, to reward exponentially individuals that acquired higher scores. Then it became $F(x) = score + CorrectSteps$ (even though we were aware it could influence negatively the algorithm by being too influential in what action the individual should take), where $CorrectSteps$ are the number of moves that an individual takes that (using the Manhattan Distance) bring it closer to the nearest apple. This was made to reward individuals that take the shortest path to an apple, even though they might die before they get there, especially in earlier generations. Both of these fitness functions didn't improve the algorithm, in fact, they had the opposite effect, particularly the second one, where in the early generations individuals learned that if they spun around in circles, until they were "killed" due to the 100 moves rule, they would get a much higher fitness than they could ever hope to get if they actually tried playing the game. Finally, the second fitness function presented in section 2.1 was tested. The best average scores and fitness were achieved with fitness sharing, no elitism, tournament selection, arithmetic crossover, geometric mutation, crossover and mutation rates of 0.85 and 0.2 respectively and mutation step of 0.3. Higher average scores were achieved, however, at the cost of an increase in standard deviation in the latter generations. Even though fitness sharing was also applied, phenotypic and genotypic entropy remained about equal to the previously given values of -1.1 and -6.6.



Figures 3 and 4: Average score and fitness during evolution, using the second implemented fitness function

This tradeoff between higher average scores and higher standard deviation, was not considered favourly and so, the first fitness function was again used. For the same reason highlighted before, several configurations of parameters were once again tested, particularly with arithmetic crossover and with elitism once again present and testing if increasing the number of elites, to for example 5 or 10, would improve the model. Small further adjustments to the crossover and mutation rates and mutation step were also made.

The best configuration found was with fitness sharing, elitism with one elite, tournament selection with a tournament of 20 individuals, arithmetic crossover with a rate of 0.9 and geometric mutation with a rate of 0.2 and mutation step of 0.25. The standard deviation increased a little, but much higher average scores were reached. This final configuration was then ran with 250 individuals and 1000 generations. The best individual in the last generation reached a new high score of 35, however, this score was obtained at about generation 800, meaning in the last 200 generations there was no apparent improvement, and the algorithm might have reached a *plateau*.



Figs. 5 and 6: Average score and score reached with 1000 generations of the best found configuration for the genetic algorihm

The same configuration was also ran again 5 times, with 100 individuals and 500 generations, but now with a random generation of the positions of the apples, to see if the results achieved with a deterministic approach would transfer to a closer implementation of the classic Snake Game. Unlike in the beginning of the project, some evolution could be seen in the population. As expected, there is a higher variance, but in some runs, high scores comparable to the best configuration achieved with fixed positions of the apples were achieved.

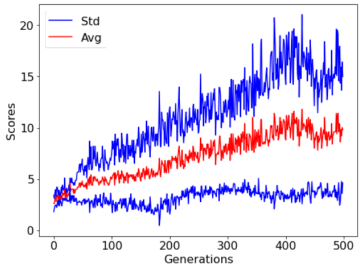


Fig. 7: Average scores achieved with random initialization of the apples' positions

4. CONCLUSIONS

Overall, this study shows that the usage of Genetic Algorithms, combined with Neural Networks, is a powerful step towards the path of development of intelligent machines. Moreover, it shows that it is possible to optimize the parameters of an algorithm in a deterministic and controlled implementation, and then transfer those parameters into a random and more realistic scenario and reach good results.

In the future, more steps could be taken to prevent and combat premature convergence. Although fitness sharing improved the results gathered, it was not enough to prevent a lack of diversity in the population.