NOVA IMS Information Management School

Computational Intelligence for Optimization: Group Project Report

Learning to Play Snake

Group: BOTS
Tiago Gonçalves | m20201053
Serdar Cetiner | m20201016
Oguz Kokes | m20201006
Berfin Sakallioglu | m20200545

1. INTRODUCTION

An optimization problem is a computational problem in which the objective is to find the best of all possible solutions. The main objective of this project is to solve an optimization problem using Genetic Algorithms (GA). In this project, using these search-based optimization algorithms, we taught a Neural Network model to play the snake game. Genetic Algorithms were used to optimize the set of weights for the Neural Network across generations. In this report we present the idea behind our code, as well as explaining the decisions that were taken and analysing the results obtained.

2. IMPLEMENTATION

Our project consisted of teaching a Neural Network to play Snake, using genetic algorithms to optimize its weights. It was implemented in *Python* and the *Keras* library was used to facilitate the implementation and modelling of the Neural Networks, providing a simple but flexible way of operating with their weights. Besides this, an implementation of the Snake game was necessary, which we took and adapted from the AILearner's "Training Snake Game Using Genetic Algorithm" project [1]. The implementation of the genetic algorithms and the respective operators was mainly based on the code developed during the classes, with the necessary adaptations to our problem.

2.2. IMPLEMENTING THE GAME

2.1.1. SNAKE GAME

The only change in code to the referenced game implementation was a reshaping of the snake's playground from a 50x50 grid to a 20x20 one, which we considered more adequate to the problem in hands.

2.1.2. NEURAL NETWORKS AND RUNNING THE GAME

In order to run the game using a Neural Network, the code that was already developed for these purposes – *run_game_with_ML* function – was adapted to work with a *Keras* model. The architecture that we set for our Neural Network was also based on the mentioned code, with 7 inputs, two hidden layers with 9 and 15 neurons, and an output layer with a *softmax* activation function, that outputs the 3 possible directions that the snake can follow – right, left, or same direction. Moreover, the *run_game_with_ML* function not only runs the game with the Neural Network, but also returns the fitness value achieved by the snake while playing the game. As an addition, we set it to return the score of the snake for later analyses. Initially, the function had a limit of 2000 moves per game, to avoid infinite loops from halting the evolution process.

Since *Keras* was used to implement the Neural Networks, some specificities had to be taken into consideration. *Keras* models necessarily have biases values so we implemented model initializations that always set these to zero. Moreover, a function was developed to automatically create the Neural Network based on two arguments: model architecture and an array of weight matrixes. As an example, a model with an architecture of the type 7-5-3 would then have an array with 7x5 and 5x3 matrixes for its weights. These arrays of weight matrixes were the core of our individuals. This function was useful to automatize the initialization process of individuals based on randomized weight values. A function to retrieve the weights of the *Keras* models without the biases was also developed.

1 https://github.com/TheAILearner/Training-Snake-Game-With-Genetic-Algorithm

[1] https://github.com/TheAILearner/Training-Snake-Game-With-Genetic-Algorithm

2.1.3. FITNESS FUNCTION

Our project clearly represented a maximization problem, as our main goal was for the snakes to achieve higher scores while evolving. The fitness function that we set initially, to assess if everything was working correctly with our implementation, was inspired on the original function from the AILearner project, and is given by the following expression:

$$fitness_1 = score \times 5000 - 150 \times death - penalty_1 + 2 \times award_1$$

On this function, $penalty_1$ refers to the number of times that the snake moves in the same direction for more than 8 consecutive movements, and $award_1$ is the number of times the snake changed direction. This fitness function awarded 2 points each time a snake changed direction and penalized the snakes in case they crashed (-150 points) or in the case of more than 8 consecutive moves in the same direction (-1 point). A significant reward of 5000 points was given to snakes each time they ate an apple. After several initial tests, we noticed that most of the snakes were staying in infinite rotation loops generation after generation. We concluded that this fitness function was not appropriate, since it was rewarding these snakes instead of the most “courageous” ones that were looking for food but eventually crashed. Trying to fix these problems and enhance the results of the evolution process, we developed a new version of this fitness function:

$$fitness_2 = score \times 5000 - 150 \times death - penalty_1 - 1000 \times penalty_2 + 2 \times award_1$$

The idea behind the new function is the same as before, but $penalty_2$ was introduced: if a snake goes for more than 300 moves without eating any apple, it is penalized with -1000 points and the game instantly ends. This new fitness function highly penalized snakes that stayed in loops without eating an apple, particularly at early stages of the evolution process. The introduction of this new penalty boosted the evolution process and allowed the snakes to start evolving and performing better along the generations. Moreover, it allowed us to increase the limit of 2000 moves per game to 10000, since the game would automatically end if the snakes were not eating apples, accelerating the evolution process. This was the fitness function that we used throughout all our tests, and the results are presented on the final section.

2.2. GENETIC ALGORITHMS

On this sub-section we summarize the code that we have developed to implement the genetic algorithms and the evolution process. As referred previously, it was mainly based on the code developed in the practical classes during the semester – with the necessary changes to fit it to our problem. Each individual consisted in an array of weight matrixes, as already mentioned.

2.2.1. SELECTION

The three selection operators we used were the ones developed during the classes – Fitness Proportionate Selection, Tournament Selection and Rank Selection. No changes were made to the Tournament and Rank Selection operators, as they were already compatible with maximization and minimization, and flexible for every kind of problem. However, some adaptations needed to be added to the FPS operator. As our problem could involve negative fitness values, an adjustment part was implemented for this operator to be compatible with these. This adjustment consisted in the following: In case there are negative fitness values in the population, the minimum one is picked. Then its absolute value is added to every individual’s fitness, and Fitness Proportionate Selection can be normally performed afterwards. Naturally, it stops being exactly **fitness proportionate**, but this was the way we adapted it to work with negative fitness values (which after some generations of the evolution process start being residual, in our particular case). This operator is still only compatible with maximization problems.

2.2.2. CROSSOVER

Given the specificities of our individuals, three crossover operators were implemented from scratch, namely *weights_swap_co*, *arithmetic_co* and *blend_co*. Each of them takes *parent1* and *parent2* as arguments, and is summarized below:

***weights_swap_co* –** Specific argument: *max_swaps*. Randomly generates a number (*n_swaps)* between 1 and *max_swaps*. Then swaps randomly chosen weights (with the same index on each of the parents) from *parent1* and *parent2 n_swaps* times.
***arithmetic_co* –** Specific argument: *max_points*. Randomly generates a number (*n_co)* between 1 and *max_points*. Then performs arithmetic crossover between randomly chosen weights from *parent1* and *parent2 n_co* times, given a randomly generated *alpha*.
***blend_co* –** Specific arguments: *max_points* and *alpha*. Randomly generates a number (*n_co)* between 1 and *max_points.* Randomly generates a value *gamma* that depends on *alpha.* The functioning is the same as *arithmetic_co,* but it can expand the possible values of the weights beyond the $[-1,1]$ interval.

2.2.3. MUTATION

The three mutation operators were also implemented from scratch: *swap_mutation*, *inversion_mutation* and *box_mutation*. Each of them takes *offspring* as an argument:

***swap_mutation* –** Specific argument: *max_swaps*. Randomly generates a number (*n_swaps*) between 1 and *max_swaps*. Swaps randomly selected weights on the individual *n_swaps* times.
***inversion_mutation* –** No specific arguments. Randomly chooses a weight matrix and an array from it. Randomly selects 2 indexes of the selected array and inverts the sequence of weights between those 2 indexes. The inversion is performed only once.
***Box_mutation* –** Specific arguments: *max_changes* and *mutation_step.* Randomly generates a number (*n_changes*) between 1 and *max_changes*. Performs the following operation *n_changes* times: generates a value between -*mutation_step* and *mutation_step* from a uniform distribution and adds the generated value to a randomly chosen weight.

2.2.4. *CHARLES* LIBRARY

The *Charles* library was adapted to our problem as well. Our *Individual* class took 2 inputs to be initialized – architecture and weights matrixes. These matrixes were randomly initialized in case *weights* = None. The evaluate method was linked to the *run_game_with_ML2* function – from which the fitness and the score of an individual was saved. For the *Population* class, the *indiv_units* was added for it to be initialized – which corresponds to the architecture of the individuals. Moreover, the *evolve* method is the one developed during the classes – the only changes performed were a check to ensure that the crossover always happened between different parents and saving the best performing model of the evolution process in the end. Besides this, we developed an additional *probabilistic_evolve* method. This method gives additional randomness to the evolution process – at each selection, crossover and mutation operation, one of the 3 operators is randomly chosen (by generating a number between 1 and 3, that is assigned to a specific operator). Developing this method, our goal was to increase diversity on the populations during the respective evolution processes, aiming to increase the probability of getting smarter individuals. The results of the runs with different configurations and with the *probabilistic_evolve* method are presented on the next section.

3. RESULT ANALYSIS & DISCUSSION

Our initial approach for the evaluation of the algorithm was to create a large sample size for all possible configurations. However, as the algorithm has three options for each operator types, coupled with the elitism option, the resulting test space had 54 unique configurations. Due to technical limitations on our computational power, creating a statistically significant sample size was not possible.

To accommodate a still significant sample for analyses 9 configurations were selected and run 10 times each with populations of 60 individuals and 100 generations. Due to our findings in development and from intuition all runs except for the *probabilistic_evolve* utilised elitism. *probabilistic_evolve* was run an additional 10 times with and without elitism to accommodate the probabilistic nature of the method. In total the algorithm was run 110 times for different configurations for the testing. For the evaluation, the maximum score achieved in each run for each configuration were taken and averaged to avoid misinterpretation due to outliers.

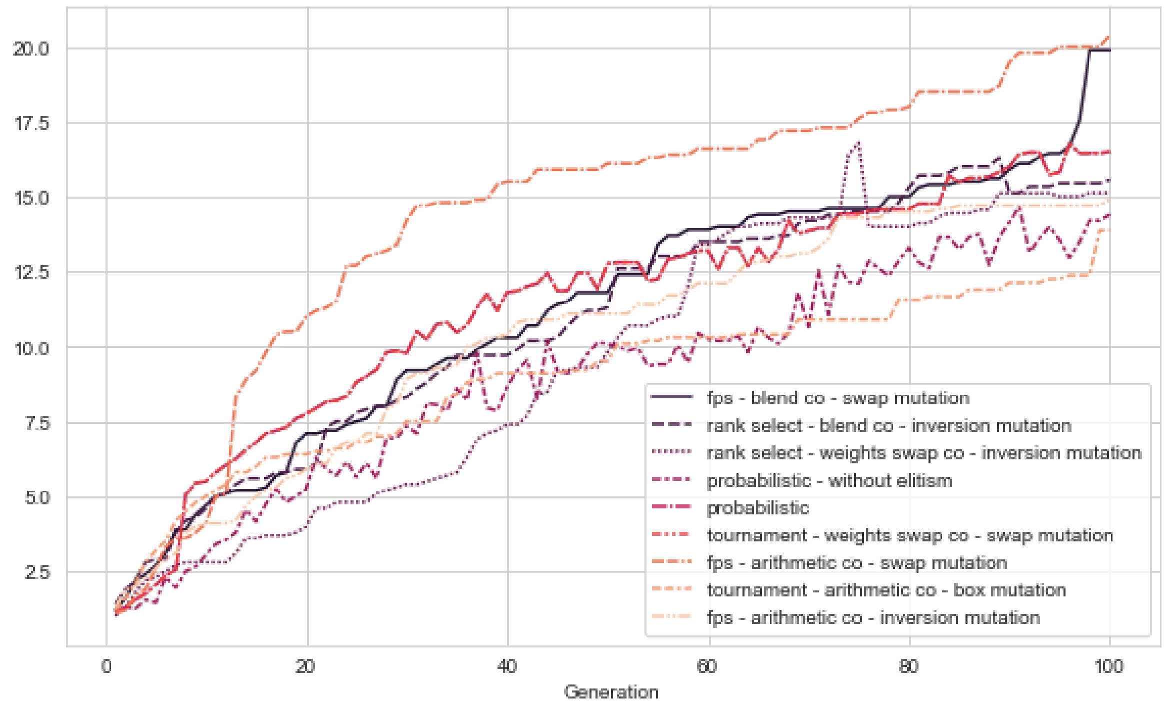


Figure 1. Average max-of-run snake score across generations

As it is visible on Fig. 1, on average, across all generations the configuration with Fitness Proportionate Selection, Arithmetic Crossover and Swap Mutation performed the best, in a consistent way. Regarding the *probabilistic_evolve* method, the 20 runs with and without elitism allowed us to assess its impact on the evolution process. Besides guaranteeing a more stable evolution of the maximum score per generation (and consequently the fitness values), including elitism also resulted in higher average maximum scores per generation as can be assessed more clearly on Fig. 4.

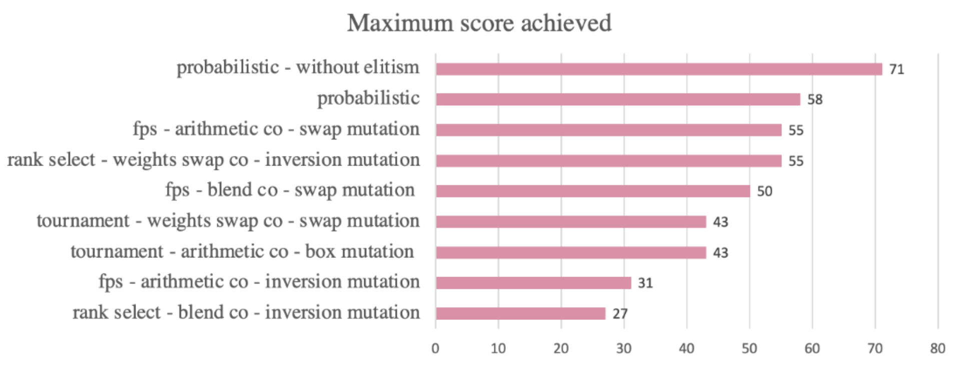
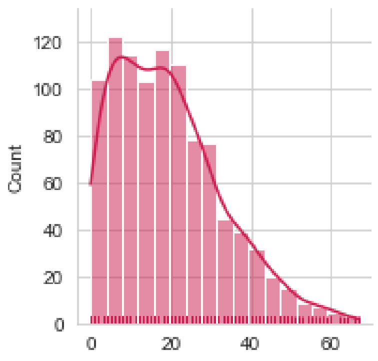


Figure 2. Maximum score achieved by configuration

On the maximum score achieved, *probabilistic_evolce without elitism* achieved a score of 71 corresponding to a fitness of 362,984 on generation 94. As our goal was to get the highest score for the snake, the corresponding Neural Network was run with 1000 games.



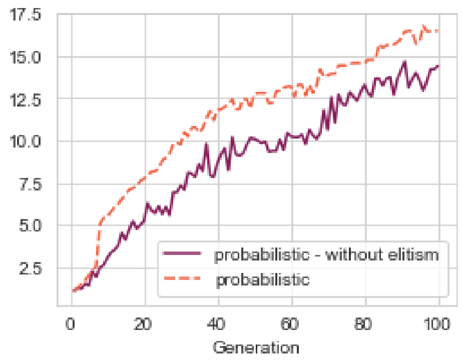


Figure 3. Score distribution for the top performing snake

Figure 4. Avg max-of-run snake score by elitism

In these 1000 runs, 67 was the highest score achieved while having an average score of 19.3 and standard deviation of 13.5. *probabilistic_evolve* by its nature is volatile in terms of scores achieved. When we look at the average of max-of-run scores, *probabilistic_evolve* with elitism scored higher in average in almost all generations but it was not able to achieve the highest score.

4. CONCLUSION

Through implementation of genetic algorithms, we were able to create snakes that can confidently play the game on their own. In the development of this particular project, we found that the most important aspect was the fitness function. Its improvement not only affected the quality of the snakes but also lowered the amount of time required to test.

In all configurations, our snakes have gotten more competent over generations promising further growth given enough generations. However, as these tests were computationally expensive, robustly testing the effects of different selection, crossover and mutation operators proved to be difficult. Improvement of this project lies on the efficiency of the algorithm as reduced testing times will result in larger samples to test.

Even with these limitations, we were able to pass the benchmark of 40 we set for ourselves and achieve a highest score of 71. As such we can consider the outcome of this project successful. [2]

[2] The “smartest” snake can be seen here.