NOVA IMS Information Management School

LEARNING TO PLAY SNAKE

Computational Intelligence for Optimization

https://github.com/glopes00/Snake_Group_G

Gonçalo Lopes - m20210679
Tiago Oom Sousa - r20181077

1. Introduction

The aim of this project was to teach a model to play Snake. As we know, in order to achieve a good score, we need to make sure that the snake eats as many apples as possible, while still being able to avoid colliding with the wall and with itself.

There are several concepts that need to be implemented for that to happen, such as neural networks, an adequate fitness function, and selection, crossover, and mutation operators. In this report, we will address previous approaches to this problem, the implementation of proper genetic algorithms, and the performance measurement that was made, in order to evaluate our model.

2. Problem Representation

Our project code was based on an Github repository called "Training-Snake-Game-Using-Genetic-Algorithm", where the objective is to create a program that plays the snake game with the help of neural networks and genetic algorithms. To better understand the way this project was divided, we will briefly explain the main topics of the repository.

2.1 Neural Network

The neural network used in this project had the following architecture:

- Input layer (with 7 neurons)
- Hidden layer 1 (with 9 neurons)
- Hidden layer 2 (with 15 neurons)
- Output layer (with 3 neurons)

2.2 Input Variables

As mentioned above, there are 7 different inputs of the neural network. These inputs are shown below, and they take into consideration the position of the snake, the position of the apple, and the board´s limits:

1- Is left blocked (0 or 1) – checks if snake collides with limit of the board or with itself
2- Is front blocked (0 or 1)
3- Is right blocked (0 or 1)
4- Direction to the apple (x-axis)
5- Direction to the apple (y-axis)
6- Snake's current direction (x-axis)
7- Snake's current direction (y-axis)

2.3 Output Variables

Moreover, we also stated above that there are 3 possible outputs of the neural network. These are obtained using a Softmax function and can be one of the following 3: turn left, keep going in the same direction, or turn right (values -1, 0, and 1 respectively).

2.4 Fitness Function

In order to keep track if the snakes were progressing as we evolve into next generation, a fitness function was created. This function uses a combination of penalties and awards (related to the snake behavior), and the value obtained will be used to choose the best snakes of each generation in order to passed them to the following generation (elitism). Later in this report, we will discuss how we improved this fitness function by adding new rewards and penalties.

2.5 Game Layout

All the layout of the game and the display of the snake was also imported from the Github repository, alongside the functions used for the snake game to run smoothly. This way, we can actually see the snakes moving throughout the board and better understand their behavior.

3. Approach with Genetic Algorithms

For our implementation of the project, we transformed the representation of the population from a matrix (n individuals with m weights each) to a list by adding classes into our code (class Population and class Snake). With this alteration, we can now represent the population as a list of elements (each element being an object of the class Snake). Consequently, we were able to use the selection, crossover, and mutation operators developed during the practical classes, while creating new ones to better fit the problem at hand.

Because the Genetic Algorithm implementation is the main core of our project, we will explain in more detail the main concepts that we undertook and the challenges we faced throughout this execution.

3.1 Generation of the Initial Population

To start with, we generated the initial population by creating the above-mentioned list of elements of class Snake. Each snake has an attribute called representation that refers to the weights that will be given as the input for the Neural Networks. Here, we initialized all this weights randomly with values between -1 and 1.

We tried populations with 50, 75, and 100 snakes, in order to access to what extent the size of the initial population would influence the final results.

3.2 New Fitness Function

For the implementation of the fitness function, we considered numerous factors that could make the snake perform better and avoid errors.

This fitness function takes into account not only the obvious measurement parameters, such as *max_score* (to measure how many apples the snake eats per game) and *steps_per_game* (that determines the steps that each snake can make per game); but also several penalties and awards that were tested in numerous different combinations, along with different constant values, as we can see below:

- ***steps_without_eating*** and ***penaltyHunger*** – counts the number of steps a snake makes without eating an apple. Whenever the first variable reached 200 steps, the variable *penaltyHunger* was incremented by 1 value. Consequently, *steps_without_eating* is nullified. *PenaltyHunger* was implemented in order to avoid the snake to make a game without eating.
- ***still_front***, ***still_left*** and ***still_right*** – increments whenever the snake's move is going forward, left or right, respectively. These variables were programmed to help implement the variables below.
- ***penaltyLoop*** – penalizes a snake if it completes more than 10 loops on itself. In other words, whenever the variable *still_left* or *still_right* had a value greater than 40. These last two variables were then set to zero. *PenaltyLoop* aims to prevent the snake to create a loop in itself, a situation that happened a lot in our initial snakes.
- ***awardFront*** – awards the snake to make a move forward, incrementing this variable by 1 each time *still_front* was greater than 3. This award was created to encourage the snake to avoid getting into a loop.
- ***death*** – if the snake hits the board boundaries or eats itself, death was incremented by 1 value and the game breaks.

- ***avg_steps*** – penalizes the snake for the average steps that it takes between apples. If the score of the snake (*max_score*) was null, the *avg_steps* receives a negative value (300).

After numerous iterations studying the effect of the variables on the snake's performance and fitness and modifying the constants of the penalties and awards to a value that was optimal, we came to the conclusion that the fitness function should be implemented as shown below.

```
fitness = max_score * 4000 - death * 100 - penaltyLoop * 5 - penaltyHunger * 100 + awardFront * 2
```

Figure 1 - Fitness Function

All in all, we ended up penalizing the snake whenever it dies, enters a loop or does not eat for more than 200 steps. On the other hand, we reward the snake when it eats an apple and when it makes more than 3 steps on the same direction (with a small constant).

3.3 Selection Operators

Having all our initial snakes created and their respective fitness calculated, we can now proceed to the selection algorithms. Here, we implemented the tournament selection developed during class, as well as a new selection operator that we designed. This new operator, called "tournament_new", tries to increase the chance of choosing only the best snakes from the previous generation to be used as parents for the following generations. But how many snakes are considered the best ones? After trying to tune the best threshold to implement, we decided that 25% would be a consistent choice. This means that, if we are using a population size of 50 snakes, we will extract snakes for the tournament from a list containing the best 10 snakes from the previous generation.

Moreover, the fitness proportionate selection operator was also taken into consideration for our project. However, because a lot of the fitness are negative, this implementation fails a lot of the times. We tried fixing this by adding the minimum value of all the fitness (this way eliminating negative values), but the results were not as satisfactory as a tournament implementation. Because of this, we focused more on the tournament selections.

3.4 Crossover Operators

After having our parents chosen, it is time to perform the crossover operators. This operator will have an 90% probability of occurring (crossover_prob = 0.9). In this step, we have a lot of different operators that we can use and that were developed during the practical classes, such as: single point crossover, cycle crossover, partially matched/mapped crossover, and arithmetic crossover.

Nevertheless, the same way as we did in the selection operators, we decided to create a new crossover customized specifically for our project to try to improve our results. The new operator, named "crossover_mix", consists in a combination between arithmetic and uniform crossovers. Basically, one of the offspring is calculated with arithmetic crossover, and the other is calculated with uniform crossover (where each weight is selected randomly from one of the corresponding weights of the parents by using a toss of a coin). This new operator was created to try to find the best set of weights of the parents (uniform part), while still generating new weights to retain diversity (arithmetic part).

3.5 Mutation Operators

As for mutation operators, we implemented two methods already addressed during classes: inversion mutation and swap mutation. We did not focus so much on this part since the mutation probability is 10%, meaning that there is a very small chance that the snake suffers mutation. We decided to stick with this probability value since we believe that crossover operations are much more effective for the reproduction of the snakes than mutation.

3.6 Elitism

Finally, we looked into the possibility of applying elitism. We understand that this parameter can substantially increase the fitness of our snakes since we are passing the best snakes directly to the next generation, this way never losing information about them. This being said, we implemented in our code the option of applying elitism or not, as well as the number of snakes we wanted to copy directly to the new generations (parameter named n_elit).

4. Performance Measurement

Moving on to performance measurement, we wanted to quantify how well our model performed with different selection, crossover and mutation operator. Besides, we also wanted to determine whether the presence of elitism was beneficial to our model.

When moving on to the analysis of the performance, we started testing the model with 100 generations and with a population size of 50 individuals per generation. Later on, we conclude that the snakes were evolving well through the first 50 to 60 generations but, after that, the fitness and score values of the best snakes would not change, and almost every time ended up getting worse throughout the last generations. To bypass this, we decided to increase the population size to 100, and run the model through only 50 generations. We ended up increasing the fitness and score values.

As mentioned above, we tried different selection and crossover methods, as well as the presence of elitism. Because our program takes quite some time to perform, we decided to run only 8 different combinations of operators and running each one of them 7 different times. The average values of the fitness obtain, as well as the average of the maximum score were computed and are shown in the table below.

Selection	Crossover	Mutation	Elitism	Average Fitness	Average Score
tournament	arithmetic_co	swap_mutation	FALSE	36 182	9
			TRUE (n_elit=5)	51 410,40	12,8
	crossover_mix		FALSE	54 620,50	13,75
			TRUE (n_elit=5)	83 222	20,6
tournament_new	arithmetic_co		FALSE	18 506,8	4,6
			TRUE (n_elit=5)	28 911,2	7,2
	crossover_mix		FALSE	52 475,90	13,2
			TRUE (n_elit=5)	58 960	14,7

Table 1 - Performance Measurement

Taking into consideration the results obtained and the table above, we can take some conclusions:

- The performance of the model is always better whenever we are in the presence of elitism.
- The combination tournament and crossover_mix was the combination that led to better results in terms of both fitness and maximum score.
- The problems that we were facing regarding the stagnation of the snakes throughout the last generations was properly fixed when we changed to *population_size*=100 and *number_of_generations*=50 since the best snake was found (most of the times) in one of the last generations. Furthermore, improvements of the snakes were made progressively as the number of generations increased.

5. Conclusions

To conclude our report, we ended up with a satisfactory fitness function that led to reasonable performance results. However, theses results may not be illustrative since it exists a big random factor present in the snake game: the apple position is defined randomly every time it appears. Because of this, even snakes with the same values of weights will behave in a completely different way. This was definitely a big difficulty we faced while trying to design the optimal fitness function.

Moreover, the time spent to run our program was also one of our main challenges. Because it took a lot of computational power, it was difficult to test every desired combination properly. This led to only running each combination 7 times, which may lead to not very representative results.

Nevertheless, we managed to create snakes that were increasing its performance throughout the generations and that were learning with previous snakes to overcome their errors and eat more apples along the way, that was the objective of this project.

6. References

Atul, Kang & (2022). Training-Snake-Game-Using-Genetic-Algorithm. [online] GitHub. Available at: https://github.com/TheAILearner/Training-Snake-Game-With-Genetic-Algorithm

Gutgutia, Y. (2021). ygutgutia/Snake-Game-Genetic-Algorithm. [online] GitHub. Available at: https://github.com/ygutgutia/Snake-Game-Genetic-Algorithm

Kang & Atul (2018). Snake Game with Genetic Algorithm. [online] TheAILearner. Available at: https://theailearner.com/2018/11/09/snake-game-with-genetic-algorithm/

Salazar, R.A. (2022). Top Genetic Algorithm Github repositories. [online] GitHub. Available at: https://github.com/rafa2000/Top-Genetic-Algorithm