Using Genetic Algorithm to Learn and Play the Snake Game

Mayav Rohan Antani
Electircal and Computer Engineering
University of Waterloo
Student ID: 21082382

***Abstract*—This project explores the application of genetic algorithms (GAs) in training artificial intelligence (AI) agents to effectively play the classic Snake game. The study leverages the principles of evolution, including selection, crossover, and mutation, to iteratively optimize the performance of AI agents. A feedforward neural network, integrated with GA, is employed as the decision-making system for the Snake game. The network processes game state inputs to predict optimal moves while GAs refine the network's weights over multiple generations. The effectiveness of this approach is demonstrated through simulations, where agents progressively adapt to the game environment, learning strategies that maximize survival and scoring efficiency. This research highlights the potential of combining evolutionary computation with neural networks for solving complex control problems.**

I. Introduction

The automation of complex tasks through computer systems is a fundamental goal of artificial intelligence. Mimicking human behavior often requires insights from evolutionary biology, where organisms improve and adapt to their environment over generations. By applying similar principles, this study employs genetic algorithms (GAs) and neural networks to train models capable of learning and achieving objectives autonomously.

To demonstrate these concepts, the Snake game is selected as a test environment. The game involves a player controlling a snake to maximize its score by eating food while avoiding collisions with the game boundaries and its own body. The challenge lies in navigating the growing snake in a constrained space, requiring strategic decision-making.

In this study, the player's role is replaced by an AI agent controlled by a feedforward neural network. The network is trained to act as the "brain" of the snake, making real-time decisions to ensure survival and score maximization. The GA optimizes the network weights, simulating the process of natural evolution to enable the agent to adapt and improve its performance over successive generations.

This approach involves multiple computational and biological concepts, including game development, artificial neural networks, natural selection, and evolutionary strategies. Through iterative learning, the AI agent evolves to play the Snake game more effectively, demonstrating the synergy between evolutionary computation and machine learning

II. Literature Review

[1] The Pygame library served as the backbone for game development in this project. Its comprehensive documentation provided detailed instructions on using modules for rendering graphics, handling events, and managing game loops. Features like collision detection were critical for implementing the Snake game's logic, including smooth movement, boundary constraints, and object interactions. Additionally, the ability to control the frame rate and manage game events allowed the integration of AI-driven gameplay seamlessly.

[2] This article presents a broad overview of evolutionary algorithms (EAs), including genetic algorithms, and their applications in optimizing complex systems. It emphasizes the adaptability of EAs to solve problems where explicit mathematical models are unavailable.

[3] This chapter discusses the implementation and theoretical underpinnings of metaheuristic algorithms, particularly genetic algorithms. It highlights practical considerations such as balancing exploration (diversity) and exploitation (convergence) during the evolutionary process. These principles directly influenced the choice of selection methods (best selection vs. roulette selection) and the mutation strategy in the project.

[4] This comprehensive review traces the history and advancements of genetic algorithms, covering their theoretical foundations and diverse applications. The discussion on mutation techniques and selection mechanisms was instrumental in shaping the project's genetic algorithm design. It underscored the importance of maintaining genetic diversity through mutation, a principle applied in the project to prevent convergence to suboptimal solutions.

[5] This paper examines the application of genetic algorithms in game-playing scenarios, focusing on how GAs can simulate learning and adaptation over time. The insights from this work validated the feasibility of using GAs to train AI agents for the Snake game. The concept of evolving agents through iterative improvements, guided by a fitness function, was particularly relevant.

[6] This tutorial served as a practical guide for building the basic Snake game using Pygame. It provided foundational code examples for implementing core game mechanics, such as snake movement, food placement, and collision detection.

The modularity of the examples allowed for easy customization, which was essential for integrating the neural network and genetic algorithm components.

III. Methodology

The methodology section provides a comprehensive explanation of how the Snake game, neural network, and genetic algorithm were developed and integrated to train an AI agent. This process involves modular design and deliberate choices of tools and techniques, each serving a critical purpose in achieving the project's objectives.

A. Developing the Game

First step to implementing genetic algorithm to the snake game would be to make a playable game that resembles the snake game in terms of the objectives, constraints and overall concept [6]. For this to be achieved, an understanding of python pygame, vector arithmetic, and object oriented programming is necessary. Key game components include:

- **Snake and Food Representation**: Implemented using vector arithmetic to manage movement and positioning.
- **Collision Detection**: Ensures the snake detects boundaries and self-collisions, fundamental for game rules.

The python libraries used in completing the game were:

- pygame [1] - allows people to develop games and provides functionality to make the process easier.
- random - makes it possible to implement randomization upto certain extent.
- numpy - helps in complex numeric calculations involving arrays.
- math - simplifies implementing mathematical concepts to the code.

This is where all the logic of the game is stored for execution like drawing the snake and food, moving the snake, collision detection with various things, calculating distance and angle between the snake and the food, and checking if a direction is blocked for the snake. All these functions are preliminary for the game to work properly and learn in the future. This is also where all the constants that will decide what the game would look like and how fast it would be played is stored. The figure 1 shows a snap shot of the game window mid-game.

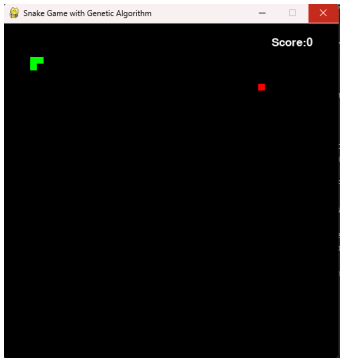

Fig. 1. Game Window

The game logic integrates with the neural network and genetic algorithm, enabling AI-driven gameplay while adhering to the game rules. This modular design facilitates future expansions, such as changing the reward structure or incorporating new constraints.

B. Feed Forward Neural Network for Decision Making

In order for the agent to make decisions on its own, a neural network is built with 2 hidden layers. The input layer has 7 nodes, first hidden layer has 9 nodes, second hidden layer has 15 nodes and the output layer has 3 nodes. The goal of this neural network is to predict the next move to be made by the snake to survive and increase the score. To do this it is provided with the current state of the game as an input in terms of if the left or right side of the snake or the front of the snake is being blocked by either the boundary or itself, and the direction of the food with respect to the head of the snake, and finally the current direction in which the snake is traveling. These are the seven inputs for the neural network. Looking at the structure of the network, the number of trainable weights can be calculated. In this case the number of trainable weights are 243.

1) Structure of the Neural Network: The neural network is not very complex, all the layers are fully connected and both hidden layers make use of the $tanh$ activation function while the output layer uses the $softmax$ function to get the probability in order to guess the next step to be taken. Finally, the output is comprised of three nodes, thus the interpretation would be:

- $[1, 0, 0]$ will move snake left.
- $[0, 0, 1]$ will move the snake right.
- $[0, 1, 0]$ will keep moving the snake in the current direction.

C. File to run the game using the Neural Network

This file is used to play the game using the neural network. It takes the display screen, game clock and the neural network weights as the inputs to start the game and then gives the fitness value and in game scores for each generation.

This file has the $run\_game\_with\_ML$ function which takes the inputs and starts by storing the maximum scores, penalties, number of test games to be played by each chromosome in the generation and the number of steps allowed to be taken per game. These are all necessary to keep track of the score and to limit the length of the game since after a few generations the snake would, in theory, learn to avoid colliding with the boundaries and itself. As the function starts executing, each test game starts with the initial position of the snake and the count of the number of times the predicted direction is the same as the direction of the movement of the snake while keeping track of the direction the snake was traveling in previously.

Then the control enters the loop for each step in the game. Here different information about the snake is collected like the current direction of the snake, which directions are blocked with respect to the snake, the angle of the snake with the

food, normalized direction vector of food with respect to snake and the normalized direction vector of the snake. These are all the required inputs for the neural network to learn and predict the next step to be taken. After the prediction, the check of if the predicted direction is same as the direction in which the snake was already moving is done and the count is updated accordingly. Next, the function checks if the predicted direction would lead to a collision with either the boundaries or with itself, if there is a collision, a penalty is given to avoid such predictions in the future.

Finally, on playing the game for that one step, the updated position of the snake, food and score are returned and the max score is calculated. After this, the function checks if the number of times the predicted direction being the previous direction is more than 5 provided that the predicted direction is not going straight, if this is the case there should be a penalty given to the snake and so a variable is used to keep track of the same. On exhausting the number of steps in a game, the distance between the food and the snake is calculated and added to the penalty in order to encourage getting as close to the food as possible. In the end this function returns the fitness score of each chromosome in the generation along with the in game scores to keep track of and see how the learning process changes the scores in the future.

D. Genetic Algorithm for Optimization

This file takes car of the genetic algorithm [3]–[5] part of the game. This file has the functions for calculating the population fitness, selection of the parents to be used for mating and evolving, crossover between the selected parents to produce the off-springs and the mutation function for the off-spring.

1) Calculating Population Fitness: This function keeps track of the fitness values of the population and the scores obtain by them in that generation. It takes the population as input and then iterates over it to run the $run\_game\_with\_ML$ function in the other file to get the fitness values and the scores of the game, displays the fitness of each chromosome to the terminal and store them in new variables to return to the main file for visualization.

2) Selection Criteria: Two methods have been used to select the parents in a population to produce off-springs, choice by roulette and selecting the top k best chromosomes as the parents. The functions take the population, fitness values and number of parents to be selected as their input variables and then proceed to the selection process.

- **Choice By Roulette**: This is a common method where the relative fitness of the population is calculated and then the selection of the parents is done based on that relative fitness. This makes it so that each chromosome has a chance to be selected even if it does not have a very high fitness value. In other words, this method uses probabilities proportional to the fitness value of the chromosome. Here to calculate the relative fitness, firstly all the fitness values are made non-negative by adding to them the absolute value of the smallest or lowest fitness value in that generation. The relative fitness is then calculated for each chromosome in the generation by diving the non-negative fitness values by its sum. This will make the cumulative probabilities for the selection of the parents. Then based on the number of parents to be selected, random numbers are generated and the cumulative probabilities are used to select the parents using that random number.
- **Selecting Best**: This was a way to ensure the improvement of the chromosomes over generations as only the best chromosomes in each generation would be selected to produce off-springs. Here, the population, fitness values, and number of parents to be selected are taken as input and then the index of chromosome with the highest fitness value is stored and used to select it as a parent for the next generation while also reducing its fitness value for the time being to as low as possible so that it is not selected again in the selection stage.

Finally, both the functions return the parents selected from the population in that generation.

3) Crossover: The crossover splits up the parent chromosomes and recombines them. Crossover point can be chosen randomly to increase diversity of new population. The figure 2 shows multiple ways crossover can happen. Here the crossover

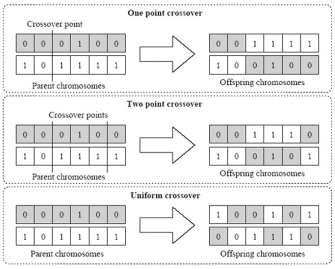

Fig. 2. Crossover

is done between two parents that are not identical to get variation in the next generation of off-springs. Uniform crossover is done each gene in the offspring is randomly selected from one of the two parents.

4) Mutation: Mutation is used to maintain genetic diversity from one generation of a population of chromosomes to the next. It is analogous to biological mutation. The purpose of mutation in GA is preserving and introducing diversity. Mutation should allow the algorithm to escape local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. Mutation can be done with a formula or randomly. Figure 3 shows a simple example of how mutation works. The mutation function here takes the off-spring as the input and makes use of random number generations to introduce mutations to the off-spring. This helps introduce much needed variation and diversity in the population via sudden change in the chromosome.

E. Main file

This file is the driver code of the entire program, this file is run in order to start execution of the code. All the

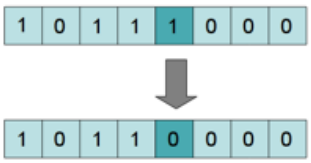

Fig. 3. Mutation

important variables required for the proper execution of the game with genetic algorithm are defined here like number of chromosomes per generation, the number of weights, number of generations, number of parents to be selected and more. It then starts a loop till the number of generations have been exhausted to keep on playing the game and learning the fitness values and the game scores. The calculation of the average fitness per generation, highest fitness in the generation and the highest scores in a generation are also done here. Finally, the visualization of the graphs of high score over generations and average fitness over generations are also plotted here. The basic flow of this file is shown below:

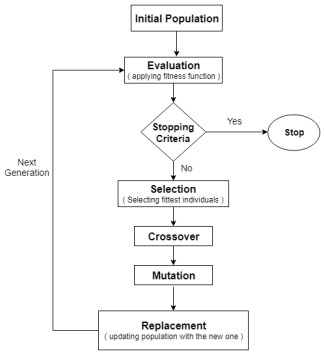

Fig. 4. Execution Flow

IV. Experimentation and Results

The experiments were performed for all the selection methods included for 20 generations with 50 chromosomes per generation, with 10 parents being selected per generation for crossover and the results were as follows:

On looking at the results in 5,7,8, and6 from the experimentation, it can be seen that selecting best chromosomes from a generation as parents has a better average fitness value and higher scores over generations.

Next, the best selection method was used for 100 generations with 50 chromosomes on each generation and 10 parents being selected for crossover and the results were as follows

On looking at 9 and 10 it can be noted that the snake starts off with a very low score and fitness value and over generations starts learning how to survive and get higher scores. This is to be expected as the neural network becomes better at predicting the next move as number of generations increases. The best generation in terms of the average fitness score was recorded to be generation 80 but we can see that the fitness score

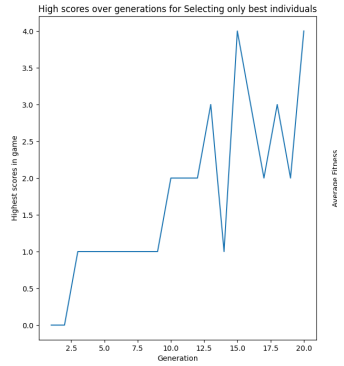

Fig. 5. High score over generation For best selection

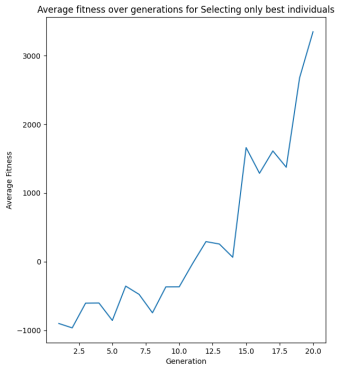

Fig. 6. Average Fitness over generation For best selection

did not continue on a downward trend after that, instead it stayed around the same level. This is because of the crossover, mutation and selection parameters that introduces variation to the learning of the model.

V. Future Scope

As there are many randomness in genetic algorithms, the experiment result could be biased. Further research can be done by controlling the arbitrary factors. Another way is to change the neural network and the weights to see how that would affect the training and improve on it. Other ways for further development would be to use some other selection methods and improving the predictions by using different kinds of penalties and rewards.

VI. Conclusion

When we look at and compare the results of the roulette selection and the best selection methods, it is clear that roulette lags behind in terms of both, high scores and average fitness values. This may be because there is a higher chance of worse parents to be selected in the roulette selection as compared to when selecting only the best chromosomes in the other method. This difference could lead to crossover between two parents that are not very good at surviving and could lead to producing sub-optimal off-springs that would in turn reduce the average fitness over generations.

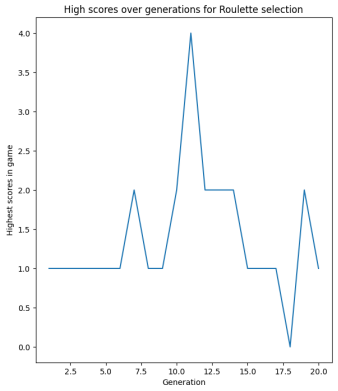

Fig. 7. High score over generations for Roulette Selection

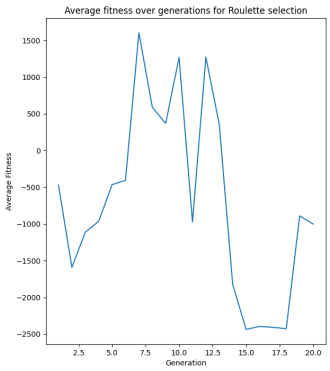

Fig. 8. Average Fitness over generations for Roulette Selection

Based on the results seen in the previous section, it can be seen that over time the snake learns to survive longer and score more thus increasing the fitness value. First, the snake just circles around in one place so as to increase the survival rate and to avoid crashing into walls and itself, then it starts learning to navigate through the window to reach the food. We can also see that for some generations the learning of the snake becomes slow. By the later half of the generations, the snake becomes much better at navigating, tracking and eating the food all while avoiding collisions. The main reason why the score for the game does not keep on increasing to infinite is because of the limit placed on the number of steps to be taken during each game. This limit also encourages the snake to reach the food as soon as possible and in as few steps as possible.

We can conclude that GA do improve the optimization overtime. However, time complexity cannot be measured exactly due to the high arbitrary aspect.

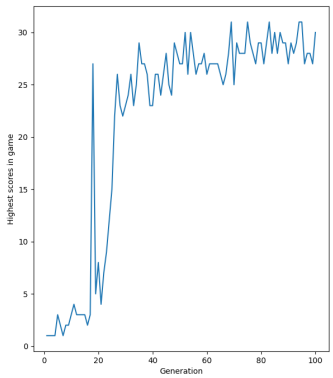

Fig. 9. High Score over Generations

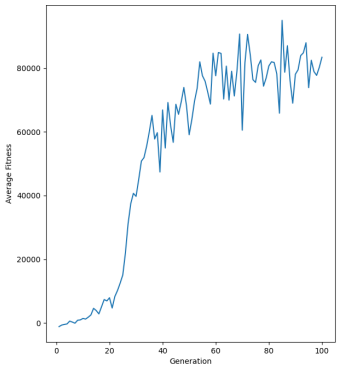

Fig. 10. Average Fitness over Generations

REFERENCES

[1] ”Pygame Front Page” Pygame documentation . https://www.pygame.org/docs/ (Accessed Nov. 1, 2024).

[2] Zbigniew Michalewicz, Marc Schoenauer, Evolutionary Algorithms, Editor(s): Hossein Bidgoli, Encyclopedia of Information Systems, Elsevier, 2003, Pages 259-267, ISBN 9780122272400, https://doi.org/10.1016/B0-12-227240-4/00065-4. (https://www.sciencedirect.com/science/article/pii/B0122272404000654)

[3] Chun-Wei Tsai, Ming-Chao Chiang, Chapter Seven - Genetic algorithm, Editor(s): Chun-Wei Tsai, Ming-Chao Chiang, In Uncertainty, Computational Techniques, and Decision Intelligence, Handbook of Metaheuristic Algorithms, Academic Press, 2023, Pages 111-138, ISBN 9780443191084, https://doi.org/10.1016/B978-0-44-319108-4.00020-4. (https://www.sciencedirect.com/science/article/pii/B9780443191084000204)

[4] Katoch, S., Chauhan, S.S. & Kumar, V. A review on genetic algorithm: past, present, and future. *Multimed Tools Appl* **80**, 8091–8126 (2021). https://doi.org/10.1007/s11042-020-10139-6

[5] Marks, Robert. (2002). Playing Games with Genetic Algorithms. Studies in Fuzziness and Soft Computing. 100. 10.1007/978-3-7908-1784-3$_2$.

[6] ”Snake Game in Python- Using Pygame module” GeeksForGeeks. https://www.geeksforgeeks.org/snake-game-in-python-using-pygame-module/ (Accessed Nov. 1, 2024).