The Snake to Beat All Snakes: Genetic Algorithm in Neural Networks

Ishizaki
Dept. of Computer Science
North Central College
Naperville, Illinois
jishizaki@noctrl.edu

Andersen
Dept. Of Computer Science
North Central College
Naperville, Illinois
dandersen@noctrl.edu

Kaleel
Dept. Of Computer Science
North Central College
Naperville, Illinois
ukaleel@noctrl.edu

Abstract— **This project explores the development of an AI-driven Snake game utilizing neural networks and genetic algorithms to autonomously control the snake. The AI receives the state of the game prior to predicting every move, weighting each input differently based on its generated weights either bred or mutated. The primary objective is for the snake to consume as many fruits as possible without colliding with obstacles. Neural networks process real-time game state inputs, including the snake's direction, danger indicators, and positions of the fruit and tail, to predict optimal movements. Genetic algorithms are employed to evolve the neural networks over successive generations, selectively breeding the best-performing networks to enhance their decision-making capabilities. The project demonstrates significant improvements in AI performance, showcasing the effectiveness of combining neural networks with genetic algorithms for real-time decision-making in dynamic environments. These findings have potential implications for various applications in autonomous systems and game AI development.**

Keywords—Artificial Intelligence, Genetic Algorithm, Neural Networks, Snake

I. Introduction

The Snake game, a classic arcade-style game, has been a staple in programming exercises and AI development due to its simplicity and engaging dynamics. This project aims to leverage advanced artificial intelligence techniques, specifically neural networks and genetic algorithms, to create an autonomous snake that can play the game effectively.

By integrating these AI methods, we address the challenge of developing a system capable of real-time decision-making in a dynamic environment.

The primary motivation behind this project is to explore the potential of neural networks optimized through genetic algorithms in controlling a game agent. Traditional approaches to AI in games often rely on predefined rules and behaviors, which can be limiting. In contrast, our approach involves the application of machine learning principles to enable the snake to learn and adapt its strategies over time, improving its ability to avoid obstacles and maximize its score. This report details the methodology, implementation, and results of our AI-driven Snake game. We examine the design and architecture of the neural networks, the genetic algorithm optimization process, and the performance evaluation metrics. The ultimate goal is to demonstrate the efficacy of combining neural networks with genetic algorithms for developing intelligent game agents, with potential applications extending beyond gaming to various autonomous systems

II. Problem Definition And Summary

The Snake game has long been a quintessential example used in AI research and game development to test various algorithms' effectiveness in real-time decision-making scenarios. Despite the popularity of traditional rule-based AIs, these approaches often lack the flexibility to adapt and evolve in dynamic environments. Our project addresses the challenge of creating an autonomous AI capable of effectively controlling the snake, maximizing its score by consuming fruits while avoiding collisions with walls and its own body. The primary goal is to develop a system where multiple neural networks, representing different "snake AIs," evolve over generations to improve their performance. Each neural network processes game state inputs, such as the snake's position relative to the fruit, its own body, and potential dangers, predicting the optimal direction for the snake to move. This approach leverages genetic algorithms to simulate the process of natural evolution, selectively breeding and mutating the best-performing neural networks to create successive generations. The problem lies in optimizing these neural networks to enhance their decision-making capabilities and adaptability in a highly dynamic game environment. By addressing this challenge, we aim to demonstrate the potential of combining neural networks with genetic algorithms for developing intelligent game agents, with broader applications in autonomous systems and AI-driven decision-making processes.

III. Work Distribution

The work was evenly distributed among all three team members. Usman created the base code architecture, constructed the neural network and genetic algorithm, and led discussions among team members. Jo created the repository and dealt with any GitHub conflicts. He also implemented and modified the existing visuals to fit the project's needs through pygame. Dylan worked extensively with the fitness function for the genetic algorithm and ensured the movements and rules of Snake were still true in the modified Snake game we worked with.

IV. Code Structure

A. Base Code

The project built upon the python code provided by GeeksForGeeks. It initially allowed for user input to play the game of Snake regularly, but through modification during the project, it now only takes input from the neural network as it decides what moves the snake should make. We also utilized neural network structure code from an article published by Medium but modified it as it works with backpropagation and a more traditional style of neural networks.

B. Neural Network Architecture

1) Structure: The neural network takes in many different inputs from the get_game_state function. The function is run prior to the calculation of every move, and is then passed to the neural network which then outputs one of four directions for the snake to move. The input layer takes in 20 different inputs, which are either one-hot encoded variables or vectors that represent the general location of a certain attribute on the grid. The neural network itself follows a basic multilayer perceptron (MLP) format with one input layer with 20 inputs, four hidden layers with varying numbers of neurons, and four different outputs for the direction the snake can move. However, rather than backpropagation, the neural network relies on the genetic algorithm to improve over time. Neural network biases and weights are modified by the cross-breeding process.

2) Inputs: The neural network receives 20 different inputs from the get_game_state function. The direction variable is a one-hot encoded variable that is split into four, representing the four directions a snake can move. Whichever variable has the value of one represents to the neural network what direction the snake is currently facing. Similarly, the danger variable is also one hot encoded, representing whether or not there is a wall or part of the snake's body to either four directions, with one representing that there is danger present. There are also positional variables that represent the position of the tail of the snake, as well as the fruit. The tail's direction is also passed to the neural network in a one-hot encoded fashion, as well as the fruit's direction.

3) Population: The initial population of neural networks consist of 200 randomly generated weights and biases that are run through the evaluate_fitness function to determine what score they get, before they are sorted and crossbred in the genetic_algorithm function. As they are crossbred, a new population of 200 is born, and the process repeats for however many generations are desired for the current run.

C. Genetic Algorithm Optimization

As the snake runs, its fitness score is calculated and stored in an array with all the other scores of the same generation, which are then sorted from greatest to lowest. The highest scoring snakes are then crossbred between the top 5 and top 10 scoring neural networks, interchanging their weights and biases between a pair of neural networks, until the children population is equal to the desired amount. In terms of optimizing this algorithm, we did not see much room for growth here, and our primary parameters to optimize were our fitness function itself, the type and number of inputs given to the neural network, and possibly the structure of the neural network itself.

D. Fitness Evaluation

The fitness function not only rewards the snake for eating fruits but also penalizes inefficient movements, collisions, and repetitive patterns, such as circular motion. The score of a snake is calculated as the snake is running from move to move. If the snake grabs a fruit, the snake's score is added to. It also adds and subtracts a smaller factor to the score depending on whether or not the snake is getting closer to the fruit. If the snake hits a wall or itself, its score is reduced, which helps when dealing with initial generations which are unable to get to fruits and more concerned with staying alive.

E. Crossover

The crossover function combines the weights and biases of two neural networks to produce a "child" network that inherits traits from both. A new network with the same architecture is created, and for each layer, random values determine whether parameters are inherited from the first or second parent. Random matrices guide this selection, with values above 0.5 copying from the first parent and others from the second. This approach ensures a balanced mix of traits, fostering diversity in the offspring and potentially combining the strengths of both parents to enhance performance after further training or evolution.

F. Mutation

We utilized dynamic mutation rates that were dependent on how well each generation was performing. If the highest max scaled fitness score was less than 0.5, we increased the mutation rate. Otherwise, we would decrease the mutation rate if we saw the performance to be adequate enough. The base mutation rate at the beginning was 0.1, as we often saw stagnation occur relatively quickly in certain models, we wanted to ensure mutation could occur in those situations to try to get out of local maxima.

G, Visualization of the Game

The visualization of the Snake game provides an engaging and intuitive way to assess the neural network's decision-making capabilities in real time. Developed using Pygame, the function displays a graphical representation of the game environment, including the snake's position, the fruit's location, and the snake's body as it grows with each consumed fruit. Key information, such as the current generation, neural network index, and the number of fruits eaten, is prominently shown on the game screen, offering valuable insights into the AI's performance. The snake's movements are determined by predictions from the neural network, which are translated into directional changes while adhering to game rules, such as avoiding collisions with walls or its own body. Beyond its visual appeal, the visualization serves a practical purpose in performance analysis. It tracks progress by logging the results of the best-performing neural networks every ten generations, specifically documenting the number of fruits eaten. This data is recorded in a text file, enabling a systematic evaluation of the AI's evolution over time. By combining dynamic graphical feedback with performance logging, the visualization function enhances understanding of the neural network's strengths and areas for improvement, which ultimately supports more effective AI training and development.

V. Experimentation

A. Experimental Results

We ran a variety of experiments to test different factors, including but not limited to the grid size, fitness function variations, and the type of inputs given to each neural network. Within this paper, we will only be looking at the results from varying grid size and the type of inputs given to the neural network. There were four inputs that we manipulated out of the 20 as to whether or not they could change the models' performance, as we found the others to be completely necessary. The four manipulated inputs were the relative fruit position and relative tail position, each represented by two X and Y coordinate variables. The grid size was also manipulated from 500x500 to 200x200.

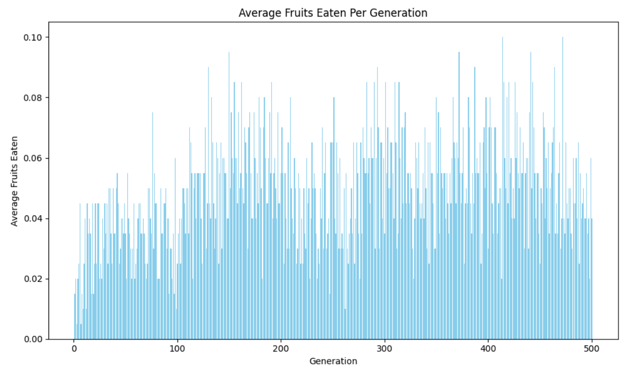

Fig 1. Figure 1 represents the average number of fruits the models with no relative fruit inputs with grid size 500x500 had obtained.

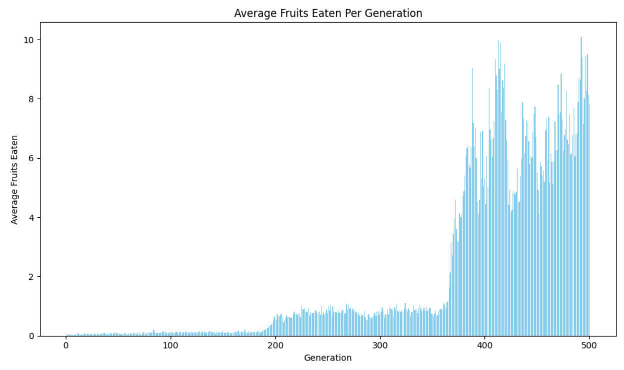

Fig 2. Figure 2 represents the average number of fruits the models with no relative fruit inputs or relative tail inputs with grid size 200x200 had obtained.

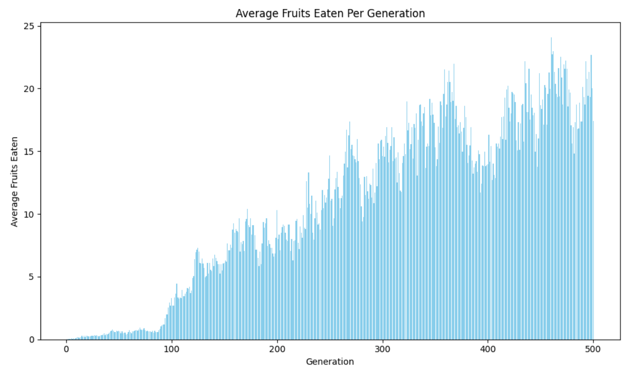

Fig 3. Figure 3 represents the average number of fruits the models with all 4 manipulated variable inputs included with grid size 200x200 had obtained.

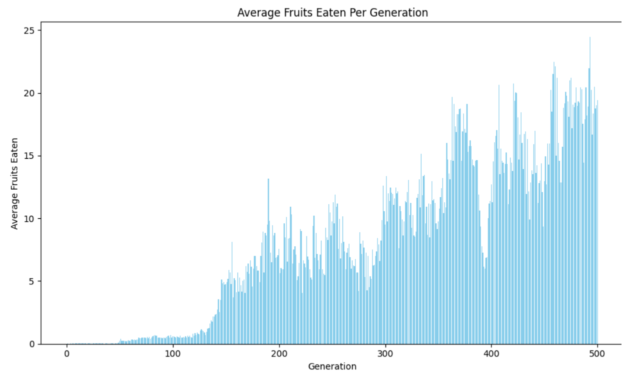

Fig 4. Figure 4 represents the average number of fruits the models with no relative frit inputs or relative tail inputs with grid size 500x500 had obtained.

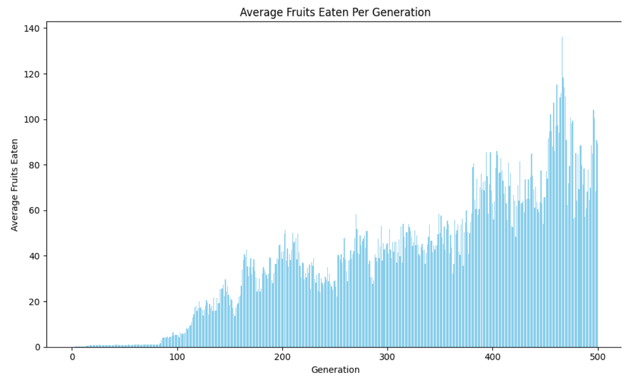

Fig 5. Figure 5 represents the average number of fruits the models with all 4 manipulated inputs included with grid size 500x500 had obtained.

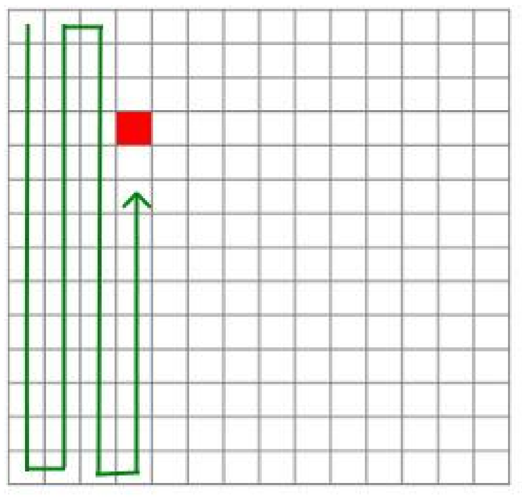

Fig 6. Figure 6 represents our predicted ideal movement of a snake

B. Discussion

The models that ran without the relative fruit position inputs ran the worst out of the five experiments, averaging less than 1 fruit per neural network across all generations. The models with none of the four manipulated inputs performed better than the model with only two. The models that performed best had all four manipulated variables added. However, the models with none of the four manipulated inputs reached an algorithm state that resembled our predicted ideal movement of a snake much quicker than the better performing models, which can be seen visually represented in Figure 6. The problem with them lies in the fact that they would decide to either hit a wall or their body while performing the movement, which caused its performance to suffer. The fact that they were also unaware of the fruit positioning also meant that they were simply scanning the entire grid over and over again, rather than attempting to seek out the fruits themselves. The better performing models also saw rare peaks of performance in reaching over 1000 fruits obtained, but still only represented a fraction of the grid.

VI. Challenges And Learning Outcomes

A. Neural Network Design Differences

We initially struggled to set up the neural network itself because of the structure differences compared to neural networks we have worked with in the past. Not utilizing both training datasets and backpropagation completely changes the structure of the neural network, as now the genetic algorithm is replacing the cost error correction.

B. Hitting Local Maxima

While our models were able to learn and play the game well to an extent, they often reached their peak and then plateaued. They would either no longer produce better results, or even produce worse results at times. In that regard, further modification of the fitness function or types of inputs may have been necessary to overcome such plateaus. On many occasions, there were times when the model would not learn at all until a couple hundred generations in, versus learning within the first 100 generations, due to sheer randomness. Additional generations may have also proven to work, but the run time for such models would be considerably longer than our current run times.

C. Run Time for Models

Our models that ran using smaller grid sizes and number of generations saw run times of under two hours. However, as we increased the grid size past a certain size, for example 500x500, the time to complete rose significantly. Raising the number of generations also significantly increased the run time, with relatively good models of grid size 500x500 and 500 generations caused the program to take over half a day to run to completion. Had we implemented parallel processing through either multithreading or CUDA, it may have been possible to see a significant drop in run time, allowing us to run higher numbers of generations.

VII. Conclusion

We were unable to breed a neural network that was able to beat the game of Snake. However, we were able to produce Snake algorithms that could at least beat most human players at the final generations, with a peak score above 1000 fruits. Further generations could have created even better models, but the time to run said models would rise exponentially to over a day, or even multiple days. The manipulation of the fitness function or other inputs could have also produced better models, but we were unable to create such models ourselves.

VIII. References

[1]GeeksforGeeks, "Snake Game in Python Using Pygame module," *GeeksforGeeks*, Apr. 26, 2021. Accessed: Dec. 08, 2024. [Online]. Available: https://www.geeksforgeeks.org/snake-game-in-python-using-pygame-module/

[2]M. Medium, "A neural network from scratch in python - ML Medium," *Medium*, Nov. 05, 2023. Accessed: Dec. 08, 2024. [Online]. Available: https://medium.com/@amazing_space_woodchuck_218/a-neural-network-from-scratch-in-python-64665118158d

[3]N. Ambuehl, "Investigating Genetic Algorithm Optimization Techniques in Video Games," Digital Commons @ East Tennessee State University. Accessed: Dec. 08, 2024. [Online]. Available: https://dc.etsu.edu/honors/748

[4]S. D. Mikulcik, "Application of Neural Networks for Intelligent Video Game Character Artificial Intelligences," Encompass. [Online]. Available: https://encompass.eku.edu/honors_theses/367