Snake AI

Grant Holmes

December 2020

1 Introduction

Inspired by natural selection and related biological concepts, genetic learning algorithms are a subset of deep learning techniques that can effectively train agents in dynamic environments.

An initial population performs a given task and those that perform best are considered the ”fittest” of the population. New members of the population are created by crossing over the traits of two or more fit members. These new members are subjected to random genetic mutations, diversifying the population set. Not all of these changes are positive, and the worse off members die off without passing down their traits. The continuation of this process over many generations eventually yields a ”naturally selected” population representing a diverse and complete solution set.

While these algorithms can only provide a non-deterministic approximation to a given problem or task, they shine when confronting complex problems without explicit solutions. They have many applications in the machine learning world such as providing an alternative way to training neural networks, learning hyper parameters, and many search and optimization problems. One such application of genetic learning algorithms is in the pursuit of training AI agents, such as the snakes from the popular game ”Snake”.

2 Methods

2.1 Engine

I first programmed a graphics engine and accompanying design tools in Python to integrate with GUI library, Pygame. I next created a basic and configurable single player game of ”Snake”. I can easily change and adjust many features on the fly such as map size, initial snake length, and game speed. Correctly designing a human played version is a critical first step, as the AI snakes use the same game environment. A strong object oriented paradigm allows the snakes to be switched between controllers completely independently from their structural components.

2.2 Neural Networks

Each AI snake has an artificial feed forward neural network to dictate its movements. The most effective network architecture has 24 inputs, a single hidden layer with 16 inputs, and an output layer with 3 outputs.

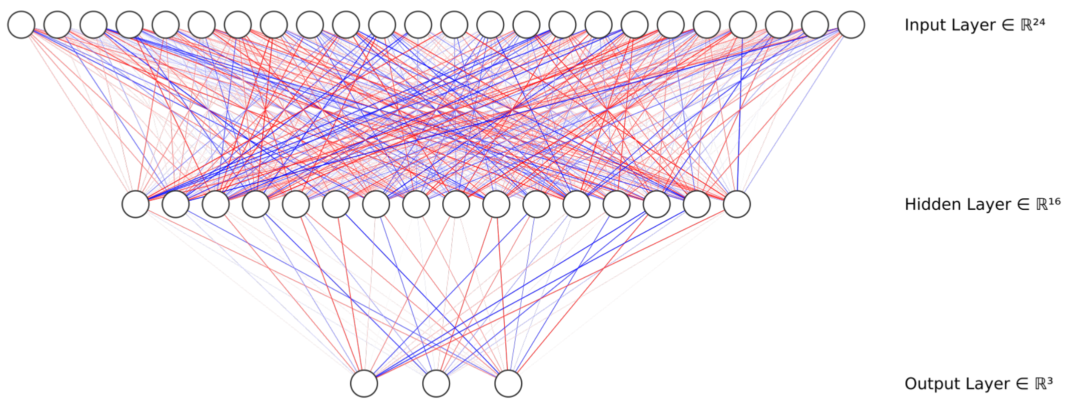

Figure 1: Neural Network Architecture

The 24 input layer is really a flattened 3x8 matrix containing important data regarding the snakes immediate surroundings. Each size 8 column vector represents the closeness of the head of the snake to a particular target in all 8 octilinear directions. The three targets are food, the snake's body, and the wall of the map respectively. Here, closeness is defined as

$$Closeness = \frac{1}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}} \tag{1}$$

where (x_1, y_1) and (x_2, y_2) are the coordinates of the snake's head and target respectively.

The hidden layer is necessary to achieve non-linearity through the network and each hidden neuron utilizes the sigmoid function as its activation.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Also using the sigmoid activation function, the output layer's activation represents the direction the snake should move. If the 1^{st} node has the greatest activation, the snake should turn left, if the 2^{nd} node is most active, the snake should go forward, and if the last node if most active, the snake should turn right.

In theory, the artificial neural network assigned to the snake should make an imformed decision about how the snake should move based on input data from the environment.

2.3 Genetic Learning

Upon instantiation, every neural network's weights and biases are randomly initialized. This makes the inputs have random, uncoordinated affects on the outputs. To actually be able to establish a productive relationship between the inputs and outputs, the network needs to be trained. Traditionally, neural networks are trained using backpropogation, a technique to proportionally adjust the network's weights and biases using multi-variable derivatives to settle the difference between expected and actual output. While backpropogation is useful in many circumstances, it requires a labeled training set. In the context of this application, the network would need access to a large set of recorded snake games where all correct moves are made.

An organic and potentially more feasible approach would be to use a genetic learning algorithm. The algorithm can be abstractly described in several steps

1. Create initial snake population, each with a unique neural network controller
2. Have each snake in the population play a game
3. Assess each snake's performance with a fitness function, forming a set of "fit" snakes
4. Pair up fit snakes and crossover their neural networks to produce a new snake for the next generation
5. Randomly mutate some of the neural networks of these new snakes
6. Repeat steps 1-5 where the set of snakes resulting from crossover and mutation is now the initial population of the next generation

This process is continued until the target number of generations has been reached, or until a sufficiently fit snake emerges from the population pool.

The fitness function is one of the most important components of the algorithm. It is used to determine which snakes performed well vs which snakes performed poorly, and drives the evolution process. Here I define the fitness function as

$$fitness(snake) = \begin{cases} \text{score}^3 * \text{age} + 1, & \text{if } \textit{snake moved in all directions} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

This function incentivizes the snake to eat food to increase its score and to survive as long as it can. These are related, as the snake starves to death and dies if it surpasses a certain number of moves without eating any food. The condition is necessary to discourages snakes from relying on only 2 directions, e.g. only going straight and turning right.

Crossover can be handled in many ways. Here, copies of each of the parents neural networks are made. A neuron of the first copy is over written by a randomly selected neuron of the second copy, and a neuron of the second copy is over written by an randomly selected original neuron of the first copy. In theory, this would allow a snake who has learned to go towards food and a snake who has learned to not crash into the wall to produce a new snake that pursues food and also doesn't crash into the wall.

Mutation is another important element in the procedure. In this application, a randomly selected weight or bias is overwritten by a random float value. These mutations allow new strategies to be pursued and to ensure that the population doesn't get stuck in a local fitness maximum.

There are 3 outstanding hyper parameters: crossover rate, mutation rate, and population size. Crossover rate and mutation rate determine the degree of crossover and mutation that occur when transitioning from one generation to the next. Population size determines the number of snakes starting out for each generation. Too small of a population size will ensure the snakes won't be diverse and robust enough to learn while too large of a population size increases training time dramatically.

2.4 Complexity and Performance

This algorithm in general can be very computationally complex and scales with the population size. Here especially, complexity and training time are important to consider as each snake has to play an entire simulated game. To improve performance, I used multi-core parallel processing. When assessing and evaluating the snakes by having them play a game, I divide the population by the number of available CPU cores and dedicate a core to each group. These cores create new instances of my program and synchronize their results after running independently in parallel. This improves run time significantly, but training time is still a big concern.

3 Discussion and Results

The combination of artificial neural networks and a genetic learning algorithm proved very successful. The top-performing and even mid-performing AI snakes are able to easily outperform most people I know. The highest score a snake achieved was 108. This snake came from a population size of 500 with a crossover rate of 0.7 and mutation rate of 0.3. This batch of snakes evolved over a 100 generations and took over 12 hours to train on a 6 core powerful CPU.

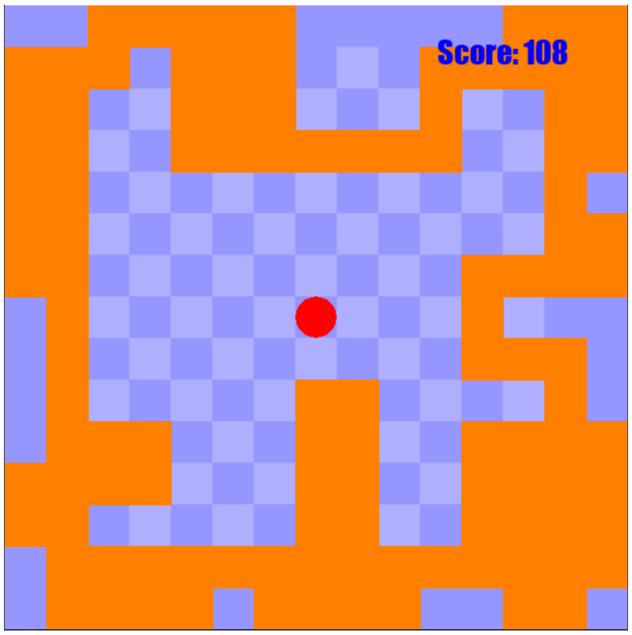

Figure 2: Tragic Death of Best AI Snake after Scoring 108 Points!

See figures at end of document for all results!

4 Conclusion

Non traditional methods such as genetic learning algorithms are powerful machine learning tools. Another interesting avenue for this project would be adding another hierarchy of neural networks. This high level network could decide whether the snake should follow its lower level neural network controller or whether it should select another behavior such as an A* path-finding algorithm or Hamiltonian cycle algorithm.

5 References

- Original Source Code: https://github.com/gholmes829/Snake-AI
- https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3: :text=A
- https://alexlenail.me/NN-SVG/AlexNet.html

6 Figures and Results Cont.

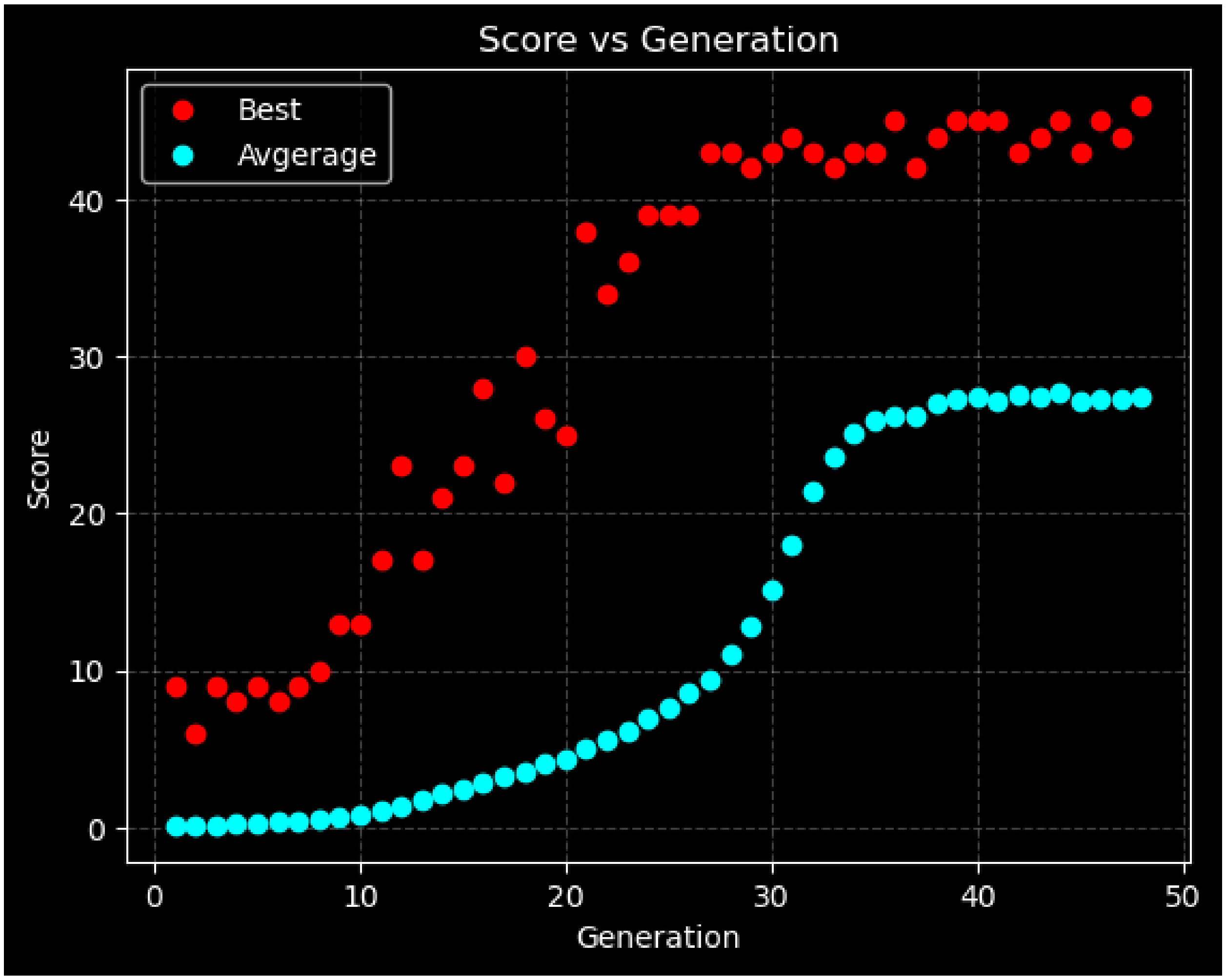

Figure 3: Scores Increase then Reach Natural Limit

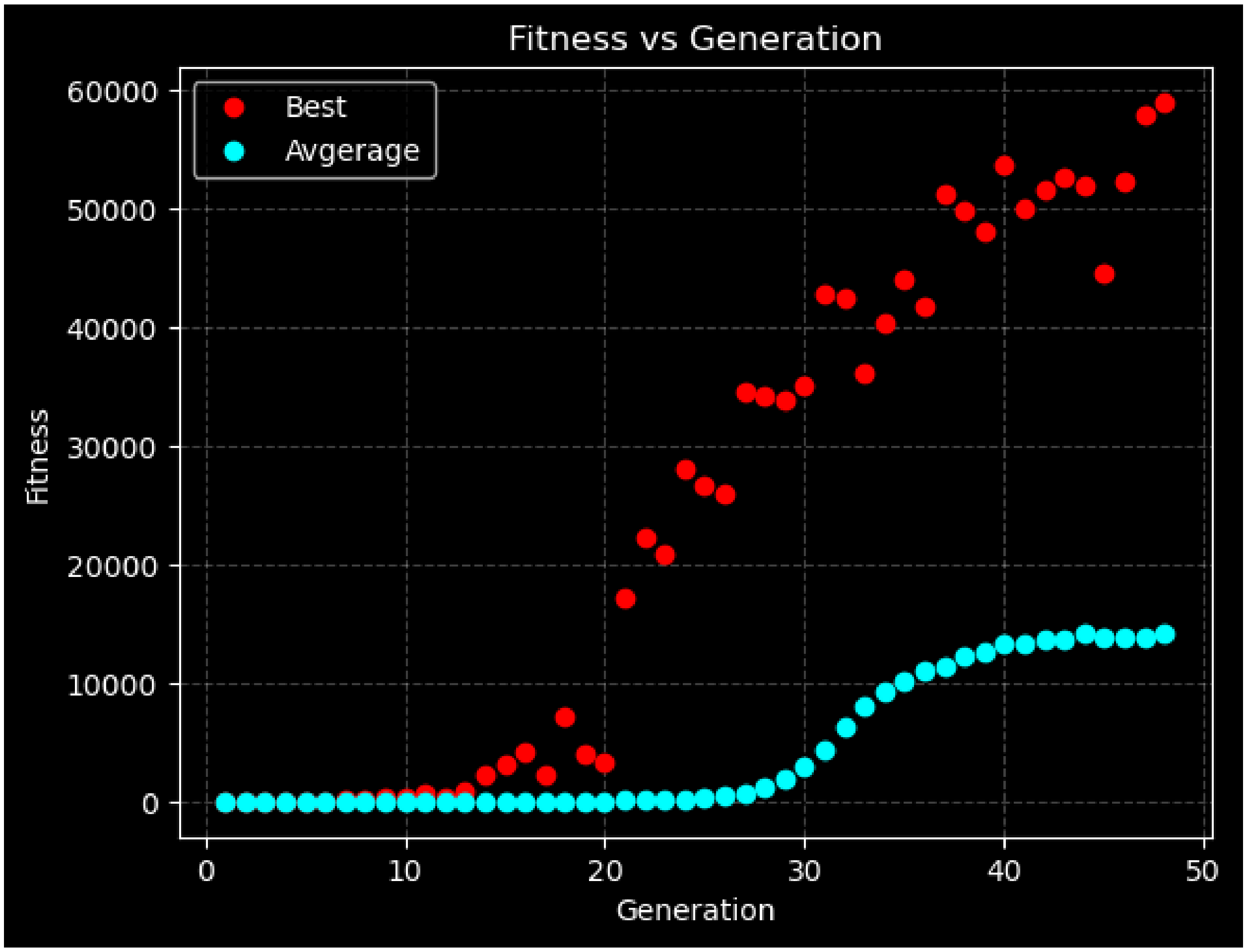

Figure 4: Best Snakes Have Much Better Fitnesses than Average

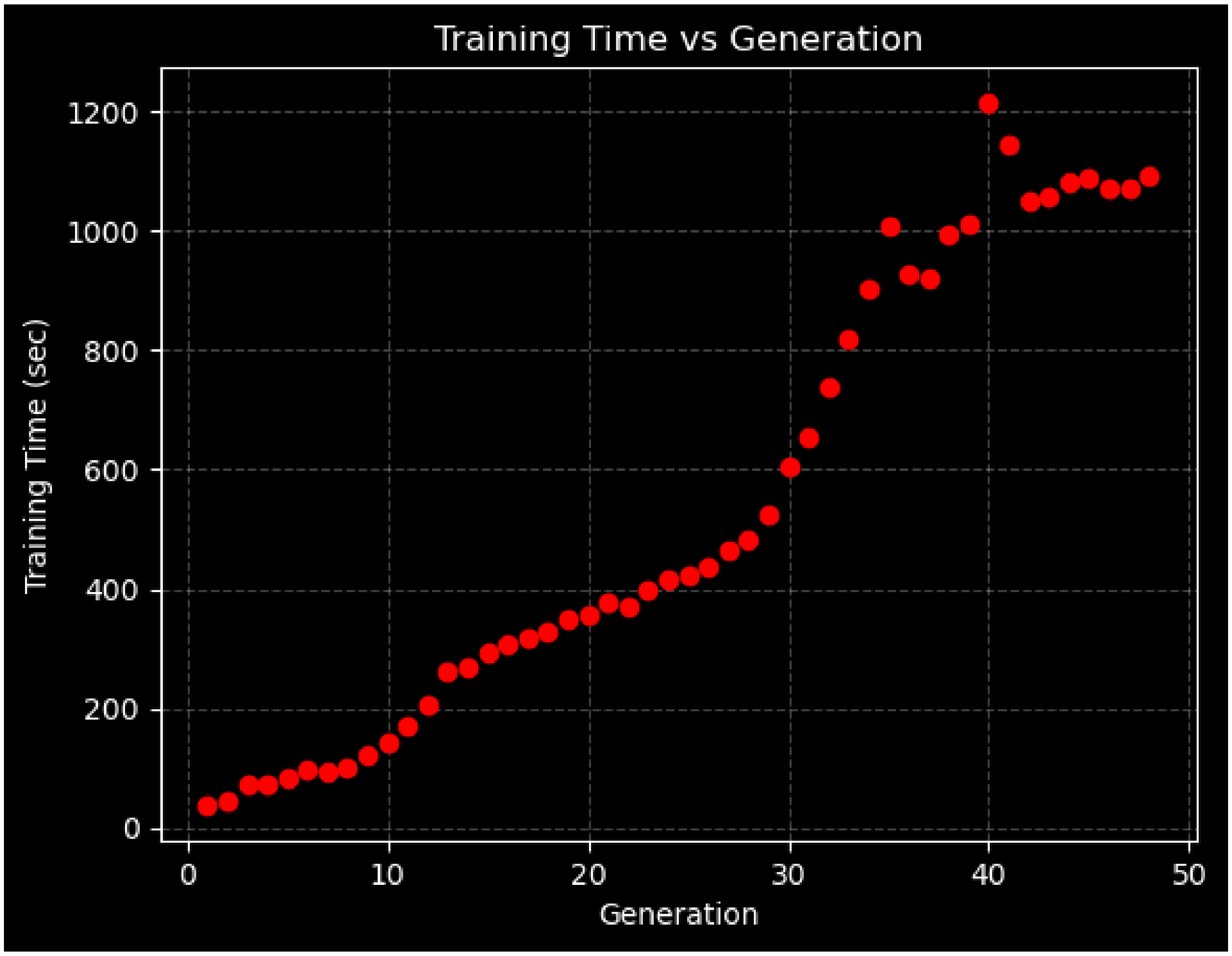

Figure 5: Training Time Increases Significantly with Complexity

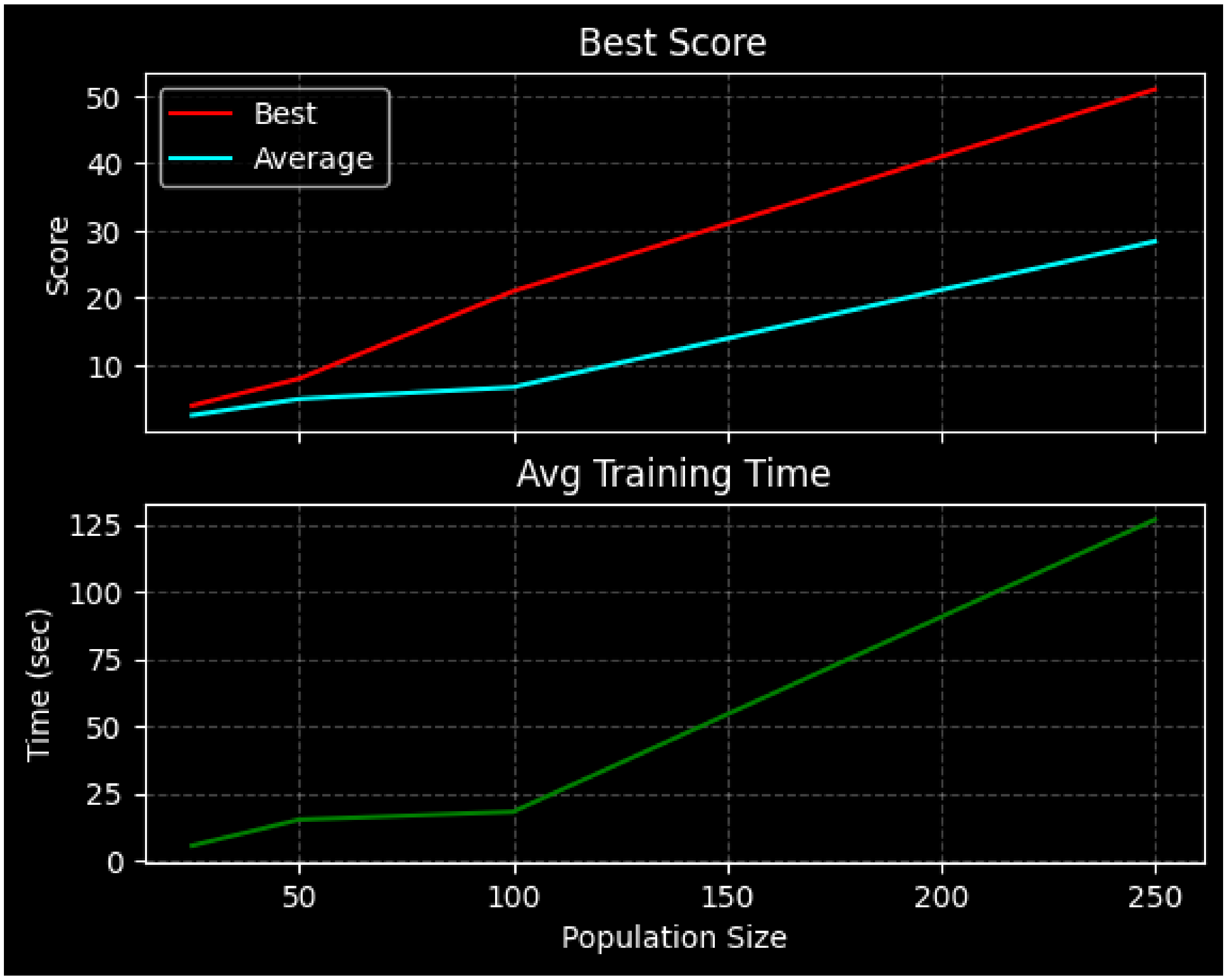

Figure 6: Higher Population Sizes Yield Better Results

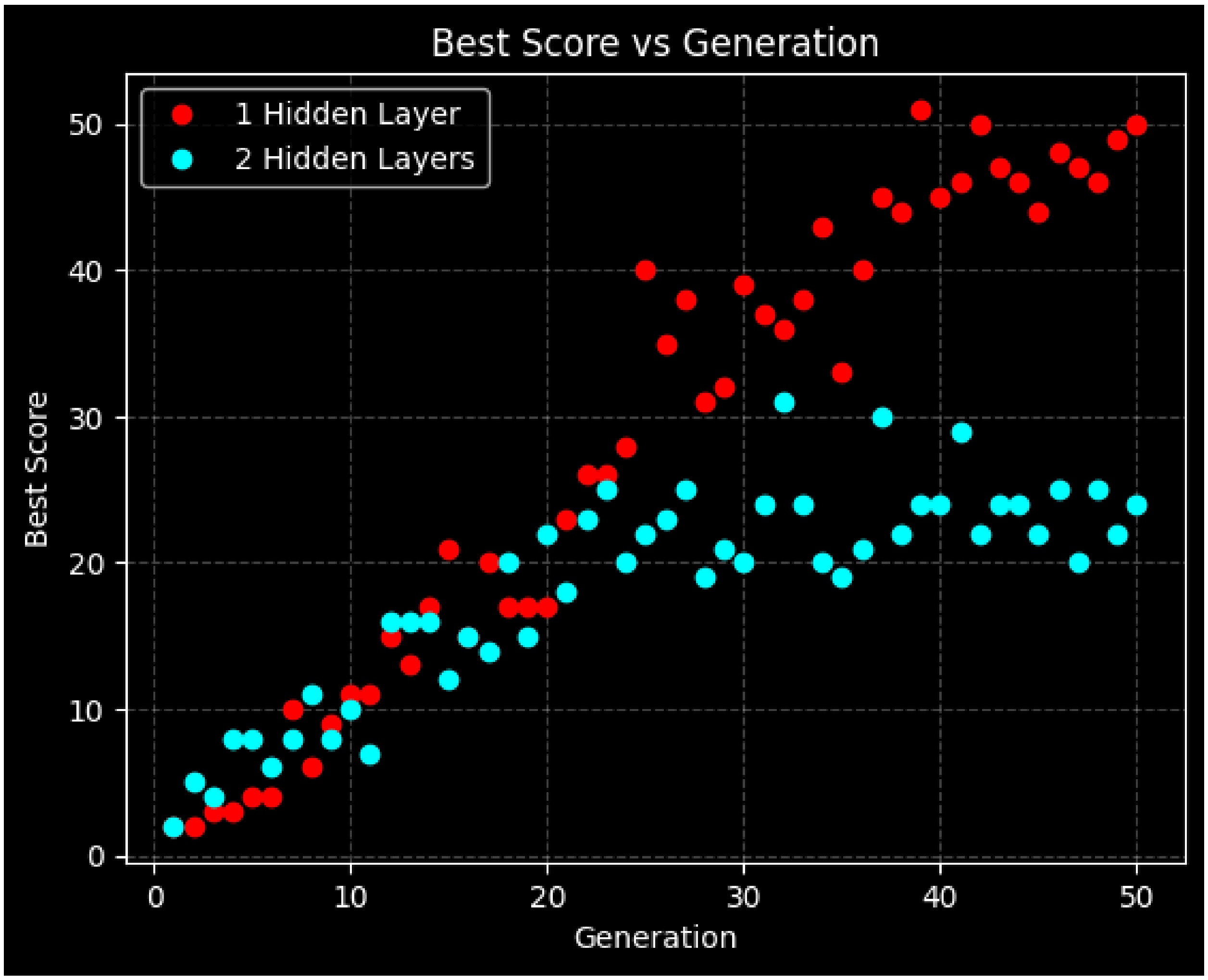

Figure 7: 1 Hidden Layer Performs Better than 2