Brainy Boa: Exploring NeuroEvolution with Snake

Jasper Pieterse (s1017897)
Daria Mihaila (s1124590)

June 28, 2024

Contents

1 Introduction

NeuroEvolution (NE) blends evolutionary programming with machine learning, using evolutionary algorithms (EAs) to fine-tune the structure and parameters of neural networks, similar to natural selection. NE is especially useful when obtaining gradient information is difficult or impossible [1]. One key application of NE is in game playing, where it offers an alternative to Reinforcement Learning [2]. In these cases, getting precise gradient information to guide learning is challenging, making NE a good choice for a gradient-free approach.

Essentially, an EA is a form of biased random search, and the design of this bias falls to the researcher [3]. While EAs can be widely used, there's a risk of treating them as a one-size-fits-all solution, which might not be as effective as using a method tailored to the specific problem [CITE]. In developing game-playing agents with NE, it's important to make thoughtful design decisions that incorporate domain knowledge into the algorithm.

The Snake game, where a snake eats apples and grows longer until it runs into the wall or its own body, is an ideal environment for studying how these design choices affect the learning of the game-playing agent. Its simplicity and computational feasibility make it a suitable candidate for exploring NE concepts without being bogged down by complexity.

A critical design choice is determining what information the agent receives about its environment and how it interacts with it, represented by the neural network's input and output, respectively. There are numerous online implementations of NeuroEvolution applied to the Snake game, each employing different inputs and outputs [4]–[6]. However, there is no consensus on the most effective approach. This gap in established wisdom presents an opportunity for analysis and experimentation, potentially yielding insights applicable to other games or NE methods in general.

1.1 Problem Statement

In this work, we aim to use the NeuroEvolution of Augmenting Topologies (NEAT) method to evolve the weights, biases, and topology of a neural network that learns to play the Snake game. Our goal is to analyze and explore how varying design choices, such as input/output features and environmental constraints, impact the learning process of the game-playing agent. Specifically, we intend to explore the following objectives:

1. Examine the influence of the frame of reference for inputs and outputs.
2. Assess the impact of relative versus binary encoding input features.
3. Investigate how added obstacles affect the agent's behaviour.
4. Evaluate the agent's adaptability to environmental constraints by reducing the allowed time to eat an apple.

For these objectives, we aim to analyse both the qualitative change in the behaviour the game-playing agent learns, as well as quantitative aspect of how 'fast' the agent learns. From this analysis, we aim to extract insights that can be applied to NE algorithms in general, beyond just the Snake game.

2 Literature Review

In this section, we review aspects of the established literature directly relevant to our problem statement.

2.1 Adaptation to Environmental Constaints

Risto Miikkulainen, recently explored the use of NE for evolving both neural network topology and hyperparameters in Neural Architecture Search (NAS) [7]. The study noted an interesting adaptation to environmental constraints: the necessity to only partially train networks due to limited resources, caused the evolutionary process to bias the evolution toward fast learners rather than top performers.

We would like to explore this aspect of neuroevolution in our project as well. We will do this by adding an experiment where the snake has a limited time to eat an apple, which should encourage the development of faster, more efficient snakes rather than just longer ones. This will help see how adapting to constraints can direct evolutionary processes in neural networks.

2.2 Fitness Function Design

The work of Vignesh et al. [8] investigates how varying fitness functions affect the task performance of the snake. They develop both greedy and non-greedy fitness functions to evaluate their effectiveness in environments with and without dynamic obstacles. For example, for the non-greedy strategy, they implemented a reward for the snake staying alive, countering the greedy strategy where the snake overly focuses on eating the apple, sometimes leading to its demise by running into its tail.

While Vignesh et al. focused mostly on altering the fitness function to adapt behaviour, it did not explore the potential of different input encodings to adapt behaviour. Our project aims to explore the importance of varying and optimizing the input features of the snake, rather than analyzing different fitness function strategies.

3 Methods

In this section, we explain the setup for the Snake game environment, the NE algorithm we used to train the game-playing agent and our visualization methods. We then detail the series of experiments conducted to analyze how different input features influence the snake's learning and strategies.

The complete source code for our Brainy Boa project, along with necessary data and usage instructions, is available on our GitHub repository. The Snake game was developed using the PyGame library [9]. The NEAT Algorithm was implemented using the `neat-python` package [10]. Our code is a modified version of the code in this repository, which served as a baseline model. The main modifications made to this source code include:

- Refactored the code into a class structure, allowing separate instances of the Brainy Boa agent to run and compare. Previously, the code used global variables, requiring a full restart for each run and external storing of results for comparisons.
- Added the option to enable a dynamic obstacle.
- Added various input and output encodings (described in the methods).
- Added the option to switch from a cardinal frame of reference (NSEW) to the snake's frame of reference.

3.1 Game Environment

The goal of the Snake game is to guide the snake to eat as many food blocks (apples) as possible without biting itself or colliding with a wall. The snake starts with a body length of one and grows by one unit each time it eats an apple. The game ends if the snake collides with a wall, runs into itself, or fails to eat an apple within 100 time steps. This time limit was chosen to be equivalent to the grid size (10x10), and can be adjusted to impose environmental constraints on the snake, and we conducted an experiment to explore the effects of this adjustment.

Each simulation begins with the snake and an apple randomly placed on a 10x10 grid, with the snake's initial direction also randomized. The snake's movements are determined by the outputs from a neural network, which chooses the next direction based on sensory inputs from the current game state. The game state is updated based on this chosen direction. Additionally, there is an option to enable a dynamic obstacle—a 1x1 square that spawns randomly on the grid and changes position each time the snake eats an apple. This obstacle is programmed to never spawn directly in front, left, or right of the snake, nor on the snake's body.

3.2 Input Features

The neural network controlling the snake is a simple Multilayer Perceptron (MLP), whose structure evolves through the NEAT algorithm. The network receives a set of input features, encoded as values between zero and one, which help the snake navigate the game. The outputs of the network correspond to the possible movement directions of the snake, with the direction chosen based on the output neuron with the highest activation.

We hypothesize that optimal inputs for the snake involve balancing simplicity with sufficient information to enable advanced game-playing techniques. We provided the snake with handcrafted input features based on its environment. These features include:

- Relative Wall: Inverted distances to walls in each direction to fall within the range [0,1].
- Relative Body: Inverted distances of the closest segment of the snake's body in each direction.

- Relative Apple: Inverted distances of the apple's position relative to the snake's head in each direction.
- Relative Obstacle: Inverted distances to obstacle in each direction

The code allows for any combination of these features to be used as input information for the snake. Notably, the snake can "see" through obstacles, apples, and its own body to detect walls, obstacles, or apples.

3.2.1 Frame of Reference

We implemented an option to provide the input and outputs from the snake's first-person point of view, differing from the human perspective of the game. This approach offers two potential advantages:

- It removes the need for the neural network to understand the size of the environment, the locations of various objects within it, and the snake's heading direction. By simplifying the problem from the AI's perspective, we make it easier for the AI to solve.
- Smaller initial networks are easier to evolve and find solutions more easily [11]. Using the snake's frame of reference reduces the number of input nodes from four (NSEW) to three (front, left, right), which simplifies the network at the cost of some information. Additionally, in the NSEW frame, the snake always has one 'invalid' output since it cannot move back into its tail.

3.2.2 Dummy Inputs

To determine whether the NEAT algorithm's performance is influenced more by a reduced network size or the quality of input features when switching between frames of reference, we controlled for these variables using dummy inputs. These inputs encode no useful information but ensure that the network size for the 'snake' frame of reference matches that of the 'NSEW' frame of reference. We experimented with two options:

- A constant value of 0.5: This is easier for the network to filter out, helping us determine if the network benefits from reduced complexity in the input space.
- Randomly varying dummy inputs. At each step, a random number is drawn from a uniform distribution $\mathcal{U}(0, 1)$. This introduces more noise than the constant value, providing a more challenging scenario.

Additionally, we added a dummy output that had no influence on the state of the game to maintain consistency in network architecture.

3.2.3 Binary Features

The binary features are motivated by the idea that the snake needs only the following information to make decisions:

1. Where is the apple?
2. In which direction can I move without dying?

Absolute distances are not necessary to obtain this information. Inspired by this, we added the option to use the following input features with binary indicators [1]:

- Binary Wall: The presence of walls directly next to the snake's head in each direction (1 for wall, 0 for no wall).
- Binary Body: Binary indicators of segments of the snake's body immediately next to the snake's head in each direction (1 for body, 0 for no body).
- Binary Apple: Binary indicators of the apple's position of apple being present along the axis relative to the snake's head in each direction (1 for apple, 0 for no apple).
- Binary Obstacle: Binary indicators of obstacle immediately next to the snake's head in each direction (1 for obstacle, 0 for no obstacle)

[1] An observant reader might notice that to have this information, one could combine the wall and obstacle features into a single one. This point is discussed further in the discussion section.

3.3 Neuroevolution Method

The key concept of EAs is to evolve a population of solutions through selection, crossover, and mutation. In NE, these solutions are represented by neural networks. We used the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [11] for evolving our neural networks. NEAT is a method that adjusts both the weights and the structure of neural networks through evolutionary algorithms. It can add or remove nodes (neurons) and connections during the evolutionary process, allowing the network to adapt its complexity to the problem at hand. We chose NEAT over simpler NE methods that evolve only weights and biases because NEAT is a well-established method [12] and has a readily available Python package, `neat-python`.

3.3.1 General Setup

We employed an evolutionary process for a population of N_{pop} neural network genomes, evolving over N_{gen} generations. Each genome's fitness was assessed based on the number of apples the snake consumes before the game ends, whether through collision or reaching the maximum time limit. The evolution process concluded after completing the designated number of generations. To ensure reliability, each genome's fitness score was averaged over 10 runs.

Occasional 'lucky' or 'unlucky' runs caused sharp fitness spikes, which diminished in subsequent generations [13]. Increasing the number of fitness evaluations could mitigate these fluctuations but would also increase computation time. We opted for 10 runs to remain computationally feasible.

To further ensure reliability, we conducted five runs with 30 generations each. Although some experiments did not converge within this timeframe, longer runs (500+ generations) indicated that the overall strategy evolved by the snake did not significantly differ. We focused on measuring quantitative rather than qualitative differences, balancing runs and generations to account for NEAT's stochasticity while remaining within feasible computation time.

3.3.2 Network Initialization

Following the original NEAT paper [11], we started with small initial networks to take advantage of evolving topology. This approach searches for solutions in the lowest-dimensional weight space, introducing new structures incrementally through structural mutations. Only beneficial structures survived through fitness evaluations.

We initialized neural networks with an input layer consisting of selected input features and an output layer with selected output features, starting without hidden nodes to minimize initial complexity. Initial connections were randomly made for 50% of the nodes, restricted to feed-forward connections to avoid loops. Nodes used a standard sigmoid activation function, and connection weights were initialized with a mean of 0.0 and a standard deviation of 1.0.

3.3.3 Crossover Strategy

In NEAT, matching genes are parts of neural networks that align between genomes, while excess genes represent structures absent in the other genome. During reproduction, matching genes were randomly selected from either parent, and all excess genes were included from the fitter parent. If both parents were equally fit, excess genes were chosen randomly from both.

We maintained a constant population size of N_{pop} genomes, using elitism and a survival threshold. The top two individuals from each species automatically moved to the next generation, and only the top 20% of the remaining individuals, based on fitness, were allowed to reproduce. Various elitism rates were tested, and we found this rate provided the best results for our analysis.

3.3.4 Speciation Strategy

NEAT groups similar network architectures into species through "speciation," protecting new structural innovations by allowing them time to improve over generations before competing with the broader population. Speciation is based on a compatibility score calculated using the number of excess genes and the average weight differences of matching genes, including disabled ones. The formula used is:

$$\delta = \frac{c_1 E}{N} + c_2 \cdot \overline{W}$$

where c_1 and c_2 adjust the importance of each factor, and N is the number of genes in the larger genome.

Species were formed by calculating distances between genomes. Each species was represented by a random genome from the previous generation. A new genome was placed in the first species where it matched the representative genome; if no match was found, a new species was formed with the genome as its representative. We used a compatibility threshold of $T_c = 3.0$ to classify genomes into species based on their similarities.

3.3.5 Mutation Strategy

In NEAT, genomes can undergo mutations each generation that may affect weights, biases, nodes, or connections, and these types of mutations can happen simultaneously. *Connection mutations* create a new link between two previously unconnected nodes with a randomly assigned weight.*Node mutations* involve disabling an existing connection and inserting a new node in its place, effectively splitting the connection into two: one leading into the new node with a weight of 1, and another leaving it with the original weight. This approach reduces the impact of mutations, allowing the network to integrate new changes smoothly.

The mutation rate for existing connection weights is set with a specific value μ_{weight} with a mutation power[2] of 0.5, and weights could vary between -30 and 30. Likewise, the bias mutation rate is denoted by μ_{bias}, with the same mutation power of 0.5. The mutation rate of both adding and removing nodes is defined by μ_{node}. The mutation rate of both adding and removing connections is defined by μ_{connect}.

For the weights and biases, the μ mutation rate has a replacement rate η counterpart. When a μ mutation occurs, a randomly drawn sample from a zero-centered Gaussian distribution with an std equal to the mutation power is added to the existing value. However, η corresponds to the rate at which a mutation can completely replace a value with a newly initialized value

3.4 Visualization Methods

We used several visualization techniques to both monitor and present the results of our experiments. We tracked the evolutionary progress across generations, focusing on metrics such as average fitness, the standard deviation of fitness, and the fitness of the top-performing genome. Importantly, we visualized the top-performing genome at each point in the evolution process. This approach means that rather than tracking the evolution of a single genome over time, we followed the evolution of the best genome within the entire population at each generation.

To analyze the behaviour of the snakes and the strategies they learned, we simulated a selected genome playing the Snake game, highlighting the neural network activations that influenced the snake's decisions. Excitatory connections are marked in orange and inhibitory ones in blue, with colour intensity indicating the level of activation.

To support our analysis of the strategies and behaviours, we recorded specific simulations that most accurately reflected the typical patterns and strategies observed across multiple runs. These simulations were chosen based on criteria such as the frequency of specific behaviours, consistency in decision-making patterns, and the overall success rate of the strategies employed.

4 Results

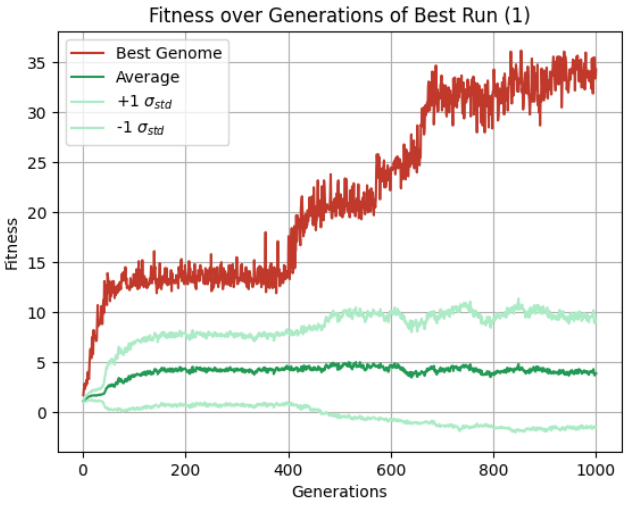
In this section, we first provide a qualitative analysis of the snake's behaviour and its learned strategies, followed by a quantitative comparison of the fitness achieved with different input and output feature combinations.

The baseline model, utilizing the NSEW frame of reference with relative wall, relative body, and relative apple input features, achieved a maximum average fitness of 35.8 over five runs when trained for 1000 generations. The best-performing run is visualized in Figure 1a. In comparison, the same model with binary input features achieved a maximum average fitness of 41.3 over five runs, as shown in Figure 1b.

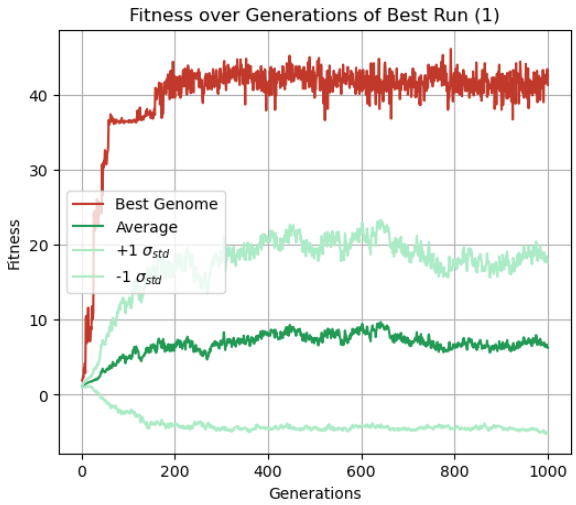
4.1 Strategies

Throughout our experiments, we consistently observed the evolution process converging to a strategy we refer to as the *Wall Hugger* strategy, regardless of the input features used. This strategy involves the snake trailing along

[2]Mutation power refers to the standard deviation of the zero-centered Gaussian distribution from which a bias mutation is drawn



(a) All Relative Input Features



(b) All Binary Input Features

Figure 1: **Left:** Evolution of the fitness across 1000 generations for all relative input features in the NSEW frame of reference. **Right:** The same using all binary input features. The red line indicates the evolution of the best genome in the population, while the dark green line tracks the evolution of the population's average fitness. The light green lines indicate the standard deviation of the population fitness. The model with relative input features takes about 800 generations to converge to a fitness of about 33. The model with binary input features takes about 200 generations to converge to a fitness of around 41.

a certain number of walls (two, three, or four) before pursuing the fruit. Once the fruit is consumed, the snake returns to the initial wall to repeat the cycle.

The Wall Hugger strategy variants include:

- 2-Wall Hugger, typically achieving a fitness between 0 and 15.
- 3-Wall Hugger, typically achieving a fitness between 15 and 30.
- 4-Wall Hugger, typically achieving a fitness between 30 and 45.

Improvements in fitness can be either gradual or sudden. Gradual improvement is exemplified in the 700-1000 generation region of Figure 1a. The 3-Wall Hugger clip was taken at generation 630, just before the second fitness spike. The 4-Wall Hugger clip was taken just after this spike at generation 650. An improved version of the 4-Wall Hugger can be seen in Figure 1b. The difference is that the snake returns to the wall it came from instead of aiming for the 'third wall,' as shown in this clip (taken from the best genome in Figure 1b).

All versions of this strategy share a common trait: when effective (i.e., the snake avoids random turns into walls or its own body), the snake typically dies by running into its own tail. Improvements in fitness occur precisely because this running into its own tail is avoided for longer.

When adding obstacles to the environment, the snake employs the same strategy but learns to avoid the obstacles. However, it often becomes stuck in a loop where it avoids the obstacle but cannot pursue the fruit from the wall, resulting in the game ending due to reaching the time limit. This effect is highlighted in this simulation, taken from a converged run with our optimized snake model (detailed further below).

When reducing the number of timesteps the snake has to eat an apple, the snake learns to avoid its own body and employs a different strategy. It still trails along the wall but now goes for the apple more frequently as soon as it appears in its line of sight, rather than waiting to dive for the apple from a fixed wall. The snake often dies because it runs out of time. This behaviour is showcased in this simulation.

In an experiment where we evolved the model for 1000 generations with all binary input features and 30 timesteps to eat the apple, we observed fitness convergence within 30 generations, fluctuating around a value of 20. Overall, the fewer timesteps the snake had to eat the apple, the lower the fitness it could achieve (results available on GitHub).

4.2 Quantitative Input Comparison

We observed a significant difference in fitness between two frames of reference: the cardinal (NSEW) and the snake's (both input and output). Over 30 generations of simulation, the average population fitness differed by approximately 3, as shown in Figure 2. The fitness gap for the best genomes, although not shown here (available on GitHub), was approximately 20. Additionally, simulations with fixed or random dummy inputs performed similarly to those without dummy inputs, albeit with large standard deviations that prevent a definitive conclusion on the superior approach.

When obstacles were introduced, similar trends were noted. However, overall fitness was lower, and the gap between the NSEW and snake frames reduced to approximately 0.7. Notably, the standard deviation for high-performing runs was lower in this scenario (plot available on GitHub).

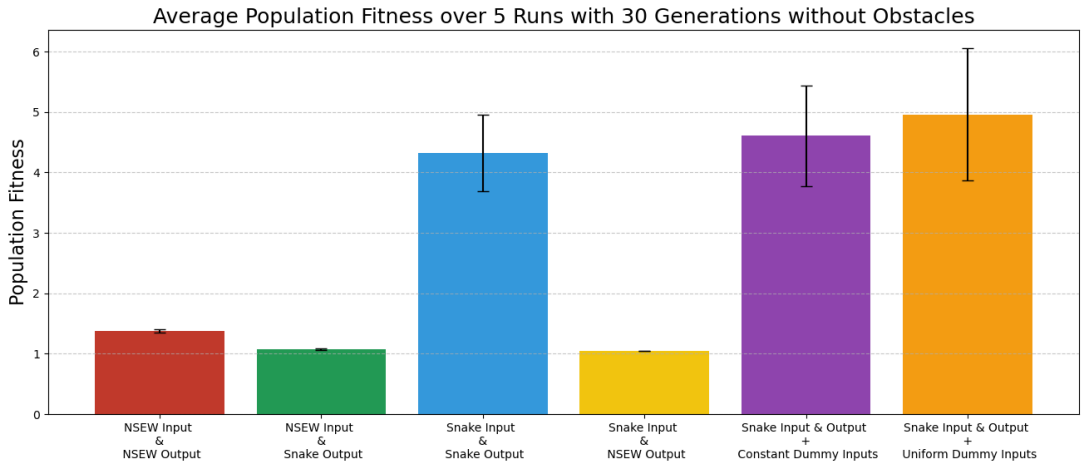


Figure 2: Average population fitness over 5 runs with 30 generations without obstacles for different input and output feature combinations. The red bar denotes the fitness for NSEW input and NSEW output, the green bar for NSEW input and snake output, the blue bar for snake input and snake output, the yellow bar for snake input and NSEW output, the purple bar for snake input and output with constant dummy inputs, and the orange bar for snake input and output with uniform dummy inputs. Error bars represent the standard deviation of the population fitness.

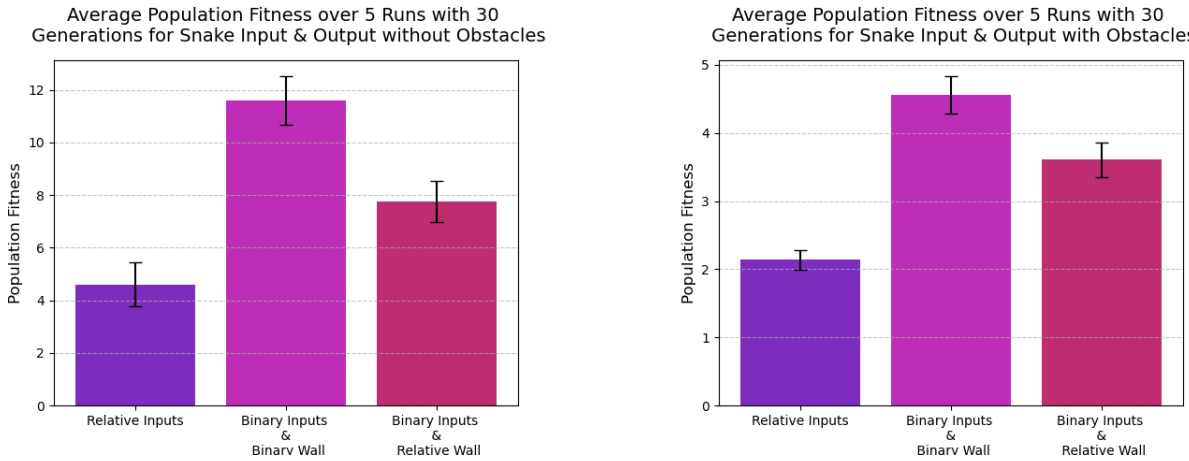
Next, we conducted an experiment on the snake's input features across three different scenarios: all relative, all binary, and mixed (all binary except for the wall, which is relative), as shown in Figure 3b. We observe an improvement of about 7 in population fitness for the all-binary input features compared to the all-relative input features. For the winner solution (available on GitHub), the fitness gap is about 10 after 30 generations. The binary input features achieve the highest average fitness to date, at 38.2. The mixed input features perform better than the relative features but worse than the all-binary features. Introducing obstacles results in the overall same trends, albeit paired with a decrease in population fitness of about 50%, as shown in Figure 3a.

5 Discussion

In this section, we discuss our results, the opportunities they present for future work, and the limitations of our study. We suggest these could be topics for students in future classes. Our project is well-documented, providing a solid foundation for further exploration, implementation, or addressing of the limitations. Additionally, Appendix B presents further ideas that, while not directly relevant to our current discussion, could inspire future work.

5.1 Stagnant Strategies

We observed that there is almost no change in the learned strategy. Changing the input/output features of the network alone does not induce a change in observed behaviour but merely helps the algorithm reach the same behaviour faster. We hypothesize this is because the "wall hugger" strategy forms either a global or local minimum in which the NE process gets stuck. This minimum might remain regardless of the input features used.



(a) Obstacles Disabled

(b) Obstacles Enabled

Figure 3: Average population fitness over 5 runs comparing relative, binary, and mixed input strategies. (a) Performance without obstacles. (b) Performance with obstacles. Error bars represent the standard deviation of the population fitness.

If the wall hugger strategy is a global minimum, it would imply that the input features we chose are ill-fitted to adopt a new strategy, meaning the snake does not have the right information to learn a better strategy. However, this seems unlikely. For example, by looking at the video clips of the wall hugger, it seems the snake can extend its lifetime, and thus its fitness, by simply learning to avoid crashing into its own body once it gets long. The snake has the information to do this but does not use it. However, the algorithm is capable of using this information, as observed when we lowered the number of timesteps the snake has to eat an apple, exerting environmental pressure on the snake.

One might wonder why, if the snake did learn to avoid its own body (something we would label as beneficial, a 'better' strategy), it still failed to obtain a better overall score. There are two main reasons for this:

1. The longer the snake gets, the more steps it needs to reach the apple (it needs to circumvent its own body). At some point, the 'optimal' number of timesteps to reach the apple is outside the time limit. To address this, one could try to have an adaptive time limit, where the time limit is increased as the snake gets longer.
2. Given the current inputs, the snake often cannot 'find' the apple in time. This is because it has no vision of where the apple is, preventing it from taking the quickest route. Diagonal vision can help the snake spot the apple earlier.

One solution to the problem of local minima would be to configure NEAT to perform a wider search of the space by changing its parameters to favour exploration over exploitation. This could be done by carefully tuning more aggressive mutation rates or adapting the crossover strategy used. However, it is important to note that for this particular problem, where we already supply the algorithm with human bias through the hand-crafted input features, we can more explicitly emphasize this bias in our search of the parameter space. For example, we chose to give the network information about its body so that it could use it to avoid running into its own body. If we know what result we want (the snake not running into its body), we perhaps need to enforce this more than just giving it the information.

This can be done in various ways, such as altering the fitness function or by putting environmental constraints on the snake. Currently, our fitness function awards a point for each apple eaten. We could refine this by incorporating penalties for moving away from fruit or rewards for navigating toward it, avoiding obstacles, or simply surviving longer, potentially leading to diverse behaviours.

5.2 Model Optimization

Our results show that switching to the snake's frame of reference and using binary input features significantly increases fitness. This is largely because the "wall hugger" strategy only uses information that is directly supplied by these input features: The snake essentially turns left every time has a wall directly in front of it, then turns left towards an apple from a fixed wall. Depending on its action directly after eating the apple (turn left, go straight,

turn right, turn right twice), you get different variations of the wall hugger strategy, repeating this cycle. Any other information, such as relative distances, is redundant for this strategy to work, and thus hinders learning.

The models with dummy inputs surprisingly do not perform significantly better or worse than the one without them. This suggests that the improvement in fitness for the snake's frame of reference is not because a smaller network is easier to evolve but rather due to the simplification of the problem.

When designing input features, it's crucial to consider the desired outcome. The snake frame of reference and binary input features optimize the wall hugger strategy, which might not be ideal for developing more complex strategies. For exploring a variety of strategies, starting with more general input features is advisable, although this requires more computational resources. Once a suitable strategy is found, the evolution process can be optimized by fine-tuning the input features based on the agent's needs. For instance, combining wall and obstacle input features might enhance performance but would likely reinforce the wall-hugger strategy even more. Binary input features may not be the best starting point for exploring new strategies.

5.3 Fitness Improvement Hypothesis

One open question is what the actual causes are of improvements in fitness. Figure 1 showcases that changes in fitness can occur both gradually and rapidly. One aspect of fitness improvement is strategy change, such as the snake learning to add small loops when consuming fruit to increase path length.

We hypothesize a second source of improvement comes from fixing mistakes. Two common mistakes observed in our simulations are:

- Deaths due to random wall/body collisions.
- Deaths from suicidal apple chasing, where the snake sometimes risks the 'safe' route for a second apple.

We hypothesize that small, gradual improvements come primarily from fixing infrequent errors the game-playing agent makes. Larger spikes in fitness can occur because of either adopting a more beneficial strategy or by fixing frequent mistakes. Thus, the frequency of mistakes in randomized trials is proportional to the size of fitness improvement once the error is fixed.

To test this hypothesis accurately, we need to measure how often specific mistakes occur. Given the complexity of the snake game, this is challenging. A feasible, though labour-intensive, approach is to repeatedly run a specific genome to track conditions before death, establishing mistake frequency.

The neural network is deterministic, so it behaves the same given identical initial conditions and apple/obstacle spawns. By evolving a network through repeated deterministic simulations to fix a specific mistake, we can then test it in a randomized setting. The fitness improvement from fixing the mistake can be measured and compared with its frequency. This process, repeated for multiple mistakes, can establish a connection between mistake frequency and fitness improvement, either proving or disproving the hypothesis. While simpler methods might exist, this approach's complexity is beyond our current project's scope.

5.4 Memory Feature

We experimented with a memory input feature that stored the last N directions the snake moved in and included these in the sensory input. The directions were encoded as $1/(N+1)$ for $N = \{\text{up: } 0, \text{down: } 1, \text{left: } 2, \text{right: } 3\}$ in the NSEW frame and $N = \{\text{forward: } 0, \text{left: } 1, \text{right: } 2\}$ in the Snake frame of reference.

Our initial experiments showed a decrease in fitness, so these results were not included in the main study but are available on GitHub. This decrease might be due to the complexity of the encoding. Future work could explore using a simpler binary encoding that specifies whether the previous turn was a left or right turn, which might perform better due to its increased simplicity.

5.5 Limitations

Our project has several important limitations that we chose not to address, focusing instead on achievable results with the available tools and time.

Firstly, the `neat-python` package provides limited statistics and visualization tools. We can only visualize the top-performing genome at each generation, which may give the false impression that a single genome is being optimized

over time, as shown in Figure 1a. In reality, this likely represents multiple genomes continuously taking over. This limitation prevents us from analyzing the top n-solutions or their average genetic distance effectively. Although a speciation plot is available, it lacks depth and insight. Additionally, while theoretically possible, our attempts to restore populations from checkpoints and analyze each genome have been unsuccessful.

Another limitation is our lack of analysis of the network topologies that evolved. We omitted this due to the absence of a quantitative and systematic method for analyzing the structure of the population as a whole, allowing only loose claims about the structures of the best genomes. Additionally, our fitness versus generation plots sometimes show negative fitness values for the standard deviation, which is theoretically impossible since fitness scores cannot be less than zero—the worst-case scenario is a fitness of zero, indicating the immediate death of the snake. We used the standard deviation as supplied by the `neat-python` package, but a custom implementation using the interquartile range for error measurement would have provided a more accurate measure of variance.

6 Conclusion

In this study, we investigated how different input and output features affect the evolution of strategies in a game-playing agent playing the Snake game using NeuroEvolution of Augmenting Topologies (NEAT). Our results indicate that altering input features mainly accelerates the learning of a fixed strategy rather than changing the strategy itself. While the provided information is essential for the NE algorithm to develop intelligent strategies, it does not ensure that the algorithm will fully exploit this information. A broader search or more biased search could be beneficial.

We found that optimizing input features can significantly speed up convergence by focusing on the information the algorithm actually uses. However, this optimization should be done carefully, as it can reinforce specific strategies. To explore a wide range of possible strategies, it is best to start with more general input features, even though this requires more computational resources. Once a suitable strategy is identified, the evolution process can be optimized by fine-tuning the input features based on what the agent actually uses.

Several promising avenues for future research emerged from our study. One area of interest is how altering the fitness function or introducing environmental constraints can guide the evolution process to discover a broader range of behaviors. Additionally, further investigations could focus on experimenting with adaptive time limits and diagonal vision to improve strategy development. Finally, exploring how the game-playing agent learns could test the hypothesis that the frequency of mistakes in randomized trials is proportional to the size of fitness improvement once the error is fixed.

A Team Contributions

Initially, both team members worked on understanding the baseline model. Jasper developed the first version of the report, performed the literature review, and gave the flash talk. Daria worked on the code and a (not-included) first version of an educational interface. About halfway through the project, they switched roles: Jasper finished the code, and Daria focused on writing.

B Future Directions

Although not directly inspired by our results, these are ideas we found in the literature or came up ourselves that could make interesting extensions of the project

Direct Competition

Introducing competition among snakes for fruits could add a layer of complexity, mimicking natural selection processes. Options include non-colliding snakes or snakes that must avoid each other, similar to slither.io, adding a dynamic competitive element.

Weight Agnostic Neural Networks

Exploring the use of Weight Agnostic Neural Networks (WANNs) to play Snake would be novel. WANNs evolve the topology of a neural network without adjusting the weights [14]. The objective is to create a network that can still perform tasks, like playing Snake, with weights drawn from a random distribution. This is akin to how animals can instinctively perform certain tasks at birth without fine-tuning their neural connections. This project could offer insight into how network topology alone can influence performance.

Deep NeuroEvolution

Exploring a more biologically inspired approach could involve using a neural network architecture that mimics natural sensory inputs, such as a Convolutional Neural Network (CNN) acting as the 'retina' for the snake. This method could autonomously learn to recognize complex features that we currently manually specify for the Multilayer Perceptron (MLP). Implementing this model would be technically demanding, likely requiring GPU support for effective training. A simplified version might be developed for smaller-scale applications, potentially manageable on a single desktop for games like a small-grid Snake.

Bilevel Optimization

Neural Architecture Search typically focuses on finding the best network topology and hyperparameters. Traditionally, this search is done without gradient information, relying on manual adjustments or scalable architectures like EfficientNet . Manual tuning is often limited by the complex interactions between parameters, covering only a small portion of the possible configurations. Implementing bilevel optimization could automate the tuning of NEAT's hyperparameters through an evolutionary process. This approach would require access to multiple CPUs since each optimization cycle would involve running a series of genetic algorithm simulations.

Reinforcement Learning

In the context of game-learning, reinforcement learning stands as the primary alternative to neuroevolution. Comparing our gradient-free neuroevolution approach with reinforcement learning, which typically employs gradients, could reveal differences in performance and efficiency, especially on a variety of tasks beyond the Snake game. This comparison could draw inspiration from research on Atari games [2] and existing reinforcement learning implementations for the Snake game available on GitHub.

References

[1] E. Galván and P. Mooney, "Neuroevolution in deep neural networks: Current trends and future challenges," *IEEE Transactions on Artificial Intelligence*, vol. 2, no. 6, pp. 476–493, 2021.

[2] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[3] Z. B. Zabinsky *et al.*, "Random search algorithms," *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.

[4] Talendar, *Neuroevolutionary_ snake*, https://github.com/Talendar/neuroevolutionary_snake, Accessed: 2024-06-27, 2024.

[5] toshNaik, *Neuroevolutionsnake*, https://github.com/toshNaik/NeuroevolutionSnake, Accessed: 2024-06-27, 2024.

[6] Somnef, *Snake_ neat_ ai*, https://github.com/Somnef/snake_neat_ai, Accessed: 2024-06-27, 2024.

[7] R. Miikkulainen, J. Liang, E. Meyerson, *et al.*, "Evolving deep neural networks," in *Artificial intelligence in the age of neural networks and brain computing*, Elsevier, 2024, pp. 269–287.

[8] K. Vignesh Kumar, R. Sourav, C. Shunmuga Velayutham, and V. Balasubramanian, "Fitness function design for neuroevolution in goal-finding game environments," in *Advances in Computational Collective Intelligence: 12th International Conference, ICCCI 2020, Da Nang, Vietnam, November 30–December 3, 2020, Proceedings 12*, Springer, 2020, pp. 503–515.

[9] P. Shinners, *Pygame*, http://pygame.org/, 2011.

[10] A. McIntyre, M. Kallada, C. G. Miguel, C. Feher de Silva, and M. L. Netto, *neat-python*.

[11] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[12] E. Papavasileiou, J. Cornelis, and B. Jansen, "A systematic literature review of the successors of "neuroevolution of augmenting topologies"," *Evolutionary computation*, vol. 29, no. 1, pp. 1–73, 2021.

[13] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evolutionary intelligence*, vol. 1, pp. 47–62, 2008.

[14] A. Gaier and D. Ha, "Weight agnostic neural networks," *Advances in neural information processing systems*, vol. 32, 2019.