SOFT COMPUTING PROJECT

Topic: The Influence of Genetic Algorithms on Learning Possibilities of Artificial Neural Networks

Presented By:

Harshit Kumar More (16)

Kunal Sati (25)

INTRODUCTION

1.1 Neural Networks

Neural networks are made of interconnected neurons arranged in input, hidden, and output layers. Their behavior depends on the network's topology and the weights on the connections, which are adjusted during learning. Training usually starts with random weights and can be supervised, reinforcement-based, or unsupervised depending on the available feedback. Neurons update their states either synchronously or asynchronously and may use uniform or mixed activation functions. While multilayer networks enable deep learning, choosing the right topology is challenging—too few neurons limit learning capability, while too many lead to overfitting and poor generalization.

1.2 Optimization Algorithms

Evolutionary algorithms imitate natural evolution to solve problems by evolving a population of candidate solutions over generations. Instead of adjusting parameters through error signals, they use selection, crossover, and mutation to gradually improve fitness, making them suitable for complex tasks without clear target outputs. The process begins with a random population, evaluates fitness, selects strong individuals as parents, and produces new solutions through genetic operations. This cycle continues until a satisfactory solution emerges. Since random starting points can cause the algorithm to converge too early on poor solutions, using multiple populations helps reduce this risk and improves exploration.

1.3 Natural Selection & Genetic Algorithms

Genetic algorithms mimic natural evolution by treating each candidate solution as a chromosome-like binary string. A population of these individuals evolves over time, with fitter solutions having a higher chance of being selected while still preserving diversity. New offspring are created through crossover between selected parents, and mutation introduces small random changes to prevent stagnation. The algorithm repeatedly performs selection, crossover, mutation, and evaluation until it converges on a strong solution.

2. Related Works

Genetic algorithms (GAs) have been widely studied as a method for training neural networks, including in challenging domains like Atari games in the Arcade Learning Environment. Research often compares GAs with reinforcement learning approaches such as Deep-Q Learning and actor-critic models, with results showing that GAs can be surprisingly competitive and sometimes even faster or more effective. Beyond gaming, GAs are used to train or optimize neural networks in fields such as climate science, groundwater monitoring, medical diagnosis, bioinformatics, airline scheduling, robotics, and CAD optimization. They are also used to evolve neural network architectures, including CNN designs. Overall, combining GAs with neural networks forms a powerful hybrid method that frequently outperforms using either technique alone.

3. Demonstration of Neural Network Learning Using Genetic Algorithms

The SnakeAI project by Greer Viau, written in Processing and released under the MIT license [35], provides a simple visual representation of a neural network's structure and weights. The full source code is available on GitHub. For our work, we adapted the idea but rebuilt both the neural network and

the learning process from scratch, without using frameworks. A simplified process diagram is shown in Figure 1.

3.1. Game Snake as the Brain of a Neural Network

Snake is a classic grid-based game where the snake grows and the score increases whenever food is eaten, but the game ends if it collides with a wall or itself. In the SnakeAI version, the game runs on a 40×40 grid, with the snake starting at length 1 and a score of 0. The purpose is to show how a neural network controls the snake's movement.

Each snake is driven by a multilayer neural network with 24 input neurons and four output neurons (one for each direction). One or more fully connected hidden layers sit between them; the original setup uses two hidden layers of 16 neurons each, but the architecture is configurable. During play, input neurons visualize incoming data through color changes, and the output neuron with the highest activation—shown in green—selects the movement direction. Connection weights are also visualized, with red representing negative weights and blue representing positive ones.

3.2 Sensors – Input

The network receives inputs representing the snake's "vision." It measures the distance, in all eight directions, to three types of objects: walls, its own body, and food. Diagonal distances are treated like horizontal and vertical ones. This gives **24 input values** (3 objects × 8 directions).

3.3 Fitness

While the game score is simply the amount of food eaten, evolution requires a more detailed fitness measure. Each step the snake survives gives **+1 fitness**, and each food item gives **+100**. To prevent snakes from endlessly looping and gaining fitness without improving, a snake can take at most **200 steps** unless it eats food. Each time it eats, it receives an additional **100 steps**, up to a limit of **500 total steps**.

3.4 Evolution

The original SnakeAI evolves a population of 2000 randomly initialized neural-network snakes. Each snake senses distances, processes them through its hidden layers, and moves in the direction chosen by the strongest output neuron. Snakes gain points by eating food and die on collisions; a generation ends when all snakes have died. Fitness is calculated, the best network is carried over unchanged, and the rest of the population is produced through crossover among the top performers, followed by mutating 5% of all weights. The default architecture uses two hidden layers of 16 neurons with a mutation rate of 0.05.

To support research, the SnakeAI.pde file was extended without modifying core classes. The enhanced version measures performance across adjustable numbers of generations, individuals, and evolutions. It logs results for each generation, saves the best network from every evolution, and produces a final summary with scores and execution times. Parameters like generation count, population size, and number of evolutions (e.g., 50, 2000, 20) can be configured, and all results are saved to disk.

3.5.2 Other Algorithm Modifications

A key update was adding an option to disable game rendering using:

Boolean showAnimation = false;

Turning off animation greatly speeds up training, since the program no longer displays the best snake after each generation. Visualization can be done later by loading the saved network of the best individual.

After training, the program can also show the full NN topology, weight colors, input changes, and the active output neuron. All additions keep the program fully compatible—setting showAnimation = true restores the original behavior.

With these changes, a single environment can handle both computation and visualization, making it easier to compare trained individuals from different NN topologies by score or strategy.

3.6 Neural Network Topology

The neural network's structure is defined by two parameters:

int hidden_nodes = 16;

int hidden_layers = 2;

ReLU was chosen as the activation function.

To compare performance, 20 different topologies were tested:

- **1 hidden layer:** 1×4, 1×8, 1×16, 1×24, 1×32, 1×48
- **2 hidden layers:** 2×4, 2×8, 2×16, 2×24, 2×32, 2×48
- **3 hidden layers:** 3×4, 3×8, 3×16, 3×24, 3×32, 3×48
- **4 hidden layers:** 4×8, 4×16

3.7 Evolutionary Algorithm Changes

To evaluate evolutionary behavior, the program was modified to allow different population sizes and mutation levels. Fitness rules and game logic were kept unchanged for compatibility.

Key parameters:

float mutationRate = 0.05;

int numOfSnakes = 2000;

Tests were run with populations of **500, 2000, and 4000** snakes, and mutation levels of **5%, 10%, 15%, 20%, and 50%**

3.8 Test Environment

SnakeAI runs in the Processing IDE. All initial tests used Windows 10 (64-bit), Processing 3.5.4, on identical laptops with an **i7-8550U CPU**. For example, running 60 evolutions of 50 generations (topologies 2×32 to 2×4, 5% mutation) required **98 hours** across three machines.

To scale up, a virtual server environment was created using Alpine Linux 3.9.3, XFCE, Processing 3.5.4, VMware ESX 6.5, and OpenJDK 1.8. Each VM ran with **4 CPUs at 3.4 GHz**. All physical hosts used identical CPUs to ensure consistent timing.

In total, more than **4800 evolutions** were executed across **80 VM runs**, taking over **1620 hours** (67 days). All results were saved to a central repository and exported for statistical analysis using JASP.

4. Analysis of Results

The goal of this study was to determine whether changes to the genetic algorithm improve the speed and effectiveness of neural network training. To evaluate this, we tested multiple network topologies and evolutionary parameter settings, comparing their performance—both improvements and declines—against a reference architecture.

The reference setup used the original topology of **two hidden layers with 16 neurons each**, a **population of 2000 individuals**, and a **5% mutation rate**. Unless stated otherwise, every tested topology and GA parameter combination was evaluated through **60 evolutions**, each consisting of **50 generations** with **2000 individuals per generation**, matching the reference conditions. All recorded results are available at:

"https://github.com/josefczyz/snakeAIstatistics (accessed 1 April 2022)"

Each dataset directory also includes a **Snakes** folder containing the saved neural network configurations of the best individuals from each evolution. These can be loaded back into the program to observe their behavior under new conditions. Because food placement in the game is random, a trained model may not always reproduce its original performance.

Every folder also includes an **EvolutionScores** directory, which stores the results for each evolution—showing the score of every generation and the total computation time required.

4.1 Comparison of Neural Network Topologies

4.1.1 Effect of the Number of Hidden Layers

Table 1 summarizes how different numbers of hidden layers (each with 16 neurons) perform at a 5% mutation rate.

The results show that **one hidden layer outperformed the original two-layer design**, while adding more layers (three or four) **reduced performance**. The improvement from using a single layer, however, came with a noticeable increase in computation time (from 11 to 17 hours).

4.1.2 Effect of the Number of Neurons per Layer

The next comparison looks at how the number of neurons in each hidden layer affects performance, still using two hidden layers as the reference design.

The results (Table 2 and Figures 3–5) show:

- Fewer than 16 neurons per layer → **worse performance**
- More than 16 neurons → **slight improvement**, mainly in maximum score
- Topologies **2×24**, **2×32**, and **2×48** perform almost the same

- Computation time **increases sharply** for 2×48, despite no real gain over 2×32

This means that increasing neurons beyond a certain point gives diminishing returns while significantly raising computation cost.

Table 1. Achieved score according to the number of layers, each of 16 neurons, mutation 5%.

Topology:	1 × 16	2 × 16	3 × 16	4 × 16
Max score:	142	111	123	98
Min score:	5	5	6	6
Average:	72	60	49	41
Median:	75.5	64.5	52	27
25th percentile:	51	41	23	11
75th percentile:	96	77	73	74
Total processing time [h]	17	11	11	4

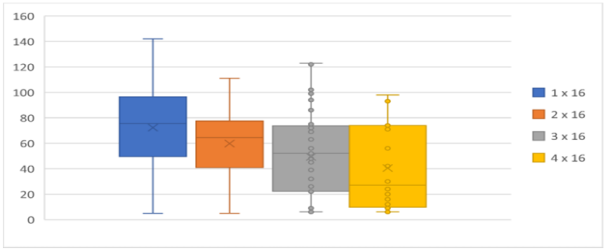

Figure 2. Achieved score according to the number of layers, mutation 5%.

Table 2. Achieved score according to the number of neurons, two hidden layers, mutation 5%.

Topology:	1 × 4	1 × 8	1 × 16	1 × 24	1 × 32	1 × 48	2 × 4	2 × 8	2 × 16	2 × 24	2 × 32	2 × 48	3 × 4	3 × 8	3 × 16	3 × 24	3 × 32
Max score:	99	117	142	130	128	140	92	96	111	123	125	120	92	114	123	113	109
Min score:	6	7	5	6	6	6	6	5	5	5	5	6	5	5	6	5	6
Average:	54	60	72	82	76	87	33	45	60	65	67	68	24	41	49	52	57
Median	53	63	75.5	81	80	91.5	22.5	42	64.5	69.5	75	75	13.5	31	52	46	65
25th percentile	33	35	51	71	59	77	8	24	41	46	46	47	7	16	23	27	33
75th percentile	75	82	96	97	98	101	57	66	77	86	91	94	32	65	73	76	82
Time [h]	9	13	17	16	25	30	6	8	11	14	16	29	4	8	11	12	14

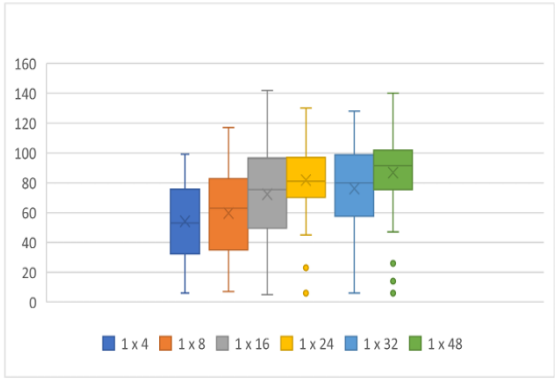

Figure 4. Achieved score according to the number of neurons, one hidden layer, mutation 5%.

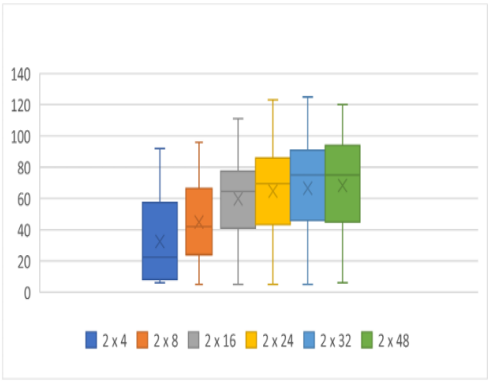

Figure 3. Achieved score according to the number of neurons, two hidden layers, mutation 5%.

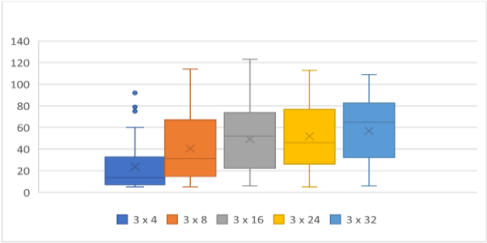

Figure 5. Achieved score according to the number of neurons, two hidden layers, mutation 5%.

Table 3. Kruskal–Wallis Test for data distribution normality.

Factor	Statistic	df	p
Topology	14.621	2	<0.001

Table 4. Dunn Post Hoc comparison test–Topologies.

Comparison	z	W_i	W_j	p	p_{bonf}	p_{holm}
1 × 16–2 × 16	2.064	109.150	89.525	0.020	0.059	0.039
1 × 16–3 × 16	3.820	109.150	72.825	<0.001	<0.001	<0.001
2 × 16–3 × 16	1.756	89.525	72.825	0.040	0.119	0.040

Networks with a single hidden layer consistently outperformed those with two layers across all tested neuron counts (4–48), with the 1×16 topology giving the best overall result. Performance dropped when using three hidden layers, and computation times decreased accordingly, matching the lower scores.

To examine the impact of layer depth more formally, the 1×16, 2×16, and 3×16 networks were compared using ANOVA. Because the data were non-normal (Kruskal–Wallis, $p < 0.001$), Dunn's post hoc test was used. Most topology pairs showed significant differences ($p < 0.05$ using Holm and Bonferroni corrections). Bonferroni found no significant difference for 1×16 vs. 2×16 or 2×16 vs. 3×16, but still confirmed a strong difference between 1×16 and 3×16.

The results also show that higher-performing individuals take longer to compute. This is illustrated in Figure 6 for the 2×16 topology, where 60 evolutions of 50 generations were run. Individuals achieving high scores early are highlighted in red, green, and yellow.

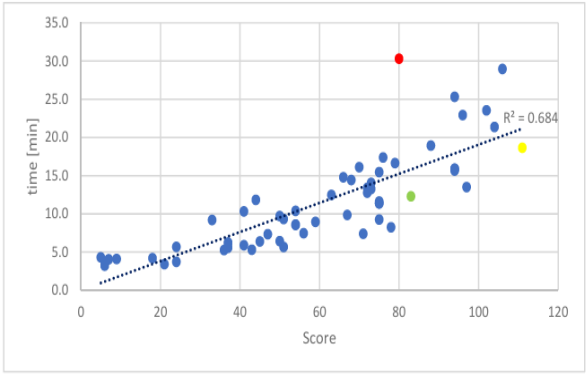

Figure 6. Time required to compute evolutions according to the achieved score, 2 × 16, mutation 5%. Individuals that are able to achieve high scores in early generations are marked in red, green and yellow.

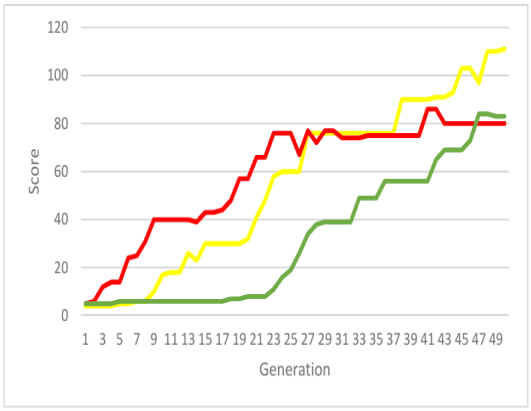

Figure 7. Evolution of selected measurements in generations from Figure 6.

The runtime of each evolution increased linearly with the score achieved—higher-scoring individuals required longer simulations. Figure 7 shows that total runtime was also influenced by when strong individuals appeared: early high performers substantially increased computation time. For example, an evolution that reached a score of 40 by generation 9 took over twice as long as one that reached the same score at generation 29, despite ending with nearly identical final scores. Another evolution that peaked at 111 required only about two-thirds the runtime of the slower, less successful one.

When comparing single-layer topologies, performance clearly improved when increasing neurons from 4 to 16, but adding more neurons beyond 16 (e.g., 24 or 32) did not yield significant statistical gains.

.

4.2. Comparison of Evolutionary and Genetic Algorithm Changes

Several parameters directly affect the evolutionary algorithm: the mutation rate, the number of individuals per generation, and the number of generations before stopping and computing the final NN configuration. Other GA components—such as how fitness is calculated or how parents are selected for crossover—can also influence performance.

4.2.1. Effect of Population Size

We first examined how the number of individuals in each generation impacts the final score. Table 5 compares results from populations of 500 and 4000 individuals against the reference topology (2×16) using 2000 individuals and a 5% mutation rate.

A larger population size improved the average, median, and percentile scores, and most of the best individuals performed better. However, this gain came with a proportional increase in computation time, as shown in Figures 8 and 9.

Table 5. Achieved score according to the number of individuals in the generation, topology 2 × 16, mutation 5%.

Topology:	2 × 16	2 × 16	2 × 16
Individuals in a generation	500	2000	4000
Max score:	75	111	149
Min score:	4	5	8
Average:	28	60	76
Median:	24.5	64.5	78
25th percentile:	7	41	64
75th percentile:	42	77	94
Total processing time [h]	2	11	24

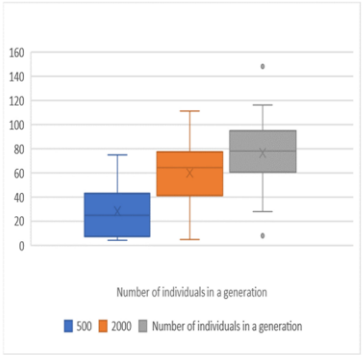

Figure 8. Achieved score according to the number of individuals in the generation, topology 2 × 16, mutation 5%.

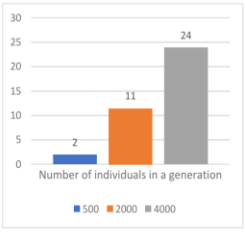

Figure 9. Time in hours necessary to compute 60 evolutions based on the number of individuals in the generation, topology 2 × 16, mutation 5%.

We then examined how population size affects topologies with different hidden-layer neuron counts. For a single hidden layer, the results with a 10% mutation rate are summarized in Table 6.

Table 6. Influence of the number of individuals on the achieved score in topologies with one hidden layer, mutation 10%.

Topology:	1 × 4	1 × 4	1 × 4	1 × 16	1 × 16	1 × 16	1 × 32	1 × 32	1 × 32
Number of individuals:	500	2000	4000	500	2000	4000	500	2000	4000
Max score:	90	149	136	106	147	147	125	147	152
Min score:	5	5	8	5	6	25	5	5	37
Average:	34	58	80	49	90	96	49	91	104
Median:	27	53.5	85	48	90.5	97	51	93	102.5
25th percentile:	8	31	70	20	76	83	17	79	88
75th percentile:	55	80	96	75	101	107	79	107	122
Total processing time [h]	2	9	17	3	18	32	6	21	54

Increasing the number of individuals per generation generally improved single-evolution results, with top performers doing better than in smaller populations (Figure 10). However, even with 4000 individuals, the best score did not improve significantly—the 1×32 topology was only about 3% better

than the next best. Training time scaled directly with population size (Figures 11 and 12), meaning these small performance gains came at much higher computational cost.

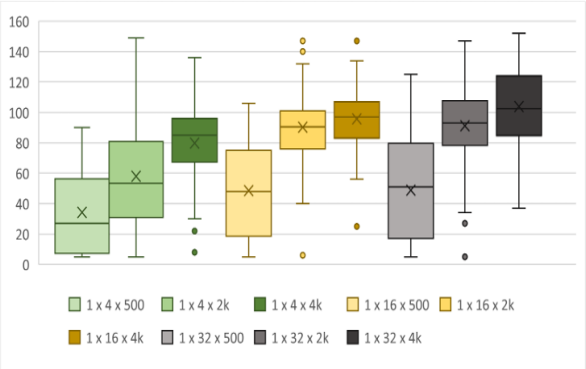

Figure 10. Influence of the number of individuals on the achieved score in topologies with one hidden layer, mutation 10%.

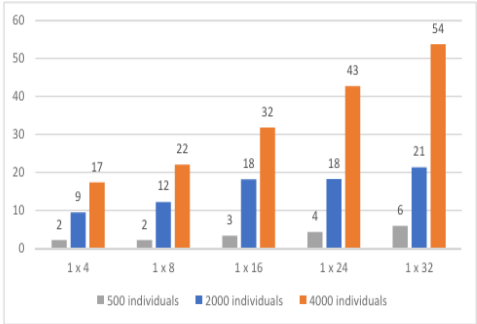

Figure 11. Time in hours necessary for the computation based on the number of individuals, one hidden layer.

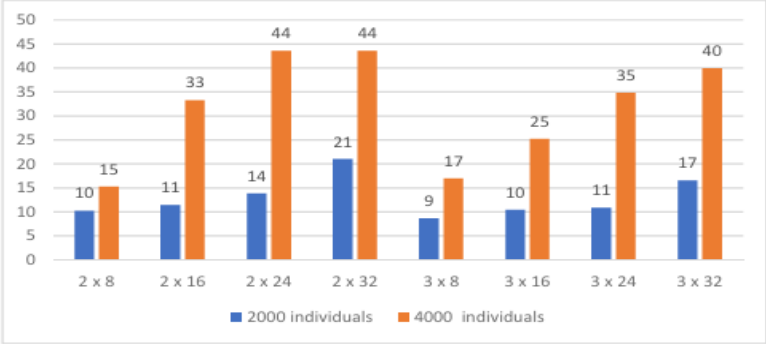

Figure 12. Time in hours necessary for the computation based on the number of individuals, two and three hidden layers.

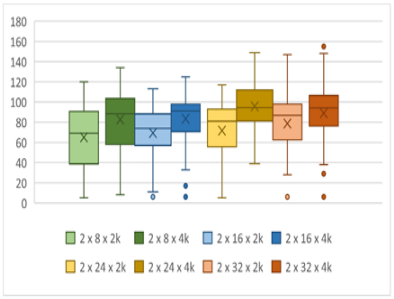

Figure 13. Influence of the number of individuals on the achieved score in a topology with two hidden layers, mutation 10%.

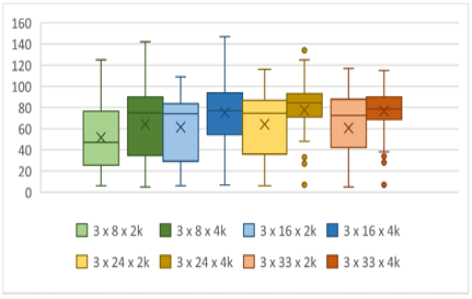

Figure 14. Influence of the number of individuals on the achieved score in a topology with three hidden layers, mutation 10%.

The same pattern appeared in topologies with two or three hidden layers: increasing the number of individuals also raised performance, but again with a 10% mutation rate the improvements came alongside higher computation times (Figures 13 and 14).

4.2.2. Influence of the Mutation Coefficient

Mutation rate is another GA parameter that affects performance, as it defines what portion of neurons receive random weight changes when new individuals are generated. Table 7 compares how different mutation rates impact the 2×16 and 1×16 reference topologies.

Table 7. Influence of mutation on the achieved results, topologies 1 × 16 and 2 × 16.

Topology:	1 × 16	1 × 16	1 × 16	1 × 16	1 × 16	2 × 16	2 × 16	2 × 16	2 × 16	2 × 16	2 × 16
Mutation:	**5%**	**10%**	**15%**	**20%**	**25%**	**5%**	**10%**	**15%**	**20%**	**25%**	**50%**
Max score:	142	147	150	150	156	111	113	122	121	137	114
Min score:	5	6	6	6	6	5	6	6	5	6	7
Average:	72	90	88	93	91	60	69	73	75	75	62
Median:	75.5	90.5	89	96	94	64.5	74	83.5	82.5	82.5	69
25th percentile:	51	76	76	89	77	41	57	58	70	62	41
75th percentile:	96	101	98	105	105	77	87	94	91	94	81
Total processing time [h]	17	18	16	19	17	11	11	13	13	12	7

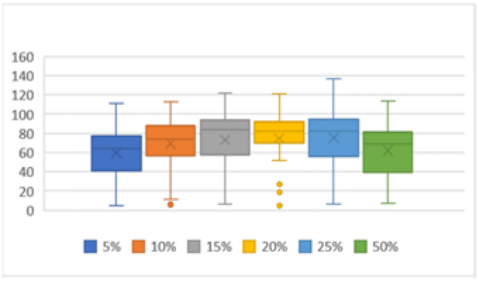

Figure 15. Influence of the 2 × 16 topology.

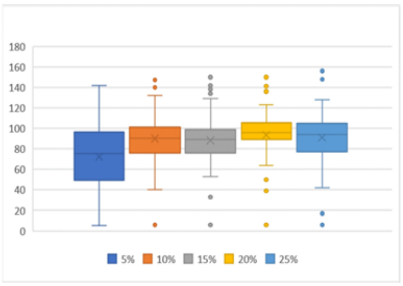

Figure 16. Influence of mutation on the achieved results, 1 × 16 topology.

Mutation rates between 15% and 25% produced almost identical performance, with average and median scores differing by less than two. The best overall score occurred at a 25% mutation rate, which was also the fastest to compute. The weakest results came from the extreme rates—5% and 50%—though these two produced nearly the same scores, with the 50% rate running in roughly half the time of the 5% rate.

Table 6 shows that increasing mutation rate and population size both improve results, but only population size has a major impact on runtime. Mutation rate changes have minimal effect on computation time. Additional interactions between mutation and network size for two-layer topologies are shown in Table 8.

Figure 17 demonstrates that a 10% mutation rate outperforms 5% across several architectures, improving both best and median scores by over 20% while keeping runtime nearly unchanged—except for the 2×32 topology, which required about 25% more processing time.

Table 8. Achieved score based on a mutation in topologies with two hidden layers.

Topology:	2 × 4	2 × 4	2 × 8	2 × 8	2 × 16	2 × 16
Mutation:	**5%**	**10%**	**5%**	**10%**	**5%**	**10%**
Max score:	92	119	96	120	111	113
Min score:	6	5	5	5	5	6
Average:	33	41	45	65	60	69
Median:	22.5	31.5	42	69	64.5	74
25th percentile:	8	13	24	40	41	57
75th percentile:	57	65	66	90	77	87
Total processing time [h]	6	5	8	10	11	11
Topology:	**2 × 24**	**2 × 24**	**2 × 32**	**2 × 32**	**2 × 48**	**2 × 48**
Mutation:	**5%**	**10%**	**5%**	**10%**	**5%**	**10%**
Max score:	123	117	125	147	120	147
Min score:	5	5	5	6	6	7
Average:	65	72	67	79	68	80
Median:	69.5	81	75	87	75	86.5
25th percentile:	46	57	46	66	47	66
75th percentile:	86	93	91	98	94	98
Total processing time [h]	14	14	16	21	29	28

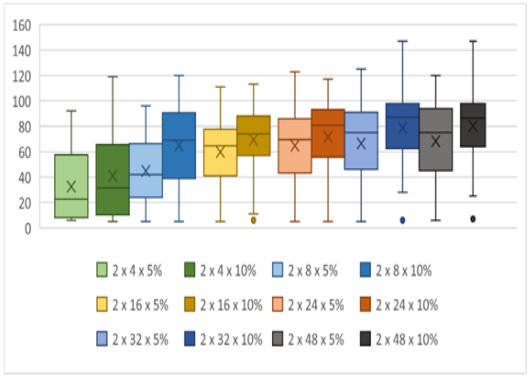

Figure 17. Achieved score based on a mutation in topologies with two hidden layers.

figure 18 shows that mutation rate strongly affects performance. With low mutation rates (≤5%), the population often got stuck in local optima and improved only slowly. Very high mutation rates (>25%) introduced too much randomness, making results heavily luck-dependent—though a large population

can offset this. The best overall score (156) was achieved with the 1×16 topology at a 25% mutation rate.

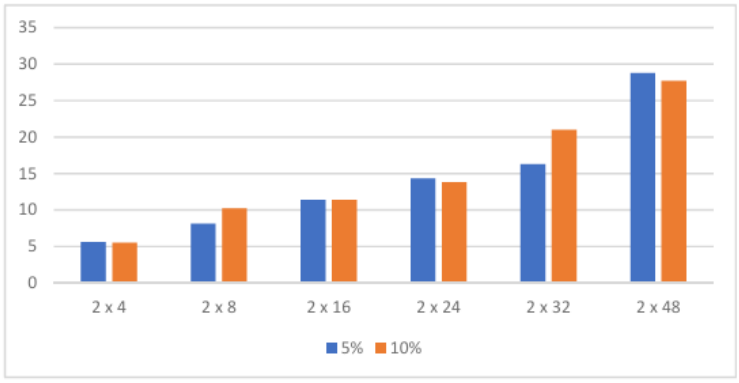

Figure 18. Computation time of topologies with two hidden layers based on the mutation level.

4.2.3. Influence of the Number of Generations

Although running more generations usually leads to better final performance, it also takes more time. To compare how quickly different models learn, we looked at their performance specifically at 50 generations. Earlier results (Table 7) suggested that deeper networks perform worse at this point, but this doesn't mean they are ineffective overall. Table 9 and Figure 18 show that networks with more hidden layers just need more generations to fully develop their performance—though this improvement comes with increased computation time.

Table 9. Influence of the number of generations on the final score.

Topology:	**4 × 8**	**4 × 8**	**1 × 16**	**1 × 16**	**1 × 24**	**1 × 24**
Mutation:	**25%**	**25%**	**10%**	**10%**	**20%**	**20%**
Number of generations:	50	200	50	200	50	200
Number of evolutions:	20	10	60	20	60	10
Max score:	77	152	147	175	133	150
Min score:	5	29	6	104	6	95
Average:	40	105	90	146	92	121
Median:	40	104.5	90.5	149	95	117.5
25th percentile:	12	95	76	147	89	106
75th percentile:	63	141	101	151	101	133
Total processing time [h]	1	12	18	98	18	39

Increasing the number of generations allows complex NN topologies to eventually reach scores similar to simpler ones. Simple networks (e.g., 1×4) learn quickly but hit a ceiling and fail to generalize—when tested in new conditions, they often make basic mistakes, such as missing food placed near a wall. In contrast, deeper or larger networks are more robust.

However, raising the generation count greatly increases runtime. Moving from 50 to 200 generations (4× more) resulted in roughly 16× longer computation time. Therefore, the number of generations must be chosen carefully so the GA remains practical.

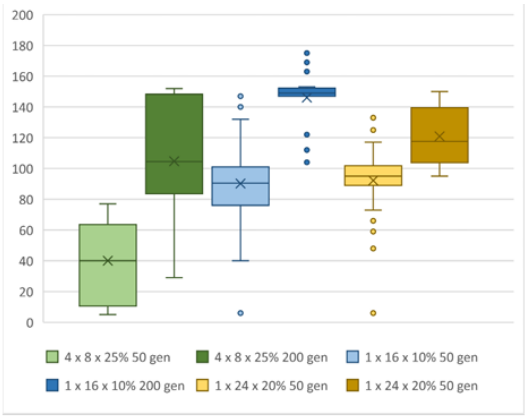

Figure 19. Influence of the number of generations in the evolution of the final score.

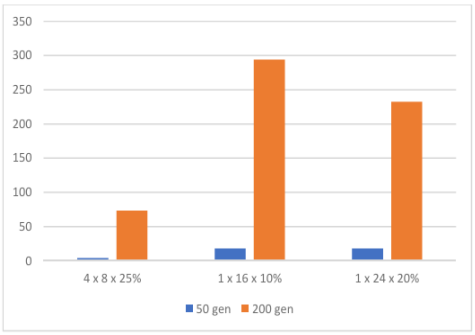

Figure 20. Time in minutes necessary to train an NN based on the number of generations in the evolution.

Because the game is highly stochastic—random food placement and random initial populations—each evolution behaves differently. Across 60 evolutions per topology, almost every topology had at least one run that failed to exceed a score of 30 even after 50 generations. Yet the same topologies could also produce surprisingly strong individuals: for example, the simple 1×4 network reached scores of 99 with 5% mutation and even 149 with 10%, all within under 9 hours of computation. This suggests a practical strategy: run fewer generations but more evolutions, relying on chance to produce a good initial population.

Best Achieved Results

No single topology or GA configuration consistently dominated. Even the smallest topology (1×4) produced a top-ten score of 149, but it generalized poorly and made more mistakes under new conditions. In contrast, complex networks required more generations to learn but generalized better and made fewer errors. Visual inspection showed that simple topologies took the most direct path to food, whereas deeper networks used the board more intelligently—an advantage when the snake grows to around 160 units, a practical performance limit for all tested configurations.

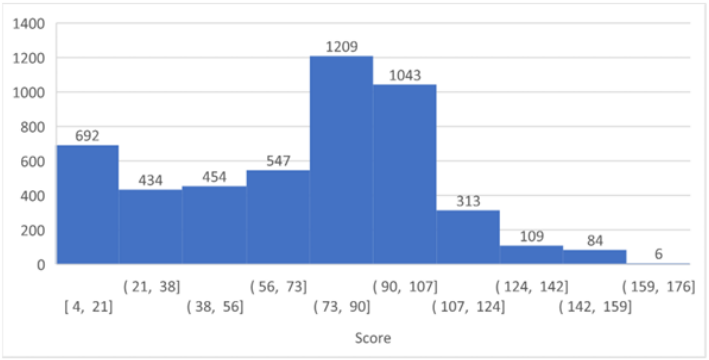

Figure 21. Frequency of the achieved results of evolutions of all topologies.

Topologies with more hidden layers showed **less variation** between their best and worst evolutions—but **only when trained for enough generations** (≈400 for four layers, ≈600 for three).

The **highest score overall (176)** came from the **1×24 topology** with **20% mutation** and **200 generations**. Even its weakest evolution still reached **105**, showing strong consistency. A different configuration—**4×8 with 25% mutation**—had a slightly lower top score, but all five of its

tested evolutions scored **138+**, matching the average of the best topology. This shows it was **stable regardless of the random initial weights**.

Figures 21–23 (not shown here) indicated that both configurations required similarly demanding computation, and they were among the **top six best-performing setups** across all experiments.

Table 10. Topologies with individuals that achieved the top-six best scores.

Topology:	1 × 24 × 20%	1 × 16 × 10%	4 × 8 × 25%	1 × 32 × 15%	2 × 32 × 10%	3 × 24 × 20%
Number of generations:	200	200	400	50	50	600
Number of individuals	2000	2000	2000	2000	4000	2000
Number of evolutions:	10	20	5	60	60	5
Max score:	176	175	159	156	155	151
Min score:	105	104	138	5	6	129
Average:	138	146	150	96	89	145
Median:	141	149	149	95.5	94	149
25th percentile:	125	147	149	85	77	147
75th percentile:	149	151	153	111	106	150
Total processing time [h]	39	98	20	27	44	74
Processing time of evolution [h]	3.9	4.9	4.0	0.5	0.7	14.7

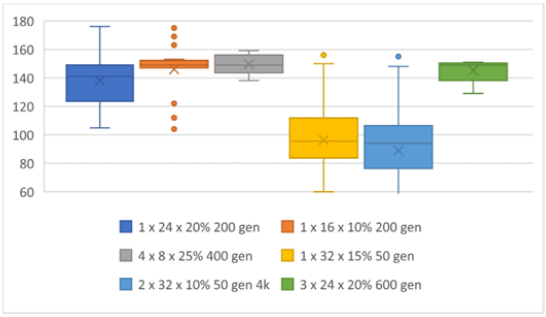

Figure 22. Topologies with individuals that achieved the top-six best scores.

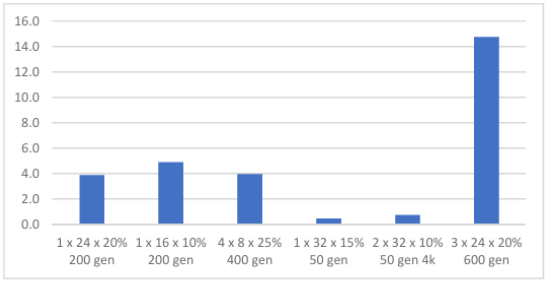

Figure 23. Time in hours necessary to train one evolution, top score.

Table 11. Best achieved results across topologies.

File Name	Topology		GA Parameters			Score	Time
	H	N	Mut	Gen	Individ.		min
T1×24m20G200R6bestSnake_176.csv	1	24	20	200	2000	176	210
T1×16m10G200R12bestSnake_175.csv	1	16	10	200	2000	175	353
T4×8m25G400R1bestSnake_159.csv	4	8	25	400	2000	159	252
T1×32m15G50R47bestSnake_156.csv	1	32	15	50	2000	156	42
T1×16m25G50R28bestSnake_156.csv	1	16	25	50	2000	156	28
T2×32m10G50R35bestSnake4000_155.csv	2	32	10	50	4000	155	96
T4×8m25G200R1bestSnake_152.csv	4	8	25	200	2000	152	109
T1×32m10G50R14bestSnake4000_152.csv	1	32	10	50	4000	152	73
T3×24m20G600R5bestSnake_151.csv	3	24	20	600	2000	151	914
T1×4m10G50R16bestSnake_149.csv	1	4	10	50	2000	149	28
T2×16m15G50R43bestSnake4000_149.csv	2	16	15	50	4000	149	41
T2×24m10G50R9bestSnake4000_149.csv	2	24	10	50	4000	149	60
T3×48m10G50R5bestSnake4000_147.csv	3	48	10	50	4000	147	107
T3×16m10G50R46bestSnake4000_147.csv	3	16	10	50	4000	147	58

5. Evaluation of the Analysis

5.1 Neural Network Learning Ability

The results show that neural networks can successfully learn to play Snake using an evolutionary algorithm. Because the game supports many effective strategies, there isn't a single optimal solution—different networks can learn different but successful behaviors. Using evolution also removes the need for large, human-generated training datasets. Unlike supervised learning, which would force the model

to mimic human gameplay, evolution only requires a score and can discover strategies humans might not think of.

All network structures tested were able to learn the game, but smaller models had trouble generalizing. Some performed well during training but failed when presented with new food placements or board setups. Larger networks, with more neurons, handled variation more reliably and gave stable results across multiple runs.

The main drawback is training speed. Deeper networks needed many more generations to achieve strong performance, making them slower to evolve. However, once trained, they were more robust to randomness and consistently avoided bad local optima. Networks with three or four hidden layers repeatedly achieved high scores (above 100) in every long run.

5.2 Impact of Evolutionary Algorithm Adjustments

Changing evolutionary parameters—population size, number of generations, mutation rate—had a strong effect on both performance and training time.

Deeper networks required many generations to learn well, and increasing the generation count drove up computation time quadratically. Increasing the population size helped produce more diverse individuals and better scores but also increased the computation time linearly with population size.

Mutation rate influenced learning differently. It had almost no impact on runtime, but it strongly affected whether a population could escape local optima. Too little mutation caused stagnation, while too much mutation destroyed useful traits and stopped meaningful progress. Finding a moderate mutation rate was key to maintaining steady improvement without losing previously learned behavior.

6. Conclusions and Future Work

This research aimed to identify an NN topology that can be trained efficiently for the Snake game using a genetic algorithm. The results showed that **simpler networks learn faster** and can demonstrate the NN's ability to learn the task effectively. In contrast, **more complex topologies require significantly longer training** and cannot match the performance of simpler networks within limited training time.

The SnakeAI project proved to be a strong didactic example of how evolutionary algorithms can train NNs without a teacher. By systematically evaluating different NN structures and GA parameters, we identified combinations that achieve high scores with minimal computation time. Notably, **increasing the mutation rate from 5% to 20% improved population performance by ~20%**, while adding only ~10% more computation time.

Other parameter changes had stronger time trade-offs:

- **More individuals per generation** → linear increase in training time
- **More generations** → quadratic time increase

These were necessary to successfully train deeper networks. Experiments with new, unseen game inputs further confirmed that **more complex topologies generalize better**, whereas shallow networks with few neurons made more frequent mistakes.

We plan to extend this work by testing trained NNs on entirely new game datasets to determine which topologies generalize the best. Finally, since genetic algorithms show diminishing returns after a point, future improvements may require combining GAs with additional training methods to continue increasing performance.

References:

1. Nicholson, C. *Artificial Intelligence (AI) vs. Machine Learning vs. Deep Learning*. Available online: https://wiki.pathmind.com/ai-vs-machine-learning-vs-deep-learning (accessed on 1 April 2022).
2. Zelinka, I. *BEN–Technical Literature; Artificial Intelligence*; Prague, Czech Republic, 2003; ISBN 80-7300-068-7.
3. Fitzsimmons, M.; Kunze, H. Combining Hopfield neural networks with applications to grid-based mathematics puzzles. *Neural Netw.* **2019**, *118*, 81–89.
4. Tian, Y.; Zhang, X.; Wang, C.; Jin, Y. An evolutionary algorithm for large-scale sparse multiobjective optimization problems. *IEEE Trans. Evol. Comput.* **2019**, *24*, 380–393.
5. Zolpakar, N.A.; Yasak, M.F.; Pathak, S. A review: Use of evolutionary algorithm for optimisation of machining parameters. *Int. J. Adv. Manuf. Technol.* **2021**, *115*, 31–47.
6. Haldurai, L.; Madhubala, T.; Rajalakshmi, R. A study on genetic algorithm and its applications. *Int. J. Comput. Sci. Eng.* **2016**, *4*, 139.
7. OpenAI. *Gym*. Available online: https://gym.openai.com/docs/#available-environments (accessed on 25 February 2021).
8. Bellemare, M.G.; Naddaf, Y.; Veness, J.; Bowling, M. The Arcade Learning Environment: An Evaluation Platform for General Agents. *J. Artif. Intell. Res.* **2013**, *47*, 253–279.
9. Such, F.P.; Madhavan, V.; Conti, E.; Lehman, J.; Stanley, K.O.; Clune, J. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv* **2017**, arXiv:1712.06567.
10. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533.
11. Comi, M. *How to Teach AI to Play Games: Deep Reinforcement Learning*. Available online: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a (accessed on 27 February 2021).
12. Stanislawska, K.; Krawiec, K.; Kundzewicz, Z.W. Modeling global temperature changes with genetic programming. *Comput. Math. Appl.* **2012**, *64*, 3717–3728.
13. Fisher, J.C. *Optimization of Water-Level Monitoring Networks in the Eastern Snake River Plain Aquifer Using a Kriging-Based Genetic Algorithm Method*; Bureau of Reclamation and U.S. Department of Energy: Reston, VA, USA, 2013.
14. Fitzgerald, J.M.; Ryan, C.; Medernach, D.; Krawiec, K. An integrated approach to stage 1 breast cancer detection. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Madrid, Spain, 11–15 July 2015; pp. 1199–1206.
15. Wang, S.; Wang, Y.; Du, W.; Sun, F.; Wang, X.; Zhou, C.; Liang, Y. A multi-approaches-guided genetic algorithm with application to operon prediction. *Artif. Intell. Med.* **2007**, *41*, 151–159.
16. Gondro, C.; Kinghorn, B.P. A simple genetic algorithm for multiple sequence alignment. *Genet. Mol. Res.* **2007**, *6*, 964–982.

17. George, A.; Rajakumar, B.R.; Binu, D. Genetic algorithm based airlines booking terminal open/close decision system. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, Chennai, India, 3–5 August 2012; pp. 174–179.

18. Ellefsen, K.; Lepikson, H.; Albiez, J. Multiobjective coverage path planning: Enabling automated inspection of complex, real-world structures. *Appl. Soft Comput.* **2017**, *61*, 264–282.

19. Vidal, T.; Crainic, T.G.; Gendreau, M.; Lahrichi, N.; Rei, W. A Hybrid Genetic Algorithm for Multidepot and Periodic Vehicle Routing Problems. *Oper. Res.* **2012**, *60*, 611–624.

20. Chung, H.; Shin, K.-S. Genetic algorithm-optimized multi-channel convolutional neural network for stock market prediction. *Neural Comput. Appl.* **2019**, *32*, 7897–7914.

21. Li, Y.; Jia, M.; Han, X.; Bai, X.-S. Towards a comprehensive optimization of engine efficiency and emissions by coupling artificial neural network (ANN) with genetic algorithm (GA). *Energy* **2021**, *225*, 120331.

22. Hamdia, K.M.; Zhuang, X.; Rabczuk, T. An efficient optimization approach for designing machine learning models based on genetic algorithm. *Neural Comput. Appl.* **2020**, *33*, 1923–1933.

23. Flórez, C.A.C.; Rosário, J.M.; Amaya, D. Control structure for a car-like robot using artificial neural networks and genetic algorithms. *Neural Comput. Appl.* **2018**, *32*, 15771–15784.

24. Amini, F.; Hu, G. A two-layer feature selection method using Genetic Algorithm and Elastic Net. *Expert Syst. Appl.* **2020**, *166*, 114072.

25. Sun, Y.; Xue, B.; Zhang, M.; Yen, G.G.; Lv, J. Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification. *IEEE Trans. Cybern.* **2020**, *50*, 3840–3854.

26. Tian, Y.; Lu, C.; Zhang, X.; Tan, K.C.; Jin, Y. Solving large-scale multiobjective optimization problems with sparse optimal solutions via unsupervised neural networks. *IEEE Trans. Cybern.* **2020**, *51*, 3115–3128.

27. Abiodun, O.I.; Jantan, A.; Omolara, A.E.; Dada, K.V.; Mohamed, N.A.; Arshad, H. State-of-the-art in artificial neural network applications: A survey. *Heliyon* **2018**, *4*, e00938.

28. Slowik, A.; Kwasnicka, H. Evolutionary algorithms and their applications to engineering problems. *Neural Comput. Appl.* **2020**, *32*, 12363–12379.

29. Bacalhau, E.T.; Casacio, L.; de Azevedo, A.T. New hybrid genetic algorithms to solve dynamic berth allocation problem. *Expert Syst. Appl.* **2021**, *167*, 114198.

30. Escamilla-García, A.; Soto-Zarazúa, G.M.; Toledano-Ayala, M.; Rivas-Araiza, E.; Gastélum-Barrios, A. Applications of Artificial Neural Networks in Greenhouse Technology and Overview for Smart Agriculture Development. *Appl. Sci.* **2020**, *10*, 3835.

31. Liu, L.; Moayedi, H.; Rashid, A.S.A.; Rahman, S.S.A.; Nguyen, H. Optimizing an ANN model with genetic algorithm (GA) predicting load-settlement behaviours of eco-friendly raft-pile foundation (ERP) system. *Eng. Comput.* **2019**, *36*, 421–433.