NOVA IMS

MAY 2023

COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION PROJECT

IMAGE CLASSIFICATION NEURAL NETWORK OPTIMIZATION WITH GENETIC ALGORITHMS

GROUP 6:
FILIPE DIAS (R20181050)
INÊS SANTOS (R20191184)
MANUEL MARREIROS (R20191223)

Index

Introduction

For our project, we performed an image classification task with the help of a single-layer neural network, where we optimized the weights using genetic algorithms. The dataset we utilized was the CIFAR-10, a widely used computer vision dataset consisting of 60,000 32x32 colour images distributed into 10 mutually exclusive classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 testing images.

Division of Labour

Inês developed the genetic algorithm and most of the genetic operators, as well as the fitness function and neural network. Manuel implemented the Particle Swarm Optimization and other genetic operators. Filipe was responsible for the pre-processing of our data and the assessment of the different configurations, producing the plots to visualize the results. Finally, each member had equal contribution to the report.

Problem Representation

Our project's code can be found in the following GitHub repository: https://github.com/filipedias18/Project_CIFO.git. Since we were dealing with an image classification problem, we decided to develop our project through a *Jupyter Notebook* in the *Google Colab* environment, to be able to visualize and present results more clearly. Another advantage of this solution was the additional computational power it offered, a much-needed asset for this type of project. In this section, we define the main points of the problem.

1. Neural Network

Our neural network consists of a **single-layer neural network** created using the Keras Functional API. We opted for this very simple architecture because the focal point in this project was the optimization process, and it would have taken significantly longer to run the different configurations of our genetic algorithms if we had more complex networks, even if it generated better results.

1.1. Input Variables

The input shape of the neural network model is (100,), meaning each input image is represented as a one-dimensional array of **100 features**. Originally, we were dealing with 3072 (32*32*3) features for each image, where 32*32 corresponds to the number of pixels of each image (width and height, respectively), and 3 to the number of colour channels (R, G, B). However, this made our project much more computationally expensive, so we decided to reduce the number of input variables to 100 using PCA. Our decision to use 100 components is supported by this chart, where we can see the trade-off between the number of components and the explained variance. The dashed line represents the cut-off point where assuming more components would not compensate the computational overhead.

1.2. Output Variables

As stated before, there are **10 possible outputs**: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. These are obtained using the *Softmax* activation function, which ensures that the outputs represent the probability distribution over the 10 classes.

2. Genetic Algorithm

In our problem, each possible solution (chromosome) of the population passed to the genetic algorithm corresponds to a set of weights (genes) that will be passed to the single layer of the neural network. The weights passed to the NN have a shape of (100, 10), where 100 corresponds to the number of features, and 10 corresponds to the number of output classes. This means that each chromosome has a shape of (1000,), which results from flattening the weights matrix into a one-dimensional array.

2.1. Fitness Function

A good way to evaluate the fitness of each chromosome would be through the accuracy of the neural network, making this a **maximization problem**, where the higher the accuracy, the better the fitness. However, another way to measure the fitness could be through the cross-entropy loss function. In that case, we would have a **minimization problem**, where the smaller the loss, the better the performance of the model. Our fitness function can evaluate both parameters and will return the adequate value according to the type of problem the user defines. Throughout the project, however, we looked at the task at hand as a maximization problem. Additionally, we developed a second fitness function using the f1-score as a performance measure. However, it did not present any advantages when compared to the first fitness function, even taking longer to run, so we opted for the **accuracy** metric.

Development of the Genetic Algorithm

Before implementing the genetic algorithm, we first established a general function that takes as arguments the population of solutions, whether it is a maximization or minimization problem, the fitness function, the genetic operators, if there is elitism, the number of generations, the crossover and mutation rates, the size of the elite population (default is 0), and additional arguments that may be required. Thus, each variation of the genetic algorithm implemented in our project always follows the same logical steps, only differing in the parameters and genetic operators. Our GA evolution takes the following steps:

1. Initialization of the population
2. For each generation:
    - 2.1. Evaluation of each individual's fitness
    - 2.2. If we are using **elitism**, we define the **elitism population**, that is, the *n* members of the population with the best fitness (or worse, if we are dealing with a minimization problem), which will be kept in the next generation
    - 2.3. For each element of the population - *n*:
        - 2.3.1. **Selection:** Selecting two parents to produce two offspring for the next generation
        - 2.3.2. **Crossover:** Crossing the parents to produce new offspring, according to a given **crossover rate**
        - 2.3.3. **Mutation:** Mutating the offspring according to a given **mutation rate**
3. Add the offspring to the next generation
4. Register the best chromosome and its fitness
5. Calculate and return evaluation metrics

We established 9 genetic algorithms, each implementing a different combination of genetic operators.

Selection Algorithms

We implemented three different selection algorithms: **Tournament Selection**, **Roulette Wheel Selection**, and **Rank-Tournament Selection**, a combination of Rank-Based and Tournament selections that we developed. When using the traditional Rank-Based Selection, we were obtaining suboptimal results, leading us to believe that we were getting stuck in local optimums. By mixing this method with Tournament selection, we can get a better balance between exploitation and exploration by attributing a 50% chance of each algorithm to be picked at each selection stage.

At this stage, we decided that the selection algorithms should return two parents, rather than calling them twice to obtain the two parents separately. This way, we eliminate the possibility of receiving duplicate parents. This approach enhances the effectiveness of the crossover and mutation operators by introducing diverse genetic material from different parents.

Genetic Operators

Crossover Operators

For the crossover operators, we went with **single and double-point crossovers**, **uniform crossover**, and **arithmetic crossover**.

Mutation Operators

As for mutation operators, we implemented four methods: **gaussian mutation**, **gaussian mutation 2**, **swap mutation** and **inversion mutation**.

While it is not the point to exhaustively describe each operator, it is worth pointing out the difference between the normal gaussian mutation and gaussian mutation 2. The former mutates a single gene of the chromosome at random, while the latter iterates over each gene, mutating them in accordance with a given probability, leading to a bigger difference between the original chromosome and its mutated counterpart.

Elitism

Elitism is an important parameter when trying to increase fitness, since it allows us to preserve the best-performing individuals from one generation to the next. In our implementation, we have an elitism flag that allows to easily enable/disable it, as well as an *elite_size* variable that controls the number of individuals to be copied unchanged to the next generation. In all our models the elitism flag was set to True because we wanted to ensure that the top-performing individuals were not lost due to random exploration or excessive perturbations introduced by other genetic operators.

Configurations

As a consequence of the No Free Lunch Theorem, no formal method can exist to choose the best algorithm, or even the best algorithm configuration, to solve our problem. In that sense, we tried different configurations of functions and parameters to tackle the question at hand.

The table below provides a summary of the different combinations of selection methods and genetic operators that were tried. For each combination, a brief description of why the operators were chosen is provided. Unless stated otherwise, the **crossover rate is set to 0.8**, the **mutation rate is 0.08**, the **elite size is 3** and, if applicable, the **tournament size is 5**.

ID	Selection Method	Crossover Method	Mutation Method	Elitism
GA1	Tournament	Uniform	Gaussian	True
GA2	Roulette Wheel	Arithmetic	Swap	True
GA3	Tournament	Arithmetic	Swap	True
GA4	Tournament	Arithmetic	Inversion	True
GA5	Roulette Wheel	Single-Point	Gaussian	True
GA6	Roulette Wheel	Uniform	Gaussian	True

GA7	Tournament	Uniform	Gaussian 2	True
GA8	Rank-Tournament	Uniform	Gaussian 2	True
GA9	Rank-Tournament	Double-Point	Gaussian 2	True

GA1 consists of simple genetic operators that are not considered very computationally demanding, allowing for what can almost be considered a performance baseline for the models that follow. However, these operators involve a high degree of randomness in their processes, such as randomly selecting individuals for tournaments, randomly selecting genes for crossover, or introducing random perturbations in gene values through Gaussian mutation, meaning that this combination is very vulnerable to statistical variations and the model must be run multiple times to mitigate that.

GA2 uses a whole different set of genetic operators. With this combination, we have a selection mechanism that favours individuals with higher fitness, a crossover mechanism that explores intermediate solutions and increases diversity by introducing new values to the representation, compensating for the lower diversity of the selection method, and a mutation mechanism that introduces random changes.

In **GA3**, we utilized Tournament rather than Roulette Wheel selection to achieve a more balanced approach between exploitation and exploration. By adopting Tournament selection, diversity among the selected parents is ensured, potentially mitigating the issue of premature convergence that was observed in GA2. The crossover and mutation operators remain unchanged.

In **GA4** we kept the selection and crossover algorithms but switched to Inversion mutation. When compared to swap mutation, inversion mutation can introduce changes at a larger scale and may be useful for exploring different permutations or rearrangements of genetic material.

Moving on to **GA5**, we wanted to experiment a crossover method we hadn't implemented up until this point, single-point crossover. The other operators are the same as in GA4.

In **GA6** the selection and mutation methods remain the same, but the crossover method is changed back to Uniform, so that the resulting offspring can have a more balanced mixture of genetic material from both parents, with a higher potential for exploring different combinations of genes.

GA7 is almost the same as GA1, the one that had produced better results up until this point, but with an alternative version of the Gaussian Mutation where multiple genes at random are transformed. By doing this, we wanted to enhance the search space exploration.

GA8 is quite similar to GA7, with the main difference residing in the selection method. It is also worth pointing out that both the crossover and mutation rates were increased to 0.9 and 0.1 respectively, in order to hopefully increase exploration.

In **GA9**, we kept the higher crossover and mutation rates used in GA8, but the crossover is defined to double-point crossover.

As an addition to our project, we decided to implement **Particle Swarm Optimization**. Some advantages of applying PSO rather than a GA to this type of problem would be the former's tendency to **converge faster to optimal solution**, without getting stuck in a local optimum. Additionally, it is efficient, requiring **less parameter tuning**, and it is adequate for **continuous optimization**.

Performance Evaluation

To evaluate the various algorithms' performances, we ran them three times each. At the end of each run, we collected the results from four pre-established performance metrics. These results were then averaged.

The performance metrics utilized to evaluate the GAs were the following:

1. **Best Fitness:** The average of the best fitness registered in each iteration.
2. **Convergence Rate:** The average convergence rate of each iteration. This corresponds to the percentage of generations needed to reach a certain threshold. In our case, this threshold is 99% of the best fitness value registered in that run. A low convergence rate may indicate that the algorithm got stuck in a local optimum.
3. **Run Time:** The average amount of time it took for the algorithm to complete its course in each run.
4. **Average Improvement Rate:** The average value of how each generation has improved upon the previous one. This was then averaged among all iterations.

The main advantage of running each GA multiple times is, among several others, that we can assess their robustness, that is, if they behave consistently when encountering different initial conditions or random variations. Furthermore, this approach helps to reduce the variance associated with those random factors. Thus, by averaging our results, we can obtain a more stable and reliable estimate of each GA's performance. The final results are presented in the appendix and discussed below, with the understanding that some conclusions we take might lack in statistical robustness.

Discussion

Beginning with the **run time**, all GA configurations took around 10 minutes to run, while PSO, as expected, took almost half of that. For that reason, we "discarded" this factor when comparing the different GAs.

As for the **average best fitness**, the configurations that stood out were GA1, GA7, GA8, along with PSO, with GA1 leading the way at 0.2285. For GA1, the **average improvement rate** from one generation to the next was averaged at 0.039, which is also the best amongst all registered configurations. Not only that, but it presented a 100% convergence rate, meaning that it was continuously evolving well throughout the generations. As a matter of fact, only those four best algorithms presented a convergence rate above 90%, hinting at an efficient and effective search for good solutions on those models. Interestingly, all these implementations utilize some sort of tournament selection, meaning that the other configurations may be suffering from **premature convergence** due to the inherent selection bias of the Roulette Wheel method.

All things considered, GA1 was deemed to be the best combination of parameters and therefore we decided to run it again with 30 generations instead of 15. While it did improve, it started to converge around the 25th generation.

After analysing the results numerically, we decided to produce a line chart to visualize the evolution of the fitness values for each configuration throughout the generations. We also plotted several boxplots, one for each configuration of parameters, to assess the statistical relevance of the differences between results. The main conclusion that we took was that GA1 is undoubtfully better than all other configurations, as its lower quartile is above the upper quartile of the other boxplots, except GA4, GA7, GA8 and PSO, where more caution must be taken when doing the comparisons.

Limitations

A significant limitation that we faced was the lack of computational power. Ideally, we would have chosen a larger population size, preferably with 100 individuals, and a larger number of generations. Additionally, we would have liked to run each algorithm more than three times as well, to obtain more reliable and representative results.

Conclusions

By exploring the application of genetic algorithms for weight optimization in neural networks, this project allowed us to cement our knowledge in the field of optimization problems and genetic algorithms. Our goal was to enhance the performance of neural networks by finding the optimal set of weights that maximize accuracy, and we believe we were successful in doing that by leveraging the principles of GA's.

Through a series of experiments and evaluations, it was found that genetic algorithms can efficiently search through the weight space, converging towards better solutions over successive generations. This iterative process allows the neural network to continually adapt and improve its performance, ultimately leading to superior accuracy.

While genetic algorithms offer promising results, it is important to acknowledge that their performance heavily relies on the design of the genetic operators, such as selection, crossover, and mutation. One thing we found out is that careful parameter design, selection and balance are crucial to achieve optimal results. Additionally, the computational complexity of genetic algorithms should be taken into consideration, as they can be computationally expensive for large-scale neural networks.

In summary, genetic algorithms present a viable and effective approach for weight optimization in neural networks. Their ability to navigate complex weight spaces and uncover optimal solutions demonstrates their potential for enhancing the performance of neural networks in various domains. Future research should focus on refining the genetic operators and investigating the application of genetic algorithms in different types of neural network architectures.

Appendix

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (In Minutes)	Average Improvement Rate
GA1	1	0.2362	100	9.9	0.042
	2	0.2293	100	9.6	0.038
	3	0.2199	100	9.9	0.038
		0.2285	**100**	**9.8**	**0.039**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (In Minutes)	Average Improvement Rate
GA2	1	0.1513	73.33	9.8	0,008
	2	0.1358	46.66	9.8	0,005
	3	0.1644	66.66	9.9	0,012
		0.1505	**62.21**	**9.8**	**0,008**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA3	1	0.1560	100	10	0.015
	2	0.1661	86.66	10	0.019
	3	0.1616	60	10	0.013
		0.1612	**82.22**	**10**	**0.015**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA4	1	0.1674	26.66	9.9	0.021
	2	0.2024	80	9.9	0.029
	3	0.1510	60	9.9	0.012
		0.1736	**55.55**	**9.9**	**0,020**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA5	1	0.1558	53.33	9.7	0.009
	2	0.1534	93.33	9.7	0.010
	3	0.1713	93.33	9.8	0.015
		0.1601	**79.99**	**9.7**	**0.011**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA6	1	0.1918	100	9.7	0.019
	2	0.1881	86,66	9.8	0.023
	3	0.1828	100	9.8	0.025
		0.1875	**95.55**	**9.8**	**0.022**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA7	1	0.2125	93.33	9.9	0.031
	2	0.2301	100	9.9	0.030
	3	0.2323	93.33	10	0.046
		0.2258	**95.55**	**9.9**	**0.035**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA8	1	0.2080	93,33	9.9	0.034
	2	0.2157	73,33	9.9	0.037
	3	0.2139	100	9.8	0.041
		0.2125	**88.88**	**9.9**	**0.037**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA9	1	0.1463	80	9.4	0.007
	2	0.1791	93.33	9.3	0.024
	3	0.1802	80	9.3	0.024
		0.1685	**84.44**	**9.3**	**0.090**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
PSO	1	0.2069	86.66	5.5	0.027
	2	0.2035	93.33	5.4	0.029
	3	0.1767	93.33	3.3	0.024
		0.1957	**91.10**	**5.4**	**0.026**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA1.1	1	0.2438	80	19.1	0.023
	2	0.2409	90	19.1	0.021
	3	0.2448	83.33	19	0.017
		0.2425	**84.44**	**19**	**0.020**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA1_f1	1	0.2296	83.33	20.7	0.023
	2	0.2228	90	20.8	0.019
	3	0.2350	96.66	21.8	0.020
		0.2291	**89.99**	**21.1**	**0.021**

ID	Run	Best Fitness	Convergence Rate (%)	Run Time (in minutes)	Average Improvement Rate
GA1_loss	1	6,0448	80	10.7	0,021
	2	6,1533	93,33	10.5	0,022
	3	6,2154	100	9.7	0,022
		6,1378	**91,11**	**10.3**	**0.020**

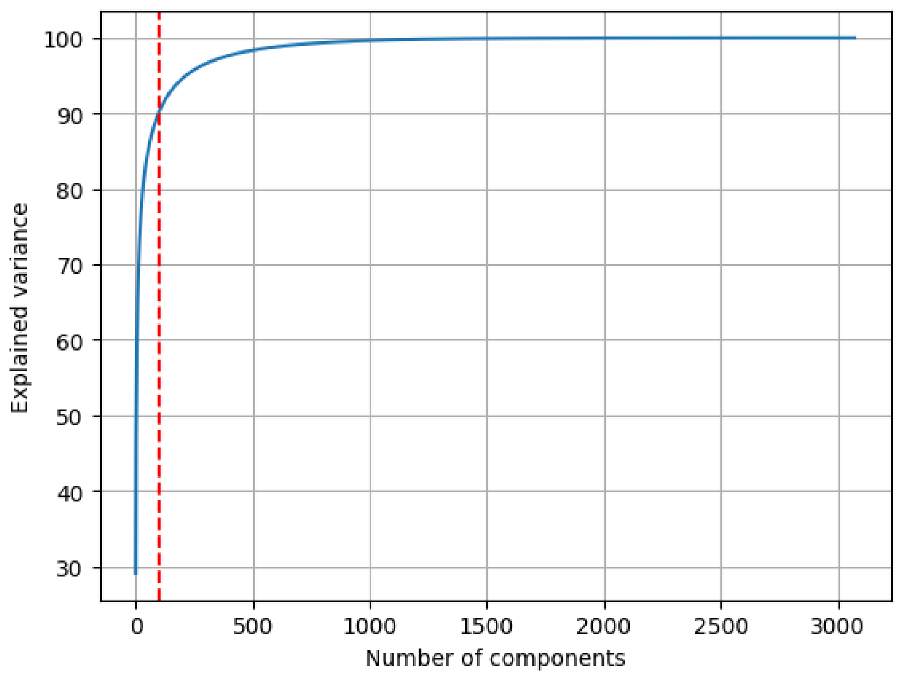

Figure 1 - PCA Components and Explained Variance

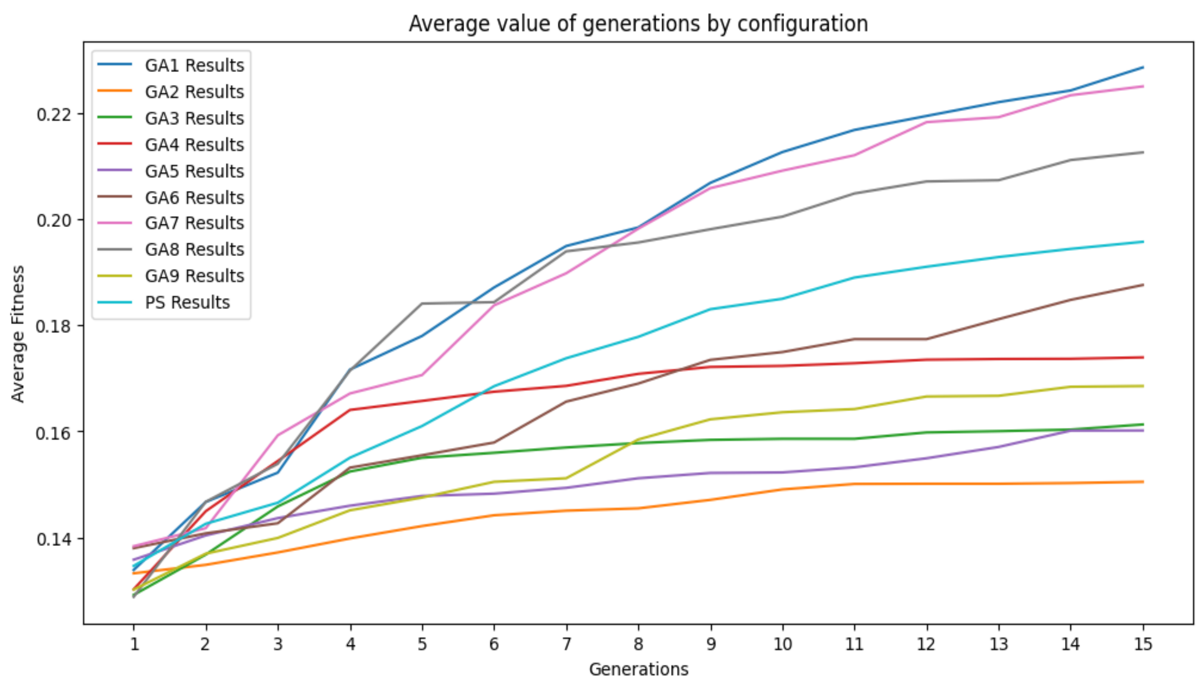

Figure 2 – Fitness Evolution Throughout Generations (by configuration)

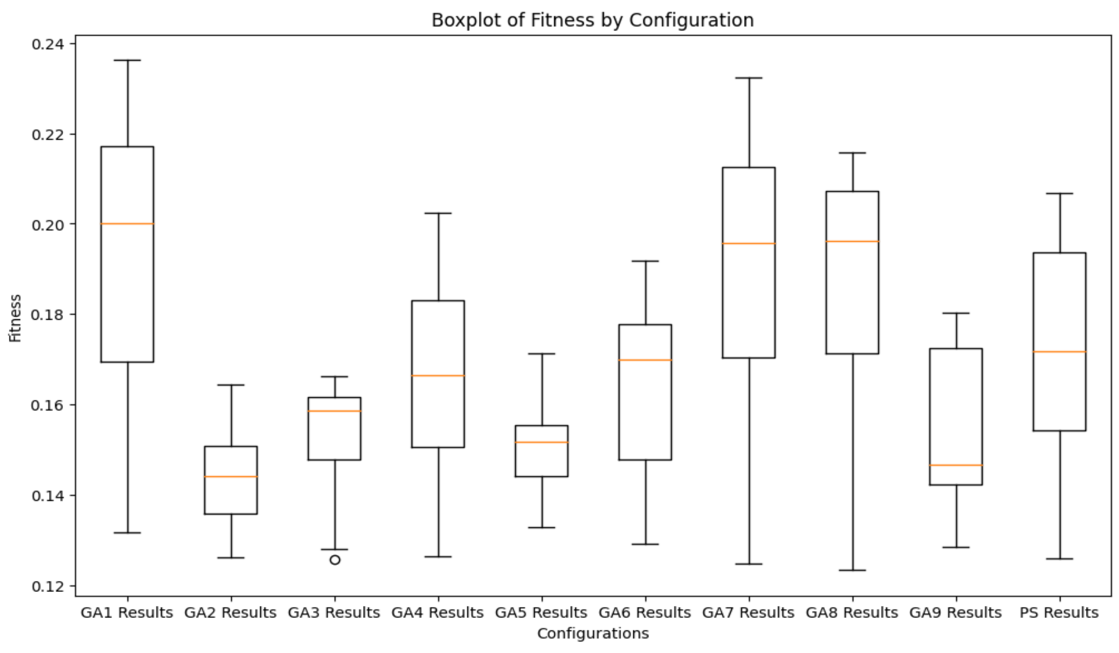

Figure 3 – Fitness Results BoxPlots (by configuration)

References

Vanneschi, L., & Silva, S. (2023). Lectures on Intelligent Systems. In Natural Computing Series. Springer International Publishing. https://doi.org/10.1007/978-3-031-17922-8

Pramoditha, R. (2021). Image Compression Using Principal Component Analysis (PCA). https://towardsdatascience.com/image-compression-using-principal-component-analysis-pca-253f26740a9f