PROJECT REPORT

Computational Intelligence for Optimization

IMAGE CLASSIFICATION

GROUP 11

Felix Gaber 20221385
Marta Dinis 20220611
Patrícia Morais 20220638
Samuel Santos 20220609

1. Problem definition

The objective of this project is to optimize the weights of a neural network using genetic algorithms. For this purpose, we employed the well-known CIFAR-10 dataset that contains images in 10 mutually exclusive classes.

Search space: considering a neural network with a total of n weights+biases, the search space is all the combinations of n weights, with each value being a real number.

Fitness function: accuracy of the neural network

Objective: maximization of fitness

2. Implementation

2.1 Description of the neural network

Considering that we're optimizing the weights of a neural network, it is relevant to describe its structure.

We defined a simple neural network with 5 layers: input, convolutional, pooling, flatten and dense. However, we only have weights to optimize in two layers: the convolutional and the dense layer (output).

The parameters for each layer were kept constant throughout the whole process - the only change is in the weights.

In our case, the metric considered is accuracy. This is, by its nature, a maximization problem and, even though the functions implemented already allow for both maximization and minimization, in practice it only makes sense to maximize this metric.

To make the implementations more abstract, we constructed our code so that you could apply it to any other neural network with the same structure - with 2 trainable layers in the same position in the network as ours.

In order to apply the project to a different dataset, we would just have to change a couple of parameters that have, by definition, to be hard coded, as the rest of the code is based on those values.

2.2 Representation

Our individual is represented as a list of lists. Inside the representation, each list contains the weights + biases for the corresponding layer in the neural network. Because each layer in the network has a different shape, this was also considered when developing the representation of the individuals, that is, the lists inside the representation can have different lengths.

Individuals are initialized by assigning random continuous values in the interval [0,1] to each of the weights. These weights are then allowed to change to any real value. In the convolutional layer we have 864 weights + 32 biases and in the dense layer we have 16576 weights + 10 output labels. We chose this representation, over the alternative of having a single list with all weights and biases. This will allow us to easily apply genetic operators within the same layer, ensuring that we're not moving values from one layer to another. Literature points out that this will work best (2019, Esfahanian[i]).

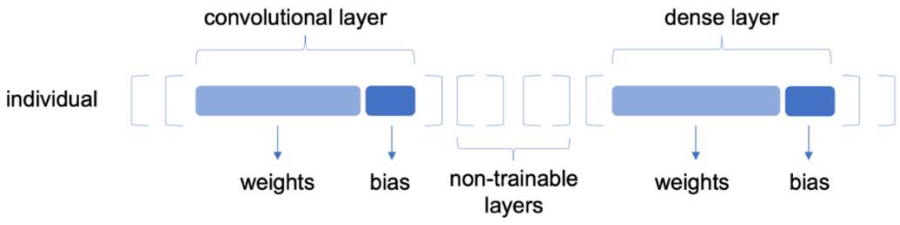

Image 1 - Representation of an individual

2.3 Fitness Function

One of the most used performance metrics for neural networks is the *accuracy* calculated on the dataset. For this reason, we decided to use this metric as the fitness measure for our genetic algorithm.

To calculate fitness, we compile the network with the weights that represent the individual and then calculate the results on the dataset.

2.4 Genetic Operators

At least three different methods were implemented for each type of genetic operator: selection, mutation, and crossover. The objective is to test various combinations of these (and other) parameters to find the 'best' one.

The selection phase does not depend on the representation of the individuals, but only on their fitness. Therefore, there were no restrictions on which selection methods to test on this genetic algorithm. The chosen methods were ***tournament*** *selection*, ***fitness proportionate*** *selection*, ***nested tournament*** *selection* and ***rank based*** *selection.*

Unlike the selection methods, mutation and crossover must take into account the fact that the individuals are represented by lists of continuous values. Another important factor to consider when implementing these methods is our very specific representation of the individuals. Although we wanted to apply crossover and mutation with a certain probability, it made more sense to apply these operators by layer of the neural network. In other words, when we are applying crossover, only the first list (layer) of the two parents will be used to generate the first list (layer) of the offspring; similarly, when we are mutating an individual, we also work layer by layer.

In order to account for these restrictions, we looked for mutation and crossover methods that worked for continuous values and adapted them so that they would be applied for each layer individually.

We implemented ***geometric*** *crossover* and ***arithmetic*** *crossover*, according to what we learnt in classes, and implemented an extra method, ***uniform*** *crossover*. Uniform crossover creates each offspring by getting, with a probability of 0.5, the value from the corresponding parent; else, it is obtained from the other parent.

The mutation methods developed were: ***geometric*** *mutation,* ***inversion*** *mutation,* ***gaussian*** *mutation,* as well as a **custom version of the *swap*** *mutation*, where it swaps various pairs of values, instead of just one, based on a certain percentage of the length of the layer.

In the case of gaussian mutation, the standard deviation of the values is calculated for each layer; after that, a random value from a normal distribution with 0 mean and the previously calculated standard deviation is added to each element (weight) of the layer. This process is repeated for each layer.

2.5 Elitism

It is logical that applying elitism will most probably improve the performance of our model. Also, the inclusion of elitism guarantees that, from generation to generation, the best fitness will never decrease. In light of this, we decided to implement not only the option of applying (or not) elitism, but also the size of the elite. Therefore, we incorporated an *elitism* parameter that will be 0, if we don't want to apply it, or any positive integer (smaller than the population size) that represents the elitism size if we do.

After experimenting, we found that it made sense to use an elitism of 1. This is consistent with our expectation, as we leave space for the creation of a wide offspring, but still ensure that we never lose the best individual. We observed no advantage with higher values, such as 5, and with lower values we kept finishing with a sub-optimal individual.

2.6 Further Implementations - grafting

We observed that our model is able to evolve for a few dozen generations, and then it doesn't improve, or takes a lot of generations to get new (and small) improvements.

We don't want to waste time in generations that are not improving, but we do need the model to improve. This resonates with a portuguese expression, "q*uerer sol na eira e chuva no nabal*" (wanting the best of both worlds or, literally, wanting sun on the threshing floor and rain on the turnip plantation).

To address this concern, we looked for inspiration in nature, and found it in a technique (grafting) and in a concept (ripe), both familiar to farmers:

Grafting is a technique that joins two plants into one, combining characteristics from both plants.

This technique is known and used since ancient Rome. When in 1860 american vines were brought to Europe, along with them came phylloxera[ii], a bug that was tolerated by american vines but destroyed european vines and actually almost ended Europe's wine production. Grafting was then employed, with american vines used as rootstock and european vines as scions. The rootstock ensures that the plant is resistant

Image 2 - Grafting

to the phylloxera, while the scion produces grapes of its quality, not from the rootstock kind.

Ripe[iii]: *(of fruit or crops) completely developed and ready to be collected or eaten.*

When performing a population evolution through n generations, we keep track of the last generation in which there was some improvement. After a certain threshold during which there was no improvement - "patience" - we use **grafting**: switch some of the parameters and continue to run with updated values. We opted to lower cross-over rate, as it starts at a high level, and to increase mutation rate, as we saw in tests that high mutation rates give good results. This process can reoccur, depending on a flag that we have. After exhausting the allowed changes of parameters, if the model is still not improving we perform an early stop - we know the model is **ripe** and stop trying to improve it.

2.6 Further Implementations – Particle Swarm Optimization (PSO)

We decided to implement Particle Swarm Optimization (PSO) (Vanneschi, 2023)[iv] and compare it to our GA models. We considered each particle to be a point in an n-dimensional space, with each coordinate corresponding to a weight (or bias) in the neural network. The global optimum is an unknown theoretical point with accuracy 1, which may or may not exist.

With minimal experimentation, we were able to obtain an accuracy of 0.194, which is not the best result we saw, but still seems to show significant potential and was obtained in less time than corresponding results with GAs. Fine-tuning of swarm size and hyperparameters (inertia, cognitive and social pressure) will likely improve the results.

The next step would be to try grafting with our best model and PSO, but we considered that to be already outside a reasonable interpretation of the project scope.

3. Hyperparameter Tuning

Starting from values recommended in articles and documentation, and after validating through some experimentation, we fixed some of the parameters, and kept them constant in order to maximize comparability of results. Population size was set to 30 (as less than that would be suboptimal) and elitism to 1, number of generations to 100 (on average models are still slightly improving after 100 generations), and ms to 0.1. Other parameters were tested in two blocks, with several runs for each combination. Results were then averaged and compared.

We tested crossover and mutation rates; best results were **xo prob 0.9** and **mut prob 0.3**. We kept these values for the following runs.

xo_prob	mut_prob			
	0,05	0,2	0,3	0,4
0,7	0,1616	0,1721	0,1753	0,1844
0,8	0,1628	0,1676	0,1838	0,1754
0,9	0,1586	0,1704	**0,1858**	0,1782

Image 3 - Crossover and mutation probabilities optimization

We also compared the combinations of the 4 selection functions with the 3 crossover methods and with the 4 mutation methods, and obtained these results:

Crossover	Mutation	Selection			
		Fitness Prop.	Nested Tourn.	Rank-Based	Tournament
Arithmetic	Custom Swap	0,1720	0,1670	0,1263	0,1647
	Gaussian	0,1510	0,1633	0,1367	0,1617
	Geometric	0,1537	0,1640	0,1327	0,1790
	Inversion	0,1613	0,1673	0,1363	0,1643
Geometric	Custom Swap	0,1613	0,1690	0,1270	0,1703
	Gaussian	0,1517	0,1677	0,1380	0,1627
	Geometric	0,1647	0,1483	0,1380	0,1657
	Inversion	0,1510	0,1603	0,1467	0,1710
Uniform	Custom Swap	0,1570	**0,1837**	0,1323	0,1667
	Gaussian	0,1493	0,1570	0,1477	0,1630
	Geometric	0,1577	0,1740	0,1407	0,1757
	Inversion	0,1573	**0,1843**	0,1340	0,1600

Image 4 - Crossover, mutation and selection functions optimization

The best result was obtained when using nested tournament selection, Uniform cross-over and Inversion mutation.

The two best models were tested with different population sizes, and it became evident that larger population sizes give better results than smaller ones:

best 2 models	population size		
	30	60	100
Custom Swap Mutation	0,1700	0,1986	0,2162
Inversion Mutation	0,1726	0,1860	0,2076

Image 5 – Evaluating population size on our two best models

Even though in the first tests inversion mutation gave us the best model, and our custom swap provided the second best answer, when running this additional test with several population sizes, we see that this positions inverted. We are aware that, since we're running just a small number of experiments, the results obtained are not to be considered statistically significant.

When running the model for several generations, we see that it keeps improving. It is, however, a very slow improvement. It doesn't seem that generic algorithms, at least using the range of parameters and functions that we're experimenting with, will get results even near what can be obtained using backpropagation.

We executed our best model, both with "unlimited patience" and changing probabilities after 20 generations without improvement. We concluded that this grafting-inspired technique was unable to significantly improve our results.

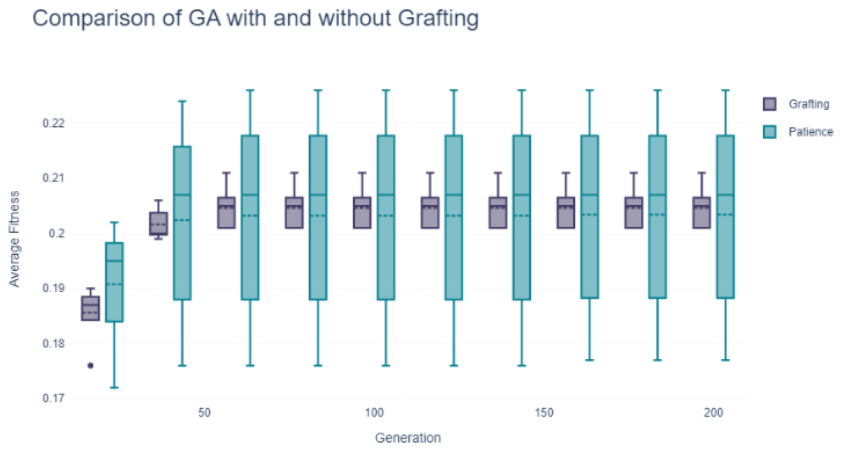

Image 6 - Comparison of GA with and without Grafting

The image compares the mean accuracy with Grafting and with (unlimited) Patience. It is evident that the means are extremely close to one another. It is also visible that the standard deviation is significantly smaller with Grafting than with Patience. While we observed this in our experiments, this again lacks strong statistical backing; we see it just as something that merits further investigation.

4. Results

Final parameters of our model:

- Population size: 100
- Number of generations: 200 (as our resources were limited, but there's evidence that GA will continue improving the results for a lot longer - 2019, Esfahanian[v]).
- Elitism: 1
- Selection method: nested tournament
- Crossover method: uniform crossover
- Mutation method: custom swap mutation
- Cross-over probability: 0.9

- Mutation probability: 0.3
- Mutation step (ms): 0.1
- Fitness function: accuracy

5. Conclusion

In terms of optimization, our model needs to optimize dozens of thousands of variables, making it complex and demanding significant computational resources. It is, therefore, understandable that our best accuracy results were below 0.25, compared to our benchmark neural network, where we achieved 0.49 by using back propagation. However, due to this difference, we can only conclude that we have poor results.

It was our understanding that the results were not a problem, as long as the algorithms were capable of improving the solutions. Therefore, we opted for incremental improvements, rather than more radical steps, such as changing the neural network structure.

It would be possible to run the model for more generations, since we know that it keeps improving, though slowly, and due to elitism we know we'll get the best result that was discovered by the model. We can therefore say that, with more computational resources, we would be able to obtain better results.

Another approach would be to implement more crossover, mutation and/or selection functions.

In order to improve results we decided to implement two new approaches; one of them was what we called "grafting" – the evolution of the parameters when the model shows no improvement after many generations; the other was Particle Swarm Optimization that, though not a genetic algorithm, is yet another evolutionary method that seeks solutions in the existing search space. The implementation of grafting leaves a whole new experimentation to conduct, as the parameters changed can be different, the magnitude of this change can be controlled, and the moment when we change them is also configurable. Regarding PSO, it would make sense to combine the strengths that it showed, namely its good performance and its lesser computational requirements. These hybrid models are a topic of interest and there are several articles published in this area.

As mentioned above, we discarded opting for more complex neural network structures. Otherwise, we could have opted for a deeper neural network or for the implementation of transfer learning techniques.

https://github.com/samueldatasci/black-knight

References

[i] Esfahanian, P., & Akhavan, M. (2019). Gacnn: Training deep convolutional neural networks with genetic algorithm. arXiv preprint arXiv:1909.13354.

[ii] https://daily.jstor.org/the-great-grape-graft-that-saved-the-wine-industry/

[iii] https://dictionary.cambridge.org/dictionary/english/ripe

[iv] L. Vanneschi and S. Silva, Lectures on Intelligent Systems, Natural Computing Series, https://doi.org/10.1007/978-3-031-17922-8_4

[v] Esfahanian, P., & Akhavan, M. (2019). Gacnn: Training deep convolutional neural networks with genetic algorithm. arXiv preprint arXiv:1909.13354.